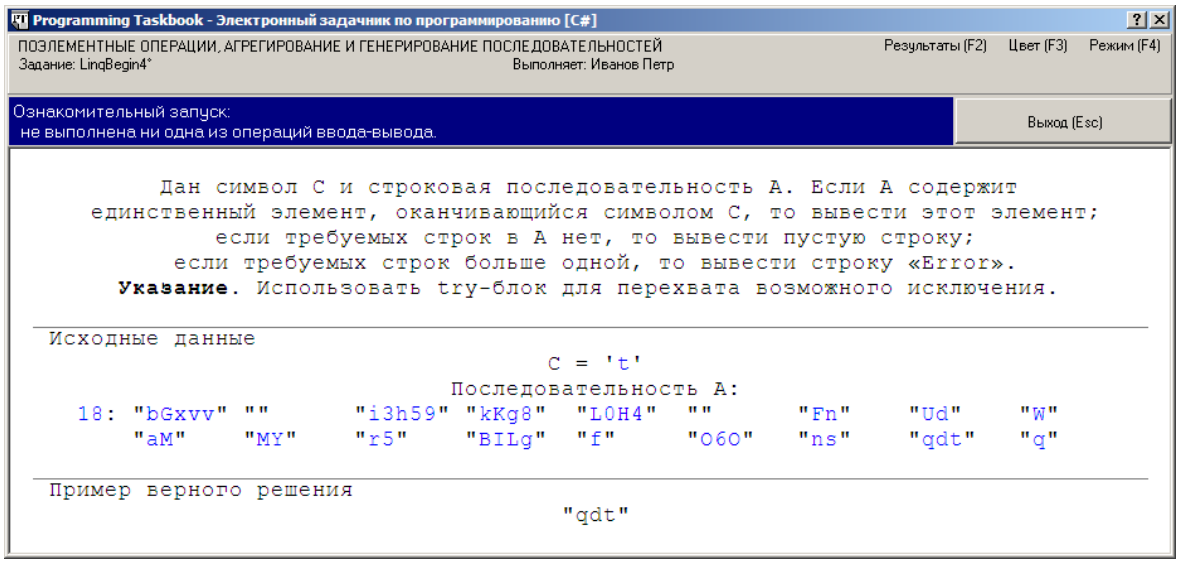
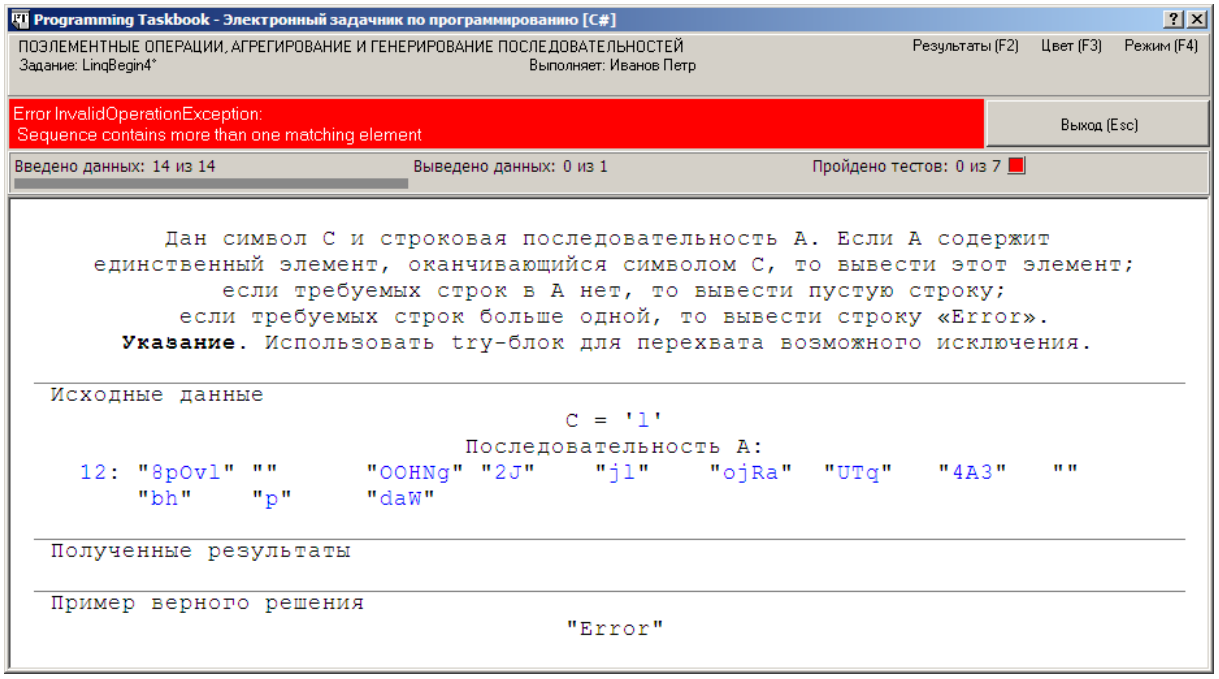
Поэлементные операции: LinqBegin4




```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using PT4;

namespace PT4Tasks
{
  public class MyTask : PT
  {

    // При решении задач группы LinqBegin доступны следующие
    // дополнительные методы, определенные в задачнике:
    //
    //   GetEnumerableInt() - ввод числовой последовательности;
    //
    //   GetEnumerableString() - ввод строковой последовательности;
    //
    //   Put() (метод расширения) - вывод последовательности;
    //
    //   Show() и Show(cmt) (методы расширения) - отладочная печать
    //     последовательности, cmt - строковый комментарий;
    //
    //   Show(e => r) и Show(cmt, e => r) (методы расширения) -
    //     отладочная печать значений r, полученных из элементов e
    //     последовательности, cmt - строковый комментарий.

    public static void Solve()
    {
      Task("LinqBegin4");

    }
```

```
public static void Solve()
{
  Task("LinqBegin4");
  char c = GetChar();
  var a = GetEnumerableString();
  Put(a.Single(e => e.Length != 0 && e[e.Length - 1] == c));
}
```



