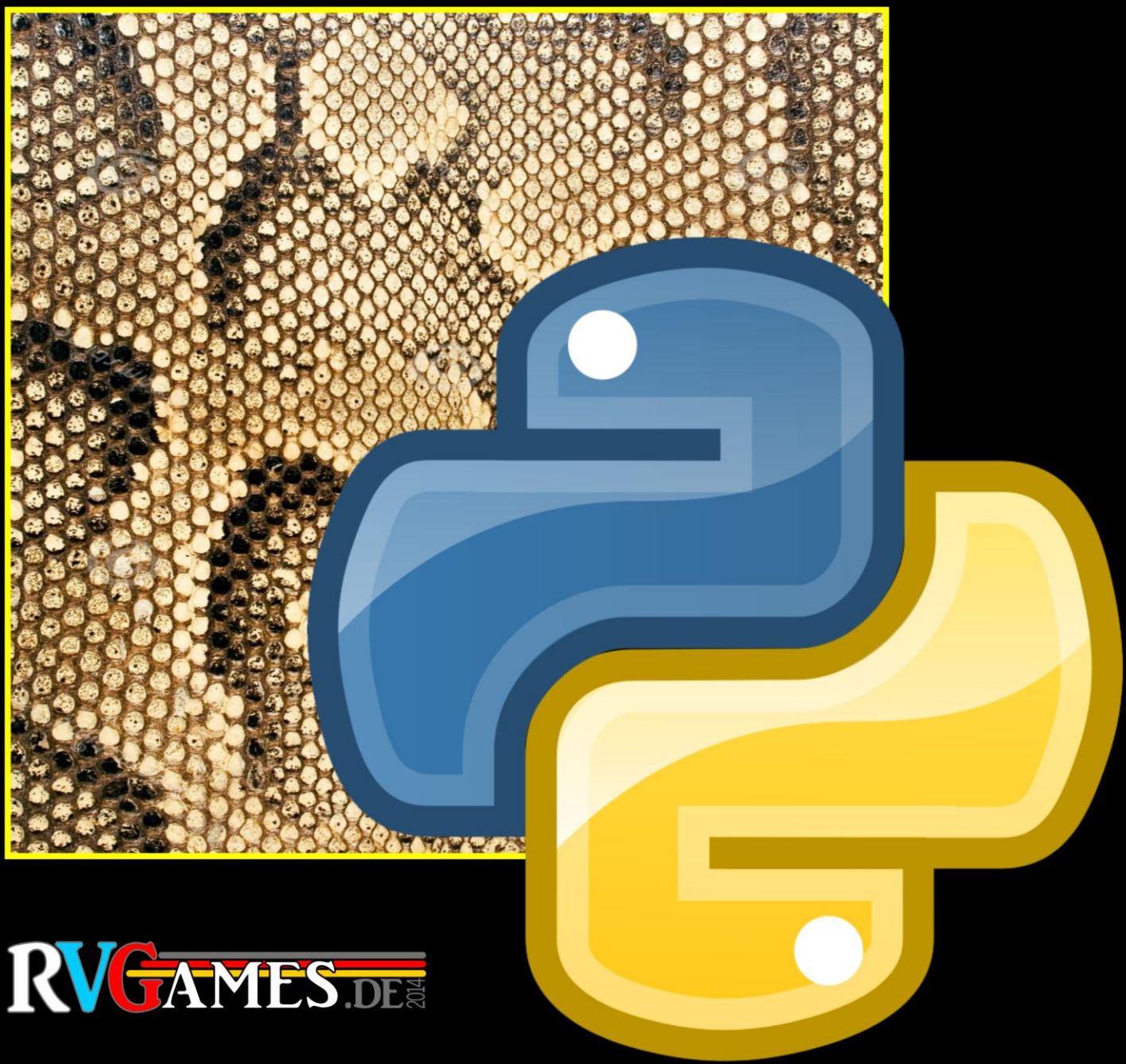
Рубанцев Валерий

Решение задач на языке Python 3.x



RVGAMES.DE 2014

Валерий Рубанцев

Решение задач на языке *Python* 3.x

Это издание представляет собой сокращённый вариант книги **Решение задач на языке Python 3.x**. Полная версия книги включает 9 глав (400 страниц).

Вы можете приобрести её (вместе с исходными кодами **всех** проектов) на сайте издательства **детскиекниги.рф**.

От автора

Python, удушающий приверженцев
других языков программирования

Во многих странах мира *Питон* используется в учебных целях, в том числе и как первый язык программирования для начинающих. В России же *Питон* в этом качестве явно недооценён.

На уровне процедурного программирования *Питон* не имеет себе равных среди современных языков. Программы, написанные на *Питоне*, как правило короче и яснее, чем, например, на более популярном в России *Си-шарпе*. В *Питоне* значительно меньше типов данных, что позволяет избежать многих проблем с их выбором и приведением. Тип *int* в *Питоне* даёт возможность работать с целыми числами произвольной длины, а тип *Decimal* – с вещественными числами любой точности. С их помощью можно разрабатывать простые, но эффективные приложения для решения математических, физических, химических и других задач, связанных с обработкой числовой информации.

К несомненным **достоинствам** *Питона* следует отнести также:

- он **бесплатен**
- **популярен** во всём мире
- **прост** в изучении, но используется профессиональными программистами
- **универсален**

- один и тот же исходный код можно запустить на компьютерах с любой из распространённых **операционных систем**: *Windows, Mac OS X, Linux*
- **поддерживается** фирмой *Майкрософт*, поэтому программы на *Питоне* можно писать непосредственно в *Visual Studio*

В этой книге подробно рассматривается решение **90 задач**: математических, комбинаторных, вероятностных, игровых.

Б***о***льшая часть задач взята из двух книг по математике для школьников:

- Нагибин Ф.Ф., Канин Е.С. - ***Математическая шкатулка***
- Кордемский Б.А., Ахадов А.А.- ***Удивительный мир чисел***

Эти книги - одни из лучших книг по математике на русском языке. Они выдержали несколько изданий и до сих пор пользуются заслуженной популярностью. Я надеюсь, что вам будет интересно решать классические задачи на компьютере.

При написании книги использовались и другие «авторитетные» источники (полный список литературы см. в конце книги):

- В. А. Дагене, Г. К. Григас, К. Ф. Аугутис - ***100 задач по программированию***
- Брудно А. Л. Каплан Л. И. - ***Олимпиады по программированию для школьников***
- Ehrhard Behrends - ***Fünf Minuten Mathematik: 100 Beiträge der Mathematik-Kolumne der Zeitung DIE WELT***
- Абрамов С.А. и др. - ***Задачи по программированию***
- Журнал ***Техника – молодёжи***

Кроме собственно решения задач, мы разработаем и **«вспомогательные» проекты**:

- Делимость чисел
- Наибольший общий делитель
- Наименьшее общее кратное

- Простые числа, Решето Эратосфена
- Факторизация чисел
- Совершенные числа
- Числовые ребусы
- Факториал
- Числа Фибоначчи
- Генерирование перестановок
- Генерирование сочетаний
- Разбиение числа на слагаемые

Несмотря на сравнительно небольшой объём книги, она охватывает все **ключевые** элементы языка *Питон*. В самом её начале вы найдёте **Тематический указатель**, который поможет вам ориентироваться во всех проектах и легко находить нужный. В конце многих глав имеются **задания для самостоятельного решения**.

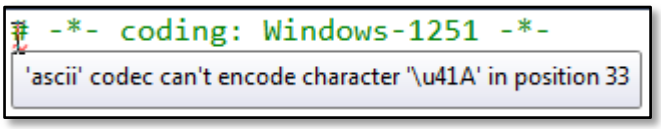
Все проекты разрабатывались в **Microsoft Visual Studio 2010** и **2013** (не *Express*!) с установленными **IronPython 2.7.4**, **Python 3.3.3** и **Python Tools for Visual Studio 2010 и 2013**.

Поскольку исходный код программ с расширением **.py* не содержит никакой специфической информации, то может быть запущен в любой другой среде разработки программ на *Питоне*, хотя бы в *IDLE*, которую мы также будем использовать при разработке проектов, так как исходный код в ней выполняется быстрее, чем в *Visual Studio*.

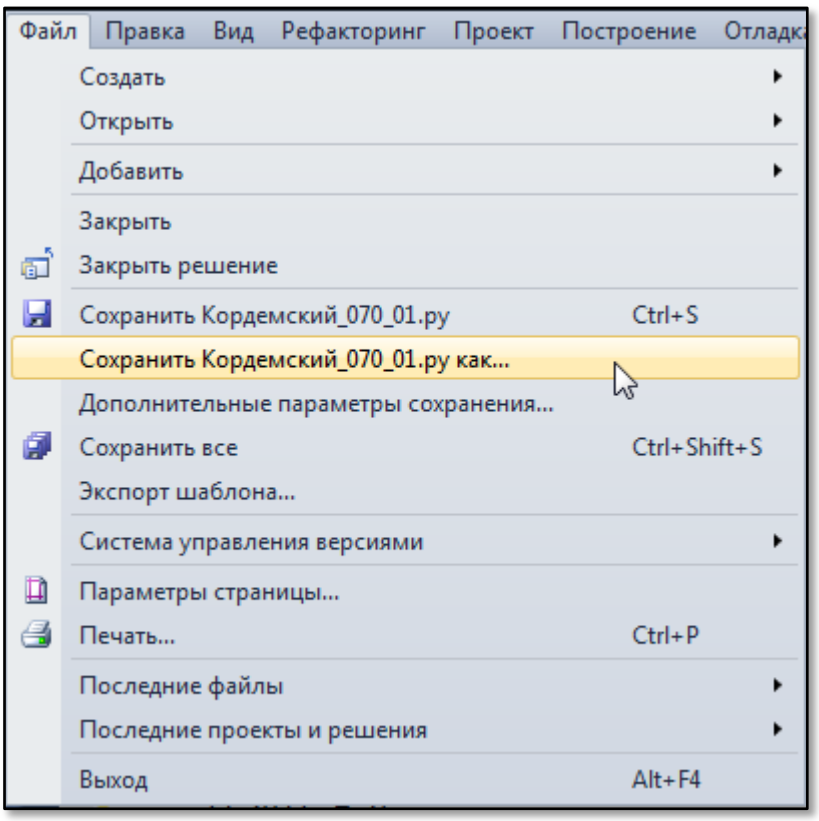
Чтобы пользоваться **русскими буквами**, после создания нового проекта вставьте в его начало строку:

```
# -*- coding: Windows-1251 -*-
```

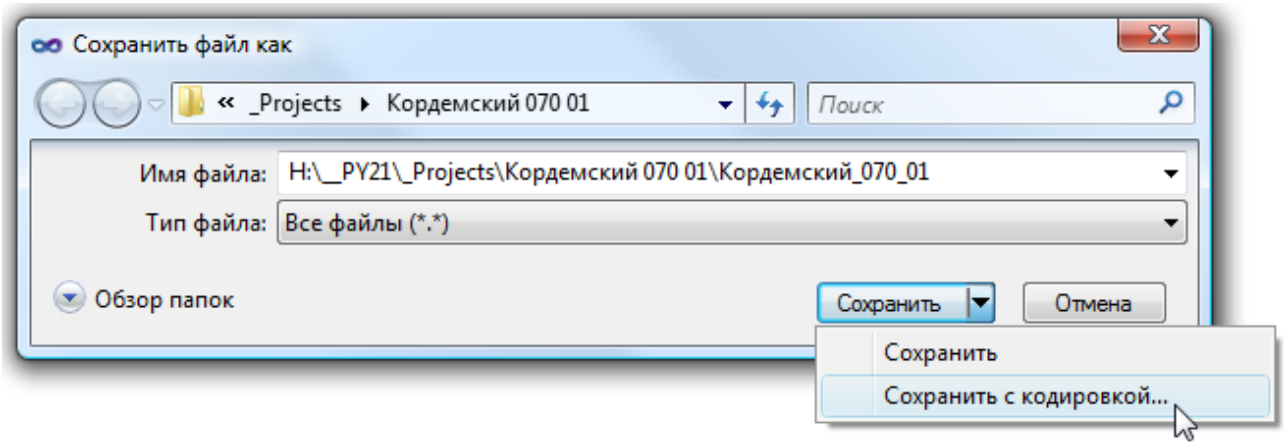
Редактор кода укажет вам на ошибку:



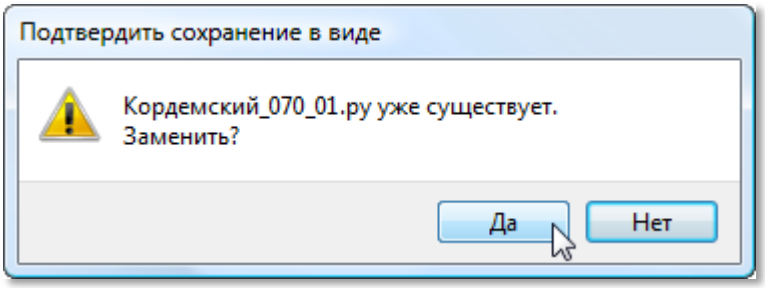
Сохраните файл программы, выполнив команду **Файл > Сохранить как...**:



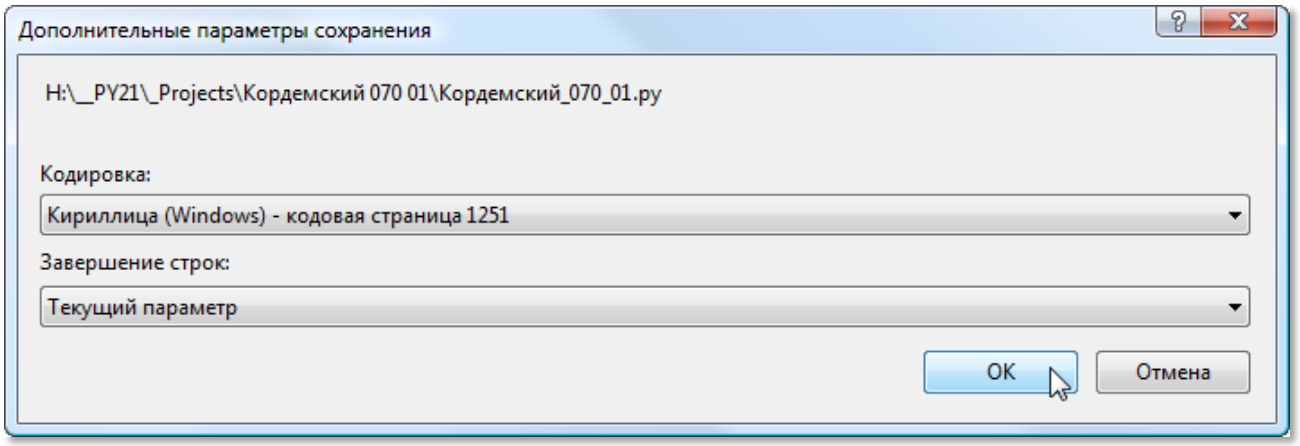
В диалоговом окне щёлкните по стрелке в правой части кнопки **Сохранить**, чтобы открылся список, и выберите из него строку *Сохранить с кодировкой...:*



В следующем окне просто нажмите кнопку **Да**,



а затем – кнопку **ОК**:



Ошибка исчезнет:

```
# -*- coding: Windows-1251 -*-
#Кордемский, с.70, Задача 1
```

Вы можете сократить эти процедуры, если сразу выполните в меню *Файл* пункт **Дополнительные параметры сохранения…**

Валерий Рубанцев

Условные обозначения, принятые в книге:

Дополнение или замечание

Ненавязчивое требование или указание

```
Исходный  код
```

Задание для самостоятельного решения

Папка с исходным кодом **программы**.

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

Оглавление

Полное оглавление

Тематический указатель

Элементы языка	Проекты
Идентификаторы	*
Выражения **Операторы**	* *
Комментарии #	*
Целый тип *int*	*
Вещественный тип *float*	*Проект Дробный ряд* *Проект Ещё один дробный ряд* *Проект Вычисляем пи и е* *Проект Тригонометрические функции*
Списки	*Проект Делится - не делится?* *Проект Тройка, семёрка и... только . . .*
Списки списков	*Проект Игра Охота на Скалоеда* *Проект Игра Охота на Скалоедов* *Проект Игра Крестики-нолики* *Проект Массивные перестановки* *Проект Сумма пятизначных чисел*
Переменные **Локальные переменные**	*
Операторы и операции присваивания **= =+ =- *= /= %=**	*Проект Тройка, семёрка и... только,* *Проект Наибольший общий делитель. . .*
Операции отношения **< > == != <= >=**	*
Логические операции **and, or и not**	*Проект Три велосипедиста,* *Проект Чудеса в решете Эратосфена . . .*
Арифметические операции **+ - * // / %**	*

Условный оператор *if* **Условный оператор *if-else*** **Вложенные условные операторы**	*
Цикл *for*	*Проект Делится - не делится? . . .*
Цикл *while*	*Проект Тройка, семёрка и... только,* *Проект Наибольший общий делитель . . .*
Вложенные циклы	*Проект Назойливый остаток,* *Проект Простые числа . . .*
Бесконечные циклы	*Проект Делится - не делится?* *Проект Назойливый остаток . . .*
Оператор *break*	*Проект Делится - не делится? . . .*
Оператор *continue*	*Проект Делится - не делится? . . .*
Классы *class* **Конструкторы**	*Проект Игра Угадай число, Проект Игра Крестики-нолики*
Поля **Методы**	*
Ключевое слово *return*	*
Класс *math*	*Проект Делится - не делится?. . .*
Класс *random*	*Проект Безошибочный прогноз; Проект Ошибочный прогноз. . .*
Форматированный вывод	*Проект Ящики,* *Проект Путёвки*

Если после название проекта стоит **многоточие** . . ., значит элемент *многократно* используется и в следующих проектах.

Звёздочка * в графе *Проекты* означает, что соответствующий элемент языка используется во *многих* проектах.

