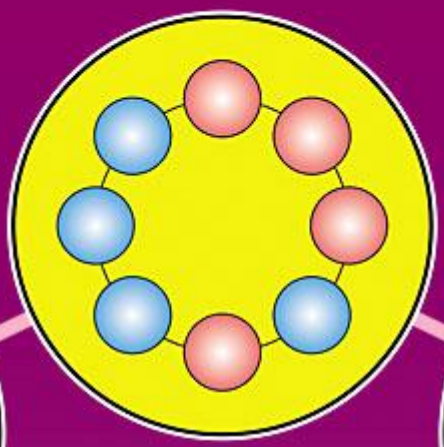
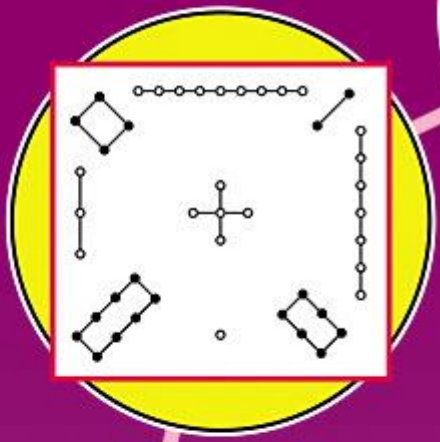
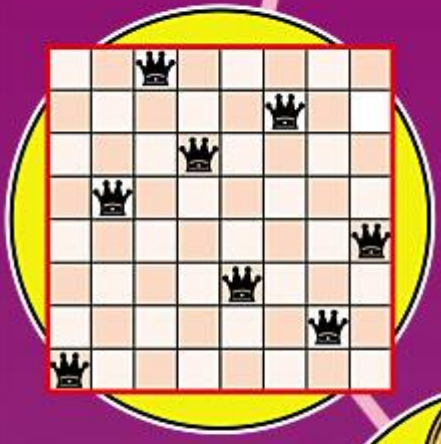
Валерий Рубанцев

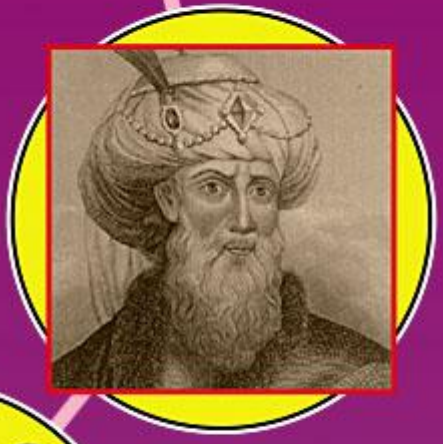




С	О	Р	О	К	А
О	С	Е	Л	О	К
Р	Е	Ф	Е	Р	И
О	Л	Е	Ф	И	Н
К	О	Р	И	Ц	А
А	К	И	Н	А	К

Как решать 100 комбинаторные задачи на компьютере





Валерий Рубанцев

Как решать комбинаторные задачи на компьютере

Комбинаторика для программистов на языке C#

RVGAMES.DE 2013

Это издание представляет собой сокращённый вариант книги **Как решать комбинаторные задачи на компьютере. Комбинаторика для программистов на языке C#. Полная** версия книги включает 10 глав (369 страниц).

Вы можете приобрести её (вместе с исходными кодами всех проектов) на сайте издательства:

RVGames.de/buy2.htm

От автора

Если хочешь быть умным,

поступай в программисты!

Козьма Прутков-программист

Комбинаторика как наука возникла сравнительно недавно. Первая книга *Рассуждения о комбинаторном искусстве* вышла в 1666 году. Написал её известный немецкий математик Готфрид Вильгельм фон Лейбниц, который и придумал название для этого раздела математики.

Однако решать комбинаторные задачи людям приходилось на протяжении всей своей истории. Как в жизни, так и в программировании *комбинаторные задачи* встречаются на каждом шагу. Например, сколько различных слов можно составить из букв русского алфавита? Сколько существует различных комбинаций при игре в кости с двумя или тремя кубиками? Сколько разных нарядов можно составить из трёх юбок и четырёх блузок? Сколько существует разбиений числа на отдельные слагаемые? Сколькими способами можно покрыть прямоугольник домино или тримино?

Многие *детско-спортивные игры* начинаются со считалок, бросания монет или жребия. *Гадания* также основаны на комбинаторике - раскладывании карт, вытягивании спичек, отрывании лепестков у ромашки… Или *азартные игры*! Какое число выпадет на рулетке, какие карты находятся в прикупе, какие числа следует зачеркнуть в карточке лото? Как составить список покупок, расписание соревнований, футбольную команду, фоторобот, пазлы или кубик Рубика, припарковаться, приготовить блюдо, рассадить учеников по партам, расставить книги по полкам, сервировать стол, декорировать комнату, собрать механизм из деталей… И даже творческие порывы не обходятся без комбинаторики! Написание стихов и музыки, графика и живопись – почти комбинаторные процессы. Недаром в этих областях искусства «компьютеры» добились впечатляющих успехов. И наконец, все люди – всего лишь комбинация генов в молекулах ДНК!

Комбинаторика – это настоящий клад для программистов! Здесь можно найти не один десяток великолепных задач, но, как вы знаете, любимым выражением Козьмы Пруткова была фраза *Нельзя объять необъятное*. Поэтому для этой книги я отобрал только **7 жемчужин** из этих комбинаторных сокровищ:

Ожерелья и браслеты

Числовые магические квадраты

Словесные магические квадраты

Задача Иосифа Флавия

Расстановка ферзей на шахматной доске

Размен денег

Расстановка знаков между числами, чтобы получилась сотня

По крайней мере, ещё столько же замечательных комбинаторных проблем осталось за бортом, но - *нельзя объять необъятное!*

Почти все задачи увлекательны, поэтому их интересно решать. Однако ими занимались и такие учёные, как Леонард Эйлер, Карл-Фридрих Гаусс, Эжен Шарль Каталан, Дональд Кнут и многие другие современные учёные, что должно убедить вас в серьёзности и глубине этих задач, многие из которых непосредственно связаны и с практическими комбинаторными проблемами, о чём мы не раз будем говорить на страницах этой книги. А **цель** её в том и состоит, чтобы показать на этих исторических примерах, *как* можно решить эти задачи по-современному, с помощью компьютера.

Впрочем, этими «большими» задачами не ограничивается список тем книги. Мы решим и «сопутствующие» комбинаторные проблемы:

Раскраска вершин правильного многоугольника

Слова Линдона и де Бройна

Кружки с цифрами

Максимальная числовая подпоследовательность

Задача Макмагона о квадратном домино

Расстановка ладей и задача о назначениях

Магическая таблица

Задачи Дональда Кнута

В конце каждой главы вы найдёте «домашние» задания для самостоятельного решения, которые помогут вам закрепить знания по комбинаторике и программированию.

Для решения задач мы будем использовать как общие *методы* –

полный перебор,
перебор с возвратами,
жадные алгоритмы,
динамическое программирование,
имитация отжига, -

так и чисто *комбинаторные* -

генерирование перестановок,
генерирование сочетаний,
генерирование разбиений,
генерирование композиций,
генерирование браслетов,
генерирование ожерелий,
генерирование слов Линдона,
генерирование слов де Бройна,
генерирование чисел Каталана,
генерирование расстановок скобок.

Все приложения написаны на языке *Си-шарп*, который идеально подходит для решения комбинаторных задач, но исходный код без труда может быть переведён на любой другой современный язык программирования, поддерживающий платформу *.NET*.

Я надеюсь, что книга будет полезна всем программистам, интересующимся комбинаторикой вообще и комбинаторными алгоритмами в частности, а также школьникам старших классов и студентам.

Валерий Рубанцев

Условные обозначения, принятые в книге:

Дополнение, примечание

Предложение, ненавязчивое требование

Задания для *самостоятельного* решения

```
Исходный код
```

Папка с **исходным кодом** приложения

Все исходные коды находятся в папке **_Projects**.

Оглавление

Полное оглавление

Как решать комбинаторные задачи на компьютере

Глава 5. Магические квадраты

Давным-давно это было… Китайская легенда гласит, что в 23 веке до нашей эры (впрочем, дотошные историки утверждают, что это событие произошло не раньше четвёртого века до нашей эры) из реки Ло выползла черепаха, на панцире которой люди обнаружили странные знаки (Рис. 5.1). Они пересчитали точки, и оказалось, что их число равно *пятнадцати* по всем горизонтальным и вертикальным рядам, а также по двум главным диагоналям «квадрата». Древние китайцы были так поражены этими необыкновенными свойствами «черепаховой» фигуры, что назвали её магическим квадратом *Ло-шу*.

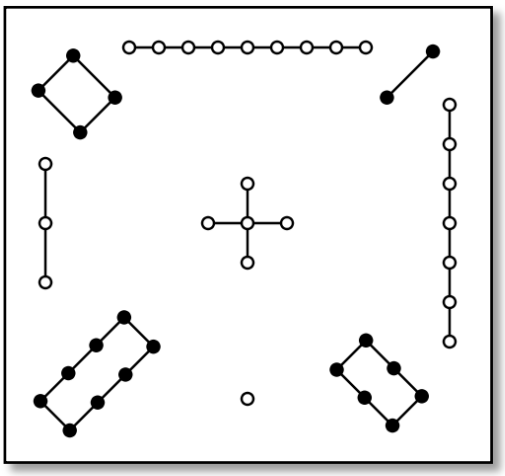


Рис. 5.1. Магический квадрат Ло-шу на панцире черепахи

Китайцы придавали квадрату Ло-шу мистические свойства и считали его символом, объединяющим предметы, людей и вселенную. *Чётные* числа в нём представляют женскую сущность *Инь*, а *нечётные* – мужскую – *Ян* (Рис. 5.2).

Рис. 5.2. Знаменитый символ Инь-Ян

В центре Ло-шу находится *пятёрка*, которая символизирует *Землю* (Рис. 5.3). Вокруг неё располагаются элементы (стихии). *Металл*, представленный числами 4 и 9, *вода* – 1 и 6, *огонь* – 2 и 7, *дерево* 3 и 8. Легко заметить, что каждый элемент содержит мужское и женское начала Инь и Ян.

Рис. 5.3. Элементы природы в квадрате Ло-шу

Из Китая магические квадраты попали в Индию, а затем - через арабские страны – в Европу. В эпоху Ренессанса немецкий математик *Генрих Корнелиус Агриппа* (1486-1536) построил магические квадраты от третьего до девятого порядка и дал им астрономическое толкование.

Эти квадраты представляли собой семь известных тогдашней науке «планет»: Сатурн, Юпитер, Марс, Солнце, Венеру, Меркурий и Луну.

Но создание первого «европейского» магического квадрата приписывают немецкому художнику, уроженцу города Нюрнберга, современнику Леонардо да Винчи Альбрехту Дюреру. На знаменитой гравюре *Меланхолия* (Рис. 5.4) в правом верхнем углу он поместил магический квадрат четвёртого порядка. Причём Дюрер составил его так ловко, что два средних числа в нижней строке образовали год создания произведения - 1514 (Рис. 5.5).

После Ло-шу это самый известный магический квадрат в мире!

Конечно, Дюрер украсил свою гравюру магическим квадратом не только для того, чтобы указать год. Дело в том, что в то время квадраты четвёртого порядка считались хорошим терапевтическим средством, и астрологи «прописывали» их для амулетов против меланхолии.

Необычным свойствам магических квадратов люди с давних времен находили применение в мистике и религии. Вырезанные на дереве, камне или металле магические квадраты служили амулетами (в этом качестве они до сих пор используются в восточных странах). Ещё в 16-17 веках люди вери-

ли, что выгравированный на серебряной пластине магический квадрат может защитить их от чумы.

Арабские астрологи более девяти веков назад употребляли магические квадраты при составлении гороскопов.

Рис. 5.4. Дюрер. «Меланхолия»

Рис. 5.5. Магический квадрат крупным планом

Математические подробности

Магический квадрат – это таблица, которая состоит из равного числа строк и столбцов. Во всех её клетках записано по одному числу. Если обозначить буквой N число строк (или столбцов) в таблице - это число называется *порядком* магического квадрата, - то в обычном магическом квадрате записаны последовательные числа от 1 до N^2. Например, в самом простом магическом квадрате порядка 3 (то есть размером 3 x 3 клетки) мы отыщем все числа от 1 до $3^2 = 9$ (Рис. 5.6). Если вы не забыли, это и есть знаменитый квадрат Ло-шу.

Магические квадраты иначе называют *волшебными*.

Таким образом, мы всегда знаем, какие числа следует записать в таблицу. Сделать это в произвольном порядке, конечно нетрудно, но ведь квадрат не зря называется *магическим*! В нём сумма чисел в каждой строке, столбце и в каждой из двух главных диагоналей должна быть *одинаковой*. Эта *сумма* называется *магической* и обозначается буквой *S*.

Магическая сумма называется также *константой* магического квадрата.

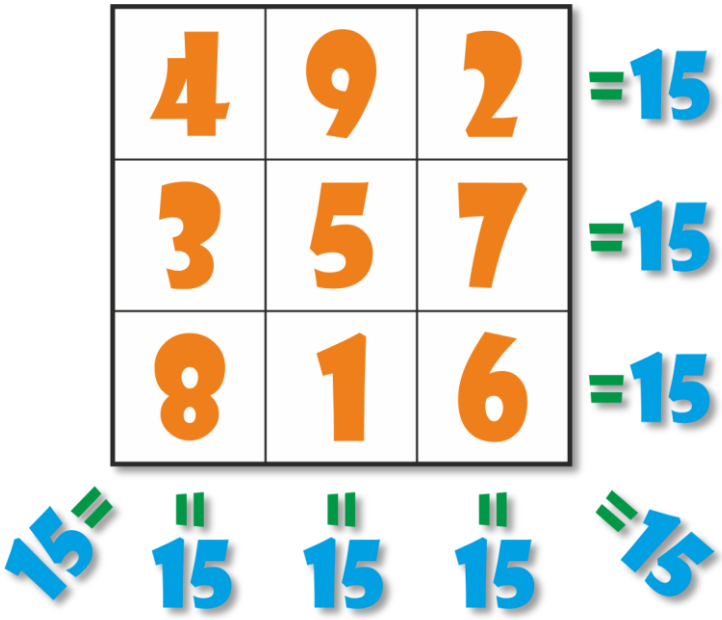


Рис. 5.6. Магический квадрат *третьего* порядка

Мы легко найдём магическую сумму любого квадрата, если сложим все числа в квадрате и разделим сумму на порядок квадрата. Последовательные натуральные числа, которые находятся в магическом квадрате, образуют *арифметическую прогрессию*, сумму членов которой мы умеем находить:

Первый член прогрессии равен 1.
Последний член прогрессии равен N^2.
Число членов прогрессии равно N^2.

Сумму всех членов прогрессии можно вычислить по формуле:

$$sum = \frac{1 + n^2}{2} \cdot n^2 = \frac{n^2(1 + n^2)}{2}$$

А *магическую сумму* – по формуле:

$$S = \frac{n(1 + n^2)}{2}$$

А сколько всего магических квадратов?

Мало ли в Бразилии Педров? -
И не сосчитаешь!

Фраза тётушки Чарли
из комедии *Здравствуйте, я ваша тётя*

Магические квадраты *первого* и *третьего* порядков существуют в единственном числе, а квадратов *второго* порядка вообще нет.

> Мы считаем только *уникальные* квадраты, которые нельзя получить из других с помощью поворотов и отражений.
>
> Имеется 8 способов размещения чисел 1..9 в квадрате 3 x 3 клетки. Но если не считать различными квадраты, совпадающие при поворотах и отражениях, то останется *единственный* квадрат Ло-шу.
>
> Подробное доказательство этих утверждений вы найдёте, например, в книге Мартина Гарднера **[ГМ90],** в главе 17 *Магические квадраты и кубы*.

Французский математик *Бернар де Бесси* (*Bernard Frénicle de Bessy*, 1605 – 1675) насчитал 880 магических квадратов *четвёртого* порядка.

Долгое время ученые оценивали число магических квадратов *пятого* порядка в 13 миллионов, пока в 1973 году американский программист Ричард Шрёппель (Richard Schroeppel) с помощью компьютера не нашёл их точное число - 275 305 224.

Существует несколько различных *классификаций* магических квадратов пятого порядка, призванных хоть как-то их систематизировать. В книге Мартина Гарднера **[ГМ90,** сс. 244-345**]** описан один из таких способов – по числу в центральном квадрате. Способ любопытный, но не более того.

Сколько существует квадратов *шестого* порядка, до сих пор неизвестно, но их примерно 1.77×10^{19}. Число огромное, поэтому нет никаких надежд пересчитать их с помощью полного перебора, а вот *формулы* для подсчёта магических квадратов никто придумать не смог.

Как составить магический квадрат?

Придумано очень много способов построения магических квадратов. Проще всего составлять магические квадраты *нечётного порядка*. Мы воспользуемся методом, который предложил французский учёный XVII века *А. де ла Лубер (De La Loubère).* Он основан на *пяти* правилах, действие которых мы рассмотрим на самом простом магическом квадрате 3 x 3 клетки.

Правило 1. Поставьте 1 в среднюю колонку первой строки (Рис. 5.7).

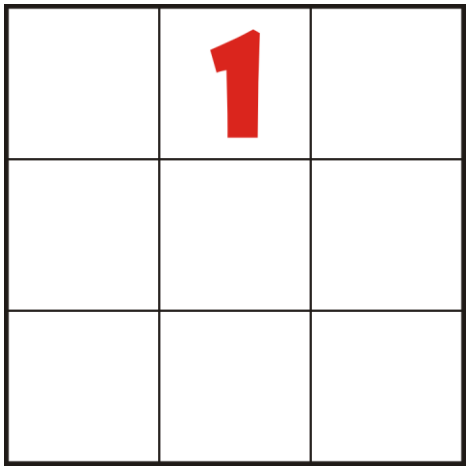


Рис. 5.7. Первое число

Правило 2. Следующее число поставьте, если возможно в клетку, соседнюю с текущей по диагонали правее и выше (Рис. 5.8).

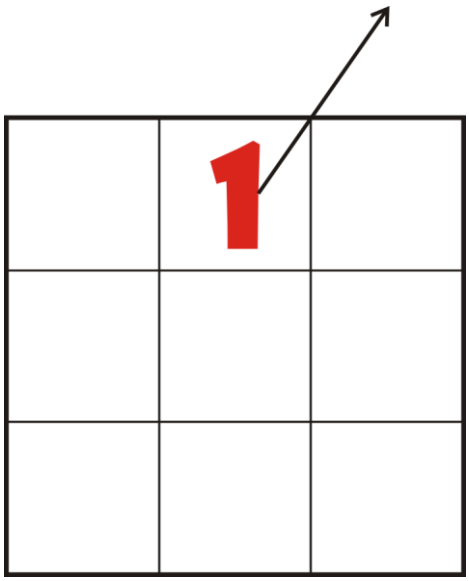


Рис. 5.8. Пытаемся поставить второе число

Правило 3. Если новая клетка выходит за пределы квадрата *сверху*, то запишите число в самую нижнюю строку и в следующую колонку (Рис. 5.9).

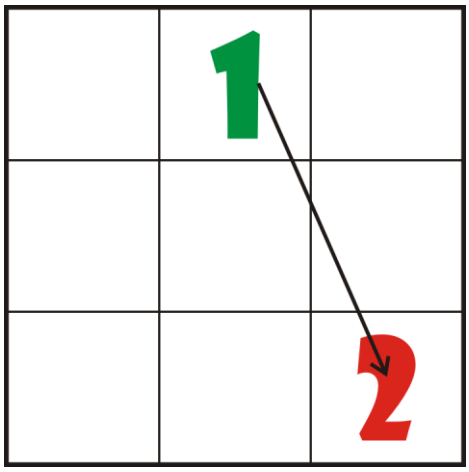


Рис. 5.9. Ставим второе число

Правило 4. Если клетка выходит за пределы квадрата *справа*, то запишите число в самую первую колонку и в предыдущую строку (Рис. 5.10).

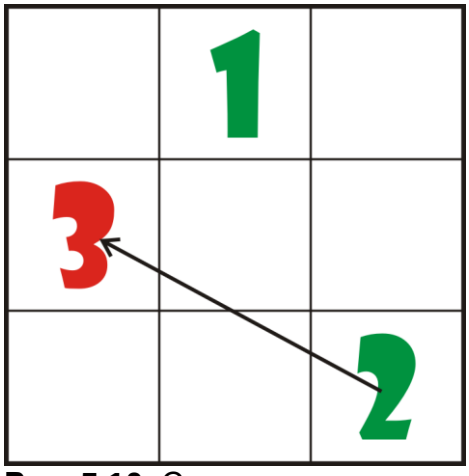


Рис. 5.10. Ставим третье число

Правило 5. Если в клетке уже *занята*, то очередное число запишите под текущей клеткой (Рис. 5.11).

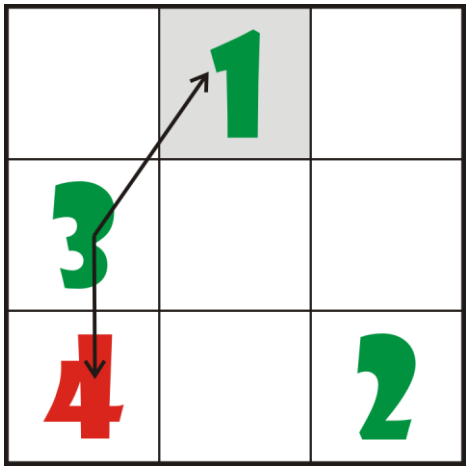


Рис. 5.11. Ставим четвёртое число

Далее переходите к *Правилу 2* (Рис. 5.12).

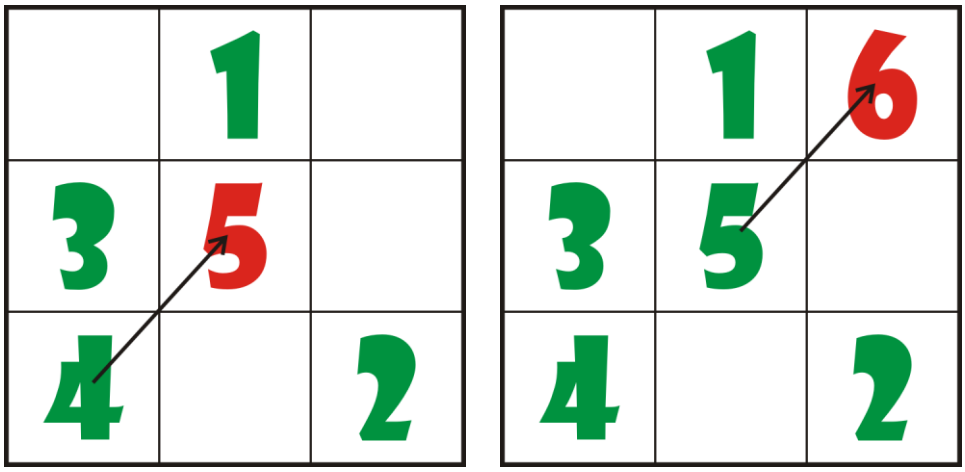


Рис. 5.12. Ставим пятое и шестое число

Снова выполняйте Правила 3, 4, 5, пока не составите весь квадрат (Рис. 5.13).

Не правда ли, правила очень простые и понятные, но всё равно довольно утомительно расставлять даже 9 чисел. Однако, зная алгоритм построения магических квадратов, мы сможем легко перепоручить компьютеру всю рутинную работу, оставив себе только творческую, то есть написание программы.

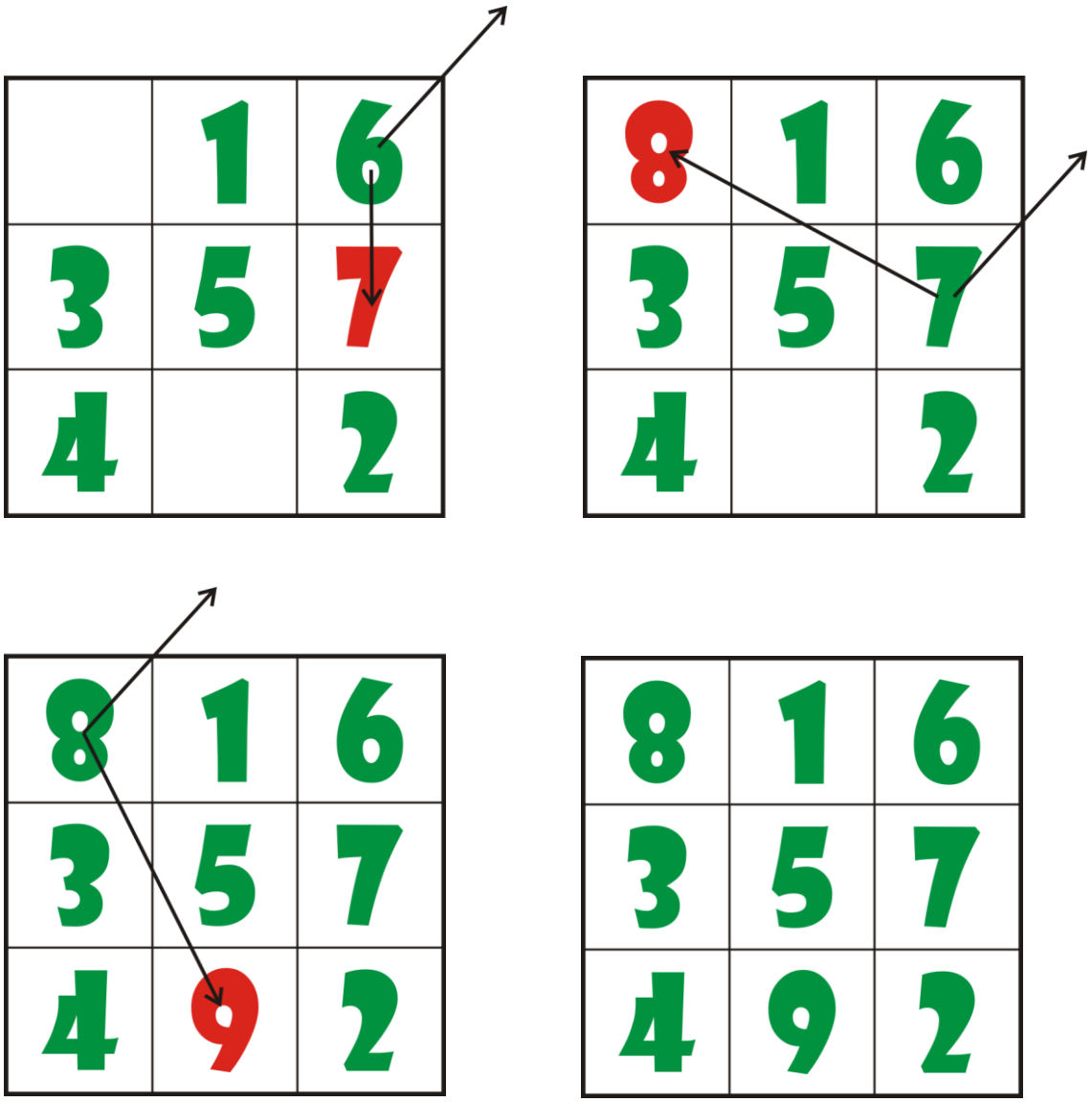


Рис. 5.13. Заполняем квадрат следующими числами

Проект *Магические квадраты* (Magic)

Набор *полей* для программы **Магические квадраты** совершенно очевиден:

```
// ПРОГРАММА ДЛЯ ГЕНЕРИРОВАНИЯ
// НЕЧЕТНЫХ МАГИЧЕСКИХ КВАДРАТОВ
// ПО МЕТОДУ ДЕ ЛА ЛУБЕРА
public partial class Form1 : Form
{
  //макс. размеры квадрата:
  const int MAX_SIZE = 27;
  //var
  int n=0;      // порядок квадрата
  int [,] mq;   // магический квадрат
  int number=0;// текущее число для записи в квадрат
```

```
  int col=0;   // текущая колонка
  int row=0;   // текущая строка
```

Метод де ла Лубера годится для составления нечётных квадратов *любого* размера, поэтому мы можем предоставить пользователю возможность самостоятельно выбирать порядок квадрата, разумно ограничив при этом свободу выбора 27-ью клетками.