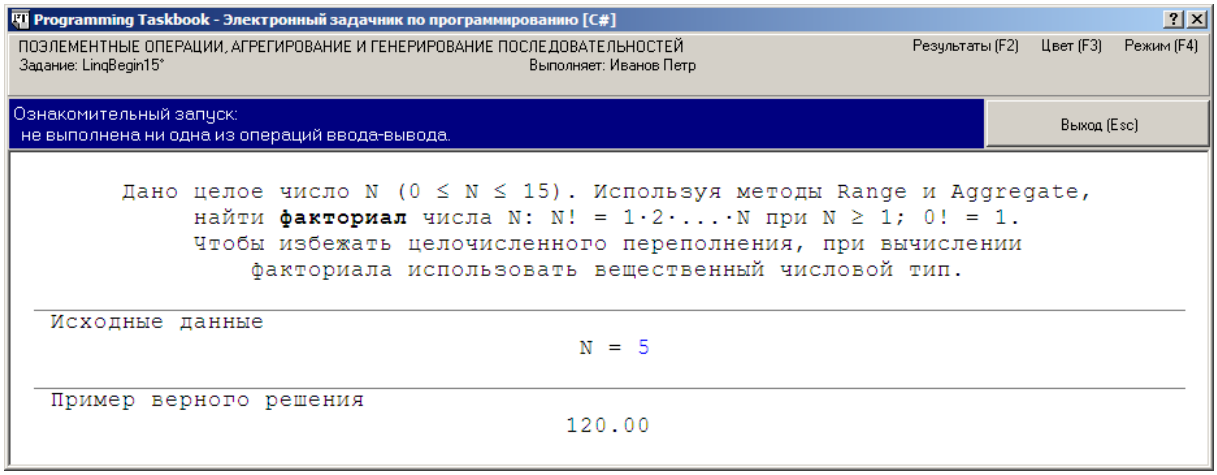

```
public static void Solve()
{
  Task("LinqBegin4");
  char c = GetChar();
  var a = GetEnumerableString();
  try
  {
    Put(a.Single(e => e.Length != 0 && e[e.Length - 1] == c));
  }
  catch (InvalidOperationException ex)
  {
    if (ex.Message.Contains("more"))
      Put("Error");
    else
      Put("");
  }
}
```

```
public static void Solve()
{
  Task("LinqBegin4");
  char c = GetChar();
  var a = GetEnumerableString();
  try
  {
    string r = a.SingleOrDefault(e => e.Length != 0 &&
      e[e.Length - 1] == c);
    Put(r != null ? r : "");
  }
  catch
  {
    Put("Error");
  }
}
```

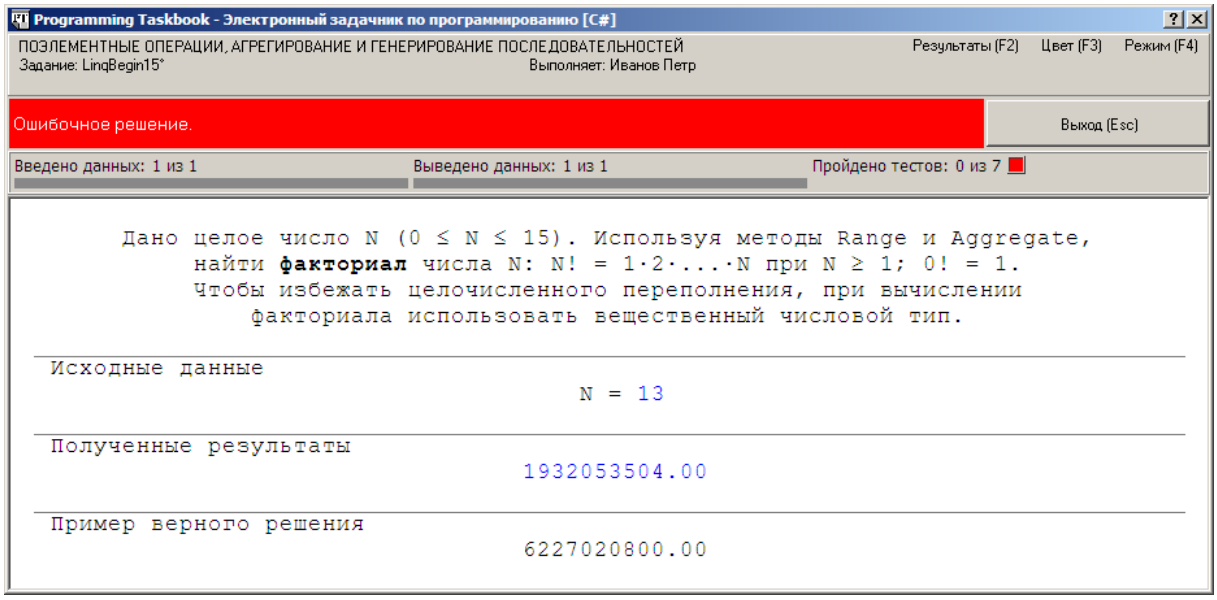
```
public static void Solve()
{
  Task("LinqBegin4");
  char c = GetChar();
  var a = GetEnumerableString();
  try
  {
    Put(a.SingleOrDefault(e => e.Length != 0 &&
      e[e.Length - 1] == c) ?? "");
  }
  catch
  {
    Put("Error");
  }
}
```

```
public static void Solve()
{
  Task("LinqBegin4");
  char c = GetChar();
  try
  {
    Put(GetEnumerableString().SingleOrDefault(e =>
      e.Length != 0 && e[e.Length - 1] == c) ?? "");
  }
  . . .
```

Операция агрегирования и генерирование последовательностей: LinqBegin15




```
public static void Solve()
{
  Task("LinqBegin15");
  int n = GetInt();
  double f = Enumerable.Range(1, n).Aggregate(1, (a, e) => a * e);
  Put(f);
}
```