Глава #1. Числа, числа, числа…

Говорят, что числа правят миром.
Нет, они только показывают, как правят миром.

Иоганн Гёте

Детский компьютер

Как вы знаете из уроков математики, чисел бесконечно много, но их можно разбить на отдельные *подмножества* по тем или иным признакам.

Самые первые числа, которые придумали ещё первобытные люди, называются **натуральными**. Они используются для подсчёта различных предметов, например, яблок или палочек, на которых вы и сами учились считать в первом классе.

Папа спрашивает у сына:
- Скажи, сколько будет, если к трём грушам прибавить ещё две груши?
Сын отвечает:
- Не знаю, папа, мы в школе решаем задачи только про яблоки!

Множество натуральных чисел обозначается большой латинской буквой **N**, поэтому само множество можно записать так: N = {1, 2, 3, ...}. Иногда к множеству натуральных чисел относят и *нуль* (отсутствие предметов вообще): $\mathbf{N_0}$ = {0, 1, 2, 3, ...}. Множество натуральных чисел является подмножеством всех чисел и также бесконечно.

Если к натуральным числам добавить *отрицательные числа* (и нуль), то получится множество **целых чисел**. Оно обозначается большой латин-

ской буквой **Z** = {... -2, -1, 0, 1, 2, ...}. Нетрудно догадаться, что и целых чисел бесконечно много.

В арифметике обычно используют именно целые числа, но встречаются алгебраические и геометрические задачи, которые невозможно решить без **дробных** *чисел*.

Рациональные числа можно представить в виде *простой (обыкновенной) дроби:*

m/n

где:

- ***m*** - целое число;
- ***n*** - натуральное число, не равное нулю (вы, конечно, помните, что на нуль делить нельзя!).

Множество рациональных чисел обозначается буквой **Q**. Если знаменатель дроби равен 1, то вся дробь равна числителю, то есть целому числу *n*. Таким образом, все целые числа являются в то же время и рациональными (множество целых чисел - это подмножество рациональных). Но не наоборот!

Рациональные числа можно представить также в виде *конечной десятичной дроби* (1/2 = 0,5) или *бесконечной периодической десятичной дроби* (1/7 = 0,1428571...).

Иррациональные числа не могут быть представлены в виде простой дроби (а также в виде конечной или бесконечной десятичной периодической дроби). Таким образом, иррациональным числом называют любое число, представимое в виде *бесконечной непериодической* десятичной дроби. Примером такой дроби служит корень квадратный из двойки. Иррациональность этого числа была известна уже древним математикам, которые доказали несоизмеримость стороны и диагонали квадрата.

Иррациональные числа обозначают буквой **I**.

Множество **действительных**, или **вещественных чисел** объединяет множества рациональных и иррациональных чисел. Их принято наглядно представлять в виде точек на *числовой прямой* (Рис. 1.1).

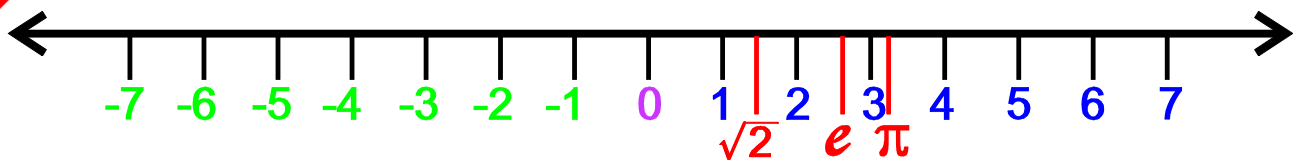


Рис. 1.1. Числовая прямая

Множество действительных чисел обозначают буквой **R** (от их латинского названия *numerus realis*).

К иррациональным числам относятся знаменитые числа - **π** (*nu*, отношение длины окружности к диаметру) и ***e*** (основание натуральных логарифмов).

В программах на языке *Питон* чаще всего используют такие **числовые типы данных**:

int – для целых чисел произвольной длины

float – для вещественных чисел

В третьей версии *Питона* появился класс **Decimal**, помощью которого легко манипулировать вещественными числами произвольной точности.

Кроме собственно чисел, нам понадобятся и два «числовых» класса. Класс **math** нужен для математических вычислений, а класс **random** – для генерирования случайных чисел.

На следующей странице вы можете снять учебный стресс, выполнив (если желаете – на время) забавный **числовой тест**.

Найди сотню!

Последовательно найдите числа от 1 до 100!

Признаки делимости чисел

Целые числа и их свойства изучает *теория чисел*, или *высшая арифметика*. В математических задачах (да и в жизни тоже) нередко нужно быстро определить, делится ли одно число на другое или нет. При этом сам результат деления неважен. У каждого натурального числа имеется, по крайней мере, два делителя - это единица и само число (у единицы они совпадают!). Если других делителей нет, то число называется **простым**. К ним мы вернёмся немного позже, а сейчас давайте вспомним **признаки** (то есть *правила*) **делимости**.

Если делитель – натуральное число, но на него можно разделить любое другое натуральное число – либо нацело, либо с остатком:

делимое : делитель = частное + остаток

Нас интересует только деление чисел, при котором остаток от деления равен **нулю**.

Признак делимости на 2

Самый простой признак: **число делится на 2 только тогда, когда его последняя цифра равна 0, 2, 4, 6 или 8**. Если число делится на 2, то оно называется **чётным**, если не делится - **нечётным**. Примеры чётных чисел: 2012, 92, 4, 76, 58. Нечётных: 2013, 91, 5, 77, 61.

Иногда этот признак формулируют проще: **число делится на 2 тогда и только тогда, когда его последняя цифра чётная**.

Признак делимости на 3

Число делится на 3 тогда и только тогда, когда сумма его цифр делится на 3. Например, число 2013 кратно 3, поскольку сумма его цифр равна 6: 2 + 1 + 3 = 6 (при подсчёте нули не учитываем).

Если сумма цифр также выражается не однозначным числом, то следует найти сумму его цифр. Если в результатае сложения цифр получится одно из чисел 3, 6 или 9, то число делится на 3. В противном случае – не делится. Например, сумма цифр числа 123456789 равняется 45. Число двузначное –

опять находим сумму его цифр: 4 + 5 = 9. Получили девятку – значит, исходное число 123456789 делится на три.

Признак делимости на 4

Очевидно, что числа, кратные четырём, должны быть чётными. Но этого мало, поэтому мы оставляем от числа только *две последние цифры* и рассматриваем получившееся двузначное число.

Если число сразу *двузначное*, то ничего отбрасывать не нужно. А если *однозначное*, то достаточно вспомнить таблицу умножения.

Если это двузначное число делится на 4, то и всё число также делится на четыре.

Почему достаточно рассмотреть только последние две цифры числа? – На этот вопрос легко ответить, если вспомнить, что сотня делится на 4 без остатка. Естественно, любое число сотен также разделится на 4, поэтому разряды сотен, тысяч и так далее в проверяемом числе можно не учитывать.

Чтобы ещё упростить проверку, сложите число десятков с половиной единиц. Если сумма чётная, то исходное число делится на 4, в противном случае не делится.

Опять проверим год 2013. Число из последних двух цифр равно 13. Оно на 4 не делится, значит, 2013 не кратно четырём. Возьмём другое число - 4567896. Оставляем две цифры - 96. Складываем 9 с половиной от 6, то есть тройкой и получаем 12. Это число кратно четырём, значит, число 4567896 делится на 4.

Признак делимости на 5

Это правило очень похоже на признак делимости на двойку. **Число делится на 5, если оно оканчивается на 0 или 5.**

Признак делимости на 6

Число делится на 6, если **одновременно выполняются признаки делимости на 2 и 3.**

Признак делимости на 7

Хорошего признака делимости чисел на 7 не существует, зато имеется немало достаточно сложных и запутанных. Из них мы выберем один – самый простой и «универсальный».

Разбиваем заданное число, *начиная с конца*, на группы, состоящие из трёх цифр. Например, если мы проверяем число 4567896, то получим **три** группы цифр:

> **4 567 896**
>
> **3 2 1**
> **+ - +**

> Кстати говоря, в книгах так зачастую и печатают длинные числа, чтобы легче было распознать разряды сотен, тысяч и так далее.

Теперь первое (считаем сзади!) число мы берём со знаком плюс, второе - со знаком минус, третье – снова о знаком плюс. То есть знаки плюс и минус *чередуются*. Составляем из чисел с их знаками арифметическое выражение и вычисляем его значение:

896-567+4 = 333

Если результат делится на 7, то и всё число также делится на 7. В противном случае не делится.

В нашем примере число 333 на 7 не делится, значит, этот вывод относится и к исходному числу 4567896.

Признак делимости на 8

Число делится на 8, если оно чётное, а **число, составленное из трёх последних цифр, делится на 8.** Так как делить трёхзначное число на 8 тоже нелегко, то можно воспользоваться тем же приёмом, что и в *признаке делимости на 4.*

> Тысяча делится на 8 без остатка. Любое число тысяч также разделится на 8, поэтому разряды тысяч и далее в проверяемом числе можно не учитывать.

Рассмотрим три последние цифры. К числу, образованному первыми двумя цифрами, добавьте половину единиц, а затем к числу десятков добавьте половину единиц получившегося числа. Если результат - чётное число, то исходное число делится на 8.

Проясним этот алгоритм на *примере*. Начнём с того же числа 2013. Последние три цифры дают двузначное число 13, которое на 8 не делится. Следовательно, не делится и число года. Возьмём другое число - 123457928. Оставляем для проверки трёхзначное число 928. Число из первых двух цифр равно 92. Складываем его с половиной единиц - 4 - и получаем 96. Дальше действуем, как в *признаке делимости на 4*: 9 + 3 = 12. Это число кратно двум, поэтому всё число 123457928 делится на 8.

Признак делимости на 9

Этот признак напоминает правило для тройки. **Число делится на 9 тогда и только тогда, когда сумма его цифр делится на 9.** Раньше мы установили, что число 2013 делится на 3, а сумма его цифр равна *шести*. Поэтому, согласно этому признаку делимости, на 9 оно не делится.

> Если сумма цифр выражается не однозначным числом, то следует найти сумму его цифр. То есть действовать так же, как и в признаке делимости на 3.

Признак делимости на 10

Ещё проще, чем признак делимости на 5. **Число делится на 10 тогда и только тогда, когда оно заканчивается на 0.** Например, число 201**0** делится на 10, а число 201**3** не делится.

Признак делимости на 11

Самое любопытное правило; не все его знают, но оно помогает очень просто определить, делится ли, например, номер автобусного билета на 11.

Чтобы узнать, делится ли число на 11, нужно подсчитать отдельно сумму цифр, стоящих на *нечётных* и *чётных* местах в исходном числе. Если они равны, то число кратно 11.

В противном случае нужно из первой суммы вычесть вторую. Если разность делится на 11, то и всё число делится на 11.

> Если сумма первых трёх цифр равна сумме трёх последних, то такой билет называется **счастливым**.

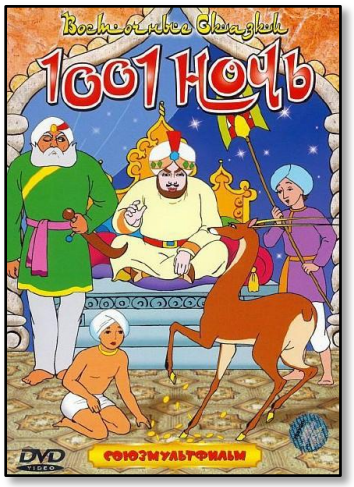
> Здесь мы для краткости употребляем слово *цифры*, но вы должны понимать, что речь идёт об однозначных **числах**, которые записываются этими цифрами.

Например, число **123453** делится на 11, так как 1 + 3 + 5 = 2 + 4 + 3 = 9. А число **123456** не делится (проверьте сами!).

Другой признак делимости на 11 полностью совпадает с признаком делимости на 7, но делить сумму чисел нужно на 11.

Признак делимости на 12

Число делится на 12 тогда и только тогда, когда **одновременно выполняются признаки делимости на 3 и 4**.



Признак делимости на 13

Признак делимости на 13 тот же самый, что для чисел 7 и 11 (*второй* способ), но делить сумму чисел нужно на 13. Поэтому я недаром назвал этот признак *универсальным*.

Интересно, что наименьшее число, которое одновременно делится на 7, 11 и 13, равняется 7 x 11 x 13 = 1001, то есть сказочному числу арабских ночей.

Признак делимости на 19

Признак делимости чисел на 19 хорошо описан в книге Якова Перельмана *Занимательная алгебра*.

Число делится без остатка на 19 тогда и только тогда, когда число его десятков, сложенное с удвоенным числом единиц, кратно 19.

Опять проверим большое число *123457928*.

Оно содержит 12345792 десятка и 8 единиц. По правилу, получаем:

12345792 + 16 = 12345808

Опять находим число десятков и единиц: 1234580 и 8 – и сумму:

1234580 + 16 = 1234596

И так продолжаем дальше:

123459 + 12 = 123471

12347 + 2 = 12349

1234 + 18 = 1252

1252 + 4 = 1256

125 + 12 = 137

13 + 14 = 27

Вывод: число 123457928 на 19 не делится.

Добавим к исходному числу шестёрку и проверим сумму на делимость:

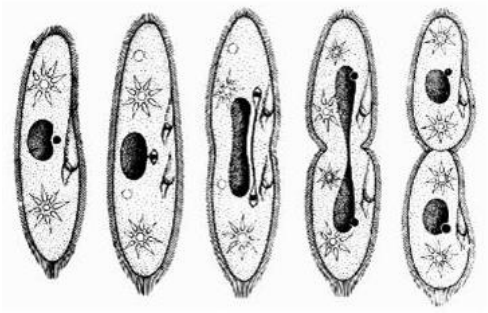
123457934
12345793 + 8 = 12345801
1234580 + 2 = 1234582
123458 + 4 = 123462
12346 + 4 = 12350
123 + 10 = 133
13 + 6 = 19

Вывод: число 123457934 на 19 делится.

> Интересные математические фокусы, связанные с признаками делимости, вы найдёте в книге Мартина Гарднера *Математические досуги* **[ГМ72]**, Глава 19.
>
> Если вы серьёзно интересуетесь свойствами чисел, то прочитайте книгу Н.Н. Воробьёва *Признаки делимости* **[ВНН88]**.

Проект *Делится - не делится?*

Бесконечный цикл ***while***
Функции *с параметрами*
Оператор деления по модулю ***%***
Оператор деления ***//***
Цикл ***for***
Условныq операторы ***if***
Оператор ***return***
Метод для извлечения квадратного корня ***math.sqrt***
Списки
Метод печати в консольном окне ***print***
Метод ввода строки в консольном окне ***input***
Преобразование строки в число типа int с помощью метода ***int***



Компьютер должен считать,
а человек - думать.

Программистская пословица

Мы вспомнили признаки делимости чисел, без которых человеку обойтись трудно, а вот компьютеру они совсем не нужны, потому что он и без них считает охотно и быстро.

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

С помощью операции деления по модулю можно легко проверить, делится ли одно число на другое нацело или нет. Например, мы хотим узнать, делится ли число 2014 на 7. Пишем:

```
if 2014 % 7 == 0:
    print(('Делится'))
else:
    print('Не делится')
return
```

```
C:\Python32\python.exe
Не делится
Press any key to continue . . .
```

Рис. 1.2. Делится – не делится

Запускаем программу и тут же узнаём, что *не делится* (Рис. 1.2). То есть нам только и нужно, что проверить остаток от деления. Если он равняется нулю, то первое число делится на второе. Вот и вся премудрость!

