Рубанцев Валерий

Развивающее программирование

Занимательные проекты на паскале

PascalABC.NET

Бесплатное издание детскиекниги.рф

Все права защищены. Никакая часть этой книги не может быть воспроизведена в любой форме без письменного разрешения правообладателей.

Автор книги не несёт ответственности за возможный вред от использования информации, составляющей содержание книги и приложений.

Copyright *Валерий Рубанцев*

Лилия Рубанцева

От автора

В книгах «**Решение задач на языке *паскаль*» *и* «Увлекательная математика с *паскалем*»** мы решали занимательные математические задачи всех времён и народов. И в этой книге вас ждёт немало интересных математических задач. Но не только! Мы также напишем **словесные, комбинаторные и графические программы.**

Математические проекты не только занимательные, но и полезные! Вы научитесь:

- вычислять наибольший общий делитель (НОД)
- вычислять наименьшее общее кратное (НОК)
- вычислять факториалы
- вычислять знаменитые числа π (отношение длины окружности к диаметру) и e (основание натуральных логарифмов)
- вычислять тригонометрические функции – синус, косинус и арктангенс
- все делители заданного числа
- находить простые числа по алгоритму Эратосфена
- находить числа Фибоначчи
- находить числа-палиндромы
- решать любые числовые ребусы (числобусы, криптарифмы)
- переводить арабские числа в римские
- составлять задачки про кроликов и фазанов

А также познакомитесь с рядами Тейлора, Лейбница, Валлиса.

Словесные проекты объединяют математику, программирование и русский язык! Вы научитесь находить:

- слова с двумя парами одинаковых букв
- слова-мамы
- слова-гуигнгнмы
- слова в словах

Комбинаторные проекты помогут вам:

- раскрашивать кубик в 3 цвета
- генерировать перестановки, размещения и сочетания

- подбрасывать монету и разменивать деньги
- подсчитывать прямоугольники
- вычислять вероятность выигрыша в *Спортлото*

Графические проекты самые наглядные!
Вы научитесь:

- чертить цветными линиями «*астроиды*», самые простые из которых украшают обложку книги.

Решая задачи, вы укрепите умения и навыки в применении таких элементов языка *PascalABC.NET,* как:

- Числовые типы данных – *integer, int64, double*
- Логический тип *boolean*
- Арифметические операции
- Простейшие математические функции
- Переменные и константы
- Операторы объявления и присваивания
- Комбинированные операторы присваивания
- Логические операторы и выражения
- Условные логические операторы *if, if – else*
- Циклы *for, while, repeat - until*
- *Массивы, списки, множества*
- Функции
- Операторы *exit, contine, break*
- Операции ввода и вывода *Read(ln), Write(ln).*

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

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

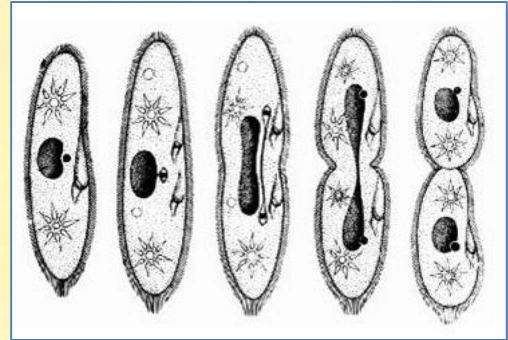
От автора ... 3
Оглавление ... 6

Делится - не делится? ... 8
Наибольший общий делитель ... 15
Наименьшее общее кратное ... 21
НОК нескольких чисел ... 24
Назойливый остаток ... 26
Решето Эратосфена ... 29
Простые числа ... 37
Простые числа 2 ... 41
Взаимно простые числа ... 45
Факторизация ... 46
Совершенные числа ... 49
Квадраты ... 52
Кубы ... 55
Факториал ... 60
Числа Фибоначчи ... 64
Числа Фибоначчи 2 ... 68
Вычисляем пи и е ... 71
Ряд Лейбница ... 75
Вычисляем число е ... 77
Тригонометрические функции ... 83
Великолепная семёрка ... 90
Квантик ... 92
Квантик 2 ... 97
Числобус ... 102
Пора по парам ... 108
Упрощаем ... 114
Геенна ... 120
Боливия ... 123
Париж-анин ... 129
Слова-мамы ... 131

Гуигнгнм 137
Кубик 142
Разноцветный кубик 150
Цветные линии 153
Астроида 156
Астроида 2 159
Римские числа 167
Монета 172
Задача Клайва Синклера 175
Разменный пункт 185
Рекурсивные перестановки 190
«Правильные» перестановки 195
«Массивные» перестановки 199
«Библиотечные» перестановки 201
Сочетания 206
Спортлото 210
Palindrome 223
Арифметический генератор 226
Задания для самостоятельного решения 232

Литература 237

Делится - не делится?



*Бесконечный цикл **repeat-until***
***Методы** с параметрами*
*Оператор деления по модулю **mod***
*Оператор деления **div***
*Цикл **for***
*Условные операторы **if** и **if-else***
*Операторы **break** и **continue***
*Оператор цикла **foreach***
*Метод для извлечения квадратного корня **Math.Sqrt***
*Массив **integer[]***
*Методы печати в Консольном окне **Console.Write** и **Console.WriteLine***
*Метод ввода строки в консольном окне **Console.ReadLine***
*Метод **ReadInteger***

Начните новый проект и сохраните его на диске в папке **Делимость**.

С помощью операции **деления по модулю** можно легко проверить, делится ли одно число на другое нацело или нет. Например, мы хотим узнать, делится ли число 2016 на 7. Пишем:

```
uses
   System;

// Делимость чисел

begin
   Console.ForegroundColor := ConsoleColor.Green;
   if (2016 mod 7 = 0) then
      Console.WriteLine('Делится')
   else
      Console.WriteLine('Не делится');
   Console.WriteLine();
end.
```

Запускаем программу и тут же узнаём, что **делится**. То есть нам только и нужно, что проверить остаток от деления. Если он равняется нулю, то первое число делится на второе. Вот и вся премудрость!

В **главном блоке** пользователь самостоятельно выбирает число, для которого программа найдёт все делители:

```
begin
   {Console.ForegroundColor := ConsoleColor.Green;
   if (2015 mod 7 = 0) then
      Console.WriteLine('Делится')
   else
      Console.WriteLine('Не делится');
   Console.WriteLine();}

   // заголовок окна:
   Console.Title := 'Делится - не делится';
   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Делимость чисел');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   //бесконечный цикл ввода данных -
   //пока пользователь не закроет программу
   //или не введёт 0:
   repeat
      var num := ReadInteger('Введите натуральное число  >  ');
      if (num = 0) then exit;

      Solve1(num);
      //Solve2(num);
      //Solve3(num);
      //Solve4(num);
      Console.WriteLine();
   until not (true);
end.
```

После того как пользователь ввёл число, мы передаём его процедуре **Solve1**, которая в цикле ***for*** пытается поделить его на числа из диапазона *1.. num* и печатает в *Консольном окне* все делители заданного числа:

```
procedure Solve1(num: integer);
begin
   //печатаем результаты в Консольном окне:
   Console.WriteLine();
   Console.WriteLine('Число ' + num + ' делится на:');
   for var i := 1 to num do
      if (num mod i = 0) then
         Console.Write(i + ' ');
   Console.WriteLine();
   Console.WriteLine();
end;
```

Делится - не делится

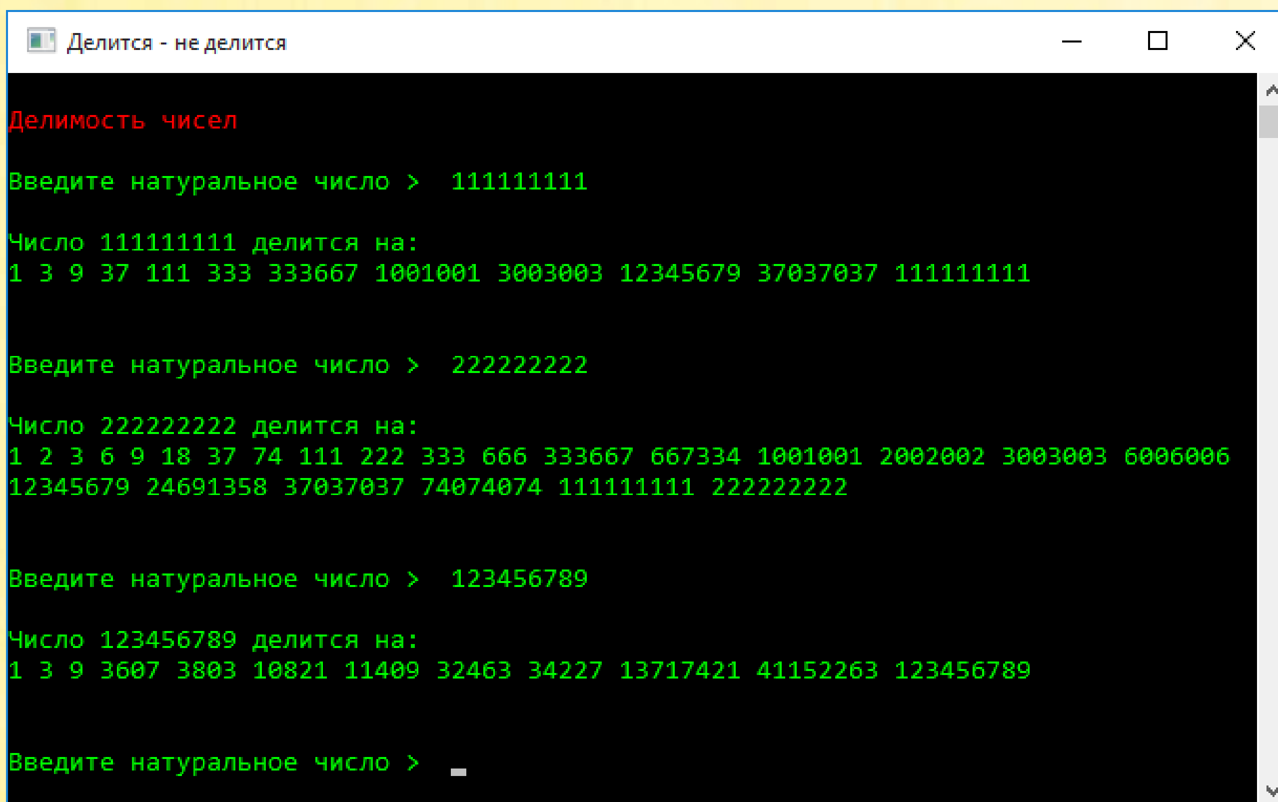
```
Делимость чисел

Введите натуральное число >  64

Число 64 делится на:
1 2 4 8 16 32 64


Введите натуральное число >  1000

Число 1000 делится на:
1 2 4 5 8 10 20 25 40 50 100 125 200 250 500 1000


Введите натуральное число >  2016

Число 2016 делится на:
1 2 3 4 6 7 8 9 12 14 16 18 21 24 28 32 36 42 48 56 63 72 84 96 112 126 144
168 224 252 288 336 504 672 1008 2016


Введите натуральное число >  2017

Число 2017 делится на:
1 2017


Введите натуральное число >
```

Задавая параметры цикла, мы исходим из того, что любое натуральное число делится на единицу и само на себя. На нуль делить нельзя, отрицательных натуральных чисел не бывает, а на числа, б*о*льшие заданного, делить нет смысла.

Таким образом, наша программа не только проверяет делимость заданного числа на все числа из диапазона *1..num*, но и находит *все его делители* в порядке возрастания.

Для проверки очень больших чисел можно **оптимизировать** нашу процедуру. Достаточно заметить, что заданное число *num* не может делиться на числа, б*о*льшие *num/2*, исключая, естественно, само число.

Немного подправим код – и пользователь может находить делители огромных чисел:




```
procedure Solve2(num: integer);
begin
   //печатаем результаты в Консольном окне:
   Console.WriteLine();
   Console.WriteLine('Число ' + num + ' делится на:');
```

```
   for var i := 1 to num div 2 do
      if (num mod i = 0) then
         Console.Write(i + ' ');
   Console.WriteLine(num);
   Console.WriteLine();
end;
```

Мы ещё во много раз ускорим поиск делителей, если заметим, что произведение симметричных относительно середины ряда делителей равно заданному числу. Например, для числа 64 мы получили такой ряд делителей:

1 2 4 8 16 32 64

Проверяем утверждение:

1 x 64 = 2 x 32 = 4 x 16 = 8 x 8 = 64

Всё верно! Единственное «неудобство» причиняют восьмёрки – они дважды входят в произведение, поэтому нам следует подумать, как оставить только **одну** из них.

Теперь легко заметить, что достаточно найти делители заданного числа в диапазоне $1..\sqrt{num}$. Вторую половину делителей мы найдём, поделив заданное число на очередной делитель.

Для хранения делителей мы заведём массив **res**:

```
procedure Solve3(num: integer);
begin
   //Массив для записи делителей:
   var res := new integer[1000];
   //текущий индекс в массиве:
   var n := 0;

   //печатаем результаты в консольном окне:
   Console.WriteLine();
   Console.WriteLine('Число ' + num + ' делится на:');
```

```
   for var i := 1 to Trunc(Math.Sqrt(num)) do
      if (num mod i = 0) then
      begin
         res[n] := i;
         n += 1;
         //не допускаем повторов делителей:
         if (i * i <> num) then
         begin
            res[n] := num div i;
            n += 1;
         end;
      end;
   foreach var i in res do
      if (i > 0) then
         Console.Write(i + ' ')
      else
         break;
   Console.WriteLine();
end;
```

Мы не знаем, сколько делителей окажется у заданного числа, поэтому число элементов в массиве **res** задаём заведомо больше, чем их может оказаться в действительности.

И вот почему. Для первого делителя – единицы – мы найдём парный делитель, равный заданному числу *num*, для второго парный делитель равен *(num / второй делитель).* То есть делители мы напечатаем не по порядку:

```
Делится - не делится

Делимость чисел

Введите натуральное число >  64

Число 64 делится на:
1 64 2 32 4 16 8

Введите натуральное число >  1111111

Число 1111111 делится на:
1 1111111 239 4649
```

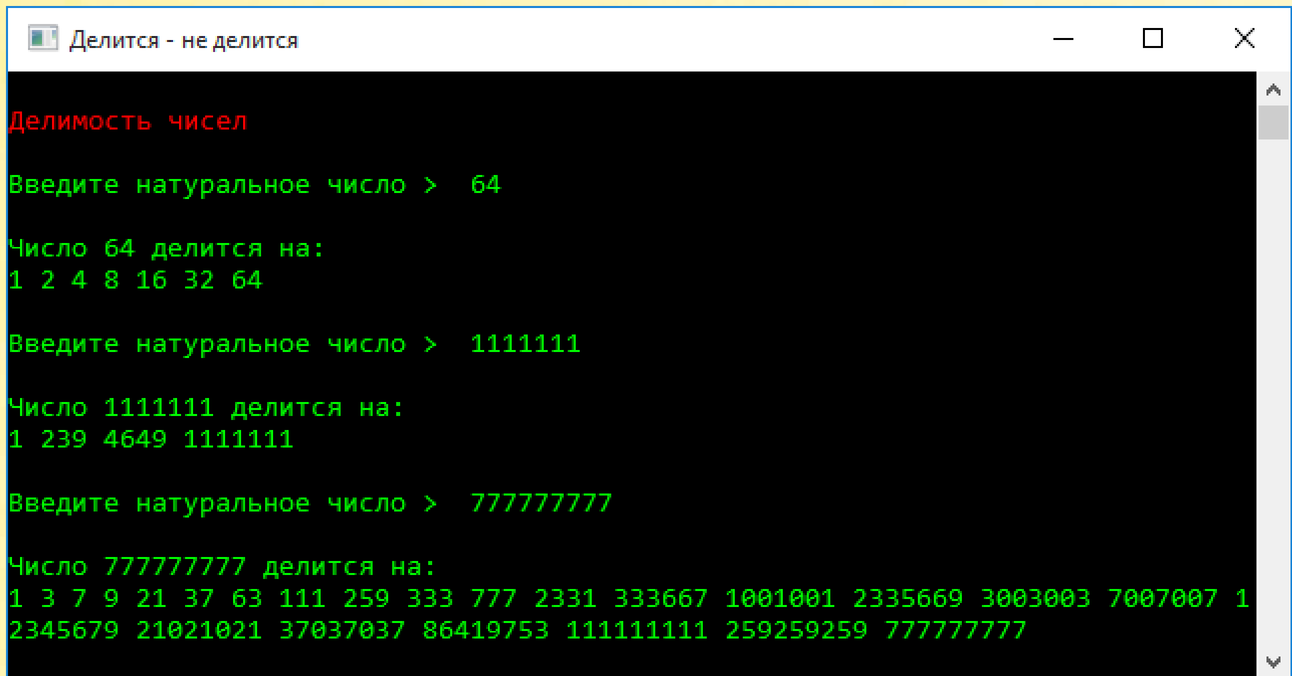
Чтобы выправить ситуацию, мы до печати результатов отсортируем массив *res* с помощью метода **Sort** класса *Array*, а затем напечатаем делители строго по ранжиру:

```
procedure Solve4(num: integer);
begin
   //Массив для записи делителей:
   var res := new integer[1000];
   //текущий индекс в массиве:
   var n := 0;

   //печатаем результаты в консольном окне:
   Console.WriteLine();
   Console.WriteLine('Число ' + num + ' делится на:');
   for var i := 1 to Trunc(Math.Sqrt(num)) do
      if (num mod i = 0) then
      begin
         res[n] := i;
         n += 1;
         //не допускаем повторов делителей:
         if (i * i <> num) then
         begin
            res[n] := num div i;
            n += 1;
         end;
      end;

   System.Array.Sort(res);
   foreach var i in res do
      if (i > 0) then
         Console.Write(i + ' ')
      else
         continue;

   Console.WriteLine();
end;
```

Правда, мы должны учесть, что теперь массив *res* будет начинаться с нулей, поскольку все делители - положительные числа, а в неиспользованных элементах массива записаны нули. С этим затруднением легко справиться, печатая только *положительные* числа в массиве.

> Конечно, для хранения делителей лучше использовать **список**.

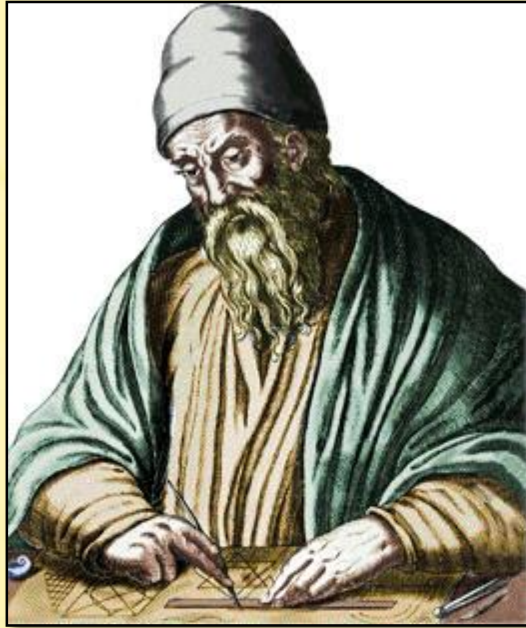
Теперь даже для огромных чисел мы мгновенно выписываем все делители:



Наибольший общий делитель

> ***Функция*** *с параметром*
> ***Функция*** *без параметров*
> *Цикл* ***for***
> ***Константы***
> *Оператор деления по модулю* ***mod***
> *Оператор деления* ***div***
> *Тип данных* ***int64***
> *Бесконечный цикл* ***repeat-until***
> *Метод* ***Convert.ToInt64***
> *Оператор* ***exit***
> *Условный оператор* ***if***
> *Условный оператор* ***if-else***

Для вычисления **НОД** мы воспользуемся *простым* и *быстрым* алгоритмами древнегреческого математика *Евклида*.



Евклид (Εὐκλείδης, ок. 325 года до н.э. - до 265 года до н.э.)

По-английски *НОД* называется **gcd** – *greatest common divisor*.

Простой алгоритм Евклида основан на следующих свойствах:

НОД(m,m) = m
НОД(m,n) = НОД(n,m)
НОД(m,n) = НОД(m-n, n) при m > n

Из них следует:

- Если m > n, то из m нужно вычесть n;
- Если m < n, то числа нужно поменять местами;
- Когда значения m и n сравняются, вычисление НОД заканчивается, и НОД = m = n.

Для вычисления *НОД* нам нужны два числа - **number1** и **number2**. Чтобы не обременять пользователя лишними заботами, мы позволим ему вводить числа в произвольном порядке, хотя для алгоритма важно, чтобы первое число было *больше* второго, поэтому выправляем ситуацию, если это необходимо.

> При *равенстве* чисел алгоритм сработает правильно.

Затем мы передаём оба числа в функцию *Euklid*, которая и реализует простой алгоритм Евклида.

```
//ПРОГРАММА ДЛЯ НАХОЖДЕНИЯ НАИБОЛЬШЕГО
//ОБЩЕГО ДЕЛИТЕЛЯ ДВУХ НАТУРАЛЬНЫХ ЧИСЕЛ (НОД)

begin
   //заголовок окна:
   Console.Title := 'НОД двух чисел';
   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('НОД двух чисел');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   //бесконечный цикл ввода данных -
   //пока пользователь не закроет программу
   //или не введёт отрицательные числа:
   repeat
      Console.Write('Введите первое число >  ');
      var number1 := Convert.ToInt64(Console.ReadLine());
      Console.Write('Введите второе число >  ');
      var number2 := Convert.ToInt64(Console.ReadLine());
      if (number1 + number2 < -1) then exit;

      //если первое число меньше второго,
      //то меняем их значения:
      if (number1 < number2) then
      begin
         var n := number1;
         number1 := number2;
         number2 := n;
      end;
      var nod: int64 := 0;
```

```
      // находим НОД:
      if (number2 = 0) then
         nod := number1
      else
         nod := Euklid(number1, number2);

      //печатаем НОД:
      Console.WriteLine('НОД = (' + number1.ToString() + ','
                          + number2.ToString() + ') = ' + nod);
      Console.WriteLine();
   until not (true);
end.
```

Если меньшее из чисел (или оба) равно нулю, то за *НОД* следует принять **первое** число. Если же оба числа положительные, то начинаем действовать по **простому алгоритму Евклида***:*

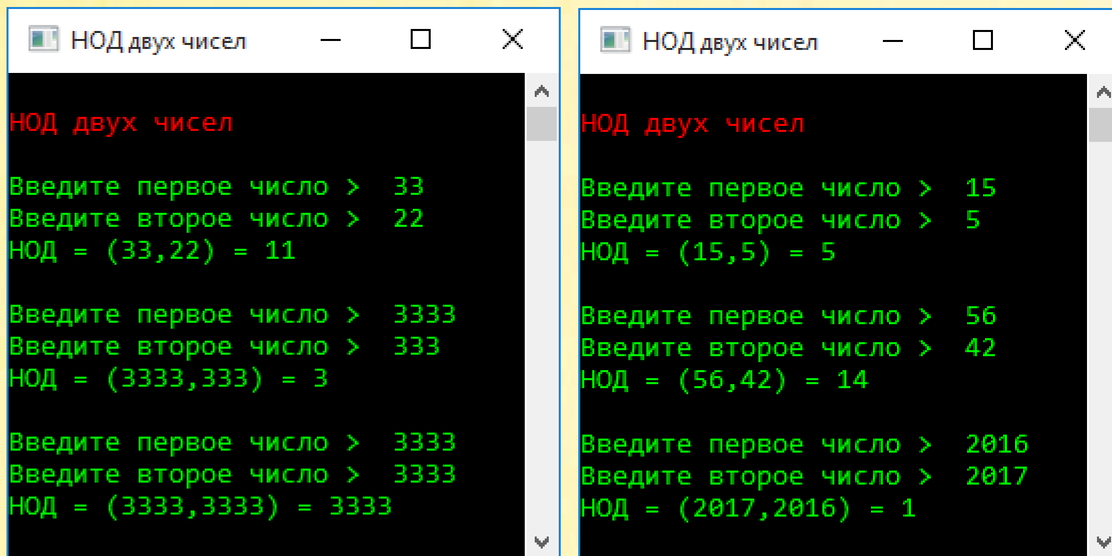
```
//ПРОСТОЙ АЛГОРИТМ ЕВКЛИДА
function Euklid(n1, n2: int64): int64;
begin
   while (n2 <> n1) do
   begin
      if (n1 >= n2) then
         n1 -= n2
      else n2 -= n1;
   end;
   Result := n1;
end;
```

В данном проекте мы используем числа типа **int64**, а не *int*, чтобы можно было вычислять *НОД* и очень больших чисел. Но тут, конечно, следует учитывать значения чисел: если они очень большие, то лучше пользоваться *быстрым*, а не простым алоритмом.

В цикле ***while*** мы на каждой итерации вычитаем из большего числа меньшее – до тех пор, пока значения переменных *n1* и *n2* не сравняются. Поскольку с каждым

разом одно из чисел *уменьшается*, то рано или поздно это условие будет выполнено.

Вычисленное значение *НОД* мы возвращаем в главный блок, который и публикует его в окне консоли. При небольших числах алгоритм работает очень быстро:



Но для очень больших чисел следует использовать **быстрый алгоритм Евклида**:

```
//находим НОД:
if (number2 = 0) then
   nod := number1
else
   //nod := Euklid(number1, number2);
   nod := SpeedEuklid(number1, number2);


//БЫСТРЫЙ АЛГОРИТМ ЕВКЛИДА
function SpeedEuklid(n1, n2: int64): int64;
begin
   while (n2 > 0) do
   begin
      var n := n1 mod n2;
      n1 := n2;
      n2 := n;
   end;
   Result := n1;
```

```
end;
```

Он отличается от простого алгоритма тем, что на каждой итерации мы находим *остаток от деления* первого числа на второе, значение второго числа присваиваем первому числу, а значение остатка - второму. Так мы продолжаем до тех пор, пока остаток от деления не обратится в нуль. Как и в первом случае, это условие обязательно будет выполнено.

Быстрый алгоритм Евклида основан на свойстве:

НОД(m,n) = НОД(n, m mod n) при m > n

Этот алгоритм не собьёшь с толку никакими числами! Вы можете взять любые числа и убедиться, что алгоритм действительно работает практически мгновенно:

```
НОД двух чисел

НОД двух чисел

Введите первое число >  1111111111111
Введите второе число >  11111111111
НОД = (1111111111111,11111111111) = 1

Введите первое число >  1111111111111
Введите второе число >  11111
НОД = (1111111111111,11111) = 1

Введите первое число >  111111111
Введите второе число >  27
НОД = (111111111,27) = 9
```

```
Выбрать НОД двух чисел

Введите первое число >  2222222333333
Введите второе число >  33
НОД = (2222222333333,33) = 1

Введите первое число >  222222333333
Введите второе число >  33
НОД = (222222333333,33) = 33

Введите первое число >  987654321
Введите второе число >  123456789
НОД = (987654321,123456789) = 9

Введите первое число >  987656789
Введите второе число >  123454321
НОД = (987656789,123454321) = 1
```

Для большего понимания работы быстрого алгоритма давайте вручную найдём *НОД* чисел 42 и 14:

```
number1 = 42

number2 = 14
```

Так как второе число больше нуля, то находим остаток от их деления:

n = 42 mod 14 = 0

И присваиваем новые значения переменным:

number1 = 14

number2 = 0

Второе число теперь равно нулю, значит, *НОД* найден – он равен второму числу, то есть 14.

Рассмотрим **другой пример**:

number1 = 56

number2 = 42

n = 56 mod 42 = 14

number1 = 42

number2 = 14

Уже после одного цикла мы пришли к первому примеру.

Наименьшее общее кратное

Бесконечный цикл ***repeat-until***
Тип данных ***int64***
Метод ***Convert.ToInt64***
Оператор ***exit***
Условный оператор ***if***
Метод *с параметрами*

Зная наибольший общий делитель двух чисел, очень просто вычислиnm их **наименьшее общее кратное** по такой формуле:

НОК = число1 * число2 / НОД (число1, число2) (1)

По-английски *НОК* называется **lcd** – *least common denominator*.

Для нахождения *НОД* мы воспользуемся быстрым алгоритмом Евклида, а **НОК** вычислим по формуле (1):

```
//ПРОГРАММА ДЛЯ ВЫЧИСЛЕНИЯ НАИМЕНЬШЕГО ОБЩЕГО
//КРАТНОГО ДВУХ ЧИСЕЛ (НОК)

begin
   //заголовок окна:
   Console.Title := 'НОК двух чисел';
   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('НОК двух чисел');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   //бесконечный цикл ввода данных -
   //пока пользователь не закроет программу
   //или не введёт отрицательные числа:
   repeat
      Console.Write('Введите первое число >  ');
      var number1 := Convert.ToInt64(Console.ReadLine());
      Console.Write('Введите второе число >  ');
      var number2 := Convert.ToInt64(Console.ReadLine());
      if (number1 + number2 < -1) then exit;

      //если первое число меньше второго,
      //то меняем их значения:
      if (number1 < number2) then
      begin
         var n := number1;
         number1 := number2;
         number2 := n;
```

```
      end;
      var nok: int64 := 0;

      //находим НОК:
      if (number1 * number2 = 0) then
         nok := 0
      else
         nok := number1 * number2 div SpeedEuklid(number1, number2);

      //печатаем НОК:
      Console.WriteLine('НОК = (' + number1.ToString() + ','
                          + number2.ToString() + ') = ' +
                            nok.ToString());
      Console.WriteLine();
   until not (true);
end.
```

Поскольку *НОД* вычисляется очень быстро, то и *НОК* мы получаем в считанные мгновения:

```
НОК двух чисел

НОК двух чисел

Введите первое число >  111
Введите второе число >  27
НОК = (111,27) = 999

Введите первое число >  1111
Введите второе число >  11
НОК = (1111,11) = 1111

Введите первое число >  2016
Введите второе число >  16
НОК = (2016,16) = 2016

Введите первое число >  2017
Введите второе число >  17
НОК = (2017,17) = 34289
```

```
НОК двух чисел

НОК двух чисел

Введите первое число >  20
Введите второе число >  16
НОК = (20,16) = 80

Введите первое число >  1111111
Введите второе число >  111111
НОК = (1111111,111111) = 123456654321

Введите первое число >  111111
Введите второе число >  111111
НОК = (111111,111111) = 111111
```

НОК нескольких чисел

> ***Функция** с параметром*
> *Тип данных **int64***
> *Цикл **for***
> *Оператор **exit***
> *Цикл **while***
> *Оператор деления по модулю **mod***

При решении некоторых математических задач нужно знать **НОК** нескольких последовательных чисел.

Найти **НОК нескольких чисел** можно так.

Сначала вычисляем *НОК2* первых двух чисел, затем вычисляем *НОК3* для *НОК2* и третьего числа и так далее.

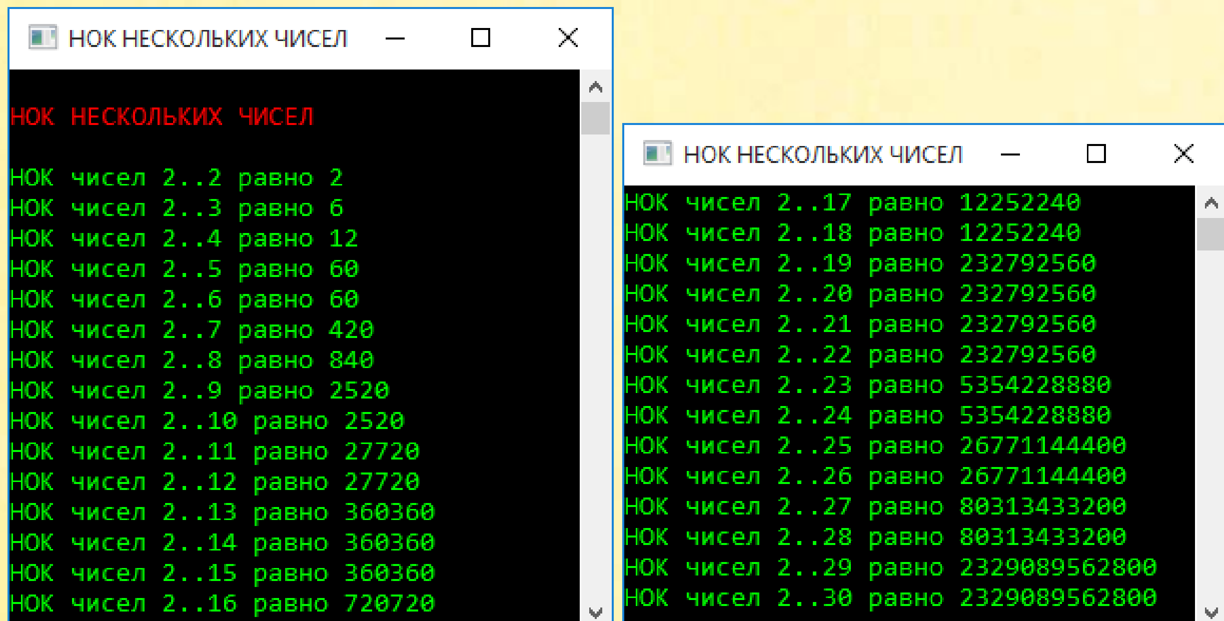
Но давайте решим эту задачу для ряда чисел произвольной длины – от 2 до **maxnok**. В **главном блоке** мы вызываем фуннцию *NOK2*, которой и передаём верхнюю границу ряда:

```
uses
   System;

//ВЫЧИСЛЕНИЕ НОК НЕСКОЛЬКИХ ЧИСЕЛ

begin
   //заголовок окна:
   Console.Title := 'НОК НЕСКОЛЬКИХ ЧИСЕЛ';
   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('НОК НЕСКОЛЬКИХ ЧИСЕЛ');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   NOK2(30);

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

В функции **NOK2** мы задаём начальное значение переменной **nok**, равное 1, что вполне разумно для наименьшего общего кратного чисел 1 и 1. Далее мы находим *НОК* для чисел 1 и 2, затем для полученного *НОК* и тройки – как и было описано выше:

```
//ВЫЧИСЛЯЕМ НОК
function NOK(number1, number2: int64): int64;
begin
   if (number1 * number2 = 0) then
      Result := 0
   else
      Result := number1 * number2 div SpeedEuklid(number1, number2);
end;

//НАХОДИМ НОК РЯДА ЧИСЕЛ
procedure NOK2(maxnok: integer);
begin
   var nokl: int64 := 1;
   for var i := 2 to maxnok do
   begin
      nokl := NOK(nokl, i);
      Console.WriteLine('НОК чисел 2..{0} равно {1}', i, nokl);
   end
end;
```

Все промежуточные значения *НОК* **печатаем** на экране:



Назойливый остаток

В книге *Увлекательная математика с паскалем* мы решали задачу о назойливом остатке и вычисляли НОК последовательных чисел от 2 до 6 (см. [KA86], решение на странице 133).

Из полученного в предыдущем проекте списка видно, что *НОК* ряда чисел 2..6 равно 60. При делении чисел вида *60n* на числа 2..6 в остатке мы всегда будем получать 0. А при делении чисел вида *(60n + 1)* – единицу, что и требуется в условии задачи. Но это число *(60n + 1)* должно делиться на 7 без остатка, то есть:

$$(60n + 1) = 7k \qquad (1)$$

Поскольку мы умеем вычислять *НОК* любого ряда чисел, то можем обобщить задачу о назойливых остатках для любых рядов, записав формулу так:

$$(nok * n + 1) = maxd * k \qquad (2)$$

Решаем задачу с помощью новой функции *Solve*:

```
begin
   //заголовок окна:
   Console.Title := 'Назойливый остаток';
   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Назойливый остаток');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   Solve(17);

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

Этой функции мы должны передать то число **maxd**, на которое делится без остатка искомое число. В задаче это семёрка, но нас, конечно, интересуют более крупные числа!

На рисунке выше хорошо видно, что наименьшее искомое число быстро растёт с увеличением *maxd*, поэтому нужно либо задавать значение переменной *max* достаточно большим, либо искать только заданное количество искомых чисел.

В функции *Solve* мы находим *НОК* ряда чисел *2..(maxd-1),* а для этого нам нужна модифицированная функция для вычисления *НОК*, которая **возвращает**, а не печатает значения:

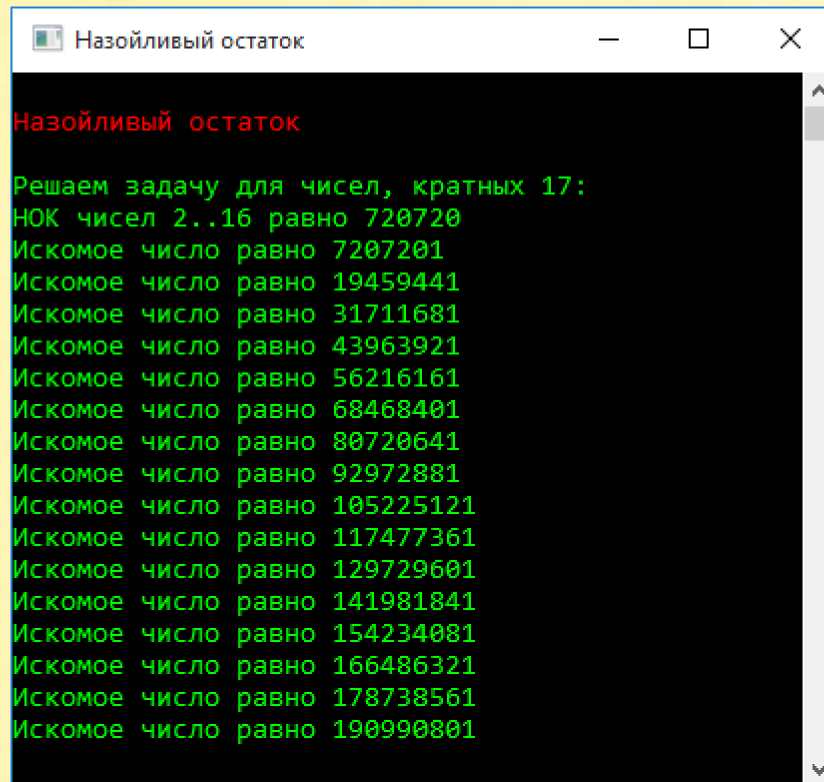
```
function NOK3(maxnok: integer): int64;
begin
   var nokl: int64 := 1;
   for var i := 2 to maxnok do
   begin
      nokl := NOK(nokl, i);
   end;
   Result := nokl;
end;
```

По формуле (2) мы ищем в цикле ***while*** числа, которые отвечают условиям задачи и печатаем их в Консольном окне:

```
procedure Solve(maxd: integer);
begin
   //макс. число:
   var max := 200000000;

   Console.WriteLine('Решаем задачу для чисел, кратных {0}: ', maxd);
   var nok := NOK3(maxd - 1);
   Console.WriteLine('НОК чисел 2..{0} равно {1}', (maxd - 1), nok);

   //искомые числа имеют вид:
   //(nok * n + 1)
   var n := 1;
   var num := nok * n + 1;

   while(num <= max) do
```

```
   begin
     //число не кратно maxd:
      if (num mod maxd = 0) then
         Console.WriteLine('Искомое число равно ' + num);
      n += 1;
      num := nok * n + 1;
   end
end;
```

Например, взяв вместо числа 7 на десяток больше, мы получим такие решения задачи:

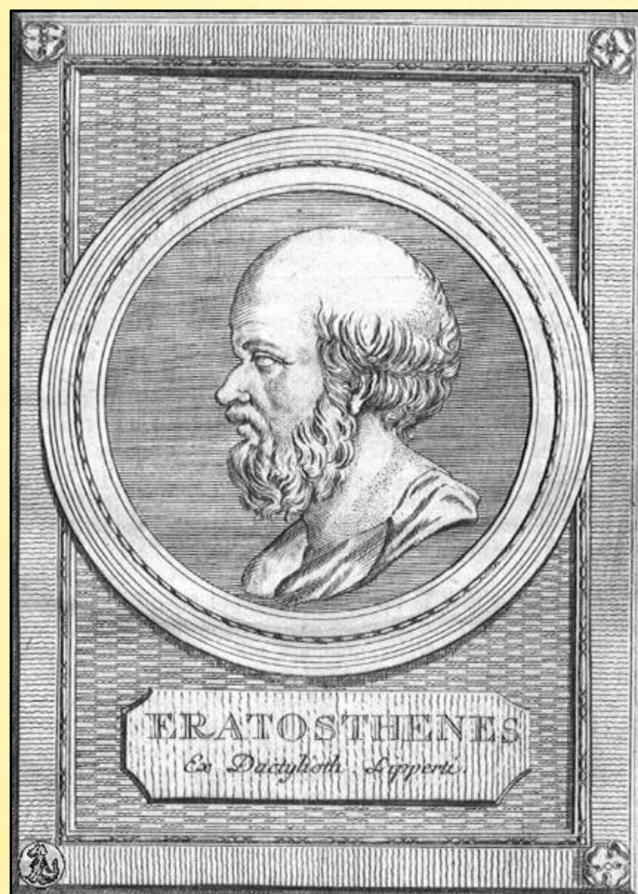


Конечно, без компьютера решать такие задачи было бы крайне затруднительно!

А теперь поэкспериментируйте и подумайте, почему для составных чисел **maxd** задача не решается.

Решето Эратосфена

Простыми называются натуральные числа, имеющие в точности *два разных* делителя. Из этого определения следует, что ни нуль, ни единица к простым числам не относятся. Также легко установить, что первое простое число - это *двойка*, потому что она делится на единицу и на саму себя (двойка – единственное *чётное* простое число!). Дальше вы легко найдёте тройку, пятёрку и семёрку. Вам не составит труда продолжить этот ряд: 11, 13, 17, 23, 29, 31. Чтобы отбросить многие *составные* (то есть не простые) числа, достаточно воспользоваться *признаками делимости*. Но проверять большие числа таким способом «опасно», потому что легко ошибиться при делении (да и вообще делить большие числа затруднительно). На этот случай греческий математик *Эратосфен* придумал надёжный способ поиска простых чисел, который в его честь назвали **решетом Эратосфена**.



Ἐρατοσθένης ὁ Κυρηναῖος (276 - 194 до н.э.)

Действует он так **[ЗП88**, Задача 557**]**.

Поиск простых чисел начинается с засыпки натуральных чисел в «решето», которое удобно представить в виде прямоугольной таблицы. *Наименьшее* число - это двойка, поскольку единица к простым числам не относится. А *наибольшее* – любое, по вашему выбору. Мы ограничимся первой сотней чисел:

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Находим в таблице **двойку** – первое простое число – и обводим её кружком. Очевидно, что все последующие числа, кратные двойке, простыми быть не могут, поэтому мы их вычёркиваем. В данном примере мы будем просто *закрашивать* клетки с составными числами:

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Как видите, уже после первого, грубого просеивания в решете осталась только половина чисел. Продвигаемся дальше по таблице и находим первое после двойки невычеркнутое число. Это **тройка** – второе простое число. Вычёркиваем все ещё не вычеркнутые числа, кратные тройке:

	②	③	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Ну а затем всё повторяется. Находим следующее простое число – **пятёрку** - и вычёркиваем числа, кратные пяти. Переходим к **семёрке**, затем к числам **11** и **13**. И так далее, пока не дойдём до конца таблицы. Числа в кружках - *простые*, все прочие – составные:

	②	③	4	⑤	6	⑦	8	9	10
⑪	12	⑬	14	15	16	⑰	18	⑲	20
21	22	㉓	24	25	26	27	28	㉙	30
㉛	32	33	34	35	36	㊲	38	39	40
㊶	42	㊸	44	45	46	㊼	48	49	50
51	52	(53)	54	55	56	57	58	(59)	60
(61)	62	63	64	65	66	(67)	68	69	70
(71)	72	(73)	74	75	76	77	78	(79)	80
81	82	(83)	84	85	86	87	88	(89)	90
91	92	93	94	95	96	(97)	98	99	100

Пользуясь этим алгоритмом, мы легко смоделируем решето Эратосфена в программе на *паскале*:

*Бесконечный цикл **repeat-until***
*Функция **ReadInteger***
*Условный оператор **if***
*Оператор **exit***
***Процедура** с параметром*
*Массив **integer[]***
*Цикл **for***
*Цикл **repeat-until***
*Оператор цикла **foreach***

Нам достаточно знать только конец диапазона чисел **end**, ведь поиск всегда начинается с *двойки*:

```
uses
   System;

//Решето Эратосфена

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

   //бесконечный цикл ввода данных -
   //пока пользователь не закроет программу
   //или не введёт 0:
   repeat
      var endnum := ReadInteger('Введите конец  диапазона  >  ');
      //если конец диапазона равен 0,
      //то программу закрываем:
      if (endnum = 0) then exit;

      //конец диапазона не меньше двойки:
```

```
      if (endnum < 2) then endnum := 2;

      //ищем простые числа:
      Primes(endnum);
   until not (true);
end.
```

Здесь мы дополнительно позаботились о том, чтобы пользователь мог закрыть программу, введя в качестве конца диапазона **нуль**. Это необязательно, но и не помешает.

Пользователь может и не знать об этой особенности вашего приложения, поэтому сообщите ему о ней.

Непосредственно поиск простых чисел мы поместим в процедуру **Primes**, которой передаём число – конец диапазона *end*:

```
//ИЩЕМ ПРОСТЫЕ ЧИСЛА
procedure Primes(endnum: integer);
begin
   Console.WriteLine();
   Console.WriteLine('Простые числа в заданном диапазоне:');
```

Решето вполне естественно представить массивом **number**:

```
   //создаём массив натуральных чисел 2..end:
   var number := new integer[endnum + 1];
```

А дальше мы действуем по **алгоритму Эратосфена**:

1. Записываем в массив *number* числа, начиная с двойки и заканчивая концом диапазона:

```
for var i := 2 to endnum do
    number[i] := i;
```

2. Переменную **prime** отведём под *текущее* простое число. Сначала это будет *двойка*:

```
var prime := 2;
```

3. «Вычёркиваем» из массива число *prime*prime*, а затем все числа, начиная с этого числа через *prime*:

4 – 6 – 8 - … *для prime= 2,*
9 – 12 – 15 - … *для prime= 3.*

И так далее:

```
repeat
    var i := prime * prime;
    while (i <= endnum) do
    begin
       number[i] := 0;
       i += prime;
    end;
```

Конечно, мы не можем реально зачеркнуть число в массиве, поэтому присваиваем соответствующему элементу массива значение **нуль**, которое будет означать, что это число *составное*.

4. Ищем первое «невычеркнутое» число – его значение в массиве должно отличаться от нулевого:

```
    //ищем следующее простое число:
```

```
      prime += 1;
      while ((prime <= Math.Sqrt(endnum)) and (number[prime] = 0)) do
         prime += 1;
```

Теперь переменная *prime* содержит *следующее* простое число.