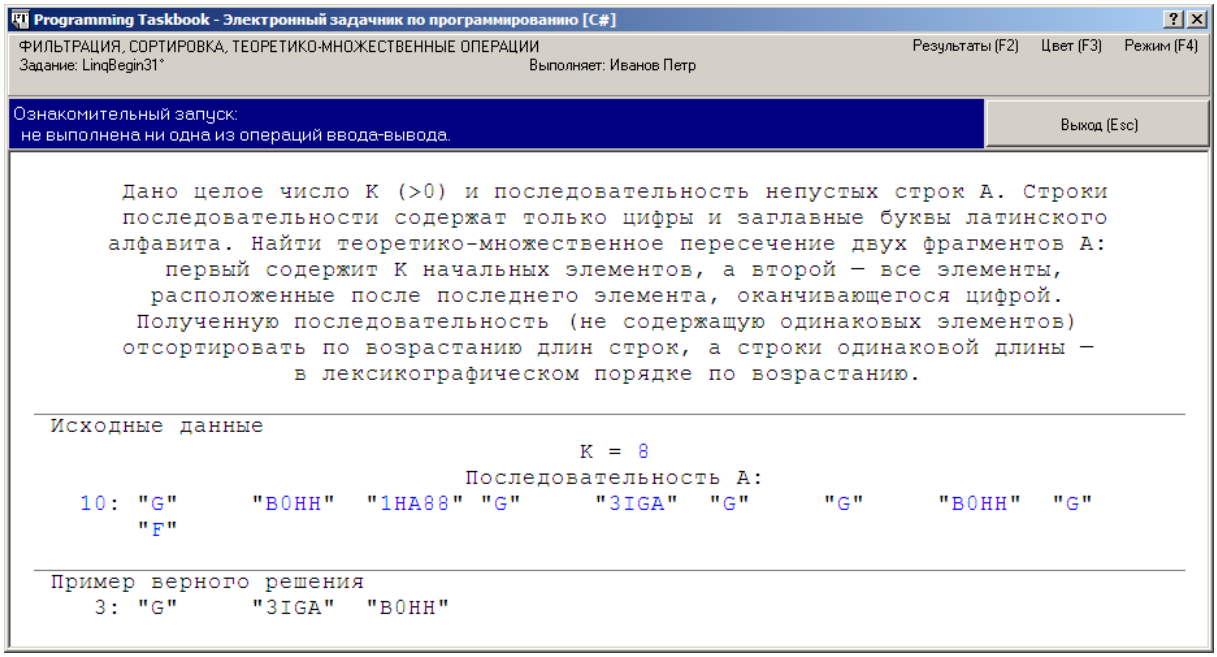

```
double f = Enumerable.Range(1, n)
  .Aggregate(1, (double a, int e) => a * e);
```

```
double f = Enumerable.Range(1, n).Aggregate(1.0, (a, e) => a * e);
```

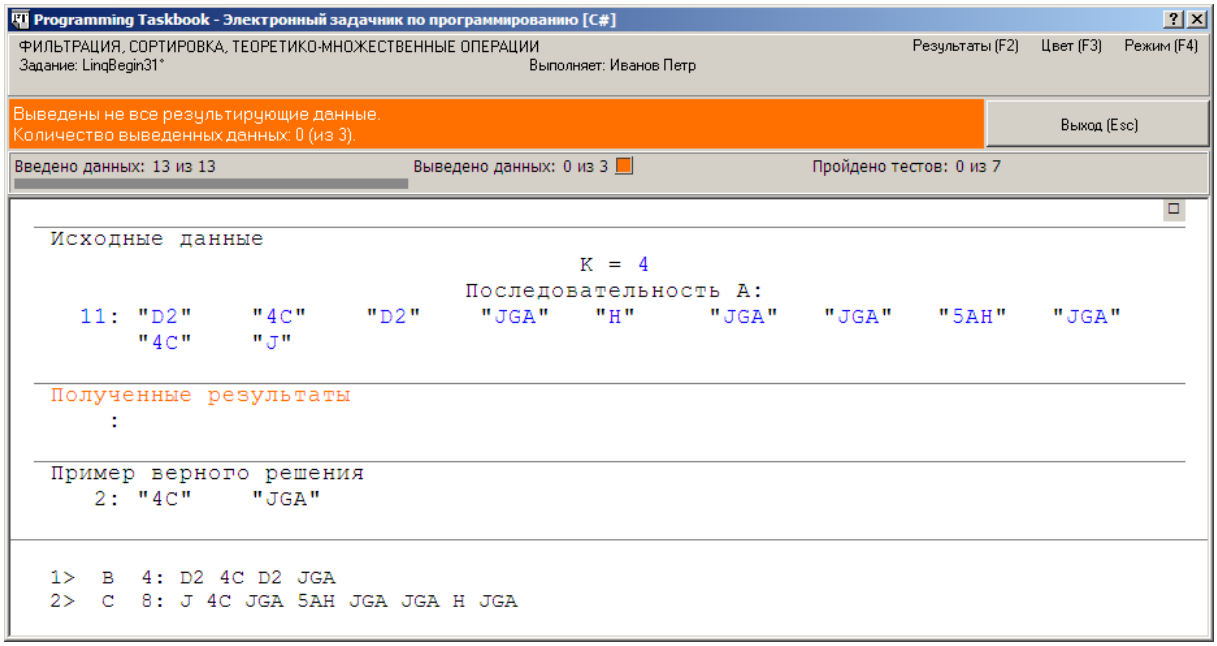
```
public static void Solve()
{
  Task("LinqBegin15");
  Put(Enumerable.Range(1, GetInt())
    .Aggregate(1.0, (a, e) => a * e));
}
```

Примечание. При анализе полученного алгоритма может возникнуть желание сократить размер исходной последовательности, генерируемой методом Range, удалив из нее первый элемент (со значением 1). Для этого достаточно изменить список параметров метода Range следующим образом: `(2, GetInt() - 1)`. Однако при тестировании данного варианта обнаружится, что он неправильно работает для случая $N = 0$, поскольку в этом случае второй параметр принимает недопустимое *отрицательное* значение.