После того как пользователь нажмёт заветную кнопку **btnGen** *Генерировать!*, метод **btnGen_Click** создаёт массив для хранения чисел и переходит в метод *generate*:

```
//НАЖИМАЕМ КНОПКУ "ГЕНЕРИРОВАТЬ"
private void btnGen_Click(object sender, EventArgs e)
{
  //порядок квадрата:
  n = (int)udNum.Value;

  //создаем массив:
  mq = new int[n+1, n+1];

  //генерируем магический квадрат:
  generate();

  lstRes.TopIndex = lstRes.Items.Count-27;
}
```

Здесь мы начинаем действовать по правилам де ла Лубера и записываем первое число – единицу – в среднюю клетку первой строки квадрата (или массива, если угодно):

```
  //Генерируем магический квадрат
  void generate(){
    //первое число:
    number=1;
rule1:
    //колонка для первого числа - средняя:
    col = n / 2 + 1;
    //строка для первого числа - первая:
    row=1;
    //заносим его в квадрат:
    mq[row,col]= number;
```

Теперь мы последовательно пристраиваем по клеткам остальные числа – от двойки до n * n:

```
    //переходим к следующему числу:
```

```
nextNumber:
    number++;
```

Запоминаем на всякий случай координаты актуальной клетки

```
    int tc=col;
    int tr = row;
```

и переходим в следующую клетку по диагонали:

```
    col++;
    row--;
```

Проверяем выполнение *третьего* правила:

```
rule3:
   if (row < 1) row= n;
```

А затем *четвёртого*:

```
rule4:
   if (col > n) {
       col=1;
       goto rule3;
   }
```

И *пятого*:

```
rule5:
   if (mq[row,col] != 0) {
       col=tc;
       row=tr+1;
       goto rule3;
   }
```

Как мы узнаем, что в клетке квадрата уже находится число? – Очень просто: мы предусмотрительно записали во все клетки *нули*, а числа в готовом квадрате *больше нуля*. Значит, по значению элемента массива мы сразу же определим, пустая клетка или уже с числом! Обратите внимание, что здесь нам понадобятся те координаты клетки, которые мы запомнили *перед* поиском клетки для следующего числа.

Рано или поздно мы найдём подходящую клетку для числа и запишем его в соответствующую ячейку массива:

```
    //заносим его в квадрат:
    mq[row, col] = number;
```

> Попробуйте иначе организовать проверку допустимости перехода в новую клетку!

Если это число было *последним*, то программа свои обязанности выполнила, иначе она добровольно переходит к обеспечению клеткой следующего числа:

```
    //если выставлены не все числа, то
    if (number < n*n)
        //переходим к следующему числу:
        goto nextNumber;
```

И вот квадрат готов! Вычисляем его магическую сумму и распечатываем на экране:

```
     //построение квадрата закончено:
     writeMQ();
  } //generate()
```

Напечатать элементы массива очень просто, но важно учесть *выравнивание* чисел разной «длины», ведь в квадрате могут быть одно-, дву- и трёхзначные числа:

```
  //Печатаем магический квадрат
  void writeMQ()
  {
    lstRes.ForeColor = Color.Black;
    string s = "Магическая сумма = " + (n*n*n +n)/2;
    lstRes.Items.Add(s);
    lstRes.Items.Add("");
    // печатаем магический квадрат:
    for (int i= 1; i<= n; ++i){
       s="";
       for (int j= 1; j <= n; ++j){
          if (n*n > 10  && mq[i,j] < 10) s += " ";
          if (n*n > 100 && mq[i,j] < 100) s += " ";
          s= s + mq[i,j] + " ";
       }
       lstRes.Items.Add(s);
    }
    lstRes.Items.Add("");
  }//writeMQ()
```

Запускаем программу – квадраты получаются быстро и на загляденье (Рис. 5.14).

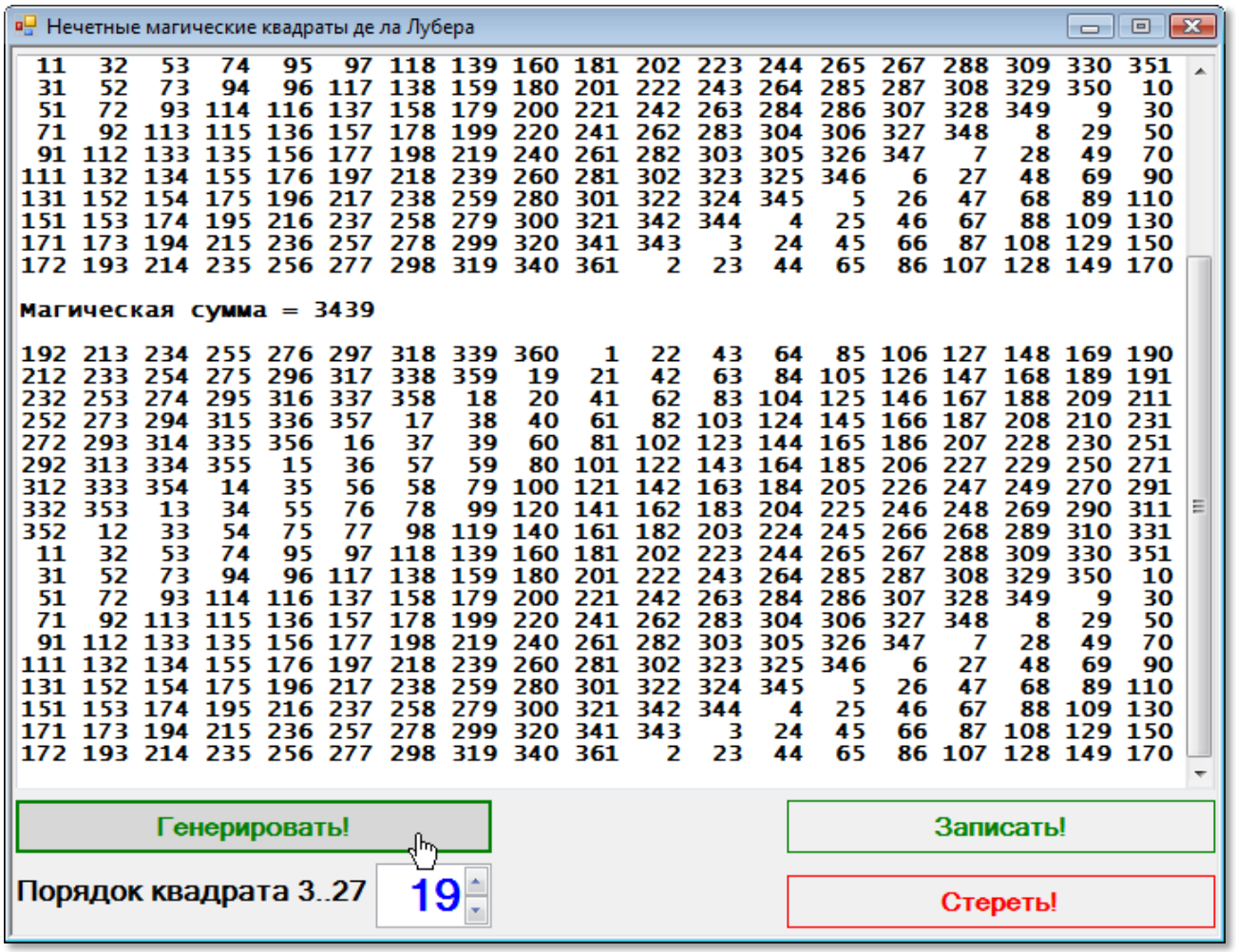


Рис. 5.14. Изрядный квадратище!

В книге С.Гудман, С.Хидетниеми *Введение в разработку и анализ алгоритмов*, на страницах 297-299 мы отыщем тот же самый алгоритм, но в «сокращённом» изложении. Он не столь «прозрачен», как наша версия, но работает верно.

Добавим кнопку **btnGen2** *Генерировать 2!* и запишем алгоритм на языке *Си-шарп* в *метод* **btnGen2_Click**:

```
//Algorithm ODDMS
private void btnGen2_Click(object sender, EventArgs e)
{
    //порядок квадрата:
    n = (int)udNum.Value;
    //создаем массив:
    mq = new int[n + 1, n + 1];
    //генерируем магический квадрат:
    int row = 1;
```

```
        int col = (n+1)/2;
        for (int i = 1; i <= n * n; ++i)
        {
            mq[row, col] = i;
            if (i % n == 0)
            {
                ++row;
            }
            else
            {
                if (row == 1)
                    row = n;
                else
                    --row;
                if (col == n)
                    col = 1;
                else
                    ++col;
            }
        }
        //построение квадрата закончено:
        writeMQ();
        lstRes.TopIndex = lstRes.Items.Count - 27;
    }
```

Кликаем кнопку и убеждаемся, что генерируются «наши» квадраты (Рис. 5.15).

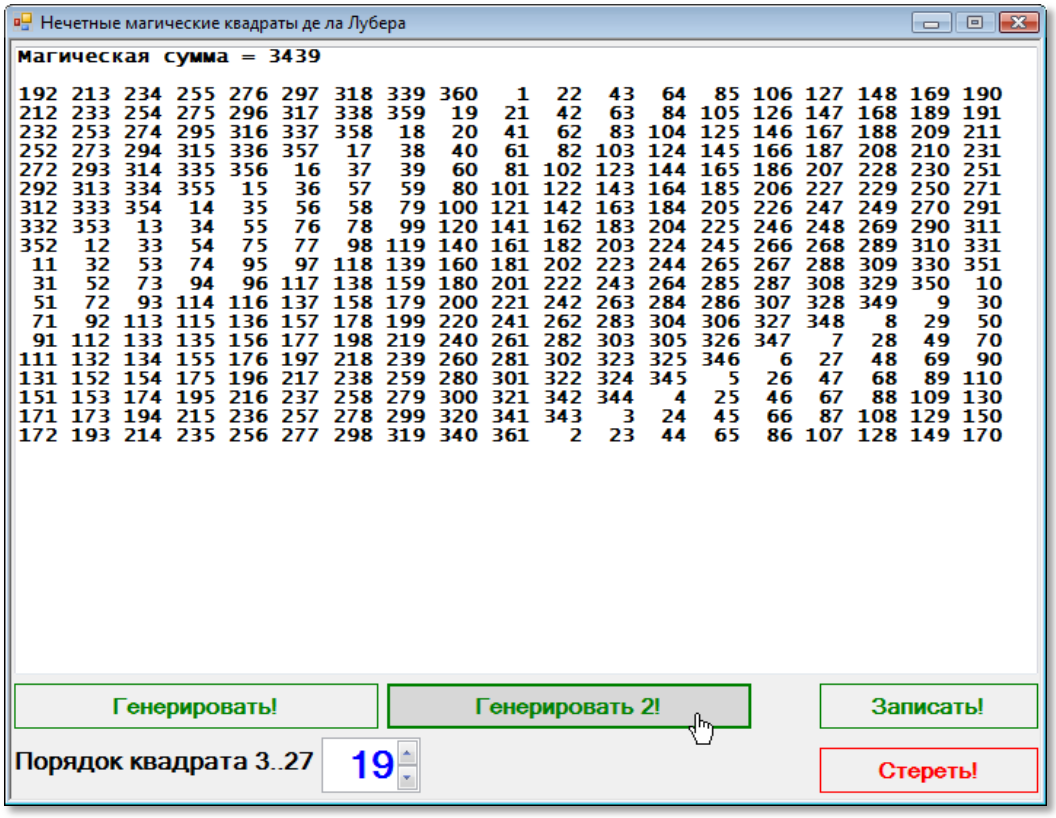
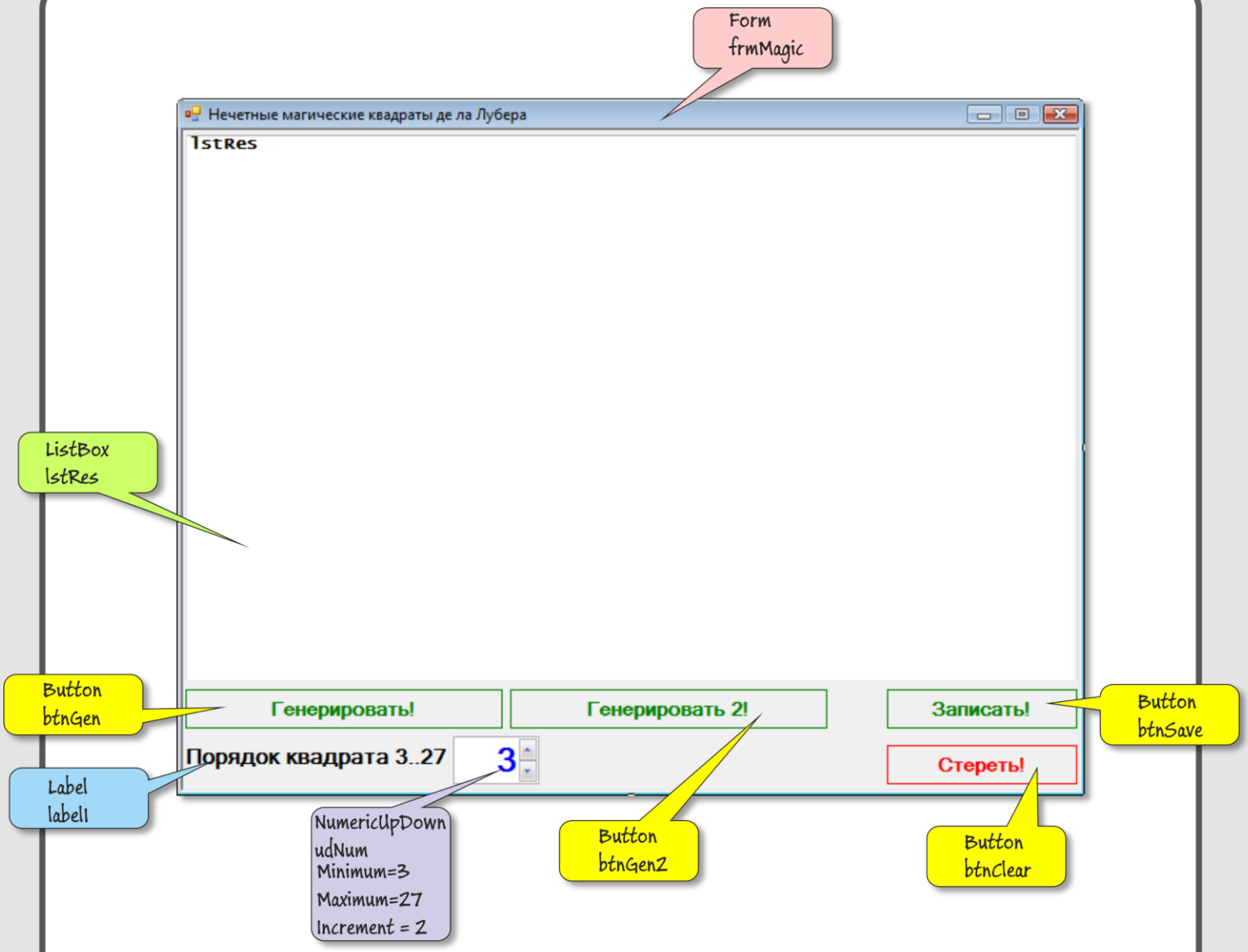


Рис. 5.15. Старый алгоритм в новом обличии



Примерный вид интерфейса приложения

Исходный код программы находится в папке **Magic**.

Магические ферзи

Магический квадрат *седьмого* порядка, составленный по методу де ла Лубера (Рис. 5.16), как показал американец *Клиффорд Пиковер*, непостижимым образом связан с другой знаменитой задачей – расстановкой ферзей на шахматном поле.

Поскольку метод де ла Лубера позволяет строить только нечётные квадраты, а шахматная доска имеет чётный порядок, то придётся взять квад-

рат, наиболее близкий к шахматной доске, то есть 7 x 7 клеток. Мы его легко построим с помощью нашей программы.

```
Магическая сумма = 175

30 39 48  1 10 19 28
38 47  7  9 18 27 29
46  6  8 17 26 35 37
 5 14 16 25 34 36 45
13 15 24 33 42 44  4
21 23 32 41 43  3 12
22 31 40 49  2 11 20
```

Рис. 5.16. Квадрат де ла Лубера

Следующий шаг: заменяем все числа в магическом квадрате остатками от их деления на семь. При этом нулевой остаток будем считать семёркой (Рис. 5.17).

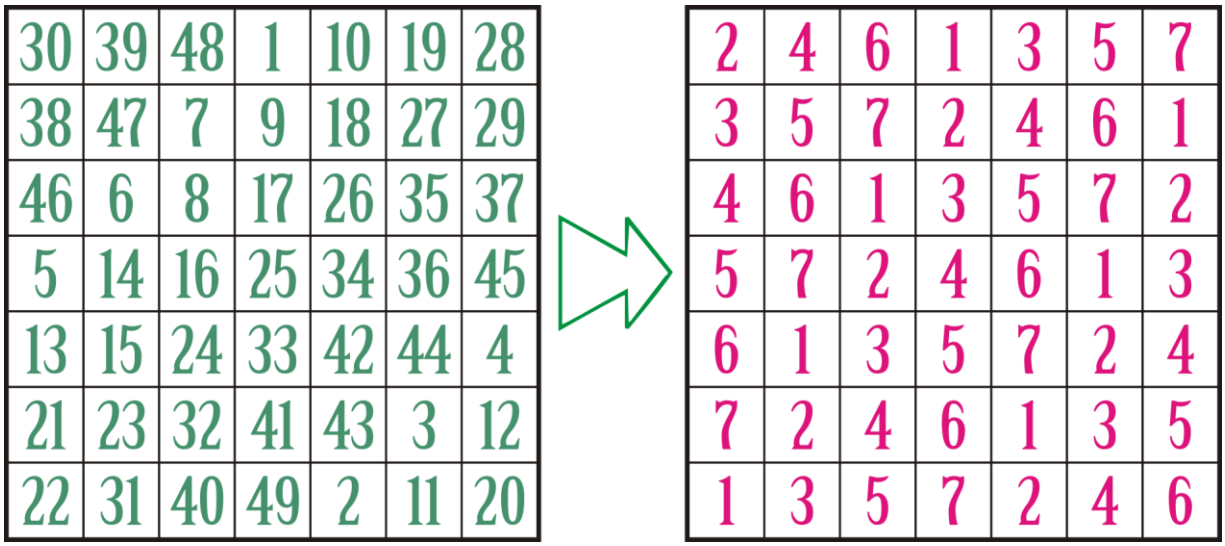


Рис. 5.17. Преобразование квадрата де ла Лубера

Новый квадрат в каждой строке и паре нисходящих диагоналей содержит решение для задачи с ферзями. Например, из первой строки вытекает такое решение (Рис. 5.18).

Одна из главных диагоналей квадрата нисходящая. Её можно использовать для решения задачи о ферзях, но мы рассмотрим другой пример - когда нисходящая диагональ является *побочной (ломаной)*, поэтому должна быть *продолжена* так, чтобы в ней оказалось ровно семь клеток (Рис. 5.19).

2	4	6	1	3	5	7
3	5	7	2	4	6	1
4	6	1	3	5	7	2
5	7	2	4	6	1	3
6	1	3	5	7	2	4
7	2	4	6	1	3	5
1	3	5	7	2	4	6

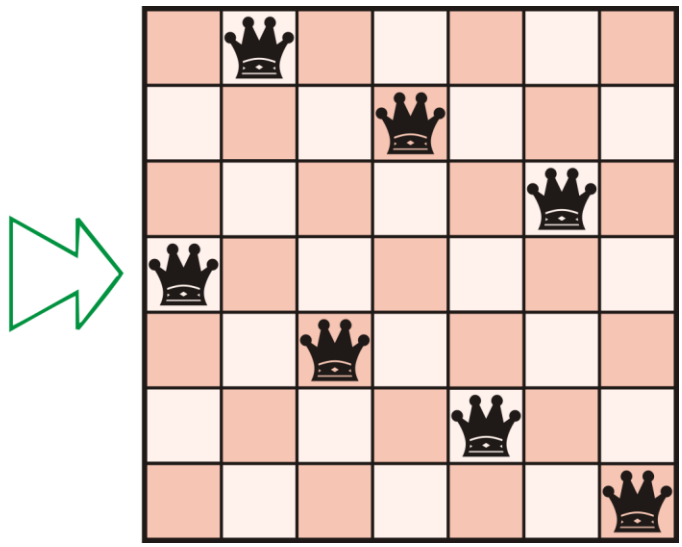


Рис. 5.18. Строки дают решение задачи с ферзями

2	4	6	1	3	5	7
3	5	7	2	4	6	1
4	6	1	3	5	7	2
5	7	2	4	6	1	3
6	1	3	5	7	2	4
7	2	4	6	1	3	5
1	3	5	7	2	4	6

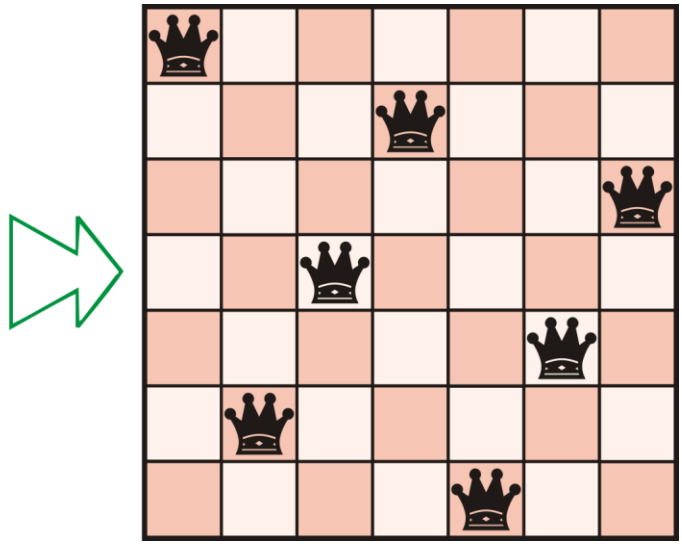


Рис. 5.19. Нисходящие диагонали дают решение задачи с ферзями

Магические квадраты четвёртого порядка

Теперь давайте составим магический квадрат *четвёртого* порядка. Один из таких квадратов изображён на гравюре Дюрера.

Для удобства нарежьте из бумаги 16 квадратиков и напишите на них числа от 1 до 16. Это самая трудная и ответственная часть задания! Дальше будет легче, особенно если вам приходилось складывать картинки из пазлов. Разложите бумажки в каре так, чтобы числа следовали друг за другом по ранжиру, то есть соблюдая субординацию (Рис. 5.20).

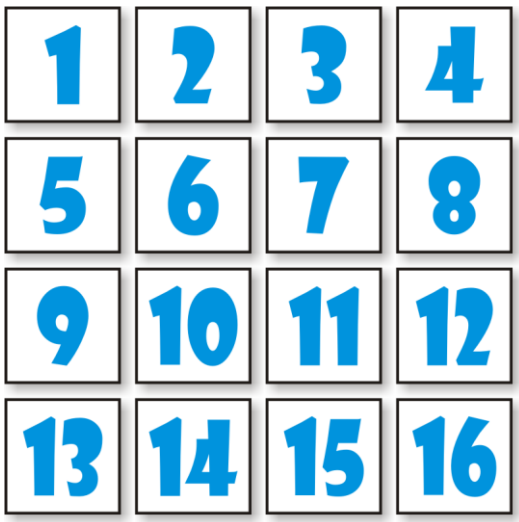


Рис. 5.20. Приняли исходное положение

Шаг 1. Поменяйте местами *вторую* и *третью* строки (Рис. 5.21).