5. Если *prime* не превосходит корня квадратного из максимального числа *end*, то переходим к пункту 3:

```
until not (prime <= Math.Sqrt(endnum));
```

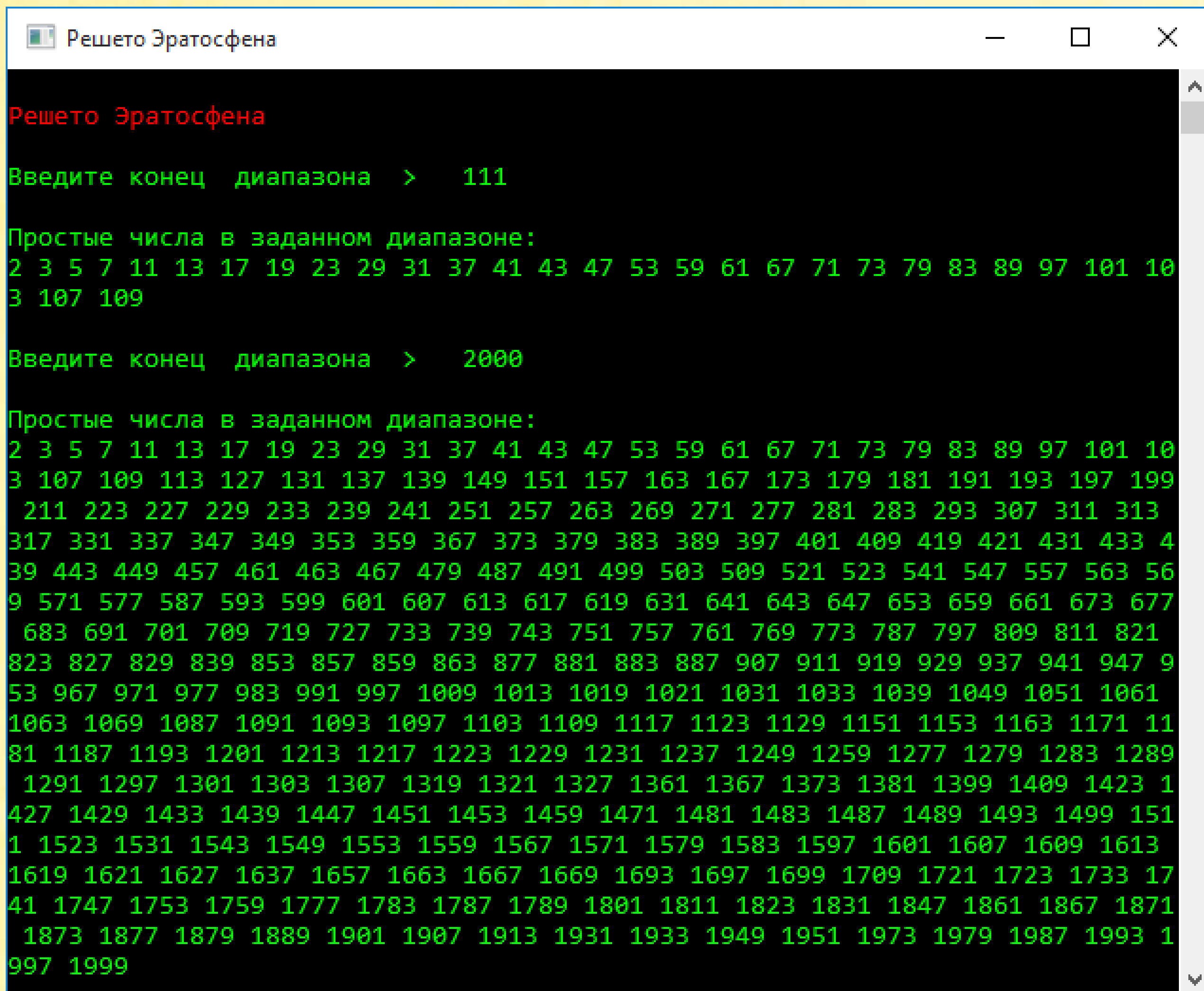
В противном случае все простые числа в заданном диапазоне уже найдены, их значения в массиве - *ненулевые*. Перебираем весь массив, находим по этому признаку простые числа и печатаем их на экране:

```
   //печатаем простые числа:
   foreach var i in number do
   begin
      if (i <> 0) then
         Console.Write(i + ' ');
   end;
   Console.WriteLine();
   Console.WriteLine();
end;
```

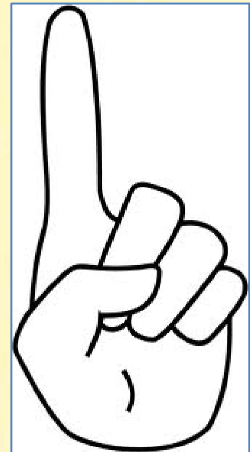
Поскольку в главном блоке действует бесконечный цикл *repeat-until*, то пользователь может и дальше «решетить» простые чисел. Несмотря на «древность» алгоритма, он действует довольно быстро, но очень жаден до памяти компьютера.

Нашу процедуру чисел можно улучшить, если вместо массива целых чисел типа *integer* использовать массив типа *boolean*. Так мы вместо 4 байтов на число затратим только один.

Этот пример наглядно показывает, что для написания эффективной программы нужен хороший алгоритм. А если алгоритм имеется, то перевести его на любой язык программирования совсем несложно.



Простые числа



*Бесконечный цикл **repeat-until***
*Функция **ReadInteger***
*Условный оператор **if***
*Оператор **exit***
***Процедура** с параметрами*
Запись в файл
*Оператор **using***
*Вложенные циклы **for***
*Оператор целочисленного деления **mod***

Всем хорош алгоритм Эратосфена, но не годится для **больших** чисел - слишком неэкономно он расходует память компьютера. Мы, конечно, и с помощью решета легко узнаем, что 2017-й год - простой. Но вот с числами 514229 или 39916801 нам уже придётся повозиться! В таких случаях правильнее один раз написать программу, чем каждый раз считать вручную.

Чтобы сделать программу более *универсальной*, мы добавим ещё одну переменную - **begnum**, чтобы пользователь мог задавать и нижнюю, и верхнюю границу диапазона для поиска простых чисел:

```
uses
   System;

//ПРОГРАММА ДЛЯ ПОИСКА ПРОСТЫХ ЧИСЕЛ
//В ЗАДАННОМ ДИАПАЗОНЕ

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

   //бесконечный цикл ввода данных -
```

```
   //пока пользователь не закроет программу
   //или не введёт 0:
   repeat
      var begnum := ReadInteger('Введите начало  диапазона  >  ');
      //если начало диапазона равен 0,
      //то программу закрываем:
      if (begnum = 0) then exit;
      //начало диапазона не меньше двойки:
      if (begnum < 2) then begnum := 2;

      var endnum := ReadInteger('Введите конец  диапазона  >  ');

      //ищем простые числа:
      Primes(begnum, endnum);
   until not (true);
end.
```

Главный блок действует почти так же, как и «эратосфеновский», но вызывает процедуру **Primes** с *двумя* параметрами:

```
//ИЩЕМ ПРОСТЫЕ ЧИСЛА
procedure Primes(n1, n2: integer);
begin
   Console.WriteLine();
   Console.WriteLine('Простые числа:');
   Console.WriteLine();
   //считаем простые числа:
   var n := 0;
   //создаём новый файл для записи результатов поиска:
   var f: PABCSystem.Text;
   assign(f, 'primes.txt');
   append(f);
   WriteLn(f, 'Простые числа:');
   WriteLn(f);
   //просматриваем все числа из заданного диапазона:
   for var j := n1 to n2 do
   begin
      var flg := true;
      //проверяем, делится ли число j
      //на числа 2..корень квадратный из числа j:
      for var i := 2 to Trunc(Math.Sqrt(j)) do
         //если делится, значит, число составное:
         if (j mod i = 0) then
```

```
          begin
             flg := false;
             break;
          end;
       //если нашли простое число,
       if (flg) then
       begin
          n += 1;
          //то печатаем его в текстовом окне:
          Console.Write(j + ' ');
          //и записываем в файл:
          WriteLn(f, j);
       end;
    end; //for j
    WriteLn(f);
    Close(f);

    Console.WriteLine();
    Console.WriteLine('Всего ' + n);
    Console.WriteLine();
 end;//Primes
```

Здесь в цикле ***for*** мы последовательно перебираем числа от *n1=begin* до *n2=end* и проверяем их на простоту.

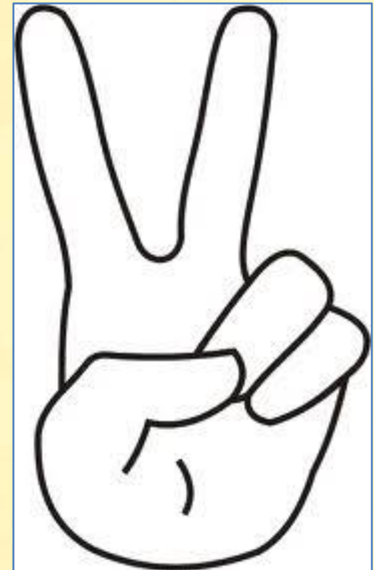
Проверка проходит так: очередное число *j* мы поочерёдно делим на все числа, начиная с двойки и кончая корнем квадратным из самого числа *j*. Так как любое натуральное число делится на единицу и само на себя, то два разных делителя у него имеются в любом случае (кроме, единицы, разумеется). Если мы обнаружим *хотя бы ещё один* делитель, то это будет перебор: число заведомо составное, и проводить испытания дальше нет смысла - мы переходим к проверке следующего числа. Если же число простое, то мы печатаем его на экране и записываем в файл:

Простые числа

```
Простые числа

Введите начало  диапазона  >   2
Введите конец  диапазона  >   1000

Простые числа:

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 10
3 107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199
 211 223 227 229 233 239 241 251 257 263 269 271 277 281 283 293 307 311 313
317 331 337 347 349 353 359 367 373 379 383 389 397 401 409 419 421 431 433 4
39 443 449 457 461 463 467 479 487 491 499 503 509 521 523 541 547 557 563 56
9 571 577 587 593 599 601 607 613 617 619 631 641 643 647 653 659 661 673 677
 683 691 701 709 719 727 733 739 743 751 757 761 769 773 787 797 809 811 821
823 827 829 839 853 857 859 863 877 881 883 887 907 911 919 929 937 941 947 9
53 967 971 977 983 991 997
Всего 168

Введите начало  диапазона  >   1001
Введите конец  диапазона  >   2000

Простые числа:

1009 1013 1019 1021 1031 1033 1039 1049 1051 1061 1063 1069 1087 1091 1093 10
97 1103 1109 1117 1123 1129 1151 1153 1163 1171 1181 1187 1193 1201 1213 1217
 1223 1229 1231 1237 1249 1259 1277 1279 1283 1289 1291 1297 1301 1303 1307 1
319 1321 1327 1361 1367 1373 1381 1399 1409 1423 1427 1429 1433 1439 1447 145
1 1453 1459 1471 1481 1483 1487 1489 1493 1499 1511 1523 1531 1543 1549 1553
1559 1567 1571 1579 1583 1597 1601 1607 1609 1613 1619 1621 1627 1637 1657 16
63 1667 1669 1693 1697 1699 1709 1721 1723 1733 1741 1747 1753 1759 1777 1783
 1787 1789 1801 1811 1823 1831 1847 1861 1867 1871 1873 1877 1879 1889 1901 1
907 1913 1931 1933 1949 1951 1973 1979 1987 1993 1997 1999
Всего 135
```

Простые числа 2

Бесконечный цикл ***repeat-until***
Функция ***ReadInteger***
Условный оператор ***if***
Оператор ***exit***
Процедура *с параметром*
Массив ***integer[]***
Бесконечный цикл ***while***
Оператор ***break***
Вложенные циклы ***for***
Метод ***Math.Sqrt***
Оператор целочисленного деления ***mod***



Недостаток нашего алгоритма кроется в делении *каждого* числа из заданного диапазона *begin..end* на *все* числа, начиная с двойки и заканчивая корнем квадратным. Но мы-то знаем, что делить следует только на уже найденные **простые** числа, которых заведомо меньше. Это значит, что мы должны их где-то хранить. Для этого вполне годится и обычный **массив**.

В этом случае начало диапазона лучше закрепить за двойкой, чтобы список простых чисел заполнялся правильно. Для разнообразия мы вместо верхней границы диапазона будем требовать от пользователя ввода общего количества простых чисел **num**, которые он желает найти:

```
uses
   System;

//ПРОГРАММА ДЛЯ ПОИСКА ПРОСТЫХ ЧИСЕЛ

begin
   //заголовок окна:
   Console.Title := 'Простые числа';
   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Простые числа');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();
```

```
   //бесконечный цикл ввода данных -
   //пока пользователь не закроет программу
   //или не введёт 0:
   repeat
      var num := ReadInteger('Введите количество простых чисел  >  ');
      //если начало диапазона равен 0,
      //то программу закрываем:
      if (num = 0) then exit;

      //ищем простые числа:
      Primes(num);
   until not (true);
end.
```

В процедуре **Primes** мы создаём массив *prime* для хранения найденных простых чисел и сразу же помещаем в него *двойку*:

```
//ИЩЕМ ПРОСТЫЕ ЧИСЛА
procedure Primes(endNumber: integer);
begin
   Console.WriteLine();
   Console.WriteLine('Простые числа:');

   //массив для хранения простых чисел:
   var prime := new integer[endNumber + 1];
   //первое простое число - двойка:
   prime[1] := 2;
   Console.Write(prime[1] + ' ');
   //всего простых чисел:
   var nPrimes := 1;
```

Поскольку двойка – *первое* простое число, а индексы в массиве начинаются с *нуля*, нам пришлось увеличить число элементов массива на *единицу*.

Цикл ***for*** мы начинаем со следующего числа, то есть с *тройки*. Поскольку все остальные простые числа *нечётные*, это избавляет нас от необходимости проверять *все* числа – мы можем ограничиться только *нечётными*, что мы и делаем, добавляя в цикле *while* **двойку** к переменной *num*:

```
var num := 3;
while (true) do
begin
   //если нашли заданное простое число,
   //сообщаем об этом пользователю:
   if (nPrimes = endNumber) then
   begin
      Console.WriteLine();
      Console.WriteLine('Простое число номер ' + endNumber +
                        ' равно ' + prime[endNumber]);
      Console.WriteLine();
      break;
   end;
```

Обратите внимание, что цикл *while* - **бесконечный**. Дело в том, что мы наперёд не знаем, какое из чисел *num* окажется по счёту *endNumber*, поэтому прекращаем цикл, когда найдём *nPrimes = endNumber* простых чисел.

Проверку очередного числа *num* выполняем так. Поочерёдно извлекаем из массива *prime* простые числа, пока не встретится нуль, – на нём найденные простые числа заканчиваются. Также мы ограничиваем проверку условием, что очередное простое число *prime[i]* не больше *Math.Sqrt(num).* Дальше действуем, как обычно:

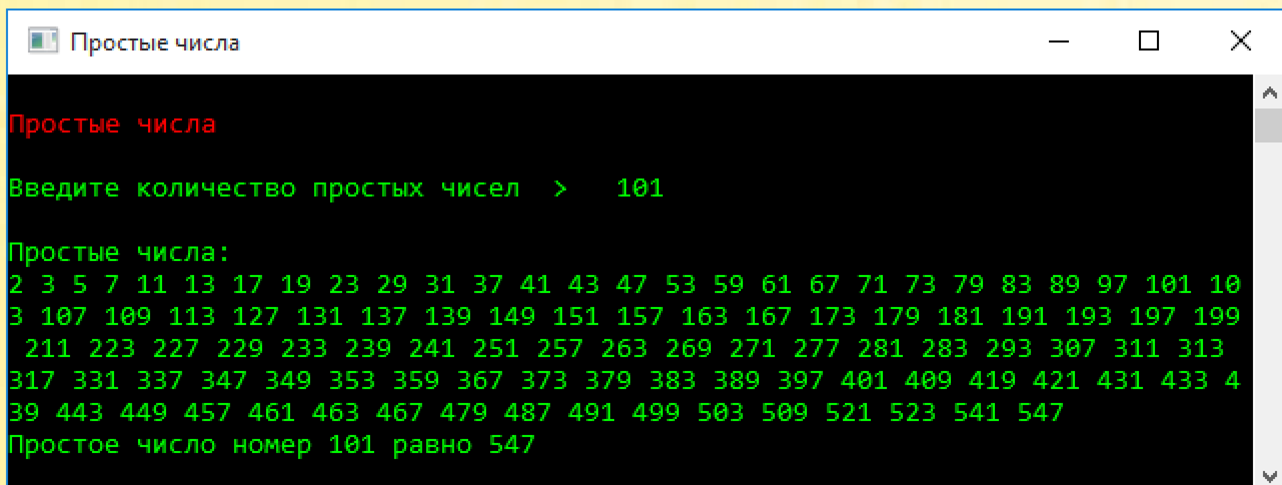
```
   //проверяем, делится число num
   //простые числа из массива prime:
   var flg := true;
   for var i := 1 to prime.Length do
   begin
     //если в массиве встретился нуль,
     //значит, простые числа закончились:
      if (prime[i] = 0) then break;
      //поиск закончен:
      if (prime[i] > Math.Sqrt(num)) then break;

      //если число num делится,
      //значит, число составное:
      if (num mod prime[i] = 0) then
      begin
         flg := false;
         break;
```

```
         end
      end;
```

И если окажется, что число *num* – простое, то мы увеличиваем счётчик простых чисел на единицу и добавляем *num* в массив:

```

      //нашли простое число:
      if (flg) then
      begin
         //подсчитываем:
         nPrimes += 1;
         //добавляем новое простое число в массив:
         prime[nPrimes] := num;
         //и печатаем его в текстовом окне:
         Console.Write(num + ' ');
      end;

      num += 2;
   end;
end;// Primes
```

По ходу поиска простых чисел процедура *Primes* печатает все простые числа на экране, а в конце сообщает, что простое число с заданным номером такое-то (у нас оно всегда последнее из найденных). Например, *101*-ое простое число равно *547*:



Взаимно простые числа

Если наибольший общий делитель двух чисел *m* и *n* равняется *единице*, то такие числа называются **взаимно простыми**:

НОД(m,n) = 1 → числа *m* и *n* - взаимно простые

Для **простых** чисел это условие выполняется всегда, поскольку у них не может быть общих делителей, больших единицы:

НОД(7,11) = 1

Составные числа взаимно просты также только тогда, когда у них нет общих простых делителей:

НОД(10,9) = 1 → числа 10 и 9 - взаимно простые

НОД(10,8) = 2 → числа 10 и 8 – не взаимно простые: у них имеется общий делитель двойка.

Более подробно о взаимно простых числах вы можете прочитать в книге **[OO80]**, на страницах 49-50.

В книге Дагене, Григаса и Аугутиса **[100]**, в *Задаче 22* рассматриваются шестерни с взаимно простыми числами зубьев, что уменьшает их износ.

Факторизация

Бесконечный цикл ***repeat-until***
Функция ***Convert.ToInt64***
Условный оператор ***if***
Оператор ***exit***
Процедура *с параметром*
Вложенные ***циклы***
Цикл ***while***
Цикл ***for***
Оператор целочисленного деления ***mod***

Основная теорема арифметики утверждает, что любое натуральное число, большее единицы, можно представить в виде произведения простых чисел, причём *единственным* способом (если не учитывать их порядок). Например:

21 = 3 * 7
23 = 1 * 23
125 = 5 * 5 * 5

Единичный (со)множитель в разложении можно и не указывать, поскольку он имеется у всех чисел.

Разложение числа на произведение простых множителей, называется его **факторизацией**.

В этом проекте мы используем самый простой способ разложения чисел – полный перебор всех чисел от двух до квадратного корня из заданного числа. Он очень простой, но довольно быстро справляется с задачей даже для очень больших чисел. Однако при встрече с большими **простыми** числами может и спасовать.

Чтобы можно было испытывать и очень большие числа, мы воспользуемся переменной **number** типа *int64*:

```
uses
   System;

//Факторизация

 //ПРОГРАММА ДЛЯ РАЗЛОЖЕНИЯ НАТУРАЛЬНЫХ ЧИСЕЛ
//НА ПРОСТЫЕ МНОЖИТЕЛИ

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

   //бесконечный цикл ввода данных -
   //пока пользователь не закроет программу
   //или не введёт 0:
   repeat
      Console.Write('Введите натуральное число (больше единицы) >  ');
      var number := Convert.ToInt64(Console.ReadLine());
      //если пользователь ввёл 0,
      //то программу закрываем:
      if (number = 0) then exit;

      //находим разложение:
      Razl(number);
   until not (true);
end.
```

В процедуру **Razl** мы передаём испытуемое число как аргумент типа *int64*, а само разложение числа даётся нам очень просто:

```
//Находим разложение заданного числа на простые множители
procedure Razl(num: int64);
begin
   Console.WriteLine();
   Console.WriteLine('Разложение числа:');
   Console.Write(num.ToString + ' = ');

   //проверяем, делится ли num на числа 2..заданное число:
   var i: int64 := 2;
```

```
    while(i <= num) do
    begin
       //если делится, выписываем делитель:
       while (num mod i = 0) do
       begin
          Console.Write(i.ToString());
          num := num div i;
          if(num >= i) then
             Console.Write(' * ');
       end;
       i += 1;
    end;
    Console.WriteLine();
    Console.WriteLine();
end;//Razl
```

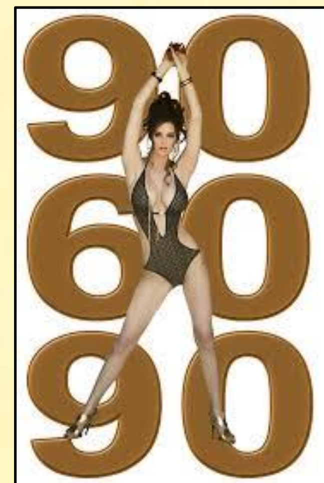
Факторизация числа

```
Факторизация числа

Введите натуральное число (больше единицы) >  111111

Разложение числа:
111111 = 3 * 7 * 11 * 13 * 37

Введите натуральное число (больше единицы) >  1111111

Разложение числа:
1111111 = 239 * 4649

Введите натуральное число (больше единицы) >  11111111

Разложение числа:
11111111 = 11 * 73 * 101 * 137

Введите натуральное число (больше единицы) >  111111111

Разложение числа:
111111111 = 3 * 3 * 37 * 333667

Введите натуральное число (больше единицы) >  1111111111

Разложение числа:
1111111111 = 11 * 41 * 271 * 9091

Введите натуральное число (больше единицы) >  11111111111

Разложение числа:
11111111111 = 21649 * 513239
```

Факторизация числа

```
Введите натуральное число (больше единицы) >  111111111111

Разложение числа:
111111111111 = 3 * 7 * 11 * 13 * 37 * 101 * 9901

Введите натуральное число (больше единицы) >  1111111111111

Разложение числа:
1111111111111 = 53 * 79 * 265371653

Введите натуральное число (больше единицы) >  11111111111111

Разложение числа:
11111111111111 = 11 * 239 * 4649 * 909091

Введите натуральное число (больше единицы) >  111111111111111

Разложение числа:
111111111111111 = 3 * 31 * 37 * 41 * 271 * 2906161

Введите натуральное число (больше единицы) >  1111111111111111

Разложение числа:
1111111111111111 = 11 * 17 * 73 * 101 * 137 * 5882353

Введите натуральное число (больше единицы) >
```

Совершенные числа



Бесконечный цикл ***repeat-until***
Тип данных ***int64***
Метод ***Convert.ToInt64***
Оператор ***exit***
Условный оператор ***if***
Процедура *с параметром*
Вложенные циклы ***for***

В книге Брудно и Каплана *Олимпиады по программированию для школьников* [БК85] имеется такая задача:

84.3. Совершенные числа.

Совершенным называется натуральное число, равное сумме своих делителей, исключая само число. Первое совершенное число равно шести: 6 = 1 + 2 + 3. Второе равно 28 = 1 + 2 + 4 + 7 + 14. Третье – 496, четвёртое – 8128. Следующие совершенные числа уже гораздо больше.

Напечатать все совершенные числа, меньшие, чем заданное M.

По-английски совершенные числа называются **perfect numbers**.

На сайте *The OEIS Foundation* http://oeis.org/A000396 вы найдёте **последовательность совершенных чисел**:

```
6
28
496
8128
33 550 336
8 589 869 056
137 438 691 328, 23 05 843 008 139 952 128
2 658 455 991 569 831 744 654 692 615 953 842 176
191 561 942 608 236 107 294 793 378 084 303 638 130 997 321 548 169 216
```

Легко заметить, что первые 4 совершенных числа достаточно маленькие, а затем они стремительно увеличиваются!

Из этого наблюдения следует, что *первые* совершенные числа можно искать **методом грубой силы**:

```
uses
   System;

//Совершенные числа

begin
```

```
   //заголовок окна:
   Console.Title := 'Совершенные числа';
   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Совершенные числа');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   //бесконечный цикл ввода данных -
   //пока пользователь не закроет программу
   //или не введёт 0:
   repeat
      Console.Write('Введите верхнюю границу  >  ');
      var number := Convert.ToInt64(Console.ReadLine());
      //если пользователь ввёл 0,
      //то программу закрываем:
      if (number = 0) then exit;

      //находим совершенные числа:
      PerfectNumbers(number);
   until not (true);
end.
```

Практически такой же алгоритм поиска совершенных чисел предлагается и в книге Брудно и Каплана:

```
procedure PerfectNumbers(max: int64);
begin
   //сумма делителей:
   var summa: int64;
   var divisor: int64;

   for var num: int64 := 2 to max do
   begin
      summa := 1;
      for divisor := 2 to Trunc(Math.Sqrt(num)) do
      begin
         if (num mod divisor = 0) then
         begin
            summa += divisor;
            var n := num div divisor;
            if (divisor <> n) then
```

```
               summa += n;
         end
      end;
      // печатаем число в консольном окне:
      if (summa = num) then
         Console.WriteLine('Число ' + num + ' совершенное');
   end;
   Console.WriteLine();
end;
```

```
Совершенные числа

Совершенные числа

Введите верхнюю границу  >  10000
Число 6 совершенное
Число 28 совершенное
Число 496 совершенное
Число 8128 совершенное
```

Увы, дальше четвёртого числа с методом *PerfectNumbers* вы не продвинетесь!

Квадраты

Бесконечный цикл ***repeat-until***
Тип данных ***int64***
Функция ***ReadInteger***
Оператор ***exit***
Условный оператор ***if***
Функция *с параметром*
Цикл ***for***

Задача 6 из книги В. А. Дагене, Г. К. Григас, К. Ф. Аугутис *100 задач по программированию* [100], страница 27:

> Квадрат любого натурального числа *n* равен сумме *n* первых нечётных чисел:

$1^2 = 1$
$2^2 = 1 + 3$
$3^2 = 1 + 3 + 5$
$4^2 = 1 + 3 + 5 + 7$
$5^2 = 1 + 3 + 5 + 7 + 9$
. . .

Основываясь на данном свойстве, **составьте программу, позволяющую напечатать квадраты натуральных чисел от** 1 до *n*.

В **главном блоке** мы организуем традиционный бесконечный цикл ввода данных от пользователя, которому предлагаем ввести число, квадрат которого он хотел бы получить:

```
uses
   System;

//Дагене, #6, с.27

begin
   //заголовок окна:
   Console.Title := 'Дагене, #6, с.27';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine('Квадраты чисел');
   Console.WriteLine();

   //бесконечный цикл ввода данных -
   //пока пользователь не закроет программу
   //или не введет 0:
   repeat
      Console.ForegroundColor := ConsoleColor.Red;
      var num := ReadInteger('Введите число  >  ');

      //если конец диапазона равен 0,
      //то программу закрываем:
      if (num = 0) then exit;

      //вычисляем квадрат заданного числа:
      var q := Quadrat(num);
      Console.ForegroundColor := ConsoleColor.Green;
```

```
      Console.WriteLine();
      Console.WriteLine('Квадрат числа ' + num + ' = ' + q.ToString());
      Console.WriteLine();
   until not (true);

end.
```

Чтобы «ворочать» большими числами, выбираем тип данных **int64**.

Метод **Quadrat** вычисляет квадрат заданного числа *num* по алгоритму, описанному выше:

```
//ВЫЧИСЛЯЕМ КВАДРАТ
function Quadrat(num: integer): int64;
begin
   //нечётное число:
   var nech := 1;
   //квадрат:
   var q: int64 := 0;
   //вычисляем квадрат как сумму нечётных чисел:
   for var i := 1 to num do
   begin
      q += nech;
      nech += 2;
   end;

   Result := q;
end;
```

Несмотря на множество операций сложения, квадраты даже очень больших чисел функция *Quadrat* находит очень быстро:

```
Дагене, #6, с.27

Квадраты чисел

Введите число  >   111111
Квадрат числа 111111 = 12345654321

Введите число  >   1111111
Квадрат числа 1111111 = 1234567654321

Введите число  >   11111111
Квадрат числа 11111111 = 123456787654321

Введите число  >   111111111
Квадрат числа 111111111 = 12345678987654321
```

Кубы

Бесконечный цикл ***repeat-until***
Функция ***ReadInteger***
Тип данных ***int64***
Условный оператор ***if***
Оператор ***exit***
Фуекция *с параметром*
Цикл ***for***
Вложенные циклы ***for***

Задача 6⊙ из книги В. А. Дагене, Г. К. Григас, К. Ф. Аугутис *100 задач по программированию* [100], страницы 27-28:

> Куб любого натурального числа ***n*** равен сумме *n* нечётных чисел, следующих по порядку за числами, сумма которых составила куб числа *n - 1*:
>
> $1^3 = 1$
> $2^3 = 3 + 5$
> $3^3 = 7 + 9 + 11$
> $4^3 = 13 + 15 + 17 + 19$
> $5^3 = 21 + 23 + 25 + 27 + 29$
>
> Основываясь на этом свойстве, **создайте программу, позволяющую напечатать таблицу кубов натуральных чисел.**

Эта задача посложнее «квадратной»! Хотя мы и знаем, сколько нечётных чисел входит в сумму, но не знаем, с какого числа начинается ряд слагаемых.

Выпишем номера нечётных чисел, с которых начинается суммирование (верхняя строка):

```
1  2  4  7  11
 1  2  3  4
  1  1  1
```

Затем найдём разность между соседними числами (средняя строка) и повторим эту операцию ещё раз. В нижней строке все числа **одинаковые**, это значит, что номер нечётного числа можно вычислить по формуле:

$$a \bullet num^2 + b \bullet num + c$$

Для нахождения коэффициентов этого выражения, следует воспользоваться **методом исчисления конечных разностей**. Он описан, например, в моей книге *Как решать комбинаторные задачи на компьютере*.

В итоге мы получим вот такую замечательную **формулу**:

$$n = num\,(num-1)\,/\,2 + 1 \qquad (1)$$

По номеру **n** мы легко найдём и само нечётное число. Оно равняется:

$$2n - 1 \qquad (2)$$

Эти две формулы можно объединить в одну:

Первое нечётное число $= num\,(num-1) + 1$

Если ряд чисел не очень сложный, то можно поискать для него формулу в Интернете. Например, для нашей последовательности 1, 2, 4, 7, 11 я сразу нашёл общую формулу:

http://znanija.com/task/141487
http://answers.yahoo.com/question/index?qid=20080903051936AAUvpkC
http://michaelnela.hubpages.com/question/87442/what-is-the-general-term-of-1-2-4-7-11-16-22

Главный блок требует только небольшой правки:

```
uses
   System;

//Дагене, #6, c.27-28

begin
   //заголовок окна:
   Console.Title := 'Дагене, #6, c.27-28';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine('Кубы чисел');
   Console.WriteLine();

   //бесконечный цикл ввода данных -
   //пока пользователь не закроет программу
   //или не введет 0:
   repeat
      Console.ForegroundColor := ConsoleColor.Red;
      var num := ReadInteger('Введите число  >  ');

      //если конец диапазона равен 0,
      //то программу закрываем:
      if (num = 0) then exit;

      //вычисляем куб заданного числа:
      var q := Qube(num);
      Console.ForegroundColor := ConsoleColor.Green;
      Console.WriteLine();
      Console.WriteLine('Куб числа ' + num + ' = ' + q.ToString());
      Console.WriteLine();
   until not (true);
end.
```

В функции **Qube** мы сначала вычисляем номер *n* первого нечётного числа суммы, а затем по формуле (2) и само нечётное число. Остальная часть функции ничем не отличается функции *Quadrat*:

```
//ВЫЧИСЛЯЕМ КУБ
```

```
function Qube(num: integer): int64;
begin
   //номер нечётного числа:
   var n := num*(num-1) div 2 + 1;
   //нечётное число:
   var nech := 2*n - 1;
   //куб:
   var q: int64 := 0;
   //вычисляем куб как сумму нечётных чисел::
   for var i := 1 to num do
   begin
      q += nech;
      nech += 2;
   end;

   Result := q;
end;
```

Начинаем осторожно проверять работу нового метода на небольших числах, кубы которых нам точно известны. А потом вы можете заняться вычислением БОЛЬШИХ кубов:

```
Дагене, #6, с.27-28

Кубы чисел

Введите число  >   1
Куб числа 1 = 1

Введите число  >   2
Куб числа 2 = 8

Введите число  >   3
Куб числа 3 = 27

Введите число  >   4
Куб числа 4 = 64

Введите число  >   5
Куб числа 5 = 125
```

```
Дагене, #6, с.27-28

Кубы чисел

Введите число  >   1111
Куб числа 1111 = 1371330631

Введите число  >   12345
Куб числа 12345 = 1881365963625

Введите число  >   3333
Куб числа 3333 = 37025927037

Введите число  >   2016
Куб числа 2016 = 8193540096

Введите число  >   2017
Куб числа 2017 = 8205738913
```

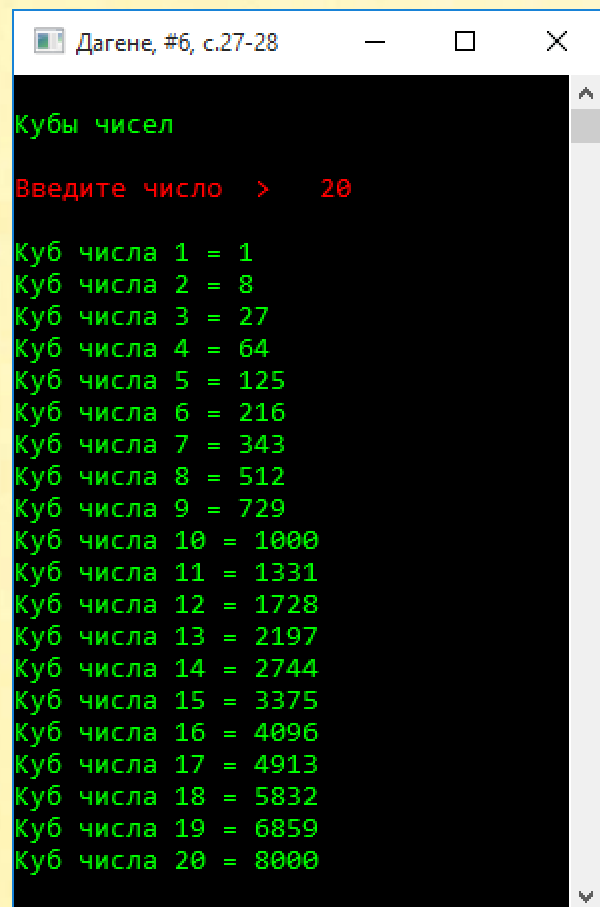
Кубическая задача успешно решена!

Пользуясь функцией *Qube,* можно составить **таблицу** кубов, о которой идёт речь в задаче, но лучше написать отдельную процедуру, тем более что она не отнимет у вас много сил:

```
procedure PrintTable(num: integer);
begin
   Console.WriteLine();
   var nech := 1;
   //для всех чисел 1..num:
   for var i := 1 to num do
   begin
      var q: int64 := 0;
      for var j := 1 to i do
       begin
         q += nech;
         nech += 2;
      end;
      Console.ForegroundColor := ConsoleColor.Green;
      Console.WriteLine('Куб числа ' + i + ' = ' + q.ToString());
   end
end;
```

Алгоритм, заложенный в эту процедуру, буквально повторяет условие задачи. Мы последовательно находим сумму нечётных чисел, продолжая их ряд с каждым новым числом, для которого находим куб.

На рисунке вы видите отчёт о проделанной работе:



Факториал

Бесконечный цикл ***repeat-until***
Функция ***ReadInteger***
Тип данных ***ulong***
Функция *с параметром*
Условный оператор ***if***
Оператор ***return***
Оператор приведения ***uint64()***
Цикл ***for***
Тип данных ***decimal***

Произведение натуральных чисел от единицы до заданного (пусть это будет *n*) называется **факториалом**. Обозначается факториал восклицательным знаком после числа:

> ***n!*** (читается: *эн-факториал*)

Чтобы его вычислить, нужно, как и следует из определения, просто перемножить все числа от единицы до этого числа, включительно:

> n! = 1 x 2 x . . . x (n-1) x n (1)

> По определению, 0! = 1.

Формула очень простая, но нетрудно догадаться, что для больших значений *n* факториал будет выражаться огромным числом. Поэтому мы сначала поэкспериментируем с числами типа *uint64*, а потом перейдём на тип *decimal*.

Обычно для вычисления факториала используют два алгоритма - *рекурсивный* и *итерационный*. Нас интересует только итерационный - он работает быстрее.

В **главном блоке** пользователь вводит число от 0 до 20, после чего вызывается функция *factorial*, которая возвращает вычисленное значение факториала:

```
uses
   System;

//Факториал
//ПРОГРАММА ДЛЯ ВЫЧИСЛЕНИЯ ФАКТОРИАЛОВ

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

   //бесконечный цикл ввода данных -
   //пока пользователь не закроет программу
   //или не введёт отрицательное число:
   repeat
      Console.ForegroundColor := ConsoleColor.Red;
      var num := ReadInteger('Введите число (0..20) >  ');
      if (num < 0) then exit;

      //находим факториал заданного числа:
      var fact: uint64 := factorial(num);

      //печатаем факториал:
      Console.ForegroundColor := ConsoleColor.Green;
      Console.WriteLine(num + '! = ' + fact.ToString());
      Console.WriteLine();
   until not (true);


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

Функция **factorial** действует строго по формуле (1):

```
//ВЫЧИСЛЯЕМ ФАКТОРИАЛ ЗАДАННОГО ЧИСЛА
```

```
function factorial(num: integer): uint64;
begin
   if (num = 0) then
   begin
      Result := 1;
      exit;
   end;

   var fact: uint64 := 1;
   for var i: uint64 := 2 to uint64(num) do
      fact *= i;
   Result := fact;
end;
```

Умножать на единицу, конечно, смысла нет.

Здесь мы учитываем, что факториал **нуля** равен *единице*, это особый случай и его нужно учесть отдельно:

```
Факториал

Введите число (0..20) >   0
0! = 1

Введите число (0..20) >   1
1! = 1

Введите число (0..20) >   5
5! = 120

Введите число (0..20) >   20
20! = 2432902008176640000
```

Попытка вычислить факториал числа 21 ни к чему хорошему не приводит. Результат получается неверным:

```
Факториал

Введите число (0..20) >   21
21! = 14197454024290336768
```

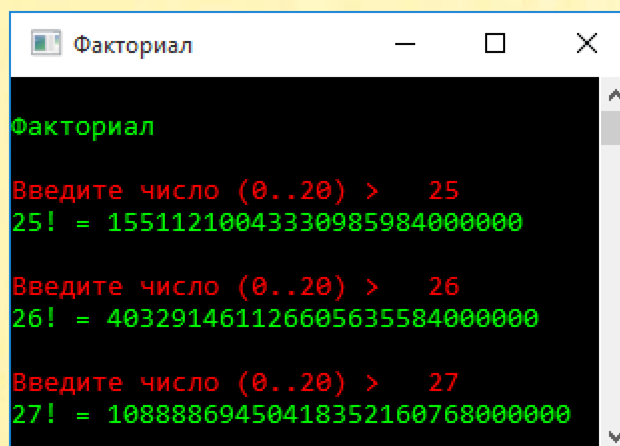
Очень большие факториалы можно вычислять с помощью чисел типа **decimal**. Для этого исправьте код в главной программе:

```
//находим факториал заданного числа:
//var fact: uint64 := factorial(num);
var fact: decimal := factorial(num);
```

И напишите новую **функцию**:

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

   var fact: decimal := 1;
   for var i := 2 to num do
      fact := fact * decimal(i);
   Result := fact;
end;
```

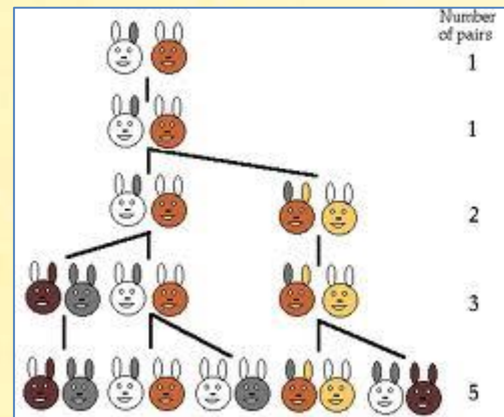
Последний факториал для этого типа данных равен 27!



А что делать, если нужен факториал, к примеру, числа 8000 или даже 50000? - Воспользоваться структурой **BigInteger**, которая позволяет использовать *целые* числа *любой* длины.

Числа Фибоначчи

Бесконечный цикл ***repeat-until***
Функция ***ReadInteger***
Массив ***decimal[]***
Тип данных ***decimal***
Условный оператор ***if***
Цикл ***for***
Функция *с параметром-массивом*
Тип данных ***int64***
Оператор ***or***
Оператор ***exit***



Эти числа открыл *Фибоначчи* (он же *Леонардо Пизанский*), средневековый математик, автор *Книги абака (Liber Abaci)*, которую он написал в 1202 году.

Леонардо Пизанский (Leonardo Pisano, 1170 - 1250), по прозвищу Фибоначчи (Fibonacci)

Впрочем, дотошные историки утверждают, что этот ряд чисел был известен в Индии задолго до Фибоначчи, где он использовался при стихосложении.

В трактате *Книга абака* Фибоначчи рассмотрел математическую модель, связанную с кроликами. Он взял пару взрослых кроликов (точнее, кролика и крольчиху) и предположил, что они могут производить на свет потомство каждый месяц. Причём у них всегда рождается пара крольчат разного пола, у которых через два месяца также рождаются крольчата. Фибоначчи решил подсчитать, сколько будет кроликов через год, если за это время ни один кролик не умрёт. Числа Фибоначчи как раз и отражают рост популяции кроликов.

В книгах по программированию принято находить числа Фибоначчи с помощью красивого *рекурсивного* алгоритма, который основан на рекуррентном определении самих чисел:

Первое число Фибоначчи = **0**
Второе число = **1**

Все последующие равны сумме двух предыдущих, то есть:

Третье число = 0 + 1 = **1**
Четвёртое = 1 + 1= *2*
Пятое = 1 + 2 = *3*

и так далее, до бесконечности.

Рекурсивные алгоритмы обычно довольно короткие, но не всегда быстрые. Поэтому для ускорения вычислений мы воспользуемся **методом динамического программирования**, то есть просто *запомним* уже найденные числа Фибоначчи в массиве. Этим мы убьём сразу двух зайцев (или кроликов) – и заданное число получим, и все предыдущие числа сохраним!

В **главном блоке** пользователь вводит любое неотрицательное число (но не больше 139),после чего функция *Fibo* находит числа Фибоначчи от нулевого до заданного.