В функции **main** пользователь самостоятельно выбирает число, для которого программа найдёт все его делители:

```
# -*- coding: Windows-1251 -*-
#Делимость чисел


#ГЛАВНАЯ ФУНКЦИЯ
def main():
    print('Делимость чисел')
    print()

    #бесконечный цикл ввода данных -
    #пока пользователь не закроет программу
    #или не введет 0:
    while True:
        print("Введите натуральное число  >  ", end = '')
        num = int(input())
        if (num == 0): return
        Solve1(num)
        print()
```

После того как пользователь ввёл число, мы передаём его функции **Solve1**, которая в цикле *for* пытается поделить его на числа из диапазона *1.. num* и печатает в консольном окне все делители заданного числа (Рис. 1.3):

```
def Solve1(num):
    #печатаем результаты в консольном окне:
    print()
    print("Число " + str(num) + " делится на:")
    for i in range(1, num+1):
         if (num % i == 0):
             print(str(i) + " ", end = '')
    print()
    print()
```

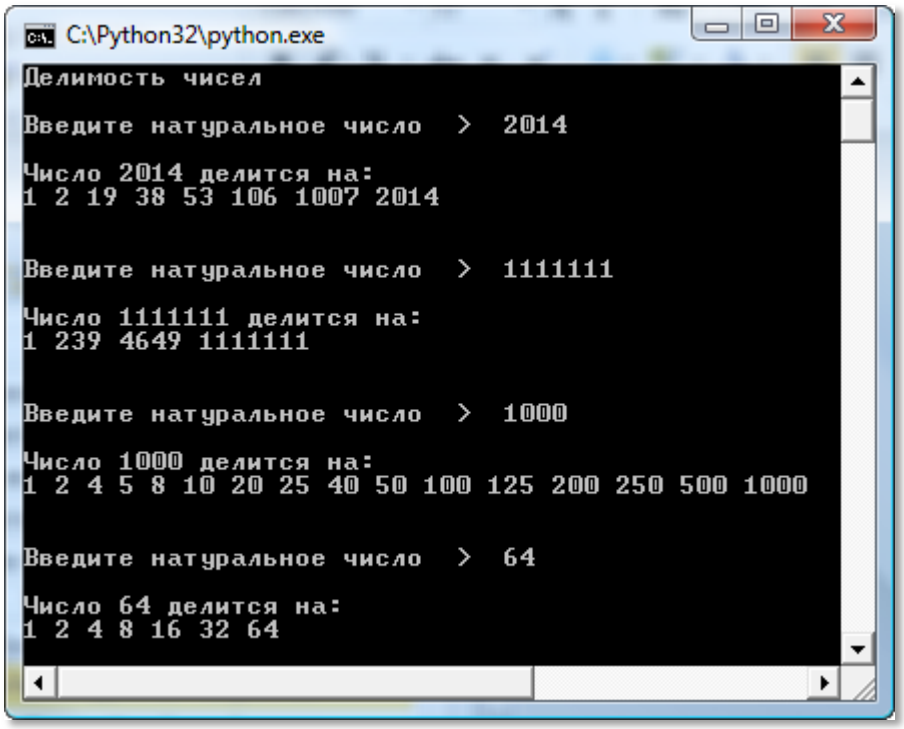


Рис. 1.3. Программа в действии

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

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

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

Немного подправим код – и пользователь может находить делители огромных чисел (Рис. 1.4):

```
def Solve2(num):
    #печатаем результаты в консольном окне:
    print()
    print("Число " + str(num) + " делится на:")
    for i in range(1, num//2+1):
         if (num % i == 0):
             print(str(i) + " ", end = '')
    print(num)
    print()
```

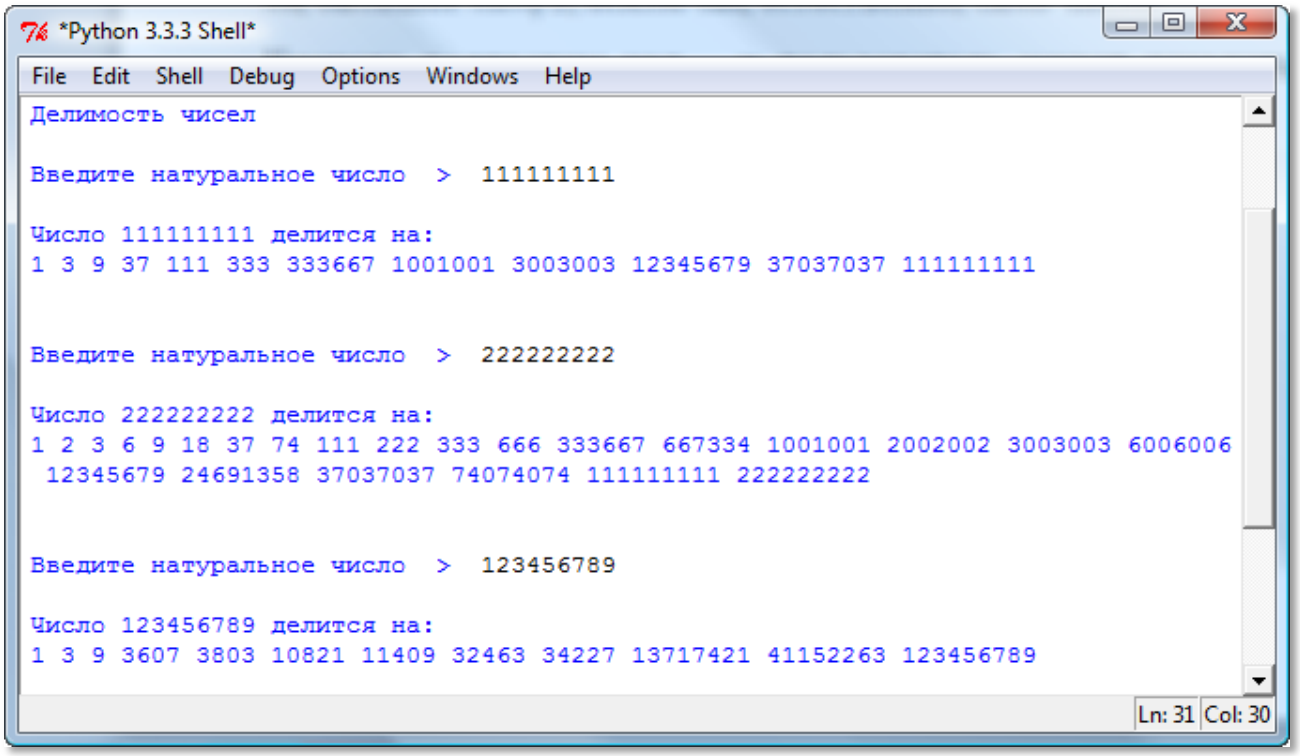


Рис. 1.4. Оптимизированная программа

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

> **1 2 4 8 16 32 64**

Проверяем утверждение:

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

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

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

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

```
def Solve3(num):
    #список для записи делителей:
    res = []
    for i in range(1, round(math.sqrt(num))+1):
         if (num % i == 0):
            res.append(i)
            #не допускаем повторов делителей:
            if (i*i != num):
                res.append(num // i)
    #печатаем результаты в консольном окне:
    print()
    print("Число " + str(num) + " делится на:")
    print(*res, sep=' ', end='')
    print()
```

И вот почему. Для первого делителя – единицы – мы найдём парный делитель, равный заданному числу *num*, для второго парный делитель равен *(num / второй делитель).* То есть делители мы напечатаем не по порядку (Рис. 1.5).

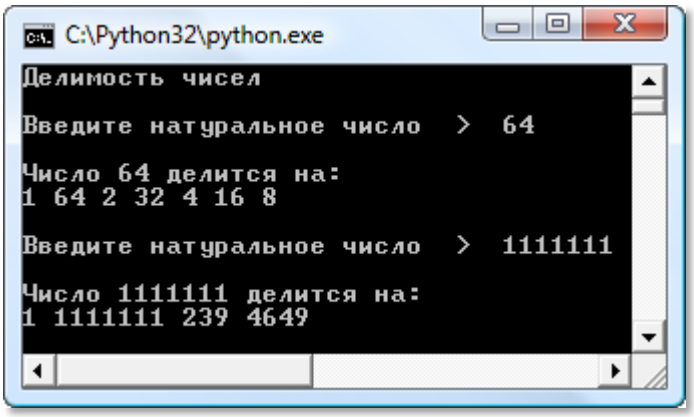


Рис. 1.5. Неупорядоченные делители

Чтобы выправить ситуацию, мы до печати результатов отсортируем список *res* с помощью метода **sort** списочного класса, а затем напечатаем делители строго по ранжиру:

```
def Solve4(num):
    #список для записи делителей:
    res = []
    for i in range(1, round(math.sqrt(num))+1):
         if (num % i == 0):
            res.append(i)
            #не допускаем повторов делителей:
            if (i*i != num):
```

```
                res.append(num // i)
    #печатаем результаты в консольном окне:
    res.sort()
    print()
    print("Число " + str(num) + " делится на:")
    print(*res, sep=' ', end='')
    print()
```

Теперь даже для огромных чисел мы мгновенно выписываем все делители (Рис. 1.6).

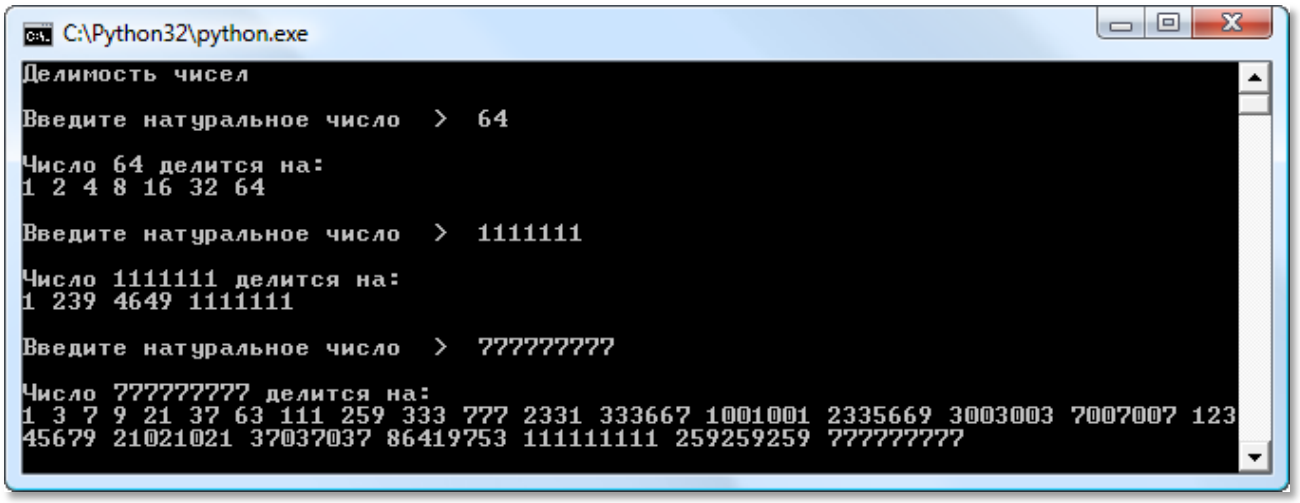
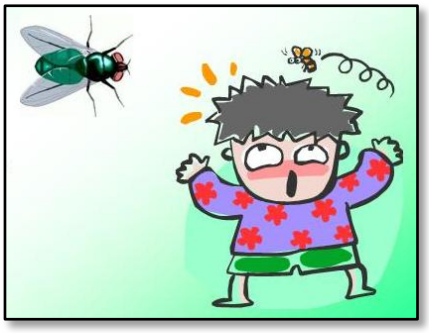


Рис. 1.6. Молниеносное нахождение делителей

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

Проект *Назойливый остаток*

Вложенные циклы ***for***
Бесконечный цикл ***while***
Условный оператор ***if***
Оператор ***%***
Оператор ***break***
Оператор ***continue***



В книге Бориса Кордемского и Аскера Ахадова *Удивительный мир чисел* **[КА86]** вы найдёте немало интересных задач, в том числе и на делимость. На странице 86 авторы предлагают решить задачу **Назойливый остаток**:

> Некоторые числа, кратные числу 7, при делении на 2, на 3, на 4, на 5 и на 6 дают остаток 1.
>
> **Найдите наименьшее из таких чисел.**

Эта же задача напечатана в книге Фёдора Нагибина и Евгения Канина *Математическая шкатулка* **[Нагибин88]**, задача 42, страницы 18-19, но в более занимательной форме:

> Колхозница привезла на рынок для продажи корзину яиц. Продавала она их по одной и той же цене. После продажи яиц колхозница пожелала проверить, верно ли она получала деньги. Но вот беда: она забыла, сколько у неё было яиц. Вспомнила она только, что когда перекладывала яйца по 2, то оставалось одно яйцо; одно яйцо оставалось также при перекладывании яиц по 3, по 4, по 5, по 6. Когда же она перекладывала яйца по 7, то не оставалось ни одного.
>
> **Помоги колхознице сообразить, сколько у неё было яиц.**

Бросаемся на помощь бестолковой колхознице и в функции **main** вызываем функцию *Solve* для решения этой головоломки:

```
# -*- coding: Windows-1251 -*-
#Кордемский, с.86, Задача 8

#ГЛАВНАЯ ФУНКЦИЯ
def main():
    print("Назойливый остаток")
```

```
    print()
    Solve()

. . .

if __name__ == "__main__":
    main()
```

Поскольку речь идёт о **натуральных** числах, то мы можем начать наши поиски с *единицы*:

```
#РЕШАЕМ ЗАДАЧУ
def Solve():
    #мин. число:
    minnum = 1
```

Если число 1 не является решением задачи, то мы переходим к двойке, к тройке, и так далее – пока не найдётся искомое число **num**.

Так как мы всякий раз добавляем к начальному значению переменной **minnum** единицу, то вполне разумно делать это в цикле **for**, в котором переменная **num** и будет играть роль *переменной цикла*.

Обычно к циклу *for* прибегают тогда, когда число повторов точно известно. В нашем же случае, мы знаем только начало цикла (*min = 1*), но не знаем, на каком числе он закончится. Впрочем, мы вполне разумно можем предположить, что у колхозницы было не очень много яиц, и задать заведомо большее значение для верхней границы цикла. Так мы организуем почти бесконечный цикл, который прервём сразу же, как только искомое число будет обнаружено. Для этого мы воспользуемся *флажком* – логической переменной **flg**. Если очередное число *num* выдержит все проверки, значение флага останется верным (*True*), а мы с помощью оператора *break* прервём цикл *for* и напечатаем решение задачи в консольном окне:

```
    for num in range(minnum, 1000):
        flg = True
        #число не кратно семи:
        if num % 7 != 0:
            continue;
```

Проверку очередного числа на остаток, равный единице, также можно проводить в цикле, чтобы не выписывать несколько одинаковых условий.

Если число *num* при делении хотя бы на одно из чисел 2..6 не даёт в остатке 1, то мы сбрасываем флажок и прерываем **внутренний** цикл *for*:

```
        for d in range(2, 6+1):
            if (num % d != 1):
                flg = False
                break
        if flg:
           break
    print("Искомое число равно " + str(num))
    print()
```

На Рис. 1.7 вы видите ответ на эту задачу.

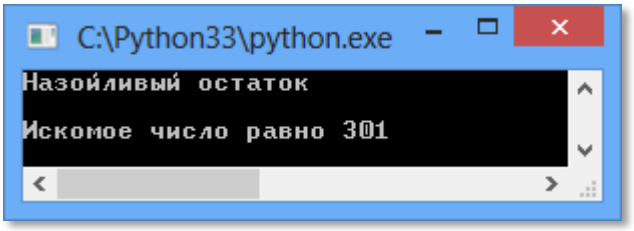


Рис. 1.7. Яиц было немало!

Настоящий **бесконечный цикл** легко получить с помощью конструкции **while True**:

```
def SolveWhile():
    #мин. число:
    num = 0
    while True:
        flg = True
        num += 1
        #число не кратно семи:
        if num % 7 != 0:
            continue;
        for d in range(2, 6+1):
            if (num % d != 1):
                flg = False
                break
        if flg:
            break

    print("Искомое число равно " + str(num))
    print()
```

Не знаю, как вам, но мне интересно, а *есть ли ещё и другие числа, которые имеют назойливый остаток*?

Имея компьютер, мы легко утолим своё любопытство – достаточно написать новую функцию **Solve2**, которая поразительно напоминает первую версию:

```
def Solve2(num):
    #печатаем результаты в консольном окне:
    print()
    print("Число " + str(num) + " делится на:")
    for i in range(1, num//2+1):
         if (num % i == 0):
             print(str(i) + " ", end = '')
    print(num)
    print()
```

Тут, конечно, следует учесть, что бесконечный цикл уже не годится, потому что он действительно станет бесконечным, если искомых чисел очень много (а это нетрудно предвидеть).

Запускаем приложение и видим, что среди первой десятки тысяч натуральных чисел довольно много подходящих под условие задачи (Рис. 1.8).

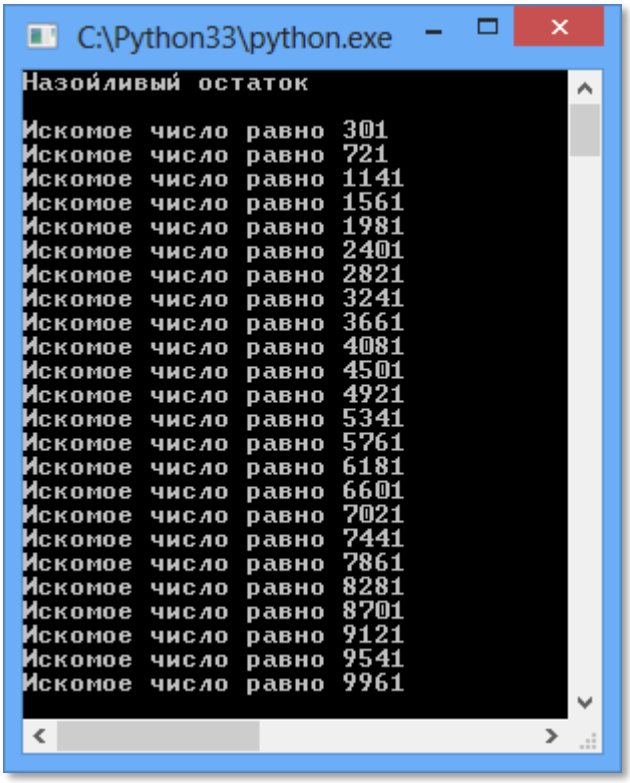


Рис. 1.8. Неназойливый список назойливых чисел

Также нетрудно подметить такую **закономерность**. Если обозначить через **n** номер искомого числа, то все числа с назойливыми остатками можно легко найти по формуле:

$$num = 301 + 420(n-1)$$

Любопытно, что если решать задачу для чисел, кратных не 7, а б***о***льшим числам, то это непременно должны быть **простые** числа. Например, решения для чисел 11 и 13 показаны на Рис. 1.9.