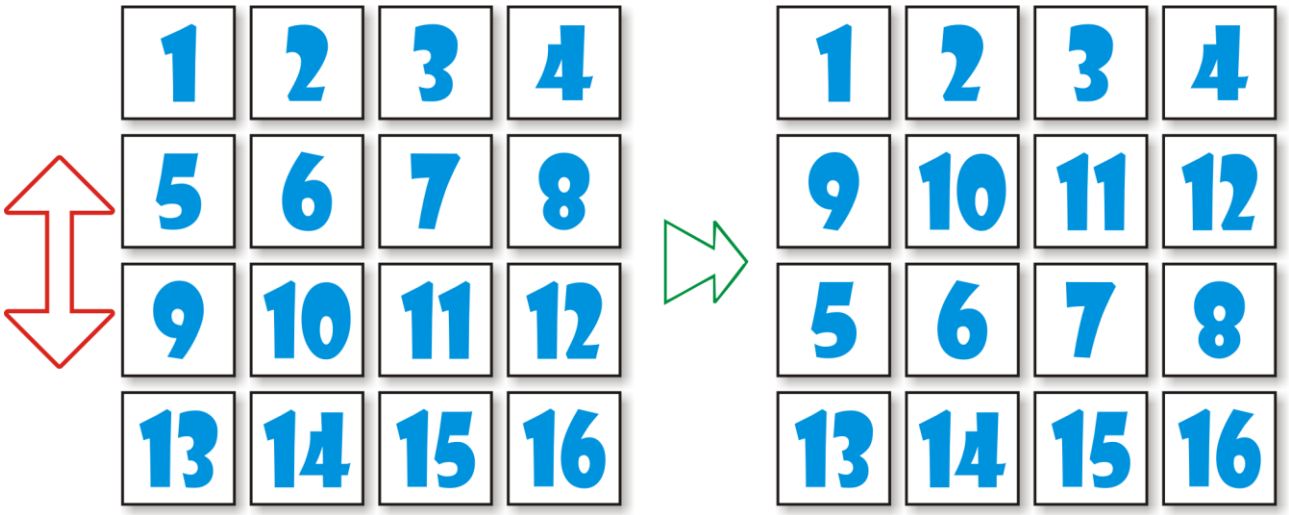


Рис. 5.21. Первый шаг

Шаг 2. Переложите числа во *второй* и *четвёртой* строках в обратном порядке (Рис. 5.22).

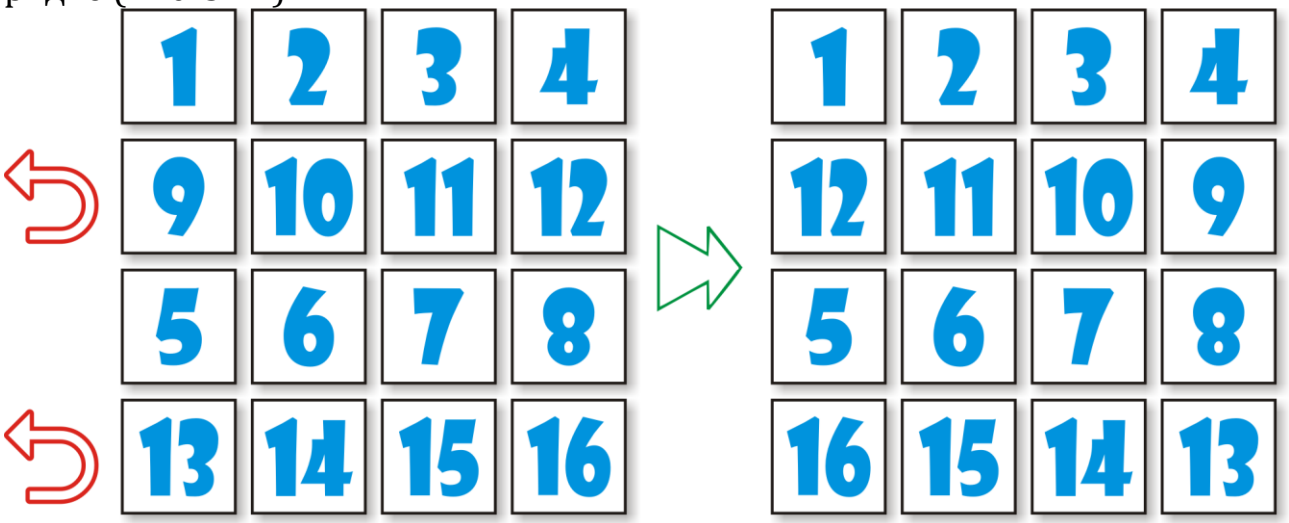


Рис. 5.22. Второй шаг

Шаг 3. Переложите числа во *втором* и *третьем* столбцах в обратном порядке (Рис. 5.23).

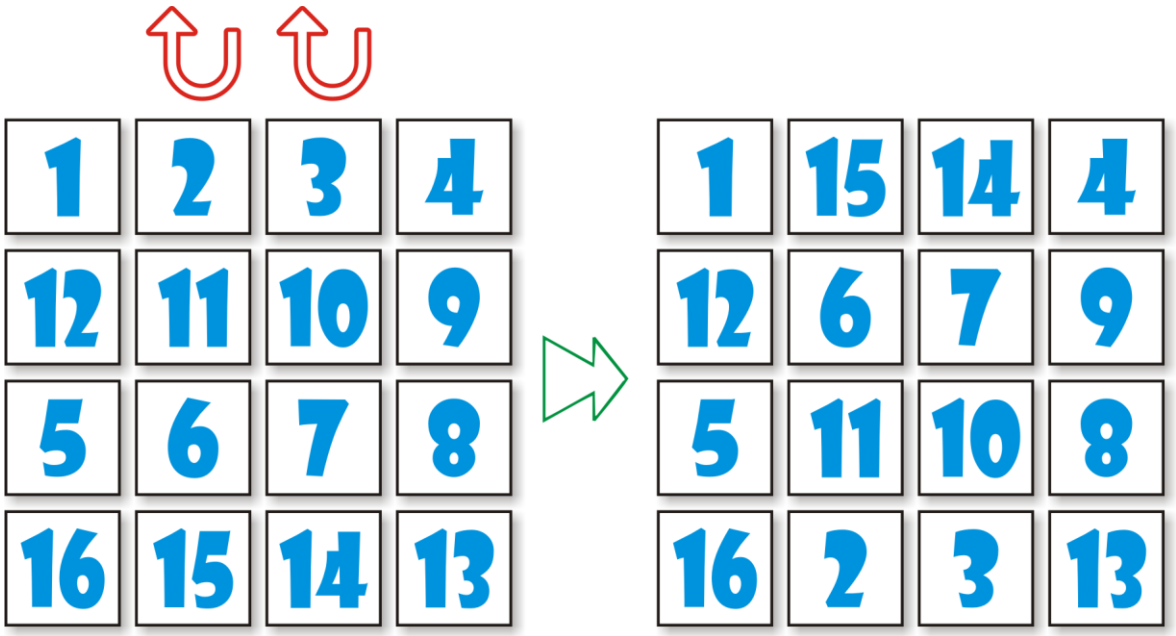


Рис. 5.23. Третий шаг

Шаг 4. Переложите числа в *третьей* и *четвёртой* строках в обратном порядке (Рис. 5.24).

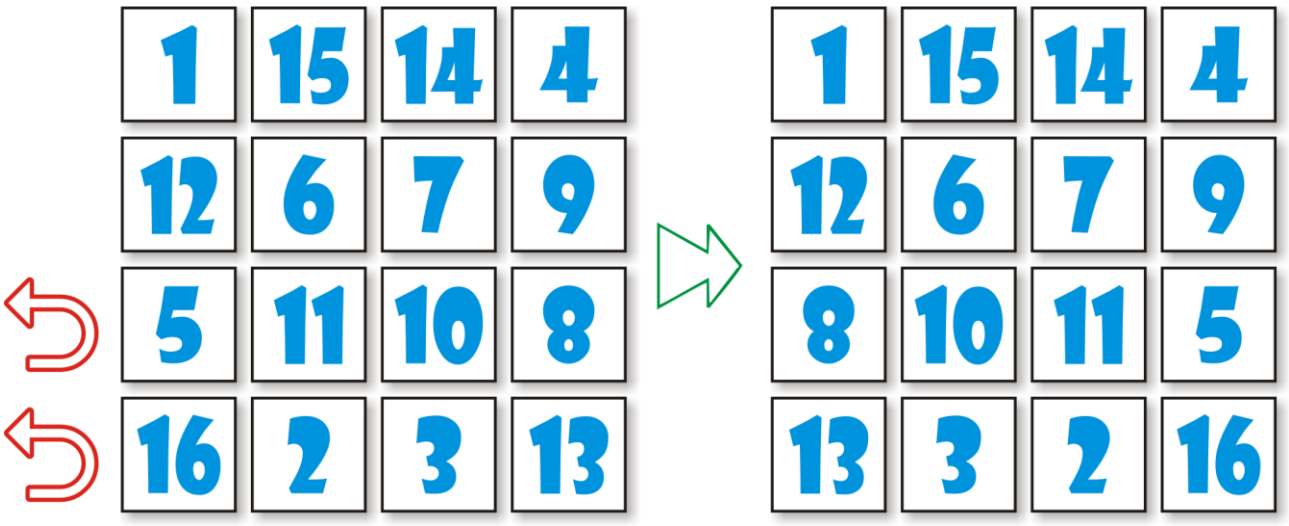


Рис. 5.24. Четвёртый шаг

Магический квадрат готов! И почти как у Дюрера, - только год стоит не в последней, а в первой строке. Впрочем, вы можете перевернуть квадрат вверх ногами, чтобы дата оказалась внизу. Осталось переставить *первый* и *четвёртый* столбец – и наш квадрат превращается в «дюреровский» (Рис. 5.25).

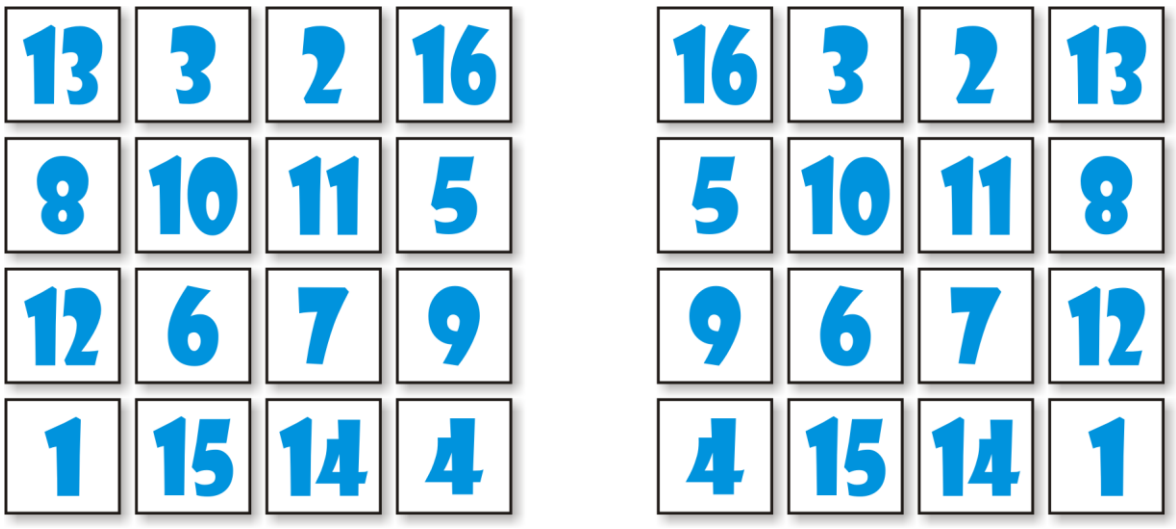


Рис. 5.25. Смастерили Дюреровский квадрат!

Мы не знаем, как именно построил Дюрер свой магический квадрат, но, возможно, и «нашим» способом.

Кстати говоря, в магическом квадрате, который мы построили, *больше* магических сумм, чем «требуется». Магическая сумма, которая в квадратах четвертого порядка равна 34, повторяется не только в четырёх строках, четырёх столбцах и двух диагоналях, но и

- в пяти квадратиках 2 x 2 клетки (Рис. 5.26).

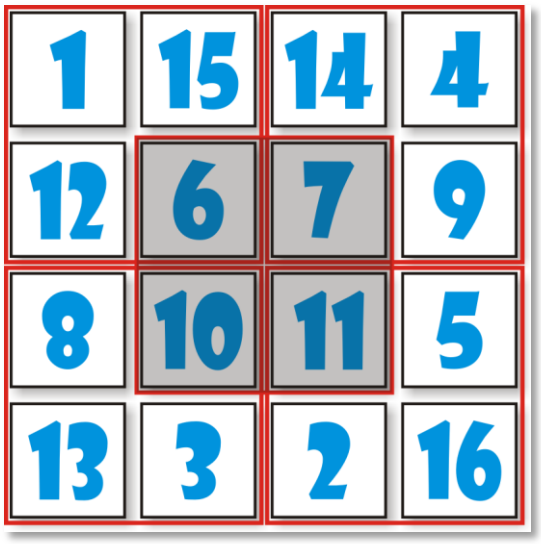


Рис. 5.26. Дополнительные магические квадратики 2 x 2

- в углах четырёх квадратов 3 x 3 клетки (Рис. 5.27).

Рис. 5.27. Дополнительные магические квадраты 3 х 3

- в углах двух прямоугольников 2 х 4 клетки (Рис. 5.28).

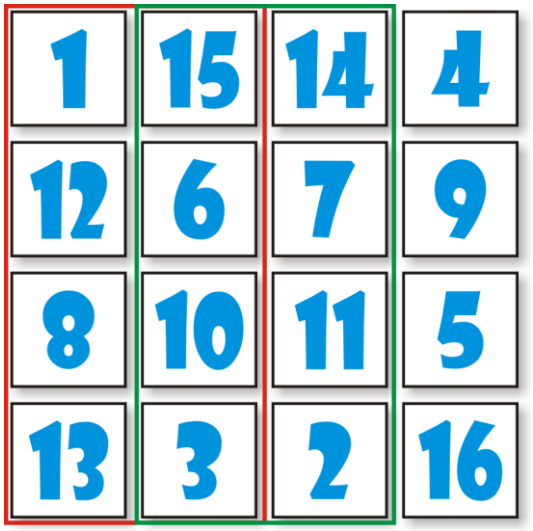


Рис. 5.28. Дополнительные магические прямоугольники 2 х 4

- в углах самого квадрата 4 х 4 клетки (Рис. 5.29).

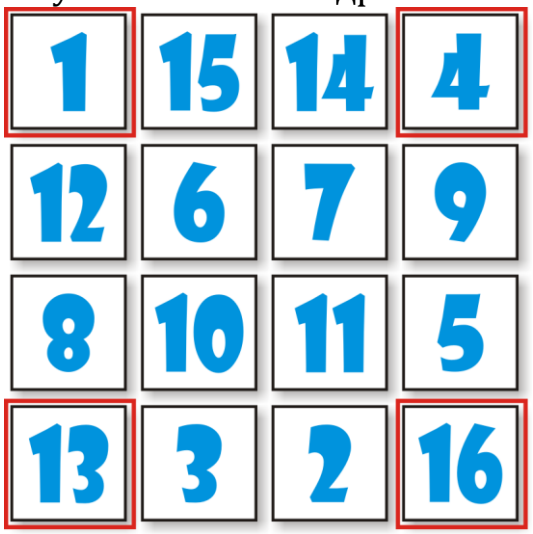


Рис. 5.29. Дополнительные магические прямоугольники 2 х 4

Итого – *22* раза. Но, оказывается, и это не предел. До предела мы дойдём, если выполним ещё два шага.

Шаг 5. Временно уберите *третью* и *четвёртую* строки. Переложите *вторую* строку на место *четвёртой*, а затем верните *третью* и *четвёртую* строки на свободные места. То есть третья строка займет в квадрате место *второй*, а четвёртая – *третьей* (Рис. 5.30).

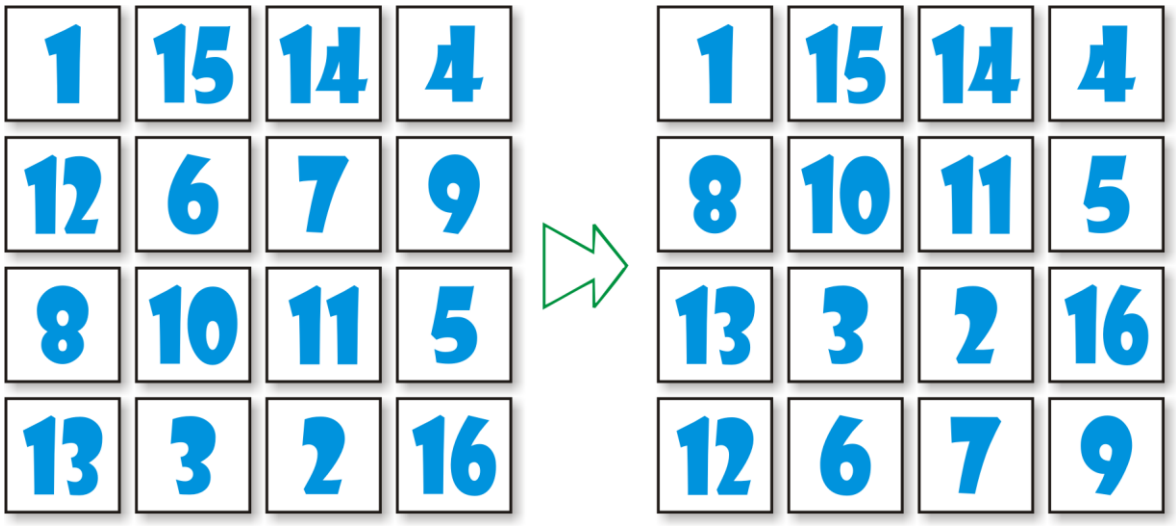


Рис. 5.30. Пятый шаг

Шаг 6. Поменяйте местами *третий* и *четвёртый* столбец (Рис. 5.31).

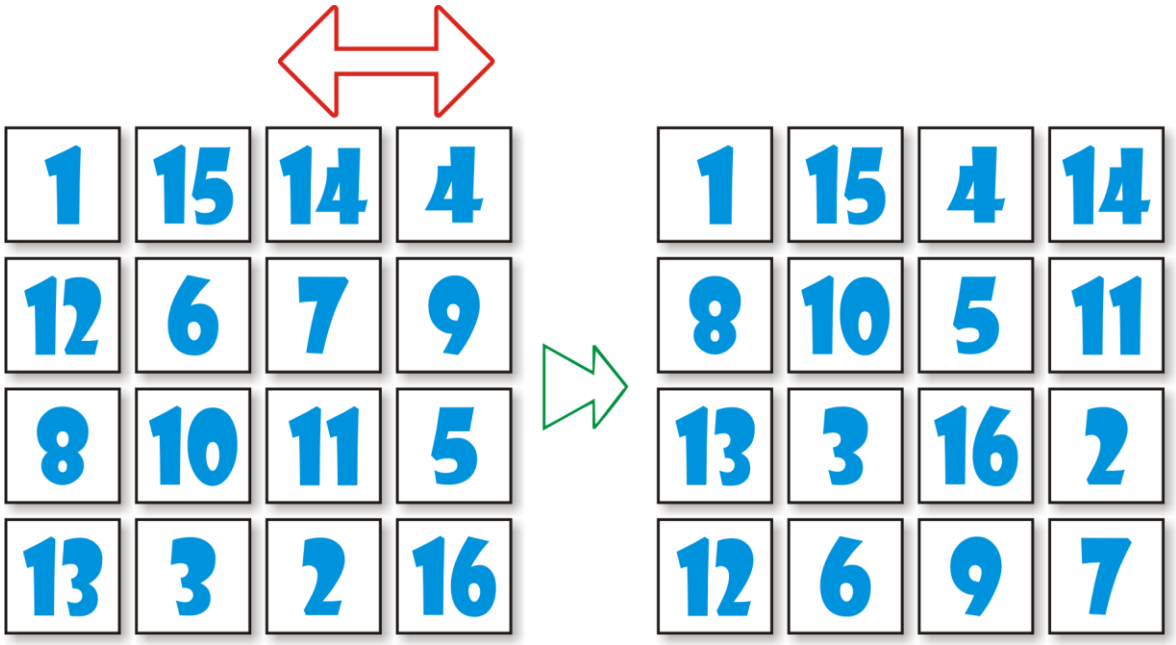


Рис. 5.31. Шестой шаг

Легко заметить, что все числа, кроме первых двух, заняли в квадрате другие места, и теперь магическая сумма повторяется уже *30* раз. Конечно, в

этом квадрате сохранились все 22 магические суммы прежнего квадрата, но к ним добавились ещё 11:

- в углах девяти (а не пяти) квадратов 2 x 2 клетки.
- в углах шести (а не двух) прямоугольников 2 x 4 клетки.

Найдите самостоятельно *все* магические суммы!

Проект *Чётно-чётные квадраты* (DEMS)

В уже упомянутой добрым словом книге С.Гудмана и С.Хидетниеми, на страницах 301-302 приведён алгоритм *DEMS* для составления любых *чётно-чётных* магических квадратов (то есть порядок квадрата должен быть кратен *четырём*).

Работа алгоритма начинается с расстановки *меток* в левой верхней четверти квадрата. В каждой строке и каждом столбце нужно поставить N/4 меток. Сделать это можно произвольно, но алгоритм *DEMS* ставит метки в шахматном порядке сразу во всей верхней половине квадрата (Рис. 5.32). В качестве метки используется *минус единица*.

-1	1	1	-1
1	-1	-1	1

Рис. 5.32. Метки расставлены

Мы видим, что метки из левого верхнего квадранта отражаются относительно вертикальной оси симметрии квадрата в правый верхний квадрант. Затем их нужно отразить относительно горизонтальной оси в нижнюю половину квадрата. Впрочем, описываемый алгоритм обходится без этой операции.

После того как метки расставлены, мы просматриваем все квадраты слева направо, сверху вниз. Если в очередной клетке метка *отсутствует*, мы ставим первое ещё не вышедшее число *i* из ряда 1..N^2. Отражаем клетку относительно центра квадрата и ставим в неё число $N^2 + 1 - i$. Если же в нём находится *метка*, то ставим число $N^2 + 1 - i$ - и число i в клетку-отражение. Следуя этому правилу, мы поставим в первую клетку число 16, а в последнюю – 1 (Рис. 5.33).

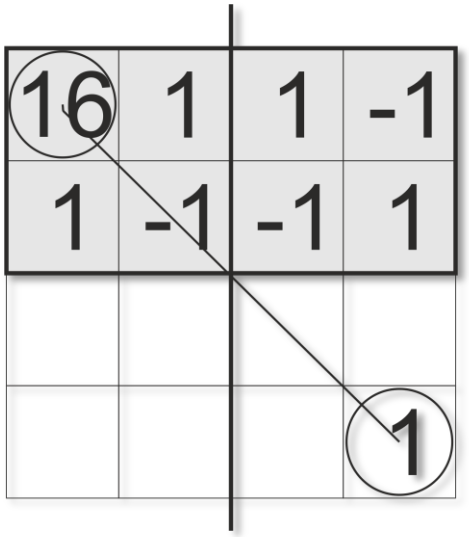


Рис. 5.33. Первая пара чисел на своих местах

В следующей клетке квадрата метки нет, поэтому в неё мы должны поставить двойку – единица уже вышла. Опять отражаем клетку относительно центра квадрата и вписываем число 16+1-2 = 15 (Рис. 5.34).

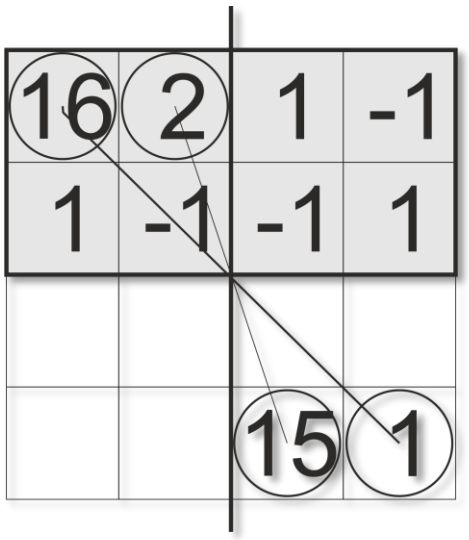


Рис. 5.34. И вторая пара чисел нашла своё место

В следующие клетки пойдут числа 3 и 14, и так мы продолжаем сканировать квадрат, пока не заполним его целиком (Рис. 5.35).

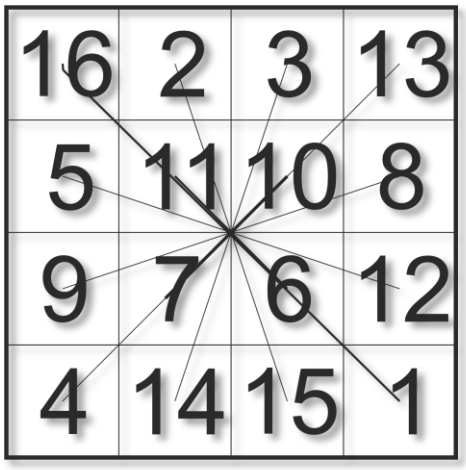


Рис. 5.35. Квадрат готов к употреблению

Вы, должно быть, обратили внимание, что способ построения магического квадрата таков, что сумма двух чисел, расположенных симметрично относительно центра квадрата, всегда равняется $N^2 + 1$. В нашем случае - семнадцати. Такие магические квадраты называются *ассоциативными*.

Ну, а теперь за дело! Немного подновим интерфейс приложения *Magic* и запишем алгоритм *DEMS* на языке *Си-шарп*:

```
//Algorithm DEMS
private void btnGen_Click(object sender, EventArgs e)
{
    //порядок квадрата:
    n = (int)udNum.Value;
    //создаем массив:
    matrix = new int[n*n+1];

    //генерируем магический квадрат:
    int halfn = n / 2;
    int mark = 1;
    int middle = n / 2;
    int upper = n * n + 1;
    int half = n * n / 2;

    //Ставим метки в первой половине массива:
    for (int k = 1; k <= halfn; ++k)
    {
      for (int j = 1; j <= halfn; ++j)
      {
        matrix[middle-j+1] = mark;
        matrix[middle+j] = mark;
        mark = -mark;
      }

        //Переходим к следующей строке:
        middle += n;
```

```
            mark = -mark;
        }
          //расставляем числа:
          for (int i = 1; i <= half; ++i)
          {
              //будущая координата числа i:
              int l=i;
              if (matrix[i] < 0)
                  l= upper-i;
              matrix[l] = i;
              matrix[upper - l] = upper - i;
              //matrix[l] = upper-i;
              //matrix[upper-l] = i;
          }

        //построение квадрата закончено:
        writeMQ();
        lstRes.TopIndex = lstRes.Items.Count - 27;
    }
```

Если вы хотите *перевернуть* квадрат, то раскомментируйте строчки

```
//matrix[l] = upper-i;
//matrix[upper-l] = i;
```

и закомментируйте пару строк над ними.