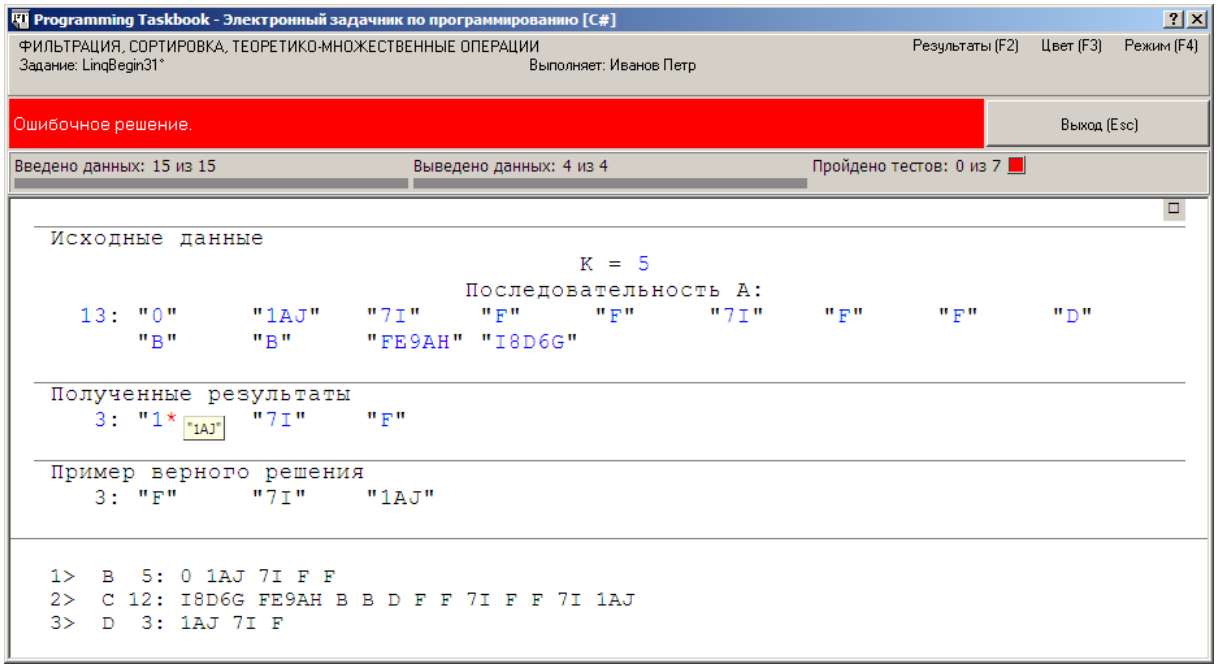
Фильтрация, сортировка, теоретико-множественные операции: LinqBegin31




```
int k = GetInt();
var a = GetEnumerableString();
var b = a.Take(k);
var c = a.Reverse()
  .TakeWhile(e => !char.IsDigit(e[e.Length - 1]));
b.Show("B");
c.Show("C");
```



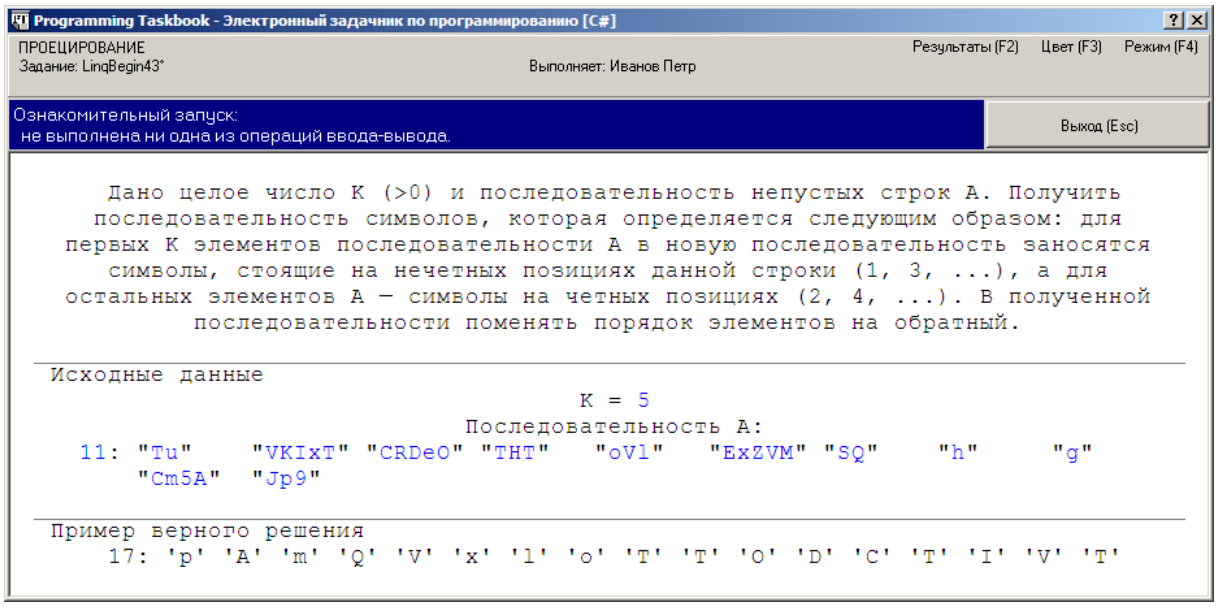

```
var b = a.Take(k).Show("B");
var c = a.Reverse()
  .TakeWhile(e => !char.IsDigit(e[e.Length - 1])).Show("C");
b.Intersect(c).Show("D").Put();
```




```
b.Intersect(c).Show("D")
  .OrderBy(e => e.Length).ThenBy(e => e)
  .Put();
```

```
public static void Solve()
{
  Task("LinqBegin31");
  int k = GetInt();
  var a = GetEnumerableString();
  a.Take(k).Show("B")
    .Intersect(a.Reverse()
      .TakeWhile(e => !char.IsDigit(e[e.Length - 1])).Show("C"))
    .Show("D")
    .OrderBy(e => e.Length).ThenBy(e => e)
    .Put();
}
```

Проецирование: LinqBegin43



Tu VKIxT CRDeO THT oVl ExZVM SQ h g Cm5A Jp9

Для первых K элементов:

```
e.Where((c, i) => i % 2 == 0)
```

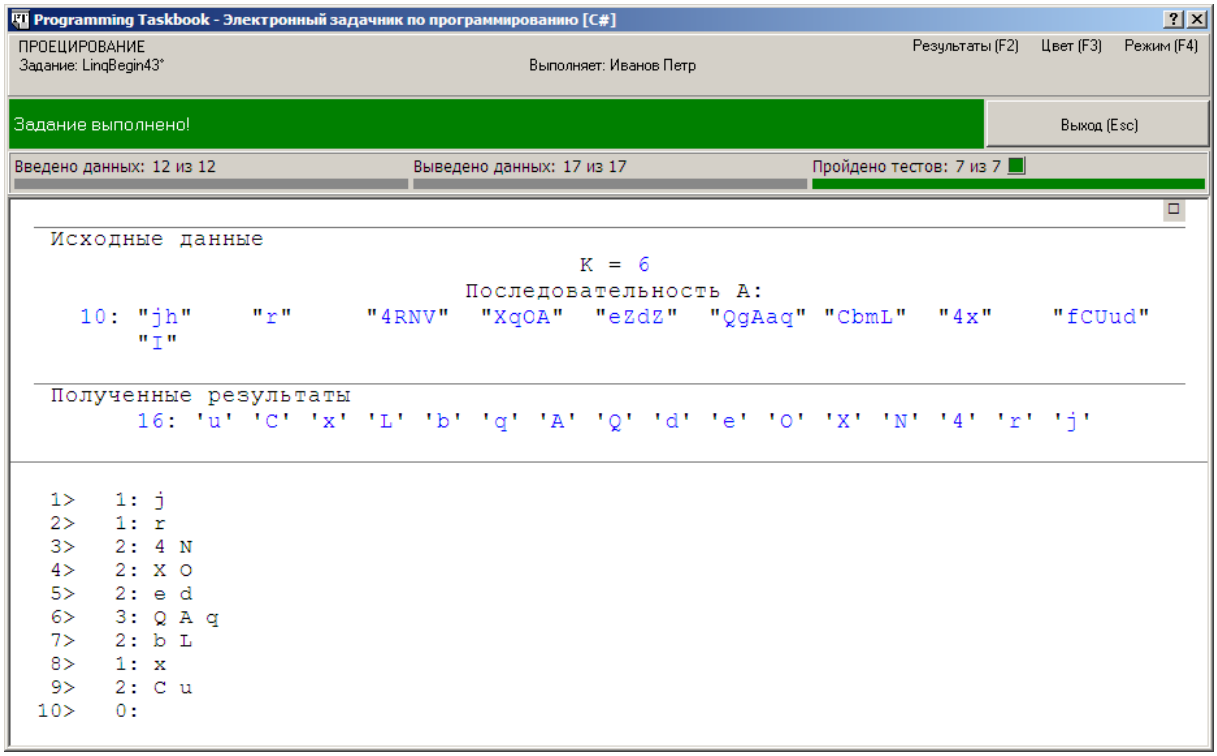
Для остальных элементов:

```
e.Where((c, i) => i % 2 == 1)
```