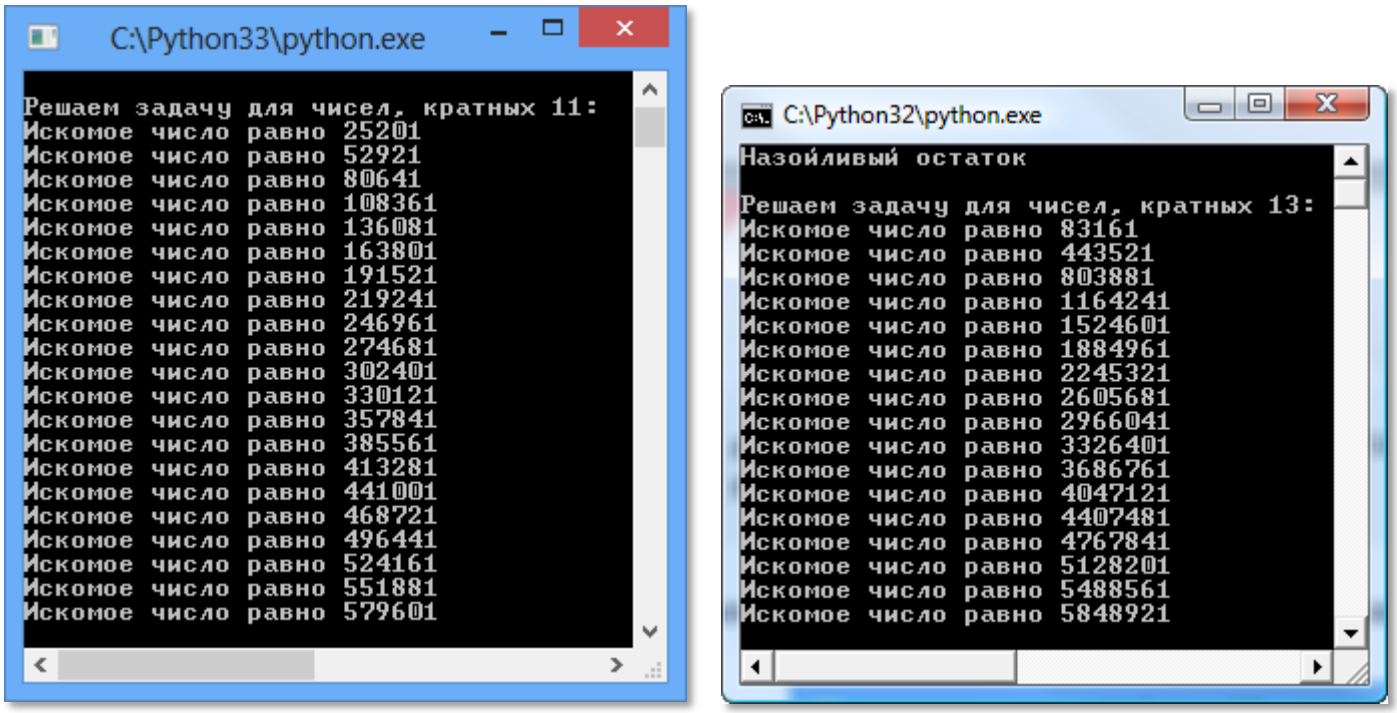


Рис. 1.9. Исследования продолжаются

Чтобы понять, почему так происходит, нам нужно научиться вычислять *НОД*, *НОК* и находить простые числа!

Исходный код программы находится в папке **Кордемский 086 08**.

Задания для самостоятельного решения

Вы ошиблись в подсчёте

***Удивительный мир чисел.*. Задача 4 (2.2)], страница 50**

Ученик покупает 18 карандашей, 6 тетрадей, 12 ластиков, 9 блокнотов и несколько тетрадей для рисования по 15 к. Девушка-продавец выписала чек на 1 р. 52 к. Взглянув на чек, мальчик сразу же сказал продавцу: «Вы ошиблись в подсчёте». Девушка пересчитала и исправила свою ошибку.

Как удалось пареньку так быстро обнаружить просчёт?

Ответ: Стоимость каждой покупки, а значит и общая сумма кратна трём, но 1 р. 52 к. на три не делится.

Задача #24

Математическая шкатулка

Какое целое число делится (без остатка) на любое целое число, отличное от 0?.

Ответ: 0

Задача #25

Математическая шкатулка

Сумма каких двух натуральных чисел равна их произведению?

Ответ: 2 + 2 = 2 x 2-

Глава #2. НОД, НОК и компания

А теперь мы напишем две родственные программы – для вычисления **НОД** (*наибольшего общего делителя* двух чисел) и **НОК** (*наименьшего общего кратного*).

Проект *Наибольший общий делитель*

***Функция** с параметром*
***Фуекция** без параметров*
*Цикл **for***
*Оператор деления по модулю **%***
*Оператор деления **//***
*Бесконечный цикл **while***
*Метод **int***
*Оператор **return***
*Условный оператор **if***
*Условный оператор **if-else***
*Цикл **while***
Комбинированные операторы присваивания

Для вычисления **НОД** мы воспользуемся *простым* и *быстрым* алгоритмами древнегреческого математика *Евклида* (Рис. 2.1).

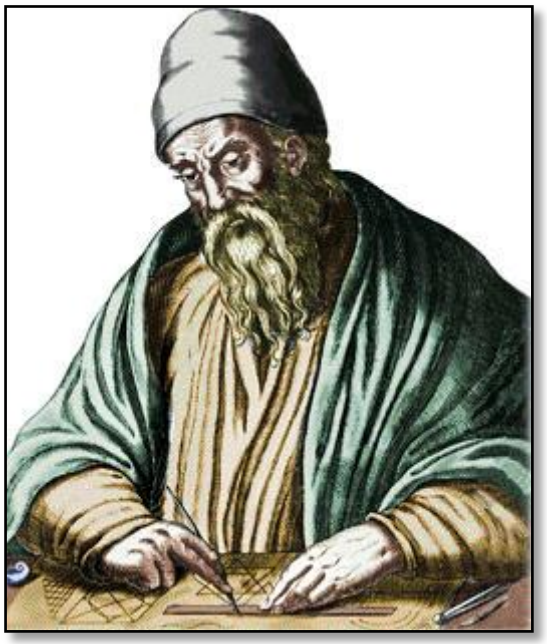


Рис. 2.1. Евклид (Εὐκλείδης, ок. 325 года до н.э. - до 265 года до н.э.)

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

Простой алгоритм Евклида основан на следующих свойствах:

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

Из них следует:

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

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

При *равенстве* чисел алгоритм сработает правильно.

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

```
# -*- coding: Windows-1251 -*-
#ПРОГРАММА ДЛЯ НАХОЖДЕНИЯ НАИБОЛЬШЕГО
#ОБЩЕГО ДЕЛИТЕЛЯ ДВУХ НАТУРАЛЬНЫХ ЧИСЕЛ (НОД)


#ГЛАВНАЯ ФУНКЦИЯ
def main():
    print('НОД двух чисел')
    print()

    #бесконечный цикл ввода данных -
    #пока пользователь не закроет программу
    #или не введёт отрицательные числа:
    while True:
        print("Введите первое число >  ", end = '')
        number1 = int(input())
        print("Введите второе число >  ", end = '')
```

```
        number2 = int(input())
        if (number1 + number2 < -1): return

        #если первое число меньше второго,
        #то меняем их значения:
        if (number1 < number2):
            number1, number2 = number2, number1

        #находим НОД:
        if (number2 == 0):
            nod = number1
        else:
            nod = Euklid(number1, number2)

        #печатаем НОД:
        print("НОД = (" + str(number1) + "," + str(number2) + ") = " +
str(nod))
        print()
```

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

```
#ПРОСТОЙ АЛГОРИТМ ЕВКЛИДА
def Euklid(n1, n2):
    while (n2 != n1):
        if (n1 >= n2): n1 -= n2
        else: n2 -= n1;
    return n1
```

> Тут, конечно, следует учитывать значения чисел: если они очень большие, то лучше пользоваться *быстрым*, а не простым алоритмом.

В цикле *while* мы на каждой итерации вычитаем из большего числа меньшее – до тех пор, пока значения переменных *n1* и *n2* не сравняются. Поскольку с каждым разом одно из чисел *уменьшается*, то рано или поздно это условие будет выполнено.

Вычисленное значение *НОД* мы возвращаем методу *main*, который и публикует его в окне консоли. При небольших числах алгоритм работает очень быстро (Рис. 2.2).

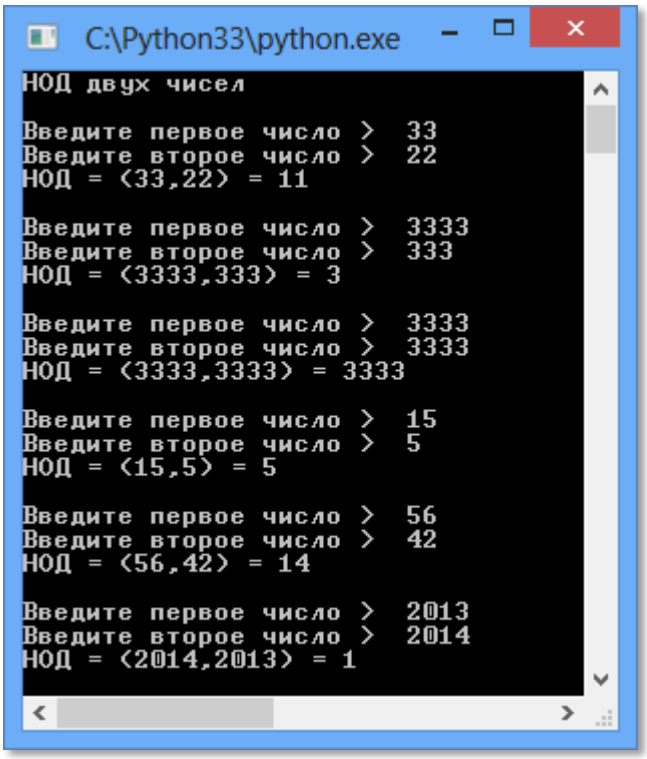


Рис. 2.2. Вычисляем *НОД* двух чисел

Нагнетаем обстановку, задавая всё более «солидные» числа, и простой алгоритм начинает «буксовать» (Рис. 2.3).

А при дальнейшем увеличении чисел он попросту впадает в к***о***му! И тут нам на выручку приходит **быстрый алгоритм Евклида***:*

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

```
#БЫСТРЫЙ АЛГОРИТМ ЕВКЛИДА
def SpeedEuklid(n1, n2):
    while (n2 > 0):
        n1, n2 = n2, n1 % n2
    return n1
```

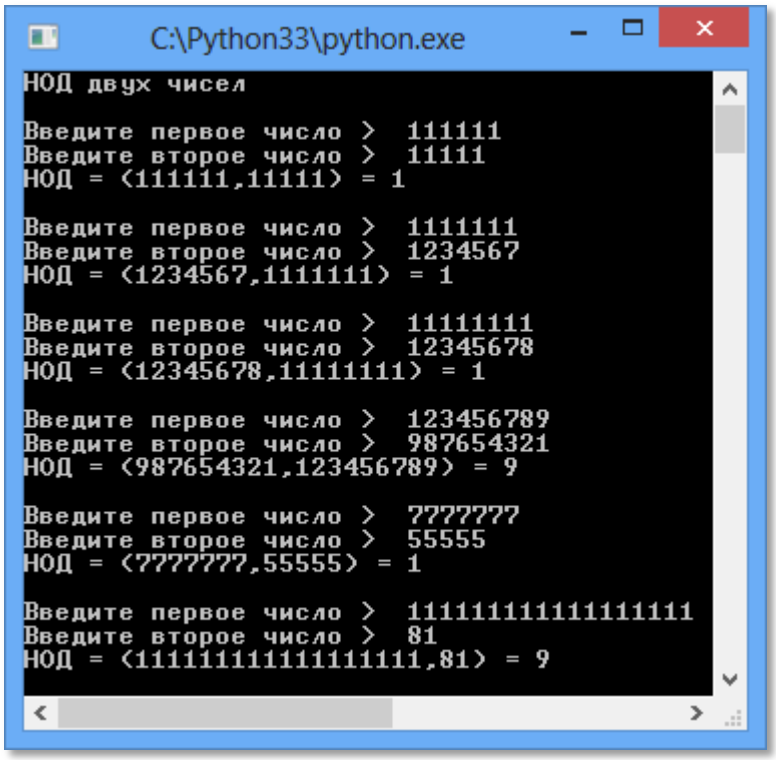


Рис. 2.3. Вычисляем *НОД* больших чисел

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

> **Быстрый алгоритм Евклида** основан на свойстве:
>
> НОД(m,n) = НОД(n, m%n) при m > n

Этот алгоритм не собьёшь с толку никакими числами (Рис. 2.4)! Вы можете взять любые числа и убедиться, что алгоритм действительно работает практически мгновенно.

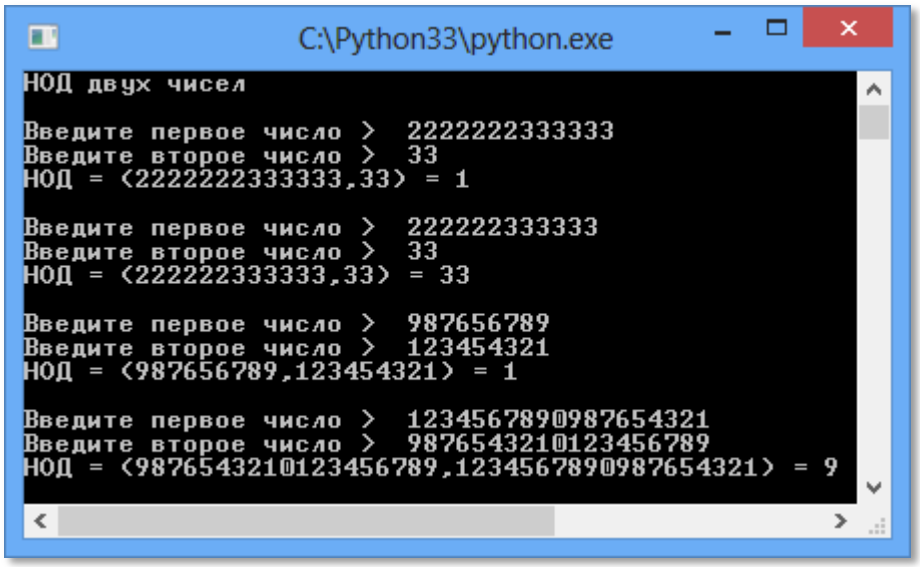


Рис. 2.4. Наша программа легко справляется и с очень большими числами!

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

number1 = 42

number2 = 14

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

n = 42 % 14 = 0

И присваиваем новые значения переменным:

number1 = 14

number2 = 0

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

Рассмотрим **другой пример**:

```
number1 = 56
number2 = 42
n = 56 % 42 = 14
number1 = 42
number2 = 14
```

Уже после одного цикла мы пришли к первому примеру.

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

Проект *Наименьшее общее кратное*

Бесконечный цикл ***while***
Метод ***int***
Оператор ***return***
Условный оператор ***if***
Функция *с параметрами*

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

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

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

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

```
# -*- coding: Windows-1251 -*-
#ПРОГРАММА ДЛЯ ВЫЧИСЛЕНИЯ НАИМЕНЬШЕГО ОБЩЕГО
#КРАТНОГО ДВУХ ЧИСЕЛ (НОК)


#ГЛАВНАЯ ФУНКЦИЯ
def main():
    print('НОК  двух чисел')
    print()

    #бесконечный цикл ввода данных -
    #пока пользователь не закроет программу
    #или не введёт отрицательные числа:
    while True:
        print("Введите первое число >  ", end = '')
        number1 = int(input())
        print("Введите второе число >  ", end = '')
        number2 = int(input())
        if (number1 + number2 < -1): return

        #если первое число меньше второго,
        #то меняем их значения:
        if (number1 < number2):
            number1, number2 = number2, number1
```

```
        #находим НОК:
        if (number1 * number2 == 0):
            nok = 0
        else:
            #вычисляем НОК:
            nok= number1 * number2 // SpeedEuklid(number1, number2)
        #печатаем НОК:
        print("НОК = (" + str(number1) + "," + str(number2) + ") = " +
str(nok))
        print()
```

Поскольку *НОД* вычисляется очень быстро, то и *НОК* мы получаем в считанные мгновения (Рис. 2.5).

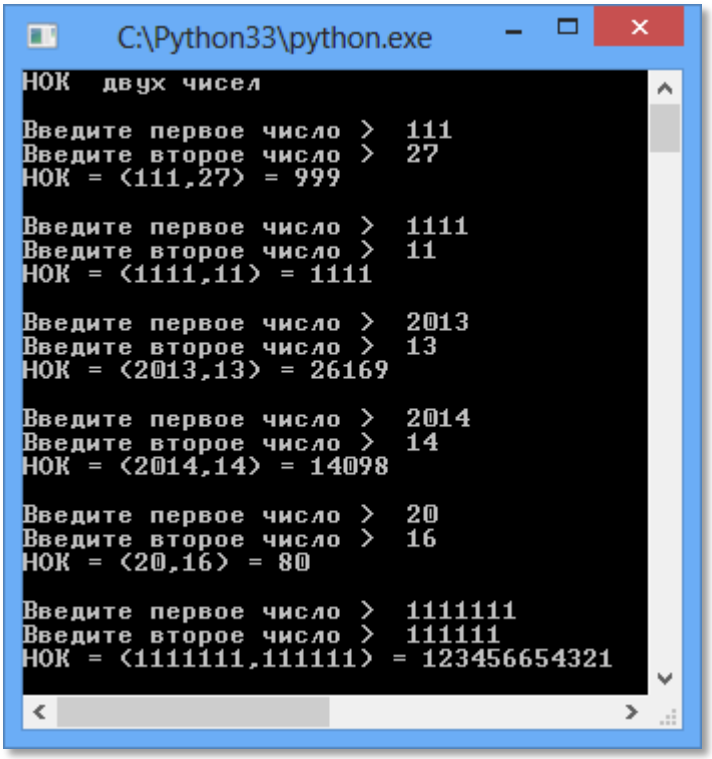


Рис. 2.5. Вычисляем *НОК*

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

Глава #3. Простые числа

Будьте проще – и к вам потянутся люди.

Кредо простых чисел

Простыми называются натуральные числа, имеющие в точности *два разных* делителя. Из этого определения следует, что ни нуль, ни единица к простым числам не относятся. Также очень легко установить, что первое простое число - это *двойка*, потому что она делится на единицу и на саму себя (двойка – единственное *чётное* простое число!). Дальше вы легко найдёте тройку, пятёрку, семёрку (кто посмелее, доберётся и до туза). Вам не составит труда продолжить этот ряд: 11, 13, 17, 23, 29, 31. Чтобы отбросить многие *составные* (то есть не простые) числа, достаточно воспользоваться *признаками делимости*. Но проверять большие числа таким способом «опасно», потому что легко ошибиться при делении (да и вообще делить большие числа затруднительно). На этот случай греческий математик *Эратосфен* (Рис. 3.1) придумал надёжный способ поиска простых чисел, который в его честь назвали **решетом Эратосфена**.