Здесь вместо *двумерного* массива *mq* используется *одномерный* массив *matrix*, поэтому *метод печати* **writeMQ** готовых квадратов нам придётся адаптировать к новым реалиям:

```
//Печатаем магический квадрат
void writeMQ()
{
  string s = "Магическая сумма = " + (n*n*n +n)/2;
  lstRes.Items.Add(s);
  lstRes.Items.Add("");
  // печатаем магический квадрат:
  for (int i= 1; i<= n; ++i)
  {
          s = "";
          for (int j= 1; j <= n; ++j)
          {
            int id = (i-1) * n + j;
            if (n*n > 10  && matrix[id] < 10) s += " ";
            if (n*n > 100 && matrix[id] < 100) s += " ";
            s= s + matrix[id] + " ";
          }
     lstRes.Items.Add(s);
```

```
    }
    lstRes.Items.Add("");
  }//writeMQ()
```

Запускаем новое приложение – оно работает, как кремлёвские часы (Рис. 5.36).

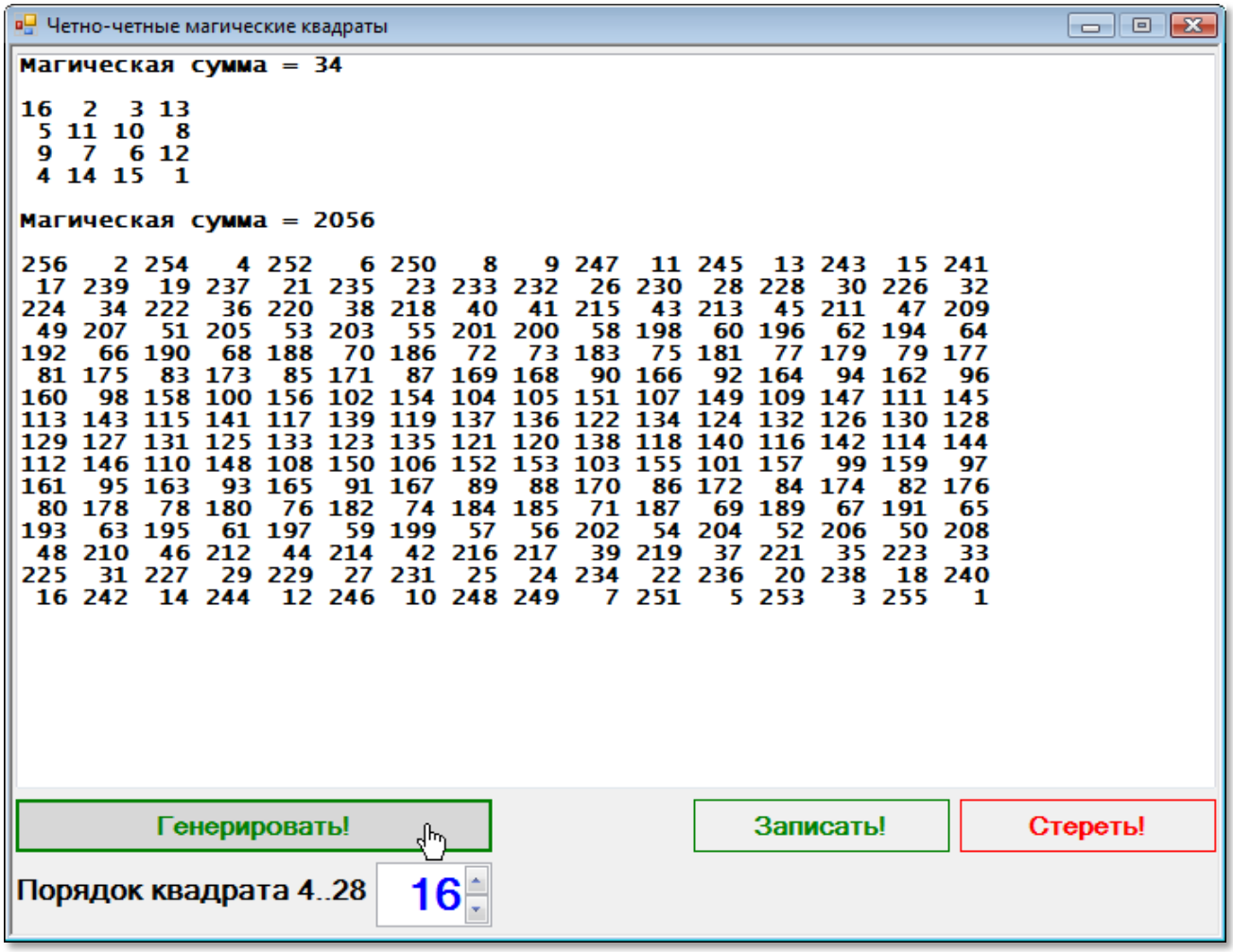


Рис. 5.36. Генерируем чётно-чётные магические квадраты

Картинку с новым *интерфейсом* программы я приводить не буду. Вы его легко получите из приложения *Magic*. Обратите только внимание на параметры компонента *udLen*!

Исходный код программы находится в папке **DEMS**.

Проект *4 x 4* (_4x4)

Шёл по улице отряд -
сорок мальчиков подряд:
раз,
два,
три,
четыре
и четырежды
четыре,
и четыре
на четыре,
и еще потом четыре.

Даниил Хармс, *Миллион*

Один магический квадрат может построить каждый. Потом его можно повертеть и получить ещё несколько экземпляров. А давайте мы отыщем *все* магические квадраты!

Если вспомнить, что магических квадратов пятого порядка чрезмерно много для полного поиска, то нам придётся ограничить свой порыв квадратами *четвёртого* порядка, которых всего 880 (уникальных, общее же их число 880 x 8 = 7040). Не очень много, но вот вариантов расстановки 16 чисел в квадрате 4 x 4 имеется 16! = 20 922 789 888 000, что совсем немало. Но мы справимся. Итак, за дело!

За основу мы возьмём наш предыдущий проект, хорошенько уменьшив ширину списка *lstRes*, поскольку квадратики будут совсем маленькие. Полей нам понадобится всего 5 штук, что ясно говорит о том, что программа будет несложной (ну, не очень сложной).

```
    //=true, если число использовано:
    bool[] flg;
    //порядок квадрата:
    int n = 4;
    //магическая сумма:
    int magsum;
    //магический квадрат:
    int[,] mq;
    //число найденных магических квадратов:
    int nVar;
```

В методе **btnGen_Click** мы просто создаём новые *массивы* для хранения магического квадрата и флажков, которые будут отмечать *использованные* числа:

```
//НАЖИМАЕМ КНОПКУ "ГЕНЕРИРОВАТЬ"
private void btnGen_Click_1(object sender, EventArgs e)
{
    //создаем числовой массив:
    mq = new int[n, n];
    flg = new bool[n * n + 1];
    //находим магическую сумму:
    magsum = (n * n * n + n) / 2;

    //пока не нашли ни одного квадрата:
    nVar = 0;
    //составляем квадрат, начиная с первого числа:
    recGenerate(1);
    //все квадраты найдены:
    lstRes.Items.Add("");
    lstRes.Items.Add("Всего вариантов " + nVar);
    lstRes.Items.Add("");
    lstRes.TopIndex = lstRes.Items.Count-27;
}
```

Сам поиск магических квадратов мы перенесём в *метод* **recGenerate**. Мы поступим мудро, если напишем его *рекурсивно*. Тогда он получится очень коротким, но «заковыристым»:

```
    //Рекурсивный метод составления магических квадратов
    void recGenerate(int hod)
    {
        int nw = 0;
        //перебираем все числа 1..n*n:
        for (int j = 1; j <= n * n; ++j)
        {
            if (test(hod, j))
            {
                nw = j;
                //нашли подходящее число-->
                //ставим его в квадрат:
                int row = (hod - 1) / n;
                int col = (hod - 1) % n;
                mq[row, col] = j;
                flg[j] = true;
                if (hod < n * n) recGenerate(hod + 1);
                else writeVar();
            }//if
            flg[nw] = false;
        }//for
    } //recGenerate
```

Самая большая проблема здесь – придумать эффективную проверку для каждого нового числа! В самом деле, если мы будем просто последова-

тельно расставлять 16 чисел по клеткам квадрата и только потом, проверять, получился ли у нас магический квадрат, то жизнь будет прожита даром.

Но мы крепко сузим круг претендентов, если последнее число каждой строки будем проверять так. Найдём сумму этого числа и ещё трёх, которые мы поставили раньше. Мы знаем, что в магическом квадрате сумма чисел во всех строках, столбцах и в двух главных диагоналях равна магической сумме, которую мы умеем вычислять. Таким образом, если сумма чисел какой-либо строки вместе с числом-претендентом не равна магической сумме, то это число ставить в квадрат нет никакого смысла. Эта проверка обеспечит нам магическую сумму во всех строках.

Но этого мало, поэтому, когда мы дойдём до *последней* строки, то будем проверять также вертикали и диагонали. Вот теперь любой составленный нашей программой квадрат будет *магическим*:

```
//ПРОВЕРКА ЧИСЛА num
bool test(int hod, int num)
{
    if (flg[num]) return false;
    int col = (hod - 1) % n;
    //номер строки:
    int row = (hod - 1) / n;
    int sum = 0;
    if (col == n - 1) //последнее число в строке
    {
        sum = 0;
        for (int i = 0; i < n - 1; ++i)
            sum += mq[row, i];
        if (sum + num != magsum) return false;
        //else return true;
    }

    //последняя строка □ вертикали:
    if (row == n - 1)
    {
        sum = 0;
        for (int j = 0; j < n - 1; ++j)
            sum += mq[j, col];
        if (sum + num != magsum) return false;
    }

    //диагонали:
    if (row == n - 1 && col == 0) //восходящая
    {
        sum = 0;
        for (int j = 1; j < n; ++j)
```

```
                sum += mq[n - 1 - j, j];
            if (sum + num != magsum) return false;
        }
        if (row == n - 1 && col == n - 1) //нисходящая
        {
            sum = 0;
            for (int j = 0; j < n - 1; ++j)
                sum += mq[j, j];
            if (sum + num != magsum) return false;
        }

        return true;
    }
```

Довольно быстро мы получим полный список из 7040 магических квадратов четвёртого порядка (Рис. 5.38).

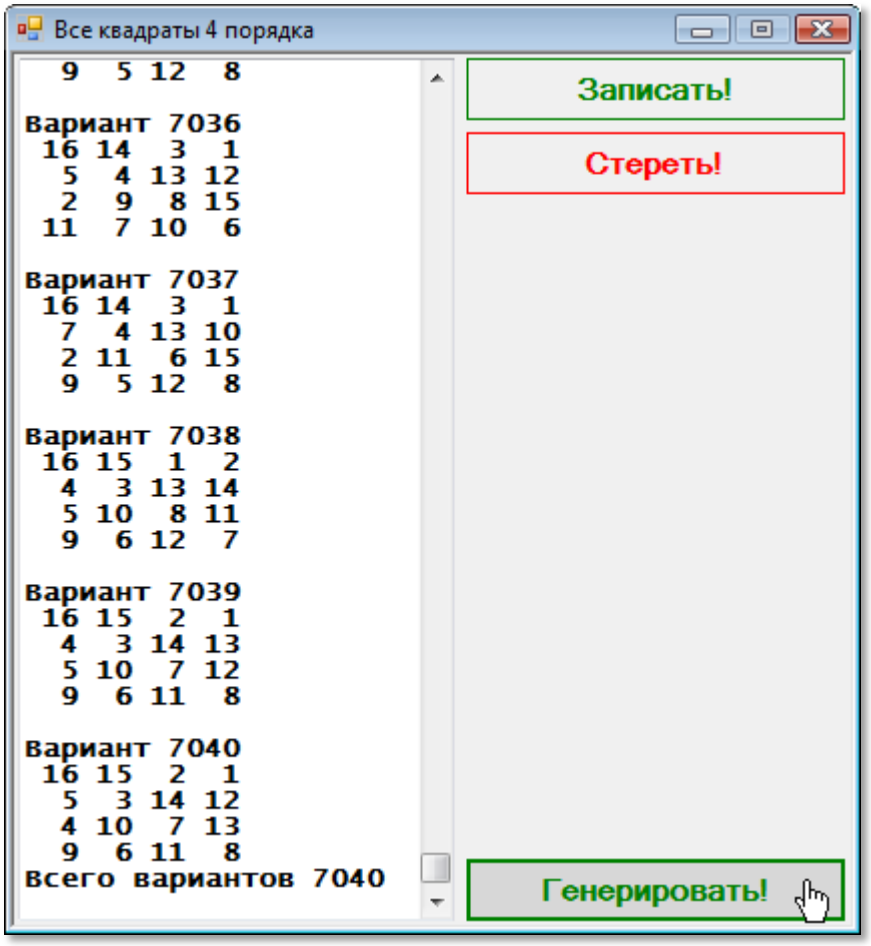


Рис. 5.38. Все четырежды четырки найдены!

Совершенно немыслимо составлять таким способом квадраты 5 х 5 клеток. Как говорил анекдотический Рабинович: «Не дождётесь!»

Интерфейсом программы займитесь самостоятельно. Здесь компонент *udLen* не нужен, так что его следует удалить.

Исходный код программы находится в папке **_4x4**.

1. С помощью программы *UniGen* вы можете составить квадраты любого размера, но только по *одному* для каждого порядка. Увы, все подобные алгоритмы действуют именно так. Но мы всё-таки можем внести некоторое разнообразие в наши квадраты. Например, если прибавить или вычесть одно и то же число (или умножить на одно и то же число), то квадрат останется магическим, а его магическая сумма изменится. Научите программу составлять такие квадраты.

2. Можно также *поворачивать* квадрат на 90 градусов и *отражать* его относительно горизонтальной и вертикальной осей. Поскольку эти операции проводятся над уже готовыми квадратами, то это задание совсем несложное.

3. Каждый магический квадрат имеет *дополнение* – другой магический квадрат, который получается из исходного, если все числа в нем вычесть из N^2+1.

4. Для квадратов *пятого* порядка известны 2 *преобразования*, порождающие новые магические квадраты.

Переставьте столбцы 1 и 5, затем переставьте строки 1 и 5.

Переставьте строки 1 и 2, а также 4 и 5, а затем столбцы 1 и 2, 4 и 5.

5. Усовершенствуйте приложение *DEMS*, чтобы оно расставляло метки в левом верхнем квадранте *случайно*. Так вы добьётесь большего разнообразия в готовых изделиях. Не более сложно заставить приложение расставлять метки *всеми* возможными способами, чтобы можно было составить *много* разных квадратов.

6. Ответы на *Пентамагики* вы найдёте в конце книги, но почему бы вам не написать приложение, которое могло бы самостоятельно составлять и решать такие головоломки?

Глава 6. Словесные магические квадраты

Если *числовые* магические квадраты «официально» признаны *первой* комбинаторной задачей в истории человечества и известны всем и каждому, то *словесные* магические квадраты популярностью не избалованы, хотя и появились всего на несколько столетий позже знаменитого квадрата Ло-шу, ещё до нашей эры.

В словесных магических квадратах все слова можно прочитать *дважды* – по горизонтали и по вертикали. Причём каждое слово *пересекается* со всеми остальными, так что такие магические квадраты дали начало самой популярной головоломке в мире - кроссвордам! Шустрые англосаксонцы по обыкновению приписали изобретение кроссвордов себе, но, как мы ясно видим, это опрометчивое утверждение можно охарактеризовать русской пословицей *Поспешишь – людей насмешишь*!

Самый знаменитый из всех словесных магических квадратов называют *Сатор* – по первому слову в этом квадрате (Рис. 6.1).

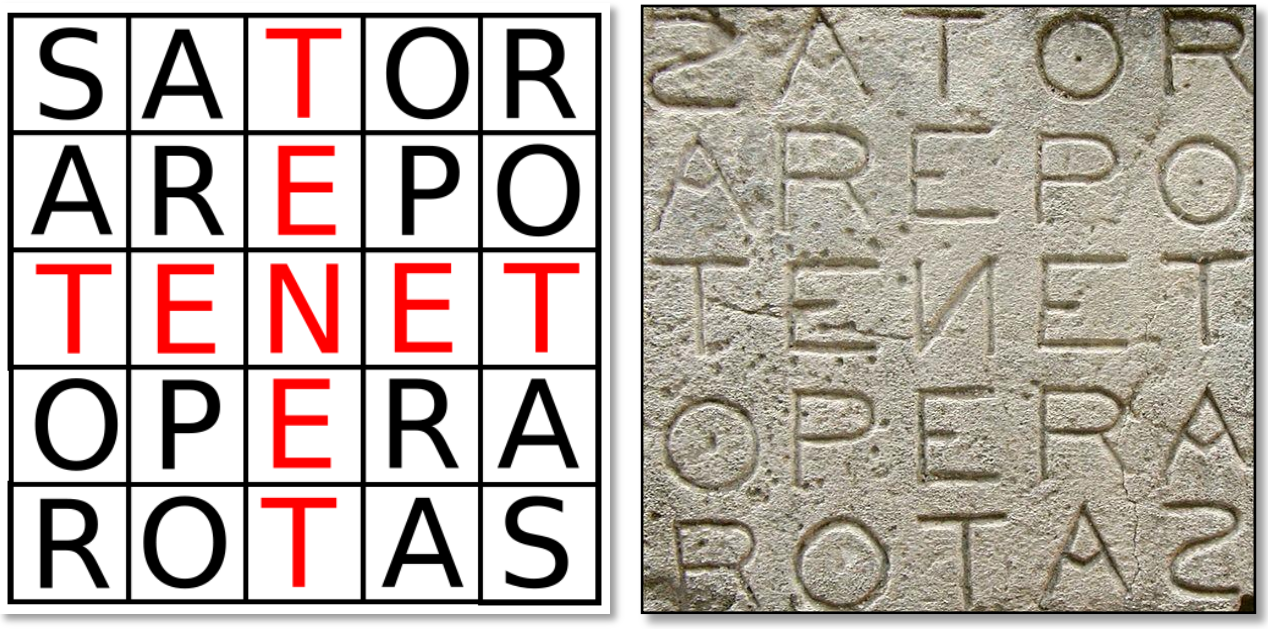


Рис. 6.1. Словесный магический квадрат *Сатор*

Точный перевод фразы *SATOR AREPO TENET OPERA ROTAS* неизвестен, но обычно его переводят с латыни как *Сеятель Арепо с трудом держит колеса*. Можно заметить, что эта фраза одинаково читается и в обратном направлении, то есть является *палиндромом*!

Удивительное дело: романе Умберто Эко *Маятник Фуко* мы найдём убийцу, которого зовут … Сатор Арепо:

- Сатор Арепо! — кричу я голосом, который бы заставил задрожать даже гром.

Отложив в сторону колесо, которое он так уверенно держал своими руками убийцы, Сатор Арепо поспешил предстать предо мной. Я узнаю его, впрочем, я и так подозревал, кто он такой. Это Лучано, калека-экспедитор, которого Неведомые Настоятели назначили исполнителем моей гнусной и кровавой задачи.

Словесные магические квадраты с отдельными словами-палиндромами и палиндромоидами не редкость, поскольку их составить куда проще, чем с осмысленными фразами (Рис. 6.2).

R	A	T	S
A	B	U	T
T	U	B	A
S	T	A	R

Рис. 6.2. Словесный магик по-английски

Подобно числовым магическим квадратам, словесный квадрат Сатор использовался в средние века как *талисман*. Первые христиане из тех же букв находчиво составили крест *Pater Noster* (*Отче наш*) (Рис. 6.3). Лишние буквы *A* и *O* обозначают в нём начало и конец (альфа и омега) всего сущего. По-моему, весьма остроумное решение проблемы.

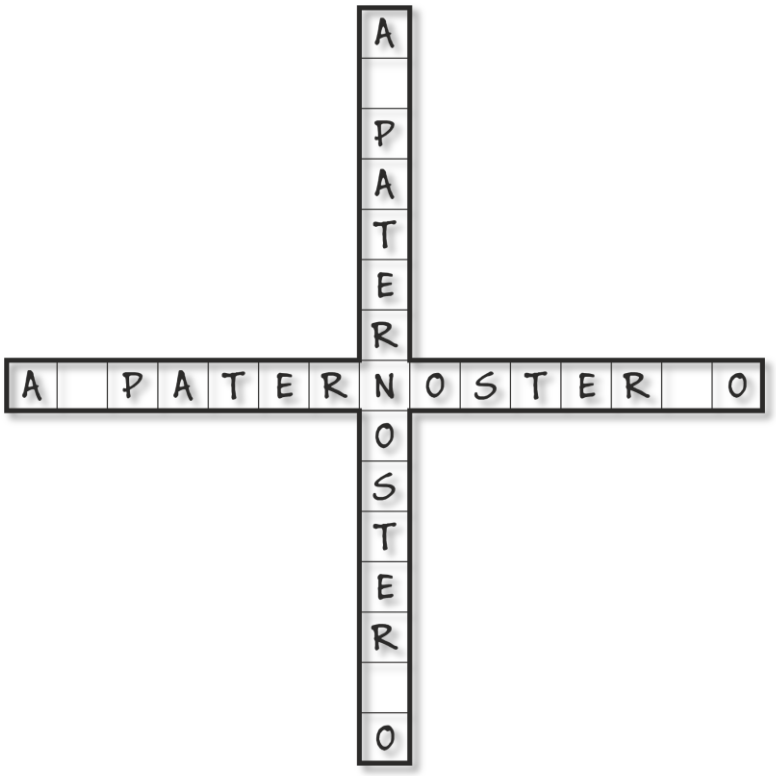


Рис. 6.3. Магический крест

Другие магические квадраты находили применение в *магических обрядах*. Вот пример такого квадрата (Рис. 6.4). Он, якобы, превращал волшебника в ворону, чтобы тот мог летать.

R	O	L	O	R
O	B	U	F	O
L	U	A	U	L
O	F	U	B	O
R	O	L	O	R

Рис. 6.4. Волшебный магический квадрат!

Впрочем, такие квадраты малоинтересны, поскольку они составлены из бессмысленных (по крайней мере, для непосвященных в таинства) слов.

Столетия спустя составление словесных магических квадратов стало излюбленным занятием личностей, склонных к литературному творчеству.

Задача непростая: собрать из «нормальных» слов магический квадрат как можно большего размера. Правда, зачастую в ход идут и редкие слова, и формы слов, а не только имена существительные, как это обычно принято в подобных головоломках. Как мы вскоре убедимся, конструирование квадратов из 2-6-буквенных слов не представляет большого труда (Рис. 6.5).

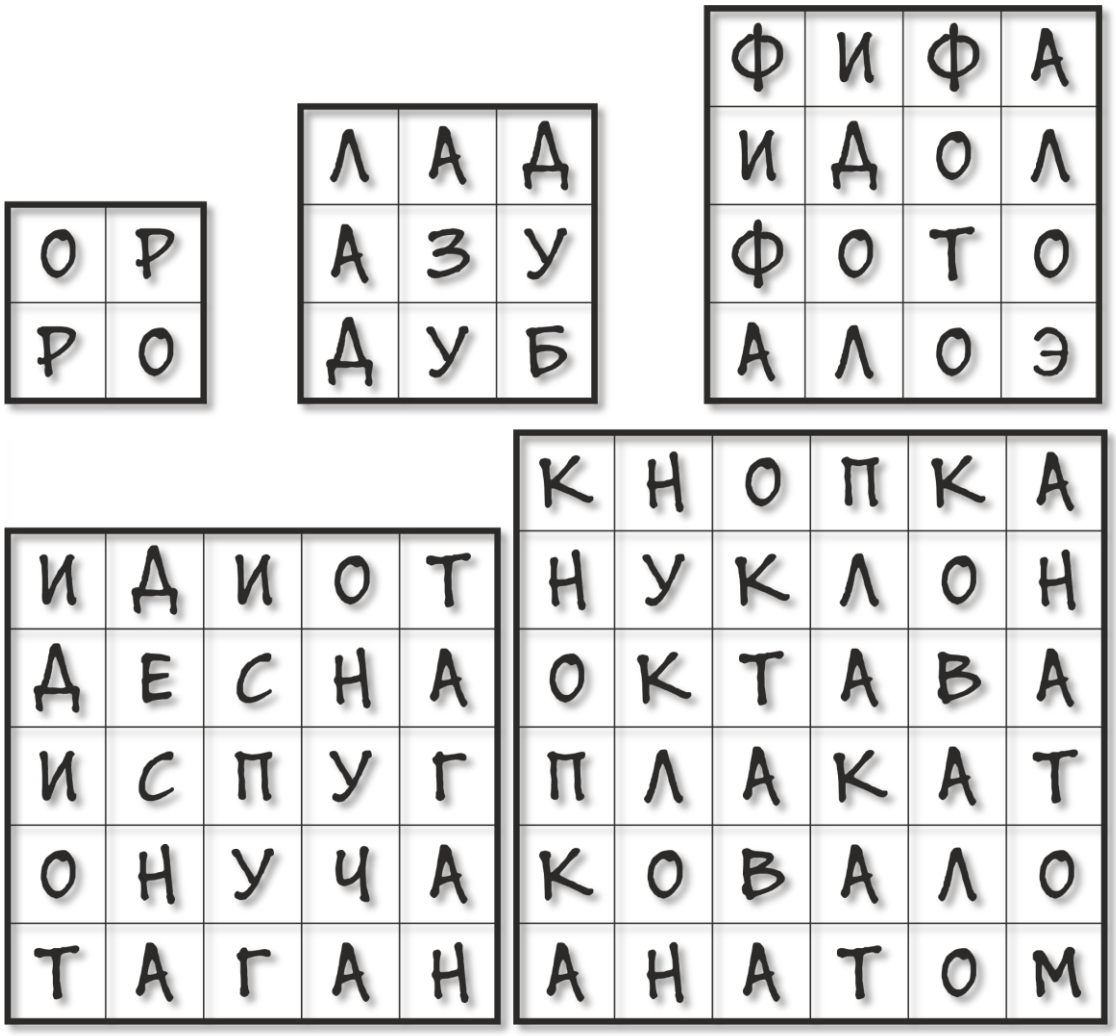


Рис. 6.5. Размер имеет значение!

> В английском языке букв меньше, поэтому англичане – не без труда! - добрались до квадрата *девятого* порядка. Посмотреть на него вы можете здесь: *http://en.wikipedia.org/wiki/Word_square*

Конечно, после такого исторического экскурса трудно отказать себе в удовольствии и не написать программу, которая умела бы составлять такие удивительные квадраты!