Чтобы сохранить все промежуточные числа, создадим массив чисел **f** типа *decimal* и сразу же поместим в него три первых числа Фибоначчи, чтобы можно было найти следующие:

```
uses
   System;

//Числа Фибоначчи
//ПРОГРАММА ДЛЯ ВЫЧИСЛЕНИЯ ЧИСЕЛ ФИБОНАЧЧИ

begin
   //заголовок окна:
   Console.Title := 'Числа Фибоначчи';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Числа Фибоначчи');
   Console.WriteLine();

   //бесконечный цикл ввода данных -
   //пока пользователь не закроет программу
   //или не введёт отрицательное число:
   repeat
      Console.ForegroundColor := ConsoleColor.Red;
      var num := ReadInteger('Введите число (0..139) >  ');
      if (num < 0) then
         exit;

      var f := new decimal[3];
      if (num > 2) then
         f := new decimal[num + 1];
      //помещаем в массив первые числа Фибоначчи:
      f[0] := 0;
      f[1] := 1;
      f[2] := 1;

      //находим все числа Фибоначчи от 0 до num:
      var nfibo := Fibo(f);
      //и печатаем их:
      Console.ForegroundColor := ConsoleColor.Green;
      for var i := 0 to num do
         Console.WriteLine('Число  Фибоначчи ' + i +
                          ' = ' + f[i].ToString());
```

```
      Console.WriteLine();
   until not (true);

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

Далее мы передаём функции Fibo этот массив, а он по длине массива *f.Length* определяет максимальное число Фибоначчи, которое желает узнать пользователь. Сам алгоритм следует определению чисел Фибоначчи:

```
//ВЫЧИСЛЯЕМ ЗАДАННОЕ ЧИСЛО ФИБОНАЧЧИ
function Fibo(f : array of decimal): decimal;
begin
    var n: int64 := f.Length - 1;
    if ((n = 1) or (n = 2)) then
    begin
      Result := n;
      exit;
    end;
    for var i := 3 to n do
        f[i] := f[i - 1] + f[i - 2];
    Result := f[n];
end;
```

В итоге мы получаем массив *f*, наполненный числами Фибоначчи, которые и распечатываем в главном блоке. Дополнительно функция *Fibo* возвращает последнее из найденных чисел. Оно соответствует заданному пользователем номеру числа Фибоначчи. Из этого следует, что если пользователю нужно только оно, то массив *f* можно объявить непосредственно в функции *Fibo*, а пользователю возвращать только последнее из найденных чисел.

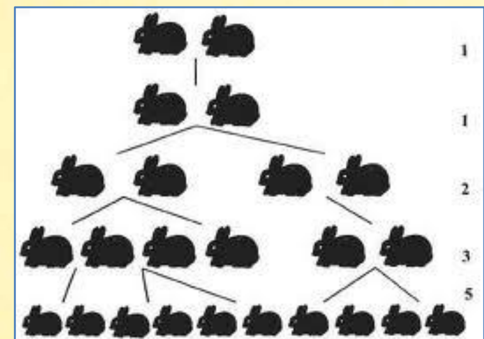
Наша уловка с массивом позволяет практически мгновенно находить совершенно невероятные числа Фибоначчи:

```
Числа Фибоначчи

Введите число (0..139) >   139
Число  Фибоначчи 0 = 0
Число  Фибоначчи 1 = 1
Число  Фибоначчи 2 = 1
Число  Фибоначчи 3 = 2
Число  Фибоначчи 4 = 3
Число  Фибоначчи 5 = 5
Число  Фибоначчи 6 = 8
Число  Фибоначчи 7 = 13
Число  Фибоначчи 8 = 21
Число  Фибоначчи 9 = 34
Число  Фибоначчи 10 = 55
Число  Фибоначчи 11 = 89
Число  Фибоначчи 12 = 144
Число  Фибоначчи 13 = 233
Число  Фибоначчи 14 = 377
Число  Фибоначчи 15 = 610
Число  Фибоначчи 16 = 987
Число  Фибоначчи 17 = 1597
Число  Фибоначчи 18 = 2584
Число  Фибоначчи 19 = 4181
Число  Фибоначчи 20 = 6765
Число  Фибоначчи 21 = 10946
Число  Фибоначчи 22 = 17711
Число  Фибоначчи 23 = 28657
Число  Фибоначчи 24 = 46368
Число  Фибоначчи 25 = 75025
```

```
Число  Фибоначчи 111 = 70492524767089125814114
Число  Фибоначчи 112 = 114059301025943970552219
Число  Фибоначчи 113 = 184551825793033096366333
Число  Фибоначчи 114 = 298611126818977066918552
Число  Фибоначчи 115 = 483162952612010163284885
Число  Фибоначчи 116 = 781774079430987230203437
Число  Фибоначчи 117 = 1264937032042997393488322
Число  Фибоначчи 118 = 2046711111473984623691759
Число  Фибоначчи 119 = 3311648143516982017180081
Число  Фибоначчи 120 = 5358359254990966640871840
Число  Фибоначчи 121 = 8670007398507948658051921
Число  Фибоначчи 122 = 14028366653498915298923761
Число  Фибоначчи 123 = 22698374052006863956975682
Число  Фибоначчи 124 = 36726740705505779255899443
Число  Фибоначчи 125 = 59425114757512643212875125
Число  Фибоначчи 126 = 96151855463018422468774568
Число  Фибоначчи 127 = 155576970220531065681649693
Число  Фибоначчи 128 = 251728825683549488150424261
Число  Фибоначчи 129 = 407305795904080553832073954
Число  Фибоначчи 130 = 659034621587630041982498215
Число  Фибоначчи 131 = 1066340417491710595814572169
Число  Фибоначчи 132 = 1725375039079340637797070384
Число  Фибоначчи 133 = 2791715456571051233611642553
Число  Фибоначчи 134 = 4517090495650391871408712937
Число  Фибоначчи 135 = 7308805952221443105020355490
Число  Фибоначчи 136 = 11825896447871834976429068427
Число  Фибоначчи 137 = 19134702400093278081449423917
Число  Фибоначчи 138 = 30960598847965113057878492344
Число  Фибоначчи 139 = 50095301248058391139327916261
```

Числа Фибоначчи 2

*Бесконечный цикл **repeat-ubtil***
*Функция **ReadInteger***
*Тип данных **decimal***
***Функция** с параметром*
*Цикл **for***
*Оператор **exit***



Легко заметить, что из всего массива чисел Фибоначчи для вычисления следующего числа нужны только **два последних** элемента. И если все числа

Фибоначчи от первого до заданного сохранять не нужно, то можно обойтись вообще без массива. И при этом мы не потеряем скорость вычислений! Вот как это делается.

Клонируйте предыдущий проект и исправьте **главный блок**:

```
uses
   System;

//Числа Фибоначчи 2
//ПРОГРАММА ДЛЯ ВЫЧИСЛЕНИЯ ЧИСЕЛ ФИБОНАЧЧИ

begin
   //заголовок окна:
   Console.Title := 'Числа Фибоначчи';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Числа Фибоначчи');
   Console.WriteLine();

   //бесконечный цикл ввода данных -
   //пока пользователь не закроет программу
   //или не введёт отрицательное число:
   repeat
      Console.ForegroundColor := ConsoleColor.Red;
      var num := ReadInteger('Введите число (0..139) >  ');
      if (num < 0) then
         exit;

      //находим число Фибоначчи:
      var nfibo := Fibo2(num);
      //и печатаем его:
      Console.ForegroundColor := ConsoleColor.Green;
      Console.WriteLine('Число  Фибоначчи ' + num +
                        ' = ' + nfibo.ToString());
      Console.WriteLine();
   until not (true);

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

Как видите, кода стало меньше, но теперь мы получим из функции *Fibo2* только **одно** число. Впрочем, вы можете вызывать этот метод в цикле и получать все числа Фибоначчи от нулевого до заданного. Конечно, придётся выполнить лишнюю работу, но скорость программы такова, что вы ничего не заметите.

Функция Fibo2 для нахождения чисел Фибоначчи тоже упростилась по сравнению с первой версией:

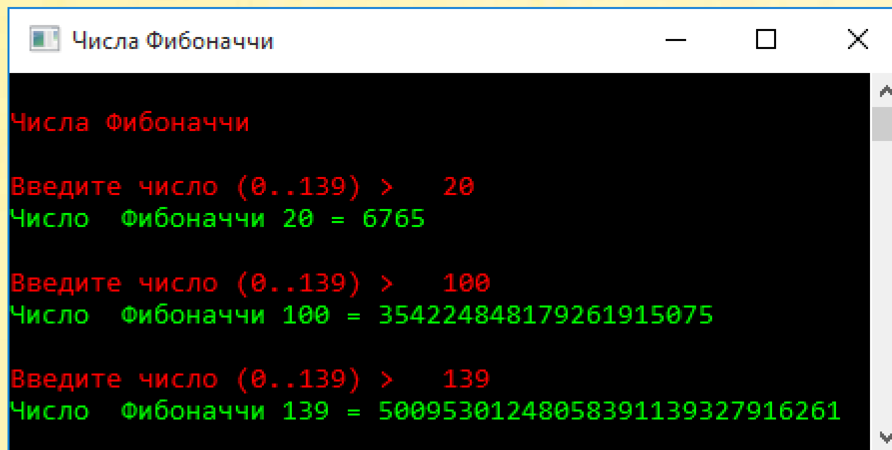
```
//НАХОДИМ ЧИСЛО ФИБОНАЧЧИ БЕЗ МАССИВА
function Fibo2(n: integer): decimal;
begin
   var f1: decimal := 0;
   var f2: decimal := 1;
   var f: decimal := 1;
   for var ri := 1 to n do
   begin
      f := f1 + f2;
      f2 := f1;
      f1 := f;
   end;
   Result := f;
end;
```

Здесь нам понадобились три переменные типа *decimal* f1, f2 и f.

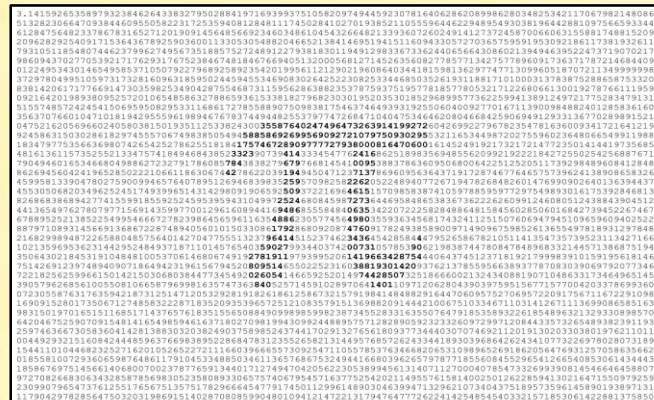
Первые два числа Фибоначчи мы задаём сразу, а все последующие вычисляем как сумму двух предыдущих. То есть *третье* (если нулевое число Фибоначчи считать первым (нередко последовательность чисел Фибоначчи начинается с единицы, а не с нуля, поэтому и возникают подобные «разночтения») число 0 + 1 = *1*, четвёр-тое 1 + 1= *2*, пятое 1 + 2 = *3* и так далее:

```
      f := f1 + f2;
      f2 := f1;
      f1 := f;
```

Проверка подтверждает наши ожидания – новый алгоритм считает быстро и аккуратно:



Вычисляем пи и е



- *Процедура без параметров*
- *Тип данных* ***double***
- *Цикл* ***for***
- *Оператор деления* ***/***
- *Комбинированные операторы присваивания*
- *Тип данных* ***decimal***
- *Метод* ***Math.Pow***

С помощью числовых рядов можно вычислить самые знаменитые иррациональные числа – ***π*** и ***e*** (поскольку они выражаются *бесконечной* десятичной дробью, то мы сможем найти только *приближённое* значение).

Современное **обозначение** этого числа греческой буквой *пи* предложил в 1706 году английский математик У.Джонсон. Он воспользовался первой буквой греческого слова *periferia* (конечно, в оригинальном написании, а не в более удобном - латинскими буквами), что значит *окружность*. Но общепризнанным в научном мире этот символ стал после того как в 1736 году Леонард Эйлер использовал его в своих работах.

История вычислений числа *пи*, которое равно отношению длины окружности к её диаметру, началась много тысячелетий назад.

Так, египетские математики считали *пи* равным $(16/9)^2$, то есть примерно равно 3,1604938…, а индийские – $\sqrt{10}$ = 3,16227766… В третьем веке до нашей эры Архимед установил, что *пи* меньше, чем 3 1/7 и больше, чем 3 10/71 → 3,1428 … 3,1408. Среднее значение этого диапазона равно 3,14185, что больше *пи* уже в четвёртом десятичном знаке. Но ещё до Архимеда, в пятом веке до нашей эры китайский математик Цзу Чунчжи нашёл более точное значение - 3,1415927... В первой половине пятнадцатого века ал-Каши, астроном и математик из Самарканда вычислил *пи* с точностью до 16 знаков после запятой. В 1615 году голландский математик Лудольф ван Цейлен довёл точность вычислений до 32 знаков. В 1873 году Вильям Шенкс после 20 лет расчётов нашёл 707 знаков числа *пи*, но в 1944 году Д.Фергюсон с помощью механического калькулятора выяснил, что верны только первые 527 знаков числа Шенкса. Для своих расчётов Шенкс использовал *формулу Дж. Мачина*:

$$\frac{\pi}{4} = 4\arctan\left(\frac{1}{5}\right) - \arctan\left(\frac{1}{239}\right)$$

А *арктангенс* вычислял по формуле

$$\arctan x = \sum_{k=0}^{\infty} \frac{(-1)^k x^{2k+1}}{2k+1} = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \cdots$$

Сам *Мачин* ещё в 1706 году по этой формуле вычислил 100 знаков числа *пи*.

John Machin (1686? - 1751)

С появлением ЭВМ скорость вычислений значительно выросла. В 1949 году электронная машина ЭНИАК за 70 часов работы вычислила более двух тысяч десятичных знаков числа *пи*. Через некоторое время были найдены 3000 знаков всего за 13 минут. В 1959 году компьютеры преодолели рубеж десяти тысяч знаков, а сейчас известно несколько десятков миллионов знаков числа *пи*. Чтобы их напечатать, потребуется несколько толстенных книг.

Число *пи* с точностью 500 знаков после запятой:

π ≈ 3,141 592 653 589 793 238 462 643 383 279 502 884 197 169 399 375 105 820 974 944 592 307 816 406 286 208 998 628 034 825 342 117 067 982 148 086 513 282 306 647 093 844 609 550 582 231 725 359 408 128 481 117 450 284 102 701 938 521 105 559 644 622 948 954 930 381 964 428 810 975 665 933 446 128 475 648 233 786 783 165 271 201 909 145 648 566 923 460 348 610 454 326 648 213 393 607 260 249 141 273 724 587 006 606 315 588 174 881 520 920 962 829 254 091 715 364 367 892 590 360 011 330 530 548 820 466 521 384 146 951 941 511 609 433 057 270 365 759 591 953 092 186 117 381 932 611 793 105 118 548 074 462 379 962 749 567 351 885 752 724 891 227 938 183 011 949 12…

Конечно, все эти знаки не упомнить (да и не надо!), а вот несколько первых знать совсем не помешает. Помню в школьной стенной газете был такой стишок для запоминания первых цифр числа *пи*:

Чтобы нам не ошибаться,
Надо правильно прочесть:
Три, четырнадцать, пятнадцать,
Девяносто два и шесть.

Я его запомнил и надеюсь, что и вам это удастся!

Формула Валлиса

$$\pi = 2\left(\frac{2}{1}\times\frac{2}{3}\times\frac{4}{3}\times\frac{4}{5}\times\frac{6}{5}\times\frac{6}{7}\times\frac{8}{7}\times\frac{8}{9}\cdots\right)$$

Английский математик *Джон Валлис* вывел эту красивую формулу в 1655 году, когда вычислял площадь круга. Она представляет собой бесконечное произведение дробей.

John Wallis by Sir Godfrey Kneller (1616 - 1703)

К сожалению, чтобы вычислить даже несколько правильных знаков числа *пи* по этой формуле, нужно затратить немало времени и сил. Однако, имея компьютер, мы можем облегчить себе задачу. Итак, пишем **программу**:

```
uses
   System;

//Вычисляем пи и е

begin
   //заголовок окна:
   Console.Title := 'Вычисляем пи и е';
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   Wallis();

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

```
//ВЫЧИСЛЕНИЕ ПИ ПО ФОРМУЛЕ ВАЛЛИСА
procedure Wallis();
begin
   var pi := 1.0;
   var i: integer;
   for i := 1 to 100000000 do
        pi *= 4.0 * i * i / (2 * i - 1) / (2 * i + 1);

   Console.WriteLine('i = ' + i + ' pi = ' + 2*pi);
   Console.WriteLine();
end;
```

После ста миллионов итераций получаем **ответ**:

```
Вычисляем пи и е
i = 100000000 pi = 3.14159265834634
```

Ряд Лейбница

Попробуем зайти с другого конца и воспользуемся знакочередующимся рядом, который предложил немецкий математик *Лейбниц*.

Gottfried Wilhelm von Leibniz (1646 – 1716)

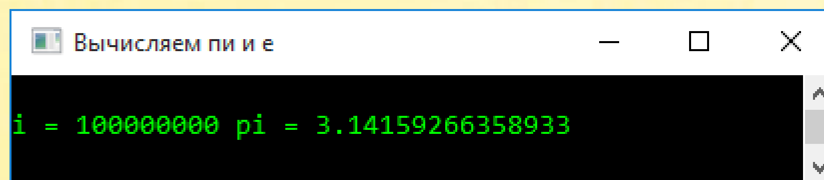
А ряд вот такой:

$$\pi = 4\left(\frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} \ldots\right)$$

Он хорош тем, что его легко приспособить под наши нужды:

```
//ВЫЧИСЛЯЕМ ПИ - Ряд Лейбница
//pi/4 = 1 - 1/3 + 1/5 - 1/7 + ...
procedure CalcPi();
begin
   var pi := 0.0;
   var neg := 1.0;
   var i: integer;

   for i := 0 to 100000000 do
   begin
        pi += neg/(i*2.0 + 1.0);
     neg := -neg;
   end;
   Console.WriteLine('i = ' + i + ' pi = ' + 4*pi);
   Console.WriteLine();
end;
```

После тех же ста миллионов итераций число *пи* найдено:



Вообще говоря, точность вычислений по этим формулам и не может быть высокой из-за того, что тип *double* грубоват для этих целей.

Вычисляем число e

Число ***e*** –основание натуральных логарифмов - равно:

e ≈ 2,7182818284 5904523536 0287471352 6624977572 4709369995 9574966967
6277240766 3035354759 4571382178 5251664274 2746639193 2003059921
8174135966 2904357290 0334295260 5956307381 3232862794 3490763233
8298807531 9525101901 1573834187 9307021540 8914993488 4167509244
7614606680 8226480016 8477411853 7423454424 3710753907 7744992069
5517027618 3860626133 1384583000 7520449338 2656029760 6737113200
7093287091 2744374704 7230696977 2093101416 9283681902 5515108657
4637721112 5238978442 5056953696 7707854499 6996794686 4454905987
9316368892 3009879312 7736178215 4249992295 7635148220 8269895193
6680331825 2886939849 6465105820 9392398294 8879332036 2509443117
3012381970 6841614039 7019837679 3206832823 7646480429 5311802328
7825098194 5581530175 6717361332 0698112509 9618188159 3041690351
5988885193 4580727386 6738589422 8792284998 9208680582 5749279610
4841984443 6346324496 8487560233 6248270419 7862320900 2160990235
3043699418 4914631409 3431738143 6405462531 5209618369 0888707016
7683964243 7814059271 4563549061 3031072085 1038375051 0115747704
1718986106 8739696552 1267154688 9570350354…

Его можно вычислить как сумму бесконечного ряда:

$$e = 1 + 1/1! + 1/2! + 1/3! + \ldots$$

Очень удобная формула для вычисления на компьютере: знаменатель каждой следующей дроби равен знаменателю предыдущей дроби, умноженному на 1, 2, 3, … Из этого следует, что достаточно предыдущую дробь последовательно делить на эти числа и добавлять к общей сумме:

```
//ВЫЧИСЛЯЕМ e
//e = 1 + 1/1! + 1/2! + 1/3! + ...
```

```
procedure CalcE();
begin
   var e: decimal := 1;
   var f: decimal := 1;
   var i := 1;

   for i := 1 to 27 do
   begin
     f := f / i;
     e := e + f;
     Console.WriteLine('i = ' + i + ' e = ' + e.ToString());
   end;
   Console.WriteLine('i = ' + i + ' e = ' + e.ToString());
   Console.WriteLine();
end;
```

Применив тип *decimal*, мы уже через 27 итераций получим 28 правильных десятичных знаков числа ***e*** после запятой:

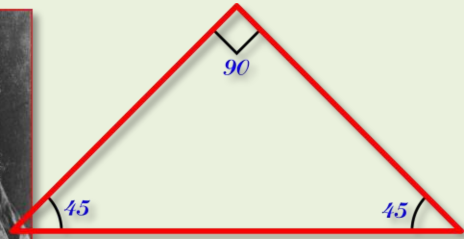
```
Вычисляем пи и е
i = 9 e = 2,7182815255731922398589065256
i = 10 e = 2,7182818011463844797178130512
i = 11 e = 2,7182818261984928651595318263
i = 12 e = 2,7182818282861685639463417242
i = 13 e = 2,7182818284467590023145578702
i = 14 e = 2,7182818284582297479122875949
i = 15 e = 2,7182818284589944642854695765
i = 16 e = 2,7182818284590422590587934503
i = 17 e = 2,7182818284590450705160477958
i = 18 e = 2,7182818284590452267081174817
i = 19 e = 2,7182818284590452349287527283
i = 20 e = 2,7182818284590452353397844906
i = 21 e = 2,7182818284590452353593574317
i = 22 e = 2,7182818284590452353602471108
i = 23 e = 2,7182818284590452353602857925
i = 24 e = 2,7182818284590452353602874042
i = 25 e = 2,7182818284590452353602874687
i = 26 e = 2,7182818284590452353602874712
i = 27 e = 2,7182818284590452353602874713
i = 27 e = 2,7182818284590452353602874713
```

В той же школьной стенной газете было и такое прозаическое правило для запоминания первых цифр числа *e*:

2 – 7 – 1828 – 1828 (год рождения Льва Толстого) – 45 - 90 – 45 (углы прямоугольного равнобедренного треугольника).

Вполне достаточно, чтобы очаровать учительницу математики!

2,7



Эти же формулы используются в книге [100], страницы 24-26 для нахождения чисел пи и е.

К сожалению, в *паскале* нет более точных вещественных типов.

В книге [100] предлагается вычислить *пи* и по такой формуле:

$$\frac{\pi^2}{8} = \frac{1}{1^2} + \frac{1}{3^2} + \frac{1}{5^2} + \ldots + \frac{1}{(2n+1)^2} + \ldots$$

Новую процедуру легко получить из метода Лейбница:

```
//ВЫЧИСЛЯЕМ ПИ
procedure CalcPi2();
begin
   var pi := 0.0;
   var i: integer;

   for i := 0 to 100000000 do
   begin
        var n := (i*2.0+1);
        pi += 1/(n*n);
```

```
    end;
    Console.WriteLine('i = ' + i + ' pi = ' + Math.Sqrt(8*pi));
    Console.WriteLine();
end;
```

Поскольку метод **Sqrt** класса *Math* возвращает значение типа *double*, то мы можем использовать при расчётах только этот тип данных:

```
Вычисляем пи и е
i = 100000000 pi = 3.14159264785158
```

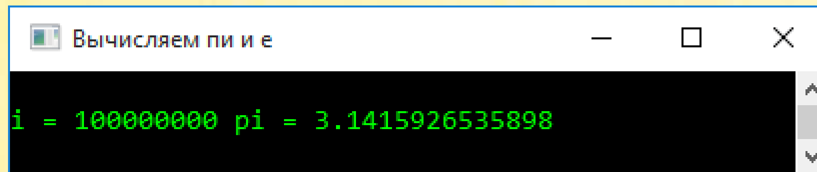
Другая формула для вычисления *пи* из той же книги:

$$\frac{\pi^2}{32} = \frac{1}{1^3} - \frac{1}{3^3} + \frac{1}{5^3} - \frac{1}{7^3} + \ldots + (-1)^{n-1}\frac{1}{(2n-1)^3} + \ldots$$

Здесь мы опять наталкиваемся на ограничение при выборе типа данных – метод **Pow** класса *Math* также возвращает значение типа *double*:

```
//ВЫЧИСЛЯЕМ ПИ
procedure CalcPi3();
begin
    var pi := 0.0;
    var neg := 1.0;
    var i: integer;

    for i := 0 to 100000000 do
    begin
        var n := i*2.0+1;
          pi += neg/n/n/n;
        neg := -neg;
    end;
    Console.WriteLine('i = ' + i + ' pi = ' +
                            Math.Pow(32*pi, 1.0/3.0));
    Console.WriteLine();
```

```
end;
```



В приведённой выше формуле из книги допущена опечатка – степень числа *пи* не 2, а 3, поэтому мы должны извлечь **кубический** корень из суммы ряда.

Следующий ряд состоит как бы из двух рядов, сумму которых можно вычислить отдельно, а затем найти *пи*:

$$\frac{\pi}{4} = 4\left(\frac{1}{5} - \frac{1}{3\cdot 5^3} + \frac{1}{5\cdot 5^5} - \frac{1}{7\cdot 5^7} + \dots\right) - \left(\frac{1}{239} - \frac{1}{3\cdot 239^3} + \frac{1}{5\cdot 239^5} - \dots\right)$$

В целом ряд напоминает ряд Лейбница, поэтому новый метод написать не очень сложно:

```
//ВЫЧИСЛЯЕМ ПИ
procedure CalcPi4();
begin
   var sum1 := 0.0;
   var sum2 := 0.0;
   var neg := 1.0;
   var i := 0;
   var n5 := 5.0;
   var n239 := 239.0;

   for i := 0 to 100000 do
   begin
        sum1 += neg*4/n5/(i*2+1);
        sum2 += neg/n239/(i*2+1);
         neg := -neg;
        n5 *= 25;
```

```
        n239 := n239*239*239;
    end;
    Console.WriteLine('i = ' + i + ' pi = ' + 4*(sum1-sum2));
    Console.WriteLine();
end;
```

```
Вычисляем пи и е
i = 100000 pi = 3.14159265358979
```

Здесь все знаки числа *пи* верные!

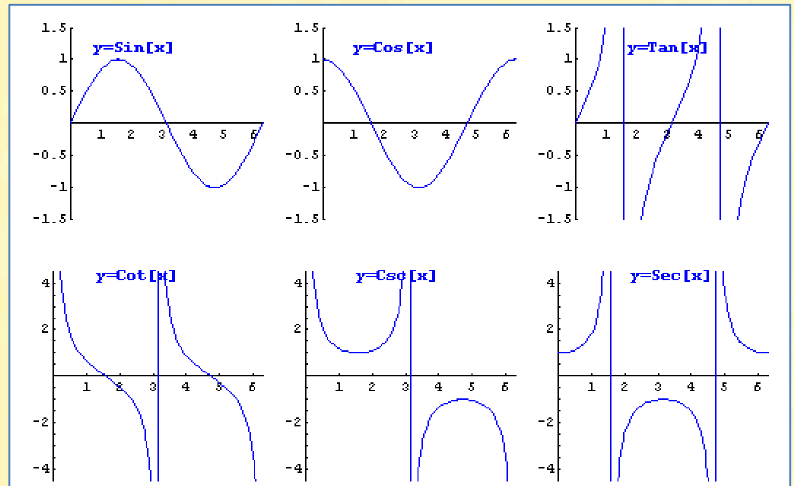
И наконец, вычислите самостоятельно *пи* по такой **формуле**:

$$\frac{2}{\pi}=\sqrt{\frac{1}{2}}\cdot\sqrt{\frac{1}{2}+\frac{1}{2}\sqrt{\frac{1}{2}}}\cdot\sqrt{\frac{1}{2}+\frac{1}{2}\sqrt{\frac{1}{2}+\frac{1}{2}\sqrt{\frac{1}{2}}}}\cdot\ldots$$

Поскольку здесь участвуют квадратные корни, то точность вычислений также будет невысокой.

Тригонометрические функции

Функция *с параметром*
Оператор ***return***
Тип данных ***double***
Оператор деления ***/***
Константа ***Math.PI***
Бесконечный цикл ***repeat-until***
Метод ***ulong.Parse***
Метод ***Math.Sin***
Метод ***Math.Cos***
Метод ***Math.Atan***
Цикл ***for***
Оператор ***return***
Условный оператор ***if***
Комбинированные операторы присваивания



Задача 5☉☉ из книги *100 задач по программированию* [100], страница 26:

Составьте функции для вычисления с указанной точностью значений тригонометрических функций путём сложения членов следующих рядов:

а) $\sin x = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} - \ldots + (-1)^n \frac{x^{2n+1}}{(2n+1)!} + \ldots$

б) $\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots + (-1)^n \frac{x^{2n}}{(2n)!} + \ldots;$

в) $\operatorname{arctg} x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \ldots + (-1)^n \frac{x^{2n+1}}{2n+1} + \ldots$

В последней формуле $|x| < 1$.

Ряд для вычисления **синуса** должен напомнить вам ряд для вычисления основания натуральных логарифмов *e*:

> 1 + 1/1! + 1/2! + 1/3! + …

А поскольку это так, то вам остаётся только учесть, что ряд **знакопеременный**, а в числителе ***x*** каждый раз увеличивается в x^2 раз.

Но сначала нужно решить проблему с вводом угла **x**. Так как пользователю удобнее вводить угол в **градусах**, а формула предпочитает **радианы**, то мы должны позаботиться о переводе градусов в радианы. Вспомним, что полной окружности соответствует угол *360˚*, или *2π* радианов. Чтобы найти, сколько радианов содержится в a градусах, составим несложную пропорцию:

> 2π радианов = 360˚
> x   радианов = a˚

Откуда находим:

> 2π a˚ = 360˚ x →
>
> x радианов = a˚* π /180˚ , или
> x радианов = a˚ /180˚* π

Переводим эту формулу на язык *паскаля* и получаем функцию **Grad2Rad**:

```
function Grad2Rad(a: double): double;
begin
   Result := a / 180 * Math.PI;
end;
```

В **главном блоке** вызываем функцию *Sin* для вычисления заданного угла и сравниваем полученное значение с тем, что выдаёт метод *Sin* класса *Math:*

```
uses
   System;

//Тригонометрические функции

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

    //бесконечный цикл ввода данных -
    //пока пользователь не закроет программу:
    repeat
        Console.Write('Градусы: ');
        var s := Console.ReadLine();
       //угол в градусах:
        var grad: double := uint64.Parse(s);
        //переводим в радианы:
        var rad := Grad2Rad(grad);

        Console.Write('Синус ' + grad + ' = ');
        var sinus := Sin(rad);
        Console.WriteLine(sinus);
        Console.WriteLine('Math.Sin = ' + Math.Sin(rad));
        Console.WriteLine();
    until not (true);

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

Приведённая в книге формула для вычисления синуса – это **ряд Тейлора**.

Так как числитель дробей постоянно и резко уменьшается, а знаменатель очень быстро растёт, то точности типа *double* хватает всего на пару десятков итераций:

```
//ВЫЧИСЛЯЕМ СИНУС ЗАДАННОГО УГЛА
function Sin(x : double): double;
begin
   var sinus := 0.0;
   var xn := x;

   var f := 1.0;
   var n := 0;
   var neg := 1.0;

   for n := 0 to 22 div 2 do
   begin
     var i := 2.0*n;
     if (i > 0) then f := f/i/(i+1);
         sinus += neg*xn*f;
         neg := -neg;
         xn := xn*x*x;
   end;
   Result := sinus;
end;
```

```
Тригонометрические функции

Тригонометрические функции

Градусы: 30
Синус 30 = 0,5
Math.Sin = 0.5

Градусы: 45
Синус 45 = 0,707106781186547
Math.Sin = 0.707106781186547

Градусы: 60
Синус 60 = 0,866025403784439
Math.Sin = 0.866025403784439

Градусы: 90
Синус 90 = 1
Math.Sin = 1

Градусы: 120
Синус 120 = 0,866025403784439
Math.Sin = 0.866025403784439

Градусы: 135
Синус 135 = 0,707106781186548
Math.Sin = 0.707106781186548
```

Запускаем программу и задаём хорошо известные углы в градусах. Рисунок убеждает нас, что наша небольшая функция *Sin* не уступает в точности встроенному методу класса *Math*!

Переходим к вычислению **косинусов**:

```
begin
. . .

        //Console.Write('Синус ' + grad + ' = ');
        //var sinus := Sin(rad);
        //Console.WriteLine(sinus);
        //Console.WriteLine('Math.Sin = ' + Math.Sin(rad));
        //Console.WriteLine();

        Console.Write('Косинус ' + grad + ' = ');
        var cosinus := Cos(rad);
        Console.WriteLine(cosinus);
        Console.WriteLine('Math.Cos = ' + Math.Cos(rad));
        Console.WriteLine();

    until not (true);
   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
end.
```

Чтобы получить функцию **Cos**, нужно в функции для вычисления синуса исправить всего несколько строчек:

```
//ВЫЧИСЛЯЕМ КОСИНУС ЗАДАННОГО УГЛА
function Cos(x : double): double;
begin
   var cosinus := 1.0;
   var xn := x*x;

   var f := 1.0;
   var n : integer;
   var neg := -1.0;

   for n := 1 to 22 div 2 do
   begin
      var i := 2.0*n;
      f := f/i/(i-1);
      cosinus += neg*xn*f;
      neg := -neg;
```

```
        xn := xn*x*x;
    end;
    Result := cosinus;
end;
```

Здесь мы иногда наблюдаем расхождение с методом *Cos* класса *Math* в последнем знаке, но в целом и эту функцию можно признать успешной:

```
Тригонометрические функции

Тригонометрические функции

Градусы: 30
Косинус 30 = 0,866025403784439
Math.Cos = 0.866025403784439

Градусы: 45
Косинус 45 = 0,707106781186547
Math.Cos = 0.707106781186548

Градусы: 60
Косинус 60 = 0,5
Math.Cos = 0.5

Градусы: 90
Косинус 90 = -6,21963119984334E-18
Math.Cos = 6.12303176911189E-17

Градусы: 120
Косинус 120 = -0,5
Math.Cos = -0.5

Градусы: 135
Косинус 135 = -0,707106781186549
Math.Cos = -0.707106781186547
```

Ряд для вычисления **арктангенса** заданного угла мало отличается от ряда для синуса, он даже проще:

```
begin
        . . .

        //Console.Write('Косинус ' + grad + ' = ');
        //var cosinus := Cos(rad);
```

```
        //Console.WriteLine(cosinus);
        //Console.WriteLine('Math.Cos = ' + Math.Cos(rad));
        //Console.WriteLine();

        Console.Write('Арктангенс ' + grad + ' = ');
        var arctan = Atan(rad);
        Console.WriteLine(arctan);
        Console.WriteLine('Math.Atan = ' + Math.Atan(rad));
        Console.WriteLine();

    until not (true);
   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
end.
```

И мы быстро справляемся с новой **функцией**:

```
//ВЫЧИСЛЯЕМ АРКТАНГЕНС ЗАДАННОГО УГЛА
function Atan(x: double) : double;
begin
   var arctan := 0.0;
   var xn := x;

   var n := 0;
   var neg := 1.0;

   for n := 0 to 1000 div 2 do
   begin
      var i := 2.0*n;
        arctan += neg*xn/(i+1);
        neg := -neg;
        xn := xn*x*x;
   end;
   Result := arctan;
end;
```

В данном ряду знаменатели увеличиваются довольно медленно, поэтому для достижения максимально возможной точности потребуется значительно больше итераций.

Также следует учитывать, что ряд правильно работает только для углов, которые меньше 1 радиана:

```
Тригонометрические функции

Тригонометрические функции

Градусы: 15
Арктангенс 15 = 0,256052769980756
Math.Atan = 0.256052769980756

Градусы: 30
Арктангенс 30 = 0,482347907101025
Math.Atan = 0.482347907101025

Градусы: 45
Арктангенс 45 = 0,665773750028354
Math.Atan = 0.665773750028354

Градусы: 50
Арктангенс 50 = 0,717505778101648
Math.Atan = 0.717505778101648
```

Великолепная семёрка

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?

Можно просто выписать на листочке все 1000 чисел и посчитать, сколько раз в них встречается семёрка. Или не выписывать числа, а перебрать их в цикле ***for***, выделить цифры и посчитать семёрки.

Но лучше написать процедуру **Solve** для решения задачи.

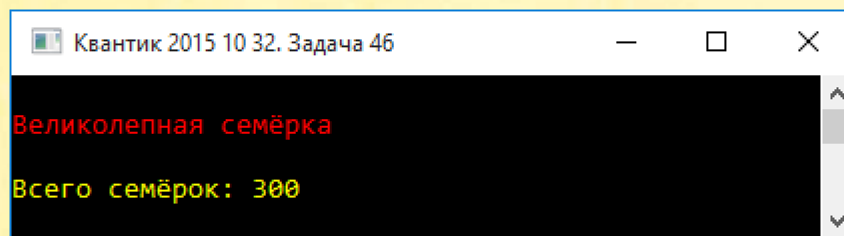
```
uses
   System;

//Квантик 2015 10 32. Задача 46
```

Заметим, что в числе 1000 семёрок нет, значит, нам достаточно проверить числа от 1 до 999. Чтобы не выделять цифры из чисел, мы организуем 3 вложенных цикла ***for***. Текущие значения переменных цикла как раз и дадут нам цифры очередного числа. 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 каждой переменной цикла:

```
//РЕШАЕМ ЗАДАЧУ
procedure Solve();
begin
   Console.ForegroundColor := ConsoleColor.Yellow;

   //число семёрок:
   var n7 := 0;

   for var a := 0 to 9 do
      for var b := 0 to 9 do
         for var c := 0 to 9 do
         begin
            var s := 0;
            s += a = 7 ? 1 : 0;
            s += b = 7 ? 1 : 0;
            s += c = 7 ? 1 : 0;
            n7 += s;
         end;
   Console.WriteLine('Всего семёрок: ' + n7);
   Console.WriteLine();
end;
```

```
begin
   //заголовок окна:
   Console.Title := 'Квантик 2015 10 32. Задача 46';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Великолепная семёрка');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   //решаем задачу:
   Solve();

   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**:



Квантик

Числовые ребусы, или **числобусы** известны каждому любителю головоломок, поэтому мы не будем повторять правила их составления и решения. При необходимости вы можете получить исчерпывающую информацию в журнале *Квантик* №10 за 2013 год, на страницах 18-21.

Приёмы решения числовых ребусов довольно многочисленны, но практически ни один ребус невозможно решить только с их помощью, логическим путём, без *перебора* вариантов. Такой способ решения задач больше годится для компьютера, чем для человека, именно поэтому числобусы лучше решать программно.

В первых двух номерах журнала *Квантик* напечатано по одному числовому ребусу, которые мы сейчас для примера и решим.

Любой числовой ребус легко решается ***методом грубой силы***, то есть простым перебором во вложенных циклах. Действительно, буквами можно зашифровать не более 10 цифр, то есть в худшем случае потребуется перебрать 10! = 3628800 вариантов, что совсем немного для современных компьютеров.

При написании программы важно учитывать, что:

- первая цифра числа не нуль
- разные буквы обозначают разные цифры
- одинаковые буквы обозначают одинаковые цифры

Числовые ребусы очень часто встречаются в занимательной литературе по математике, и вы сможете решить их точно так же, как в этих проектах.

3. В слове КВАНТИК каждую букву заменили некоторой цифрой. Одинаковые буквы (то есть две буквы К) были заменены одинаковыми цифрами, а разные – разными. Оказалось, что выполняется равенство

КВА + Н = ТИК

Найдите произведение цифр числа КВАНТИК.

В этом числовом ребусе всего 6 разных букв – К, В, А, Н, Т, И. Значит, нам потребуется выписать столько же вложенных циклов ***for***, чтобы решить головоломку.

Имена переменных циклов мы обозначим соответствующими латинскими буквами, то есть k, v, a, n, t, i.

С букв **К**, **Н** и **Т** начинаются числа *КВА*, *Н* и *ТИК*, поэтому они не могут быть нулём. Следовательно, переменные циклов **k**, **n**, **t** изменяются в диапазоне *1..9*, а переменные **v**, **a**, **i** – в диапазоне *0..9*.

Значение переменной цикла **v** не может равняться значению переменной **k**, что мы и проверяем во втором цикле ***for***:

```
uses
   System;

//Квантик

//РЕШАЕМ ЗАДАЧУ
procedure Solve();
begin
   Console.ForegroundColor := ConsoleColor.Yellow;

   for var k := 1 to 9 do
      for var v := 0 to 9 do
      begin
         if (v = k) then continue;
```

Если они совпадут, мы тут же перейдём к следующему значению переменной **v**.

Все остальные переменные циклов мы сверяем с текущими значениями переменных внешних по отношению к ним циклов. Если значения хотя бы одной пары переменных совпадают, то переходим к следующему значению проверяемой переменной цикла:

```
for var a := 0 to 9 do
begin
   if ((a = v) or (a = k)) then continue;
   for var n := 1 to 9 do
   begin
      if ((n = a) or (n = v) or (n = k)) then
         continue;
      for var t := 1 to 9 do
      begin
         if ((t = n) or (t = a) or (t = v) or (t = k)) then
```

```
            continue;
         for var i := 0 to 9 do
         begin
            if ((i = t) or (i = n) or (i = a) or
                (i = v) or (t = k)) then
               continue;
```

Сейчас мы получили **разные** значения для переменных всех циклов и должны проверить, удовлетворяют ли они условиям, то есть тому равенству, которое приведено в тексте задачи.

Так как нам известны только цифры, то мы должны составить из них *числа*. Это легко сделать, если учесть, что **k** и **t** – это сотни, **v** и **i** – десятки, **a**, **n** и **k** (в правой части равенства) – единицы:

```
if (k * 100 + v * 10 + a + n = t * 100 + i * 10 + k) then
```

Если в результате вычислений мы получили верное равенство, то печатаем ответ на экране:

```
                     begin
                        Console.WriteLine('kva + n = tik {0} +
                                          {1} = {2}
                                          Произведение = {3}',
                                          k * 100 + v * 10 + a,
                                          n, t * 100 + i * 10 + k,
                                          k * v * a * n * t * i * k);
                     end
                  end;
               end;
            end;
         end;
      end;
   Console.WriteLine();
end;
```

В **главной части** программы мы оформляем окно приложения надлежащим образом и вызываем процедуру *Solve* для непосредственного решения задачи:

```
begin
   //заголовок окна:
   Console.Title := 'Журнал Квантик № 2012, с. 35 3';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Квантик');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   Solve();

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

Как вы видите на рисунке, сам числовой ребус имеет **несколько** решений, а «хитрость» задачи заключается в том, что произведение цифр равняется **нулю**, поскольку одна из цифр в правой части равенства – всегда нуль.

```
Журнал Квантик № 2012, с. 35 3

Квантик

kva + n = tik 193 + 8 = 201  Произведение = 0
kva + n = tik 194 + 7 = 201  Произведение = 0
kva + n = tik 195 + 6 = 201  Произведение = 0
kva + n = tik 196 + 5 = 201  Произведение = 0
kva + n = tik 197 + 4 = 201  Произведение = 0
kva + n = tik 198 + 3 = 201  Произведение = 0
kva + n = tik 294 + 8 = 302  Произведение = 0
kva + n = tik 295 + 7 = 302  Произведение = 0
kva + n = tik 297 + 5 = 302  Произведение = 0
kva + n = tik 298 + 4 = 302  Произведение = 0
kva + n = tik 395 + 8 = 403  Произведение = 0
kva + n = tik 396 + 7 = 403  Произведение = 0
kva + n = tik 397 + 6 = 403  Произведение = 0
kva + n = tik 398 + 5 = 403  Произведение = 0
kva + n = tik 496 + 8 = 504  Произведение = 0
kva + n = tik 498 + 6 = 504  Произведение = 0
kva + n = tik 597 + 8 = 605  Произведение = 0
kva + n = tik 598 + 7 = 605  Произведение = 0
```

Смогли бы вы узнать это без перебора? – Вот решение из журнала *Квантик*:

3. Прибавив к трёхзначному числу **КВА** цифру **Н**, мы получаем трёхзначное число с другой старшей цифрой. Значит, **КВА** и **ТИК** близки к числу, оканчивающемуся на два нуля – а именно, к числу **Т00**, – через которое мы перескочили, прибавив **Н**. Но цифра **Н** не больше 9. Тогда **ТИК** превышает число **Т00** меньше чем на 9, откуда **И** = 0. Значит, произведение цифр числа **КВАНТИК** тоже равно 0.

Данный числобус можно решить и почти без перебора, если хорошенько догадаться, но такие ребусы встречаются нечасто. Например, следующая головоломка – типичный пример числового ребуса с обширным перебором вариантов.

Квантик 2

8. В слове КВАНТИК каждую букву заменили некоторой цифрой. Причём одинаковые буквы (то есть две буквы К) были заменены одинаковыми цифрами, а разные – разными.

При этом оказалось, что выполняется следующее равенство:

$$КВ^{А} = НТИК$$

Чему равно число КВАНТИК?

Вторую задачу можно решить абсолютно так же, как и первую, но это неинтересно!

Вложенные циклы образно можно показать так:

Давайте заменим обычные циклы *for* циклами ***foreach*** для *последовательностей*.

Тогда для каждой переменной цикла **k**, **v**, **a**, **n**, **t**, **i** мы сначала должны создать последовательности в заданном диапазоне, то есть для начальных букв *1..9*, а для все остальных – *0..9*.

> Последовательности можно создать и непосредственно в заголовке цикла *foreach*:
>
> ```
> foreach var k in Range(1, 9) do
> ```

> Показатель степени **A** не может быть меньше двух, поэтому для переменной цикла **a** выбран диапазон *2..9*.

```
uses
   System;

//Квантик 2

//РЕШАЕМ ЗАДАЧУ
procedure Solve();
begin
   Console.ForegroundColor := ConsoleColor.Yellow;
   var kr := Range(1, 9);
   var vr := Range(0, 9);
   var ar := Range(2, 9);
   var nr := Range(1, 9);
   var tr := Range(0, 9);
   var ir := Range(0, 9);
```

Второе «нововведение» относится к проверке цифр на неповторяемость. Для этого мы приспособим множество цифр **digits**:

```
   var digits: set of 0..9;
```

Первую цифру мы сразу помещаем в множество:

```
foreach var k in kr do
begin
   digits += [k];
```

Все остальные претенденты на попадание в множество сначала должны пройти строгую проверку со стороны метода *Contains*, который возвращает *true*, если новая цифра уже имеется в множестве. Тогда мы переходим к проверке следующей цифры:

```
   foreach var v in vr do
   begin
      if (digits.Contains(v)) then
         continue;
      digits += [v];
      foreach var a in ar do
      begin
         if (digits.Contains(a)) then
            continue;
         digits += [a];
         foreach var n in nr do
         begin
            if (digits.Contains(n)) then
               continue;
            digits += [n];
            foreach var t in tr do
            begin
               if (digits.Contains(t)) then
                  continue;
               digits += [t];
               foreach var i in ir do
               begin
                  if (digits.Contains(i)) then
                     continue;
```

Наполнив множество цифрами, мы проверяем, выполняется ли требуемое равенство. Для удобства отдельно вычисляем значения его *левой* и *правой* частей, а затем сравниваем их между собой:

```
var left := Math.Pow(k * 10 + v, a);
var right := n * 1000 + t * 100 + i * 10 + k;
```

Если окажется, что они равны, то мы печатаем ответ на экране:

```
    if (left = right) then
    begin
       Console.WriteLine('kv|a = ntik {0}|{1} = {2}',
                         k * 10 + v, a, n * 1000 + t *
                         100 + i * 10 + k);
       Console.WriteLine('Число КВАНТИК = {0}',
                         k * 1000000 + v * 100000 +
                         a * 10000 + n * 1000 + t * 100 +
                         i * 10 + k);
    end
 end;
```

В конце каждого цикла необходимо **удалить** текущее значение соответствующей переменной, иначе в множестве останутся их предыдущие значения, что приведёт к неверному решению:

```
                 digits -= [t];
              end;
              digits -= [n];
           end;
           digits -= [a];
        end;
        digits -= [v];
     end;
     digits -= [k];
  end;

  Console.WriteLine();
end;
```

Главный блок программы – традиционный: настройка окна приложения и вызов процедуры *Solve*:

```
begin
   //заголовок окна:
   Console.Title := 'Журнал Квантик 2012 №2, с.35 8';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Квантик 2');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   Solve();

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

Решение у задачи – *единственное:*

Журнал Квантик 2012 №2, с.35 8

```
Квантик 2

kv|a = ntik 93|2 = 8649
Число КВАНТИК = 9328649
```

А теперь сравните наше «изящное» решение с журнальным, которое откровенно озадачивает:

8. Заметим сразу, что $К \neq 0$, так как число КВ начинается на К, а число не может начинаться с нуля. Но тогда и $В \neq 0$, так как при возведении в степень круглого числа получается тоже круглое число, а число НТИК оканчивается на К, и $К \neq 0$.

Так как двузначное число КВ, возведенное в степень А, равно четырехзначному числу НТИК, то А не меньше 2 (очевидно) и не больше 3 (ведь даже самое маленькое двузначное число 10, возведённое в 4-ю степень, даст уже пятизначное число 10000).

Значит, А – это либо 2, либо 3. Проверим оба эти варианта.

1. Если $А = 3$, то $К = 1$ или $К = 2$ ($К < 3$, так как 30^3 уже пятизначное число).

Пусть $К = 1$. Тогда $В \neq 1$, но легко проверить, что никакая другая цифра, возведённая в третью степень, не даст число, оканчивающееся на 1. Значит, этот случай невозможен.

Пусть $К = 2$. Тогда $В = 1$ (так как $В \neq 0$, а 22^3 даёт в результате уже пятизначное число). Но

при возведении в любую степень число, оканчивающееся на 1, даёт в результате число, оканчивающееся на 1, а $К \neq 1$. Противоречие.

2. Пусть $А = 2$. Тогда:

$К \neq 1$, так как даже 19^3 – всего лишь трёхзначное число (361), и $К \neq 2$, так как $А = 2$.

Далее, $К \neq 3$, $К \neq 7$ и $К \neq 8$, так как ни одна цифра при возведении в квадрат не даст числа, оканчивающегося на 3, 7 или 8. Также $К \neq 5$, так как тогда и $В = 5$ (чтобы $В^2$ оканчивалось на 5), а В и К – разные цифры.

Проверим вариант $К = 4$. В этом случае В – либо 2, либо 8 (иначе $В^2$ не будет оканчиваться на 4). Но $В \neq 2$, так как $А = 2$, и, значит, $В = 8$. Но $48^2 = 2304$, то есть $Н = 2$, а это невозможно, так как $А = 2$. Следовательно, $К \neq 4$.

Проверим вариант $К = 6$. В этом случае В – либо 4, либо 6 (иначе $В^2$ не будет оканчиваться на 6). Но $В \neq 6$, так как $К = 6$. Значит, $В = 4$. Но $64^2 = 4096$, то есть $Н = 4$, что невозможно, так как тогда $Н = В$. Следовательно, $К \neq 6$.

Осталась единственная возможность: $К = 9$. Тогда В – либо 3, либо 7. Проверим вариант $В = 3$: тогда $93^2 = 8649$, что соответствует условию задачи. Если $В = 7$, то $97^2 = 9409$, откуда $К = Н$, что противоречит условию.

Итак, окончательный и единственный ответ: КВАНТИК = 9328649.

Числобус

В журнале *Квант* большинство числобусов не относится к классическим, поэтому к ним нужно искать другие подходы. Но некоторые, с виду «нестандартные» числобусы, можно решить без особых затей.

В шестом номере за 1970 год была напечатана такая **задача**:

1218. В этой задаче на деление бук-

вами зашифрованы цифры. Одинаковые буквы означают одинаковые цифры. Попробуйте расшифровать пример:

```
_ABCD : CD = BCD
  CD
  ---
 _EC
  DF
  ---
 _BCD
  BCD
  ---
    0
```

Здесь, кроме делимого, делителя и частного, даны и промежуточные произведения. Они облегчают ручное решение головоломки, но нам они не нужны, так что мы будем решать сокращённый вариант задачи:

ABCD : CD = BCD

Как видите, от всей задачи осталась только первая строка, что, конечно, ничуть не изменило её сути.

Главный блок программы нужен нам исключительно для украшательства окна приложения и вызова процедуры *Solve*:

```
begin
   //заголовок окна:
   Console.Title := 'Журнал Квант 1970 №6, с.54';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Квант');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   Solve();

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

Давайте заменим длинное ключевое слово *integer* коротким словом *int*, которое используется в *Си-шарпе* для того же типа целых чисел. При этом мы перекроем одноимённую функцию **Int**, возвращающую целую часть вещественного числа, но в данном проекте она нам всё равно не понадобится.