Рис. 3.1. Ἐρατοσθένης ὁ Κυρηναῖος (276 - 194 до н.э.)

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

Поиск простых чисел начинается с засыпки натуральных чисел в «решето», которое удобно представить в виде прямоугольной таблицы. *Наименьшее* число - это двойка, поскольку единица к простым числам не относится. А *наибольшее* – любое, по вашему выбору. Мы ограничимся первой сотней чисел (Рис. 3.2).

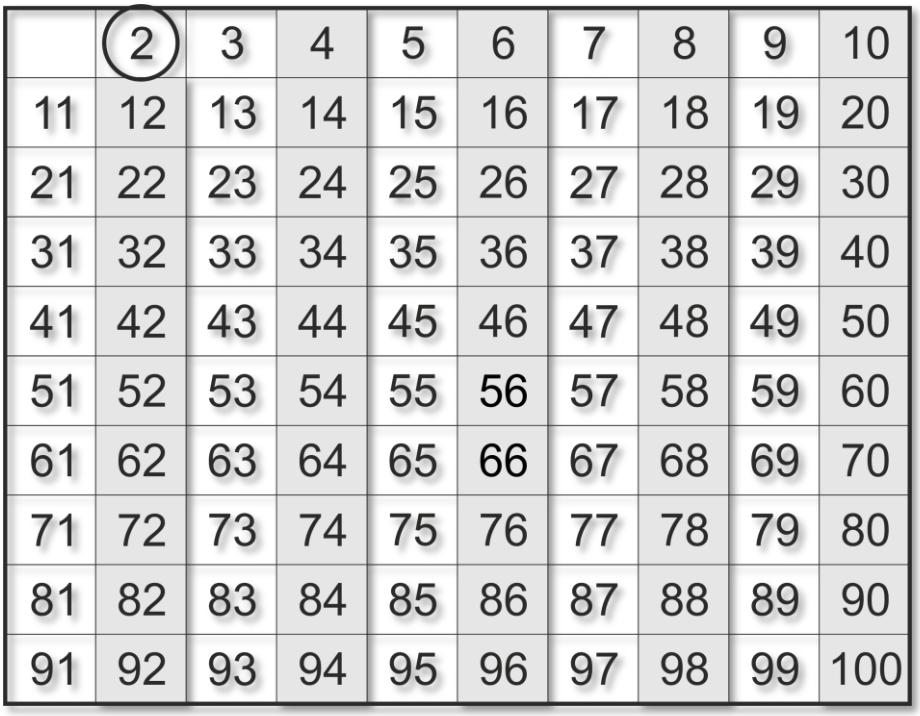
	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Рис. 3.2. Числа - в решете

Находим в таблице **двойку** – первое простое число – и обводим её кружком. Очевидно, что все последующие числа, кратные двойке, простыми быть не могут, поэтому мы их вычёркиваем. В данном примере мы будем просто *закрашивать* клетки с составными числами (Рис. 3.3).

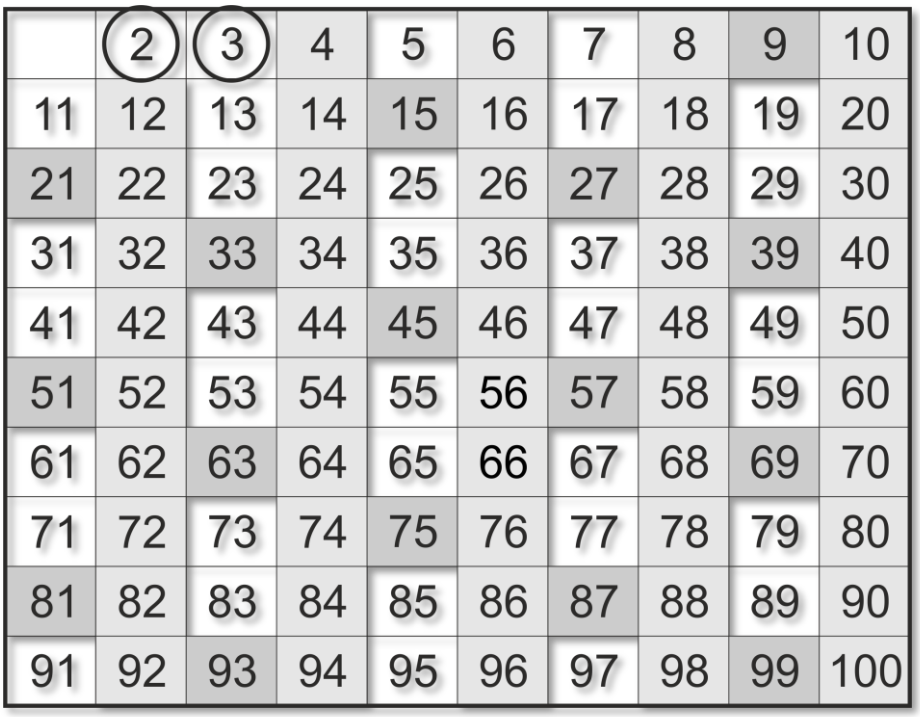
Как видите, уже после первого, грубого просеивания в решете осталась только половина чисел. Продвигаемся дальше по таблице и находим первое после двойки невычеркнутое число. Это **тройка** – второе простое число. Вычёркиваем все ещё не вычеркнутые числа, кратные тройке (Рис. 3.4).

Ну, а затем всё повторяется. Находим следующее простое число – **пятёрку** - и вычёркиваем числа, кратные пяти. Переходим к **семёрке**, затем к числам **11** и **13**. И так далее, пока не дойдём до конца таблицы. Числа в кружках - *простые*, все прочие – составные (Рис. 3.5).



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

Рис. 3.3. Вычеркнули чётные числа



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

Рис. 3.4. И кратные тройке

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

Рис. 3.5. Просеивание закончено!

Проект *Чудеса в решете Эратосфена*

Бесконечный цикл ***while***
Метод ***int***
Условный оператор ***if***
Оператор ***return***
Функция *с параметром*
Списки
Цикл ***for***
Цикл ***while***

Как вы убедились, просеивание чисел – занятие необременительное и даже весёлое, но всё-таки требует некоторого внимания и сноровки, поэтому лучше воспользоваться компьютером, тем более что алгоритм поиска простых чисел незамысловатый, и мы без труда переведём его на язык *Питон*.

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

```
# -*- coding: Windows-1251 -*-
#Решето Эратосфена
```

```
import math

#ГЛАВНАЯ ФУНКЦИЯ
def main():
    print('Решето Эратосфена')
    print()

    #бесконечный цикл ввода данных -
    #пока пользователь не закроет программу
    #или не введет 0:
    while True:
        s = "Введите конец  диапазона  >  "
        print(s, end = '')
        end = int(input())
        #если конец диапазона равен 0,
        #то программу закрываем:
        if (end == 0): return
        #конец диапазона не меньше двойки:
        if (end < 2): end = 2
        #ищем простые числа:
        Primes(end)
        print()
```

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

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

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

```
#ИЩЕМ ПРОСТЫЕ ЧИСЛА
def Primes(end):
    print()
    print("Простые числа в заданном диапазоне:")
```

Решето вполне естественно представить списком **number**. А дальше мы действуем по **алгоритму Эратосфена**:

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

```
#создаём список натуральных чисел 2..end:
number = list(range(2,end+1))
```

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

```
prime = 2
```

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

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

И так далее:

```
while (prime <= math.sqrt(end)):
    for i in range(prime * prime, end+1, prime):
        number[i-2] = 0
```

Конечно, мы не можем реально зачеркнуть число в списке, поэтому присваиваем соответствующему элементу списка значение **нуль**, которое будет означать, что это число *составное*. Из индекса элемента списка нужно вычесть двойку, поскольку нулевой индекс принадлежит этому числу!

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

```
    #ищем следующее простое число:
    prime += 1
    while (prime <= math.sqrt(end) and number[prime-2] == 0):
        prime += 1
```

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

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

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

```
    print()
    #печатаем простые числа:
    for i in number:
        if (i != 0):
            print(str(i) + " ", end='')
    print()
    print()
```

Поскольку в методе *main* действует бесконечный цикл *while*, то пользователь может и дальше «решетить» простые чисел (Рис. 3.6). Несмотря на «древность» алгоритма, он действует довольно быстро, но очень жаден до памяти компьютера.

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

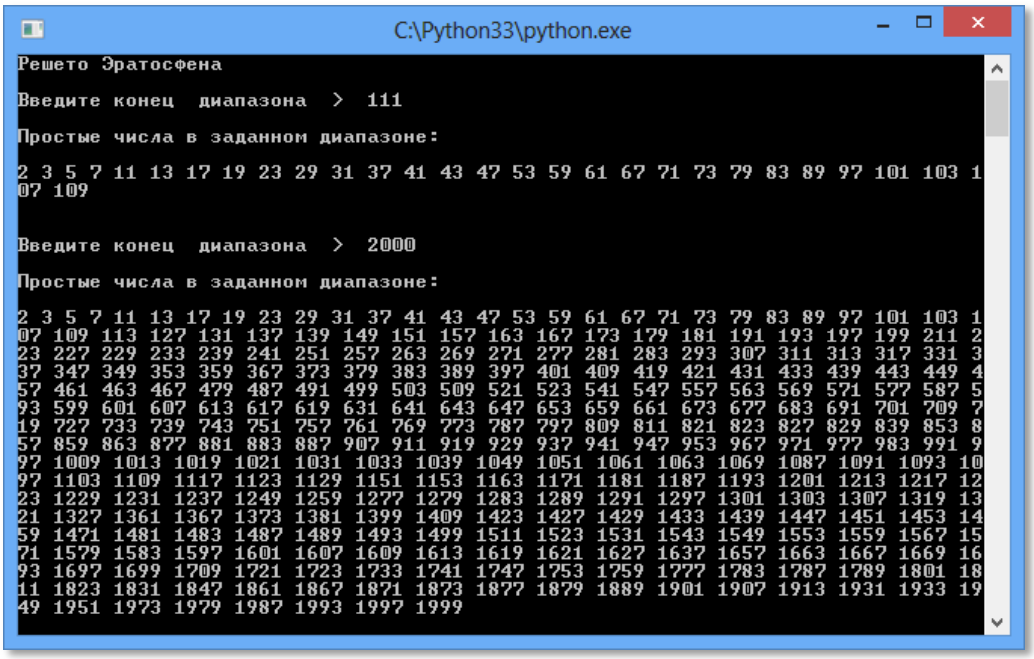


Рис. 3.6. Решето Эратосфена в действии!

Исходный код программы находится в папках **Решето Эратосфена**.

Проект *Простые числа*

Бесконечный цикл ***while***
Метод ***int***
Условный оператор ***if***
Оператор ***return***
Функция *с параметрами*
Вложенные циклы ***for***
Оператор целочисленного деления ***%***

Всем хорош алгоритм Эратосфена, но не годится для *больших* чисел - слишком неэкономно он расходует память компьютера. Мы, конечно, и с помощью решета легко узнаем, что 2011-й год - простой (естественно, только с математической точки зрения), а следующим простым годом будет 2017-й. Но вот с числами 514229 или 39916801 нам уже придётся повозиться! В таких случаях правильнее один раз написать программу, чем каждый раз считать вручную.

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

```
# -*- coding: Windows-1251 -*-
#ПРОГРАММА ДЛЯ ПОИСКА ПРОСТЫХ ЧИСЕЛ
#В ЗАДАННОМ ДИАПАЗОНЕ


import math

#ГЛАВНАЯ ФУНКЦИЯ
def main():
    print('Простые числа')
    print()

    #бесконечный цикл ввода данных -
    #пока пользователь не закроет программу
    #или не введет 0:
    while True:
        s = "Введите начало  диапазона  >  "
        print(s, end = '')
        begin = int(input())
        #если начало диапазона равно 0, то программу закрываем:
        if (begin == 0): return
        #начало диапазона не меньше двойки:
        if (begin < 2): begin = 2
```

```
        s = "Введите конец  диапазона   >  "
        print(s, end = '')
        end = int(input())

        #если конец диапазона равен 0,
        #то программу закрываем:
        if (end == 0): return
        #конец диапазона не меньше двойки:
        if (end < 2): end = 2
        #ищем простые числа:
        Primes(begin, end)
        print()
```

Функция *main* действует почти так же, как и «эратосфеновский», но вызывает функцию **Primes** с *двумя* параметрами:

```
#ИЩЕМ ПРОСТЫЕ ЧИСЛА
def Primes(n1, n2):
    print()
    print("Простые числа в заданном диапазоне:")

    #считаем простые числа:
    n = 0
    #создаем новый файл для записи результатов поиска:
    w = open('primes.txt', 'a')
    w.write("\nПростые числа:\n\n")
    #просматриваем все числа из заданного диапазона:
    for j in range(n1, n2+1):
        flg = True
        #проверяем, делится ли число j
        #на числа 2..корень квадратный из числа j:
        for i in range(2, round(math.sqrt(j))+1):
            #если делится, значит, число составное:
            if (j % i == 0):
                flg = False
                break

        #если нашли простое число,
        if (flg):
            n += 1
            #то печатаем его в консольном окне:
            print(str(j) + " ", end='')
            #и записываем в файл:
            w.write(str(j) + " ")

    w.write("\n")
    w.close()
```

```
    print()
    print("Всего " + str(n))
    print()
```

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

Проверка проходит так: очередное число *j* мы поочерёдно делим на все числа, начиная с двойки и кончая корнем квадратным из самого числа *j*. Так как любое натуральное число делится на единицу и само на себя, то два разных делителя у него имеются в любом случае (кроме, единицы, разумеется). Если мы обнаружим *хотя бы ещё один* делитель, то это будет перебор: число заведомо составное, и проводить испытания дальше нет смысла - мы переходим к проверке следующего числа. Если же число простое, то мы печатаем его на экране (Рис. 3.7) и записываем в файл.

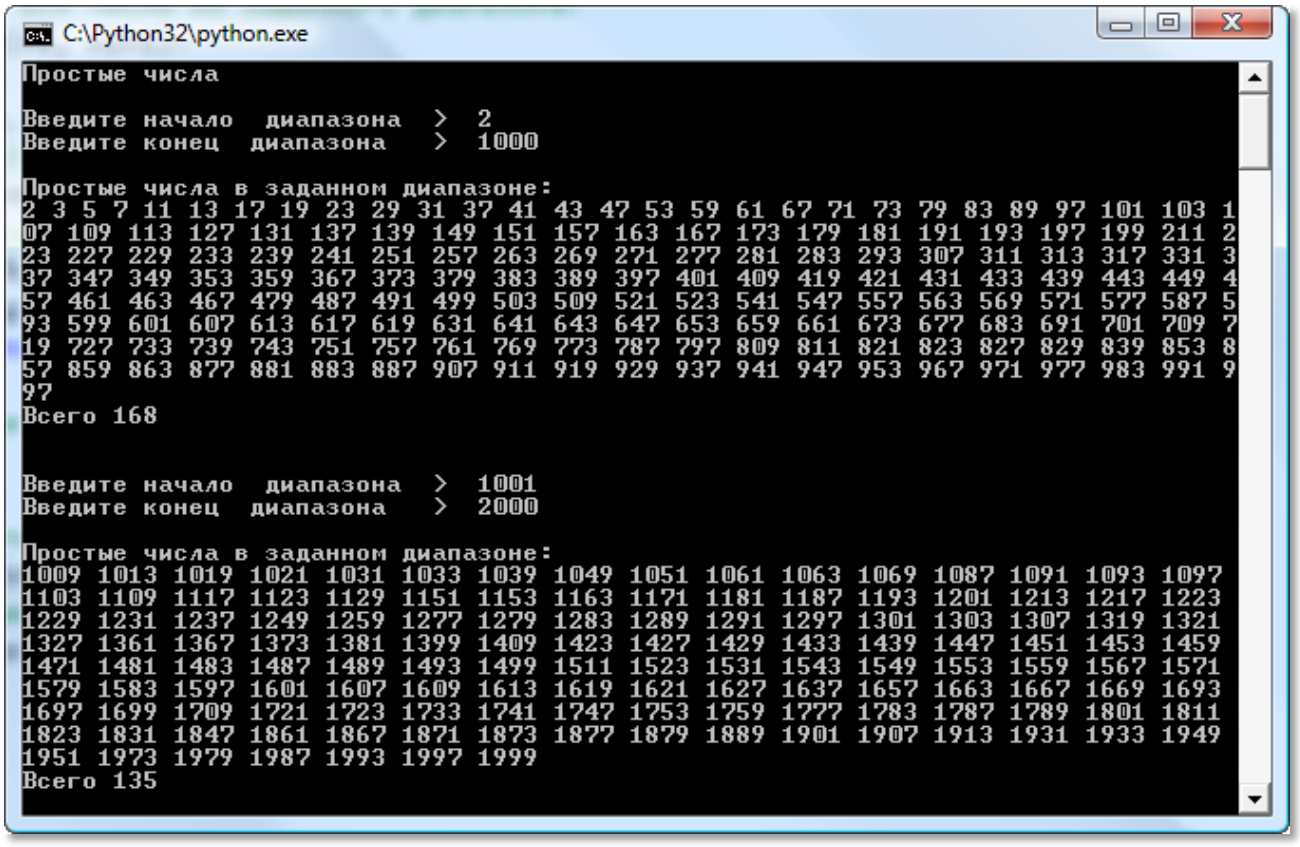


Рис. 3.7. Простые числа найдены!

Исходный код программы находится в папке **Простые числа**.

Взаимно простые числа

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

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

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

> **НОД(7,11) = 1**

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

> **НОД(10,9) = 1** → числа 10 и 9 - взаимно простые
>
> **НОД(10,8) = 2** → числа 10 и 8 – не взаимно простые: у них имеется общий делитель двойка.