Проект *Магические слодраты* (WordSquares)

Мы сразу откажемся от глобальной задачи – составить вообще *все* возможные квадраты, поскольку труд окажется непосильным. Мы предоставим пользователю возможность самому выбрать *первое* слово для квадрата, а все остальные будет подбирать наша программа.

Нетрудно догадаться, что нам потребуется *словарь*, из которого мы и будем выбирать слова для квадрата. Все слова в квадрате имеют равную длину, поэтому вполне разумно воспользоваться фракционным словарем.

Б***о***льшая часть *полей* нашей программы как раз и будет задействована при загрузке словаря и хранении списка подходящих слов. Другие поля понадобятся нам для работы с магическим квадратом – для хранения порядка квадрата, сл***о***ва пользователя и массива слов в квадрате:

```
//ПРОГРАММА ДЛЯ СОСТАВЛЕНИЯ
//СЛОВЕСНЫХ МАГИЧЕСКИХ КВАДРАТОВ
//порядок квадрата:
int N = 0;
Random rand = new Random();
//словарь:
rvDict dict;
//магический квадрат:
string[] mq;
//число подходящих слов:
//int nLex;
//число найденных магических квадратов:
int nVar;
//список подходящих слов:
List<string> lex= new List<string>();
//=true, если слово использовано:
List<bool> flg= new List<bool>();
```

Словарь по умолчанию лучше загрузить сразу же при запуске приложения, чтобы не заставлять пользователя делать это самостоятельно. А затем он может выбрать любой словарь из папки с программой или загрузить собственный.

```
public frmWS()
{
    InitializeComponent();
    //загружаем словарь:
    dict = new rvDict("frc.txt");
    dict.CreateBeginWord();
}
```

В текстовое поле **txtWord1** пользователь должен впечатать первое слово для квадрата. Мы должны разумно ограничить его длину. Действительно, квадрат из одной буквы составить можно и без помощи компьютера, а для квадрата из семибуквенных слов практически невозможно наугад выбрать первое слово.

После нажатия на *кнопку* **btnGen** мы переходим в *метод* **btnGen_Click**, где по длине слова легко определяем *порядок* квадрата и создаём новый строковый массив для него. Первое слово нам уже известно, поэтому мы тут же записываем его в квадрат:

```
//СОСТАВЛЯЕМ МАГИЧЕСКИЙ КВАДРАТ
private void btnGen_Click(object sender, EventArgs e)
{
  //первое слово:
  string  word1 = txtWord1.Text.ToUpper();
  //Длина слова:
  N = word1.Length;

  //если слово отсутствует, выбираем случайно из словаря:
  if (word1 == "")
  {
     N = (int)udLength.Value;
     int id= rand.Next(dict.beginWord[N],
             dict.beginWord[N]+dict.numWord[N]);
     word1 = (string)dict.list[id];
     txtWord1.Text = word1.ToUpper();
  }
  else if (N < 2 || N > 6)
  {
      MessageBox.Show("В слове должно быть от двух до шести
                       букв!");
      return;
  }
  //создаем массив для слов:
  mq = new string[N + 1];
  mq[1] = word1;
  flg.Clear();

  //составляем список подходящих слов:
  //nLex = 0;
  string letters = word1.Substring(1, N - 1);
  lex.Clear();

  //длина всех слов должна быть равна порядку квадрата n:
  for (int i = dict.beginWord[N]; i < dict.beginWord[N] +
               dict.numWord[N]; ++i)
  {
      //не включаем в список первое слово:
```

```
        if ((string)dict.list[i] == word1) continue;
        //слово должно начинаться с буквы в letters:
        char ch = ((string)dict.list[i])[0];
        //если нашли слово,
        if (letters.IndexOf(ch) >= 0)
        {
            //добавляем его в список:
            lex.Add((string)dict.list[i]);
            //слово не использовано:
            flg.Add(false);
        }
    }
    //пока не нашли ни одного квадрата:
    nVar = 0;
    //составляем квадрат, начиная со второго слова:
    recGenerate(2);
    lstRes.Items.Add("");
    lstRes.Items.Add("Найдены все решения: " + nVar.ToString());
    lstRes.Items.Add("");
    lstRes.TopIndex = lstRes.Items.Count - 27;
}
```

Так как словарь *dict* довольно большой, а нам придётся перебирать все слова заданной длины, то лучше сразу об этом призадуматься и постараться уменьшить перебор. Это очень просто! Первое слово квадрата даёт нам начальные буквы всех остальных слов. Это легко понять, если посмотреть, как составлен магический квадрат *Sator*. Нам известна длина слов и их начальные буквы, поэтому мы составляем словарик *lex*, в который включаем только те слова, которые *могут быть* поставлены в квадрат. Он, безусловно, будет значительно короче общего списка слов. Также мы не забываем заполнить список флажков «фальшивыми» значениями, которые отмечают еще неиспользованные слова.

Иногда первое слово не приходит в голову или его нужно выбрать случайно. Поэтому мы установим на форме компонент **udLength**, в котором удобно задавать длину слов (или порядок квадрата). Но тут может возникнуть непреднамеренная коллизия со словом в поле *txtWord1*, поскольку программа будет теряться в догадках, выбирать ли ей слово случайно или брать то, которое ввёл пользователь. Чтобы рассеять её сомнения, следует очистить поле *txtWord1*, дважды кликнув на нём:

```
private void txtWord1_MouseDoubleClick(object sender, MouseEven-
tArgs e)
{
   txtWord1.Text = "";
```

```
}
```

Вот теперь программа поймёт вас правильно и выберет слово случайно.

И вот у нас всё готово для составления магического квадрата, и мы отправляемся прямиком в рекурсивный метод *recGenerate*.

Конечно, метод должен быть именно *рекурсивным*, поскольку действия при составлении квадрата практически полностью повторяются для каждого нового слова: ставим второе слово с учётом первого, затем ставим третье слово с учётом первых двух, и так далее, пока квадрат не будет построен или мы не убедимся, что его построить нельзя.

Единственная трудность здесь - в организации *проверки* слов из списка: годятся ли они для квадрата или нет.

Опять обратимся к магическому квадрату *Sator*. В нашем случае *первое* слово уже стоит в квадрате:

```
sATOR
A....
T....
O....
R....
```

Хорошо видно, что второе слово оно должно начинаться с буквы *A* (*второе* слово – *вторая* буква в исходном слове), а все остальные четыре буквы могут быть любыми.

Ставим *второе* слово:

```
saTOR
arEPO
TE...
OP...
RO...
```

Третье слово должно начинаться с букв *TE* - а это *третья* буква *первого* слова и *третья* буква *второго* слова.

Если решение ещё не пришло вам в голову, то продолжайте составлять квадрат. Рано или поздно вы найдёте простой алгоритм для составления *подстроки* **shablon**, с которой и должно начинаться очередное слово:

```
 //создаём шаблон для слова num:
 string shablon = "";
 for (j = 1; j < num; ++j)
     shablon += mq[j].Substring(num - 1, 1);
```

Остальное, как говорится, дело техники, и рекурсивный метод готов:

```
//Рекурсивный метод составления магических квадратов
void recGenerate(int num)
{
    int j = 0;
    //создаем шаблон для слова num:
    string shablon = "";
    for (j = 1; j < num; ++j)
        shablon += mq[j].Substring(num - 1, 1);

    //перебираем все слова в списке lex:
    foreach (string s in lex)
    {
        //оно должно начинаться с букв shablon:
        if (s.StartsWith(shablon))
        {
            //нашли подходящее слово-->
            //ставим его в квадрат:
            mq[num] = s;
            flg[j] = true;
            if (num < N) recGenerate(num + 1);
            else writeVar();
            ++j;
        }//if
        flg[j] = false;
    }//for
} //recGenerate
```

Найденные магические квадраты мы *печатаем* в списке *lstRes*:

```
//ПЕЧАТАЕМ МАГИЧЕСКИЙ КВАДРАТ
void writeVar()
{
    //нашли еще один магический квадрат:
    nVar++;
    lstRes.Items.Add("");
    lstRes.Items.Add("Вариант: " + nVar.ToString());
    for (int i = 1; i <= N; ++i) lstRes.Items.Add(mq[i]);
    //прокручиваем список вниз:
```

```
    lstRes.TopIndex = lstRes.Items.Count - 27;
    lstRes.Invalidate();
    Application.DoEvents();
}
```

Ну вот и пришло время позабавиться!

Запускаем программу и вводим слова из 2-6 букв. Результат получаем мгновенно (Рис. 6.6).

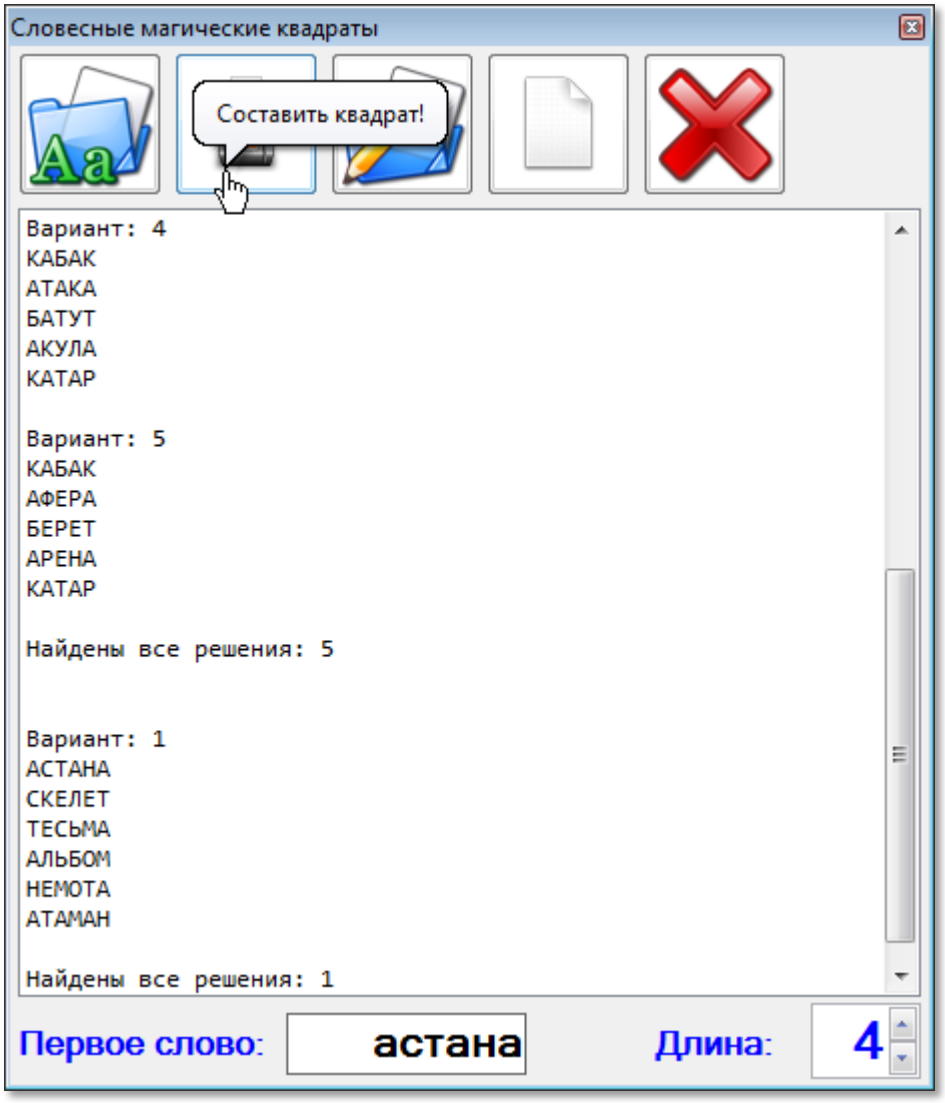


Рис. 6.6. Магическое производство налажено!

Так мы можем составить сколько угодно магических квадратов!

Если вам наскучит словарь по умолчанию, вы в любой подходящий момент можете загрузить другой, нажав *кнопку* **btnLoadDict**:

```
//ЗАГРУЖАЕМ СЛОВАРЬ ПО ВЫБОРУ
private void btnLoadDict_Click(object sender, EventArgs e)
{
    dict = new rvDict();
    dict.Load();
    //фракционируем список:
    //dict.Fract();
    dict.CreateBeginWord();
}
```

Для составления *больших* магических квадратов нужен и солидный словарь. Например, с файлом *SSSRLfrc.txt* я нашёл 21 решение для ведущего слова *МОСКВА* (Рис. 6.7), тогда как словарь по умолчанию не находит вообще ни одного!

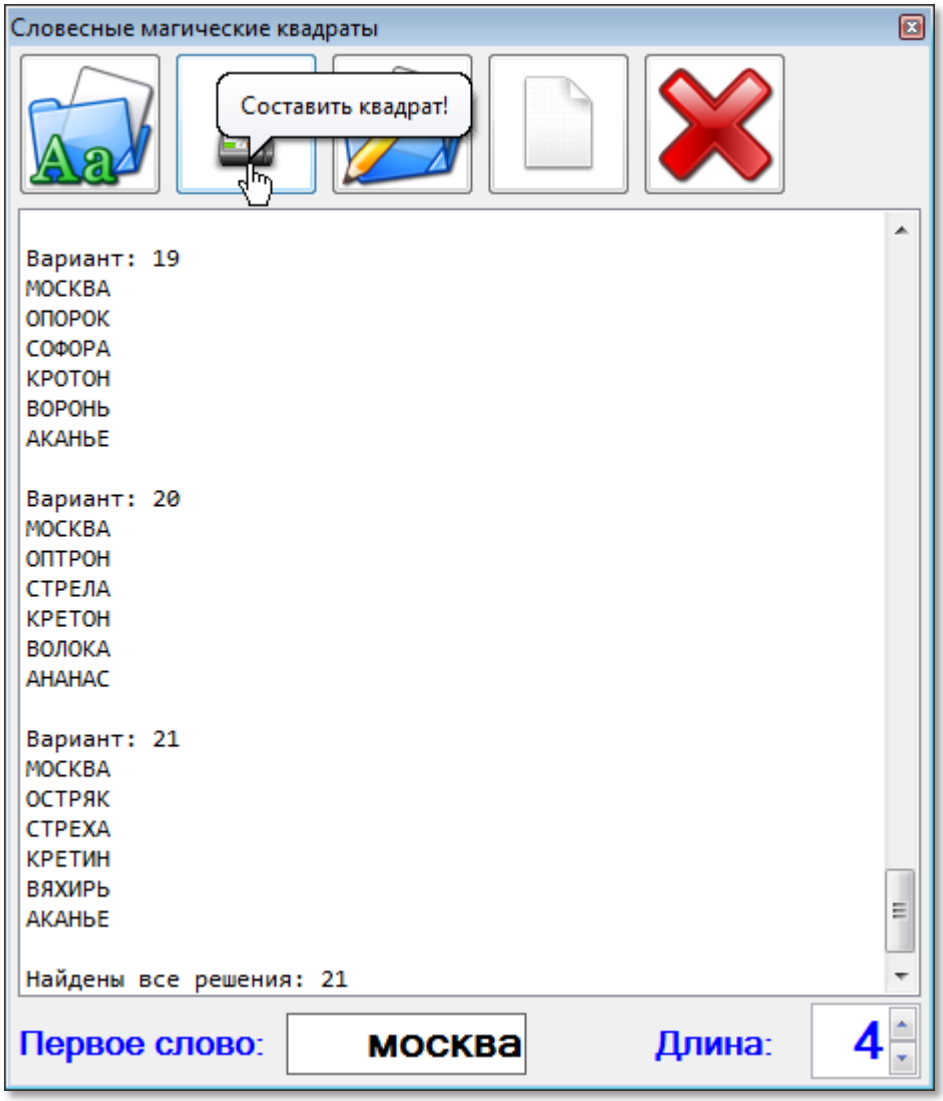


Рис. 6.7. Москва – как много в этом!..

А с иностранными словарями, например *DeutschUnicodeFRC.txt*, вы можете составлять магические квадраты на чуждых вам языках, даже не зная ни одного иностранного слова (Рис. 6.8)!

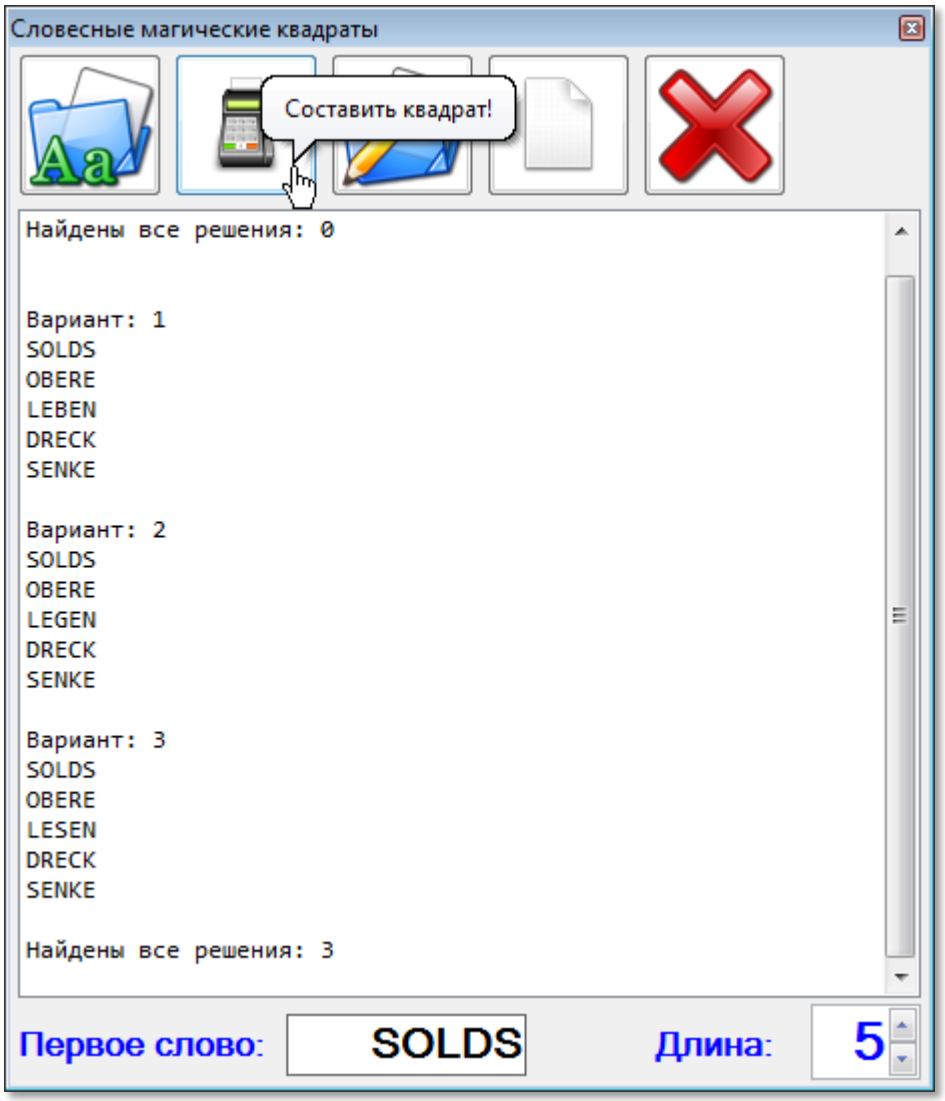
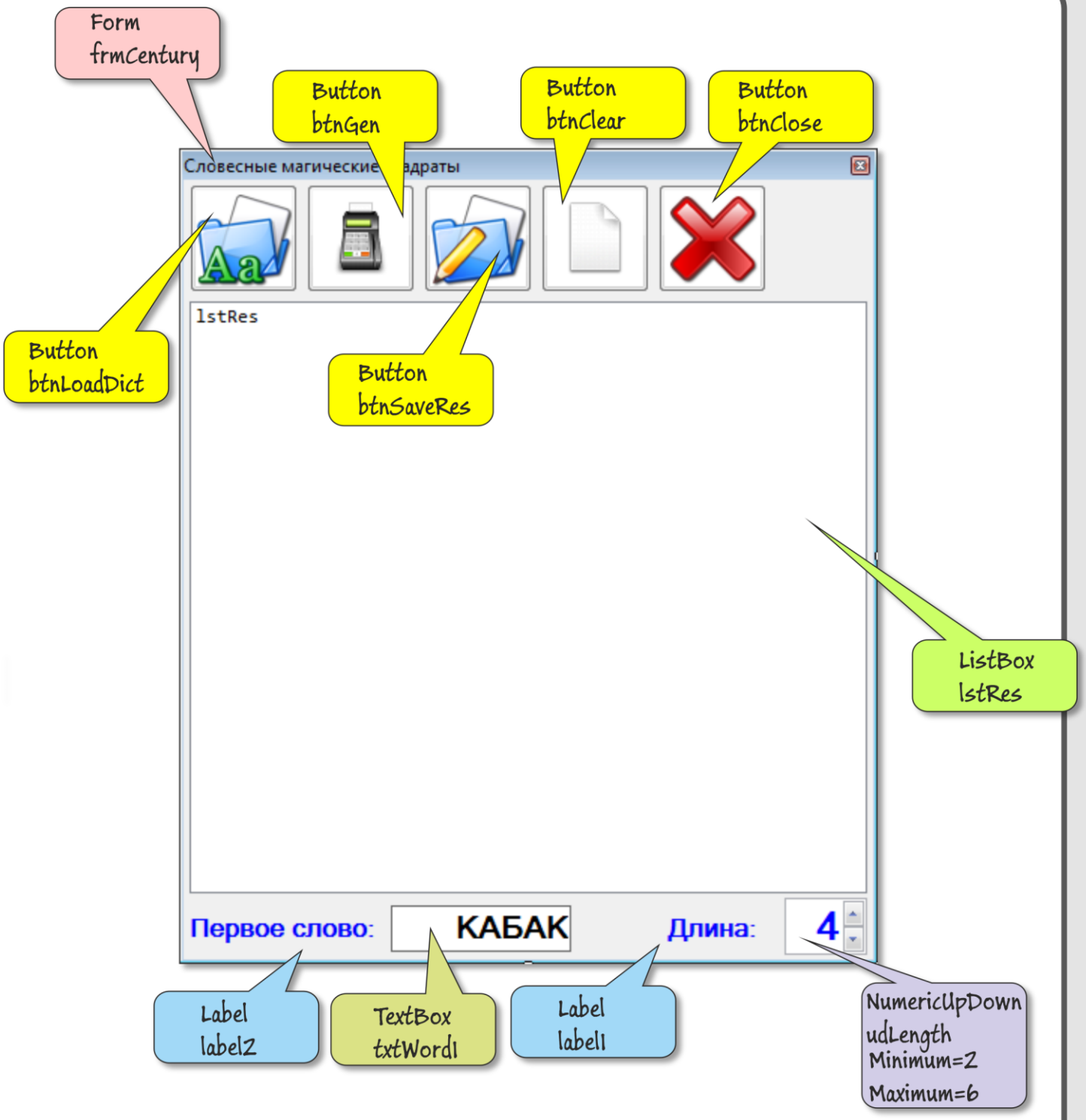


Рис. 6.8. Шпрехен Зи дойч?

Исходный код программы находится в папке **WordSquares**.



Примерный интерфейс приложения

1. Хотя наш алгоритм работает достаточно быстро, но при поиске *всех* квадратов заданного размера он будет неуклюже составлять список подходящих слов для каждого нового начального слова. Лучше один раз и навсегда запомнить индексы начала не только слов заданной длины, но и каждой буквы в отдельности. Тогда при необходимости вы сразу выйдете на нужную часть списка. Для

хранения начальных индексов годится структура данных *Dictionary<>,* а вот для индексов начала и конца слов, а также флажка их использования придётся завести *структуру*.

Ещё одна возможность оптимизации – записывать слова в сетку не последовательно, а выбирать такую начальную букву, на которую имеется меньше всего подходящих слов. Например, если первое слово ПАРОЛЬ, то нет никакого смысла перебирать начальные слова, поскольку на мягкий знак слов нет вообще.

2. В нынешнее время словесные квадраты утратили свою магическую силу и используются исключительно как *головоломки*.

В простейшем варианте – это обычный *кроссворд* (Рис. 6.9). Иногда такие задания встречаются в головоломных журналах, но магические квадраты здесь сильно уступают своим потомкам: они маленькие по размерам и все слова в сетке *повторяются* дважды, что не очень хорошо действует на умных людей. Совсем маленькие квадраты и вовсе неинтересны, а большие трудно составлять, поэтому в них неизбежно попадают редкие слова.