Для элемента с индексом n:

```
e.Where((c, i) => i % 2 == (n < k ? 0 : 1))
```

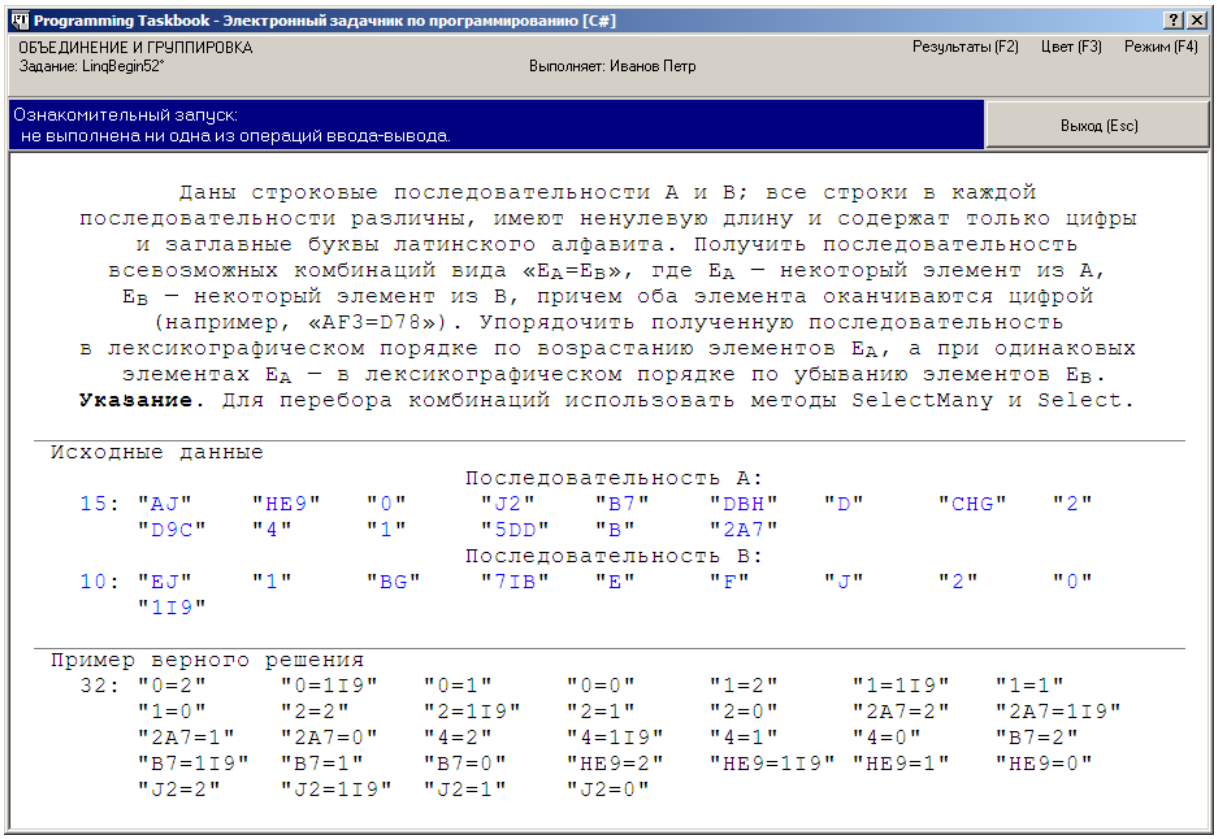
```
public static void Solve()
{
  Task("LinqBegin43");
  int k = GetInt();
  GetEnumerableString()
    .SelectMany((e, n) =>
       e.Where((c, i) => i % 2 == (n < k ? 0 : 1)).Show())
    .Reverse().Put();
}
```



Примечание. Если метод Show вызывается для последовательности, формируемой в каком-либо лямбда-выражении, то возможна ситуация, когда при выполнении программы окно отладки не появится на экране. Данная ситуация связана с особенностью обработки запросов LINQ, возвращающих последовательности. Подобные запросы являются *отложенными*, т. е. выполняются не в момент их появления в программе, а лишь впоследствии, когда к полученной последовательности будет применена одна из операций, выполняющихся немедленно (например, поэлементная операция, операция агрегирования или операция экспортирования), или когда будет организован цикл по перебору элементов этой последовательности (подобные циклы организуются, в частности, в методах Show и Put). Если ни одно из указанных действий в программе не выполняется, то не будут выполнены и отложенные запросы, а значит, и входящие в их лямбда-выражения методы Show. Для моделирования подобной ситуации достаточно удалить в предыдущей программе вызов метода Put.

```
public static void Solve()
{
  Task("LinqBegin43");
  int k = GetInt();
  var a = GetEnumerableString();
  a.Take(k).SelectMany(e => e.Where((c, i) => i % 2 == 0))
    .Concat(a.Skip(k)
      .SelectMany(e => e.Where((c, i) => i % 2 == 1)))
    .Reverse().Put();
}
```