Аналогично можно «переименовать» и ключевое слово *boolean*, которое, впрочем, не встречается в исходном коде нашей программы:

```
uses
   System;

type
   int = integer;
   //bool = boolean;

//Квант
```

Добравшись до процедуры **Solve**, мы тут же обзаводимся *последовательностями*, которые хранят все возможные значения для букв *A*, *B*, *C*, *D* числобуса и для переменных соответствующих циклов:

```
//РЕШАЕМ ЗАДАЧУ
procedure Solve();
begin
   Console.ForegroundColor := ConsoleColor.Yellow;
   var ar := Range(1, 9);
   var br := Range(1, 9);
   var cr := Range(1, 9);
   var dr := Range(0, 9);
```

Здесь мы учли, что буквы *A*, *B* и *C* **не могут заменять нуль**, поскольку с них начинаются числа.

Сила и новизна рассматриваемого способа решения числобусов заключена в множестве **digits**:

```
//множество использованных цифр:
var digits := new HashSet<int>();
```

Сразу после создания множество пустое, то есть не содержит ни одной цифры, что и понятно: мы ещё не приступили к решению задачи.

В первом, внешнем цикле переменная **a** последовательно получает значения из последовательности *ar*: 1, 2, ... , 9. Так как использованная цифра не должна повторяться, то мы помещаем её в множество *digits*:

```
foreach var a in ar do
begin
   digits.Add(a);
```

Теперь в нём находится цифра (однозначное число), которой мы заменили букву **А** в числобусе.

Буква **В** получает свои значения точно так же, но во вложенном цикле:

```
   foreach var b in br.Except(digits) do
   begin
      digits.Add(b);
```

Но если мы будем брать значения для переменной цикла **b** из последовательности *br*, то рано или поздно попадётся цифра, которая уже использована нами для буквы *А*. Чтобы этого избежать, мы удаляем из последовательности *br* цифру для буквы *А*. В результате выполнения операции:

```
br.Except(digits)
```

сама последовательность *br* не изменится, а в цикле будет использована её копия за вычетом использованной ранее цифры.

Так как в новой, укороченной последовательности заведомо нет использованной цифры, то нам нет нужды каждый раз сверять значение переменной *b* со значением переменной *a*. Код становится простым и изящным!

Но у нас осталось ещё 2 цикла – действуем в них аналогично:

```
foreach var c in cr.Except(digits) do
begin
   digits.Add(c);
   foreach var d in dr.Except(digits) do
   begin
```

В самом последнем цикле добавлять цифру в множество *digits* не нужно, так как она повториться уже никак не сможет.

В данной задаче всего 4 буквы и столько же циклов. Но если бы их было больше, то мы легко добавили бы и другие циклы.

Теперь мы получили значения для всех букв числобуса и должны проверить, выполняются ли условия задачи, то есть равна ли левая часть равенства правой:

```
var left := a * 1000 + b * 100 + c * 10 + d;
var right := (b * 100 + c * 10 + d) * (c * 10 + d);
```

Если это так, то мы печатаем решение на экране:

```
if (left = right) then
begin
   Console.WriteLine('ABCD : CD = BCD {0}:{1} = {2}', left,
                     b * 100 + c * 10 + d, c * 10 + d);
      end;
end;
```

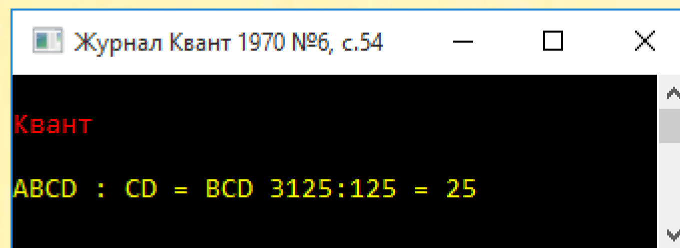
Если бы мы наверняка знали, что решение единственное, то дальнейший перебор нам бы не понадобился. Но поскольку сомнения остаются, то мы доводим перебор до конца.

Следующие действия нужно выполнить и в том случае, если значения букв найдены, но равенства не получилось:

```
           digits.Remove(c);
         end;
         digits.Remove(b);
      end;
      digits.Remove(a);
   end;

   Console.WriteLine();
end;
```

В конце каждого цикла мы удаляем из множества *digits* значение соответствующей переменной, иначе в нём быстро окажутся все однозначные числа.

Несмотря на наше усердное старание, мы нашли **единственное решение** задачи:



Пора по парам

В июньском номере журнала *Квантик* за 2015 год предлагается решить такую задачу:

> **Задача 1.** Найдите слово (имя существительное, нарицательное, в единственном числе), в котором имеются две одинаковые буквы, сразу вслед за которыми идут две другие одинаковые буквы.

Её условие сразу адресует нас к названию одной из лучших современных команд КВН *Пора по парам:*

Впрочем, это каламбурное сочетание слов можно встретить и на театральных афишах, и в объявлениях:

Я вспомнил только одно слово - *геенна*. В словаре Ожегова оно присутствует, но известно ли оно «любознательным школьникам», для которых и предназначается эта задача?

Я решил поискать и другие слова, но уже **компьютерным** способом.

Понятно, что программа для решения этой задачи несложная, поэтому её можно написать за несколько минут.

В **главной части** программы мы выбираем *файл со списком слов* и вызываем процедуру *AABB*:

```
uses
   System, System.IO;

//Квантик 2015 06 27 1

begin
   //заголовок окна:
   Console.Title := 'Квантик 2015 06 27 1';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Пора по парам');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   //словарь:
   var fileName := 'makarov_frc.txt';
   //ищем слова:
   AABB(fileName);
   //var fileName := 'EnDictionary_frc.txt';
   //ищем слова:
   //AABBCC(fileName);

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

В этой процедуре мы загружаем все слова в строковый массив **spisok**:

```
//РЕШАЕМ ЗАДАЧУ
procedure AABB(fn: string);
begin
   Console.ForegroundColor := ConsoleColor.Yellow;

   //нашли слов:
   var nWords := 0;
   //загружаем словарь:
   var spisok := System.IO.File.ReadAllLines(fn);
```

А затем проверяем все слова в цикле ***foreach***:

```
   //просматриваем все слова:
   foreach var wrd in spisok do
   begin
```

Метод **Match** класса регулярных выражений *Regex* устанавливает поле *Success* в *true*, если найдено совпадение с заданным условием. Для нашего случая годится простое выражение:

'(.)\1'

которое срабатывает, если в очередном слове *wrd* имеются две одинаковые смежные буквы:

```
      var letters := String.Empty;

      var match := Regex.Match(wrd, '(.)\1');
      if (match.Success) then
      begin
```

Однако нам нужна не одна, а **2 пары** одинаковых букв, поэтому мы ищем следующее совпадение:

```
         //пара одинаковых букв:
         letters += match.Value;
         //ищем следующую пару:
         match := match.NextMatch();
```

Но так мы найдём слова, которые содержат не менее двух пар одинаковых букв, но они не обязательно будут следовать друг за другом:

Среди найденных слов безусловно имеются и те, что нам нужны. Например, на рисунке вы можете видеть «моё» слово *геенна*, а также слова *вакуумметр* и *массообмен*, которые также подходят под условие задачи.

> Иногда и такие слова вам могут пригодиться.

Всего в списке 100 слов, которые можно просмотреть и вручную, однако несколько дополнительных строк в программе избавят нас и от этой необходимости.

Мы предусмотрительно записали в строковую переменную *letters* найденные парные буквы. На рисунке выше они напечатаны вслед за словами. Например, для тестового слова *геенна* это буквы *ЕЕНН*. Если в проверяемом слове пары одинаковых букв стоят рядом, то с помощью строкового метода **Contains** мы легко обнаружим эту «группировку»:

```
         if (match.Success) then
         begin
            //вторая пара одинаковых букв:
            letters += match.Value;
            //пары букв должны стоять рядом:
            if (wrd.Contains(letters)) then //закомментировать
            begin
               Console.WriteLine(wrd + ' ' + letters);
               nWords += 1;
            end
         end
      end
   end;
   Console.WriteLine('Всего найдено слов ' + nWords);
   Console.WriteLine();
end;
```

Если между парами одинаковых букв имеются другие буквы, то этот метод слово проигнорирует.

Всего мне удалось обнаружить **13 слов**, удовлетворяющих условию задачи:

```
Квантик 2015 06 27 1

Пора по парам

ГЕЕННА ЕЕНН
ВАКУУММЕТР УУММ
МАССООБМЕН ССОО
ВАКУУММЕТРИЯ УУММ
МЕТАЛЛООПТИКА ЛЛОО
ВОЕННООБЯЗАННАЯ ННОО
ВОЕННООБЯЗАННЫЙ ННОО
КРИСТАЛЛООПТИКА ЛЛОО
МЕТАЛЛООБРАБОТКА ЛЛОО
ПАРААГГЛЮТИНАЦИЯ ААГГ
ТЕЛЕГАММААППАРАТ ММАА
МИЦЕЛЛООБРАЗОВАНИЕ ЛЛОО
КРИСТАЛЛООБРАЗОВАНИЕ ЛЛОО
Всего найдено слов 13
```

Рис

Какое именно из этих слов имел в виду автор задачи, мне неизвестно, но наиболее «детское» из этих слов – *военнообязанный*.

Если внимательно приглядеться к найденным словам, то можно найти единственное слово, в котором идут подряд **3 пары** одинаковых букв. Это причудливое слово *телеграммаапапарат*, которое начинается со слова *телега*, так что не каждый прочитает его правильно с первого раза.

Всегда интересно покопаться и в *иноязычных* словарях – на предмет сравнения их с русским языком.

В **английском языке** множество слов с двумя парами букв, поэтому мы напишем метод **AABBCC**, который отыщет нам слова только с *тремя* смежными парами. Он мало отличается от более простого метода, рассмотренного нами выше, поэтому мы не будем останавливать на нём наше внимание:

```
procedure AABBCC(fn: string);
begin
   Console.Clear();
   Console.ForegroundColor := ConsoleColor.Green;

   //нашли слов:
   var nWords := 0;
   //загружаем словарь:
   var spisok := System.IO.File.ReadAllLines(fn);
   //просматриваем все слова:
   foreach var wrd in spisok do
   begin
      var match := Regex.Match(wrd, '(.)\1');
      var letters := String.Empty;
      if (match.Success) then
      begin
         //пара одинаковых букв:
         letters += match.Value;
         //ищем следующую пару:
         match := match.NextMatch();
         if (match.Success) then
         begin
            //вторая пара одинаковых букв:
            letters += match.Value;
            match := match.NextMatch();
```

```
            if (match.Success) then
            begin
               //вторая пара одинаковых букв:
               letters += match.Value;
               //пары букв должны стоять рядом:
               if (wrd.Contains(letters)) then
               begin
                  Console.WriteLine(wrd + ' ' + letters);
                  nWords += 1;
               end
            end
         end
      end
   end;
   Console.WriteLine('Всего найдено слов ' + nWords);
   Console.WriteLine();
end;
```

А сразу перейдём к результатам эксперимента.

В английском языке нашлось всего 3 однокоренных слова с тремя смежными парами букв:

```
Квантик 2015 06 27 1
BOOKKEEPER OOKKEE
BOOKKEEPERS OOKKEE
BOOKKEEPING OOKKEE
Всего найдено слов 3
```

Возможно, нас больше порадовали бы эстонские и финские *слооваарии*, но я таковых не имею под рукой…

Упрощаем

Любителям программирования такой способ решения простой, по сути, задачи может показаться несколько вычурным. Предыдущий проект хорош для изучения возможностей платформы *.NET*, но задачу можно решить и с помощью простого **цикла**.

Для начала сделаем список слов *глобальной* переменной, чтобы загружать файл в главном блоке программы, что более естественно, а использовать его в процедуре *Solve*:

```
uses
   System, System.IO;

type
   int = integer;

//Квантик 2015 06 27 1

//загружаем словарь:
var
   spisok: array of string;
```

При запуске программы в список загружаются русские или английские слова по вашему выбору:

```
begin
   //заголовок окна:
   Console.Title := 'Квантик 2015 06 27 1';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Двойные буквы');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   //словарь:
   var fileName := 'makarov_frc.txt';
   spisok := System.IO.File.ReadAllLines(fileName);
   //ищем слова:
   Solve(2);
   //var fileName := 'EnDictionary_frc.txt';
   //spisok := System.IO.File.ReadAllLines(fileName);
   //ищем слова:
   //Solve(3);

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

```
end.
```

Теперь список доступен в процедуре *Solve*, и мы можем приступить к поиску слов. Обратите внимание, что мы передаём этой процедуре число пар смежных букв, которые она должна найти в словах.

Понятно, что это число не должно быть меньше единицы. Тогда при значении параметра **np**, равном **1**, процедура напечатает слова с одной парой двойных букв (или более). При значении **2** – слова с двумя парами букв (или более). И наконец, при значении, **большем 2**, – слова, в которых не менее трёх пар двойных букв.

Легко посчитать, что в словах, которые содержат *np* пар двойных букв, должно быть не меньше *2 * np* букв. Если это не так, то такие слова мы сразу же, без проверки пропускаем:

```
//РЕШАЕМ ЗАДАЧУ
procedure Solve(np: int);
begin
   Console.ForegroundColor := ConsoleColor.Yellow;

  if (np < 1) then
      exit;
   //нашли слов:
   var nWords := 0;

   //просматриваем все слова:
   foreach var wrd in spisok do
   begin
      //Длина слова:
      var len := wrd.Length;
      //пропускаем короткие слова:
      if (len < 2 * np) then
         continue;
```

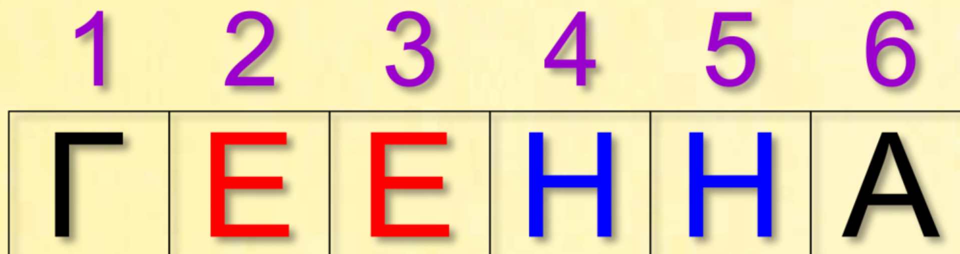
Все остальные слова мы **проверяем** так.

Начинаем с самой первой буквы слова, а заканчиваем буквой, которая занимает позицию *2 * np* от конца слова:

```
for var i := 1 to len - 2 * np + 1 do
begin
```

Действительно, нужных нам букв в слове должно быть *2 * np* штук, поэтому, если мы начнём проверку с последующих букв слова, то нам их заведомо не хватит:

np = 2
len = 6



По условию задачи, первая проверяемая буква должна совпадать со второй. Если же они различаются, то дальше проверять слово смысла нет, и мы переходим к проверке следующего слова:

```
    //совпадает первая пара букв?
    if (wrd[i] <> wrd[i + 1]) then
       continue;
```

Аналогично мы проверяем (если нужно) вторую и третью пару букв, которые должны следовать сразу друг за другом:

```
    //совпадает вторая пара букв?
```

```
        if ((np = 2) and (wrd[i + 2] <> wrd[i + 3])) then
           continue;

        //совпадает третья пара букв?
        if ((np > 2) and (wrd[i + 4] <> wrd[i + 5])) then
           continue;
```

Таким образом, начиная с первой буквы слова (её индекс в слове равен 1), мы проверяем буквы с такими *индексами*:

12 34 56

Если все пары букв совпали, мы печатаем слово на экране:

```
        Console.WriteLine(wrd);
        nWords += 1;
```

Если мы ищем, например, слова с одной парой букв, то слова с двумя парами двойных букв будут напечатаны дважды. Нам этого не нужно, поэтому после печати слова мы сразу же переходим к следующему слову в списке:

```
        //избегаем слов с большим число пар двойных букв:
        break;
     end
```

Если в слове больше чем *2 * np* букв, а нужных нам пар букв мы так и не нашли, то переходим ко второй букве (её индекс в слове равен 2) и снова выполняем все описанные выше проверки.

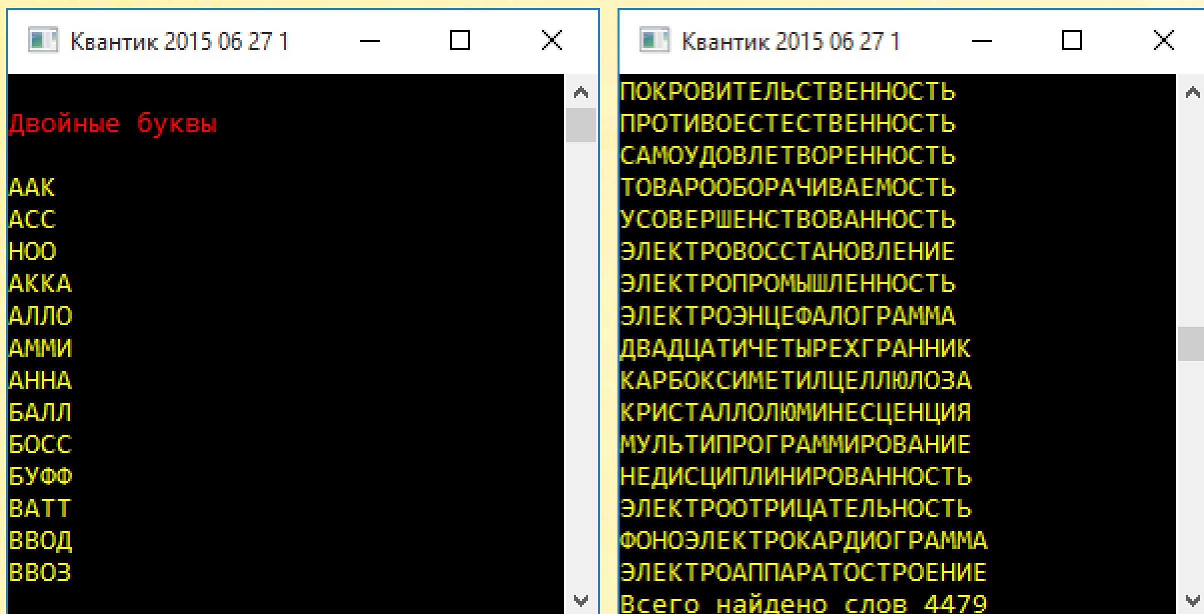
Когда позиция первой проверяемой буквы превысит значение *len - 2 * np + 1*, проверку слова необходимо закончить.

В конце процедуры *Solve* мы печатаем число найденных слов:

```
  end;
   Console.WriteLine('Всего найдено слов ' + nWords);
```

```
   Console.WriteLine();
end;
```

Так как список слов для двух и трёх пар двойных букв мы уже составили, то давайте найдём слова с одной парой двойных букв. Таких слов огромное множество:



Если вы хотите, чтобы ваш код выглядел более «профессионально», то сразу отберите слова с нужным числом букв, тогда вам не придётся проверять их длину в цикле ***foreach***:

```
spisok := spisok.Where(w -> w.Length >= 2 * np).ToArray();
//просматриваем все слова:
foreach var wrd in spisok do
begin
   //Длина слова:
   var len := wrd.Length;
   //пропускаем короткие слова:
   //if (len < 2 * np) then
   //   continue;
```

Геенна

Эту задачу мы уже решили двумя способами - с помощью регулярных выражений и в цикле *for*. Второй способ оказался простым и понятным, а первый слишком заумным. Однако если использовать регулярные выражения более эффективно, то решение получится совсем не страшным!

В **главном блоке** программы мы указываем ей, какой словарь мы хотим использовать для поиска слов, после чего последовательно вызываем процедуры *АА* и *ААА*:

```
uses
   System, System.IO;

//Квантик 2015 06 27 1

begin
   //заголовок окна:
   Console.Title := 'Квантик 2015 06 27 1';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Геенна');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   //словарь:
   var fileName := 'makarov_frc.txt';
   //ищем слова:
   AA(fileName);
   //var fileName := 'EnDictionary_frc.txt';
   //ищем слова:
   AAA(fileName);

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

Первая отыщет слова с двумя парами одинаковых букв, а вторая – с тремя.

Начнём с первой процедуры.

После выполнения этого кода в массиве **spisok** окажутся все слова из указанного файла *fn*:

```
//РЕШАЕМ ЗАДАЧУ
procedure AA(fn: string);
begin
   Console.ForegroundColor := ConsoleColor.Yellow;

   //нашли слов:
   var nWords := 0;
   //загружаем словарь:
   var spisok := System.IO.File.ReadAllLines(fn);
   //spisok.Println();
```

Теперь мы просматриваем весь список и отбираем слова, в которых имеются 2 пары одинаковых букв, идущих подряд.

Обратите внимание, что для этого нам потребовалась всего **одна** строка с регулярным выражением:

```
   //просматриваем все слова:
   foreach var wrd in spisok do
   begin
      var match := Regex.Match(wrd, '(.)\1(.)\2');
      if (match.Success) then
      begin
         Console.WriteLine(wrd);
         nWords += 1;
      end
   end;
   Console.WriteLine('Всего найдено слов ' + nWords);
   Console.WriteLine();
end;
```

Нетрудно догадаться, как можно найти слова с **тремя парами** одинаковых букв:

```
procedure AAA(fn: string);
begin
   Console.ForegroundColor := ConsoleColor.Yellow;

   //нашли слов:
   var nWords := 0;
   //загружаем словарь:
   var spisok := System.IO.File.ReadAllLines(fn);

   //просматриваем все слова:
   foreach var wrd in spisok do
   begin
      var match := Regex.Match(wrd, '(.)\1(.)\2(.)\3');
      if (match.Success) then
      begin
         Console.WriteLine(wrd);
         nWords += 1;
      end
   end;
   Console.WriteLine('Всего найдено слов ' + nWords);
   Console.WriteLine();
end;
```

Если бы в русском языке были бы слова с большей кратностью, мы бы их нашли, просто наращивая длину строки в методе Match класса *Regex*. Причём это можно сделать в цикле, и тогда метод решения задачи будет универсальным!

Наш элегантный регулярный способ поиска слов отработал на славу:

```
Квантик 2015 06 27 1

Геенна

ГЕЕННА
ВАКУУММЕТР
МАССООБМЕН
ВАКУУММЕТРИЯ
МЕТАЛЛООПТИКА
ВОЕННООБЯЗАННАЯ
ВОЕННООБЯЗАННЫЙ
КРИСТАЛЛООПТИКА
МЕТАЛЛООБРАБОТКА
ПАРААГГЛЮТИНАЦИЯ
ТЕЛЕГАММААППАРАТ
МИЦЕЛЛООБРАЗОВАНИЕ
КРИСТАЛЛООБРАЗОВАНИЕ
Всего найдено слов 13

ТЕЛЕГАММААППАРАТ
Всего найдено слов 1
```

Боливия

В том же номере журнала *Квантик* и в том же конкурсе по русскому языку даётся **следующая задача**:

Задача 2. Из названия страны *Боливия* можно, отбросив две первые буквы, получить название другой страны – *Ливия*.

Из названия жителя одной европейской столицы можно таким же способом получить название жителя другой европейской столицы. Что это за столицы?

И.Б.Иткин

Ответ: **Парижанин – рижанин**, а столицы – **Париж** и **Рига**.

Согласитесь, было бы весьма любопытно найти и другие слова, о которых шла речь в задачке про *Боливию-Ливию* и *Парижанина-рижанина*.

Имея под рукой список слов и компьютер, мы легко удовлетворим своё любопытство!

Костяк программы, как и подобает настоящим программистам, мы копируем из нашего предыдущего проекта:

```
uses
   System, System.IO;

//БоЛивия

type
   int = integer;

//загружаем словарь:
var
   spisok: array of string;
```

```
begin
   //заголовок окна:
   Console.Title := 'Квантик 2015 06 27 2';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('БоЛивия');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   //словарь:
   var fileName := 'makarov_frc.txt';
   spisok := System.IO.File.ReadAllLines(fileName);
   //ищем слова:
   Solve(4);
   //var fileName := 'EnDictionary_frc.txt';
   //spisok := System.IO.File.ReadAllLines(fileName);
   //ищем слова:
   //Solve(4);

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

Если в главном блоке программы изменения носят чисто косметический характер, то в процедуре **Solve** нам уже придётся слегка пораскинуть мозгами. Начнём с положительных примеров.

В слове *БоЛивия* более короткое слово *Ливия* целиком расположено в самом конце длинного слова:

Боливия → Бо**Ливия**

Пусть длина более короткого слова равна **nright**. Тогда в длинном слове должно быть хотя бы на одну букву больше. Например, если мы хотим найти слова, заканчивающиеся на слово *Ливия*, в котором 5 букв, то должны просматривать слова, которые не короче 6 букв:

```
//РЕШАЕМ ЗАДАЧУ
procedure Solve(nright: int);
```

```
begin
   Console.ForegroundColor := ConsoleColor.Yellow;

   if (nright < 1) then
      exit;
   //нашли слов:
   var nWords := 0;

   //просматриваем все слова:
   foreach var wrd in spisok do
   begin
      //Длина слова:
      var len := wrd.Length;
      //if (len > 6) then exit;
      //пропускаем короткие слова:
      if (len < nright + 1) then
         continue;
```

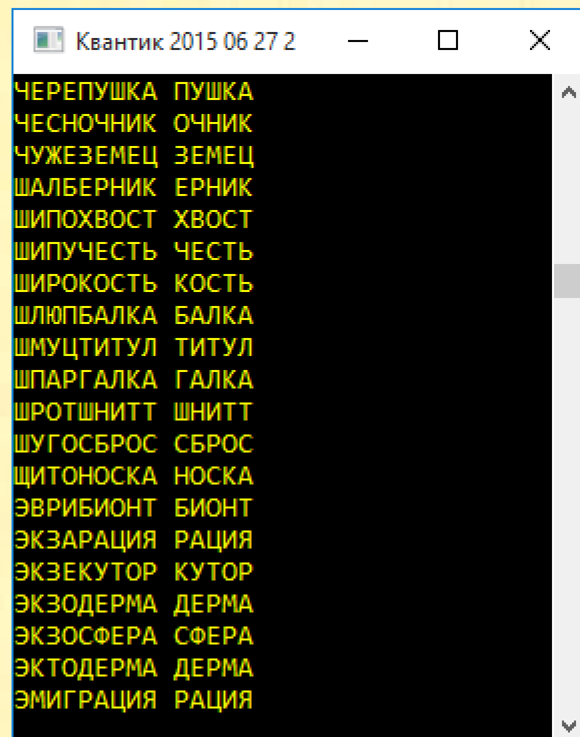
Найдя в списке слово подходящей длины, мы вырезаем из него «хвостик» заданной длины:

```
      var send := wrd.Substring(len-nright);
```

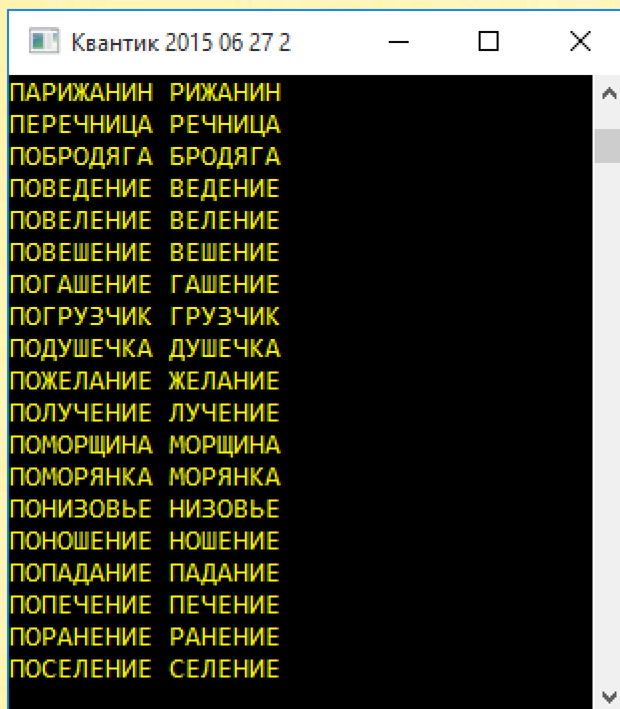
Это совсем необязательно окажется осмысленное слово, поэтому «вырезку» мы должны найти в нашем списке слов. Если она там присутствует, то мы печатаем длинное и короткое слово на экране:

```
      if (spisok.Contains(send)) then
      begin
         Console.WriteLine(wrd + ' ' + send);
         nWords += 1;
      end

   end;
   Console.WriteLine('Всего найдено слов ' + nWords);
   Console.WriteLine();
end;
```

Запускаем программу – и все слова найдены:



Так как в нашем словаре отсутствуют имена собственные, то ни *Ливии*, ни *Боливии* мы, к сожалению, не найдём, а вот неразлучную парочку парижанина с рижанином – непременно:



Давайте подойдём к решению задачи с другого конца – с короткого.

Пусть нам нужно найти короткие слова заданной длины **nshort**, которыми заканчиваются слова более длинные:

```
procedure Solve2(nshort: int);
begin
   Console.ForegroundColor := ConsoleColor.Yellow;
```

Но короткие слова хоть и короткие, но не короче *двух* букв:

```
  if (nshort < 2) then
      exit;
   //нашли слов:
   var nWords := 0;
```

Мы легко составим список всех слов заданной длины из загруженного словаря:

```
   //список коротких слов:
   var short := spisok.Where(s -> s.Length = nshort);
```

Аналогично составляем список длинных слов, в которых должно быть по крайней мере на одну букву больше:

```
   //список длинных слов:
   var long := spisok.Where(s -> s.Length > nshort);
```

Просматриваем список коротких слов и отбираем в список **res** длинные слова, которые заканчиваются на короткое:

```
   //просматриваем короткие слова:
   foreach var wrd in short do
   begin
      //список длинных слов, оканчивающихся на wrd:
      var res := long.Where(s -> s.EndsWith(wrd));
```

Если нам удалось найти подходящие длинные слова, то мы печатаем наши находки парами: короткое слово – длинное слово:

```
      //если список не пустой,
      if (res.Count > 0) then
      //печатаем слова:
      begin
         foreach var s in res do
         begin
            Console.WriteLine(wrd + ' ' + s);
```

```
            nWords += 1;
         end;
         Console.WriteLine();
      end;
   end;
   Console.WriteLine('Всего найдено слов ' + nWords);
   Console.WriteLine();
end;
```

Для **5-буквенных** коротких слов нашлось 4078 длинных:

```
Квантик 2015 06 27 2

ЩИПКА  РАСЩИПКА

ЩУРКА  ЯЩУРКА
ЩУРКА  ПРИЩУРКА

ЭКРАН  КИНОЭКРАН
ЭКРАН  ПОЛИЭКРАН
ЭКРАН  ТЕЛЕЭКРАН
ЭКРАН  ВАРИОЭКРАН
ЭКРАН  СВЕТОЭКРАН
ЭКРАН  СТЕРЕОЭКРАН

ЭЛИТА  СУПЕРЭЛИТА

ЭМАЛЬ  НИТРОЭМАЛЬ
ЭМАЛЬ  СТЕКЛОЭМАЛЬ

ЭНЗИМ  КОЭНЗИМ

ЭТИКА  ПОЭТИКА

ЮРИСТ  ВОЕНЮРИСТ
ЮРИСТ  АВАНТЮРИСТ
ЮРИСТ  МИНИАТЮРИСТ

Всего найдено слов 4078
```

Этот способ хорош тем, что все длинные слова печатаются не вразброс, а плотной кучкой, что очень удобно, например, при поиске рифм:

*Мой знакомый **юрист** –*
*Большой авант**юрист**!*

Париж-анин

Мы нашли все длинные слова, которые **заканчиваются** короткими. Давайте напишем проект *Париж-анин* для поиска длинных слов, которые **начинаются** с коротких.

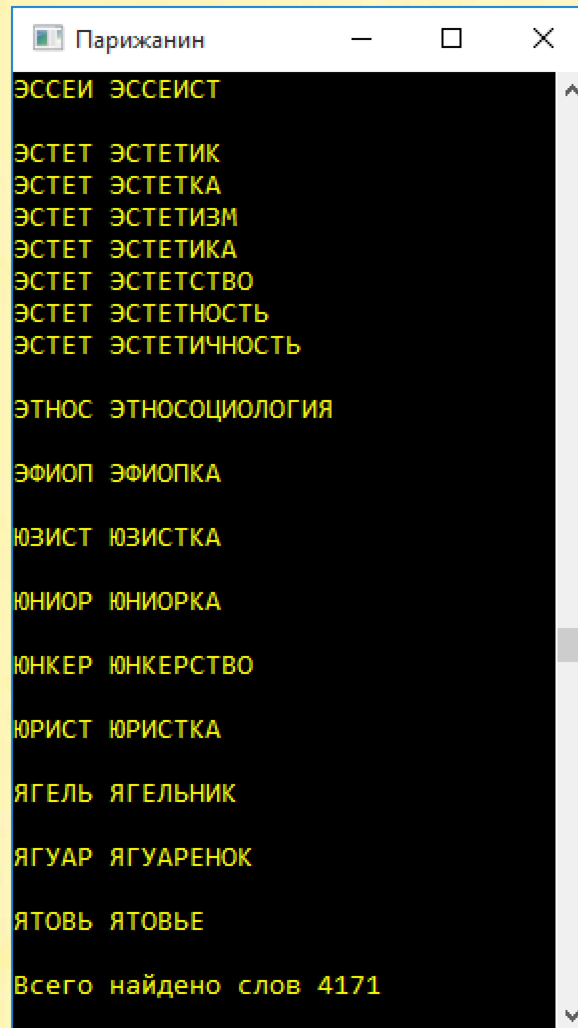
```
uses
   System;

//Париж-анин

type
   int = integer;

procedure Solve(filename: string; nshort: int);
begin
   Console.ForegroundColor := ConsoleColor.Yellow;

   if (nshort < 2) then
      exit;

   //загружаем словарь:
   var spisok := ReadAllLines(fileName);
   //нашли слов:
   var nWords := 0;

   //список коротких слов:
   var short := spisok.Where(s -> s.Length = nshort);
   //список длинных слов:
   var long := spisok.Where(s -> s.Length > nshort);

   //просматриваем короткие слова:
   foreach var wrd in short do
   begin
      //список длинных слов, начинающихся на wrd:
      var res := long.Where(s -> s.StartsWith(wrd));

      //если список не пустой,
      if (res.Count > 0) then
      //печатаем слова:
      begin
         foreach var s in res do
```

```
         begin
            Console.WriteLine(wrd + ' ' + s);
            nWords += 1;
         end;
         Console.WriteLine();
      end;

   end;
   Console.WriteLine('Всего найдено слов ' + nWords);
   Console.WriteLine();
end;

begin
   //заголовок окна:
   Console.Title := 'Парижанин';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Париж-анин');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   //словарь:
   var fileName := 'makarov_frc.txt';
   //var fileName := 'EnDictionary_frc.txt';
   //длина коротких слов:
   var len := 5;
   //ищем слова:
   Solve(fileName, len);

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

Если исключить из внимания и рассмотрения подготовительные операции, то нам пришлось изменить **единственную** строку (она выделена красным цветом) в проекте *Боливия*, чтобы решить эту задачу.

И всегда, прежде чем браться за решение новой задачи, поищите аналогичную ей или подобную, код которой нужно только немного подправить. Так вы сэкономите время и нервы.

А длинных слов, которые начинаются с коротких, в русском языке очень много. Например, для коротких 5-буквенных слов наша программа нашла 4171 длинное слово:



Слова-мамы

Слова-мамы – это слова, состоящие из двух одинаковых половин, как, например, первое человеческое слово мама.

Таких слов в русском языке немного, и их легко найти обычным способом, но мы прибегнем к **регулярным выражениям**.

Для начала мы выберем словарь для загрузки и в цикле ***for*** вызовем процедуру *Solve* для поиска всех возможных слов-мам:

```
uses
   System;

. . .

begin
   //заголовок окна:
   Console.Title := 'Слова-мамы';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Слова-мамы');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   //словарь:
   var fileName := 'makarov_frc.txt';
   //var fileName := 'EnDictionary_frc.txt';

   //ищем слова:
   for var n := 2 to 5 do
      Solve(fileName, n);

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

В процедуре **Solve** мы загружаем заданный словарь и отбираем из него только те слова, число букв в которых ровно в 2 раза превышает число букв в первой половине слова:

```
//РЕШАЕМ ЗАДАЧУ
procedure Solve(fn: string; n: integer);
begin
   Console.ForegroundColor := ConsoleColor.Yellow;
```

```
   //длина слов для поиска:
   var len := n + n;
   //нашли слов:
   var nWords := 0;
   //загружаем словарь:
   var spisok := ReadAllLines(fn)
                 .Where(w -> w.Length = len);
```

Регулярное выражение для поиска слов-мам совершенно бесхитростное. В круглых скобках следует написать столько точек, сколько букв в половине слова. После чего поставить обратную дробную черту и единицу:

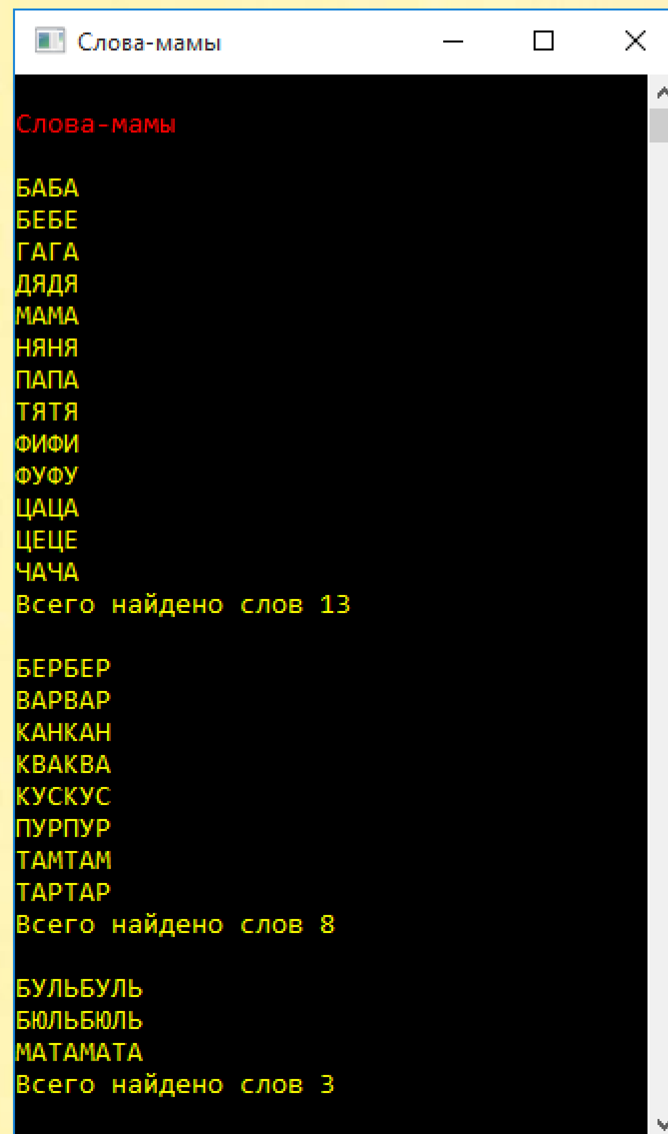
```
   //регулярное выражение:
   //(..)\1 - для n = 2
   var re: string := '(';
   for var i := 1 to n do
      re += '.';
   re += ')\1';
```

Последние 2 символа в регулярном выражении отберут из списка только те слова, в которых вторая половина **совпадает** с первой, то есть именно такие, которые нам и нужны!

В случае успеха переменная **match** получит значение *true*, и мы опубликуем слово на экране:

```
   //просматриваем все слова:
   foreach var wrd in spisok do
   begin
      var match := Regex.Match(wrd, re);
      if (match.Success) then
      begin
         Console.WriteLine(wrd);
         nWords += 1;
      end
   end;
   Console.WriteLine('Всего найдено слов ' + nWords);
   Console.WriteLine();
end;
```

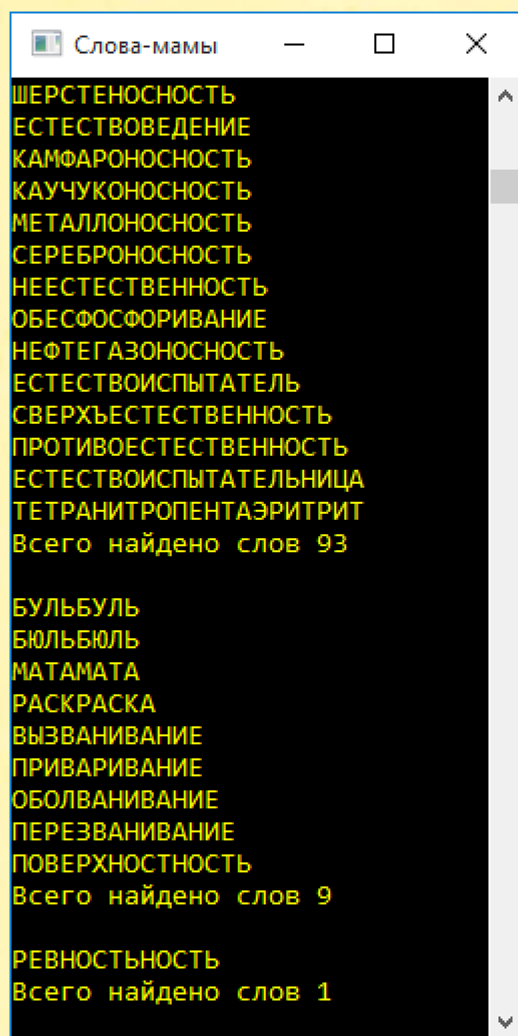
Наш способ просева и отсева слов оказался коротким и простым. На рисунке вы видите, что слов-мам в русском языке совсем немного:



Но давайте ослабим требования к искомым словам. Например, не будем принудительно ограничивать длину слов:

```
//загружаем словарь:
//var spisok := ReadAllLines(fn)
//                .Where(w -> w.Length = len);
var spisok := ReadAllLines(fn);
```

И вот что у нас получилось в результате этого послабления:



Мы напросеивали гораздо больше слов (назовём их *полумамами*), чем в первом случае.

Правда, мы нашли всего одно слово – РЕВНОСТЬНОСТЬ, в котором повторяется группа из пяти букв, но это слово – результат опечатки. Правильное написание обходится без мягкого знака в середине слова.

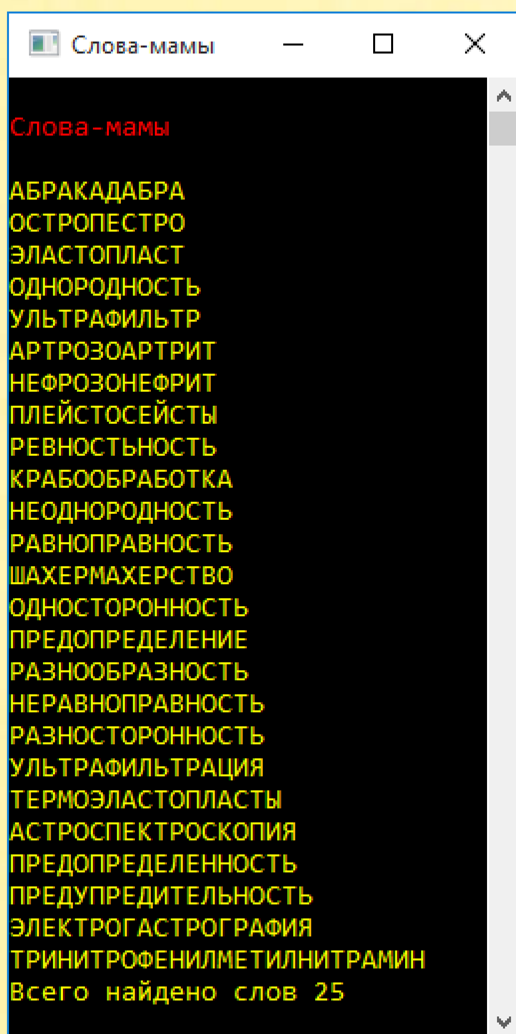
Но не будем ограничиваться полумерами и полумамами, а всерьёз займёмся поиском **слов-мачех**, в которых повторяющиеся фрагменты слов не должны стоять вплотную друг к другу.