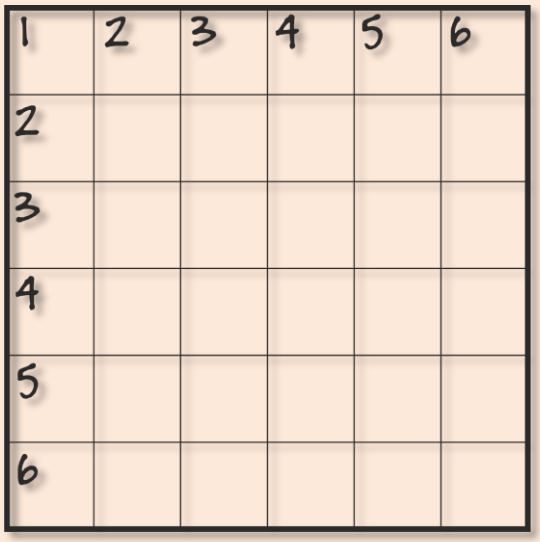
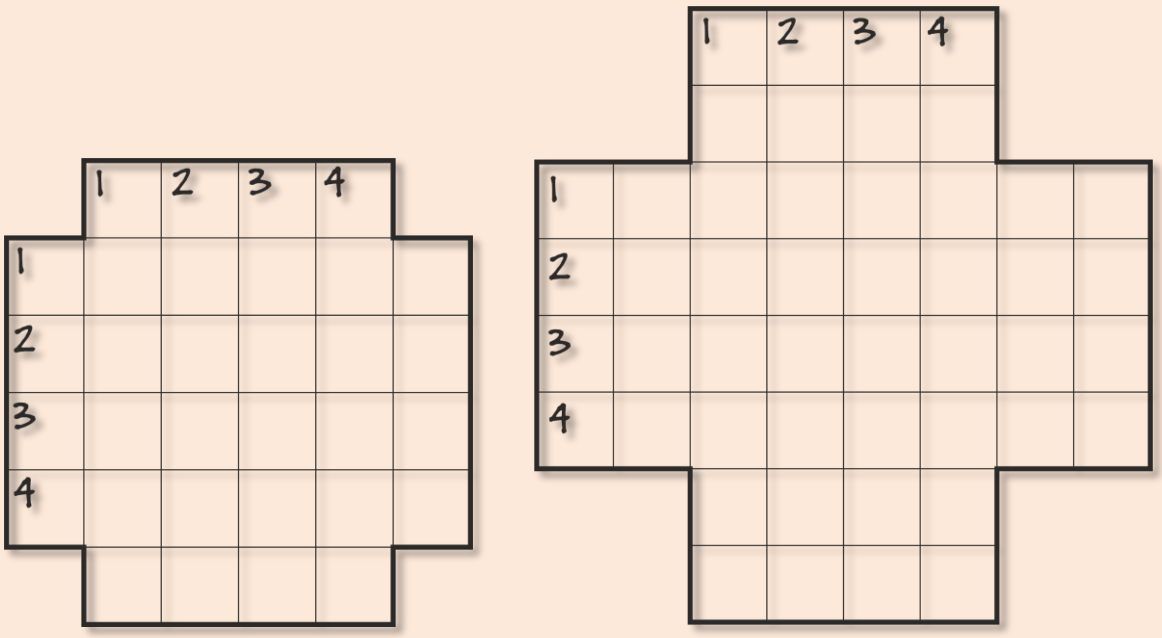


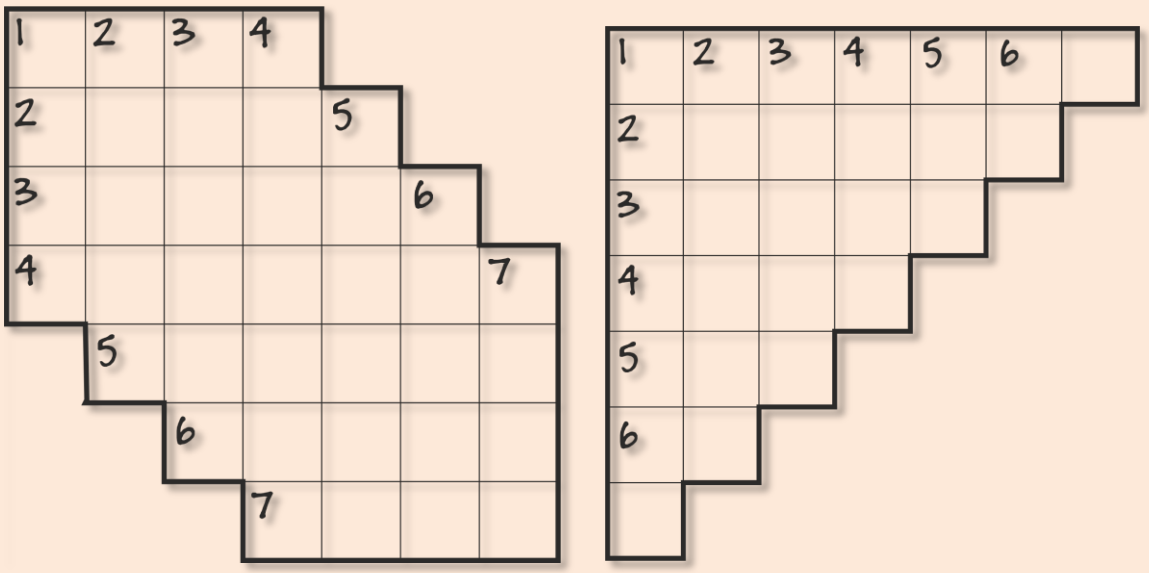
Рис. 6.9. Магический кроссворд

1. Вот тебе проблема!
2. Приятный запах денег.
3. Главный герой третьей главы.
4. Жидкий ароматизированный крахмал для машинной и ручной стирки.
5. Традиционное блюдо грузинской кухни.
6. Потеря тонуса мышц.

Некоторое разнообразие привносят новые формы сетки, отличные от квадратной (Рис. 6.10):



Магические кресты



Магическая лестница Магический треугольник

Рис. 6.10. Магические фигуры

Другой тип заданий скорее логический и напоминает знаменитые пазлы. Нужно правильно расставить слова в пустом квадрате (Рис. 6.11).

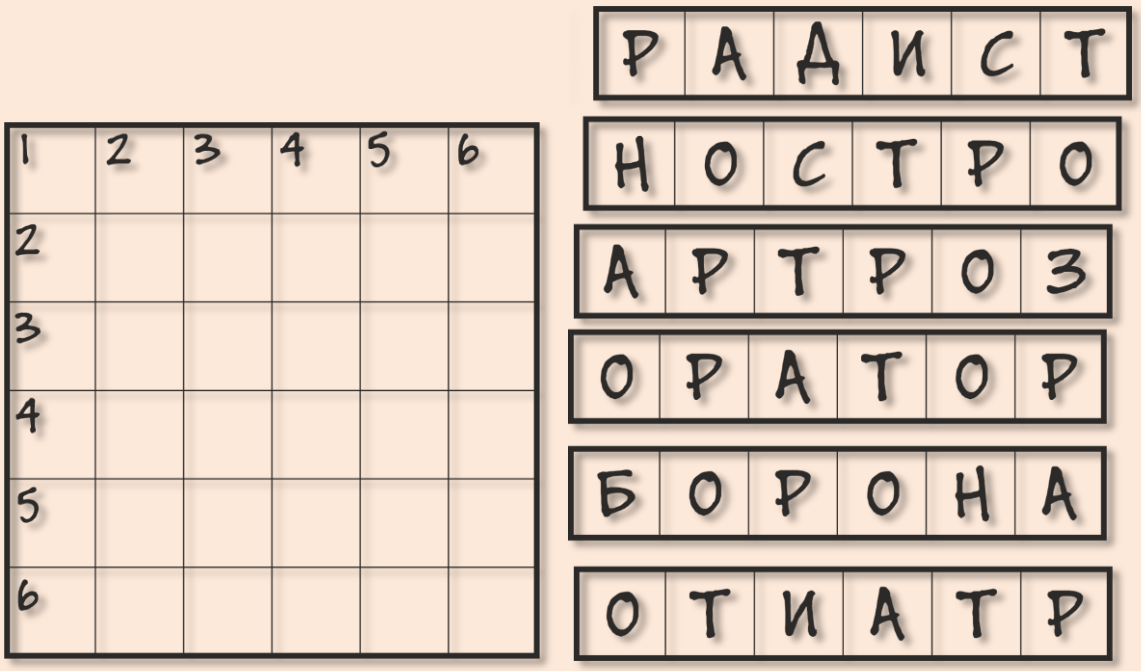


Рис. 6.11. Магический пазл

И наконец, самая популярная головоломка из этой серии. Первое слово записано в квадрат (Рис. 6.12).

Рис. 6.12. Магическая сетка

Сетка сопровождается таблицей, в которой указаны буквы и их число – для тех слов, которые нужно найти (Рис. 6.13).

А	Е	И	К	Л	Н	О	Р	С	Ф	Ц
2	4	4	3	2	2	2	2	1	2	1

Рис. 6.13. Магическая статистика

Последняя строка этой таблицы оставлена пустой, чтобы отгадчик мог отмечать вышедшие буквы. По этим данным нужно «вычислить» все слова и заполнить сетку.

Эта задача также логическая, но предполагает и знание слов. Легко догадаться, что табличная буква, присутствующая в *единственном* числе, должна стоять на главной нисходящей диагонали квадрата, а остальные буквы располагаются симметрично относительно этой диагонали.

Глава 9. Считаем деньги

Махнём не глядя,
как на фронте говорят...

Песня из фильма Щит и меч

Задача о размене денег возникла задолго до появления самих денег – ещё при натуральном обмене. Действительно, роль денег тогда исполняли различные ценные предметы: ракушки, зерно, шкуры, соль, плоды, животные. Пережитки далёкого «бартерного» прошлого сохранились до наших дней. Именно так и сейчас обмениваются коллекционеры книг, значков, марок и прочей утвари.

Первыми настоящими, металлическими деньгами начали пользоваться лидийцы в 8 веке до нашей эры. Ну, а дальше пошло-поехало – и теперь без денег уже ничего и не идёт, и не едет.

Но для нас важнее комбинаторная сторона этого меркантильного вопроса, а именно: как правильно разменять одни деньги на другие и при этом не остаться внакладе.

С разменом денег тесно связана другая задача – помоложе, но тоже имеющая многовековую историю – о *разбиении чисел*. Если бы в какой-либо денежной системе были монеты всех достоинств, начиная с одного «цента», то достаточно было бы решить только эту задачу. Но, как мы знаем, человечество проявило здесь немалую изобретательность, поэтому размен денег даётся значительно труднее.

Проект *Разменный пункт* (Разбиения)

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

$$n = n_1 + n_2 + \ldots + n_k? \qquad (1)$$

Последовательность 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 идет обо *всех* вариантах разбиения, то их число обозначают *P(n).* Мы легко найдём, 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 и дальше, но уже сейчас явно прослеживается её *рекурсивная* сущность. Однако мы начнём с *итерационного* алгоритма, описанного Витольдом Липским в книге **[ЛВ88,** Алгоритм 1.22**]**. Нам нужно только перевести его на *Си-шарп*.

Заданное число мы будем, как обычно, получать из компонента **udNum** и обозначим его буквой *n*. Все слагаемые мы поместим в *массив* **adds**, их общее число в текущем разбиении мы сохраним в *поле* **nAdds**, а число найденных вариантов – в поле **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.

```
//слагаемые:
int[] adds;
//число слагаемых:
int nAdds = 0;
//число повторов каждого слагаемого:
int[] r;
//число разбиений:
int nVar = 0;
```

Нажав на кнопку **Все разбиения (Ит)!**, мы по заданному числу *n* создаём оба массива и переходим в *метод* **part**:

```
//РАЗБИВАЕМ ЧИСЛО ИТЕРАЦИОННО
private void btnGenerate_Click(object sender, EventArgs e)
{
    int n = (int)udNum.Value;
    adds = new int[n + 1];
    r    = new int[n + 1];
    string s = "Все разбиения числа " + n.ToString() + ": ";
    lstVar.Items.Add(s);
    part(n);
    s = "Всего " + nVar.ToString();
    lstVar.Items.Add(s);
```

```
    lstVar.Items.Add("");
    lstVar.TopIndex = lstVar.Items.Count - 22;
}

//Генерируем все разбиения числа n
//и выводим в список
void part(int n)
{
    nVar = 0;
    //первое разбиение равно числу n:
    adds[1] = n;
    r[1] = 1;
    nAdds = 1;
    print(nAdds);
    //находим следующие разбиения:
    while (adds[1] > 1)
    {
        int sum = 0;
        if (adds[nAdds] == 1)
        {
            sum += r[nAdds];
            --nAdds;
        }
        sum += adds[nAdds];
        --r[nAdds];
        int l = adds[nAdds] - 1;
        if (r[nAdds] > 0) ++nAdds;
        adds[nAdds] = l;
        r[nAdds] = sum / l;
        l = sum % l;
        if (l != 0)
        {
            ++nAdds;
            adds[nAdds] = l;
            r[nAdds] = 1;
        }
        print(nAdds);
    }
}
```

Всякий раз, когда метод *part* сгенерирует новое разбиение, мы его печатаем на экране:

```
//ПЕЧАТАЕМ РАЗБИЕНИЕ
void print(int d)
{
    ++nVar;
    string s = "";
    for (int i = 1; i <= d; ++i)
      for (int j = 1; j <= r[i]; ++j)
```

```
        {
            s += (adds[i].ToString() + " ");
        }
    lstVar.Items.Add(s);
    //прокручиваем список вниз:
    lstVar.TopIndex = lstVar.Items.Count - 22;
    this.lstVar.Invalidate();
    Application.DoEvents();
}
```

И вот уже без особого напряжения мы получаем полный список разбиений числа *12* (Рис. 9.1).

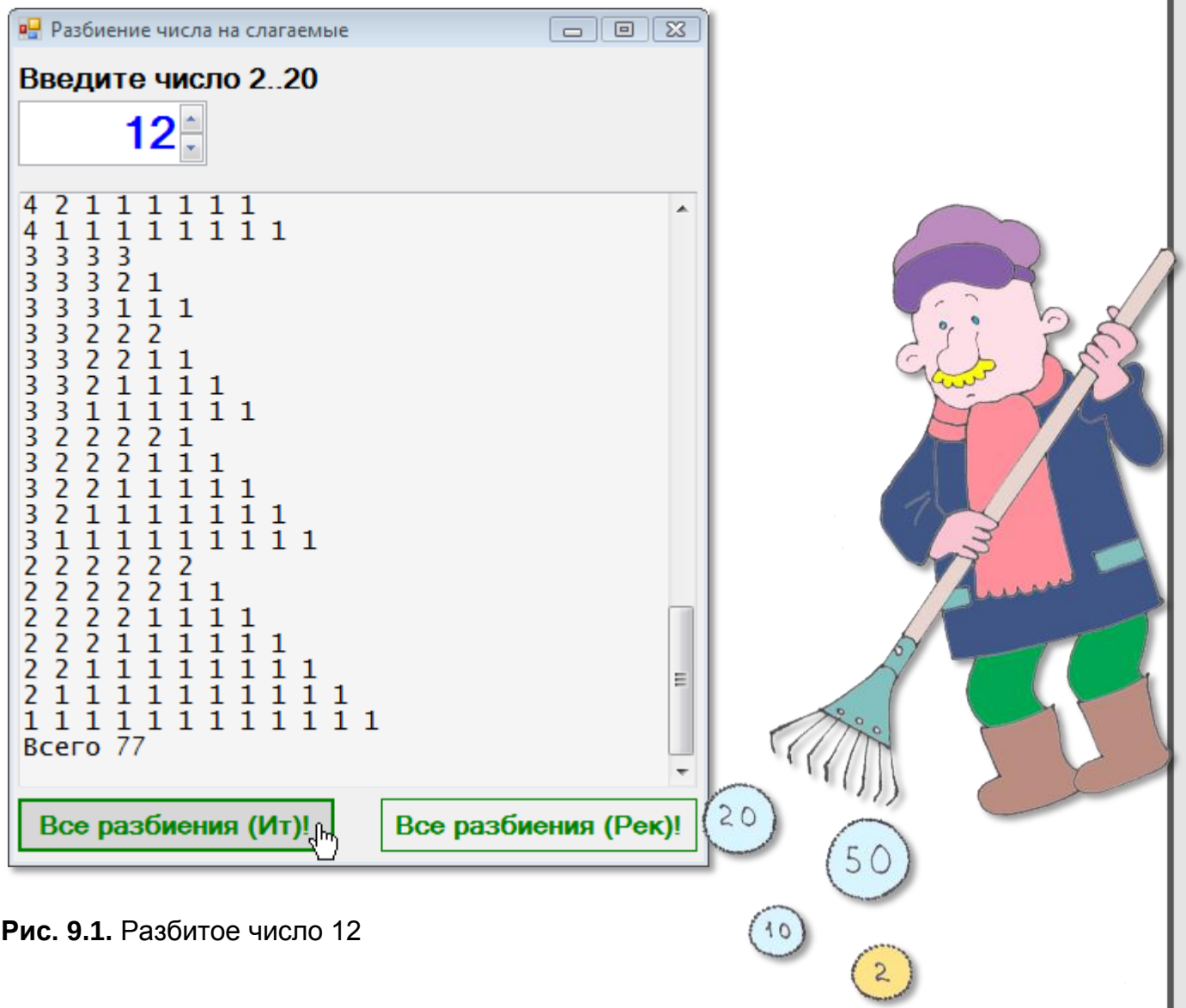


Рис. 9.1. Разбитое число 12

Попробуем подойти к решению задачи с другого конца - с *рекурсивного*. Здесь нам на помощь спешит другой алгоритм **[KS98,** Algorithm 3.1**]**, который мы слегка заточим под наши нужды.

Чтобы не нагружать излишне нашу единственную кнопку, подселим к ней соседку. Она вызывает другой метод и обходится без массива *r*:

```
//РАЗБИВАЕМ ЧИСЛО РЕКУРСИВНО
private void btnGenRec_Click(object sender, EventArgs e)
{
    int n = (int)udNum.Value;
    adds = new int[n + 1];
    string s = "Все разбиения числа " + n.ToString() + ": ";
    lstVar.Items.Add(s);
    nVar = 0;
    RecPartition(n,n,0);
    s = "Всего " + nVar.ToString();
    lstVar.Items.Add(s);
    lstVar.Items.Add("");
    lstVar.TopIndex = lstVar.Items.Count - 22;
}
```

А вот и рекурсивная разбивалка чисел:

```
void RecPartition(int n, int B, int N)
{
    if (n == 0)
    {
        nAdds = N;
        print(adds);
    }
    else
    {
        for (int i = 1; i <= Math.Min(B, n); ++i)
        {
            adds[N + 1] = i;
            RecPartition(n - i, i, N + 1);
        }
    }
}
```

Заметьте, насколько сократился алгоритм! Добавляем метод печати очередного разбиения:

```
void print(int[] a)
{
    ++nVar;
    string s = "";

    for (int i = 1; i <= nAdds; ++i)
    {
        int j = a[i];
        s += j.ToString();
```

```
            if (i != nAdds) s += "+";
        }
        lstVar.Items.Add(s);
        //прокручиваем список вниз:
        lstVar.TopIndex = lstVar.Items.Count - 22;
        this.lstVar.Invalidate();
        Application.DoEvents();
    }
```

Нажимаем кнопку **Все разбиения (Рек)!** – и *вуаля*, как говорят французы (Рис. 9.2).

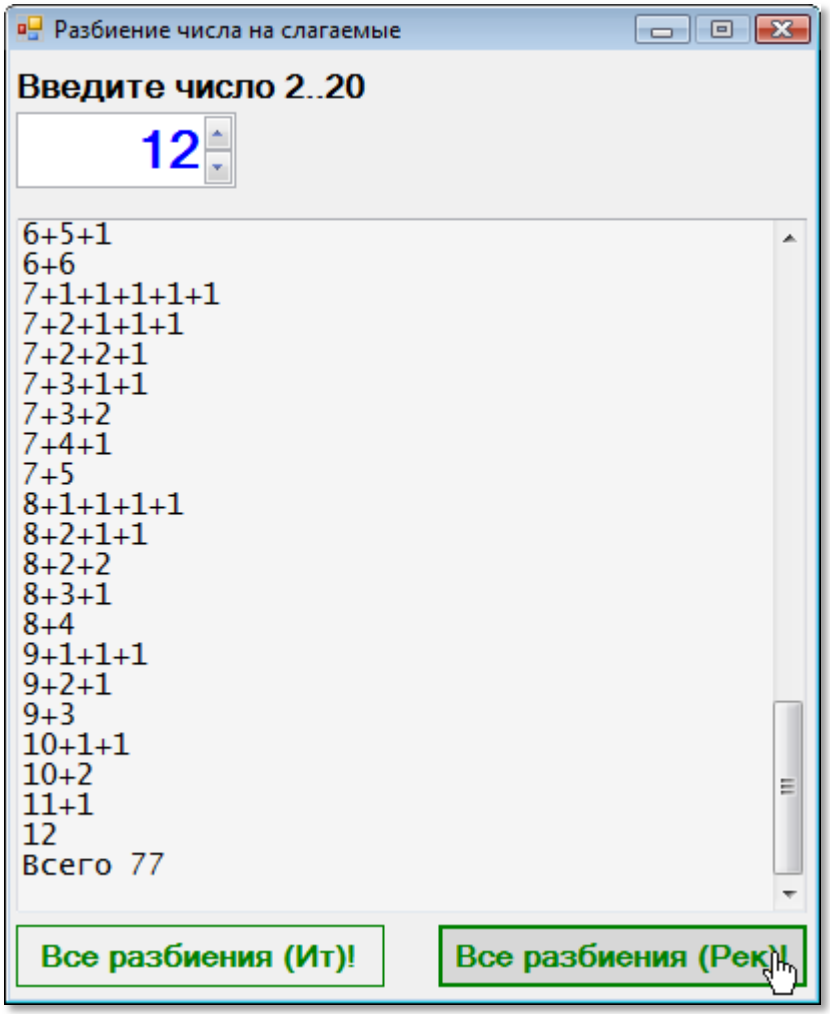


Рис. 9.2. Рекурсивная разбивалка в действии!

Более того, рекурсивный алгоритм получился даже более *универсальным*, чем итерационный. Например, мы можем, иначе подготовив его вызов, сгенерировать не все разбиения, а только те, в которых *наибольшее* слагаемое не превышает заданного числа *k*.

Дополним интерфейс *кнопкой* **btnK** *Все разбиения!* и *компонентом NumericUpDown*. В *методе* **btnK_Click** обратите внимание на выделенные строчки:

```
//ГЕНЕРИРУЕМ ВСЕ РАЗБИЕНИЯ,
//В КОТОРЫХ НАИБОЛЬШЕЕ СЛАГАЕМОЕ РАВНО k
private void btnK_Click(object sender, EventArgs e)
{
    int n = (int)udNum.Value;
    adds = new int[n + 1];
    int k =  (int)udK.Value;
    if (k > n)
    {
      udK.Value=n;
      k = n;
    }
    string s = "Все разбиения числа " + n.ToString();
    lstVar.Items.Add(s);
    s= "k = " + k + ":";
    lstVar.Items.Add(s);
    nVar = 0;
    adds[1] = k;
    RecPartition(n-k, k, 1);
    s = "Всего " + nVar.ToString();
    lstVar.Items.Add(s);
    lstVar.Items.Add("");
    lstVar.TopIndex = lstVar.Items.Count - 22;
}
```

Итак, перед вызовом нашего рекурсивного метода мы присваиваем первому слагаемому значение *k*, которое и остаётся неизменным во всех разбиениях, а все остальные слагаемые дополняют первое до числа *n*, но при этом не превышают *k* (Рис. 9.3).

В нашем разменном примере эта ситуация может возникнуть при отсутствии «крупной» мелочи.

Диаграммы Феррерса

Разбиения чисел – для наглядности – принято изображать *диаграммой Феррерса* (*Ferrers Diagram* или *Ferrers-Young Diagram*). Для разбиения (1) она состоят из *k* строк, каждая из которых соответствует слагаемому в разбиении и состоит из последовательности точек, число которых равно значению слагаемого.

Например, для разбиения *8 = 4 + 3 + 1* можно построить такую диаграмму (Рис. 9.4).

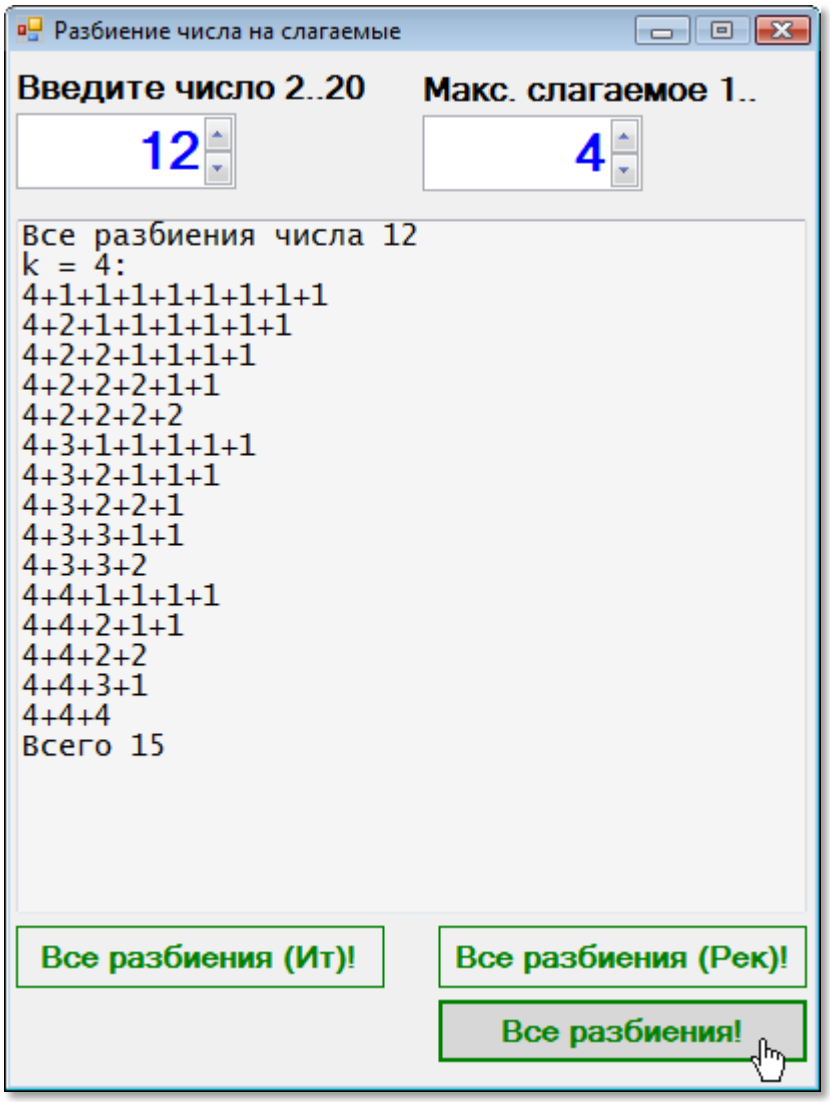


Рис. 9.3. Разбиваем на меткие кусочки!

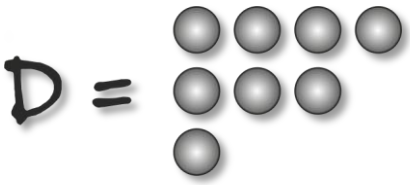


Рис. 9.4. Разбиение числа 8

Если перевернуть диаграмму Феррерса так, чтобы столбцы и строки поменялись местами (эта операция называется *транспозицией*), то мы получим *сопряжённое разбиение* заданного числа (Рис. 9.5).

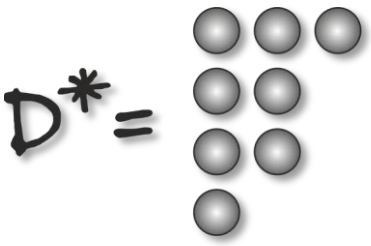


Рис. 9.5. Сопряжённое разбиение числа 8

Легко видеть, что исходному разбиению соответствует сопряжённое *8 = 3 + 2 +2 + 1*.