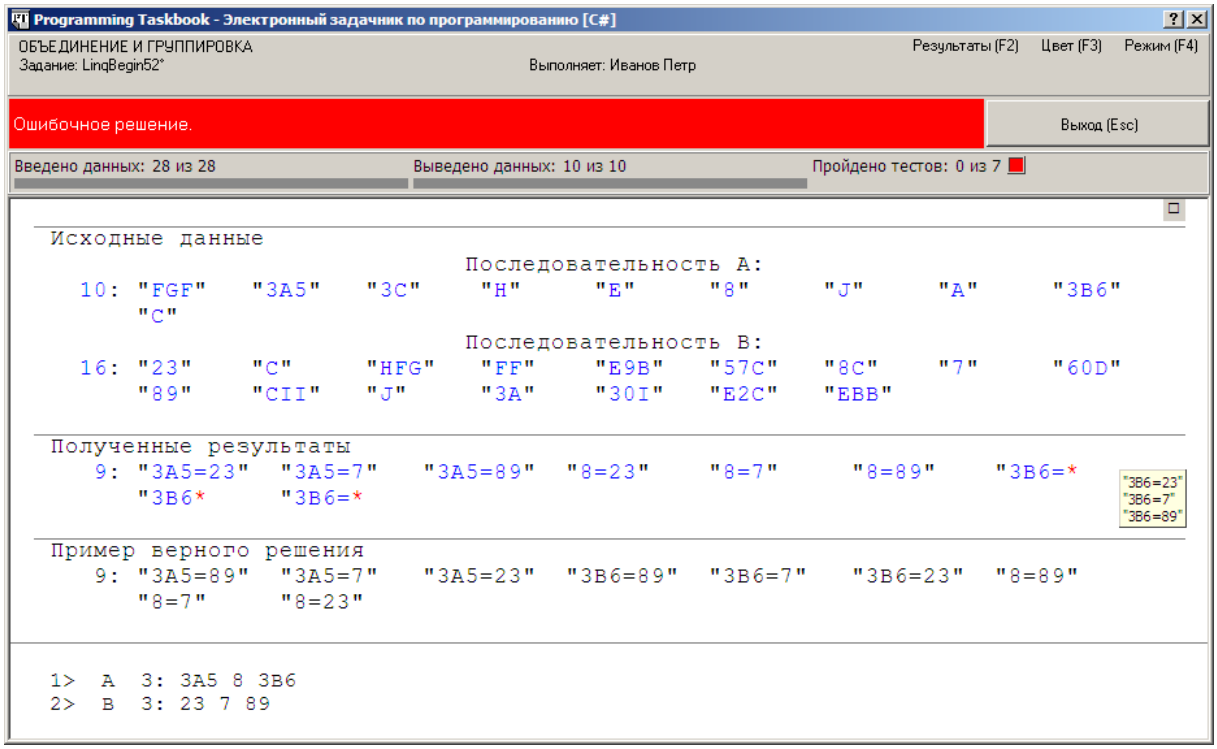
Объединение: LinqBegin52, LinqBegin54




```
var a = GetEnumerableString()
  .Where(e => char.IsDigit(e[e.Length - 1])).Show("A");
var b = GetEnumerableString()
  .Where(e => char.IsDigit(e[e.Length - 1])).Show("B");
```

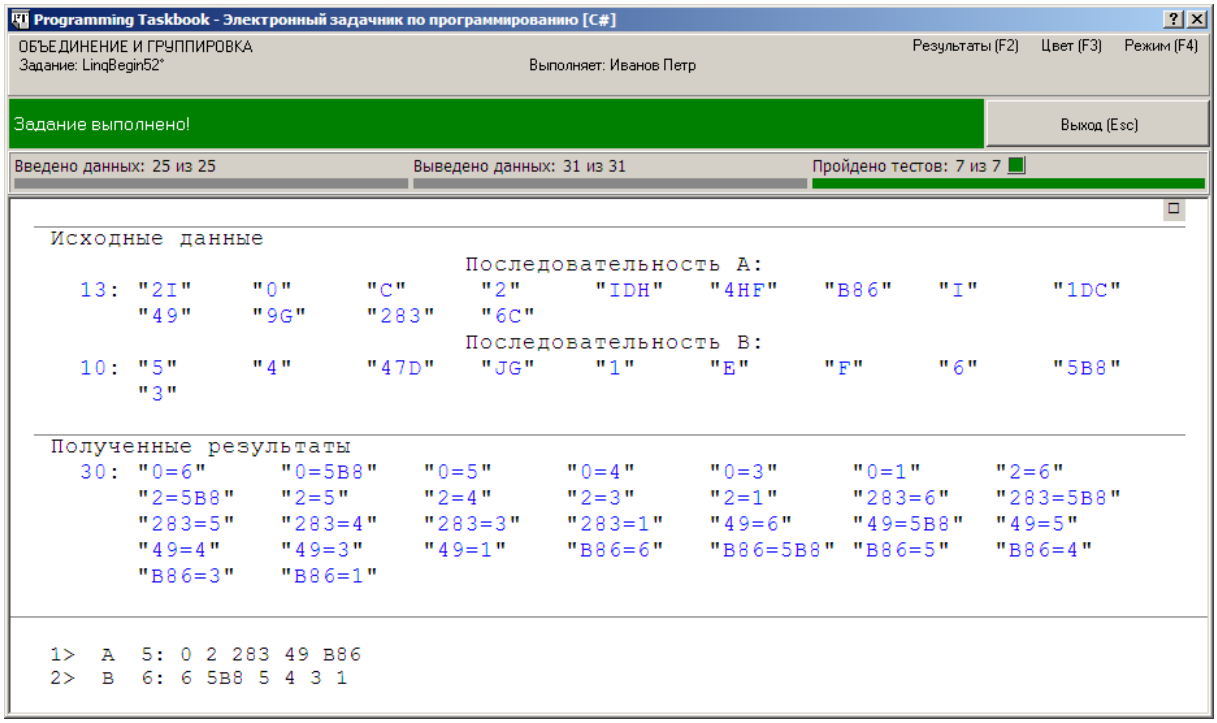
Для доступа к последнему символу строки `e` вместо выражения `e[e.Length - 1]` можно использовать запрос LINQ `e.Last()` (при этом строка `e` будет интерпретироваться как *последовательность символов*).

```
a.SelectMany(e1 => b.Select(e2 => e1 + "=" + e2)).Put();
```




```
a.SelectMany(e1 => b.Select(e2 => e1 + "=" + e2))
  .OrderBy(e => e.Split('=')[0])
  .ThenByDescending(e => e.Split('=')[1])
  .Put();
```

```
public static void Solve()
{
  Task("LinqBegin52");
  var a = GetEnumerableString()
    .Where(e => char.IsDigit(e[e.Length - 1]))
    .OrderBy(e => e).Show("A");
  var b = GetEnumerableString()
    .Where(e => char.IsDigit(e[e.Length - 1]))
    .OrderByDescending(e => e).Show("B");
  a.SelectMany(e1 => b.Select(e2 => e1 + "=" + e2)).Put();
}
```