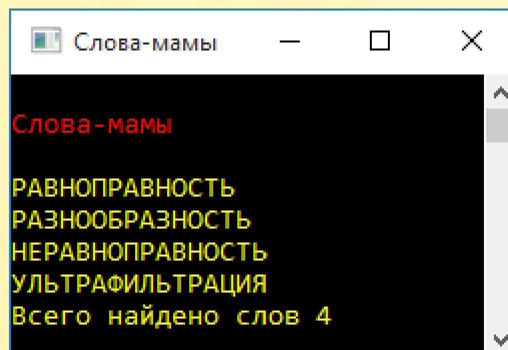
Чтобы не портить наш код, я приведу пример регулярного выражения для слов-мачех с **пятибуквенными** фрагментами:

```
re := '(.....).+\1';
```

Как видите, для допущения зазоров между группами букв нужно добавить в регулярное выражение точку и плюс после закрывающей круглой скобки.

В русском языке нашлось всего 4 слова-мачехи с пятибуквенными повторяющимися фрагментами и 25 – 4 = 21 слово – с четырёхбуквенными (включая ошибочное слово *ревностьность*):





Гуигнгнм

Гуигнгнмы – это разумные лошади из книги Джонатана Свифта *Путешествия Гулливера*.

Но нас в этом проекте интересуют не лошади, хотя бы даже и разумные, а слова, в которых согласные буквы кучно расположились плотной группой.

К ним смело можно отнести и **слова с непроизносимыми согласными**, в которых некоторые звуки не произносятся:

лестница
солнце
агентство

Итак, наша **задача** – найти слова, в которых имеется большая группа согласных.

В начале программы и в главном блоке мы выполняем традиционные мероприятия, призванные подготовить приложение к выполнению поставленной задачи, которая непосредственно решается в процедуре *Solve*:

```
uses
   System;
```

```
//Слова с большим числом согласных

type
   int = integer;

. . .

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

   //словарь:
   var fileName := 'makarov_frc.txt';
   //var fileName := 'EnDictionary_frc.txt';
   //длина группы согласных:
   var len := 3;
   //ищем слова:
   Solve(fileName, len);

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

В процедуре **Solve** мы перво-наперво загружаем словарь в строковый массив *spisok*, а затем отбираем только те слова, в которых общее число букв больше, чем заданное число согласных в группе *nsogl*:

```
procedure Solve(filename: string; nsogl: int);
begin
   Console.ForegroundColor := ConsoleColor.Yellow;

   //загружаем словарь:
   var spisok := ReadAllLines(fileName);
   //нашли слов:
   var nWords := 0;
```

```
//список слов:
var words := spisok.Where(s -> s.Length > nsogl);
```

Для формирования **шаблона** для поиска слов нам необходим список согласных, который нужно заключить в квадратные скобки:

```
//согласные:
var sogl := '[БВГДЖЗКЛМНПРСТФХЦЧШЩ]'; //ЬЙ
```

К согласным буквам относится также **Й**, но она как-то выпадает из ряда других согласных. Мягкий знак только изменяет звучание согласных, поэтому будет лишним при подсчёте согласных букв. Но вы можете самостоятельно расширить или сузить предложенный список согласных.

Шаблон для поиска слов включает список согласных, за которым в фигурных скобках следует число букв в группе:

```
var re := sogl + '{' + nsogl.ToString() + '}';
```

А дальше всё просто: мы последовательно просматриваем весь загруженный словарь и применяем к каждому слову метод **Match** класса *Regex*:

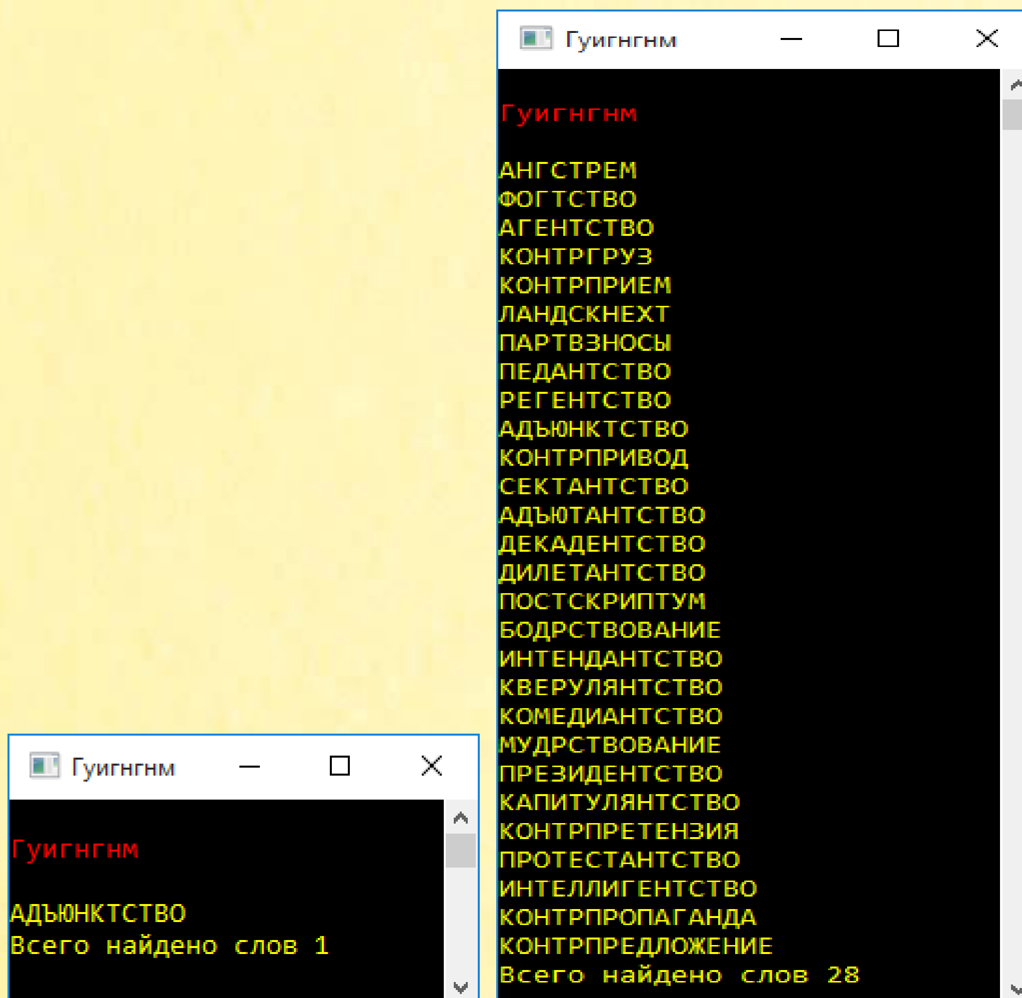
```
//просматриваем слова:
foreach var wrd in words do
begin
   var match := Regex.Match(wrd, re);
```

Если очередное слово укладывается в шаблон, то переменная **match** получает значение *true*, и мы печатаем слово на экране:

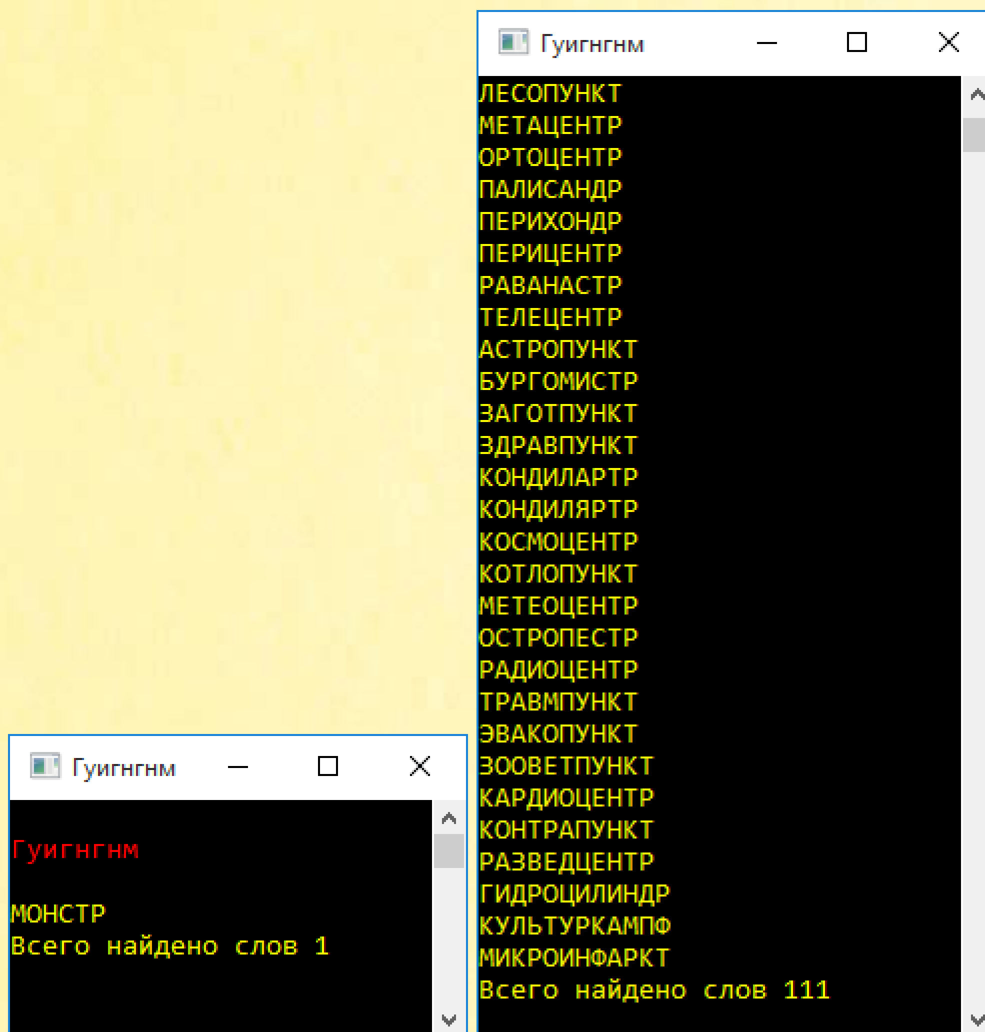
```
   if (match.Success) then
   begin
      Console.WriteLine(wrd);
      nWords += 1;
```

```
        end
    end;
    Console.WriteLine('Всего найдено слов ' + nWords);
    Console.WriteLine();
end;
```

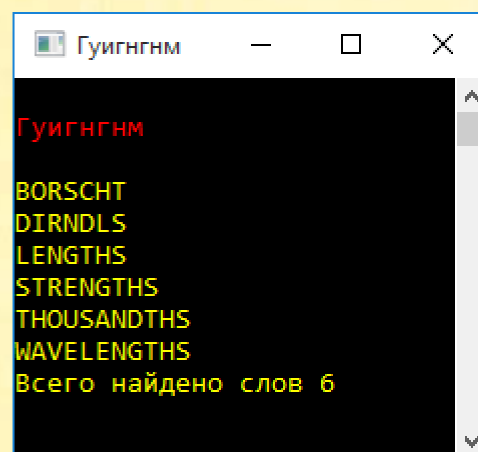
К счастью, в русском языке всего одно слово с шестью согласными, идущими подряд, и ещё 27 слов, в которых пять таких согласных. Так что за безопасность наших языков можно быть спокойными!



Так как в нашем словаре не оказалось ни одного гуингнгма, то самым гуингнгнмнообразным словом признано единственное – что символично! – слово **монстр**, заканчивающееся на 4 согласных буквы. Ещё 110 слов имеют неприятное свойство заканчиваться на 3 согласных:



Удивительно, но в английском языке, в котором слова в среднем короче, чем в русском, я отыскал 6 слов с пятью согласными на конце:



Предлагаю вам самостоятельно поискать слова-гуингнгмы и в других языках. Например, в немецком языке слово *монстр* пишется как *Monster*, то есть не относится к гуингнгмам, но зато всем известное слово ***Herbst*** заканчивается как раз на 4 согласных буквы. Другое немецкое слово ***Dirndl*** известно далеко не каждому. Оно означает женское платье, которое традиционно надевают на пивной праздник Октоберфест:

В Германии немало причудливых наречий. Одно из самых интересных – баварское. Я выписал для вас несколько баварских гуингнгмов: *Dōdschn, Duddln, Fraibialädschn, Globiàschdn, Gsäichds, Millibiddschn, Quadratratschn, Wadschn, Waiswuàschd, Zānbiàschdn.* Надеюсь, они вам понравились настолько, что вы захотели выучить и все остальные. И это правильно! Ведь даже если у вас ничего не получится, то после баварского вам даже китайский язык не покажется сложным. А ещё лучше – учите русский язык!

Кубик

Ещё одна задача из июньского номера журнала *Квантик*:

> **1 (3 балла).** Можно ли раскрасить грани куба в три цвета так, чтобы каждый цвет присутствовал, но нельзя было увидеть одновременно грани всех трёх цветов, откуда бы мы ни взглянули на куб? (Одновременно можно увидеть только три любые грани, имеющие общую вершину.)
>
> *Егор Бакаев*

Первая проблема, которую нам нужно решить, – **найти все тройки граней, которые можно видеть одновременно**.

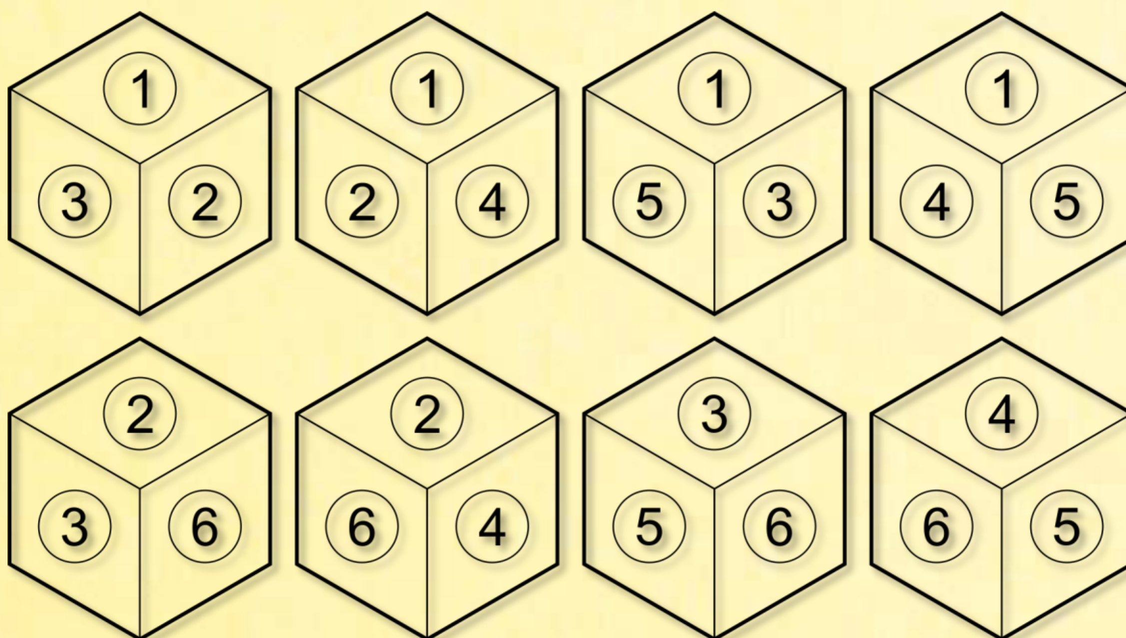
Проблема чисто комбинаторная, и она легко решается, если рассматривать не обычный кубик, а *игральный*, все грани которого пронумерованы так, что сумма очков на противоположных гранях равна 7. Это значит, что, если мы видим грань с 1, то грань с 6 очками окажется на противоположной стороне.

Следующее наблюдение. У каждой грани куба имеется по 4 соседки, две из которых могут быть видны вместе с ней (но не все сразу, естественно). При этом сумма очков на этих гранях не может равняться 7, по описанной выше причине.

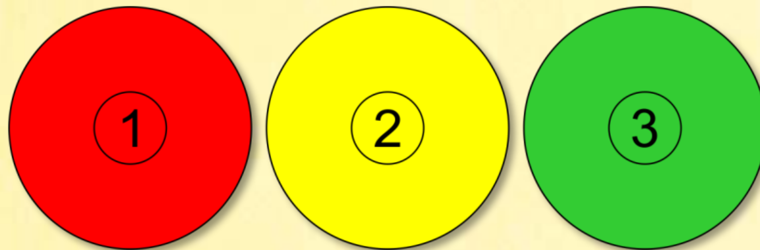
Чтобы не запутаться в гранях, начнём с **единицы** и повернём кубик так, чтобы она оказалась *наверху*. Эта грань имеет 4 соседки – **2**, **3**, **4**, **5**. Из этой четвёрки чисел нужно выбирать такие пары, которые в сумме не дают 7. Это сделать очень просто, но мы должны проверить на настоящем кубике, в каком порядке следуют цифры. Надо отметить, что существуют разные варианты оцифровки кубиков, и вы можете выбрать любой.

Затем кубик нужно поворачивать так, чтобы верхнее положение поочерёдно занимали грани **2**, **3** и **4**. Грани 5 и 6 не дают новых троек.

В итоге мы получим **все 8 троек граней куба**, которые можно видеть одновременно:



Теперь грани куба мы должны раскрасить в 3 разных цвета. Сами по себе цвета не имеют никакого значения, поэтому я выбрал «светофорный» набор, чтобы не ломать голову над решением этой колористической проблемы:



Существует «простая» **формула для подсчёта вариантов раскраски граней куба в N разных цветов**:

$$\frac{N^6 + 3N^4 + 12N^3 + 8N^2}{24}$$

Было бы совершенно неприлично, имея под рукой компьютер, считать по этой громоздкой формуле на листочке бумаге.

Поэтому мы пишем небольшую программу, в главном блоке которой вызываем функцию **GetNumColoring**, и она быстро и – главное - точно всё за нас посчитает:

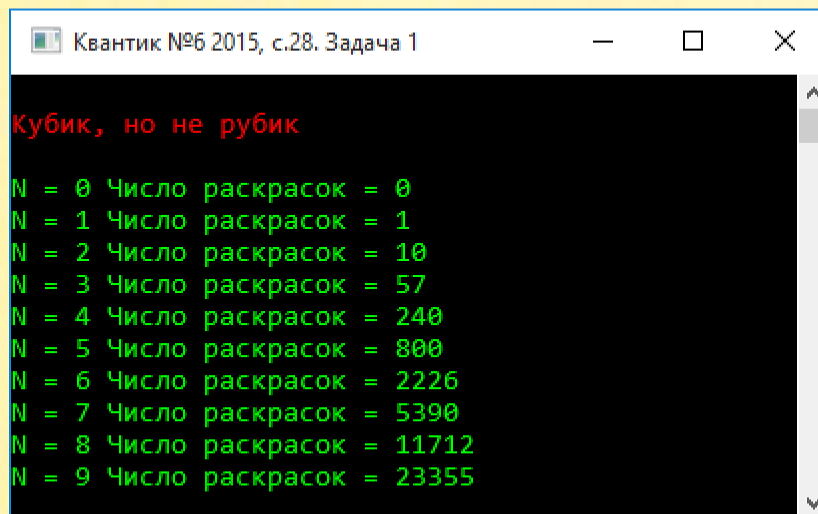
```
uses
   System;

type
   int = integer;

//Квантик 2015 06 28 1

begin
   //заголовок окна:
   Console.Title := 'Квантик №6 2015, с.28. Задача 1';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Кубик, но не рубик');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();
```

```
   for var i := 0 to 10 - 1 do
      Console.WriteLine('N = ' + i + ' Число раскрасок = ' +
                               GetNumColoring(i));

   Console.WriteLine();
   Solve();

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

А в этой функции мы должны аккуратно переписать обнародованную выше формулу на понятный компьютеру язык:

```
function GetNumColoring(n: int): int;
begin
   var res := n * n * n * n * n * n +
              3 * n * n * n * n + 12 * n * n * n + 8 * n * n;
   Result := res div 24;
end;
```

Запускаем программу и получаем короткий список вариантов раскраски для N = 0..9:



Если вас гложет любопытство, а что же за цифирь там, дальше, то вы можете увеличить переменную цикла до интересующего вас значения, а если не можете, то по адресу http://oeis.org/A047780 найдёте более длинный список вариантов раскрасок:

0, 1, 10, 57, 240, 800, 2226, 5390, 11712, 23355, 43450, 76351, 127920, 205842, 319970, 482700, 709376, 1018725, 1433322, 1980085, 2690800, 3602676, 4758930, 6209402, 8011200, 10229375, 12937626, 16219035, 20166832, 24885190, 30490050

Впрочем, нас из этого списка интересует только число **57**, показывающее сколько кубиков можно по-разному раскрасить в 3 цвета. У нас число вариантов будет заведомо меньше, потому что среди 57 раскрасок имеются и такие , которые не согласуются с условиями задачи. Например, мы не можем использовать меньше трёх цветов, а формула считает и одноцветные, и двухцветные кубики. Но даже если бы мы знали, что раскрасок должно быть 57, то всё равно формула никак не помогла бы нам их найти.

Опять перекладываем свои обязанности на плечи железного друга, помогая ему новой процедурой для решения задачи.

Совершенно очевидно, что перебор раскрасок небольшой, а задачу нужно решить всего 1 раз. Поэтому применяем **метод грубой силы** и просто выписываем 5 вложенных циклов для генерирования всех возможных раскрасок граней куба в 3 цвета.

Для удобства обозначаем цвета числами **1**, **10** и **100**. Эта уловка позволит нам быстро проверить, что 3 видимые грани не окрашены в 3 разных цвета. Действительно, если это так, то сумма цветов граней будет равна 111, в отличие от остальных случаев.

```
//РЕШАЕМ ЗАДАЧУ
procedure Solve();
const
   NUM_FACES = 6;
begin
   Console.ForegroundColor := ConsoleColor.Yellow;
```

```
var faces := new int[NUM_FACES];
var color := new int[](1, 10, 100 );
```

Нам предстоит ещё одна проверка, в которой мы должны убедиться, что в текущей раскраске использованы все 3 цвета. Это можно сделать разными способами. Например, записать все цвета граней в массив **faces**, а потом проверить, что он содержит все 3 цвета.

И последнее замечание.

Так как мы должны раскрасить не игральный кубик, а обычный, то на нём нет ни чисел, ни точек, поэтому все грани совершенно одинаковы и равноправны. Следовательно, мы любую грань можем считать верхней. Для удобства сделаем её красного цвета, тогда одним циклом у нас будет меньше:

```
//первая грань всегда красная:
var f1 := color[0];
faces[0] := f1;
//вторая грань:
foreach var f2 in color do
begin
   faces[1] := f2;
   //третья грань:
   foreach var f3 in color do
   begin
      faces[2] := f3;
      //мы видим все 3 цвета:
      if (f1 + f2 + f3 = 111) then
         continue;
      //четвёртая грань:
      foreach var f4 in color do
      begin
         faces[3] := f4;
         if (f1 + f2 + f4 = 111) then
            continue;
         //пятая грань:
         foreach var f5 in color do
         begin
            faces[4] := f5;
```

```
                if (f1 + f3 + f5 = 111) then
                   continue;
                if (f1 + f4 + f5 = 111) then
                   continue;
                //шестая грань:
                foreach var f6 in color do
                begin
                   faces[5] := f6;
                   if (f2 + f3 + f6 = 111) then
                      continue;
                   if (f2 + f4 + f6 = 111) then
                      continue;
                   if (f3 + f5 + f6 = 111) then
                      continue;
                   if (f4 + f5 + f6 = 111) then
                      continue;

                   //кубик должен быть раскрашен
                   //в 3 цвета:
                   if (not faces.Contains(color[0]) or
                                 not faces.Contains(color[1]) or
                                 not faces.Contains(color[2])) then
                      continue;
                   //нашли вариант раскраски -
                   //печатаем его:
                   var s := f1 + ' ' +
                                        f2 + ' ' +
                                        f3 + ' ' +
                                        f4 + ' ' +
                                        f5 + ' ' +
                                        f6;
                   Console.WriteLine(s);
                end
             end
          end
       end
   end;
   Console.WriteLine();
end;
```

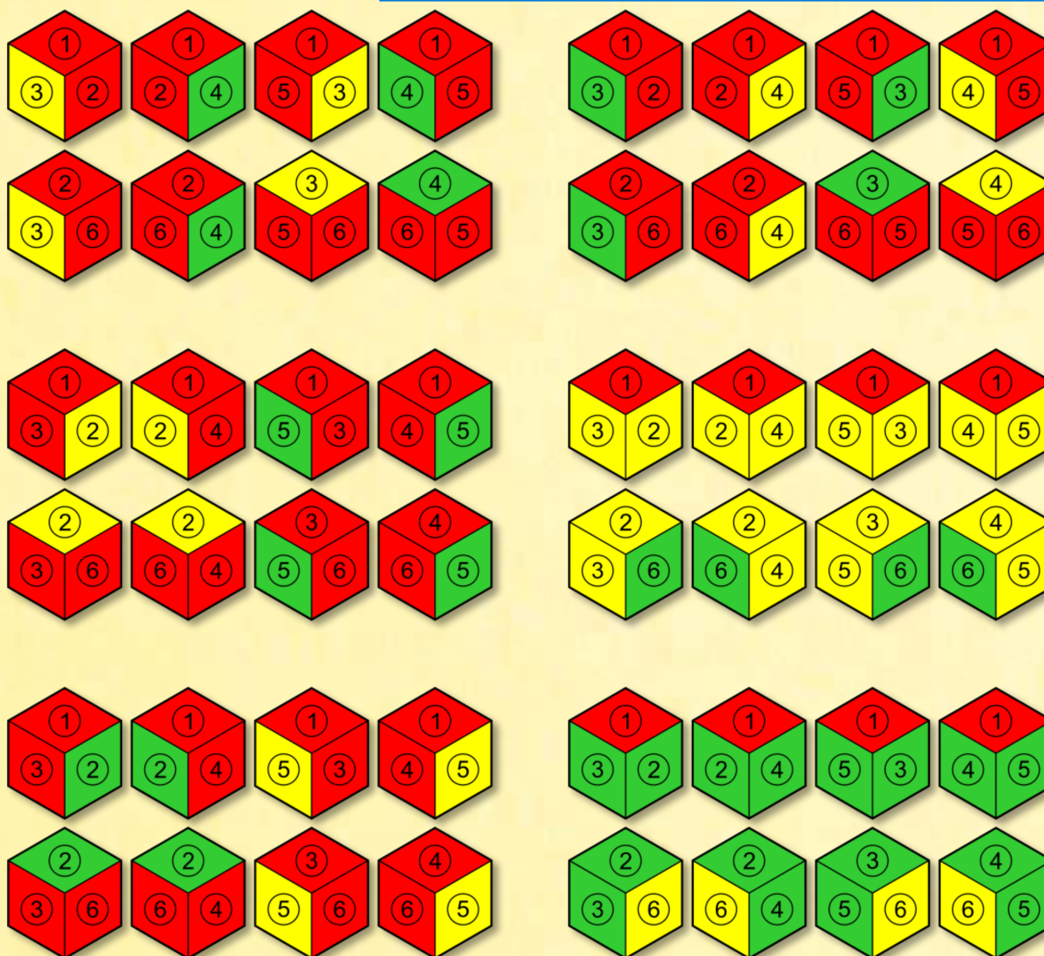
Запускаем программу и тут же получаем **ответ: существует 6 способов раскраски граней куба в три цвета** так, чтобы никакие 3 цвета нельзя было бы видеть одновременно:

Программисты вполне удовлетворились бы полученными числами, но обычным людям они ничего не говорят. Требуется расшифровка и визуализация результатов!

Так как мы обозначили числом 1 красный цвет, числом 10 – жёлтый, а числом 100 – зелёный, то без труда раскрасим грани кубика в полном соответствии с полученным отчётом:

Квантик №6 2015, с.28. Задача 1

```
Кубик, но не рубик

N = 0 Число раскрасок = 0
N = 1 Число раскрасок = 1
N = 2 Число раскрасок = 10
N = 3 Число раскрасок = 57
N = 4 Число раскрасок = 240
N = 5 Число раскрасок = 800
N = 6 Число раскрасок = 2226
N = 7 Число раскрасок = 5390
N = 8 Число раскрасок = 11712
N = 9 Число раскрасок = 23355

1 1 10 100 1 1
1 1 100 10 1 1
1 10 1 1 100 1
1 10 10 10 10 100
1 100 1 1 10 1
1 100 100 100 100 10
```



На числа в кружочках не обращайте внимания, они призваны только показать раскраску граней игрального кубика, чем мы занимались раньше.

В условии задачи не требуется найти все раскраски кубика, так что мы даже перевыполнили план. А **ответ** на задачу положительный и утвердительный: **кубик раскрасить можно!**

> Эта задача перекликается с японским садом на 15 камнях, из любой точки которого видны только 14 из них.

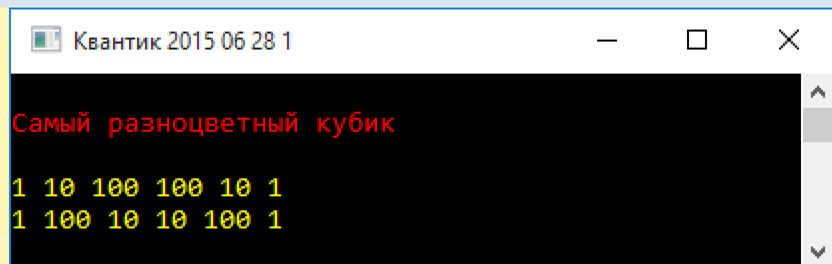
Разноцветный кубик

Давайте решим **противоположную** задачу и раскрасим грани кубика так, чтобы *всегда были видны 3 разноокрашенные грани*.

Эта задача даже легче первой, поскольку нам не нужно дополнительно проверять, сколько цветов мы использовали, - это условие выполняется автоматически:

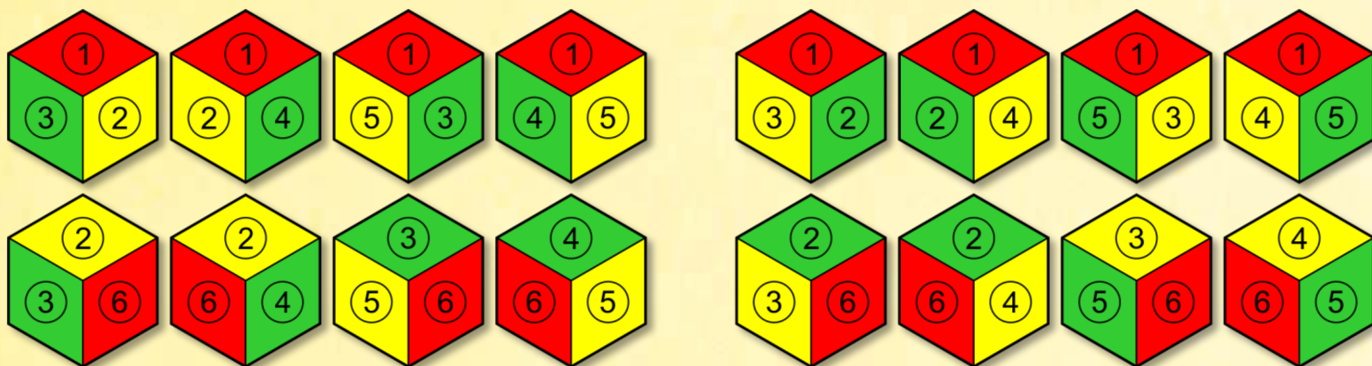
```
//РЕШАЕМ ЗАДАЧУ
procedure Solve();
begin
   Console.ForegroundColor := ConsoleColor.Yellow;

   var color := new int[](1, 10, 100 );

   //первая грань всегда красная:
   var f1 := color[0];
   //вторая грань:
   foreach var f2 in color do
   begin
      //третья грань:
      foreach var f3 in color do
      begin
         //мы видим все 3 цвета:
         if (f1 + f2 + f3 <> 111) then
            continue;
         //четвёртая грань:
```

```
         foreach var f4 in color do
         begin
            if (f1 + f2 + f4 <> 111) then
               continue;
            //пятая грань:
            foreach var f5 in color do
            begin
               if (f1 + f3 + f5 <> 111) then
                  continue;
               if (f1 + f4 + f5 <> 111) then
                  continue;
               //шестая грань:
               foreach var f6 in color do
               begin
                  if (f2 + f3 + f6 <> 111) then
                     continue;
                  if (f2 + f4 + f6 <> 111) then
                     continue;
                  if (f3 + f5 + f6 <> 111) then
                     continue;
                  if (f4 + f5 + f6 <> 111) then
                     continue;

                  //нашли вариант раскраски -
                  //печатаем его:
                  var s := f1 + ' ' +
                                       f2 + ' ' +
                                       f3 + ' ' +
                                       f4 + ' ' +
                                       f5 + ' ' +
                                       f6;
                  Console.WriteLine(s);
               end
            end
         end
      end
   end;
   Con-
sole.WriteLine();
end;
```



На этот раз мы получили только 2 варианта раскраски кубика.

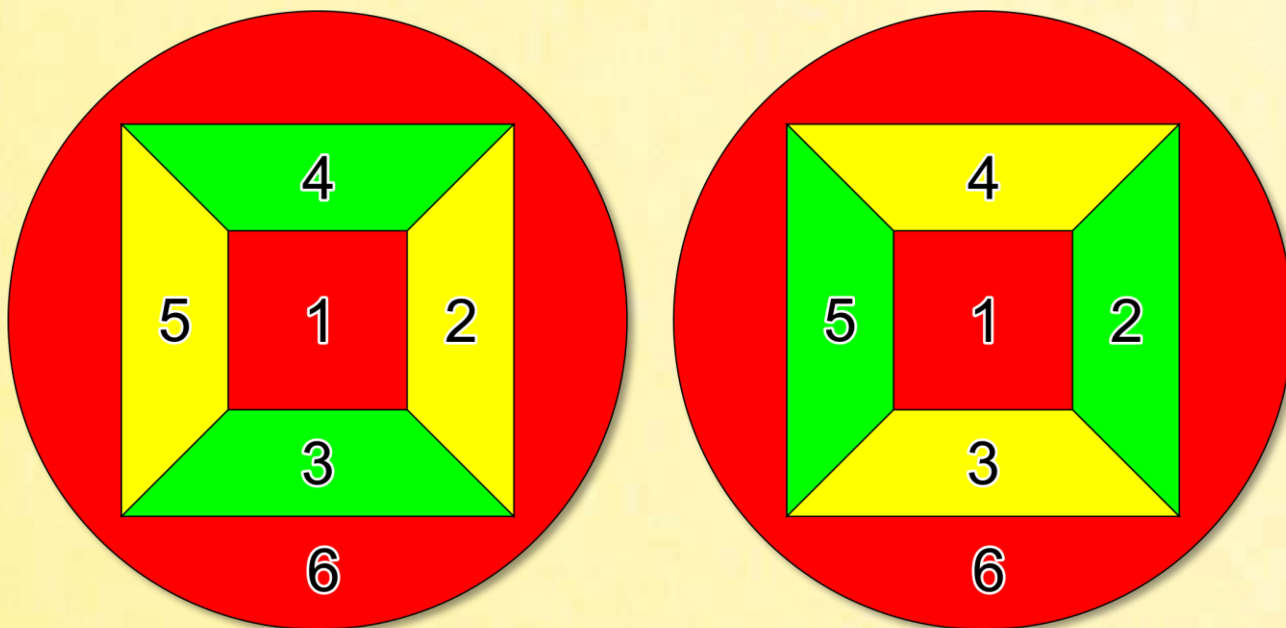
Верхняя и нижняя грани всегда красные, а остальные просто меняются цветами: жёлтые становятся зелёными, и наоборот:



А действительно ли эти два кубика *разные* или нам только так кажется?

Принято считать, что кубики одинаковые, если совпадают их развёртки. Однако этот способ проверки очень неудобен, поскольку у кубика больше десятка разных развёрток.

Но давайте посмотрим на кубик сверху так, чтобы были видны все боковые грани, а затем подрисуем и нижнюю грань. Теперь раскрасим грани кубика так, как показано на рисунке выше, и тогда мы получим 2 кубика, у которых видны все грани одновременно:



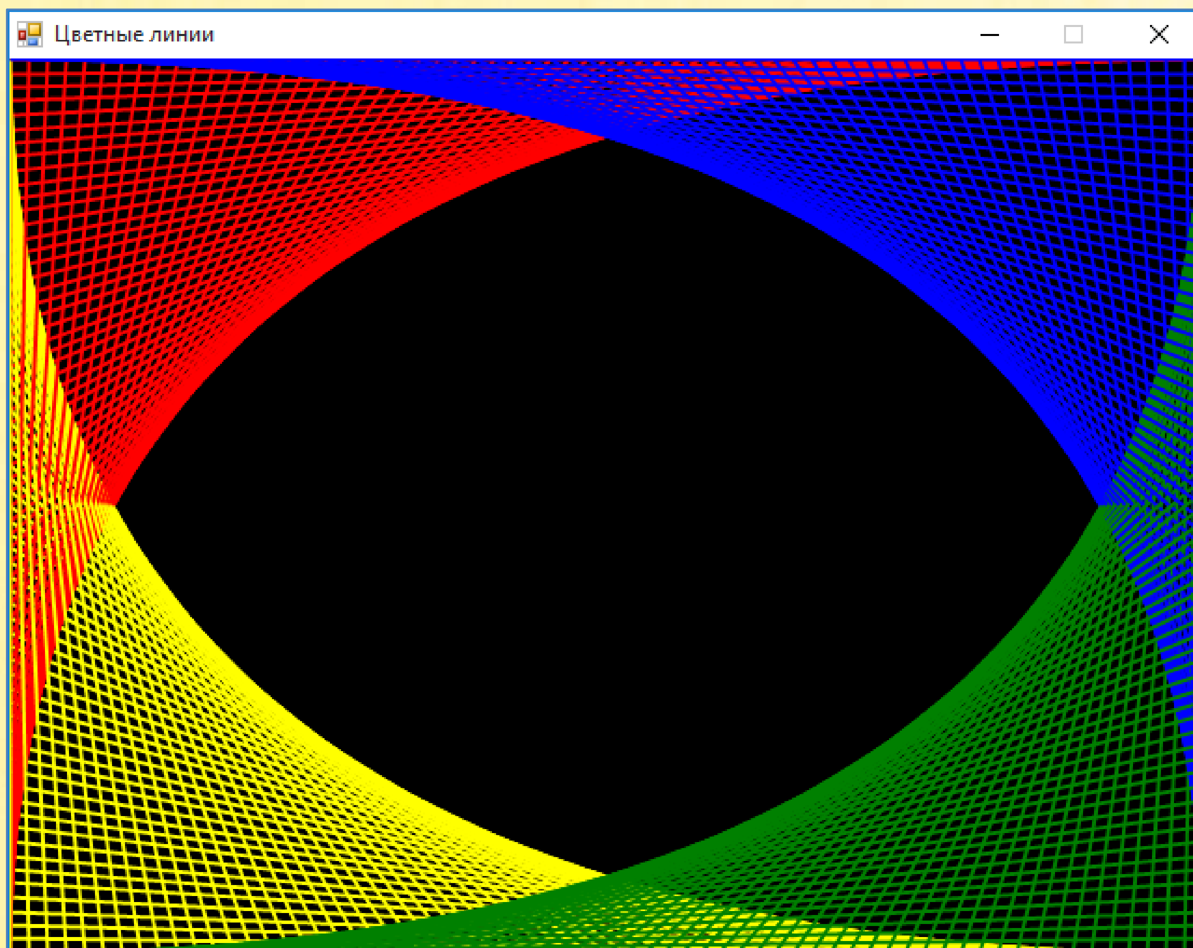
Если мы повернём левый кубик на 90 градусов, то он в точности совпадёт с правым. При этом мы не учитываем оцифровку кубика, которая в условии задачи отсутствует.

Таким образом, существует **единственная** раскраска кубика в 3 цвета, такая, что все тройки видимых граней окрашены по-разному.

Проверьте этим способом решения из проекта *Кубик*!

Цветные линии

Сейчас мы напишем совсем короткую программу, которая нарисует великолепную картинку:



Чтобы каждый из четырёх наборов линий имел свой собственный цвет, мы перед рисованием прямой линии задаём её цвет в процедуре *SetPenColor*.

Как изменяются **координаты** начала и конца линии, хорошо видно на рисунке – они скользят по границам клиентской области окна с заданным шагом *10* пикселей.

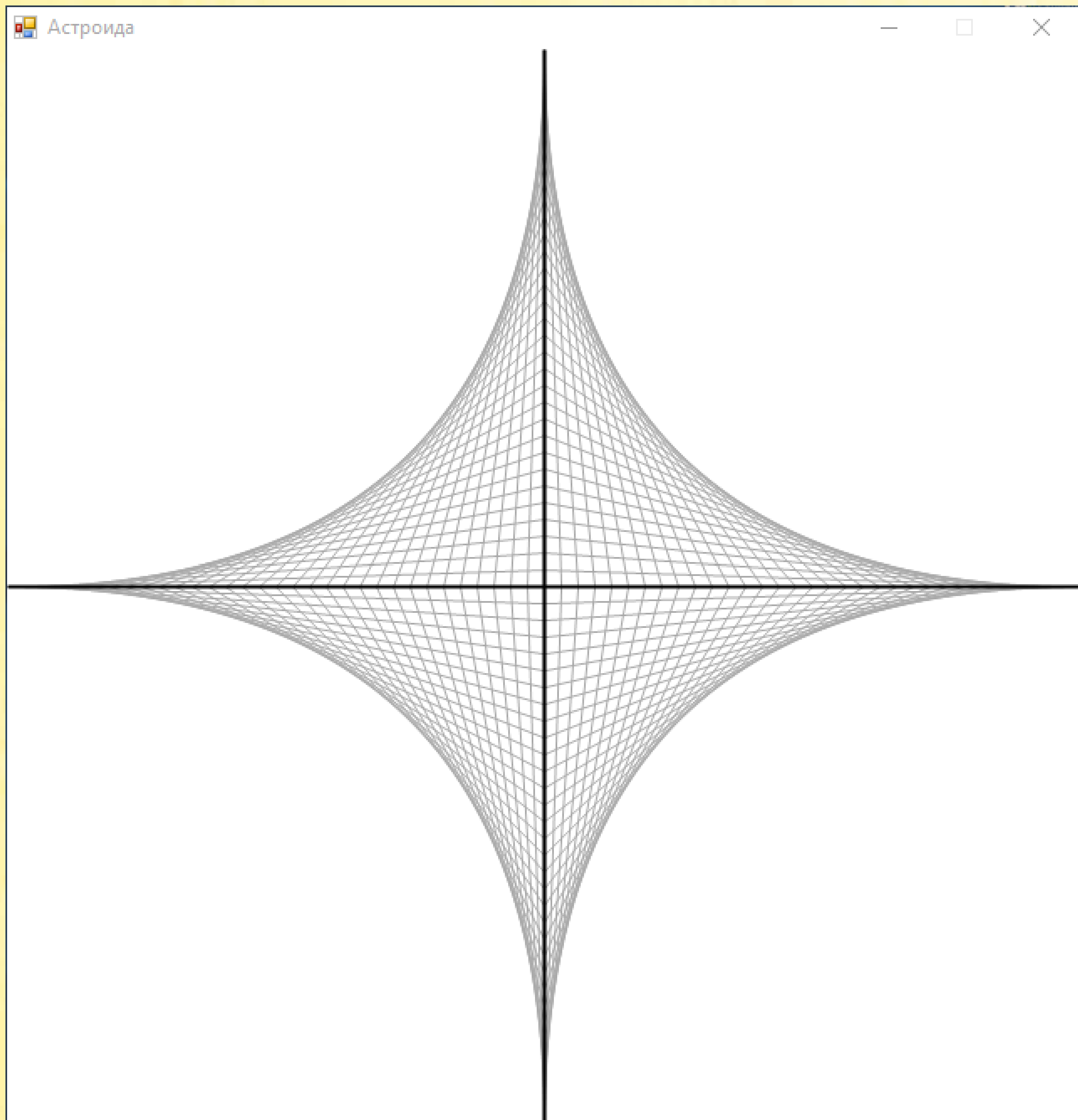
Изменяя значение этого параметра, а также цвет линий, вы сможете получить и другие узоры.

```
//ПРОГРАММА "ЦВЕТНЫЕ ЛИНИИ"

uses
   GraphABC;

const
   //толщина линий:
   penWidth = 2;

//
//   ОСНОВНАЯ ПРОГРАММА
//
begin
   SetWindowTitle('Цветные линии');
   SetWindowWidth(640);
   SetWindowHeight(480);
   Window.CenterOnScreen();
   Window.IsFixedSize := true;
   Window.Clear(Color.Black);
   var height := Window.Height;
   var width := Window.Width;

   //толщина линий:
   SetPenWidth(penWidth);
   //Устанавливаем режим сглаживания:
   SetSmoothing(true);
   //Включаем режим сглаживания:
   SetSmoothingOn;
```

```
   //отношение высоты окна к ширине:
   var ratio:= height / width;

   var x:=0;
   //Чертим цветные линии
   while(x <= width) do
   begin
      //красная линия:
      SetPenColor(Color.Red);
      Line(0, Round(x*ratio), width-x, 0);

      //жёлтая линия:
      SetPenColor(Color.Yellow);
      Line(0, Round((width-x)*ratio), width-x,
           Round(width*ratio));

      //синяя линия:
      SetPenColor(Color.Blue);
      Line(width-x, Round(0*ratio), width,
           Round((width-x)*ratio));

      //зелёная линия:
      SetPenColor(Color.Green);
      Line(width-x, Round(width*ratio), width,
           Round(x*ratio));
      x += 10;
   end;//While

end.
```

Астроида

Если проводить отрезки иначе, то мы получим **астроиду**:



Астроида – это плоская кривая, похожая на четырёхконечную звезду, от греческого названия которой она и получила название.

Мы, конечно, не начертили настоящую астроиду. Наша астроида – это «иллюзорная» **огибающая** семейства отрезков.