Более подробно о взаимно простых числах вы можете прочитать в книге **[ОО80]**, на страницах 49-50.

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

Проект *Разложение числа на простые множители*

Бесконечный цикл ***while***
Метод ***int***
Условный оператор ***if***
Оператор ***return***
Функция *с параметром*
Вложенные ***циклы***
Цикл ***while***
Цикл ***for***
Оператор целочисленного деления ***%***

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

21 = 3 * 7

23 = 1 * 23

125 = 5 * 5 * 5

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

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

В этом проекте мы используем самый простой метод для разложения чисел – полный перебор всех чисел от двух до квадратного корня из заданного числа. Он очень простой, но довольно быстро справляется с задачей даже для очень больших чисел (Рис. 3.9). Однако при встрече с большими *простыми* числами может и спасовать.

```
# -*- coding: Windows-1251 -*-
#Факторизация
```

```
#ПРОГРАММА ДЛЯ РАЗЛОЖЕНИЯ НАТУРАЛЬНЫХ ЧИСЕЛ
#НА ПРОСТЫЕ МНОЖИТЕЛИ

import math

#ГЛАВНАЯ ФУНКЦИЯ
def main():
    print('Факторизация числа')
    print()

    #бесконечный цикл ввода данных -
    #пока пользователь не закроет программу
    #или не введет 0:
    while True:
        s = "Введите натуральное число (больше единицы) > "
        print(s, end = '')
        number = int(input())
        #если пользователь ввёл 0, то программу закрываем:
        if (number == 0): return
        #находим разложение:
        Razl(number)
```

В функцию **Razl** мы передаём испытуемое число, а само разложение числа даётся нам очень просто:

```
#Находим разложение заданного числа на простые множители
def Razl(num):
    print()
    print("Разложение числа:")

    print(str(num) + " = ", end='')
    #проверяем, делится ли num на числа 2..заданное число:
    for i in range(2, num+1):
        #если делится, выписываем делитель:
        while (num % i == 0):
              print(str(i), end='')
              num //= i
              if(num >= i): print(" * ", end='')
    print()
    print()
```

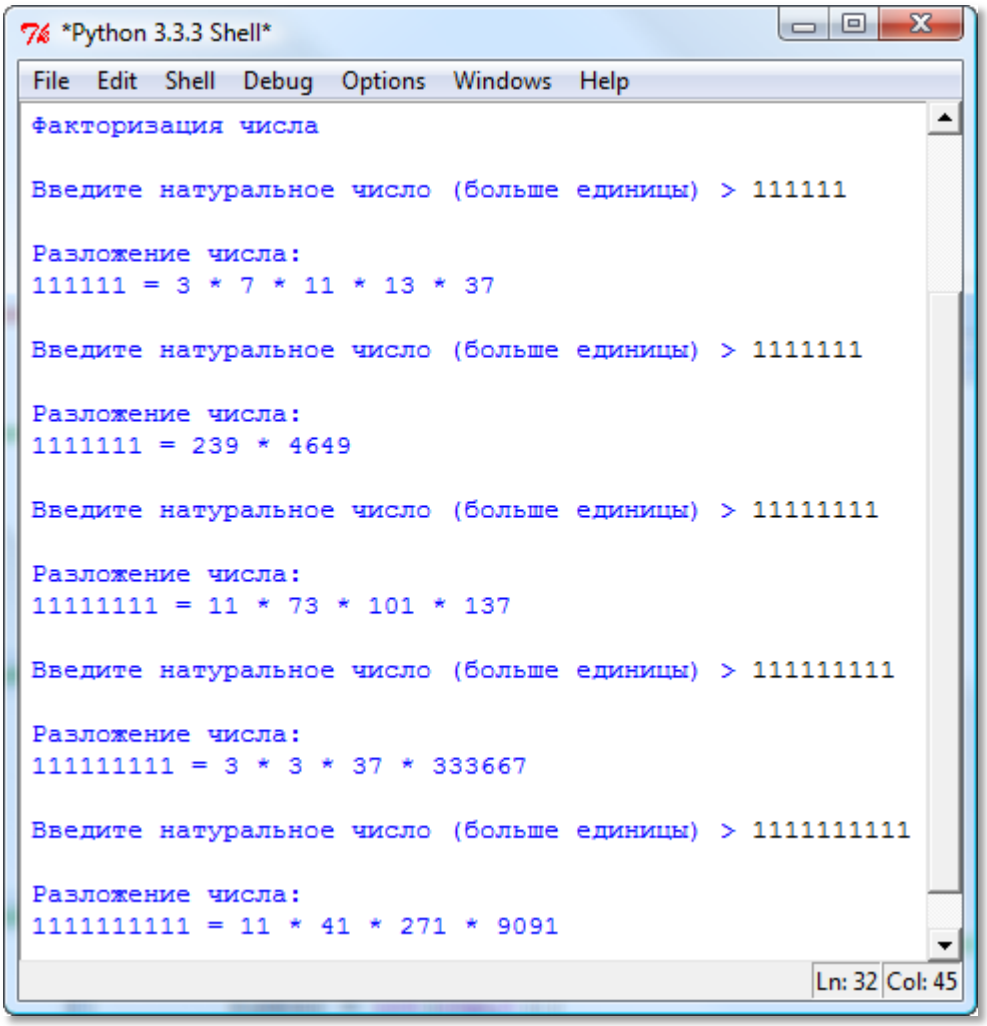


Рис. 3.9. Разлагаем на простые множители большие числа!

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

Задания для самостоятельного решения

Простые числа

1. Напишите метод, который определяет, являются ли несколько чисел **взаимно простыми**.

2. Найдите **четвёрки** простых чисел, принадлежащих одному десятку. Например, 11, 13, 17, 19. **[ЗП88**, Задача 558**]**.

3. Натуральное число называется **полусовершенным**, если оно равно сумме *всех* или *некоторых* своих делителей, исключая само число: 6, 12, 18, 20, 24, 28, 30, 36, 40. Например, 12 = 1 + 2 + 3 + 6 или 12 = 2 + 4 + 6.

Из этого определения следует, что всякое совершенное число является и полусовершенным, то есть полусовершенных чисел в заданном диапазоне не меньше, чем совершенных. Напишите программу, которая находила бы несколько полусовершенных чисел.

4. Два (различных) натуральных числа называются **дружественными**, если сумма всех делителей (исключая само число) первого числа равна второму числу, и наоборот. Первая пара дружественных чисел была найдена несколько тысячелетий тому назад. Это числа 220 и 284. Следующая пара отыскалась только в 1860 году – 1184 и 1210. Сейчас известно несколько миллионов дружественных чисел. Напишите программу для их поиска **[ЗП88**, Задача 560**]**. Учитывайте, что числа в паре либо оба *чётные*, либо оба *нечётные*.

Глава #4. Числовые ребусы

Первый числовой ребус составил *Генри Дьюдени*:

SEND + MORE = MONEY

В числовых ребусах словами зашифрованы **числа**, причём одинаковые буквы обозначают одинаковые цифры, и наоборот, одинаковые цифры обозначены одинаковыми буквами.

Задача состоит в том, чтобы заменить буквы соответствующими им цифрами так, чтобы получилось верное равенство.

Обычно числовые ребусы решаются с помощью *логических* рассуждений, но практически никогда не удаётся обойтись без предположений, то есть без перебора вариантов. А с такой работой даже самый захудалый компьютер справится быстрее любого из нас. А от вас требуется только одно – написать простую программу.

Как решать числобус Генри Дьюдени, я уже много раз рассказывал в своих книгах, поэтому мы отдадим должное другим криптарифмам.

Проект *Каковы жуки?*

***Функция** без параметров*
*Вложенные циклы **for***
*Условный оператор **if***
*Оператор **continue***
*Оператор **return***
*Оператор **or***

Задача 9 из книги *Удивительный мир чисел* **[КА86]**, страница 100:

> **Решите уравнение** в целых неотрицательных числах:
>
> 520•(Ж•У•К•И + Ж•У + Ж•И + К•И + 1) = 577•(У•К•И + У + И).

Числовые ребусы вполне успешно можно решать **методом полного перебора**. В них каждая цифра заменена буквой, но, поскольку цифр только десять, то каждая буква может принимать всего 10 разных значений - от 0 до 9. В любом ребусе не более десятка разных букв, то есть в худшем случае нам придётся проверить 10 000 000 000 вариантов. Современным компьютерам это вполне по силам. Впрочем, так бездумно компьютер не используют, поэтому даже при полном переборе следует разумно ограничивать число вариантов. Обычно сделать это очень просто, поскольку некоторые способы решения криптарифмов лежат на поверхности и не требуют глубоких размышлений. Например, для уменьшения числа вариантов достаточно учесть тот очевидный факт, что все буквы должны иметь **разное** значение. Это следует из условия самой головоломки: *разным буквам соответствуют разные цифры*. Вот теперь можно смело браться за любой числовой ребус.

Функция **main** просто вызывает функцию *Solve*, а затем печатает число найденных решений:

```
# -*- coding: Windows-1251 -*-
#Кордемский, с.100, Задача 09

#ГЛАВНАЯ ФУНКЦИЯ
def main():
    print("Каковы жуки?")
    print()
    nVar = Solve()
```

```
    print("Найдены все варианты решения - " + str(nVar))
    print()
```

Функция **Solve**, как это обычно и бывает при полном переборе, очень простая. Записываем столько вложенных циклов *for,* сколько разных букв в ребусе. В нашем примере всего 4 разные буквы, поэтому и циклов тоже будет 4. Начиная со второй буквы, делаем проверку: её цифровое представление не должно совпадать с предыдущими буквами. Найдя значение каждой буквы, проверяем **условие**:

```
520*(Ж*У*К*И + Ж*У + Ж*И + К*И + 1) == 577*(У*К*И + У + И)
```

Если оно выполняется (то есть левая часть выражения *равна* правой), то мы выписываем найденное решение в консольном окне, а затем ищем другие решения:

```
#РЕШАЕМ ЗАДАЧУ
def Solve():
    result = 0
    for Ж in range(0, 9+1):
        for У in range(0, 9+1):
            if (У == Ж): continue
            for К in range(0, 9+1):
                if (К == У or К == Ж): continue
                for И in range(0, 9+1):
                    if (И == К or И == У or И== Ж): continue
                    if (520*(Ж*У*К*И + Ж*У + Ж*И + К*И + 1) ==
577*(У*К*И + У + И)):
                        result += 1
                        print("Вариант # " + str(result))
                        s = "Ж = " + str(Ж)
                        print(s)
                        s = "У = " + str(У)
                        print(s)
                        s = "К = " + str(К)
                        print(s)
                        s = "И = " + str(И);
                        print(s)
                        print()

    return result
```

В итоге мы найдём *единственное* решение этого ребуса (Рис. 4.1).

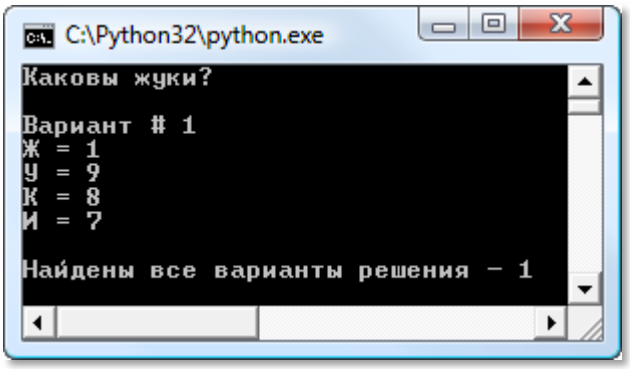


Рис. 4.1. Задача решена!

Точно так же вы можете решить любой числобус, НО – каждый раз вам придётся писать **новую** функцию *Solve*! Такова плата за простоту программы…

Исходный код программы находится в папке **Кордемский 100 09**.

Задания для самостоятельного решения

Поиграем в прятки

***Удивительный мир чисел*. Задача 8 (9)], страница 73**

Все 10 цифр спрятались под буквами в каждом из пяти равенств:

1) Д • Г У В А = Б Ж Я И Е
2) Е • Д А И Г = Б • У В Ж Я
3) Е Ж • Г В А = Б У Я Д И
4) ЖИ • ДАГ = ЕУВБЯ
5) УА • ВБГ = ИЯ • ЖДА

Задача несложная, но трудоёмкая: чтобы решить её, придётся выписывать 10 вложенных циклов!

ЛОБ ТРИ САМ

***Удивительный мир чисел*. Задача 2, страница 70**

Это трёхзначные числа, такие, что

ЛОБ +ТРИ = САМ

Расшифруйте сложение, обходясь без цифры нуль.

В любом возможном решении должна обнаружиться определенная закономерность в числе «САМ». Какая?

Семейство, спрятавшееся в «БАКУ»

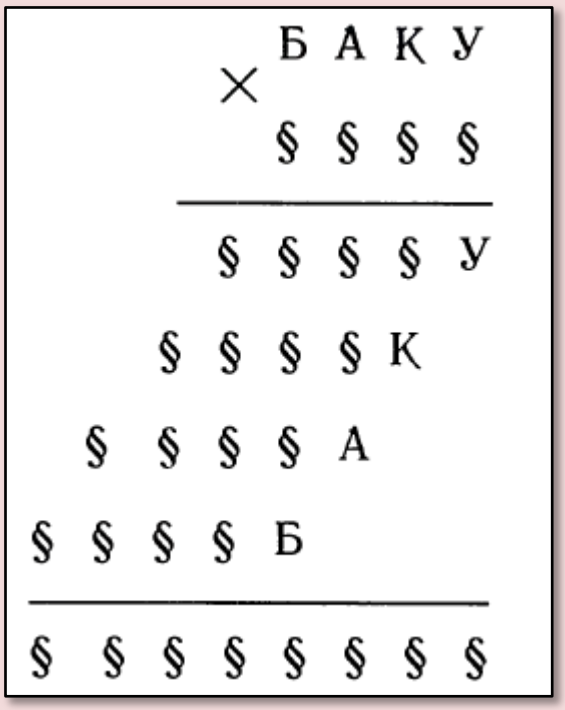
Удивительный мир чисел. .Задача 5, страница 77

а) Расшифруйте произведение чисел **БАКУ** и **§§§§**, зная, что в каждом сомножителе цифры образуют (слева направо) строго убывающую последовательность.

Какое семейство цифр спряталось за буквами Б, А, К, У?

б) Расшифруйте в этом ребусе ещё и второй множитель (**§§§§**), если строго убывающую последовательность образуют цифры только перво-

го сомножителя (**БАКУ**) и если в произведении вторая цифра слева - нуль.



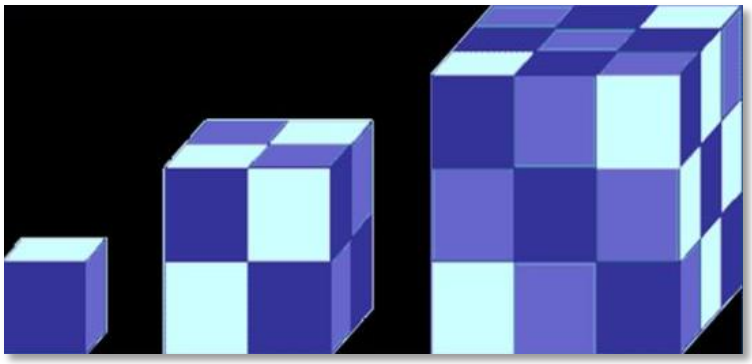
Глава #5. Степени и корни

Зри в корень!

Козьма Прутков

Из множества занимательных и поучительных задач со степенями и корнями мы решим всего 4^2-$\sqrt{9}$, но зато очень интересных!

Проект *Кубическое число*



***Функция** с параметром*
*Функция **round***
*Метод **math.pow***
*Цикл **for***
*Условный оператор **if***
*Оператор **break***

Задача 1 из книги *Удивительный мир чисел* **[КА86]**, страница 89:

> **Найдите число**, куб которого - данное число:
>
> 1) **M** = 996 703 628 669;
>
> 2) **N** = 1 011 443 374 872.

Самый простой способ – извлечь из заданных чисел кубический корень.

При этом мы должны учесть, что метод **pow** не всегда возвращает точное значение, хотя по смыслу задачи оба корня должны быть целыми числами. В этом случае достаточно округлить корень до ближайшего целого числа встроенной функцией *round*:

```
# -*- coding: Windows-1251 -*-
#Кордемский, с.89, Задача 1

import math

#ГЛАВНАЯ ФУНКЦИЯ
def main():
    print('Кубическое число')
```

```
    print()
    Solve(996703628669)
    Solve(1011443374872)
    print()

#РЕШАЕМ ЗАДАЧУ
def Solve(num):
    print("Кубическое число = " + str(round(math.pow(num,1/3.0))))
    print()
```

На Рис. 5.1 показано решение задачи:

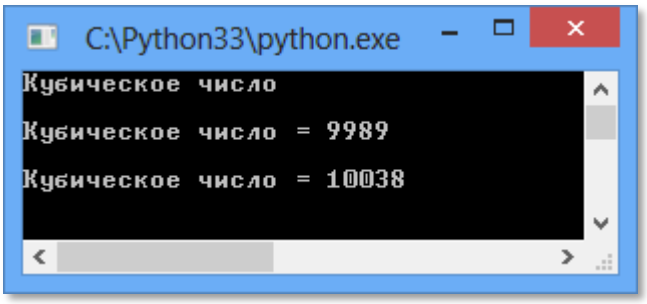


Рис. 5.1. Извлекли корни

Поскольку кубические корни даже из очень больших чисел невелики, то можно найти их простым перебором, даже начиная с нуля!

Пишем вторую версию функции для решения задачи:

```
def Solve2(num):
    i = 0
    while True:
        if (i*i*i > num): break
        if (i*i*i == num):
            print("Кубический корень из числа %i = %i" %(num, i))
            break
        i += 1
    print()
```

И получаем точные значения искомых корней (Рис. 5.2).