Вы пока периферийно думайте над свойствами этих разбиений, а мы в это время научимся *рисовать* диаграммы Феррерса. Я не думаю, что они настолько важны для нас, чтобы мы расщедрились на полноценную графику, так что мы будем просто ставить вместо кружочков буквы *O*.

Подпишите под вызовом метода *print* ещё и новый метод *printFerrers*, который для каждого разбиения составит диаграмму Феррерса:

```
void RecPartition(int n, int B, int N)
{
    if (n == 0)
    {
        nAdds = N;
        print(adds);
        printFerrers(adds);
    }
    . . .
```

В *методе* **printFerrers** мы для каждого слагаемого создаём новую строку из букв *O*, взятых в количестве, равном очередному слагаемому *a[i]:*

```
//ПЕЧАТАЕМ ДИАГРАММУ ФЕРРЕРСА
void printFerrers(int[] a)
{
    string s = "";

    for (int i = 1; i <= nAdds; ++i)
    {
        s = new String('O', a[i]);
        lstVar.Items.Add(s);
    }
    lstVar.Items.Add("");
    //прокручиваем список вниз:
    lstVar.TopIndex = lstVar.Items.Count - 22;
    this.lstVar.Invalidate();
    Application.DoEvents();
}
```

И вот что у нас получилось (Рис. 9.6).

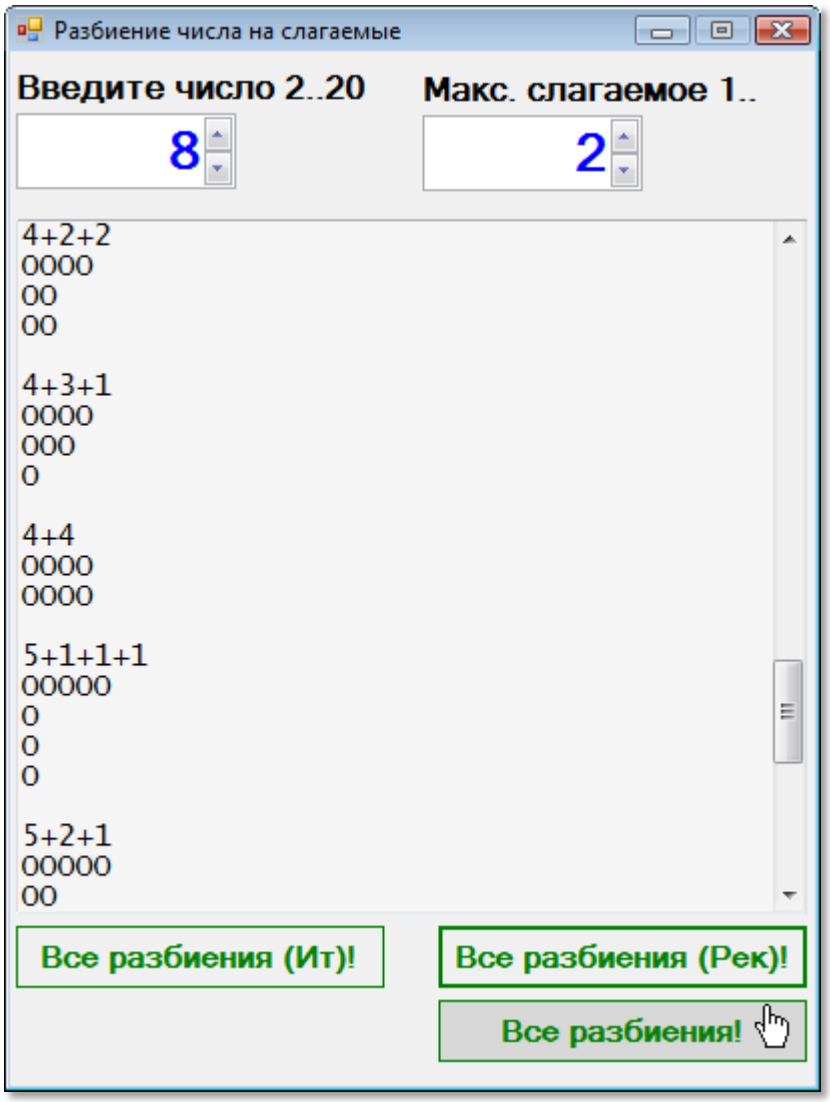


Рис. 9.6. Наше разбиение!

Теперь давайте *транспонируем* диаграмму, а затем ещё раз подумаем, зачем нам это нужно.

Поскольку задача чисто геометрическая, то мы справляемся с ней одним махом:

```
int nAdds2;
int[] ConjPartition(int[] a)
{
    int[] b = new int[a.Length];
    int n = a.Length;
    nAdds2 = a[1];

    for (int i = 1; i < n; ++i)
        b[i] = 1;

    for (int j = 2; j <= nAdds; ++j)
      for (int i = 1; i <= a[j]; ++i)
          ++b[i];
```

```
    return b;
}
```

Обратите внимание: нам пришлось ввести новое *поле nAdds2* для хранения числа слагаемых в *массиве b*, что повлекло за собой правки в методах *printFerrers* и *RecPartition*:

```
//ПЕЧАТАЕМ ДИАГРАММУ ФЕРРЕРСА
void printFerrers(int[] a, int nPart)
{
  string s = "";
  for (int i = 1; i <= nPart; ++i)
  . . .


void RecPartition(int n, int B, int N)
{
  if (n == 0)
  {
     nAdds = N;
     print(adds);
     printFerrers(adds, nAdds);
     printFerrers(ConjPartition(adds), nAdds2);
    . . .
```

Зато мы можем печатать не только «прямые» диаграммы Феррерса, но и сопряжённые (Рис. 9.7).

И вот пришла пора вернуться к нашему животрепещущему вопросу об этих диаграммах. И по ним, и по методу *ConjPartition* мы видим, что количество разбиений числа *n* на *k* частей в точности равно числу его разбиений, в которых наибольшая часть имеет значение *k*:

```
nAdds2 = a[1];
```

Число разбиений числа *n* на *k* частей обозначают *P(n,k).*

Несколько полезных равенств:

$P(0,0) = 1$

$P(n,0) = 0 \qquad n>=1$

$P(n,1) = P(n,n) = 1 \quad n>=1$

Очевидно, что сумма *всех* возможных разбиений числа *n* на *k* частей равна *P(n),* то есть

$P(n) = P(n,1) + P(n,2) + \ldots + P(n,n)$

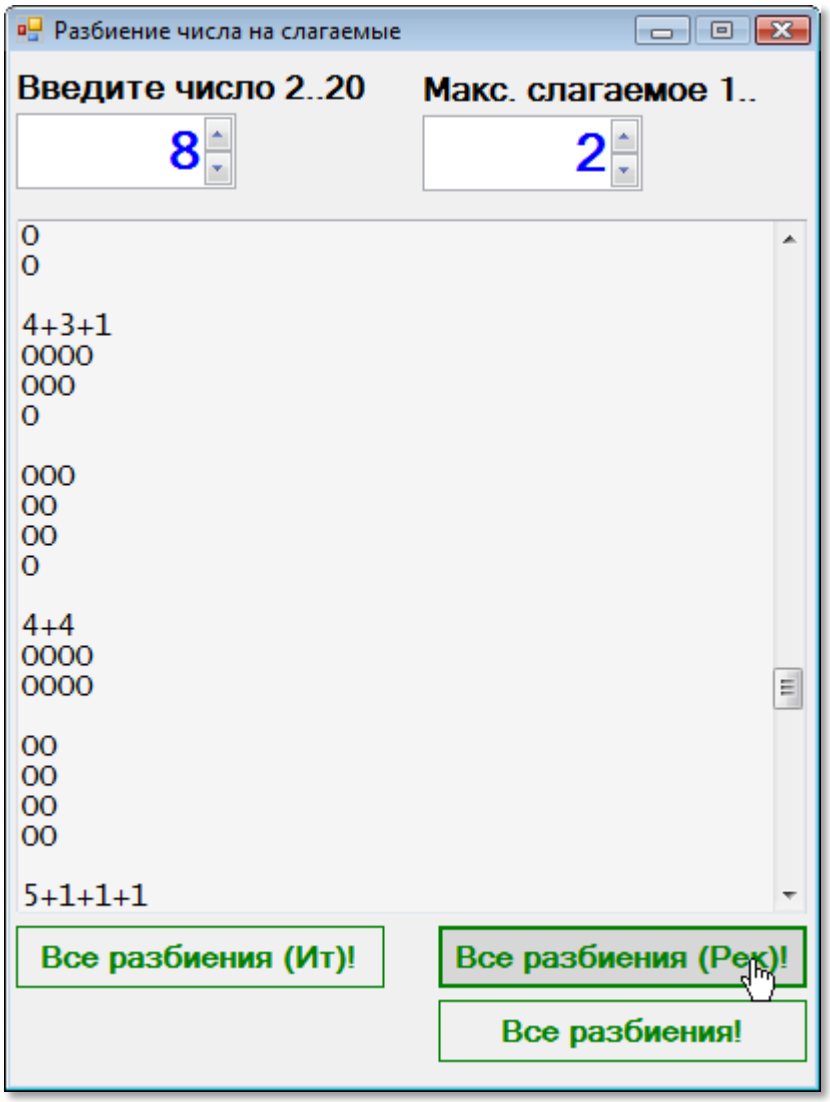


Рис. 9.7. Оттранспонировались успешно!

Ставим четвёртую - *кнопку* **btnK2** *Все разбиения на k!*, - которая действует подобно кнопке *Все разбиения!,* но вызывает рекурсивный метод *RecPartition2*:

```
//РАЗБИВАЕМ ЧИСЛО НА k ЧАСТЕЙ
private void btnK2_Click(object sender, EventArgs e)
{
    int n = (int)udNum.Value;
    adds = new int[n + 1];
    int k = (int)udK.Value;
    if (k > n)
    {
        udK.Value = n;
        k = n;
    }
    string s = "Все разбиения числа " + n.ToString();
    lstVar.Items.Add(s);
    s = "на заданное число частей " + k + ":";
    lstVar.Items.Add(s);
```

```
    nVar = 0;
    adds[1] = k;
    RecPartition2(n - k, k, 1);
    s = "Всего " + nVar.ToString();
    lstVar.Items.Add(s);
    lstVar.Items.Add("");
    lstVar.TopIndex = lstVar.Items.Count - 22;
}
```

Он, в свою очередь, почти близнец метода *RecPartition*, но печатает сопряжённое разбиение (Рис. 9.8), которое нам предоставляет метод *ConjPartition*:

```
void RecPartition2(int n, int B, int N)
{
    if (n == 0)
    {
        nAdds = N;
        print(ConjPartition(adds), nAdds2);
    }
    else
    {
        for (int i = 1; i <= Math.Min(B, n); ++i)
        {
            adds[N + 1] = i;
            RecPartition2(n - i, i, N + 1);
        }
    }
}
```

И тут нам опять пришлось поправить уже готовый *метод* **print**, чтобы он умел работать с *разными* массивами, а не только с *adds*:

```
void print(int[] a, int nPart)
{
    . . .
    for (int i = 1; i <= nPart; ++i)
    {
        int j = a[i];
        s += j.ToString();
        if (i != nPart) s += "+";
    }
    . . .
}
```

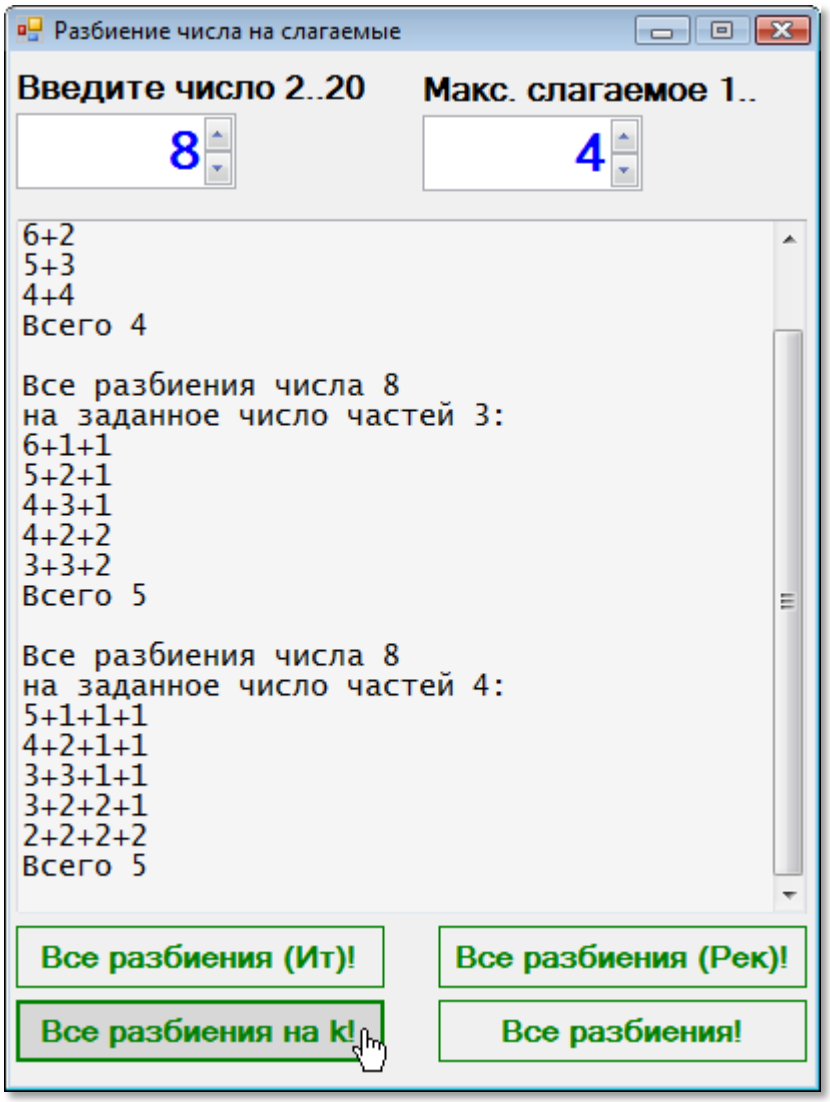


Рис. 9.8. Красиво поделились!

В книге Красиковых **[КК07],** на страницах 197-200 вы найдёте итерационную версию алгоритма, который генерирует все разбиения заданного числа в антилексикографическом порядке, то есть как и у нас. Он реализован на языке *C++,* так что можете сравнить его с нашим вариантом.
Далее, на страницах 200-202 рассматривается алгоритм разбиения числа на заданное количество частей.

В книге **[KS98],** на страницах 67-68 представлен тот же самый рекурсивный алгоритм для генерации всех разбиений числа, что и в нашей программе. А дальше рассматриваются диаграммы Феррерса и алгоритм разбиения числа на заданное количество частей.

Большие числа склонны давать огромное количество разбиений, а ведь иногда нам нужно просто узнать, сколько имеется тех или иных разбие-

ний, не генерируя их одно за другим. В той же книге **[KS98]** на этот случай уже готов *Алгоритм 3.5*, который мы любезно позаимствуем для собственных нужд.

В метод новой *кнопки* **btnEnumPart** *Посчитать P(n,k)!* мы записываем перевод алгоритма на *Си-шарп*:

```
private void btnEnumPart_Click(object sender, EventArgs e)
{
    int n = (int)udNum.Value;
    adds = new int[n + 1];
    int k = (int)udK.Value;
    int[,] p =  new int[n + 1, k + 1];
    enumPart(p, n, k);
    for (int i = 1; i <= n; ++i)
    {
        string s = "";
        int pn=0;
        for (int j = 1; j <= k; ++j)
        {
            pn += p[i, j];
            s += ("P("+i + "," + j + ")=" + p[i, j].ToString()+
                  " ");
        }
        if (n==k)
            lstVar.Items.Add("P(" + i +") = " + pn);
        lstVar.Items.Add(s);
    }
    lstVar.Items.Add("");
    lstVar.TopIndex = lstVar.Items.Count - 22;
}

void enumPart(int[,] p, int n, int k)
{
    p[0, 0] = 1;
    for (int i = 1; i <= n; ++i)
        p[i, 0] = 0;

    for (int i = 1; i <= n; ++i)
      for (int j = 1; j <= Math.Min(i,k); ++j)
      {
          if (i < 2 * j)
              p[i, j] = p[i - 1, j - 1];
          else
              p[i, j] = p[i - 1, j - 1] + p[i-j, j];
      }
}
```

Теперь нам не составит никакого труда удовлетворить своё любопытство без лишних подробностей (Рис. 9.9).

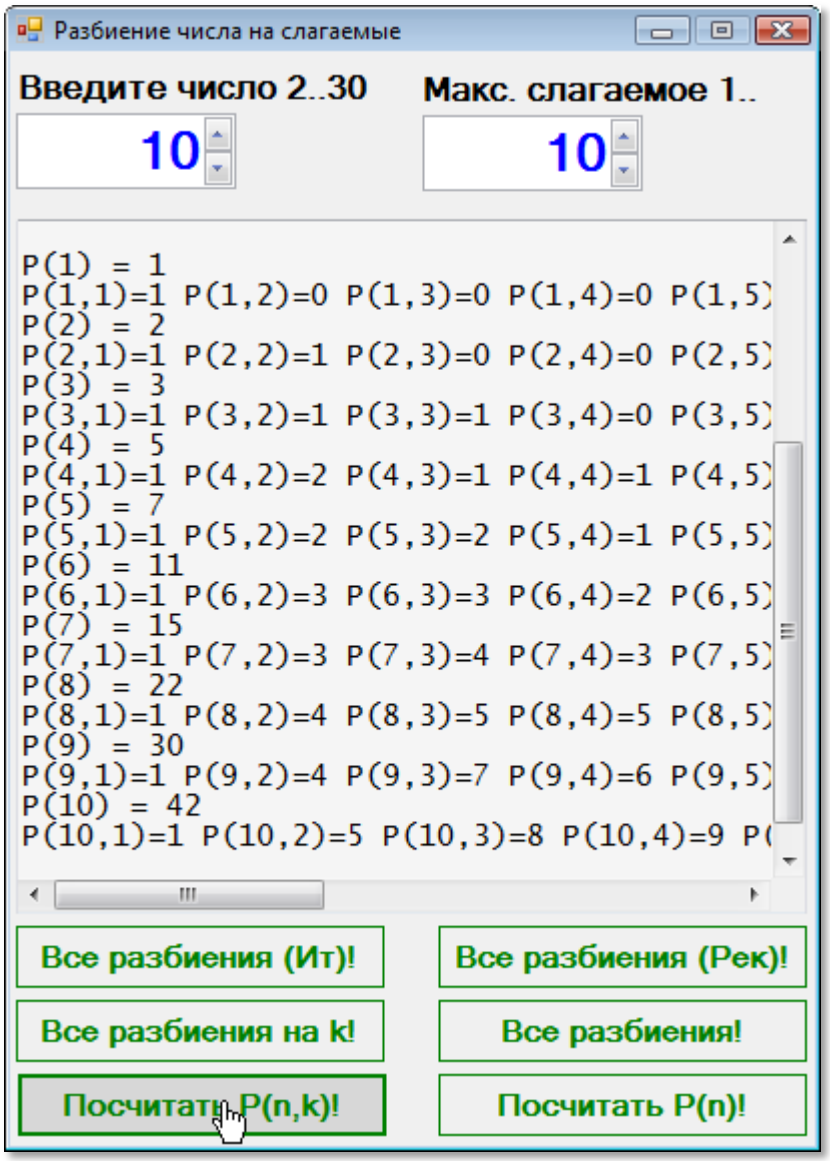


Рис. 9.9. Статистика знает всё!

Давайте ещё более сконцентрируем наш алгоритм, чтобы он выдавал только общее количество разбиений для заданного числа *n*, то есть *P(n)*.

Добавляем *кнопку* **btnEnumPart2** *Посчитать P(n)!* и переводим *Алгоритм 3.6:*

```
private void btnEnumPart2_Click(object sender, EventArgs e)
{
    int n = (int)udNum.Value;
    int[] p = new int[n + 1];
    enumPart2(p, n);
    for (int i = 1; i <= n; ++i)
    {
        lstVar.Items.Add("P(" + i + ") = " + p[i]);
    }
    lstVar.Items.Add("");
    lstVar.TopIndex = lstVar.Items.Count - 22;
```

```
}
void enumPart2(int[] p, int n)
{
    int sign = 0;
    int sum = 0;
    int w = 0;
    int w2 = 0;
    int j = 0;
    p[0] = 1;
    p[1] = 1;
    for (int i = 2; i <= n; ++i)
    {
        sign = 1;
        sum = 0;
        w = 1;
        j = 1;
        w2 = w + j;
        while (w <= i)
        {
           if (sign==1)
               sum += p[i-w];
           else
               sum -= p[i-w];

           if (w2 <= i)
           {
               if (sign == 1)
                   sum += p[i - w2];
               else
                   sum -= p[i - w2];
           }
            w += (3*j+1);
            ++j;
            w2 = w + j;
            sign = 0-sign;
        } //while
        p[i] = sum;
    } //for
}
```

Тут нужно отметить, что авторы книги **[KS98]** запутались в алгоритме и напечатали странную таблицу 3.1, в которой посчитаны не все значения *P(m,n)*, поэтому *P(m)*, начиная с *m=11*, неверные.

В главе *Даёшь сотню!* нам понадобятся *композиции* числа 9. Если в разбиениях порядок слагаемых значения не имеет (обычно они записываются в стандартной форме, но это непринципиально), то в композициях важен и порядок слагаемых в записи. Это значит, что для каждого разбиения мы

должны составить и все *перестановки* слагаемых. Если *k=1*, то число разбиений равно числу композиций и равно *единице*, то есть и разбиение, и композиция равны самому числу *n*.

Если *k=2*, то возможны варианты разбиений. Если *n*, к примеру, равняется пяти, то мы имеем два *разбиения*:

4 1
3 2

А *композиций* вдвое больше, поскольку числа в разбиениях можно переставить:

4 1 и 1 4
3 2 и 2 3

Для *k=3* число композиций утраивается по сравнению с разбиениями:

3 1 1
2 2 1

3 1 1 и 1 3 1 и 1 1 3
2 2 1 и 2 1 2 и 1 2 2

Дальше можно не продолжать…

Таким образом, мы можем дополнить алгоритм для генерирования разбиений кодом для перестановки слагаемых, но лучше использовать более простой алгоритм, который оба действия выполняет одновременно.

В книге *Handbook of Applied Algorithms*, на странице 12 приведён красивый короткий алгоритм на этот счёт. Одна беда – он не работает.

Обратимся к книге *Algorithms for Programmers: Ideas and Source Code*, где в главе 7 мы найдём «классное» решение нашей проблемы. Код несколько тяжеловат, но зато универсален: позволяет генерировать композиции не все разом, а по числу групп, причём в *прямом* (лексикографическом) и *обратном* (антилексикографическом) порядке.

Переводим код с языка *C++* на *C#* и добавляем новый класс в нашу библиотеку:

```
    public class Compositions
    {
        int N;
        int K;
        int[] p;
```

```
        int nk1; //= N-K+1

        //КОНСТРУКТОР
        public Compositions(int n, int k)
        {
            N = n;
            K = k;
            nk1 = N - K + 1;
            //nk1 >= 1 !!!
            p = new int[K+1];
            first();
        }
        public void first()
        {
            p[0] = nk1;
            p[K] = 0;
            for (int i = 1; i < K; ++i)
                p[i] = 1;
        }
        public void last()
        {
            for (int i = 0; i < K; ++i)
                p[i] = 1;
            p[K-1] = nk1;
        }
        public int next()
        {
            int i = 0;
            //ищем первое слагаемое, большее 1:
            while (p[i] == 1) ++i;
            //текущая композиция - последняя:
            if (i == K) return K;

            int v = p[i];
            p[i] = 1;
            p[0] = v-1;
            ++i;
            ++p[i];
            return i;
        }
        public int prev()
        {
            int v = p[0];
            //текущая композиция - первая:
            if (v == nk1) return K;
            p[0] = 1;

            int i = 1;
            //ищем первое слагаемое, большее 1:
            while (p[i] == 1) ++i;
            --p[i];
```

```
            p[i-1] = v+1;

            return i;
        }
        public int[] data()
        {
            return p;
        }
    }
```

Устанавливаем *кнопку* **btnComposition** *Композиции!* и запускаем генератор на полную мощность:

```
private void btnComposition_Click(object sender, EventArgs e)
{
    int n = (int)udNum.Value;
    int k = (int)udK.Value;
    if (n < k) MessageBox.Show("Число меньше слагаемых!", "Разбиения");
    //выводим в антилексикографическом порядке:
    bool rq = false;
    //выводим в лексикографическом порядке:
    //bool rq = true;
    Compositions P = new Compositions(n, k);
    if (rq) P.last();
    else P.first();
    nVar = 0;
    //только считаем:
    //if (rq)
    //    do
    //    {
    //       ++nVar;
    //    } while (P.prev() != k);
    //else
    //    do
    //    {
    //       ++nVar;
    //    } while (P.next() != k);
    //генерируем композиции:
    int[] p = P.data();
    int q = k - 1;
    do
    {
       ++nVar;
        //печатаем композицию:
        string s = "";
        for (int i = 0; i < k; ++i)
        {
            s += (p[i] + " ");
        }
        lstVar.Items.Add(s);
```

```
        if (rq) q = P.prev();
        else q = P.next();
    } while (q != k);
    lstVar.Items.Add("Всего композиций " + nVar);
    lstVar.Items.Add("");
    lstVar.TopIndex = lstVar.Items.Count - 22;
}
```

Если вы раскомментируете строки после `//только считаем:`, то можете закомментировать генерирование композиций и получить только число композиций. Порядком вывода слагаемых управляет логическая переменная *rq*.