Выражения запросов

```
a.SelectMany(e1 => b.Select(e2 => e1 + "=" + e2)).Put();
```

```
var c =
  from e1 in a
  from e2 in b
  select e1 + "=" + e2;
c.Put();
```

```
(from e1 in a
 from e2 in b
 select e1 + "=" + e2).Put();
```

```
public static void Solve()
{
  Task("LinqBegin52");
  var a = GetEnumerableString()
    .Where(e => char.IsDigit(e[e.Length - 1]));
  var b = GetEnumerableString()
    .Where(e => char.IsDigit(e[e.Length - 1]));
  (from e1 in a
   from e2 in b
   orderby e1, e2 descending
   select e1 + "=" + e2).Put();
}
```

```
public static void Solve()
{
  Task("LinqBegin52");
  var a = GetEnumerableString();
  var b = GetEnumerableString();
  (from e1 in a
   where char.IsDigit(e1.Last())
   from e2 in b
   where char.IsDigit(e2.Last())
   orderby e1, e2 descending
   select e1 + "=" + e2).Put();
}
```

Может возникнуть впечатление, что второй вариант решения (использующий выражение запроса) выполняет *больше* действий, чем первый вариант (основанный на вызовах методов LINQ). В самом деле, в первом варианте отбор строк оканчивающихся цифрой, выполняется при построении последовательностей `a` и `b`, т. е., *казалось бы*, один раз для каждой последовательности, а во втором варианте этот отбор выполняется в процессе построения перекрестного объединения, при котором обращение ко второй последовательности производится многократно (а значит, многократно проводится и отбор ее элементов).

На самом деле многократный отбор элементов для последовательности `b` будет проводиться и в первом варианте решения. Это связано с *отложенным* характером выполнения запросов, возвращающих последовательности. При выполнении оператора

```
var b = GetEnumerableString()
  .Where(e => char.IsDigit(e[e.Length - 1]))
  .OrderByDescending(e => e).Show("B");
```

в переменную `b` записывается не результат выполнения методов Where и OrderByDescending, а *сами эти запросы* (в специальном формате). Немедленно выполняются только вспомогательные методы GetEnumerableString и Show, реализованные в задачнике. При последующих обращениях к последовательности `b`, при которых требуется организовать перебор всех ее элементов, методы Where и OrderByDescending каждый раз *выполняются заново*.

```
var b = GetEnumerableString()
  .Where(e => char.IsDigit(e.Show("W").Last()))
  .OrderByDescending(e => e.Show("O"));
```

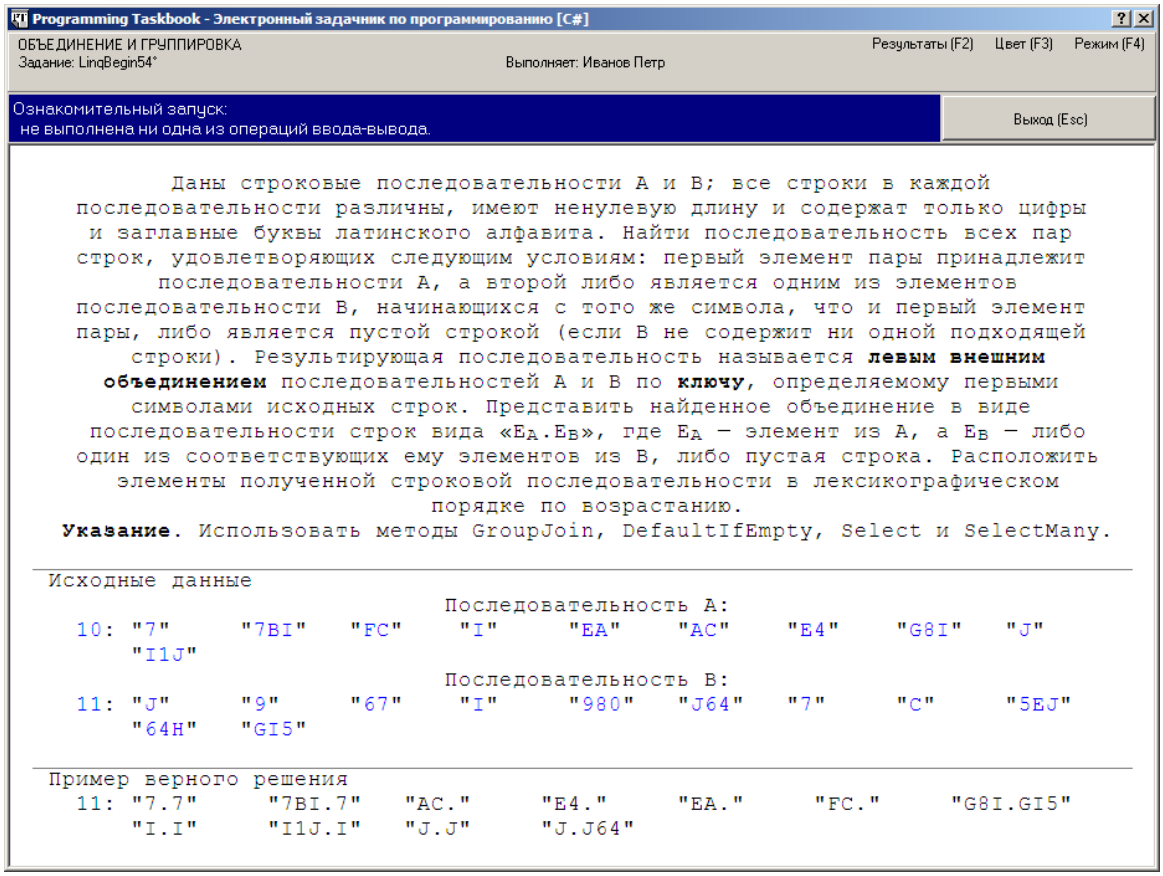
В данной модификации при каждом вызове лямбда-выражения для методов Where и OrderByDescending в раздел отладки записывается аргумент этого лямбда-выражения (в виде последовательности символов), причем аргумент для лямбда-выражения метода Where снабжается комментарием «W», а аргумент для лямбда-выражения метода OrderByDescending — комментарием «O». При запуске программы в разделе отладки будут выведены все элементы исходной последовательности *B*, снабженные комментарием «W», а затем — все оканчивающиеся цифрой элементы этой же последовательно-