А вот как их можно начертить:

```
//ПРОГРАММА АСТРОИДА

uses
   GraphABC;

const
   //толщина линий:
   penWidth = 2;
   //размеры поверхности рисования:
   HEIGHT = 640;
   WIDTH = 640;

//
//   ОСНОВНАЯ ПРОГРАММА
//
begin
   SetWindowTitle('Астроида');
   SetWindowWidth(640);
```

```
    SetWindowHeight(640);
    Window.CenterOnScreen();
    Window.IsFixedSize := true;
    //фон окна:
    Window.Clear(Color.White);

    //Устанавливаем режим сглаживания:
    SetSmoothing(true);
    //Включаем режим сглаживания:
    SetSmoothingOn;

    //центр окна:
    var xc := WIDTH div 2;
    var yc := HEIGHT div 2;

    //шаг:
    var step := 10;
    //толщина линий:
    SetPenWidth(1);
    SetPenColor(Color.DarkGray);

    //проводим линии:
    for var i := 0 to Round(yc / step) do
    begin
       //левый верхний квадрант:
       Line(0 + i*step, yc, xc, yc - i*step);
       //правый верхний квадрант:
       Line(WIDTH - i*step, yc, xc, yc - i*step);
       //левый нижний квадрант:
       Line(0 + i*step, yc, xc, yc + i*step);
       //правый нижний квадрант:
       Line(WIDTH - i*step, yc, xc, yc + i*step);
    end;

    //проводим оси -->
    //толщина линий:
    SetPenWidth(penWidth);
    SetPenColor(Color.Black);
    //горизонтальная:
    Line(0, yc, WIDTH, yc);
    //вертикальная:
    Line(xc, 0, xc, HEIGHT);

end.
```

Астроида 2

На самом деле звёзды имеют форму шара, а лучи образуются в хрусталике глаза или в линзах оптических приборов. Их может быть 4, 6 или 8:

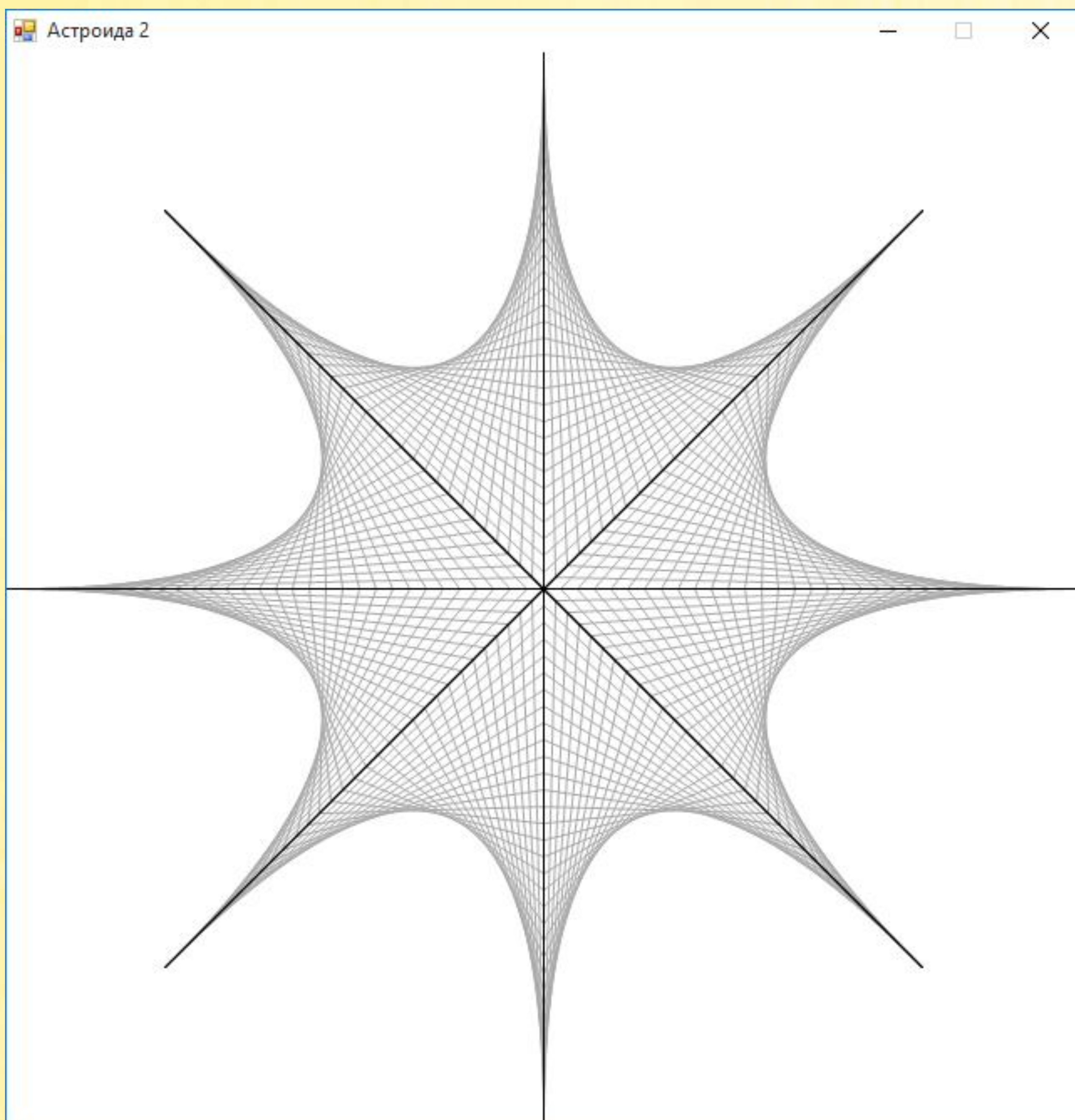
Если хорошенько постараться, то можно вычертить астроиду с **шестью** лучами:

```
//ПРОГРАММА АСТРОИДА

uses
   GraphABC;

const
   //толщина линий:
   penWidth = 2;
   //размеры поверхности рисования:
   HEIGHT = 640;
   WIDTH = 640;

   //центр окна:
var
   xc := WIDTH div 2;
   yc := HEIGHT div 2;


procedure DrawLines2(r, step, a1, a2: integer);
begin
   //число линий:
   var n := Round(yc / step);
   var rad1 := DegToRad(a1);
   var rad2 := DegToRad(a2);

   //проводим линии:
   for var i := 0 to n do
   begin
      //левый верхний квадрант:
      var x1 := Round(r div n * i * Cos(rad1));
      var y1 := Round(yc - r * Sin(rad1));
      var x2 := Round(xc - r div n * i * Cos(rad2));
      var y2 := Round(yc - r div n * i * Sin(rad2));
      Line(x1, y1, x2, y2);
   end;
end;

procedure DrawLines(r, step, a1, a2: integer);
begin
   //число линий:
```

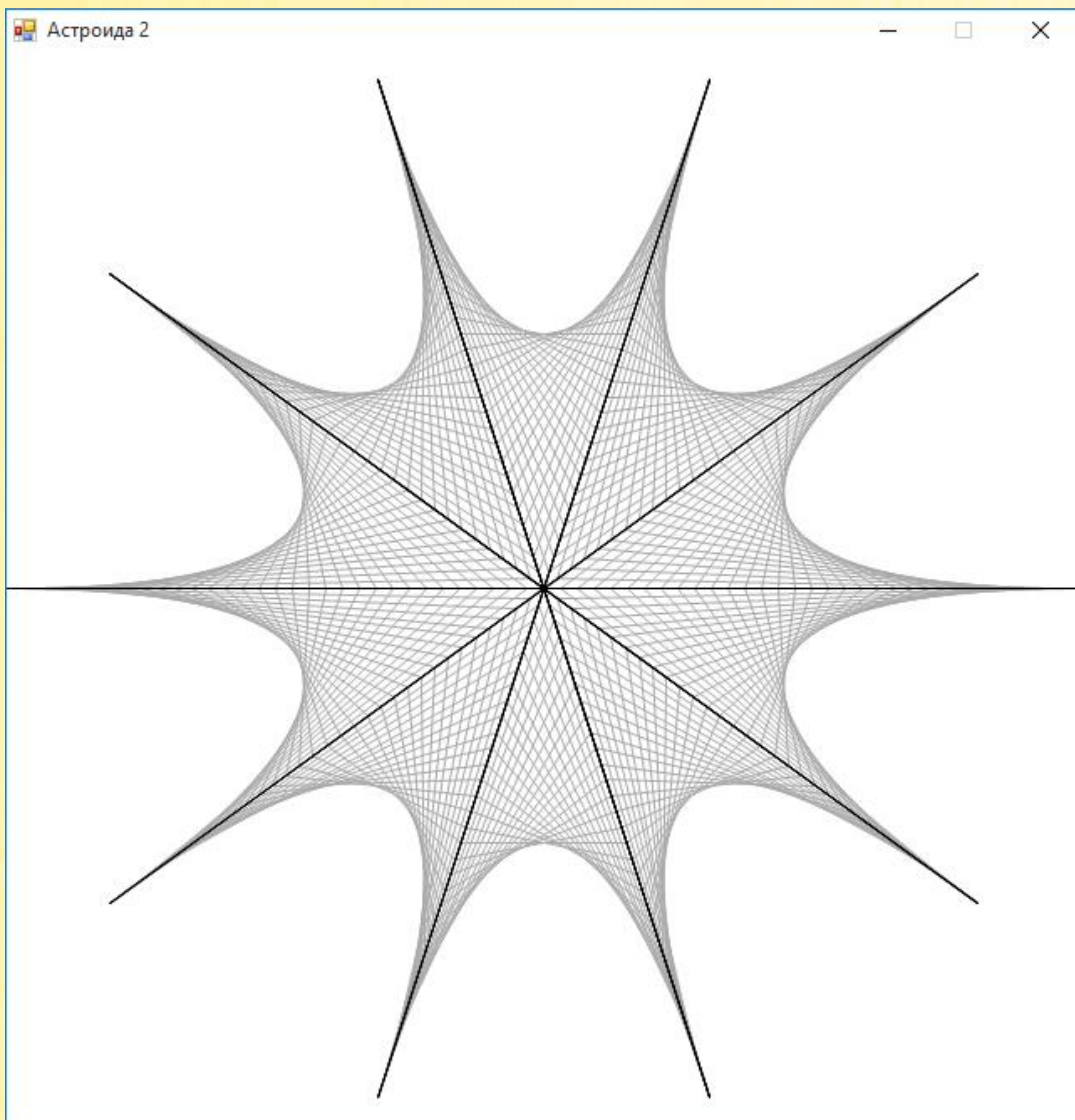
```
   var n := Round(yc / step);
   var rad1 := DegToRad(a1);
   var rad2 := DegToRad(a2);

   //проводим линии:
   for var i := 0 to n do
   begin
      var x1 := Round(xc + r div n * i * Cos(rad1));
      var y1 := Round(yc - r div n * i * Sin(rad1));
      var x2 := Round(xc + r div n * (n - i) * Cos(rad2));
      var y2 := Round(yc - r div n * (n - i) * Sin(rad2));
      Line(x1, y1, x2, y2);
   end;
end;


//
//   ОСНОВНАЯ ПРОГРАММА
//
begin
   SetWindowTitle('Астроида 2');
   SetWindowWidth(640);
   SetWindowHeight(640);
   Window.CenterOnScreen();
   Window.IsFixedSize := true;
   //фон окна:
   Window.Clear(Color.White);

   //Устанавливаем режим сглаживания:
   SetSmoothing(true);
   //Включаем режим сглаживания:
   SetSmoothingOn;

   //число лучей:
   var star := 6;//8;
   //шаг:
   var alpha := 360 div star;

   //шаг:
   var step := 10;
   //толщина линий:
```

```
   SetPenWidth(1);
   SetPenColor(Color.DarkGray);

   //длина осей:
   var r := WIDTH div 2;

   for var i := 0 to star - 1 do
   begin
      DrawLines(r, step, i * alpha, alpha * (i + 1));
   end;

   //проводим оси -->
   //толщина линий:
   //SetPenWidth(penWidth);
   //цвет линий:
   SetPenColor(Color.Black);

   //угол:
   var a := 0.0;

   //проводим линии:
   for var i := 0 to Round(180 / step) - 1 do
   begin
      var rad := DegToRad(a);
      //начало линии:
      var x1 := Round(xc - r * Cos(rad));
      var y1 := Round(yc - r * Sin(rad));
      //конец линии:
      var x2 := Round(xc + r * Cos(rad));
      var y2 := Round(yc + r * Sin(rad));
      Line(x1, y1, x2, y2);

      a += alpha;
   end;
end.
```

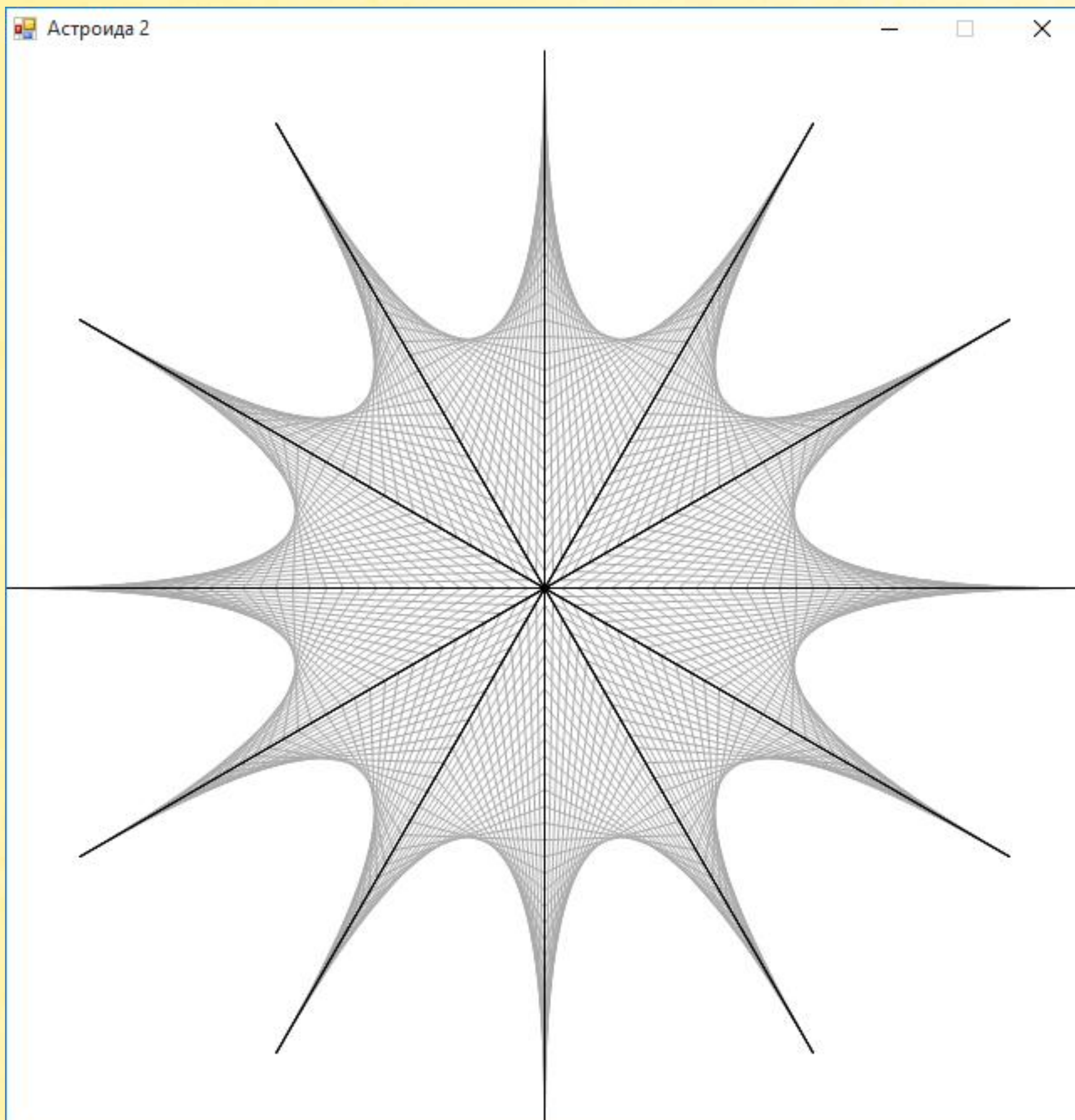
Программа получилась более сложная, но зато универсальная! **Изменив значение переменной star**, вы легко получите астроиду с **восемью** лучами:



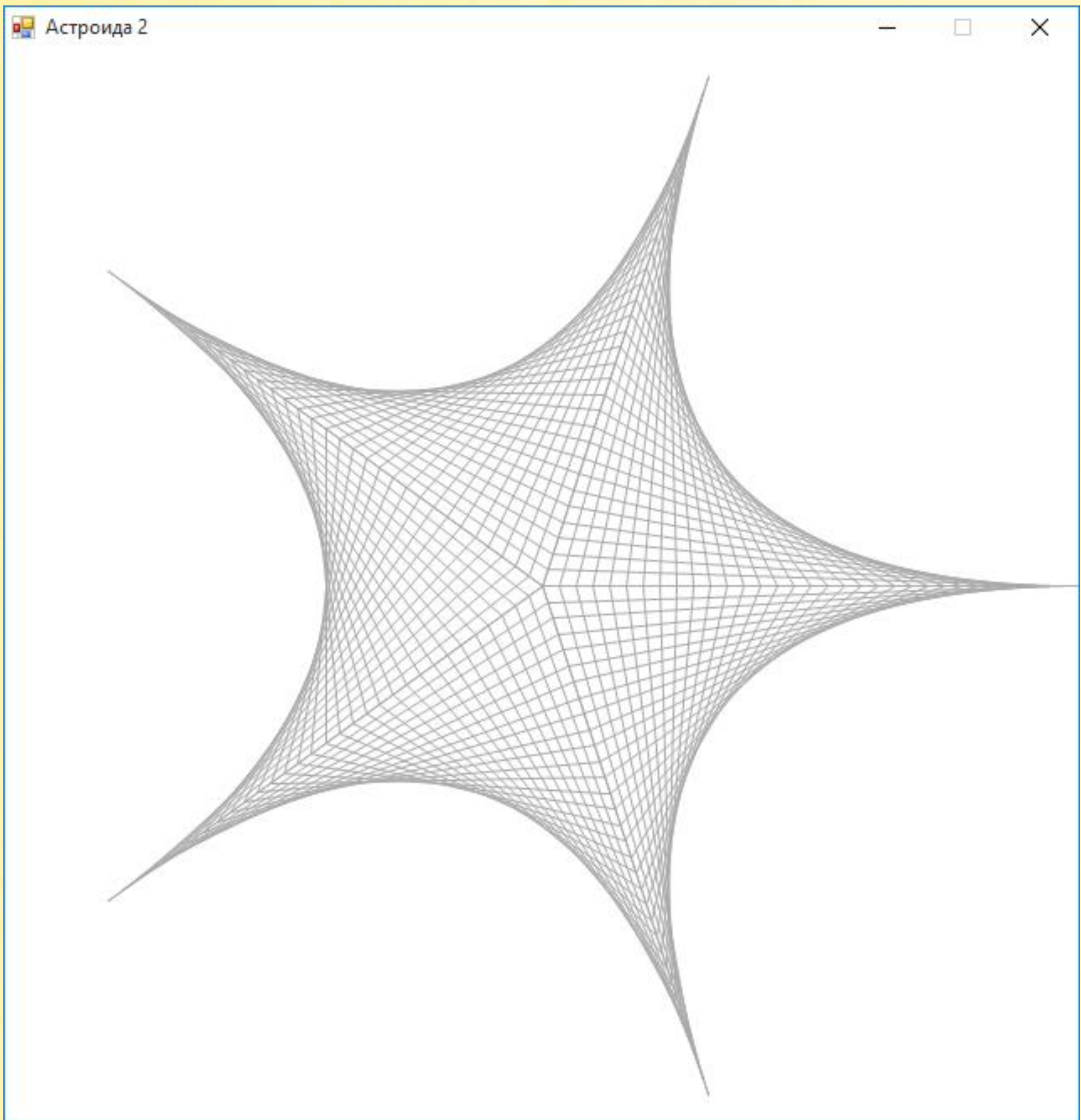
Или даже с **десятью**:



Или с **двенадцатью**:

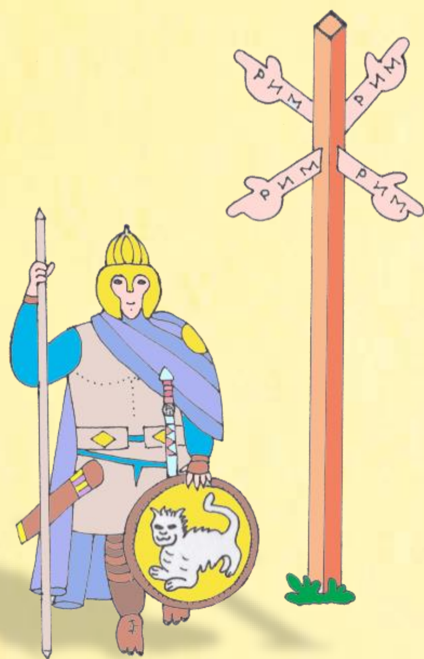


Можно чертить астроиды и с **нечётным** числом лучей, но тогда лучше обойтись без осей:



Если число лучей не является делителем числа 360, то между ними образуется зазор!

Римские числа



Древние римляне создали огромную, могущественную империю, которую назвали в свою честь, но что касается чисел, то они придумали столь вычурную систему обозначения чисел буквами, что даже простейшие арифметические вычисления давались им с большим трудом. Существенно облегчили тяготы учёбы арабские цифры (но придумали их индийцы!), пришедшие на смену римским.

Римская империя давно пала под натиском варваров, а вот их цифры сохранились до сих пор в первозданном виде. К счастью, нам не нужно пользоваться ими на уроках математики, но как украшательство мы можем найти их в книгах для обозначения глав, для подсчёта столетий, царей, съездов и других исторических событий:

Современные часы с римским интерфейсом

Поскольку компьютер не понимает не только римских цифр, но даже арабских, то мы напишем простую программу, которая поможет нам разобраться в премудростях римской нумерации.

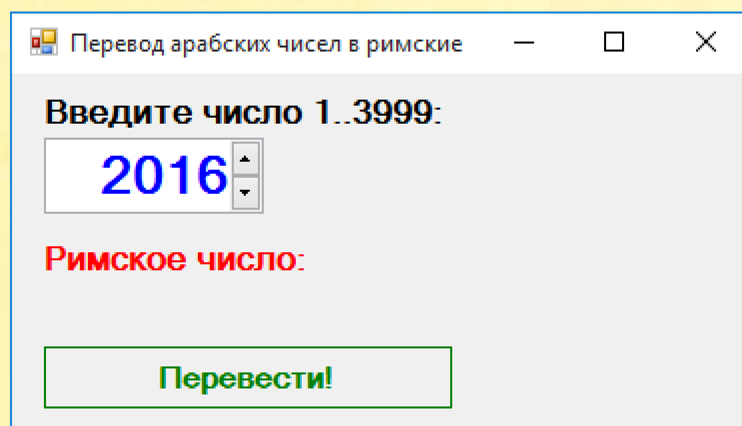
Начните новый проект. **Интерфейс** программы легко придумать: разместим на форме две *метки*, элемент управления *NumericUpDown* и *кнопку*.

Первая метка лишь объясняет пользователю назначение элемента управления *NumericUpDown*, в который он и должен ввести интересующее его арабское число. **Вторая метка** будет принимать римское число, переведённое нашим приложением «с арабского».

Мы разумно ограничим себя числом *3999* сверху и *1* снизу (а меньше у римлян и не было), потому что другие числа записывались с помощью непечатных символов. Элемент управления *udNum* не только прекрасно справляется с вводом информации, но и пристально следит за соблюдением законности: он никогда не разрешит пользователю выйти за границы дозволенного!

Снизу «венчает» наш интерфейс кнопка спокойного зелёного цвета.

Вот такая получилась замечательная картинка:



А теперь давайте создадим **словарь** для хранения чисел:

```
//словарь пар АРАБСКОЕ ЧИСЛО - РИМСКОЕ ЧИСЛО
var RNum := new Dictionary<int, string>(14);
```

При загрузке **формы** мы наполняем его содержанием:

```
procedure frmRome.Form1_Load(sender: Object; e: EventArgs);
begin
    //заполняем словарь римскими числами:
    RNum.Add(1000, 'M');
    RNum.Add(900, 'CM');
    RNum.Add(500, 'D');
    RNum.Add(400, 'CD');
    RNum.Add(100, 'C');
    RNum.Add(90, 'XC');
    RNum.Add(50, 'L');
    RNum.Add(40, 'XL');
    RNum.Add(10, 'X');
    RNum.Add(9, 'IX');
    RNum.Add(5, 'V');
    RNum.Add(4, 'IV');
    RNum.Add(1, 'I');
end;
```

В каждой паре арабско-римских чисел **ключом** служит арабское число, а **значением** - римское, которое иначе как строкой не выразишь. В словаре *RNum* мы сохраним только *13* самых необходимых сочетаний арабских и римских чисел, а остальные нам придётся вычислять. Легко понять, почему нам понадобились именно эти числа, – остальные довольно просто собрать из них. И вот как это делается.

После ввода числа и нажатия на кнопку *Перевести!* пользователь попадает в метод **btnTranslate_Click**, где в переменную *n* считывается заданное арабское число:

```
procedure frmRome.btnTranslate_Click(sender: Object; e: EventArgs);
begin
    var n := (integer)(udNum.Value);
    var s := Translate(n);
    lblRome.Text := 'Римское число: ' + s;
    lstRes.Items.Add(n.ToString() + ' -> ' + s);
end;
```

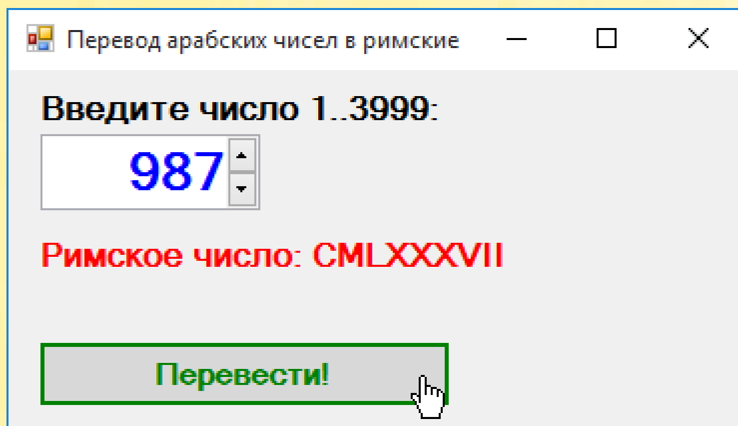
Для формирования римского числа нам потребуется *строковая переменная* **sRome**. Вся премудрость конвертирования чисел таится в двух циклах - *foreach* и *while*. Мы последовательно сравниваем заданное число *arab* с теми числами, которые хранятся в словаре *RNum*, начиная с тысяч. *Внутренний* цикл *while* как раз и нужен для того, чтобы определять, сколько в числе *arab* имеется тысяч, сотен, десятков и единиц (все остальные римские числа - 900, 500, 400, 90, 50, 9, 5, 4 - могут быть только в *единственном* числе). Если текущее значение *arab* не меньше этих чисел, то мы из него число вычитаем, а в строку *sRome* добавляем буквы, соответствующие этому числу в римской записи:

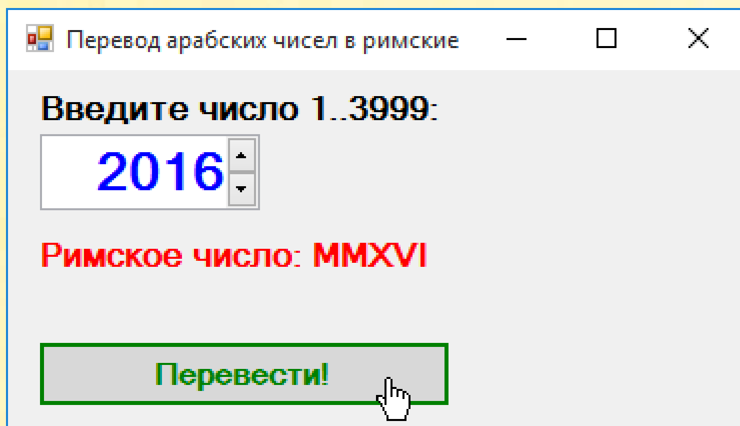
```
//ВЫЧИСЛЯЕМ РИМСКОЕ ЧИСЛО
function Translate(arab: int): string;
begin
    var sRome := '';
    foreach var num in RNum.Keys do
        while (arab >= num) do
        begin
            sRome += RNum[num];
            arab -= num;
        end;
    Result := sRome;
end;
```

Как только от числа ничего не останется, преобразования заканчиваются, и мы можем печатать на экране перевод римского числа на современный европейский математический язык:

```
lblRome.Text := 'Римское число: ' + s;
```

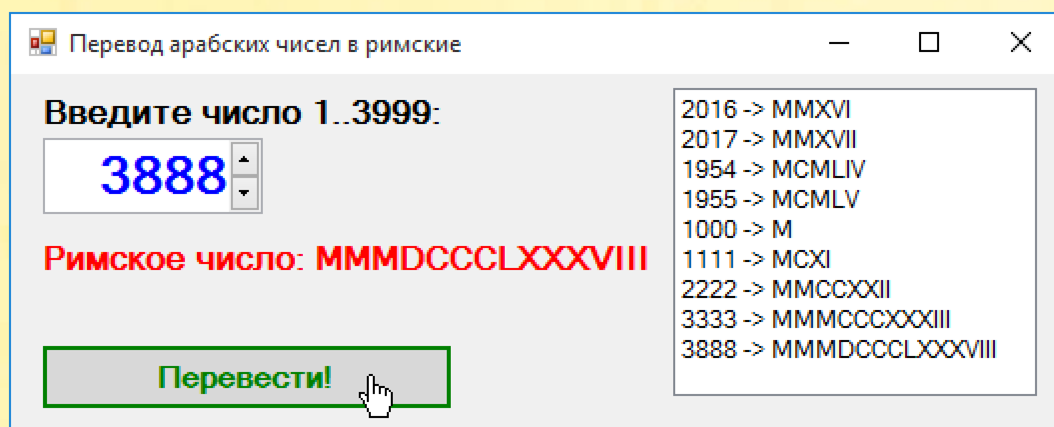
Запускайте приложение, вводите числа и нажимайте кнопку *Перевести!*





Если вы хотите запротоколировать сеанс работы с приложенем, то увеличьте размеры формы и дополните интерфейс **списком**:

Теперь ничего не пропадёт:



Монета

*Цикл **for***
***Метод** с параметрами*
*Оператор приведения **(double)***
*Условный оператор **if***
*Оператор **return***
Форматированный вывод

Жребий - это любой предмет, который можно бросить в лицо судьбе. У нас это будет **монетка**. Она очень удобна, когда, например, нужно распределить ворота команд на футбольном поле. А на третьем Чемпионате Европы по футболу в 1968 году именно монетка определила судьбу сборной СССР в полуфинале. Матч со сборной Италии, которая и принимала чемпионат, закончился вничью, а победителя тогда определяли таким немудрёным способом. Повезло итальянцам…

Монета хороша тем, что у неё 2 стороны, на которые она с **равной** вероятностью может упасть.

Понятно, что если мы подбросим монетку всего 1 раз, то выпадет либо орёл, либо решка. Судить о вероятности их выпадения невозможно. Это значит, что число подбросов нужно значительно увеличить. А это уже утомительно, поэтому пусть виртуальный эксперимент проводит компьютер – ему ничего не стоит подбросить монету и 1 миллион, и даже 10 миллионов раз кряду!

Естественно, компьютер и одного раза не сможет подбросить настоящую монетку, но зато с помощью генератора псевдослучайных чисел он может **имитировать** (или *симулировать*) выпадение орлов и решек.

В **главном блоке** мы задаём параметры эксперимента – число серий из 10 экспериментов и число подбрасываний монет в каждой серии:

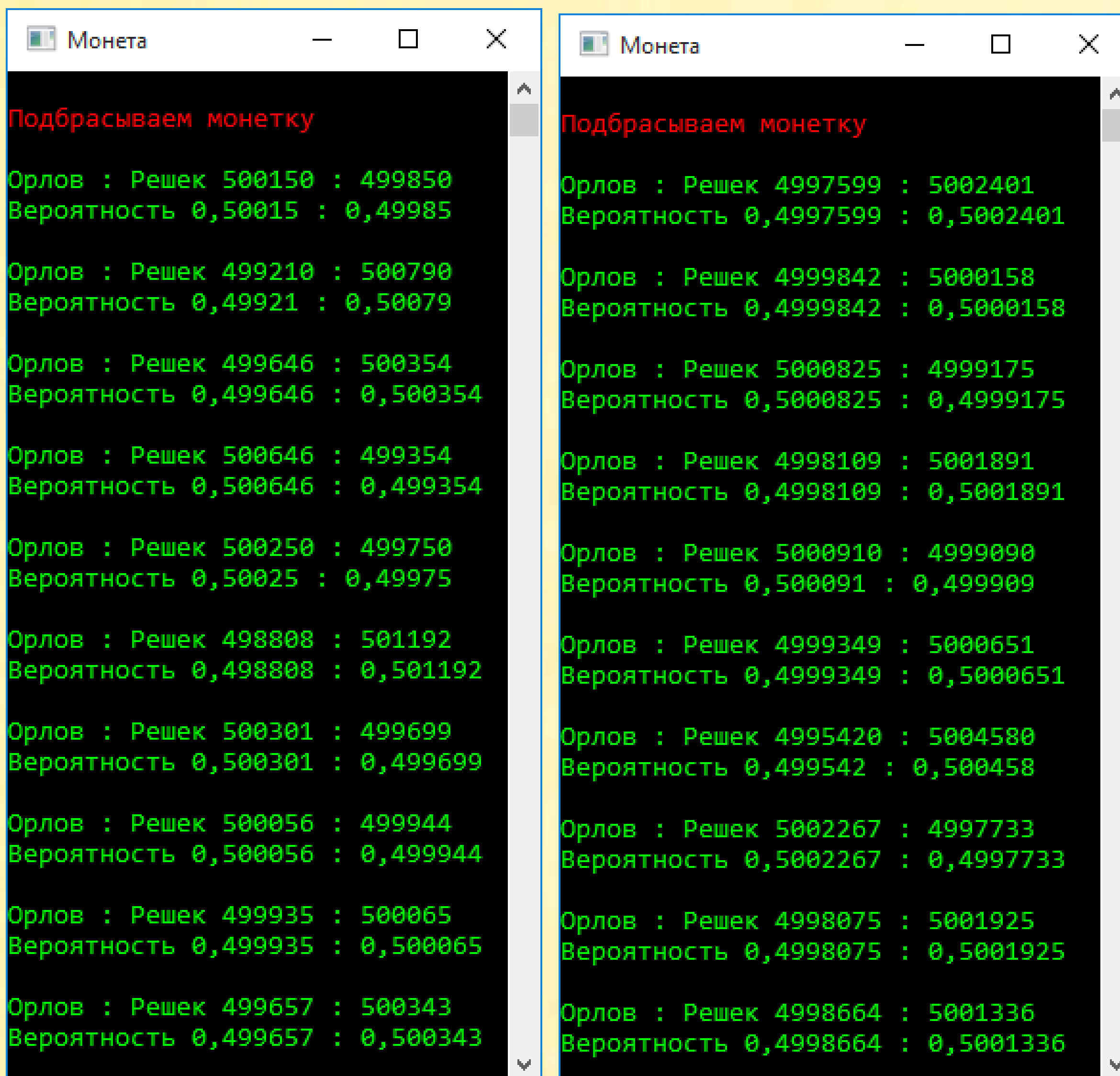
```
begin
  //заголовок окна:
```

```
   Console.Title := 'Монета';
   Console.WriteLine('');
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Подбрасываем монетку');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   //число серий:
   var ns := 10;
   for var i := 0 to ns - 1 do
   begin
      // число бросаний:
      var n := 1000000;
      var orlov := Experiment(n);

      Console.WriteLine('Орлов : Решек {0} : {1}',
                        orlov, n - orlov);
      Console.WriteLine('Вероятность {0} : {1}',
                        double(orlov) / n, 1.0 - double(orlov) / n);

      Console.WriteLine();
   end;

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

Метод **Random** с аргументом 2 будет выдавать случайную последовательность нулей и единиц. Если мы примем **1** за **орла**, **0** – за **решку**, то только успевай считать!

```
function Experiment(n: integer): integer;
begin
   var orlov := 0;
   for var i := 0 to n - 1 do
      if (PABCSystem.Random(2) = 1) then
         orlov += 1;

   Result := orlov;
end;
```

Я провёл тестирование нашей программы при 1 и 10 миллионах подбрасываниях в каждой серии. Рисунок показывает неплохое качество нашего экспериментального устройства:



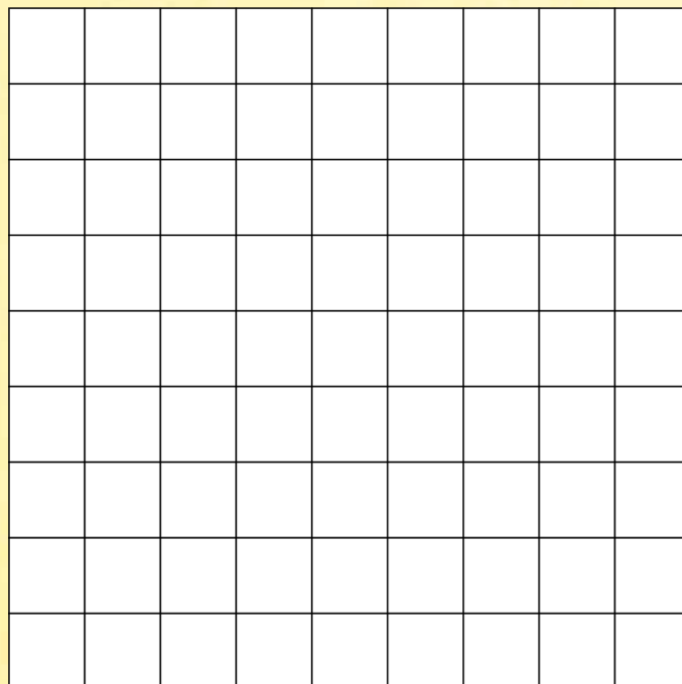
Конечно, было бы странно, если бы при случайных бросаниях число орлов и решек в точности совпадало, но эти числа должны быть довольно близки друг к другу, что мы и видим на рисунке.

Задача Клайва Синклера

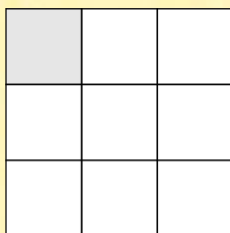
Давайте решим известную задачу. Её придумал сэр Клайв Синклер (тот самый Синклер, который сделал компьютер *ZX Spectrum*, в своё время очень популярный в Советском Союзе):

А задача такая.

Подсчитать число разных прямоугольников в сетке 9 x 9 клеток:

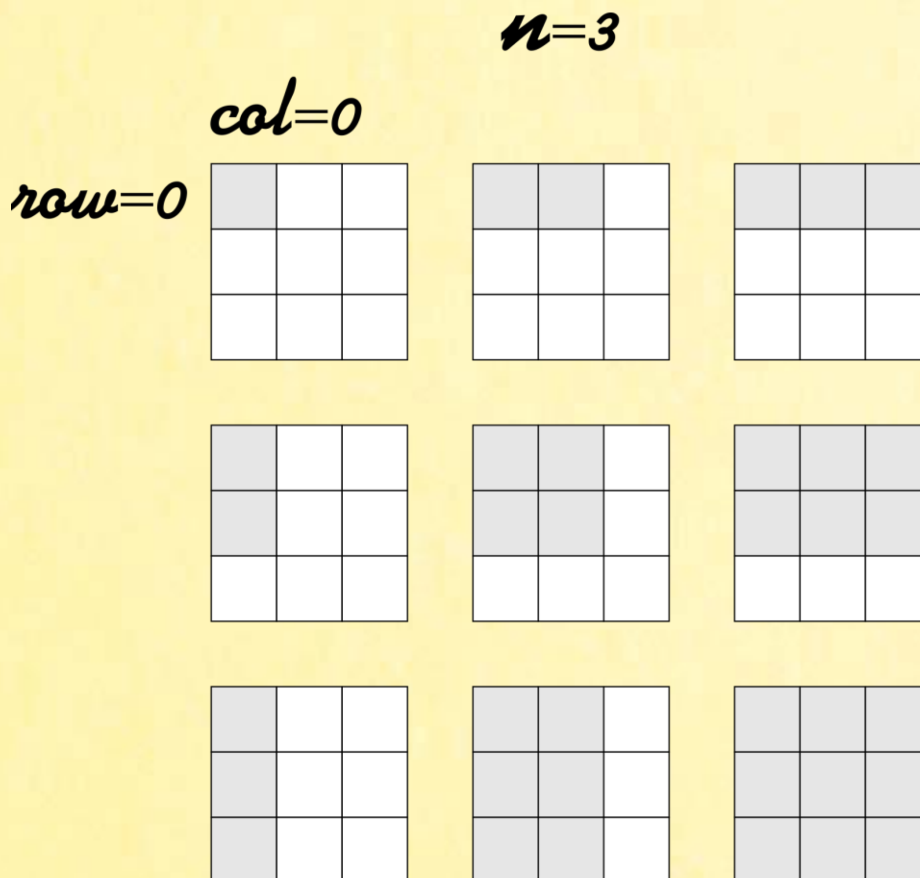


Поскольку квадрат - это тоже прямоугольник, то самый первый квадратик уже можно посчитать:

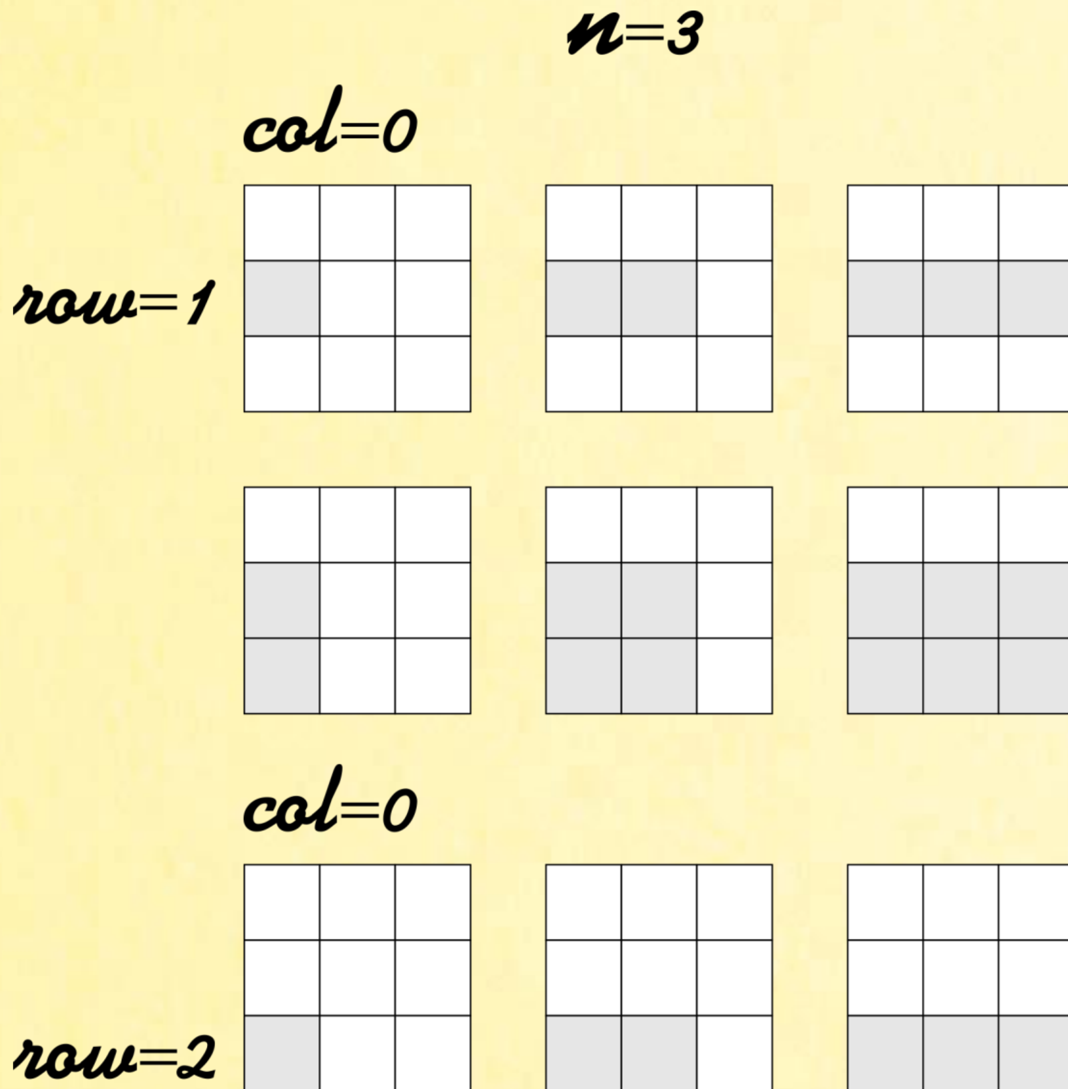


> Для уменьшения расхода бумаги мы дальше будем рассматривать сетку размером 3 x 3 клетки, а затем обобщим задачу.

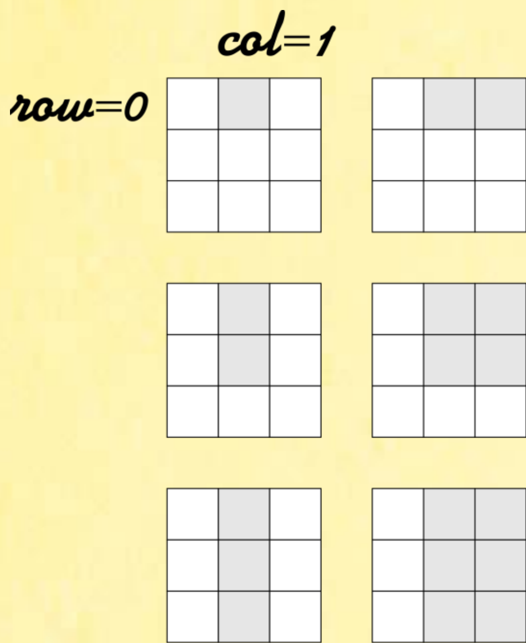
Если мы будем расширять первый прямоугольник по ширине и высоте, то всего получим 9 прямоугольников:

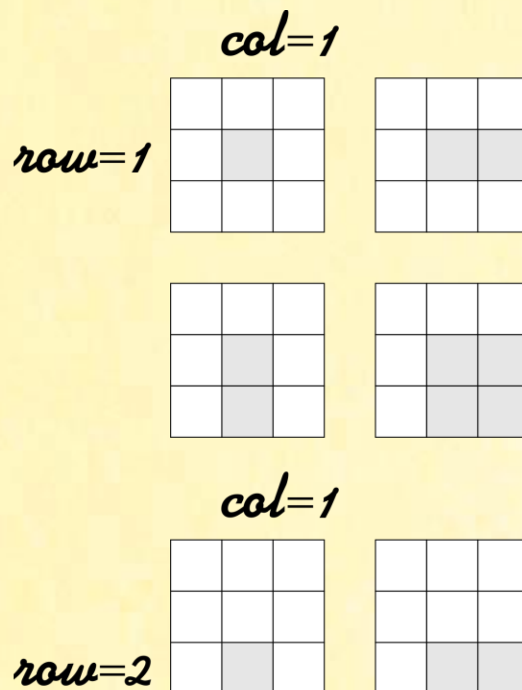


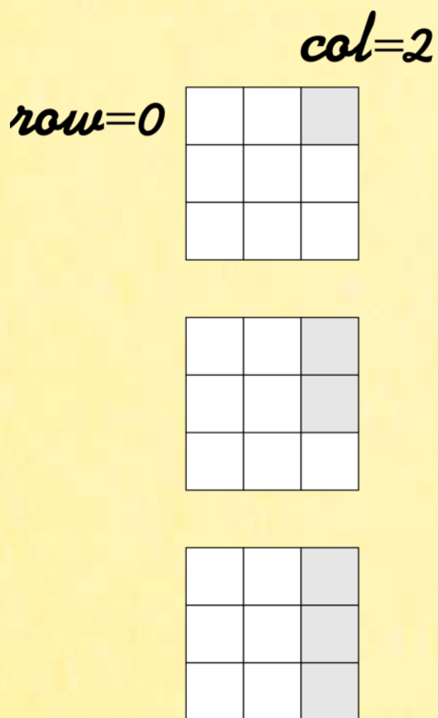
Если мы переместим угловой квадрат на 1 и 2 строки ниже, а затем выполним те же действия, то насчитаем ещё 9 прямоугольников:

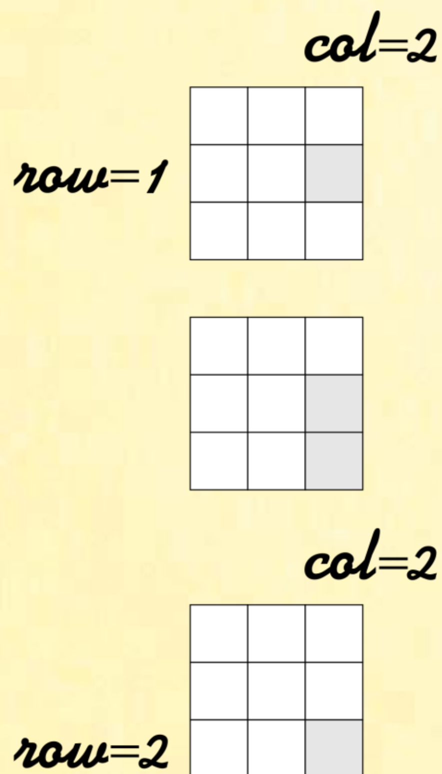


Точно так же мы подсчитаем прямоугольники для случая, когда единичный квадрат находится в первой и во второй колонках сетки:









Итого для сетки 3 x 3 мы насчитали 36 разных прямоугольников. Давайте разберёмся, как получилась эта сумма.

1. Мы передвигаем единичный квадрат от нулевой колонки до колонки 3-1 = 2 (не забывайте, что нумерация начинается с нуля!). На рисунках хорошо видно, что по ширине всегда получается *3-col* прямоугольников.

То есть:

для col=0 → 3
для col=1 → 2
для col=2 → 1

2. Мы передвигаем единичный квадрат на всех колонках от нулевой строки и до строки 3-1=2. Каждый раз по высоте получается *n-row* прямоугольников.

То есть:

для row=0 → 3
для row=1 → 2
для row=2 → 1

3. Перемножаем число прямоугольников по ширине на число прямоугольников по высоте для всех значений *col* и *row* и добавляем к общей сумме. В итоге мы получаем число всех прямоугольников в сетке.

И нам нужно только вместо размерности сетки 3 поставить число *n* – и задача решена!

Итак, в **главном блоке** пользователь вводит число – размерность сетки. Она всегда квадратная, поэтому достаточно *одного* числа.

> Вы легко обобщите задачу для *прямоугольных* сеток, но тогда пользователь должен будет ввести *два* числа.

Подсчёт прямоугольников мы произведём в функции *CalcRects*, а **главный блок** напечатает их число:

```
begin
   //заголовок окна:
   Console.Title := 'Задача Клайва Синклера';

   //бесконечный цикл ввода данных -
   //пока пользователь не закроет программу:
   while (true) do
   begin
      Console.WriteLine('');
      Console.ForegroundColor := ConsoleColor.Red;
      Console.WriteLine('Задача Клайва Синклера');
      Console.WriteLine('');

      Console.ForegroundColor := ConsoleColor.Yellow;
      Console.BackgroundColor := ConsoleColor.Black;

      Console.Write('Введите размерность квадрата (1..) >  ');
      var number := integer.Parse(Console.ReadLine());

      // находим число прямоугольников:
      var nRects := CalcRects(number);
      // и печатаем его:
      Console.ForegroundColor := ConsoleColor.Green;
      Console.WriteLine('Число прямоугольников = ' + nRects.ToString());
      Console.WriteLine('');
   end;

end.
```

Функцию **CalcRects** кодируем по нашему алгоритму:

- Число прямоугольников храним в переменной *res*.
- Изменяем номер колонки с единичным квадратиком от *0* до *n-1*.
- Изменяем номер строки с единичным квадратиком от *0* до *n-1*.
- Подсчитываем число новых прямоугольников для текущего значения *col* и *row = 0..n-1*.
- Добавляем новые прямоугольники к общей сумме *res*:

```
//ПРОГРАММА ДЛЯ РЕШЕНИЯ ЗАДАЧИ
//КЛАЙВА СИНКЛЕРА

uses
   System;

//ПОДСЧИТЫАЕМ ПРЯМОУГОЛЬНИКИ
function CalcRects(n: integer): integer;
begin
   //число прямоугольников:
   var res := 0;
   //по ширине сетки:
   for var col := 0 to n - 1 do
   begin
      //число прямоугольников по ширине:
      var nCol := n - col;
      //по высоте сетки:
      for var row := 0 to n - 1 do
      begin
         //число прямоугольников по высоте:
         var nRow := n - row;
         //добавляем:
         res += nCol * nRow;
      end;
   end;
   Result := res;
end;//CalcRects
```

Запускаем приложение и находим число прямоугольников в **любых** квадратных сетках:

```
Задача Клайва Синклера

Задача Клайва Синклера

Введите размерность квадрата (1..) >  1
Число прямоугольников = 1


Задача Клайва Синклера

Введите размерность квадрата (1..) >  2
Число прямоугольников = 9


Задача Клайва Синклера

Введите размерность квадрата (1..) >  3
Число прямоугольников = 36


Задача Клайва Синклера

Введите размерность квадрата (1..) >  4
Число прямоугольников = 100


Задача Клайва Синклера

Введите размерность квадрата (1..) >  5
Число прямоугольников = 225
```

```
Задача Клайва Синклера

Задача Клайва Синклера

Введите размерность квадрата (1..) >  6
Число прямоугольников = 441


Задача Клайва Синклера

Введите размерность квадрата (1..) >  7
Число прямоугольников = 784


Задача Клайва Синклера

Введите размерность квадрата (1..) >  8
Число прямоугольников = 1296


Задача Клайва Синклера

Введите размерность квадрата (1..) >  9
Число прямоугольников = 2025


Задача Клайва Синклера

Введите размерность квадрата (1..) >  10

Число прямоугольников = 3025
```

Для задачи Синклера **ответ** такой: **2025 прямоугольников**.

На этом можно было бы и закончить решение задачи, но трудно не обратить внимания на то, что число прямоугольников всегда (по крайней мере, для наших значений сетки) равняется **квадрату** какого-либо числа:

$1 - 1^2$
$9 - 3^2$
$36 - 6^2$
$100 - 10^2$
$225 - 15^2$
$441 - 21^2$
$784 - 28^2$

$1296 - 36^2$
$2025 - 45^2$
$3025 - 55^2$

На сайте *oeis.org* эта же последовательность чисел скрывается под кодом **A000537**:

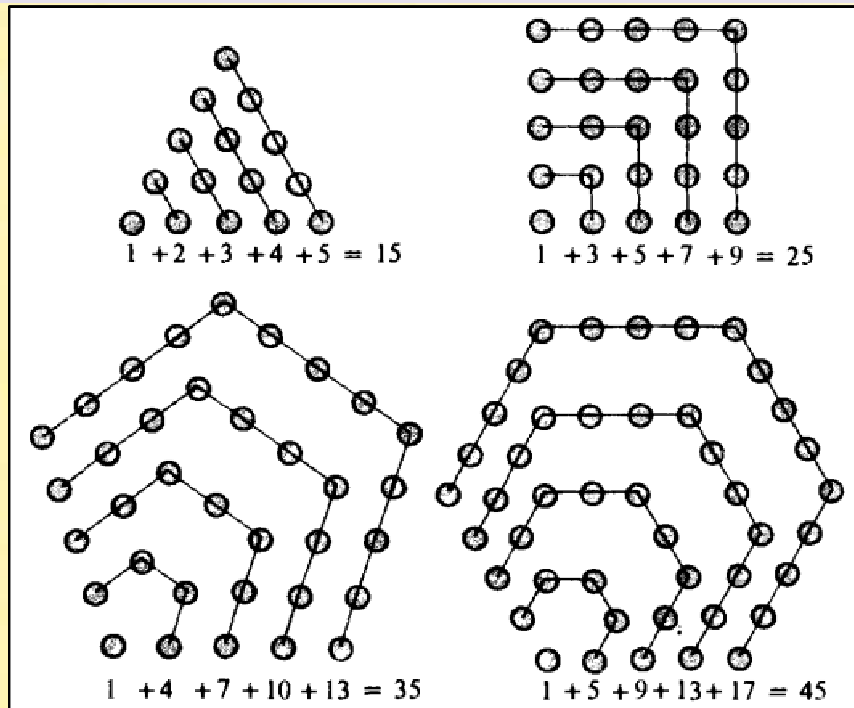
```
0, 1, 9, 36, 100, 225, 441, 784, 1296, 2025, 3025, 4356, 6084, 8281, 11025, 14400, 18496, 23409, 29241, 36100, 44100, 53361, 64009, 76176, 90000, 105625, 123201, 142884, 164836, 189225, 216225, 246016, 278784, 314721, 354025, 396900, 443556, 494209, 549081
```

И она не имеет никакого отношения к нашей задаче, а показывает *сумму n первых кубов*:

$0^3 = 0$
$0^3+1^3 = 1$
$0^3+1^3+2^3=9$
$0^3+1^3+2^3+3^3=36$
$0^3+1^3+2^3+3^3+4^3=100$
. . .

С другой стороны, последовательность 1, 3, 6, 10, 15, 21, 28, 36, 45, 55,… представляет собой *треугольные числа*.

Треугольные числа – это разновидность *фигурных* чисел, о которых можно прочитать, например, в книге Мартина Гарднера *Путешествие во времени:*



Из рисунка видно, что треугольные числа можно найти как сумму членов арифметической прогрессии:

$$1 + 2 + 3 + 4 + 5 + \ldots = n(n+1)/2,$$

где **n** – номер треугольного числа.

Решением задачи Клайва Синклера служат квадраты этих чисел, поэтому число прямоугольников в сетке размерами *n x n* клеток можно сразу же вычислить по формуле:

$$(n(n+1)/2)^2$$

Подставляем *n = 9* и получаем ответ:

$$(9x10/2)^2 = 45^2 = 2025$$

Теперь сводим наши числовые наблюдения воедино и получаем забавную формулу, связывающую кубы и квадраты чисел (а также треугольные числа):

$$0^3 = 0$$
$$0^3+1^3 = (0 +1)^2 = 1$$
$$0^3+1^3+2^3 = (0 +1 + 2)^2= 9$$
$$0^3+1^3+2^3+3^3 =(0 +1 + 2 + 3)^2 = 36$$
$$0^3+1^3+2^3+3^3+4^3 = (0 +1 + 2 + 3 + 4)^2 = 100$$
. . .

Увы, не мы первые обнаружили это удивительное свойство степеней. Древнегреческий математик Никомах доказал теорему (*Nicomachus's Theorem*), согласно которой куб любого натурального числа можно представить в виде суммы последовательных нечётных чисел:

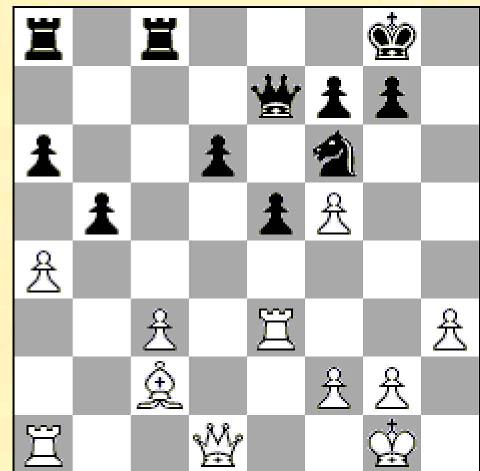
$1^3 = 1$
$2^3 = 3 + 5$
$3^3 = 7 + 9 + 11$
$4^3 = 13 + 15 + 17 + 19$
. . .

Из этой теоремы и вытекает то самое соотношение кубов и квадратов, которое мы вывели.

Вот так, странным образом задача Клайва Синклера оказалась связана с древней теоремой Никомаха!

Разменный пункт

Массив ***array of integer***
Бесконечный цикл ***do-while***
Метод ***Convert.ToInt32***
Процедура *с параметрами*
Цикл ***while***
Условный оператор ***if***
Вложенные циклы ***for***



Пусть у нас имеется сколько угодно монет любого достоинства и нам нужно составить из них сумму в **n** копеек. В комбинаторике подобная задача формулируется так.

Сколькими способами можно записать натуральное число *n* в виде суммы:

$n = n_1 + n_2 + \ldots + n_k$? (1)

Порядок слагаемых при этом не учитывается, но принято записывать их в порядке убывания, то есть от б*о*льших слагаемых к меньшим. Такая запись называется **стандартной формой разбиения** числа *n* на *k* слагаемых.

> В англоязычной литературе разбиения чисел называют **Integer Partitions**.

Если речь идёт обо всех вариантах разбиения, то их число обозначают **P(n)**.

Мы легко найдем, что:

> P(1) = 1> 1, потому что единицу невозможно представить иначе.
>
> P(2) = 2 > 2 11
>
> P(3) = 3 > 3 21 111
>
> P(4) = 5 > 4 31 22 211 1111
>
> P(5) = 7 > 5 41 32 311 221 2111 11111

Эту процедуру можно продолжить, но уже сейчас явно прослеживается её *рекурсивная* сущность. Однако мы воспользуемся *итерационным* алгоритмом, описанным Витольдом Липским в книге **[ЛВ88],** *Алгоритм 1.22.* Нам нужно только перевести его на *паскаль*.

Заданное число мы будем, как обычно, получать от пользователя и обозначим его **num**. Все слагаемые поместим в массив **adds**, их общее число в текущем разбиении сохраним в переменной **nAdds**, а число найденных вариантов – в переменной **nVar**. И для алгоритма Липского нам потребуется ещё один массив ***r***, который показывает число повторов каждого слагаемого в разбиении. Например, для разбиения пятёрки **311** – тройка повторяется 1 раз, а единица – дважды.

```
uses
   System;

//Разменный пункт

//слагаемые:
```

```
var
   adds: array of integer;
   //число слагаемых:
   nAdds := 0;
   //число повторов каждого слагаемого:
   r: array of integer;
   //число разбиений:
   nVar := 0;
```

В **главном блоке** пользователь вводит число от 2 до 20, а процедура *Generate* генерирует все его разбиения:

```
begin
   //заголовок окна:
   Console.Title := 'Разменный пункт';
   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Разменный пункт:');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   //бесконечный цикл ввода данных -
   //пока пользователь не закроет программу
   //или не введёт 0:
   repeat
      var num := ReadInteger('Введите число 2..20 > ');
      //если пользователь ввёл 0,
      //то программу закрываем:
      if (num = 0) then exit;

      //генерируем все разбиения:
      Generate(num);
      Console.WriteLine('Число разбиений = ' + nVar);
      Console.WriteLine();
   until not (true);
end.
```

В процедуре **Generate** мы по заданному числу *n* создаём оба массива и переходим в процедуру *Part*:

```
//РАЗБИВАЕМ ЧИСЛО
procedure Generate(n: integer);
begin
   adds := new integer[n + 1];
   r := new integer[n + 1];
   var s := 'Все разбиения числа ' + n.ToString() + ': ';
   Console.WriteLine(s);
   Part(n);
end;
//Генерируем все разбиения числа n
//и печатаем их
procedure Part(n: integer);
begin
   //число разбиений:
   nVar := 0;
   //первое разбиение равно числу n:
   adds[1] := n;
   r[1] := 1;
   nAdds := 1;
   Print(nAdds);
   //находим следующие разбиения:
   while (adds[1] > 1) do
   begin
      var sum := 0;
      if (adds[nAdds] = 1) then
      begin
         sum += r[nAdds];
         nAdds -= 1;
      end;
      sum += adds[nAdds];
      r[nAdds] -= 1;
      var l := adds[nAdds] - 1;
      if (r[nAdds] > 0) then nAdds += 1;
      adds[nAdds] := l;
      r[nAdds] := sum div l;
      l := sum mod l;
      if (l <> 0) then
      begin
         nAdds += 1;
         adds[nAdds] := l;
         r[nAdds] := 1;
      end;
      Print(nAdds);
   end
```

```
end;
```

Всякий раз, когда процедура *Part* сгенерирует новое разбиение, мы **печатаем** его на экране:

```
//ПЕЧАТАЕМ РАЗБИЕНИЕ
procedure Print(d: integer);
begin
   nVar += 1;
   var s := string.Empty;
   for var i := 1 to d do
      for var j := 1 to r[i] do
         s += (adds[i].ToString() + ' ');
   Console.WriteLine(s);
end;
```

И вот уже без особого напряжения мы получаем полный список разбиений числа 12:

```
Разменный пункт

Разменный пункт:

Введите число 2..20 >  12
Все разбиения числа 12:
12
11 1
10 2
10 1 1
9 3
9 2 1
9 1 1 1
8 4
8 3 1
8 2 2
8 2 1 1
8 1 1 1 1
7 5
7 4 1
7 3 2
7 3 1 1
7 2 2 1
7 2 1 1 1
7 1 1 1 1 1
6 6
```

```
Разменный пункт
4 3 1 1 1 1 1
4 2 2 2 2
4 2 2 2 1 1
4 2 2 1 1 1 1
4 2 1 1 1 1 1 1
4 1 1 1 1 1 1 1 1
3 3 3 3
3 3 3 2 1
3 3 3 1 1 1
3 3 2 2 2
3 3 2 2 1 1
3 3 2 1 1 1 1
3 3 1 1 1 1 1 1
3 2 2 2 2 1
3 2 2 2 1 1 1
3 2 2 1 1 1 1 1
3 2 1 1 1 1 1 1 1
3 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2
2 2 2 2 2 1 1
2 2 2 2 1 1 1 1
2 2 2 1 1 1 1 1 1
2 2 1 1 1 1 1 1 1 1
2 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1
Число разбиений = 77
```

Рекурсивные перестановки

Константы
*Бесконечный цикл **repeat-until***
*Функция **ReadInteger***
*Условный оператор **if***
*Оператор **exit***
*Массив **array of integer***
*Цикл **for***
***Процедура** с параметрами*
*Условный оператор **if-else***
***Процедура** с параметром-массивом*
***Рекурсивная** процедура*

Число перестановок из *n* разных элементов легко подсчитать.

На первое место можно поставить любой из *n* элементов. На второе место – любой из *(n - 1)* оставшихся. На третье – любой из *(n - 2)* оставшихся. И так далее до последнего элемента, который может занять единственное место. Общее число перестановок равно произведению этих чисел:

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

То есть факториалу числа *n*.

Вопрос в том, **как сгенерировать все перестановки**.

Оказывается, мы не первые, кто задался этим вопросом. Ещё в семнадцатом веке английские звонари научились выбивать на нескольких разных колоколах «мелодии», состоящие из всех перестановок этих колоколов. Например, для трёх колоколов нужно было сыграть такую мелодию:

> Первый колокол – второй – третий.
> Второй колокол – первый – третий.

Дальше вы и сами продолжите эту музыку.

Колоколов, конечно, было больше, а последовательность колокольных ударов нужно было держать в голове. Например, в *Книге рекордов Гиннеса* рассказывается о том, что в 1963 году за 17 с лишним часов удалось выбить на восьми колоколах все 8! = 40320 перестановок. В семнадцатом веке, конечно, эти музыкально-комбинаторные экзерсисы были короче, но, тем не менее, запомнить многие сотни перестановок было совсем непросто, поэтому звонари придумывали свои способы для «генерирования» всех колокольных перестановок. Один из таких способов мы и положим в основу компьютерной программы, которая быстро и правильно выпишет все перестановки элементов заданного множества.

Нам вполне достаточно одной константы MAX_ELEM для ограничения числа элементов в множестве, иначе их печать на экране займёт очень много времени; переменной nPerm – для хранения числа перестановок и целочисленного **массива a** – для хранения перестановки:

```
uses
   System;

//Рекурсивная генерация перестановок


const
   //макс. число элементов в
   //множестве:
   MAX_ELEM = 8;

var
   //число перестановок:
   nPerm := 0;
```

В **главном блоке** пользователь задаёт число элементов множества в диапазоне *1.. MAX_ELEM*, после чего мы создаём массив **a** и заполняем его числами *1.. MAX_ELEM*. Так мы получаем **первую** перестановку:

1 2 3 4 … MAX_ELEM

Нам гораздо удобнее переставлять именно **числа**, а не элементы других типов. Однако вы можете понимать под этими числами индексы каких-либо элементов в массиве любого типа, так что наш генератор получится вполне универсальным.

```
begin
   //заголовок окна:
   Console.Title := 'Генерируем перестановки';
   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Перестановки:');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   //бесконечный цикл ввода данных -
   //пока пользователь не закроет программу
   //или не введёт 0:
   repeat
      //число элементов:
      var nElem := ReadInteger('Число элементов
                        (1..' + MAX_ELEM +') > ');
      //если пользователь ввёл 0,
      //то программу закрываем:
      if (nElem = 0) then
          exit;

      //число перестановок:
      nPerm := 0;
      //массив для хранения очередной перестановки:
      var a := new integer[nElem];
      //начальная перестановка:
      for var e := 0 to nElem - 1 do
         a[e] := e + 1;
```

Генерирование всех остальных перестановок, а также их печать происходит в рекурсивной процедуре **PermutationRec**, которой мы передаём три аргумента:

1. всегда 0
2. число элементов в массиве nElem

3. массив **a**

```
      //генерируем перестановки:
      PermutationRec(0, nElem, a);

      Console.WriteLine('Число перестановок = ' + nPerm);
      Console.WriteLine();
   until not (true);
end.
```

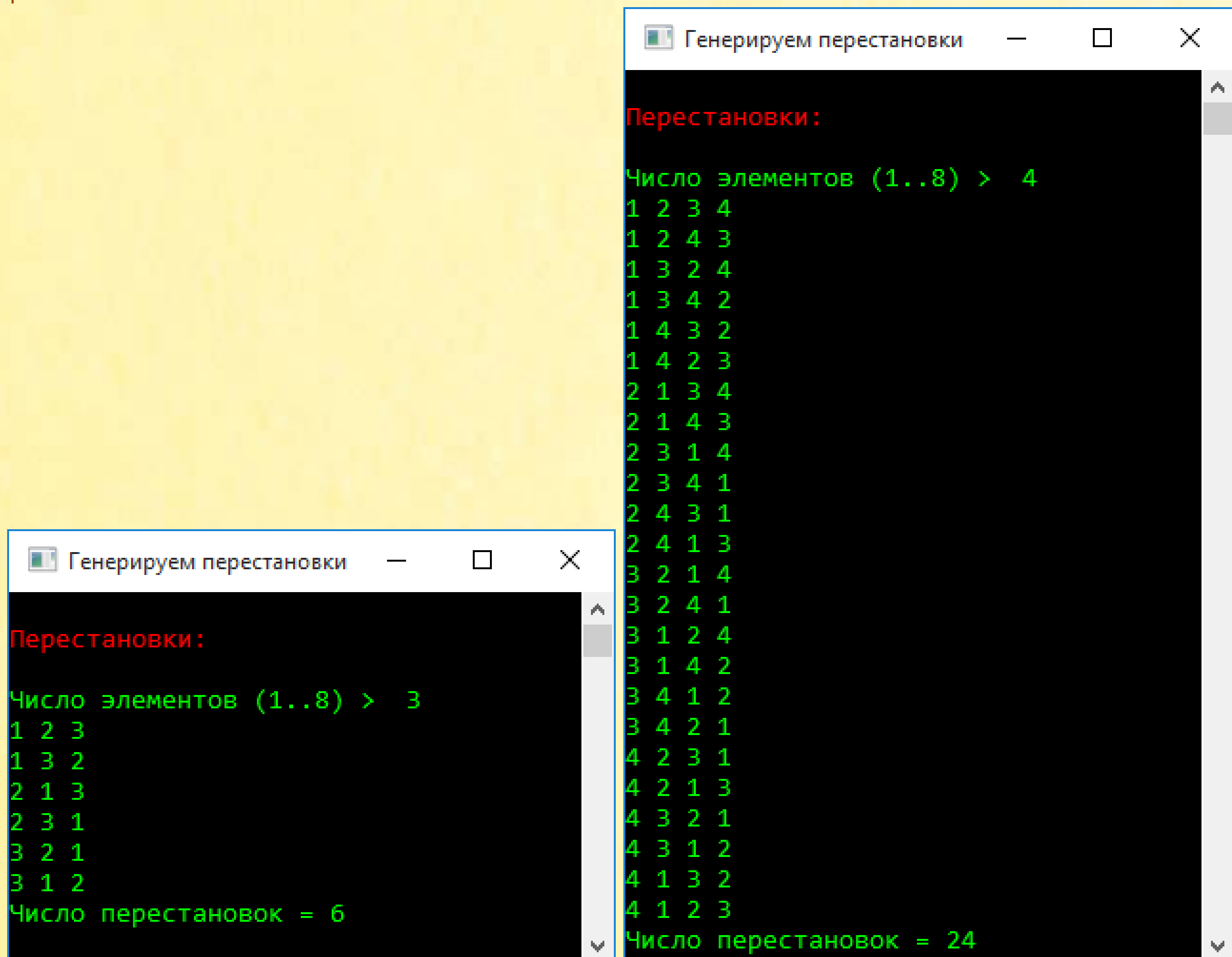
В процедуре **PermutationRec** мы переставляем элементы массива до тех пор, пока не получим новую перестановку при *k = n*, после чего печатаем её на экране с помощью метода **Println** и продолжаем генерировать новые перестановки:

```
//МЕНЯЕМ ЗНАЧЕНИЯ ЗАДАННЫХ ЭЛЕМЕНТОВ МАССИВА
//ГЕНЕРИРУЕМ ПЕРЕСТАНОВКИ
procedure PermutationRec(k, n: integer; a: array of integer);
begin
    //нашли очередную перестановку:
   if (k = n) then
   begin
      nPerm += 1;
      //печатаем её:
      a.Println();
   end
   else
   begin
      for var i := k to a.Length - 1 do
      begin
         Swap(a, k, i);
         PermutationRec(k + 1, n, a);
         Swap(a, k, i);
      end
   end
end;
```

Для удобства обмен значений двух элементов массива мы вынесли в отдельную процедуру **Swap**:

```
procedure Swap(a: array of integer; e1, e2: integer);
begin
   var tmp := a[e1];
   a[e1] := a[e2];
   a[e2] := tmp;
end;
```

Запускаем программу и проверяем её работу при различных значениях числа элементов в множестве. Рисунок убеждает нас, что программа верно генерирует перестановки.



В новых версиях *паскаля* появились **кортежи**, поэтому мы вполне можем обменять значения элементов массива a непосредственно в процедуре *PermutationRec*:

```
for var i := k to a.Length - 1 do
begin
   (a[k], a[i]) := (a[i], a[k]);
   PermutationRec(k + 1, n, a);
   (a[k], a[i]) := (a[i], a[k]);
end
```

Процедура *Swap* нам уже не нужна.

Создать массив ***a*** и заполнить его числами тоже можно проще:

```
// массив для хранения очередной перестановки
// и начальная перестановка:
var a := Range(1, nElem).ToArray();
```

«Правильные» перестановки

Константы
*Бесконечный цикл **repeat-until***
*Метод **ReadInteger***
***Метод** с параметрами*
*Массив **array of integer***
*Цикл **for***
*Цикл **while***
*Оператор **exit***

Схема перестановки колес.

Глядя на рисунки с нашими перестановками, вы легко заметите, что некоторые из них занимают не «свои» места. Например, перестановка 2 4 3 1 оказалась выше перестановки 2 4 1 3. Иногда нужно получить перестановки в **лексикографическом порядке**, то есть в таком, в каком стоят слова в словаре. Тогда перестановка 2 4 **1**

3 должна *предшествовать* перестановке 2 4 **3** 1, поскольку единица меньше тройки. Точно так же перестановка 4 1 2 3 должна занимать место выше перестановки 4 1 3 2.

Перестановки очень часто используются в различных комбинаторных задачах, поэтому нам нужно только подобрать соответствующий алгоритм и перевести его на *паскаль*. В отличие от разработки собственно алгоритма, перевод его на любой язык – дело техники.

Главный блок мы опять используем для ввода числа элементов в массиве и вызова функции *Permutation*:

```
uses
   System;

//Генерация перестановок в лексикографическом порядке

//макс. число элементов в
//множестве:
const
   MAX_ELEM = 8;


begin
   //заголовок окна:
   Console.Title := 'Генерируем перестановки';
   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Перестановки:');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   //бесконечный цикл ввода данных -
   //пока пользователь не закроет программу
   //или не введёт 0:
   repeat
      //число элементов:
      var nElem := ReadInteger('Число элементов
                              (1..' + MAX_ELEM + ') > ');
      //если пользователь ввёл 0,
      //то программу закрываем:
      if (nElem = 0) then exit;
```

```
      //генерируем перестановки:
      var n := Permutation(nElem);

      Console.WriteLine('Число перестановок = ' + n);
      Console.WriteLine();
   until not (true);
end.
```

Эта функция **нерекурсивная**, поэтому создание и инициализацию массива можно перенести непосредственно в неё. Как обычно, первую перестановку образуют последовательные числа *1..n*, которые затем переставляются иначе, чем в предыдущей программе, чтобы сохранялся лексикографический порядок перестановок:

```
//ГЕНЕРИРУЕМ ПЕРЕСТАНОВКИ
function Permutation(n: integer): integer;
begin
   //число перестановок:
   var nPerm := 0;
   //массив чисел и
   //начальная перестановка:
   var a := Range(0, n + 1).ToArray();
   a[n + 1] := 0;

   var j: integer;
   repeat
      nPerm += 1;
      // печатаем очередную перестановку:
      //PrintArray(a);
      a.Where(i -> i > 0).Println();

      var i := n;
      while (a[i - 1] > a[i]) do
         i -= 1;
      j := i - 1;
      var h := a[j];
      while (a[i + 1] > h) do
         i += 1;
      a[j] := a[i];
      a[i] := h;
      i := j + 1;
      var k := n;
```

```
      while (i < k) do
      begin
         h := a[i];
         a[i] := a[k];
         a[k] := h;
         i += 1;
         k -= 1;
      end
   until not (j <> 0);
   Result := nPerm;
end;
```

На рисунке вы видите, что все перестановки заняли свои места:

```
Генерируем перестановки

Перестановки:

Число элементов (1..8) >  3
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
Число перестановок = 6
```

```
Генерируем перестановки

Перестановки:

Число элементов (1..8) >  4
1 2 3 4
1 2 4 3
1 3 2 4
1 3 4 2
1 4 2 3
1 4 3 2
2 1 3 4
2 1 4 3
2 3 1 4
2 3 4 1
2 4 1 3
2 4 3 1
3 1 2 4
3 1 4 2
3 2 1 4
3 2 4 1
3 4 1 2
3 4 2 1
4 1 2 3
4 1 3 2
4 2 1 3
4 2 3 1
4 3 1 2
4 3 2 1
Число перестановок = 24
```

Обратите внимание, что в массиве **а** первый и последний элементы всегда равны **нулю**, поэтому при печати не используются.

«Массивные» перестановки

Часто нужно не просто напечатать перестановки на экране, а получить массив всех перестановок для дальнейшего использования в программе. В этом проекте мы превратим процедуру **Permutation** в *функцию*, которая возвращает массив перестановок:

```
uses
   System;

//Генерация перестановок в лексикографическом порядке

const
   //макс. число элементов в
   //множестве:
   MAX_ELEM = 8;

//ВЫЧИСЛЯЕМ ФАКТОРИАЛ
function Factorial(num: integer): integer;
begin
   var fact := 1;
   for var i := 1 to num do
      fact *= i;

   Result := fact;
end;


//ГЕНЕРИРУЕМ ПЕРЕСТАНОВКИ
function Permutation(n: integer): array[,] of integer;
begin
   //число перестановок:
   var AllPerm := Factorial(n);
   //массив перестановок:
   var perms := new integer[AllPerm, n];

   //текущее число перестановок:
   var nPerm := 0;

   //массив чисел и
   //начальная перестановка:
```

```
    var a := Range(0, n + 1).ToArray();
    a[n + 1] := 0;

    var j: integer;
    repeat
       for var id := 1 to n do
          perms[nPerm, id - 1] := a[id];

       nPerm += 1;
       var i := n;
       while (a[i - 1] > a[i]) do
          i -= 1;
       j := i - 1;
       var h := a[j];

       while (a[i + 1] > h) do
          i += 1;
       a[j] := a[i];
       a[i] := h;
       i := j + 1;
       var k := n;
       while (i < k) do
       begin
          h := a[i];
          a[i] := a[k];
          a[k] := h;
          i += 1;
          k -= 1;
       end
    until not (j <> 0);

    Result := perms;
end;


begin
    //заголовок окна:
    Console.Title := 'Генерируем перестановки';
    Console.WriteLine();
    Console.ForegroundColor := ConsoleColor.Red;
    Console.WriteLine('Перестановки:');
    Console.ForegroundColor := ConsoleColor.Green;
    Console.WriteLine();

    //бесконечный цикл ввода данных -
```

```
   //пока пользователь не закроет программу
   //или не введёт 0:
   repeat
      //число элементов:
      var nElem := ReadInteger('Число элементов (1..' +
                               MAX_ELEM + ') > ');
      //если пользователь ввёл 0,
      //то программу закрываем:
      if (nElem = 0) then
         exit;

      //генерируем перестановки:
      var perms := Permutation(nElem);
      //печатаем перестановки:
      Console.WriteLine();
      perms.Println();

      Console.WriteLine();
   until not (true);
end.
```

«Библиотечные» перестановки

Перестановки очень часто нужны при решении комбинаторных задач, поэтому полезно иметь готовую **библиотеку** для получения перестановок.

В папке **PermuteUtilsCS** вы найдёте исходный код класса *PermUtils* (на языке *Сишарп),* метод *Permute* которого успешно генерирует все перестановки и последовательно возвращает их оператором *yield*.

Ему нужно передать коллекцию элементов типа **T** и число элементов для перестановки:

```
public static IEnumerable<IEnumerable<T>>
              Permute<T>(IEnumerable<T> list, int count)
```

В общем случае значение параметра *count* равно числу элементов в коллекции.

Динамическую библиотеку **PermuteUtils.dll** нужно скопировать в папку с проектом.

В начало главного файла нужно добавить директиву компилятора **reference** и указать имя библиотеки:

```
{$reference 'PermuteUtils.dll'}
```

В раздел **uses** допишите пространство имён, в котором находится класс *PermUtils*:

```
uses
   System, PermuteUtils;
```

Теперь можно пользоваться библиотекой!

В **главном блоке** пользователь задаёт число элементов в коллекции:

```
//макс. число элементов в
//множестве:
const
   MAX_ELEM = 8;


begin
   //заголовок окна:
   Console.Title := 'Генерируем перестановки';
   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Перестановки:');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   //бесконечный цикл ввода данных -
   //пока пользователь не закроет программу
   //или не введёт 0:
   repeat
      //число элементов:
```

```
    var nElem := ReadInteger('Число элементов
                            (1..' + MAX_ELEM + ') > ');
    //если пользователь ввёл 0,
    //то программу закрываем:
    if (nElem = 0) then
      exit;
```

Затем мы создаём коллекцию (последовательность) чисел *1.. nElem*:

```
    //генерируем перестановки -->
    //список чисел:
    var a := Range(1, nElem);
```

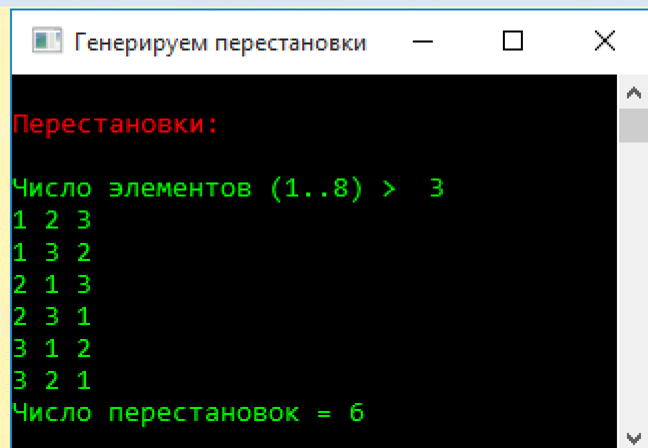
Вызываем метод **Permute**:

```
    //перестановки:
    var perms := PermUtils.Permute(a, a.Count);
```

И **печатаем** перестановки:

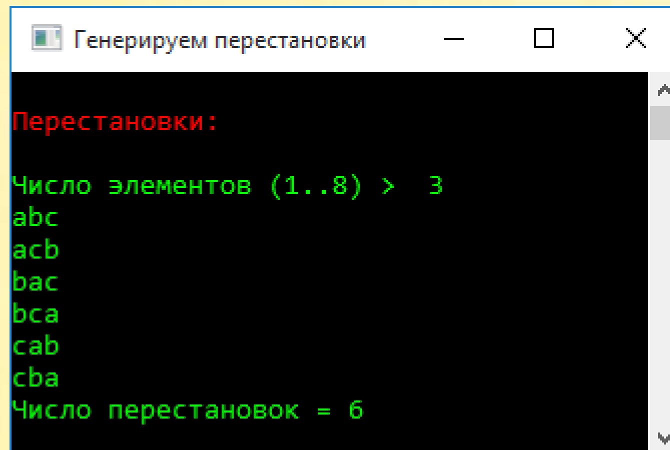
```
    foreach var p in perms do
      p.Println();

    Console.WriteLine('Число перестановок = ' + perms.Count());
    Console.WriteLine();
  until not (true);
end.
```

Запускаем программу и получаем все перестановки:



Элементы последовательности могут иметь и другой тип, например, *char*:

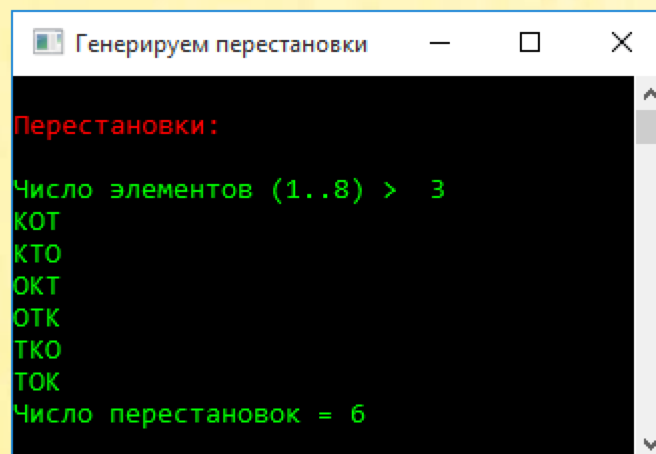
```
var a := Range('a', 'c');
```



В этом примере число элементов мы выбираем по числу элементов в коллекции, а не наоборот. В реальной программе задавать число элементов не нужно.

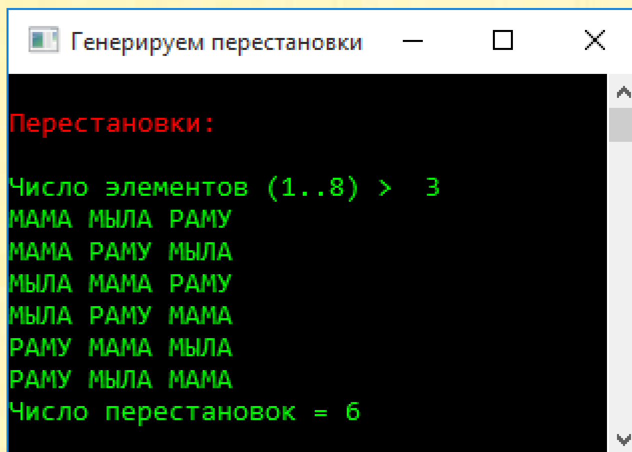
Если элементы коллекции не подчиняются какому-либо правилу, то их можно просто **перечислить** через запятую:

```
var a := Seq('К', 'О', 'Т');
```



Элементами коллекции могут быть не только отдельные буквы, но и целые **слова**:

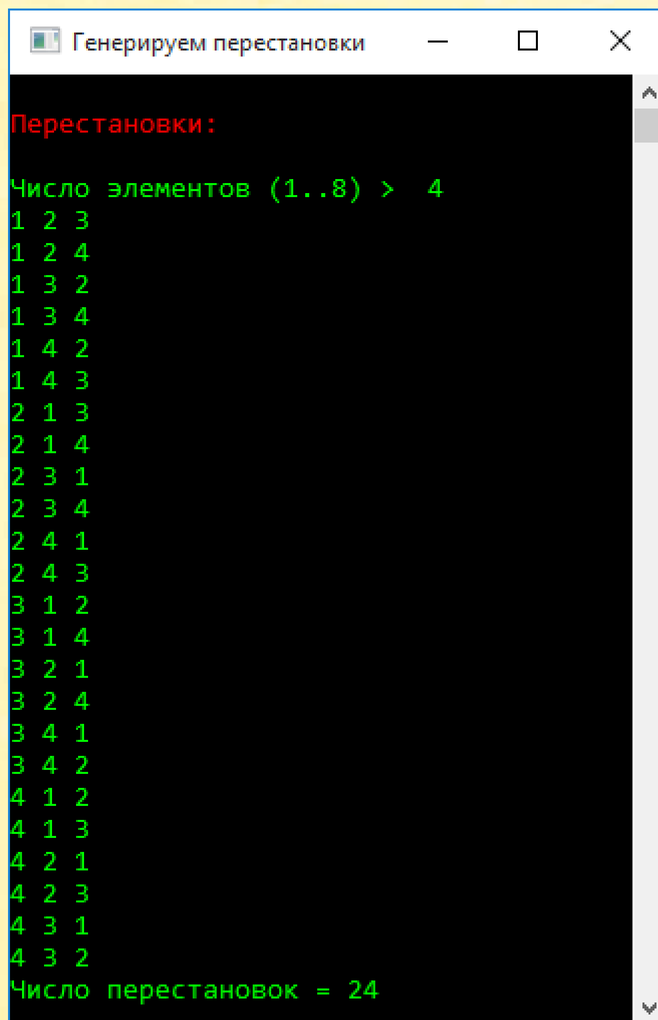
```
var a := Seq('МАМА', 'МЫЛА',
'РАМУ');
```



Эти примеры показывают, что метод *Permute* очень удобный и мощный.

Но это ещё не всё! Мы можем задать второму параметру значение, которое меньше числа элементов в коллекции и получить **размещения**:

```
    var a := Range(1, nElem);
    . . .
    //перестановки:
    var perms := Per-
mUtils.Permute(a, a.Count - 1);
```



Если в коллекции *n* элементов, а мы выбираем из них *k* элементов, то такой набор элементов называется **размещением из *n* по *k***. В размещениях, как и в перестановках важен порядок следования элементов.

Сочетания

Константы
Массив ***array of integer***
Бесконечный цикл ***repeat-until***
Метод ***ReadInteger***
Условный оператор ***if***
Оператор ***exit***
Функция *с параметрами*
Цикл ***while***
Оператор ***break***
Условный оператор ***if-else***
Процедура *без параметров*

Представим себе, что из 16 футболистов нам нужно выпустить на поле 11.

Общее число футболистов для комбинаторики - это число элементов в **множестве**, а число игроков в команде – число элементов в **подмножестве**. Таким образом, нам нужно найти **все подмножества заданного множества**. Каждое подмножество какого-либо множества иначе называют набором из заданного числа элементов, или **сочетанием**. В сочетании порядок элементов не играет роли, поэтому мы отберём только 11 игроков в команду, а как между ними поделить номера на футболках (а, значит, и их амплуа на поле), - это уже задача тренера.

Объявим глобальные константы и переменные программы **Сочетания**:

```
uses
   System;

//Сочетания

//ПРОГРАММА ДЛЯ ГЕНЕРИРОВАНИЯ
//ВСЕХ ПОДМНОЖЕСТВ k МНОЖЕСТВА ЧИСЕЛ n=1..nElem

const
   //макс. число элементов в
   //множестве:
```

```
   MAX_ELEM = 30;

var
   //число элементов в
   //подмножестве:
   k := 0;
   // массив, содержащий
   //очередное подмножество
   a := new integer[MAX_ELEM + 1];
   //номер подмножества:
   nSubset: integer;
```

В **главном блоке** пользователь должен ввести *два* числа. Важно учесть, что в подмножестве элементов *не больше*, чем в множестве:

```
begin
   //заголовок окна:
   Console.Title := 'Генерируем сочетания';
   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Сочетания:');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   //бесконечный цикл ввода данных -
   //пока пользователь не закроет программу
   //или не введёт 0:
   repeat
      //число элементов:
      var nElem := ReadInteger('Число элементов
                               (1..' + MAX_ELEM + ') > ');
      //если пользователь ввёл 0,
      //то программу закрываем:
      if (nElem = 0) then exit;

      //считываем число элементов подмножества:
      k := ReadInteger('Число элементов в подмножестве
                       (1..' + nElem + ') > ');

      if (k > nElem) then
      begin
         Console.WriteLine('Повторите ввод!');
```

```
         Console.WriteLine();
         continue;
      end;

      //генерируем все подмножества:
      var n := SubSet(nElem);
      Console.WriteLine('Число сочетаний = ' + n);

      Console.WriteLine();
   until not (true);
end.
```

Затем функция **SubSet** генерирует все подмножества:

```
//Генерируем все сочетания множества 1..nElem
//из k элементов
function SubSet(nElem: integer): integer;
begin
   var i := 0;
   for i := 1 to k do
      a[i] := i;
   nSubset := 0;
   var p := k;
   while (p >= 1) do
   begin
      //печатаем очередное подмножество:
      nSubset += 1;
      //if (nSubset > 20) then exit;
      WriteSubset();
      if (k = nElem) then break;
      if (a[k] = nElem) then p -= 1
      else p := k;

      if (p >= 1) then
         for i := k downto p do
            a[i] := a[p] + i - p + 1;
   end;
   Result := nSubset;
end;
```

Процедура **печати** очередного подмножества:

```
//ПЕЧАТАЕМ ОЧЕРЕДНУЮ ПЕРЕСТАНОВКУ
//ЭЛЕМЕНТОВ МНОЖЕСТВА
procedure WriteSubset();
begin
    var s := string.Empty;
    if (nSubset < 1000) then
        s += ' ';
    if (nSubset < 100) then
        s += ' ';
    if (nSubset < 10) then
        s += ' ';
    s += nSubset + '> ';
    for var i := 1 to k do
         s += a[i] + ' ';
    Console.WriteLine(s);
end;
```

Запускаем программу и вводим наши данные: 16 элементов в множестве и 11 – в подмножестве. Программа выдаёт огромный список из 4368 разных команд:

```
Генерируем сочетания

Сочетания:

Число элементов (1..30) >  16
Число элементов в подмножестве (1..16) >  11

   1> 1 2 3 4 5 6 7 8 9 10 11
   2> 1 2 3 4 5 6 7 8 9 10 12
   3> 1 2 3 4 5 6 7 8 9 10 13
   4> 1 2 3 4 5 6 7 8 9 10 14
   5> 1 2 3 4 5 6 7 8 9 10 15
   6> 1 2 3 4 5 6 7 8 9 10 16
   7> 1 2 3 4 5 6 7 8 9 11 12
   8> 1 2 3 4 5 6 7 8 9 11 13
   9> 1 2 3 4 5 6 7 8 9 11 14
  10> 1 2 3 4 5 6 7 8 9 11 15
  11> 1 2 3 4 5 6 7 8 9 11 16
  12> 1 2 3 4 5 6 7 8 9 12 13
  13> 1 2 3 4 5 6 7 8 9 12 14
  14> 1 2 3 4 5 6 7 8 9 12 15
  15> 1 2 3 4 5 6 7 8 9 12 16
  16> 1 2 3 4 5 6 7 8 9 13 14
  17> 1 2 3 4 5 6 7 8 9 13 15
  18> 1 2 3 4 5 6 7 8 9 13 16
  19> 1 2 3 4 5 6 7 8 9 14 15
  20> 1 2 3 4 5 6 7 8 9 14 16
  21> 1 2 3 4 5 6 7 8 9 15 16
  22> 1 2 3 4 5 6 7 8 10 11 12
  23> 1 2 3 4 5 6 7 8 10 11 13
  24> 1 2 3 4 5 6 7 8 10 11 14
```

```
Генерируем сочетания

4340> 4 5 7 8 9 10 11 13 14 15 16
4341> 4 5 7 8 9 10 12 13 14 15 16
4342> 4 5 7 8 9 11 12 13 14 15 16
4343> 4 5 7 8 10 11 12 13 14 15 16
4344> 4 5 7 9 10 11 12 13 14 15 16
4345> 4 5 8 9 10 11 12 13 14 15 16
4346> 4 6 7 8 9 10 11 12 13 14 15
4347> 4 6 7 8 9 10 11 12 13 14 16
4348> 4 6 7 8 9 10 11 12 13 15 16
4349> 4 6 7 8 9 10 11 12 14 15 16
4350> 4 6 7 8 9 10 11 13 14 15 16
4351> 4 6 7 8 9 10 12 13 14 15 16
4352> 4 6 7 8 9 11 12 13 14 15 16
4353> 4 6 7 8 10 11 12 13 14 15 16
4354> 4 6 7 9 10 11 12 13 14 15 16
4355> 4 6 8 9 10 11 12 13 14 15 16
4356> 4 7 8 9 10 11 12 13 14 15 16
4357> 5 6 7 8 9 10 11 12 13 14 15
4358> 5 6 7 8 9 10 11 12 13 14 16
4359> 5 6 7 8 9 10 11 12 13 15 16
4360> 5 6 7 8 9 10 11 12 14 15 16
4361> 5 6 7 8 9 10 11 13 14 15 16
4362> 5 6 7 8 9 10 12 13 14 15 16
4363> 5 6 7 8 9 11 12 13 14 15 16
4364> 5 6 7 8 10 11 12 13 14 15 16
4365> 5 6 7 9 10 11 12 13 14 15 16
4366> 5 6 8 9 10 11 12 13 14 15 16
4367> 5 7 8 9 10 11 12 13 14 15 16
4368> 6 7 8 9 10 11 12 13 14 15 16
Число сочетаний = 4368
```

Комбинаторную задачу мы решили, а вот какая из этих команд действительно станет командой мечты, тут комбинаторика бессильна!

Иногда полезно заранее знать, сколько же получится сочетаний при тех или иных исходных данных - не печатать же весь список подмножеств, если нам интересно узнать только их число! На этот случай в комбинаторике имеется несложная **формула**:

$$C_n^k = \frac{n!}{k!(n-k)!}$$

> В нашей программе *n = nElem*.

Как видите, в комбинаторике без факториала не обойтись!

Спортлото

Константы
*Бесконечный цикл **repeat-until***
*Метод **ReadInteger***
*Условный оператор **if***
*Тип данных **int64***
***Функция** с параметрами*
*Оператор **break***
*Условный оператор **if-else***
*Цикл **for***
*Оператор **exit***

В книге Эрхарда Берендса (Ehrhard Behrends) *Fünf Minuten Mathematik: 100 Beiträge der Mathematik-Kolumne der Zeitung DIE WELT* [BE13], на страницах 1-3 предлагаются **вероятностные** задачи, которые, как мы сейчас увидим, очень близки к задачам комбинаторным.

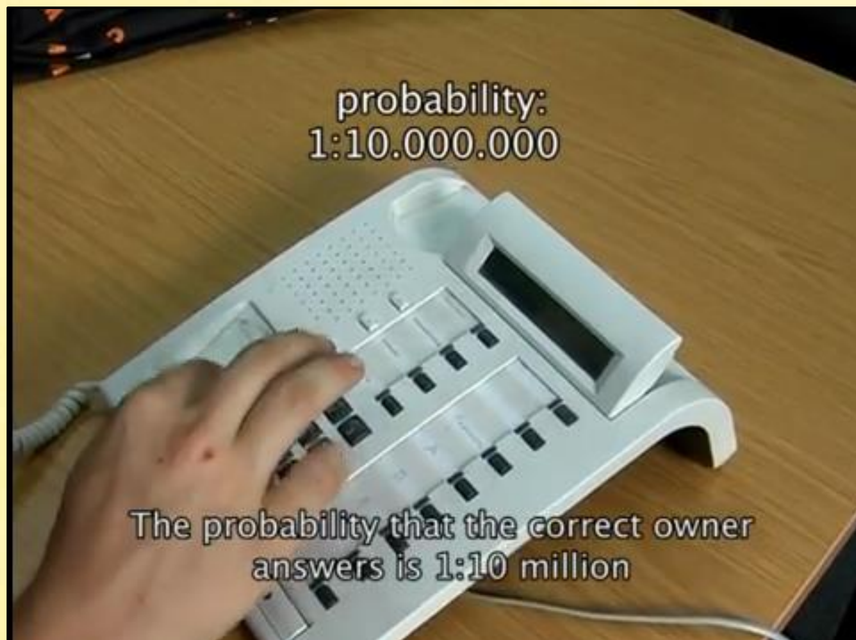
В некоем большом городе, например, в Берлине или в Гамбурге кто-то забыл в автобусе зонтик. Предположим, что все телефонные номера в этих городах состоят из 7 цифр. Какова вероятность того, что вы (а это ведь вы нашли тот забытый зонтик!), случайно набирая номер, попадёте на того самого бюргера, который потерял зонтик?

Скорее всего, не все возможные комбинации из 7 цифр могут быть телефонными номерами, но мы предположим, что это так и есть. Тогда общее число номеров равно 10^7 = 10 000 000. Если вы позвоните 1 раз, то вероятность составит 1 / 10 000 000. В общем случае вероятность можно вычислить как отношение числа благоприятных случаев (исходов) к их общему числу.

По адресу

http://www.youtube.com/watch?v=KA-gN1h15Ko

вы можете посмотреть ролик о поиске владельца потерянного зонтика:



В Германии, точно так же как и в России, многие годы играют в лото 6 из 49:

В книге утверждается, что при покупке 1 билета вы имеете всего 1 шанс из **13 983 816** угадать все 6 номеров. Это даже меньше, чем найти незадачливого владельца зонтика в большом городе. Однако вряд ли кто-нибудь станет звонить по телефону в надежде попасть на него, а вот в лото играют многие и охотно. И это легко понять: выиграть чужое гораздо приятнее, чем его вернуть.

Но давайте проверим, правильно ли подсчитал вероятность автор книги.

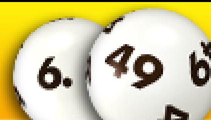
Клонируйте проект *Сочетания* и сохраните его в новой папке под названием **Спортлото**.

Поскольку 6 номеров из 49 можно выбрать многими способами, то распечатывать их нет никакого смысла.

А вот сохранить все сочетания в массиве смысл есть. Действительно, шарики выпадают из барабана совершенно случайно, поэтому **все** комбинации совершенно равноправны, и это значит, что вам не нужно ломать голову над составлением числовых комбинаций – выбирайте случайно любую из массива и смело зачёркивайте числа! Если учесть, что в каждом билете может быть до 12 комбинаций, то таким нехитрым способом вы быстро избавитесь от головной боли:

LOTTO 6 aus 49

Spielen Sie hier staatliches Lotto online.



1 2 3 4 5 6
7 8 9 10 11 12

Zufallsfelder +1 +2 +6 +12 Felder löschen Mein Glücksschein

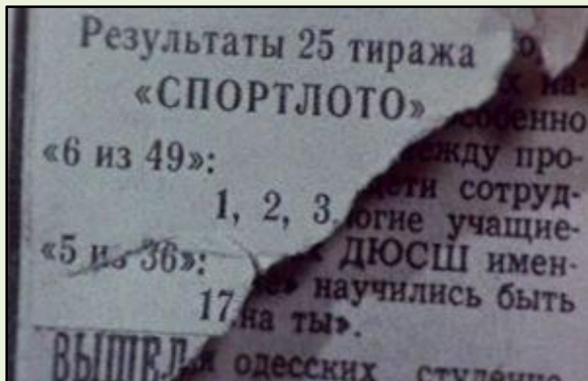
Glücksspirale | Glücksspirale
6 9 1 7 6 8 7 | Scheinnr. ändern
Spiel 77 | Spiel 77
Super 6 | Super 6
Superzahl

Mi/Sa Mi Sa
Wochenlaufzeit
1 2 3 4
Dauerschein (Info)
Schein vordatieren (Info)

Jackpot-Jäger: Spielen Sie diesen Schein weiter, bis der Jackpot geknackt ist. (Info)

Хотя, надо признать, этот способ был бы хорош, если бы только вы играли в лото. Но кроме вас, ещё несколько миллионов человек надеются на удачу, поэтому даже в случае выигрыша вам придётся делиться с такими же везунчиками, как и вы. Чтобы избежать дележа, нужно выбирать «редкие», «маловероятные» комбинации чисел, чтобы никто другой их не выбрал. Как вы помните, героиня комедии *Спортлото-82* Таня поступила именно так, зачеркнув 6 первых чисел:

И выиграла:



Таким образом, наша задача упрощается – мы уже не будем распечатывать сочетания, а ограничимся только их **подсчётом**:

```
uses
   System;

//Спортлото

//ПРОГРАММА ДЛЯ ПОДСЧЁТА
//ВСЕХ ПОДМНОЖЕСТВ k МНОЖЕСТВА ЧИСЕЛ n=1..nElem

//макс. число элементов в
//множестве:
const
   MAX_ELEM = 49;
. . .

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

   //бесконечный цикл ввода данных -
   //пока пользователь не закроет программу
```

```
   //или не введёт 0:
   repeat
      //число элементов:
      var nElem := ReadInteger('Число элементов
                              (1..' + MAX_ELEM + ') > ');
      //если пользователь ввёл 0,
      //то программу закрываем:
      if (nElem = 0) then
         exit;

      //считываем число элементов подмножества:
      k := ReadInteger('Число элементов в подмножестве
                      (1..' + nElem + ') > ');

      if (k > nElem) then
      begin
         Console.WriteLine('Повторите ввод!');
         Console.WriteLine();
         continue;
      end;

      //генерируем все подмножества:
      var n := SubSet(nElem);
      Console.WriteLine();
      Console.WriteLine('Число сочетаний = ' + n);
      Console.WriteLine();
      Console.WriteLine();
   until not (true);
end.
```

Для этого вполне годится готовая функция **SubSet** из нашего предыдущего проекта, в котором достаточно закомментировать всего строку:

```
//Генерируем все сочетания множества 1..nElem
//из k элементов
function SubSet(nElem: integer): integer;
begin
   var i := 0;
   for i := 1 to k do
      a[i] := i;
   nSubset := 0;
   var p := k;
```

```
    while (p >= 1) do
    begin
       //печатаем очередное подмножество:
       nSubset += 1;
       //WriteSubset();
       if (k = nElem) then break;
       if (a[k] = nElem) then p -= 1
       else p := k;

       if (p >= 1) then
          for i := k downto p do
             a[i] := a[p] + i - p + 1;
    end;
    Result := nSubset;
end;
```

Запускаем программу – и убеждаемся в правоте автора: **6 чисел из 49 можно выбрать 13 983 816 способами**:

```
Подсчитываем сочетания

Подсчитываем сочетания:

Число элементов (1..49) >  49
Число элементов в подмножестве (1..49) >  6
Число сочетаний = 13983816
```

Если не нужно сохранять комбинации чисел в массиве, то можно найти число сочетаний по формуле, которая нам известна из предыдущего проекта:

$$C_n^k = \frac{n!}{k!(n-k)!}$$

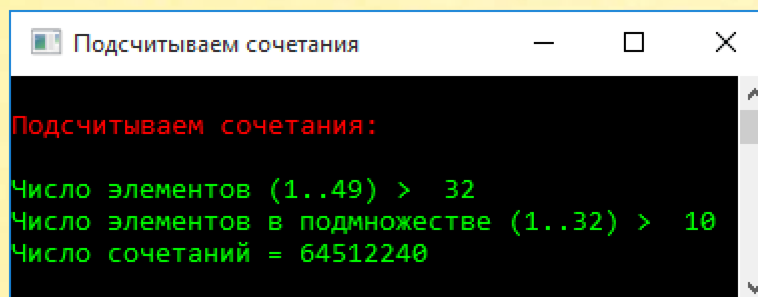
К сожалению, для подсчётов она непригодна, так как в числителе стоит огромный факториал. Но нетрудно догадаться, что первые *(n-k)* чисел этого факториала сократятся с этими же числами в знаменателе, то есть числитель равен произведению чисел от *(n-k+1)* до *n*, а в знаменателе останется только факториал числа *k*. Изловчившись, мы легко напишем новую **функцию** для подсчёта сочетаний:

```
//ВЫЧИСЛЯЕМ ФАКТОРИАЛ
function Factorial(num: integer): int64;
begin
   var fact: int64 := 1;

   for var i := 1 to num do
      fact *= i;

   Result := fact;
end;

function GetNumSubSets(n, k: integer): int64;
begin
   var chisl: int64 := 1;
   for var i := n downto n - k + 1 do
      chisl *= i;
   var znam := Factorial(k);
   Result := chisl div znam;
end;
```

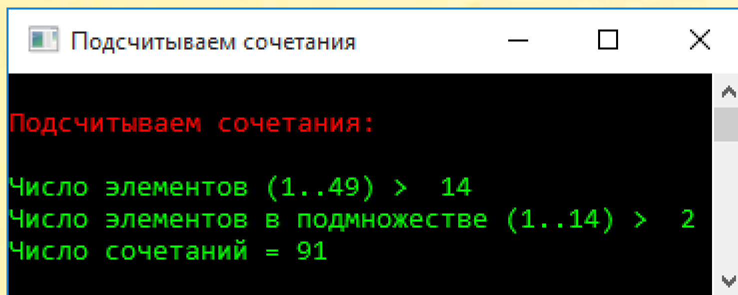
Вызываем функцию **GetNumSubSets** в главном блоке и получаем тот же результат, что и раньше:

```
      //генерируем все подмножества:
      {var n := SubSet(nElem);
       Console.WriteLine();
      Console.WriteLine('Число сочетаний = ' + n);
      Console.WriteLine();  }

      var ln := GetNumSubSets(nElem, k);
      Console.WriteLine('Число сочетаний = ' + ln);
```

Аналогично можно узнать число раскладов в игре ***скат***:

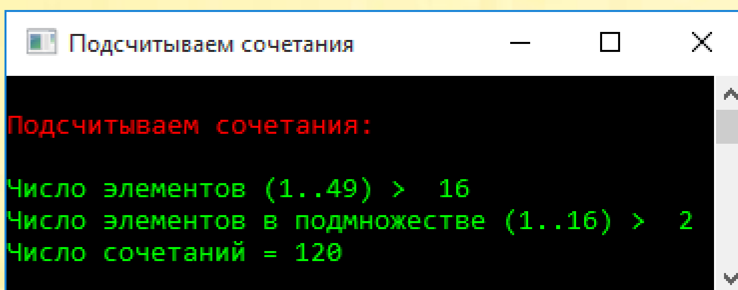


Или число **рукопожатий**, которыми обменяются 14 человек, расходясь после праздника (примеры из книги):



В Чемпионате России по футболу участвуют 16 команд. **Сколько всего матчей они проведут между собой?**

Задача решается аналогично предыдущей, но следует учесть, что каждая пара команд сыграет **дважды** – дома и на выезде. Поэтому команды проведут не 120, а 120 x 2 = 240 матчей.



В современной России играют также в *Гослото 6 из 45* и *5 из 36*:

Подсчитаем число вариантов и для них:

```
Подсчитываем сочетания

Подсчитываем сочетания:

Число элементов (1..49) >  45
Число элементов в подмножестве (1..45) >  6
Число сочетаний = 8145060

Число элементов (1..49) >  36
Число элементов в подмножестве (1..36) >  5
Число сочетаний = 376992
```

По сравнению с иноземными лотереями, российские более обнадёживающие!

В *Спортлото* играют и в Италии – но по другим правилам. В одном из вариантов нужно угадать *5 чисел из 90*. Легко подсчитать, что вероятность такого радостного события равна **1 / 43 949 268**:

```
Подсчитываем сочетания

Подсчитываем сочетания:

Число элементов (1..90) >  90
Число элементов в подмножестве (1..90) >  5
Число сочетаний = 43949268
```

Не забудьте поправить **константу**:

```
MAX_ELEM = 90;
```

В варианте **SuperEnaLotto:**

нужно угадать уже *6 чисел из 90*, что уменьшает шансы игроков до **1 / 622 614 630**:

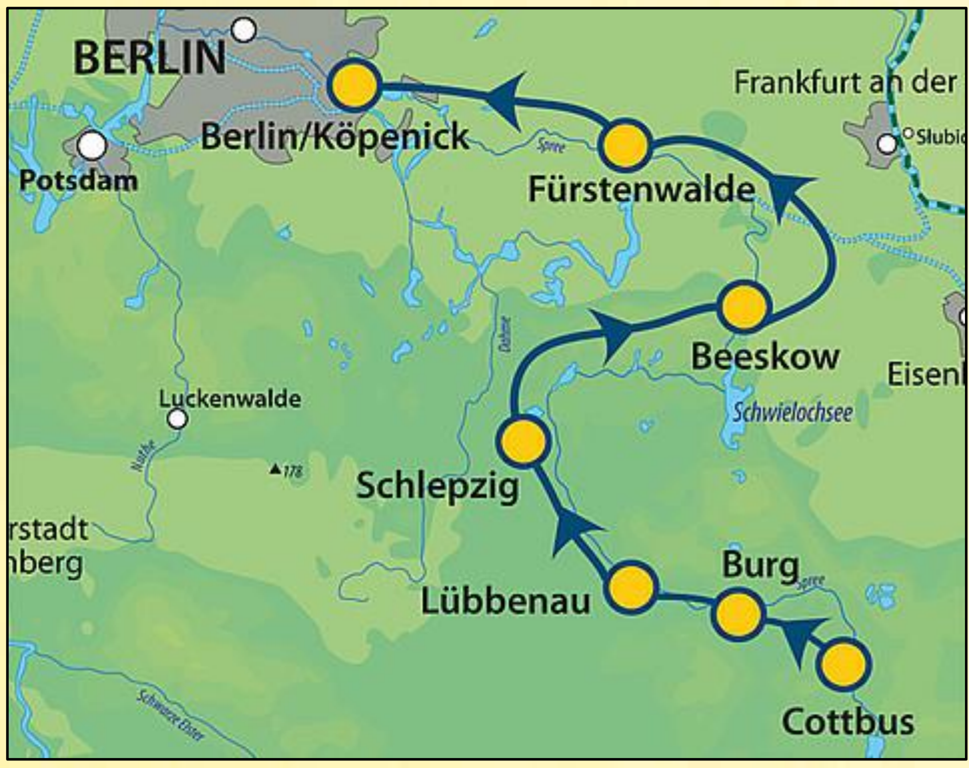
```
Подсчитываем сочетания

Подсчитываем сочетания:

Число элементов (1..90) >  90
Число элементов в подмножестве (1..90) >  6
Число сочетаний = 622614630
```

«Нет, я так не играю!» - сказал бы Карлсон.

В итальянском лото все неразыгранные деньги переходят в следующий тираж. А поскольку вероятность выигрыша главного приза невелика, то постепенно джек-пот может достигать огромных значений – до 100 миллионов евро:

И тогда из соседних стран в Италию движутся караваны автобусов с жаждущими наживы джентльменами удачи. Вот так, сравнительно дёшево можно развивать туризм и познавательный интерес к своей стране…

Чтобы показать, насколько мала вероятность выигрыша в немецкое *Спортлото,* Эрхард Берендс прибегает к такому наглядному примеру.



Расстояние между Берлином и Котбусом составляет около 140 км или в сантиметрах – 14 000 000. Как вы помните, практически столько же разных карточек можно заполнить в немецком лото.

Теперь представьте, что вы едете на автомобиле по этой дороге с завязанными глазами (опасаясь побочных эффектов и последствий, автор книги уточняет: вас **везут** по этой дороге). Где-то на обочине стоит столб диаметром в 1 см. Вы в любой момент времени можете бросить одноцентовую монету в надежде попасть в этот столб (или шест):

Вероятность этого события приблизительно равна вероятности выигрыша в немецкое лото.

Для итальянского лото *SuperEnaLotto* потребуется дорога в 6000 километров: из Рима через Берлин в Москву и обратно.

По адресу

http://www.youtube.com/watch?v=ODwm29lIt0E

вы можете посмотреть ролик с экспериментом по бросанию монеты из автомобиля в шест:

Palindrome

В книге *Gehirnjogging mit 300 Rätseln und Denksportaufgaben*:

четвёртая задача вполне может вызвать наше любопытство:

Phileas Fogg hat auf seiner Weltreise, die in London begann, bisher 15951 km zurückgelegt. Er erkennt, dass diese Zahl ein Palindrom ist, das heißt vor- und rückwärts gelesen dieselbe Zahl ergibt, und beschlißt, auf die nächste derartige Zagl zu acten. Nach 2 Tagen ist es so weit, die Gesamtstrecke ist wieder ein Palindrom. Wie weit ist er in dieser Zeit gekommen?

Про Филеаса Фогга и его кругосветное путешествие знает каждый, кто читал роман Жюля Верна *80 дней вокруг света*. По этой причине я не буду пересказывать вам условие задачи, а говорю коротко: *нам нужно найти следующий за числом 15951 палиндром*.

В **главном блоке** вызываем процедуру *Solve* для решения задачи:

```
begin
   //заголовок окна:
   Console.Title := 'Gehirnjogging. Задача 4';
   Console.WriteLine();
   Console.ForegroundColor := ConsoleColor.Red;
   Console.WriteLine('Решаем задачу Palindrome');
   Console.ForegroundColor := ConsoleColor.Green;
   Console.WriteLine();

   Solve();
   Console.Read();
end.
```

Алгоритм решения задачи очень простой.

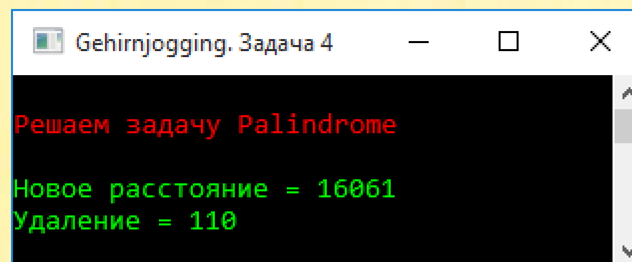
В процедуре *Solve* мы добавляем по 1 км к пройденному расстоянию, начиная с 15951, и проверяем каждое новое число на палиндромность в функции *IsPalindrome*. Ей нужно передать массив цифр заданного числа, который мы получаем от функции *GetDigits*:

```
function GetLenNum(num: integer): integer;
begin
   num := Math.Abs(num);
   var len := Trunc(Math.Log10(num)) + 1;
   Result := len;
end;

function GetDigits(num: integer): array of integer;
begin
   var len := GetLenNum(num);
   var digits := new integer[len];
   var i := 0;
   while (num > 0) do
   begin
      digits[i] := num mod 10;
      num := num div 10;
      i += 1;
   end;
   Result := digits;
end;
```

Гораздо проще проверить число на палинромность, превратив его в строку, но этот способ не совсем честный.

```
//РЕШАЕМ ЗАДАЧУ
procedure Solve();
begin
   //пройденное расстояние:
   var dist0 := 15951;
   //новое расстояние:
   var dist := dist0;
   var flg := false;
   while (not flg) do
   begin
      dist += 1;
      var digits := GetDigits(dist);
      if (IsPalindrome(digits)) then
      begin
         Console.WriteLine('Новое расстояние = '
                            + dist.ToString());
         Console.WriteLine('Удаление = '
                            + (dist - dist0).ToString());
         flg := true;
      end
   end;
   Console.WriteLine();
end;
```

Запускаем и программу и узнаём, что через 110 км Филеаса Фогга пройдёт 16061 км, а это и есть следующее число-палиндром после 15951:



Арифметический генератор

В книге *Увлекательная математика на паскале* мы решили несколько задачек про кроликов и фазанов. Например, такую:

> *У фазанов и кроликов 62 ноги и 19 голов.*
>
> ***Сколько фазанов и кроликов?***

И такую:

> В клетке находятся фазаны и кролики. Известно, что у них 35 голов и 94 ноги. Узнайте число фазанов и число кроликов.

В *Интернете*, в задачниках и книгах с головоломками вы найдёте ещё немало задачек про этих милых животных.

Новым проектом мы разом убьём и кроликов, и фазанов.

Допустим, что в задаче фигурируют от 1 до 100 кроликов и от 1 до 100 фазанов. Диапазоны изменения числа домашних животных задаются элементами управления udRMin, udRMax, udPMin и udPMax. Первая пара отвечает за численность поголовья кроликов, вторая – фазанов. Нижнюю границу численности уменьшать не рекомендуется, потому что это приведёт к вырождению задачи, а вот верхнюю вы можете поднимать сколько угодно высоко.

Из заданного диапазона мы случайным образом выбираем число четырёхногих кроликов и двуногих фазанов, после чего подсчитываем у них число ног и голов. Отсюда вытекают **поля** класса формы:

```
type
  int = integer;

implementation
var
    //общее число голов:
    heads : int;
```

```
    //общее число ног:
    legs : int;
    //число кроликов:
    numRabbits : int;
    //число фазанов:
    numPheasants : int;
    rand := new Random();
```

Принцип составления задач понятен, поэтому мы нажимаем на кнопку **btnGen** *ГЕНЕРИРОВАТЬ!* и начинаем составлять конкретную задачу:

```
procedure Form1.btnGen_Click(sender: Object; e: EventArgs);
begin
    txtRes.Clear();
    var minRabbits := int(udRMin.Value);
    var maxRabbits := int(udRMax.Value);
    if (maxRabbits < minRabbits) then
    begin
        rtxSource.AppendText('Проверьте число кроликов!');
        exit;
    end;
    var minPheasants := int(udPMin.Value);
    var maxPheasants := int(udPMax.Value);
    if (maxPheasants < minPheasants) then
    begin
        rtxSource.AppendText('Проверьте число фазанов!');
        exit;
    end;
    //число кроликов и фазанов:
    numRabbits := rand.Next(minRabbits, maxRabbits + 1);
    numPheasants := rand.Next(minPheasants, maxPheasants + 1);
    //общее число голов:
    heads := numRabbits + numPheasants;
    //общее число ног:
    legs := 4 * numRabbits + 2 * numPheasants;
```

Проверки необязательны: вы можете просто поменять местами неверные значения минимального и максимального числа кроликов или фазанов. Теперь в условие задачи нужно подставить полученные значения – и задача готова!

*У фазанов и кроликов **legs** ноги и **heads** голов.*

Сколько фазанов и кроликов?

Но тут нас ждёт разочарование со стороны русского языка: при некоторых значениях числа ног и голов получается несогласованность окончаний у ног и голов. Например, так:

*У фазанов и кроликов **466** ноги и **144** голов.*
Сколько фазанов и кроликов?

Дальше следует код, согласующий ноги и головы с правилами русского языка (надеюсь, я ничего упустил):

```
    //1 21 31 голова
    //2..4 22..24 32..34  головы
    //5..20  голов
    var mh := heads mod 10;

    var endh := 0;
    if (mh mod 10 <> 11) and (mh = 1) then
        endh := 1
    else if (mh mod 100 <> 12) and
            ((mh = 2) or (mh = 3) or (mh = 4)) then
        endh := 2
    else
        endh := 3;

    var ml := legs mod 10;
    var endl := 0;
    if (ml mod 10 <> 11) and (ml = 1) then
        endl := 1
    else if (ml mod 10 <> 12) and ((ml = 2) or (ml = 3)
             or (ml = 4)) then
        endl := 2
    else
        endl := 3;
    var sheads := String.Empty;
    var slegs := String.Empty;
    case endh of
        1: sheads := 'голова';
        2: sheads := 'головы';
    else sheads := 'голов';
```

```
    end;
    case endl of
        1: slegs := 'нога';
        2: slegs := 'ноги';
    else slegs := 'ног';
    end;
    var s := String.Format('У фазанов и кроликов {0} {1} и {2} {3}.',
                                  legs, slegs, heads, sheads );
    rtxSource.AppendText(s + NewLine + 'Сколько фазанов и кроликов?');
    Solve();
    rtxSource.AppendText(NewLine + NewLine);
end;
```

Как видите, эта часть метода превзошла и по длине, и по сложности собственно составление математической задачи – настолько могуч русский язык!

Зато теперь вы можете сочинять задачи про кроликов и фазанов в любых количествах:

Генератор задач

Задача:

У фазанов и кроликов 488 ног и 155 голов.
Сколько фазанов и кроликов?

У фазанов и кроликов 380 ног и 141 голова.
Сколько фазанов и кроликов?

У фазанов и кроликов 136 ног и 36 голов.
Сколько фазанов и кроликов?

Ответ:

Кроликов: 89
Фазанов: 66

Кроликов: 49
Фазанов: 92

Кроликов: 32
Фазанов: 4

ГЕНЕРИРОВАТЬ! СТЕРЕТЬ СТЕРЕТЬ

Кролики: 1 100
Фазаны: 1 100

Кроликов: 32
Фазанов: 4

ПРАВИЛЬНО!

ПРОВЕРИТЬ!

На всякий случай мы проверяем решение в методе **Solve** и печатаем ответ в текстовом окне *rtxTarget*.

```
//РЕШАЕМ ЗАДАЧУ
procedure Form1.Solve();
begin
    //макс. число кроликов:
    var maxRabbits := legs div 4;

    //решаем задачу:
    for var i := 0 to maxRabbits do
    begin
                //число фазанов:
        var nPheasants := heads - i;
        if (i * 4 + nPheasants * 2 = legs) then
            rtxTarget.AppendText('Кроликов: ' + i +
                                 NewLine + 'Фазанов: ' + nPheasants);
    end;
    rtxTarget.AppendText(NewLine + NewLine);
end;
```

Если вы немного прищуритесь, чтобы не заглянуть в ответ раньше времени, то сможете самостоятельно решить задачу и выставить в элементах управления **udR** и **udP** найденные своими руками поголовья кроликов и фазанов. Кнопка **btnTest** *ПРОВЕРИТЬ!* сообщит вам в текстовом поле **txtRes** своё заключение по вашему решению:

```
// ПРОВЕРЯЕМ РЕШЕНИЕ
procedure Form1.btnTest_Click(sender: Object; e: EventArgs);
begin
    var nR := int(udR.Value);
    if (nR <> numRabbits) then
    begin
        txtRes.Text := 'Проверьте число кроликов!';
        exit;
    end;

    var nP := int(udP.Value);
    if (nP <> numPheasants) then
    begin
```

```
        txtRes.Text := 'Проверьте число фазанов!';
        exit;
    end;
    txtRes.Text := 'ПРАВИЛЬНО!';
end;
```

Генератор задач

Задача:

У фазанов и кроликов 348 ног и 108 голов.
Сколько фазанов и кроликов?

У фазанов и кроликов 336 ног и 98 голов.
Сколько фазанов и кроликов?

У фазанов и кроликов 102 ноги и 45 голов.
Сколько фазанов и кроликов?

Ответ:

Кроликов: 66
Фазанов: 42

Кроликов: 70
Фазанов: 28

Кроликов: 6
Фазанов: 39

ГЕНЕРИРОВАТЬ! СТЕРЕТЬ СТЕРЕТЬ

Кролики: 1 100

Фазаны: 1 100

Кроликов: 7

Фазанов: 39

Проверьте число кроликов!

ПРОВЕРИТЬ!

Задания для самостоятельного решения

Простые числа

1. Напишите функцию, которая определяет, являются ли несколько чисел **взаимно простыми**.

2. Найдите **четвёрки** простых чисел, принадлежащих одному десятку. Например, 11, 13, 17, 19. **[ЗП88**, Задача 558**]**.

3. Натуральное число называется **полусовершенным**, если оно равно сумме *всех* или *некоторых* своих делителей, исключая само число: 6, 12, 18, 20, 24, 28, 30, 36, 40. Например, 12 = 1 + 2 + 3 + 6 или 12 = 2 + 4 + 6.

Из этого определения следует, что всякое совершенное число является и полусовершенным, то есть полусовершенных чисел в заданном диапазоне не меньше, чем совершенных. Напишите программу, которая находила бы несколько полусовершенных чисел.

4. Два (различных) натуральных числа называются **дружественными**, если сумма всех делителей (исключая само число) первого числа равна второму числу, и наоборот. Первая пара дружественных чисел была найдена несколько тысячелетий тому назад. Это числа 220 и 284. Следующая пара отыскалась только в 1860 году – 1184 и 1210. Сейчас известно несколько миллионов дружественных чисел. Напишите программу для их поиска **[ЗП88**, Задача 560**]**. Учитывайте, что числа в паре либо оба *чётные*, либо оба *нечётные*.

Формула Бине

[ЗП88]. Задача 556

n-ное число Фибоначчи можно вычислить по **формуле Бине**:

$$F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}} = \frac{\varphi^n - (-\varphi)^{-n}}{\varphi - (-\varphi)^{-1}} = \frac{\varphi^n - (-\varphi)^{-n}}{2\varphi - 1}$$

В ней буквой *фи* обозначено *золотое сечение*: $\varphi = \frac{1+\sqrt{5}}{2}$.

Из неё следует, что F_n равно ближайшему к $\frac{\varphi^n}{\sqrt{5}}$ целому числу.

Так как абсолютная величина выражения $\frac{1-\sqrt{5}}{2}$ меньше единицы, то при больших *n* числа Фибоначчи можно вычислять по *приближённой* формуле:

$$F_n \sim \frac{\varphi^n}{\sqrt{5}}$$

Напишите процедуру, вычисляющую заданное число Фибоначчи по формуле Бине.

Числовые ребусы

В журнале *Квантик* вы найдёте еще немало числовых ребусов. А можете и не искать, потому что все они представлены ниже.

2012 №3, с.26

5 (*8 баллов*). Замените в равенстве

ПИРОГ = КУСОК + КУСОК + КУСОК + . . . + КУСОК

одинаковые буквы одинаковыми цифрами, а разные – разными так, чтобы равенство было верным, а количество «кусков пирога» было бы наибольшим из возможных.

И. В. Раскина

2012 №3, с.32

12. Расшифруйте ребус:

КВАН : ТИК = 4 : 3

(Каждая буква заменена какой-то цифрой, одинаковые буквы (то есть две буквы К) заменены одинаковыми цифрами, а разные – разными.)

2012 №4, с.35

19. Расшифруйте ребус:

КВАН – ТИК = КВА – НТ

(Каждая буква заменена какой-то цифрой, одинаковые буквы заменены одинаковыми цифрами, а разные – разными.)

2012 №5, с.17. Задача 9

ЗАДАЧКА 9

НЕДЕЛЯ – ЭТО СЕМЬ ДНЕЙ

Замените в этом равенстве одинаковые буквы одинаковыми цифрами, а разные буквы – разными так, чтобы полученное равенство оказалось верным.

ДЕНЬ × 7 = Н × Е × Д × Е × Л × Я

2012 №9, с.35

44. Решите ребус :

ОДИН + ОДИН = МНОГО

(Напоминаем, что одинаковыми буквами зашифрованы одинаковые цифры, а разными – разные.)

2013 №2, с.35

4. В слове **КВАНТИК** каждую букву заменили некоторой цифрой. Причём одинаковые буквы то есть (две буквы К) были заменены одинаковыми цифрами, а разные – разными. При этом оказалось, что выполняется следующее равенство:

Найдите, при каких значениях букв это возможно.

2013 №4, с.13

1.2. Расшифруйте ребус (одинаковые буквы означают одинаковые цифры, разные – разные):

Б+БЕЕЕ=МУУУ.

2014 №6, с.25

6 (9 баллов). Известный преступник профессор Мориарти долго скрывался от Шерлока Холмса и лондонской полиции. И вот однажды полицейским удалось перехватить телеграмму, которую Мориарти прислал сообщнику:

Встречай завтра поезд СТО вагон О

Инспектор Лестрейд уже распорядился было послать наряд полиции искать нулевой вагон сотого поезда, но тут принесли ещё две перехваченные телеграммы на тот же адрес:

СЕКРЕТ – ОТКРОЙ = ОТВЕТ – ТВОЙ

СЕКРЕТ – ОТКРЫТ = 20010.

Лестрейд задумался. А Холмс воскликнул: «Теперь ясно, какой поезд надо встречать!» Инспектор удивился. «Элементарно, Лестрейд! – пояснил сыщик. – Это же шифр. В этих примерах одинаковые буквы обозначают одинаковые цифры, разные – разные, а чёрточка – это минус! Мориарти едет в поезде № ...»

Напишите номер поезда и вагона. Объясните, как мог рассуждать Холмс.

Инесса Раскина

2014 №10, с.35

50. Решите ребус: МАТЕ × М = АТИКА. (Как обычно, одинаковыми буквами обозначены одинаковые цифры, а разными – разные.)

Париж-анин

1. Напишите программу или процедуру для поиска длинных слов, в которых короткие слова находятся в **середине** (*слово в слове*).

2. Поищите слова-антигуингнгмы, в которых роль согласных исполняют **гласные**.

3. Поищите слова, в которых больше всего согласных, независимо от их места в слове. Например, в 7-буквенном слове *всплеск* всего 1 гласная и 6 согласных!

Литература

[Нагибин88]	
	Нагибин Ф.Ф., Канин Е.С. Математическая шкатулка М.: Просвещение, 1988. – 160 с.

[100]	
	В. А. Дагене, Г. К. Григас, К. Ф. Аугутис 100 задач по программированию М.:Просвещение, 1993. – 251 с. ISBN: 5-09-003864-3

[КА86] [КА96]

Б.А. Кордемский, А.А.Ахадов

Удивительный мир чисел
(МАТЕМАТИЧЕСКИЕ ГОЛОВОЛОМКИ
И ЗАДАЧИ ДЛЯ ЛЮБОЗНАТЕЛЬНЫХ)

М.: Просвещение, 1986. – 144 с.
М.: Просвещение, 1996. – 159 с

[ОО80]

Оре О.

Приглашение в теорию чисел

М.: Наука, 1980 г. - 128 с.

Библиотечка *Квант*, Выпуск 3

[ЗП88]

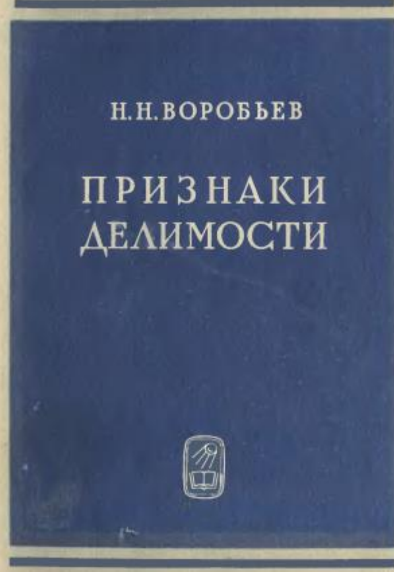
Абрамов С.А. и др.

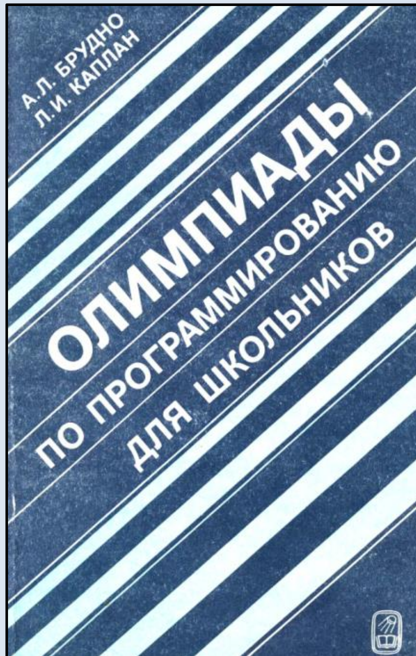
Задачи по программированию

Наука, 1988. – 224 с.

ISBN: 5-02-013774-X

Серия: Библиотечка программиста

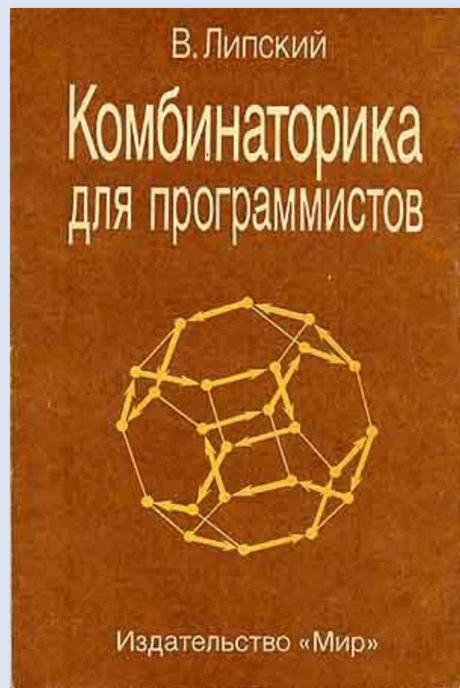
[ВНН88]	
 	Воробьёв Н.Н. **Признаки делимости** Наука. - 1988, 96 с. ISBN 5-02-013731-6

[БК85]	
 	Брудно А. Л. Каплан Л. И. **Олимпиады по программированию для школьников** Наука. - 1985, 96 с.

[BE13]	
	Ehrhard Behrends Fünf Minuten Mathematik: 100 Beiträge der Mathematik-Kolumne der Zeitung DIE WELT Springer Spektrum. - 2013, 272 с. 3-е издание ISBN: 978-3-658-00998-4

[ГМ72]	
	Гарднер Мартин Математические досуги М.: Мир, 1972. – 495 с.

[ЛВ88]

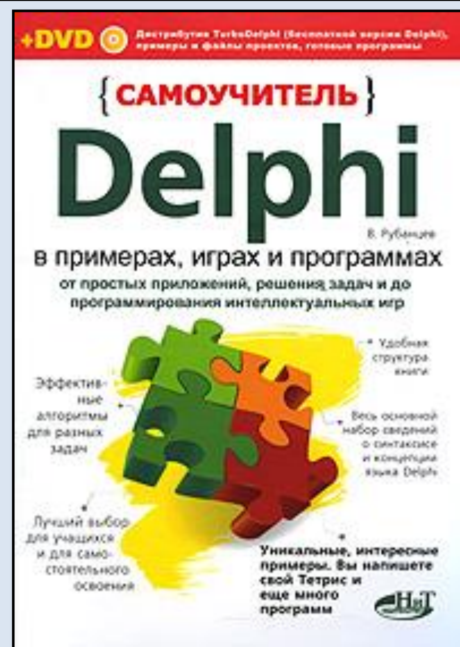


Липский В.

Комбинаторика для программистов

Москва: Мир, 1988. – 200 с.

[РВ11]



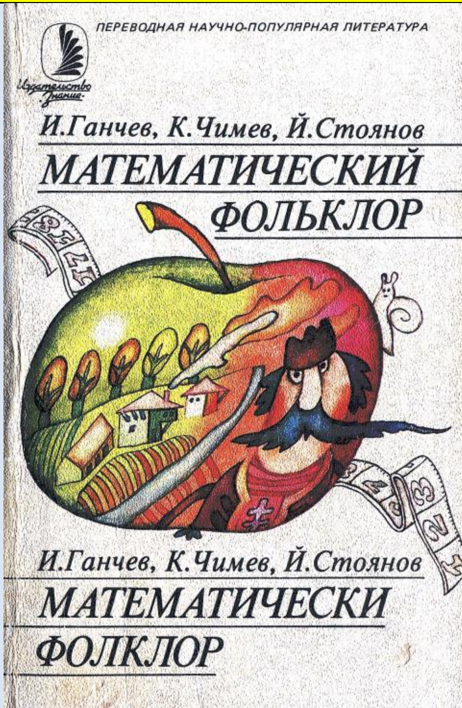
Рубанцев Валерий

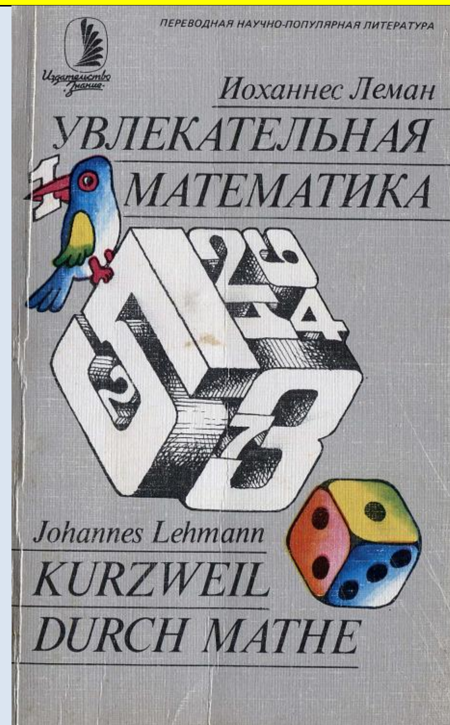
Delphi в примерах, играх и программах. От простых приложений, решения задач и до программирования интеллектуальных игр

Наука и Техника, 2011. – 672 с.

ISBN: 978-5-94387-664-6

Серия: Самоучитель

[Фольклор]	
	Ганчев, Чимев, Стоянов Математический фольклор Знание. Москва, 1985. – 208 с.

[Увлекательная математика]	
	Иоханнес Леман Увлекательная математика Знание. Москва, 1985. – 272 с.

[ХР92]	
	Росс Хонсбергер Математические изюминки Наука, 1992. - 176 с. Библиотечка «Квант». Вып. 83 ISBN 5-02-014406-1