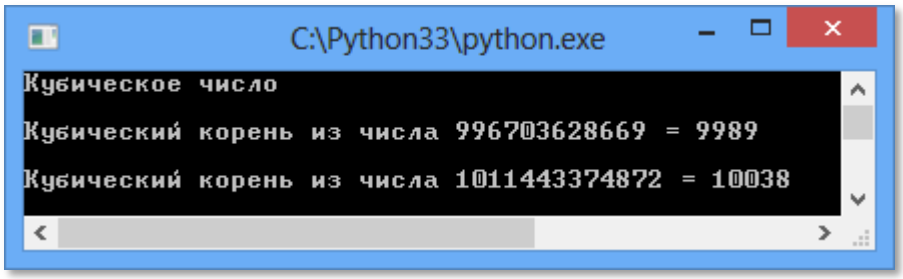


Рис. 5.2. И перебор не в тягость!

При этом нам не понадобился метод *pow* класса *math*!

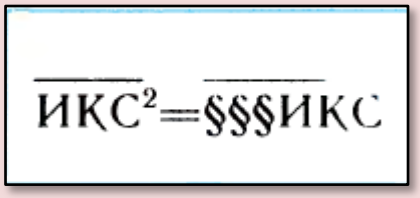
Исходный код программы находится в папке **Кордемский 089 01**.

Задания для самостоятельного решения

Задача-ребус

***Удивительный мир чисел*, страницы 69-70**

Представим задачу в форме такого **ребуса**:



Эта задача практически ничем не отличается от той, что мы решили в проекте *Девять в квадрате.*

Задача #33

Математическая шкатулка

Какую последнюю цифру может иметь квадрат натурального числа? Куб его? Четвёртая степень?

Задача #34

Математическая шкатулка

Могут ли числа 458, 523, 652 быть квадратами или кубами целого числа?

Глава #6. Числовые ряды и другие задачи

Без вычисления факториалов и чисел Фибоначчи не обходится, пожалуй, ни одна книга по программированию. А всё потому, что вычислять их легко и весело!

Проект *Факториал*

Бесконечный цикл ***while***
Метод ***int***
Функция *с параметром*
Условный оператор ***if***
Оператор ***return***
Цикл ***for***

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

n! (читается: *эн-факториал*)

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

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

По определению, 0! = 1.

Формула очень простая, но нетрудно догадаться, что для больших значений *n* факториал будет выражаться огромным числом.

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

В функции **main** пользователь вводит число, после чего вызывается функция *factorial*, которая возвращает вычисленное значение факториала:

```
# -*- coding: Windows-1251 -*-
#ПРОГРАММА ДЛЯ ВЫЧИСЛЕНИЯ ФАКТОРИАЛОВ

#ГЛАВНАЯ ФУНКЦИЯ
def main():
    print('Факториал')
    print()

    #бесконечный цикл ввода данных -
    #пока пользователь не закроет программу
    #или не введет отрицательное число:
    while True:
        s = "Введите число  > "
        print(s, end = '')
        num = int(input())
        #если пользователь ввёл отрицательное число,
        #то программу закрываем:
        if (num < 0): return
        #находим факториал заданного числа:
        fact = factorial(num)
        #печатаем факториал:
        print(str(num) + "! = " + str(fact))
        print()
```

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

```
#ВЫЧИСЛЯЕМ ФАКТОРИАЛ ЗАДАННОГО ЧИСЛА
def factorial(num):
    if (num == 0): return 1
    fact = num
    for i in range(2, num):
        fact *= i
    return fact
```

> Умножать на единицу, конечно, смысла нет.

Здесь мы учитываем, что факториал **нуля** равен *единице*, это особый случай и его нужно учесть отдельно (Рис. 6.1)!

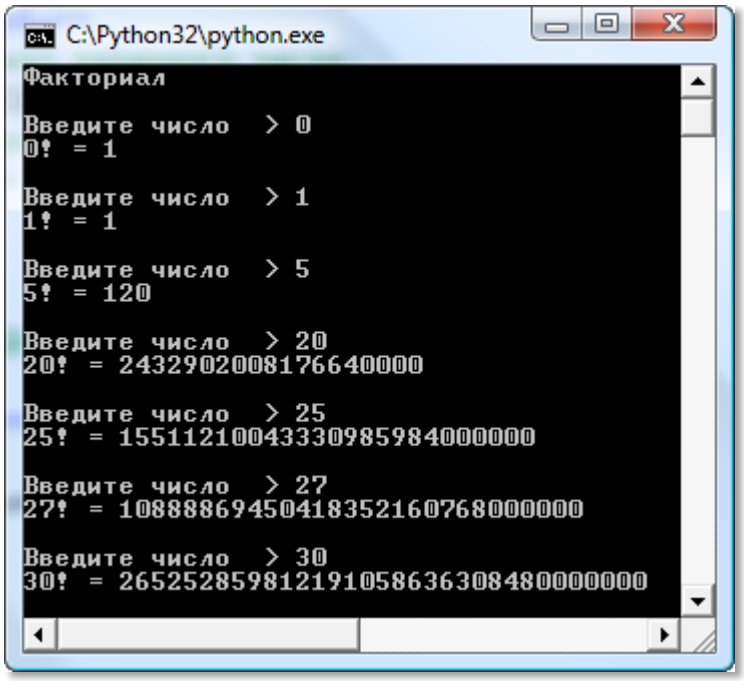
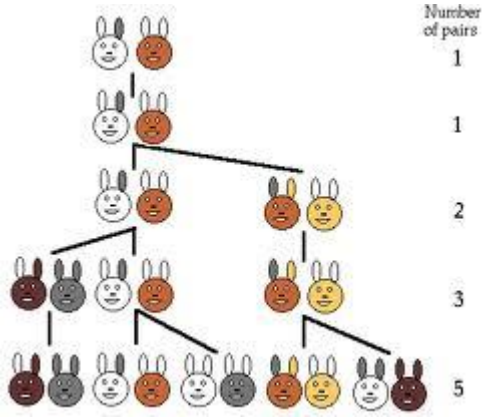


Рис. 6.1. Маленькие **и** большие факториалы

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

Проект *Числа Фибоначчи*

Бесконечный цикл ***while***
Метод ***int***
Списки
Условный оператор ***if***
Цикл ***for***
Оператор ***or***
Оператор ***return***



Числа Фибоначчи, как нетрудно догадаться, открыл *Фибоначчи* (он же *Леонардо Пизанский*), средневековый математик (Рис. 6.3), автор *Книги абака (Liber Abaci)*, которую он написал в 1202 году.

Рис. 6.3. Леонардо Пизанский (Leonardo Pisano, 1170 - 1250), по прозвищу Фибоначчи (Fibonacci)

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

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

кролик не умрёт. Числа Фибоначчи как раз и отражают рост популяции кроликов.

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

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

Все последующие равны сумме двух предыдущих, то есть:

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

и так далее, до бесконечности.

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

В функции **main** пользователь вводит любое неотрицательное число, после чего функция *Fibo* находит числа Фибоначчи от нулевого до заданного.:

```
# -*- coding: Windows-1251 -*-
#ПРОГРАММА ДЛЯ ВЫЧИСЛЕНИЯ ЧИСЕЛ ФИБОНАЧЧИ

#ГЛАВНАЯ ФУНКЦИЯ
def main():
    print('Числа Фибоначчи')
    print()

    #бесконечный цикл ввода данных -
    #пока пользователь не закроет программу
    #или не введет отрицательное число:
    while True:
        s = "Введите число  > "
        print(s, end = '')
        num = int(input())
```

```
        #если пользователь ввёл отрицательное число,
        #то программу закрываем:
        if (num < 0): return
        #находим все числа Фибоначчи от 0 до num:
        fibo = Fibo(num)
        #и печатаем их:
        for i in range(0, num+1):
            print("Число  Фибоначчи " + str(i) + " = " + str(fibo[i]))
        print()
```

Чтобы сохранить все промежуточные числа, мы создадим список чисел **f** и сразу же поместим в него три первых числа Фибоначчи, чтобы можно было найти следующие. Далее мы передаём функции **Fibo** номер числа Фибоначчи, которое желает узнать пользователь. Сам алгоритм следует определению чисел Фибоначчи:

```
#ВЫЧИСЛЯЕМ ЗАДАННОЕ ЧИСЛО ФИБОНАЧЧИ
def Fibo(n):
    f = []
    #помещаем в список первые числа Фибоначчи:
    f.append(0)
    f.append(1)
    f.append(1)
    if (n == 1 or n == 2): return n
    for i in range(3, n+1):
        f.append(f[i - 1] + f[i - 2])
    return f
```

В итоге мы получаем список *f*, до краёв наполненный числами Фибоначчи, которые и распечатываем в функции *main*. Последнее из найденных чисел соответствует заданному пользователем номеру числа Фибоначчи. Из этого следует, что если пользователю нужно только оно, то можно возвращать пользователю только последнее из найденных чисел.

Наша уловка со списком позволяет практически мгновенно находить совершенно невероятные числа Фибоначчи (Рис. 6.4)!

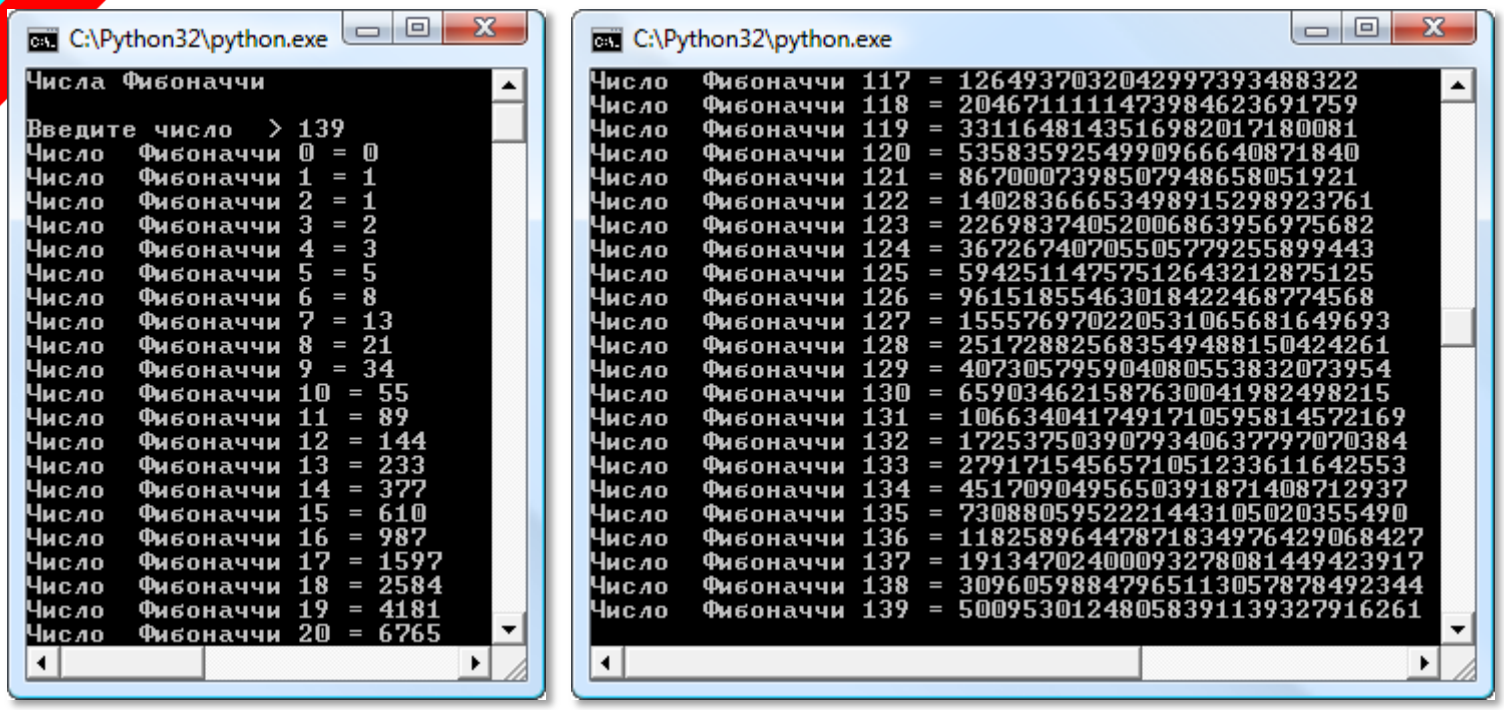


Рис. 6.4. Вычисляем мгновенно!

Исходный код программы находится в папке **Числа Фибоначчи**.

Задания для самостоятельного решения

Задача #297

Математическая шкатулка

Чтобы пронумеровать страницы некоторой книги, понадобилось 1164 цифры.

Сколько в этой книге страниц?

Ответ: 424

Задача #298

Математическая шкатулка

Сколько цифр нужно употребить для нумерации книги, в которой 634 страницы?

Ответ: 1794

Формула Бине

[ЗП88]. Задача 556

n-ное число Фибоначчи можно вычислить по **формуле Бине**:

$$F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}} = \frac{\varphi^n - (-\varphi)^{-n}}{\varphi - (-\varphi)^{-1}} = \frac{\varphi^n - (-\varphi)^{-n}}{2\varphi - 1}$$

В ней буквой *фи* обозначено *золотое сечение*: $\varphi = \frac{1+\sqrt{5}}{2}$.

Из неё следует, что F_n равно ближайшему к $\frac{\varphi^n}{\sqrt{5}}$ целому числу.

Так как абсолютная величина выражения $\frac{1-\sqrt{5}}{2}$ меньше единицы, то при больших *n* числа Фибоначчи можно вычислять по *приближённой* формуле:

$$F_n \sim \frac{\varphi^n}{\sqrt{5}}$$

Напишите метод, вычисляющий заданное число Фибоначчи по формуле Бине.

Глава #7. Диофантовы уравнения и линейное программирование

Под **диофантовыми** понимаются неопределённые уравнения, которые решаются в целых числах (часто – в натуральных). Они названы в честь древнегреческого математика Диофанта, жившего в третьем веке нашей эры (Рис. 7.1).

В честь Диофанта назван и кратер на Луне (на рисунке – в правом нижнем углу).

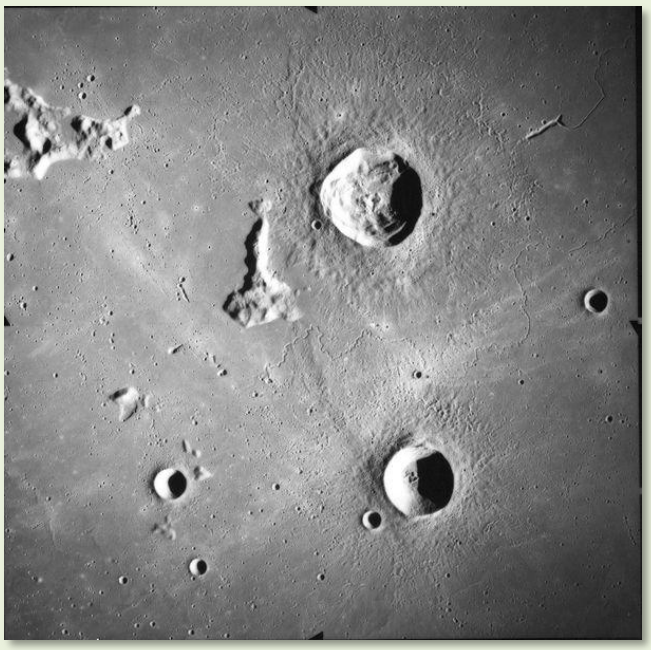


Рис. 7.1. Диофант Александриийский
Διόφαντος ὁ Ἀλεξανδρεύς, Diophantus

В тринадцатитомном труде ***Арифметика*** (до нас дошли только 6 первых книг) он показывает, как решать подобные уравнения.

В 1974 году была издана книга ***Арифметика и книга о многоугольных числах***, содержащая перевод на русский язык всех трудов Диофанта (Рис. 7.2, слева). В 2007 году книга была переиздана (Рис. 7.2, справа).

Рис. 7.2. Диофантовы книги

Более подробно о Диофанте вы можете прочитать в книге Якова Перельмана *Занимательная математика* (Рис. 7.3, слева) и Изабеллы Башмаковой (Рис. 7.4, справа).

Рис. 7.3. И книги о Диофанте

Впрочем, диофантовы уравнения появились задолго до самого Диофанта. Первое из таких уравнений было известно ещё в Древнем Вавилоне:

$$x^2 + y^2 = z^2$$

Оно было решено пифагорейцами.

Второе уравнение:

$$x^2 - ay^2 = 1$$

решил в целых числах Евклид.

О способах решения диофантовых уравнений можно узнать в книге Башмаковой, а также в ***Справочном пособии к решению задач: Диофантовы уравнения*** Дмитрия Базылева (Рис. 7.4).

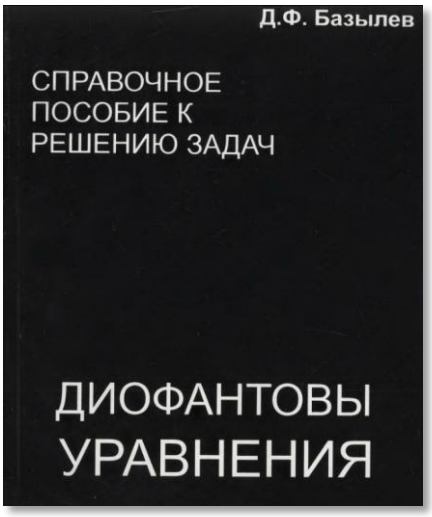


Рис. 7.4. Справочник по диофантовым уравнениям

Самая известная занимательная задача, связанная с диофантовыми уравнениями, это, безусловно, **Кролики и фазаны**.

Проект *На ферме*

***Функции** без параметров*
*Вложенные циклы **for***
*Условный оператор **if***
*Оператор **continue***

Моисеенко и Данилец – это не только кролики, но и фазаны!