сти, снабженные комментарием «O», причем *описанный набор данных будет выведен столько раз, сколько имеется элементов исходной последовательности A, оканчивающихся цифрой*. Это показывает, что отбор элементов последовательности *B* и их сортировка выполняются при обработке *каждого* элемента последовательности *A*, используемого при построении перекрестного объединения.

```
public static void Solve()
{
  Task("LinqBegin52");
  var a = GetEnumerableString()
    .Where(e => char.IsDigit(e[e.Length - 1]))
    .OrderBy(e => e).Show("A").ToArray();
  var b = GetEnumerableString()
    .Where(e => char.IsDigit(e[e.Length - 1]))
    .OrderByDescending(e => e).Show("B").ToArray();
  a.SelectMany(e1 => b.Select(e2 => e1 + "=" + e2)).Put();
}
```

Если модифицировать данную программу тем же способом, которым был модифицирован предыдущий вариант программы (добавив в лямбда-выражения методов Where и OrderByDescending для последовательности b вызовы Show), то элементы последовательности b, обрабатываемые методами Where и OrderByDescending, будут выведены на экран лишь по одному разу.

Построение плоского левого внешнего объединения: LinqBegin54



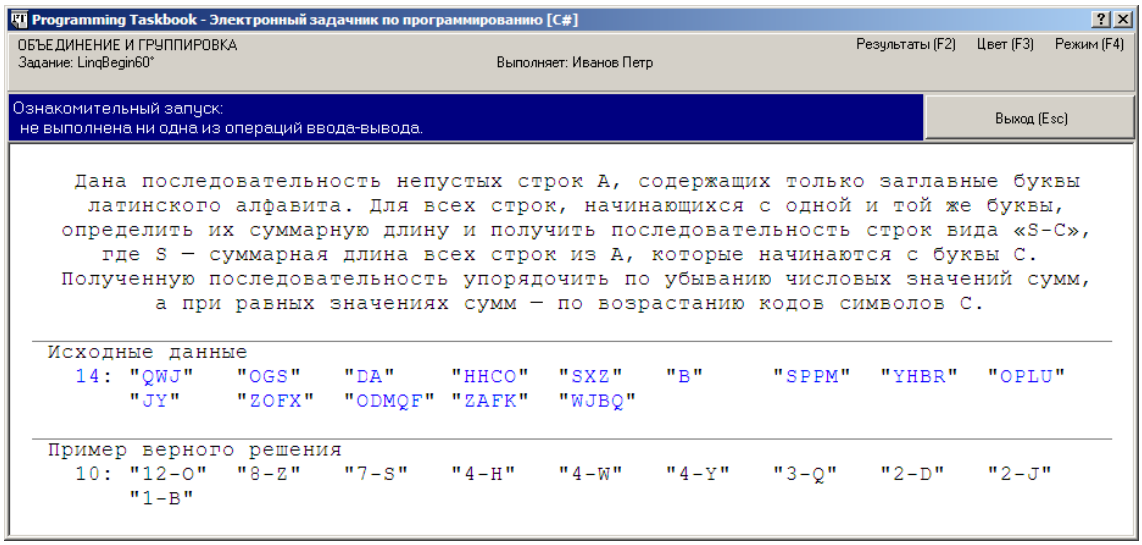
```
public static void Solve()
{
  Task("LinqBegin54");
  GetEnumerableString().GroupJoin(GetEnumerableString(),
      e => e[0], e => e[0],
      (e1, ee2) => ee2.Select(e => e1 + "." + e))
    .SelectMany(e => e).OrderBy(e => e).Show().Put();
}
```


```
public static void Solve()
{
  Task("LinqBegin54");
  GetEnumerableString().GroupJoin(GetEnumerableString(),
      e => e[0], e => e[0],
      (e1, ee2) => ee2.DefaultIfEmpty("")
      .Select(e => e1 + "." + e))
    .SelectMany(e => e).OrderBy(e => e).Show().Put();
}
```

```
public static void Solve()
{
  Task("LinqBegin54");
  (from e1 in GetEnumerableString()
   join e2 in GetEnumerableString()
   on e1[0] equals e2[0]
   into ee2
   from e in ee2.DefaultIfEmpty("")
   orderby e1, e
   select e1 + "." + e).Put();
}
```

```
(from e1 in GetEnumerableString()
 join e2 in GetEnumerableString()
 on e1[0] equals e2[0]
 into ee2
 from e in ee2.DefaultIfEmpty("")
 let r = e1 + "." + e
 orderby r
 select r).Put();
```

Группировка: LinqBegin60




```
GetEnumerableString().GroupBy(e => e[0])
  .OrderByDescending(e => e.Sum(s => s.Length)).ThenBy(e => e.Key)
  .Select(e => e.Sum(s => s.Length) + "-" + e.Key).Put();
```

```
GetEnumerableString().GroupBy(e => e[0], e => e.Length)
  .OrderByDescending(e => e.Sum()).ThenBy(e => e.Key)
  .Select(e => e.Sum() + "-" + e.Key).Put();
```

```
GetEnumerableString()
  .GroupBy(e => e[0], (k, ee) => ee.Sum(s => s.Length) + "-" + k)
  .OrderByDescending(e => int.Parse(e.Split('-')[0]))
  .ThenBy(e => e.Split('-')[1]).Put();
```

```
GetEnumerableString()
  .GroupBy(e => e[0], e => e.Length,
    (k, ee) => ee.Sum() + "-" + k)
  .OrderByDescending(e => int.Parse(e.Split('-')[0]))
  .ThenBy(e => e.Split('-')[1]).Put();
```

```
GetEnumerableString().GroupBy(e => e[0], e => e.Length,
    (k, ee) => new { k, s = ee.Sum() })
  .OrderByDescending(e => e.s).ThenBy(e => e.k)
  .Select(e => e.s + "-" + e.k).Put();
```

```
(from r in
   from e in GetEnumerableString()
   group e.Length by e[0]
 let s = r.Sum()
 let k = r.Key
 orderby s descending, k
 select s + "-" + k).Put();
```

```
(from e in GetEnumerableString()
 group e.Length by e[0]
   into r
   let s = r.Sum()
   let k = r.Key
   orderby s descending, k
   select s + "-" + k).Put();
```