Ну и для уравновешивания кнопок на форме поставим ещё одну - **btnComp100** *Композиции 100!*. В её метод клика впишем облегчённую версию алгоритма, который используется в последней главе:

```
private void btnComp100_Click(object sender, EventArgs e)
{
    int n = (int)udNum.Value;
    int k = (int)udK.Value;
    if (n < k) MessageBox.Show("Число меньше слагаемых!", "Разбиения");
    nVar = 0;
    //массив с числами композиции:
    int[] c= new int[n + 1];
    //формируем начальную композицию:
    c[1] = n - k + 1;
    for (int j = 2; j <= k; ++j)
         c[j] = 1;
    //генерируем остальные композиции:
    writeC(c, k);
    compC(c, 1, k);
    lstVar.Items.Add("Всего композиций " + nVar);
    lstVar.Items.Add("");
    lstVar.TopIndex = lstVar.Items.Count - 22;
}
void compC(int[] a, int n, int nGr)
{
    //для первой композиции:
    if (nGr == 1) return;
    int[] L = (int[])a.Clone();
    while (L[n] > 1)
    {
        --L[n];
        ++L[n + 1];
        writeC(L, nGr);
        if (n + 1 < nGr)
        {
            compC(L, n + 1, nGr);
```

```
        }
    }
}
//ПЕЧАТАЕМ ОЧЕРЕДНУЮ КОМПОЗИЦИЮ
void writeC(int[] a, int n)
{
    ++nVar;
    string s = "";
    for (int i = 1; i <= n; ++i)
        s += (a[i] + " ");
    lstVar.Items.Add(s);
}
```

Сравнительные испытания обоих алгоритмов показывают их полное согласие с комбинаторной теорией (Рис. 9.10).

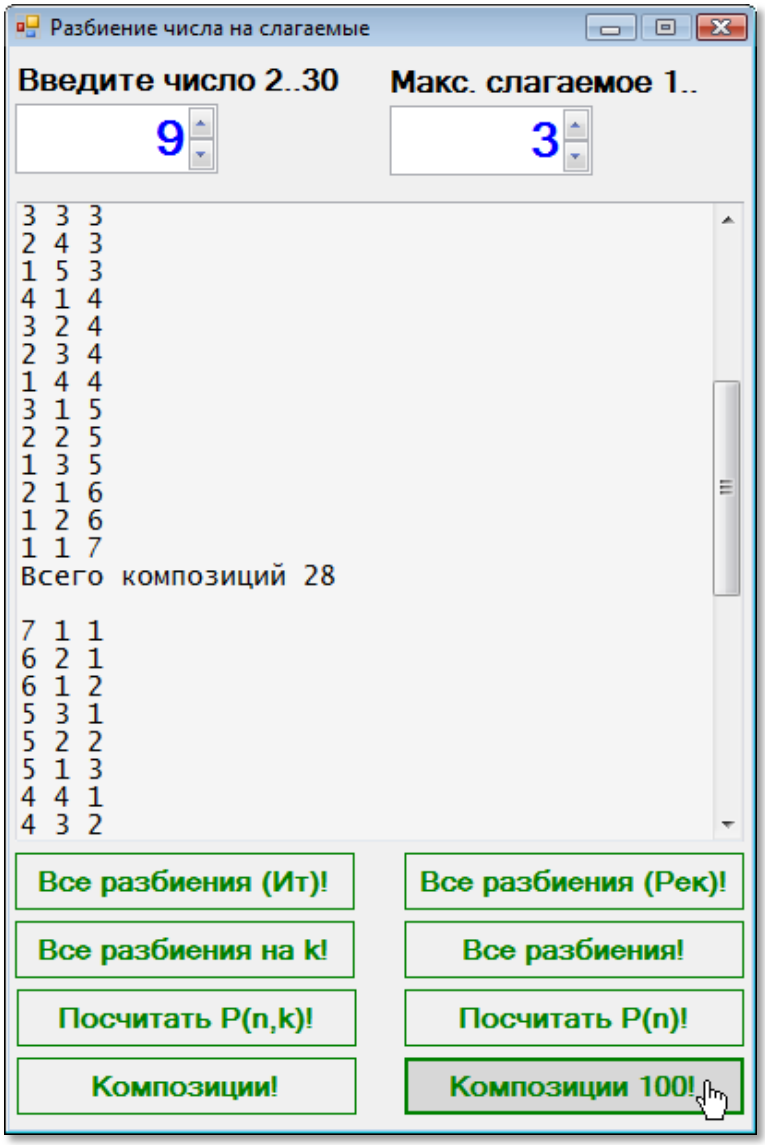
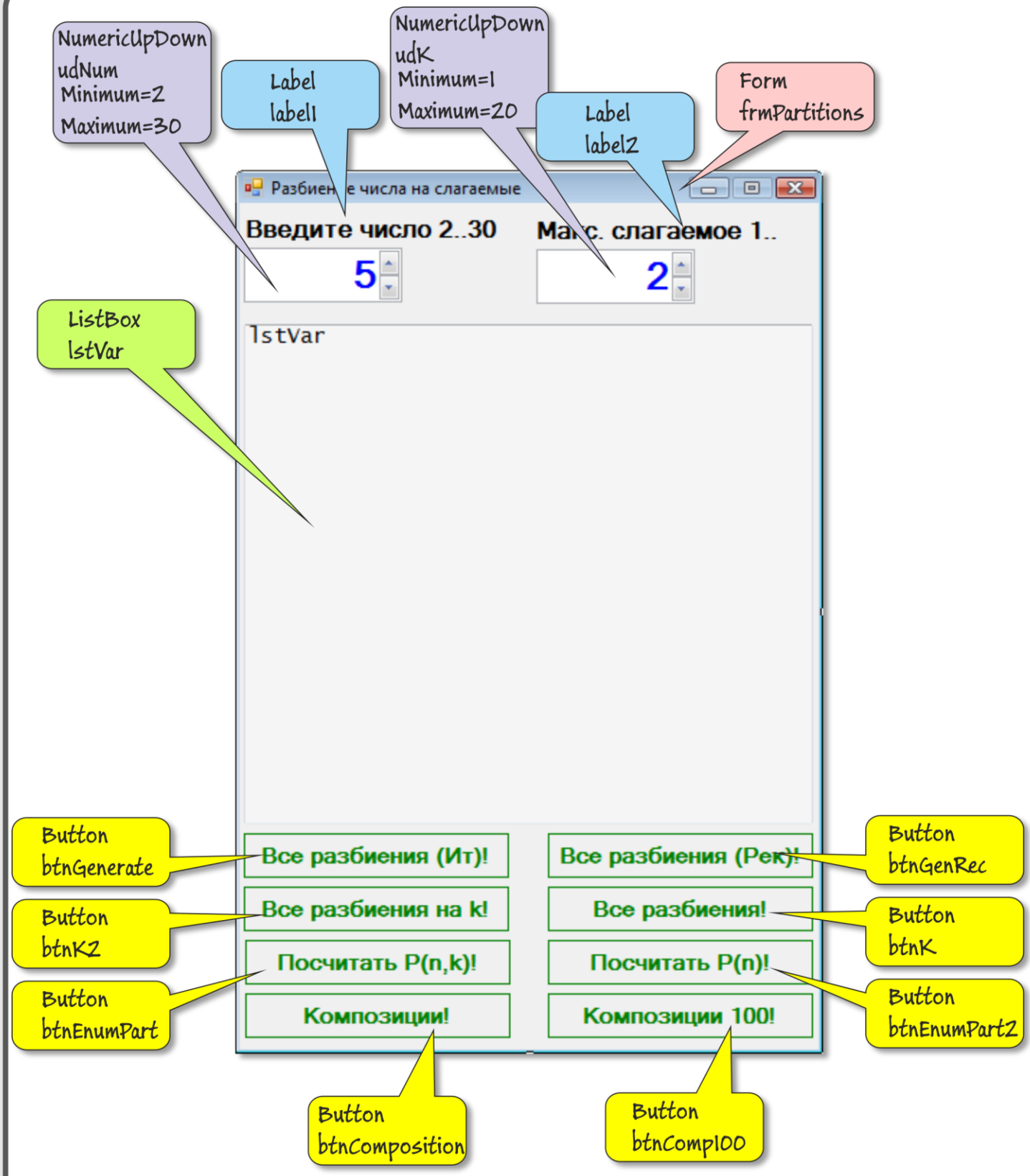


Рис. 9.10. Числовой композитор в действии!



Примерный вид интерфейса приложения

Исходный код программы находится в папке **Разбиения**.

1. Как мы знаем, наборы монет (или других объектов) могут быть разными, однако пользователи нашей программы лишены возможности задавать наборы, отличные от тех, что уже «зашиты» в коде программы. Добавьте компонент *TextBox*, чтобы пользователи могли передавать в приложение собственные наборы монет.
2. В книге А.М. и И.М. Ягломов *Неэлементарные задачи в элементарном изложении* есть ряд задач по размену денег и разбиению чисел.

Решите их помощью нашей программы и сравните затраты труда с логическими решениями авторов книги!

19. Сколькими способами можно разменять 20 копеек на монеты достоинством 5 копеек, 2 копейки и 1 копейка.

20. *Сколькими способами можно составить n копеек из монет достоинством 2 копейки и 1 копейка.*

21**. *Сколькими способами можно составить n копеек из монет:*

а) достоинством 1 копейка, 2 копейки и 3 копейки?
б) достоинством 1 копейка, 2 копейки и 5 копеек?

21**. *Сколькими способами можно составить рубль из монет достоинством в 1, 2, 5, 10, 20 и 50 копеек?*

22. Сколькими способами можно представить число n в виде суммы двух целых положительных слагаемых, если представления, отличающиеся лишь порядком слагаемых, считать одинаковыми?

Конечно, решение задач с неопределенной суммой в *n* копеек потребует и размышлений, а не только нажимания кнопки.

3. Двойные кружки

Решите такую задачу (Рис. 9.15). Впишите в кружки, расположенные на окружности, 9 чисел, сумма которых равна 100. В те кружки, что находятся снаружи, следует записывать те же числа, что и в соприкасающихся с ними кружках (то есть каждое число используется в ожерелье дважды).

Теперь составьте из любых пар чисел натуральный ряд последовательных чисел *максимальной* длины, начинающийся с единицы. Например, при таком заполнении бусин числами мы получим ряд, очень близкий к рекордному (Рис. 9.16).

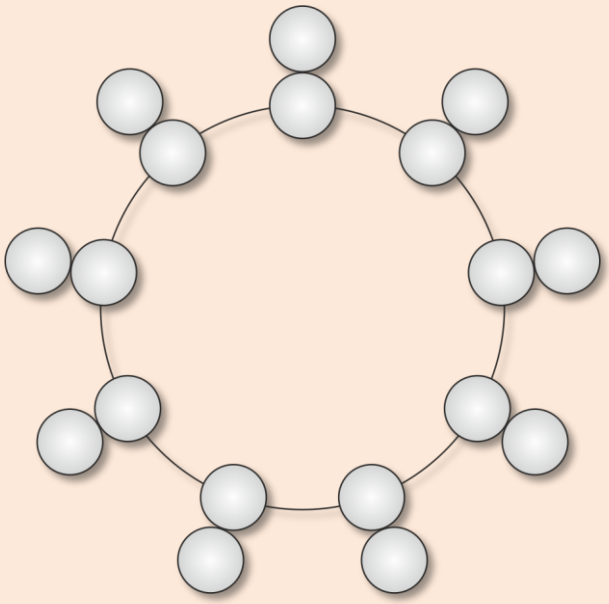


Рис. 9.15. Двойное ожерелье

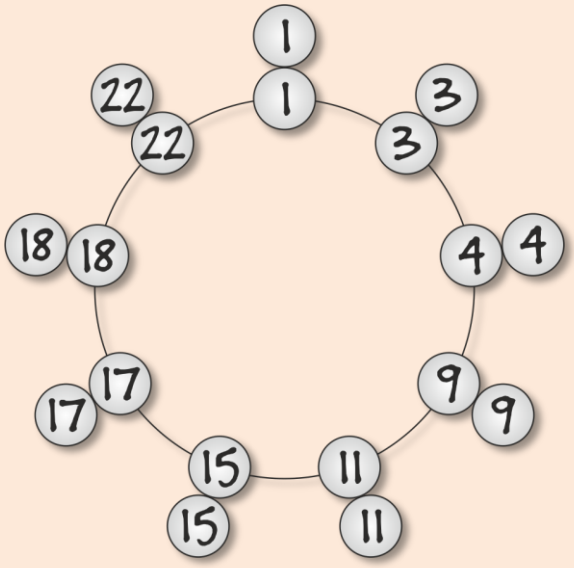


Рис. 9.16. Почти рекордный результат

Он состоит из 37 чисел: 1, 2 (1+1), 3, 4, 5 (4+1), 6 (3 + 3), 7 (4 +3), 8 (4 +4), 9, 10 (9 +1), …, 35 (18 + 17), 36 (18+18), 37 (22+15).

А самая длинная последовательность включает *40* чисел, причём задача имеет единственное решение (Рис. 9.17).

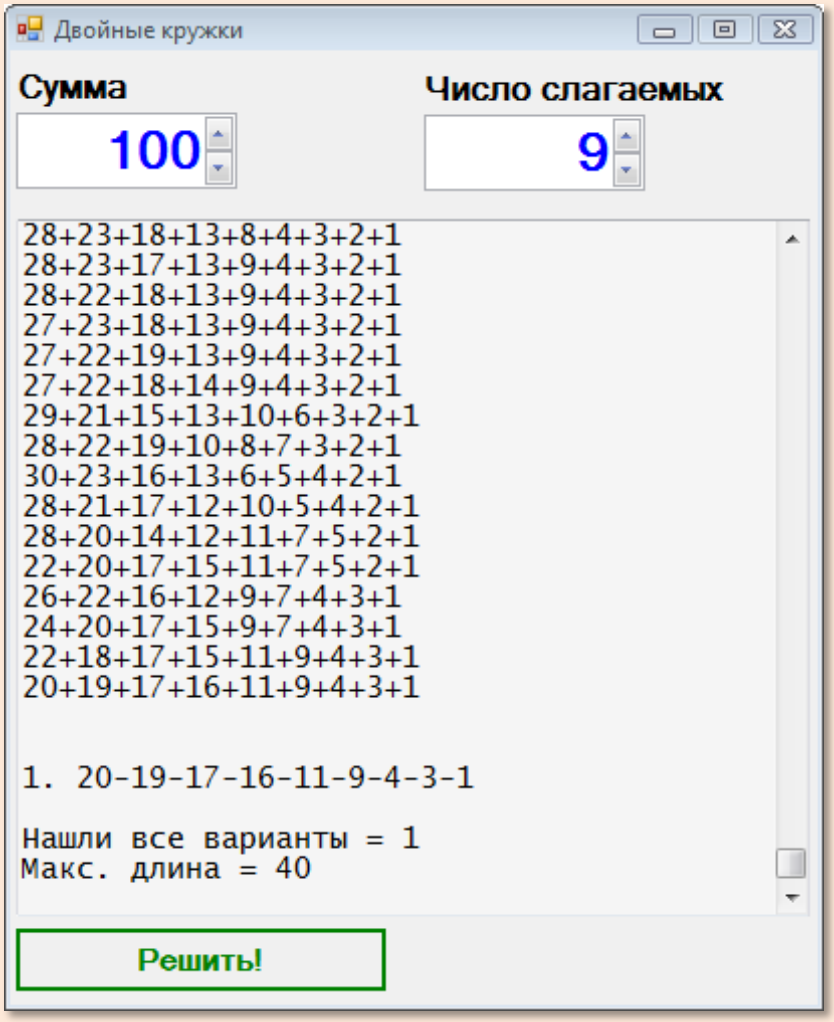


Рис. 9.17. А вот и рекорд!

Исходный код программы находится в папке **Двойные кружки**.

Ответы

Словесные магические квадраты

1 З	2 А	3 Д	4 А	5 Ч	6 А
2 А	Р	О	М	А	Т
3 Д	О	М	И	Н	О
4 А	М	И	Л	А	Н
5 Ч	А	Н	А	Х	И
6 А	Т	О	Н	И	Я

1 Б	2 О	3 Р	4 О	5 Н	6 А
2 О	Р	А	Т	О	Р
3 Р	А	Д	И	С	Т
4 О	Т	И	А	Т	Р
5 Н	О	С	Т	Р	О
6 А	Р	Т	Р	О	З

С	О	Р	О	К	А
О	С	Е	Л	О	К
Р	Е	Ф	Е	Р	И
О	Л	Е	Ф	И	Н
К	О	Р	И	Ц	А
А	К	И	Н	А	К

Литература

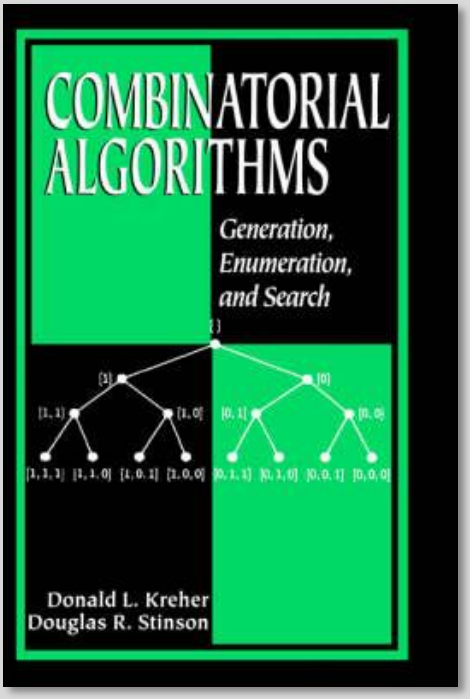
[PB13]

Рубанцев Валерий

Тотальный тренинг по Си-шарпу. Большой практикум по программированию на языке C#

RVGames, 2013. – 615 с.

[KS98]

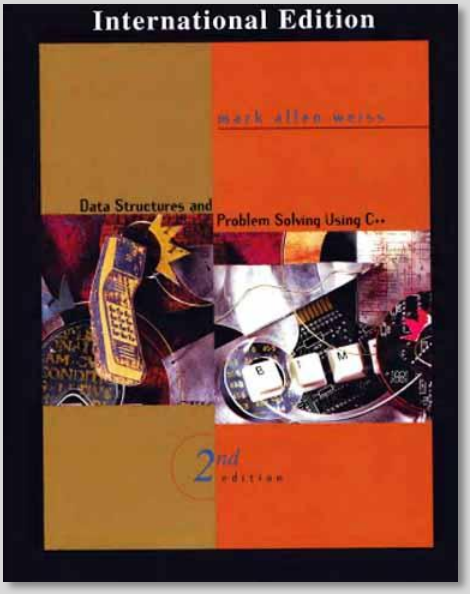


Kreher D.L., Stinson D.R.

Combinatorial Algorithms: Generation, Enumeration, and Search

CRC, 1998.- 344 с.
ISBN: 084933988X

[WM99]



Weiss M.A.

Data Structures and Problem Solving Using C++

Pearson Education, 1999. – 976 с. ISBN 0321205006

[ГМ72]

Гарднер Мартин

Математические досуги

М.:Мир, 1972. – 495 с.

[ГМ90]

Гарднер Мартин

Путешествие во времени

М.:Мир, 1990. – 341 с.

ISBN: 5-03-001166-8

[ГМ10]

Гарднер Мартин

1000 развивающих головоломок, математических загадок и ребусов для детей и взрослых

АСТ, Астрель, 2010. – 287 с.

ISBN 978-5-17-059779-6
ISBN 978-5-271-24093-5

[КД43]

Дональд Эрвин Кнут

Искусство программирования, том 4, выпуск 3. Генерация всех сочетаний и разбиений.

Вильямс, 2007. – 208 с.

ISBN: 978-5-8459-1132-2

[КД44]

Дональд Эрвин Кнут

Искусство программирования, том 4, выпуск 4. Генерация всех деревьев. История комбинаторной генерации

Вильямс, 2007. – 160 с.

ISBN: 978-5-8459-1158-2

[КК07]

Красиков И. В., Красикова И. Е.

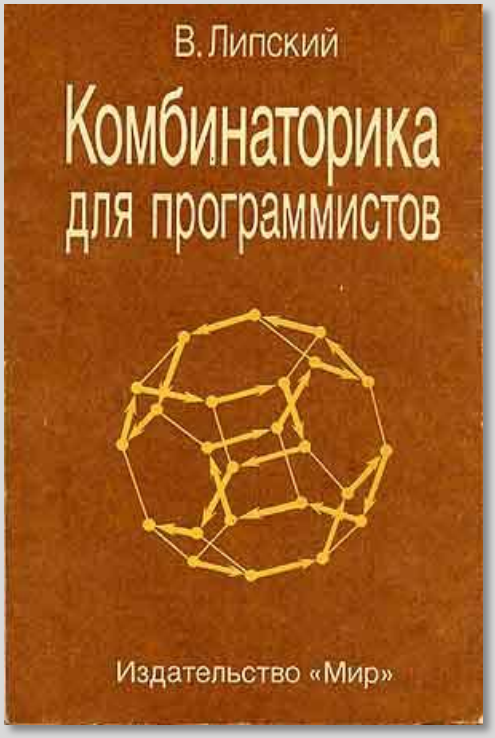
Алгоритмы

Эксмо, 2007.- 256 с.

ISBN: 978-5-699-21047-3

Серия: Просто как дважды два

[ЛВ88]

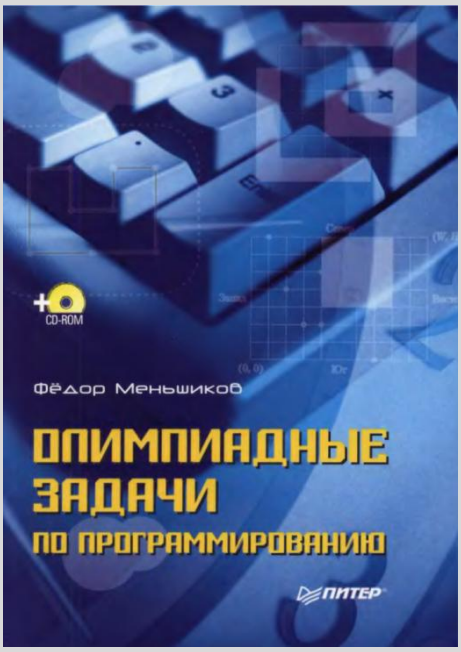


Липский В.

Комбинаторика для программистов

Москва: Мир, 1988. – 200 с.

[МФ06]



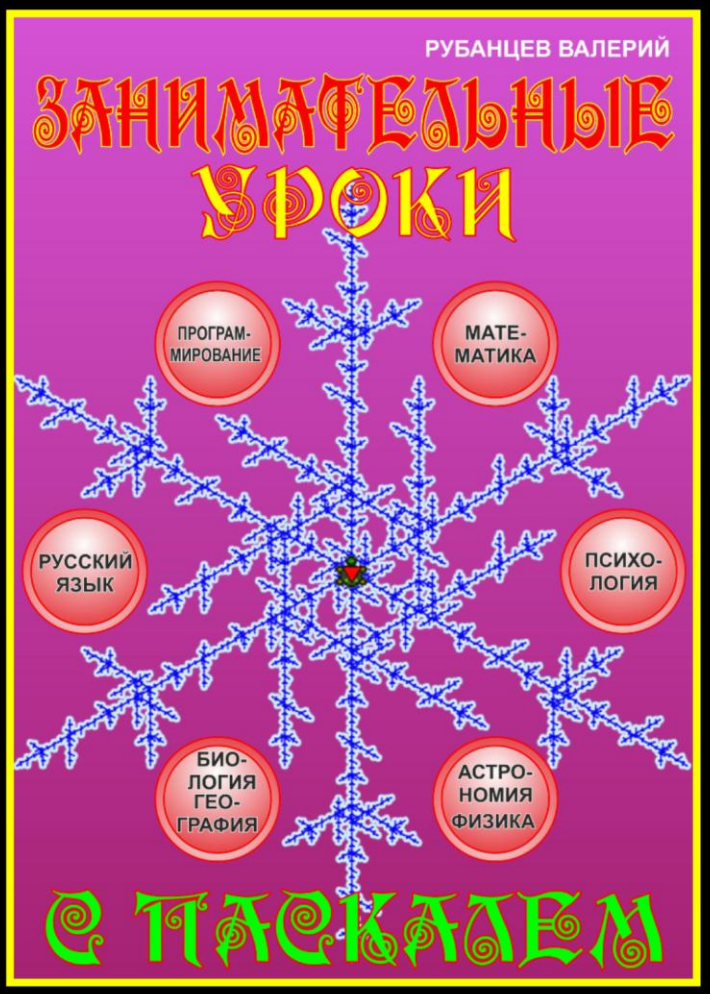
Меньшиков Фёдор

Олимпиадные задачи по программированию

Питер, СПб, 2006. – 315 с.

ISBN: 5-469-00765-0


ВАЛЕРИЙ РУБАНЦЕВ
ЗАНИМАТЕЛЬНЫЕ УРОКИ
С КОМПЬЮТЕРОМ
БИО-ЛОГИЯ
ПРОГРАМ-МИРОВАНИЕ
АСТРО-НОМИЯ
ГЕО-ГРАФИЯ
РУССКИЙ ЯЗЫК
МАТЕ-МАТИКА
ПСИХО-ЛОГИЯ
ФИЗИКА
43 урока
80 проектов на новейшем языке программирования Microsoft Small Basic
ДРЕВО ЗНАНИЙ
Тыблоки
Тьюрмиты
Тараканьи бега
Дефектный логопеф
Полярная Черепашка
Фрактальные снежинки
Осьминожьи гадания
Рекурсивные сказки
Кролики Фибоначчи
Галилеевы ядра
Дрим тим


РУБАНЦЕВ ВАЛЕРИЙ
ЗАНИМАТЕЛЬНЫЕ УРОКИ
ПРОГРАМ-МИРОВАНИЕ
МАТЕ-МАТИКА
РУССКИЙ ЯЗЫК
ПСИХО-ЛОГИЯ
БИО-ЛОГИЯ ГЕО-ГРАФИЯ
АСТРО-НОМИЯ ФИЗИКА
С ПАСКАЛЕМ


Валерий Рубанцев
ЗАНИМАТЕЛЬНАЯ ГРАФИКА НА СИ-ШАРПЕ
Самоучитель для начинающих
C#2012
.NET4.5
WINDOWS FORMS
30 ГЛАВ
70 ПРОЕКТОВ
GDI+


Валерий Рубанцев
ТОТАЛЬНЫЙ ТРЕНИНГ ПО СИ-ШАРПУ
для начинающих
НАШИ НЕНАШИ
40
.NET 4.5
Синарптак
WINDOWS FORMS
20 ГЛАВ
C#2012
60 ПРОЕКТОВ