Диофант

Задача 12 (13) из книги *Удивительный мир чисел* **[КА86]**, страница 53:

На ферме выращивают кроликов и фазанов. В настоящее время их столько, что у всех вместе 740 голов и 1980 ног.

Сколько же в настоящее время находится на ферме кроликов и фазанов?

```
# -*- coding: Windows-1251 -*-
#Кордемский, с.53, Задача 12

#ГЛАВНАЯ ФУНКЦИЯ
def main():
    print('На ферме')
    print()
    Solve()
    print()
```

Поскольку и число голов, и число ног у кроликов и фазанов выражается **целыми** числами, то достаточно перебрать все варианты распределения 740 голов по кроликам и фазанам:

```
#РЕШАЕМ ЗАДАЧУ
def Solve():
    GOLOVY = 740
    NOGI = 1980
```

```
    for kroliki in range(0, GOLOVY+1):
        for fazany in range(0, GOLOVY+1):
            if (kroliki + fazany != GOLOVY):
                continue
```

При этом мы должны учитывать, что у кроликов по 4 ноги, а у фазанов – только по 2:

```
            if (kroliki*4 + fazany*2 != NOGI):
                continue
            print("Кроликов = %i  \nФазанов = %i" % (kroliki, fazany))

    print()
```

И вмиг кролики и фазаны пересчитаны и занесены в консольное окно нашей программы (Рис. 7.5).

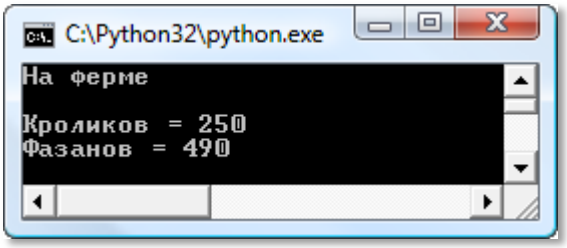


Рис. 7.5. Пересчитали поголовье и поножье

А судя по числу ног, Моисеенко и Данилец всё-таки фазаны, а не кролики!

Исходный код программы находится в папке **Кордемский 053 12**.

Задания для самостоятельного решения

Задача 11, страница 52

Удивительный мир чисел

Овчарка погналась за лисой, когда между ними было расстояние 99 м. Скачок лисы 1,1 м, скачок овчарки 2,2 м. Когда овчарка делает 19 скачков, лиса делает 29 скачков.

Сколько метров проскачут они, пока овчарка догонит лису?

Глава #9. Занимательная комбинаторика и теория вероятностей

Комбинаторика - это раздел математики, который изучает *множества* (совокупности, наборы) каких-либо элементов. Первая книга по комбинаторике вышла в 1666 году под названием *Рассуждения о комбинаторном искусстве*. Её написал известный немецкий математик Готфрид Вильгельм фон Лейбниц, который и придумал название *комбинаторика* этому разделу математики.

Как в жизни, так и в программировании *комбинаторные задачи* встречаются очень часто. Например, сколько различных слов можно составить из букв русского алфавита? Сколько существует различных комбинаций при игре в кости двумя или тремя кубиками? Сколько разных нарядов можно составить из трех юбок и четырёх блузок и так далее.

Все комбинаторные задачи решаются с помощью **комбинаторных конфигураций**: *размещений, перестановок, сочетаний, композиций и разбиений*.

А начнём мы наши комбинаторные забавы с **перестановок** элементов. Возьмём множество, состоящее из трёх разных элементов, например, русских букв - *{К, О, Т}*. Всякое «слово», составленное из всех этих букв без повторений, и называется перестановкой элементов множества.

Поскольку в множестве элементы не упорядочены, то мы выпишем их сначала в произвольном порядке, например так:

1. **КОТ**

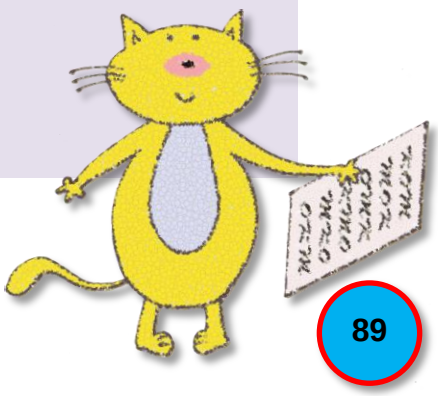
Вот мы и получили первую перестановку, а значит, и первое слово – *КОТ*. Оно «случайно» совпало с настоящим русским словом. Чтобы найти вторую перестановку, поменяем местами вторую и третью буквы:

2. **КТО**

Тоже получилось неплохо, ведь *кто* - это русское местоимение.

Давайте поменяем теперь первую и вторую буквы:

3. **ТКО**



Такого слова нет (в старой речи была частица *–тко*, имеющая тот же смысл, что и современная *-ка*: *бери-тко, читай-тко*), а вот четвёртая перестановка снова удачная. Чтобы её получить, поменяем местами вторую и третью буквы:

4. ТОК

Снова меняем первую и вторую буквы - и получаем пятую перестановку:

5. ОТК

Не ахти какое слово, но *Отдел технического контроля* тоже сгодится.

И последнюю перестановку мы найдём, переставив две последние буквы:

6. ОКТ

ОКТ – тоже сокращение от *Оптическая когерентная томография*, так что все наши перестановки из трёх букв *К, О, Т* оказались не совсем бессмысленными. Конечно, с другими буквами результат был бы другим.

Но почему мы можем утверждать, что нашли *все* перестановки множества из трёх элементов? – Давайте рассуждать логически. На первом месте в слове может стоять любая из трёх букв. На второе место можно поставить любую из двух оставшихся, а для последнего места останется только одна буква. Таким образом, всего можно составить 3 х 2 х 1 = 6 разных слов. Но мы ровно столько и составили, значит, других слов из этих букв составить нельзя (естественно, мы используем каждую букву только один раз!).

Если взять множество из четырёх разных элементов, то, рассуждая аналогично, мы придём к выводу, что из них можно составить 4 х 3 х 2 х 1 = 24 разных слова. Этот ряд легко продолжить сколь угодно далеко, а произведение чисел от единицы до заданного называется факториалом. Как его вычислять, вы уже знаете.

Проект *Генерируем перестановки*

Бесконечный цикл ***while***
Метод ***int***
Условный оператор ***if***
Оператор ***return***
Списки
Цикл ***for***
Функция *с параметрами*
Условный оператор ***if-else***
Рекурсивная *функция*

Мы научились подсчитывать *общее число* перестановок. Ну, пусть мы знаем, что из трёх разных букв можно составить 3! разных слов, но вопрос в том, *как* получить эти перестановки? С множеством из трёх элементов мы легко справились, а если взять больше – четыре, пять, а то и восемь?

Оказывается, мы не первые, кто задался этим вопросом. Ещё в семнадцатом веке английские звонари научились выбивать на нескольких разных колоколах «мелодии», состоящие из всех перестановок этих колоколов. Например, для трёх колоколов нужно было сыграть такую мелодию:

> Первый колокол – второй – третий.
> Второй колокол – первый – третий.

Дальше вы и сами продолжите эту музыку.

Колоколов, конечно, было больше, а последовательность колокольных ударов нужно было держать в голове. Например, в *Книге рекордов Гиннеса* рассказывается о том, что в 1963 году за 17 с лишним часов удалось выбить на восьми колоколах все 8! = 40320 перестановок. В семнадцатом веке, конечно, эти музыкально-комбинаторные экзерсисы были короче, но, тем не менее, запомнить многие сотни перестановок было совсем непросто, поэтому звонари придумывали свои способы для «генерирования» всех колокольных перестановок. Один из таких способов мы и положим в основу компьютерной программы, которая быстро и правильно выпишет все перестановки элементов заданного множества.

Нам вполне достаточно одной переменной **MAX_ELEM** для ограничения числа элементов в множестве, иначе их печать на экране займёт очень много времени; одной целой переменной **nPerm** – для хранения числа пе-

рестановок и целочисленного списка **a** – для хранения собственно перестановки:

```
# -*- coding: Windows-1251 -*-
#Рекурсивная генерация перестановок


#макс. число элементов в
#множестве:
MAX_ELEM = 8
#число перестановок:
nPerm = 0
```

В функции **main** пользователь задаёт число элементов множества в диапазоне *1.. MAX_ELEM*, после чего мы создаём список **a** и заполняем его числами *1.. MAX_ELEM*. Так мы получаем **первую** перестановку:

1 2 3 4 ... MAX_ELEM

Нам гораздо удобнее переставлять именно **числа**, а не элементы других типов. Однако вы можете понимать под этими числами индексы каких-либо элементов в списке любого типа, так что наш генератор получится вполне универсальным.

```
#ГЛАВНАЯ ФУНКЦИЯ
def main():
    print('Генерируем перестановки')
    print()

    global nPerm
    #бесконечный цикл ввода данных -
    #пока пользователь не закроет программу
    #или не введет 0:
    while True:
        s = "Число элементов (1.." + str(MAX_ELEM) +") >  "
        print(s, end = '')
        #число элементов:
        nElem = int(input())
        #если пользователь ввёл 0,
        #то программу закрываем:
        if (nElem == 0): return

        #число перестановок:
```

```
        nPerm = 0
        #список для хранения очередной перестановки -->
        #начальная перестановка:
        a = list(range(1, nElem+1))
```

Генерирование всех остальных перестановок, а также их печать происходит в рекурсивной функции **PermutationRec**, которой мы передаём три аргумента:

1. всегда **0**
2. число элементов в списке **nElem**
3. список **a**

```
        nPerm = 0
        #генерируем перестановки:
        PermutationRec(0, nElem, a)

        print("Число перестановок = " + str(nPerm))
        print()
```

В функции *PermutationRec* мы переставляем элементы списка до тех пор, пока не получим новую перестановку при *k == n*, после чего печатаем её на экране и продолжаем генерировать новые перестановки:

```
#ГЕНЕРИРУЕМ ПЕРЕСТАНОВКИ
def PermutationRec(k, n, a):
    global nPerm
    #нашли очередную перестановку:
    if (k == n):
        nPerm += 1
        #печатаем её:
        print(*a, sep='', end='')
        print()
    else:
        for i in range(k, len(a)):
            a[k], a[i] = a[i], a[k]
            PermutationRec(k+1, n, a)
            a[k], a[i] = a[i], a[k]
```

Запускаем программу и проверяем её работу при различных значениях числа элементов в множестве. Рис. 9.1 убеждает нас, что программа верно генерирует перестановки.

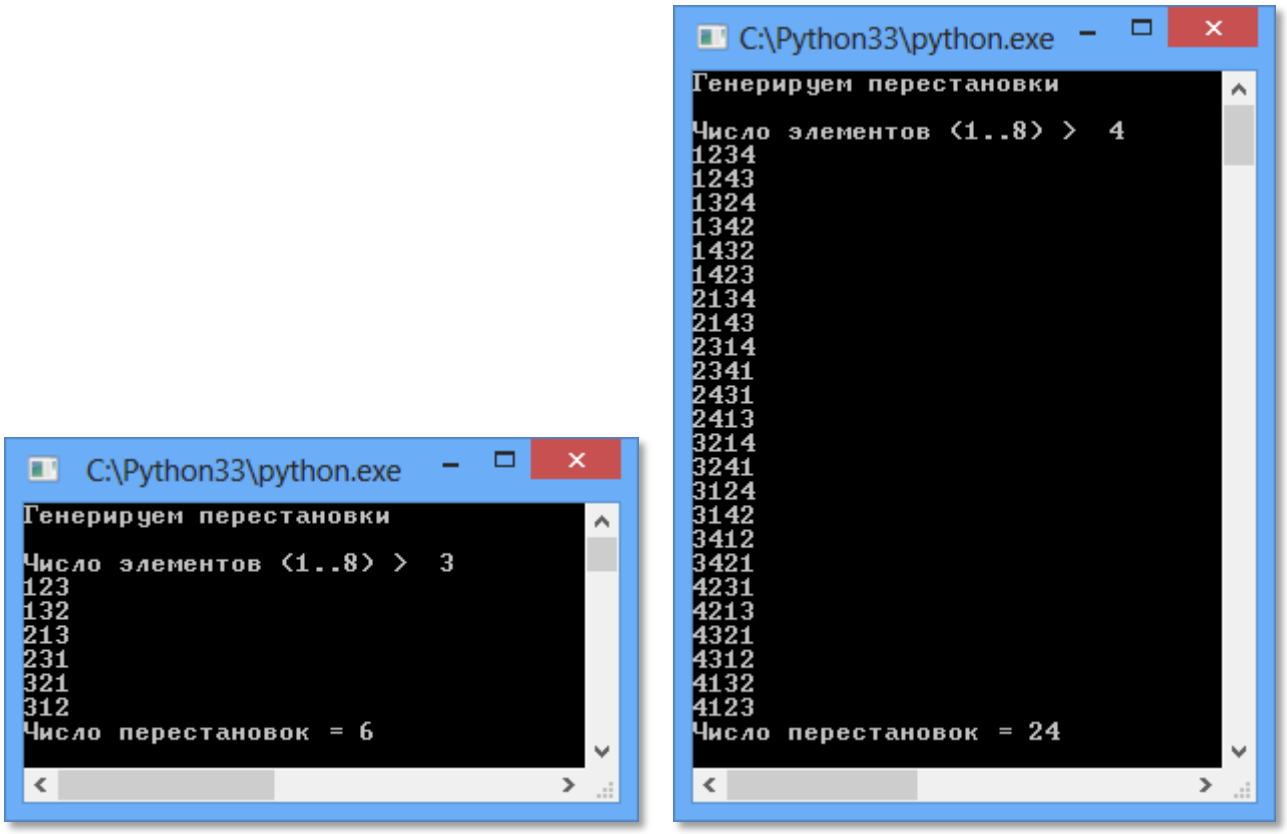


Рис. 9.1. Перестановочная программа в работе!

Исходный код программы находится в папке **Перестановки Рекурс**.

Проект *Как собрать Дрим тим?*

Список
Бесконечный цикл ***while***
Метод ***int***
Условный оператор ***if***
Оператор ***return***
Оператор ***continue***
Функция*с параметрами*
Цикл ***while***
Оператор ***break***
Условный оператор ***if-else***
Функция *без параметров*

Давайте решим ещё одну жизненно важную задачу. Представим себе, что из 16 футболистов нам нужно выпустить на поле 11.

Общее число футболистов для комбинаторики - это число элементов в **множестве**, а число игроков в команде – число элементов в **подмножестве**. Таким образом, нам нужно найти **все подмножества заданного множества**. Каждое подмножество какого-либо множества иначе называют набором из заданного числа элементов, или **сочетанием**. В сочетании порядок элементов не играет роли, поэтому мы отберём только 11 игроков в команду, а как между ними поделить номера на футболках (а, значит, и их амплуа на поле), - это уже задача тренера.

Объявим новые *глобальные переменные* программы **Сочетания**:

```
# -*- coding: Windows-1251 -*-
#Сочетания

#ПРОГРАММА ДЛЯ ГЕНЕРИРОВАНИЯ
#ВСЕХ ПОДМНОЖЕСТВ k МНОЖЕСТВА ЧИСЕЛ n=1..nElem

#макс. число элементов в
#множестве:
MAX_ELEM = 30

#число элементов в
#подмножестве:
k = 0
#список, содержащий
#очередное подмножество:
a = []
```

```
#номер подмножества:
nSubset = 0
```

В функции **main** пользователь должен ввести *два* числа. Важно учесть, что в подмножестве элементов *не больше*, чем в множестве:

```
#ГЛАВНАЯ ФУНКЦИЯ
def main():
    print('Генерируем сочетания')
    print()

    global k
    #бесконечный цикл ввода данных -
    #пока пользователь не закроет программу
    #или не введет 0:
    while True:
        s = "Число элементов (1.." + str(MAX_ELEM) +") >  "
        print(s, end = '')
        #число элементов:
        nElem = int(input())
        #если пользователь ввёл 0,
        #то программу закрываем:
        if (nElem == 0): return

        #считываем число элементов подмножества:
        s = "Число элементов в подмножестве (1.." + str(nElem) +") >
"
        print(s, end = '')
        k = int(input())

        if (k > nElem):
            print("Повторите ввод!")
            print()
            continue

        #генерируем все подмножества:
        n = SubSet(nElem)
        print("Число сочетаний = " + str(n))
        print()
```

Затем функция **SubSet** генерирует все подмножества:

```
//Генерируем все сочетания множества 1..nElem
//из k элементов
#Генерируем все сочетания множества 1..nElem
```

```
#из k элементов
def SubSet(nElem):
    global nSubset
    global a
    a.append(0)
    for i in range(1, k+1):
       a.append(i)

    nSubset = 0
    p = k
    while (p >= 1):
        #печатаем очередное подмножество:
        nSubset += 1
        WriteSubset()
        if (k == nElem): break
        if (a[k] == nElem): p -= 1
        else: p = k

        if (p >= 1):
            for i in range(k, p-1, -1):
                a[i] = a[p] + i - p + 1
    return nSubset
```

Функция **печати** очередного подмножества:

```
#ПЕЧАТАЕМ ОЧЕРЕДНУЮ ПЕРЕСТАНОВКУ
#ЭЛЕМЕНТОВ МНОЖЕСТВА
def WriteSubset():
    if (nSubset > 29): return
    s = ""
    if (nSubset < 1000):
        s += " "
    if (nSubset < 100):
        s += " "
    if (nSubset < 10):
        s += " "
    s += str(nSubset) + "> "
    for i in range(1, k+1):
        s += str(a[i]) + " "
    print(s)
```

Запускаем программу и вводим наши данные: 16 элементов в множестве и 11 – в подмножестве. Программа выдаёт огромный список из 4368 разных команд (Рис. 9.7)! Комбинаторную задачу мы решили, а вот какая из этих команд действительно станет командой мечты, тут комбинаторика бессильна!

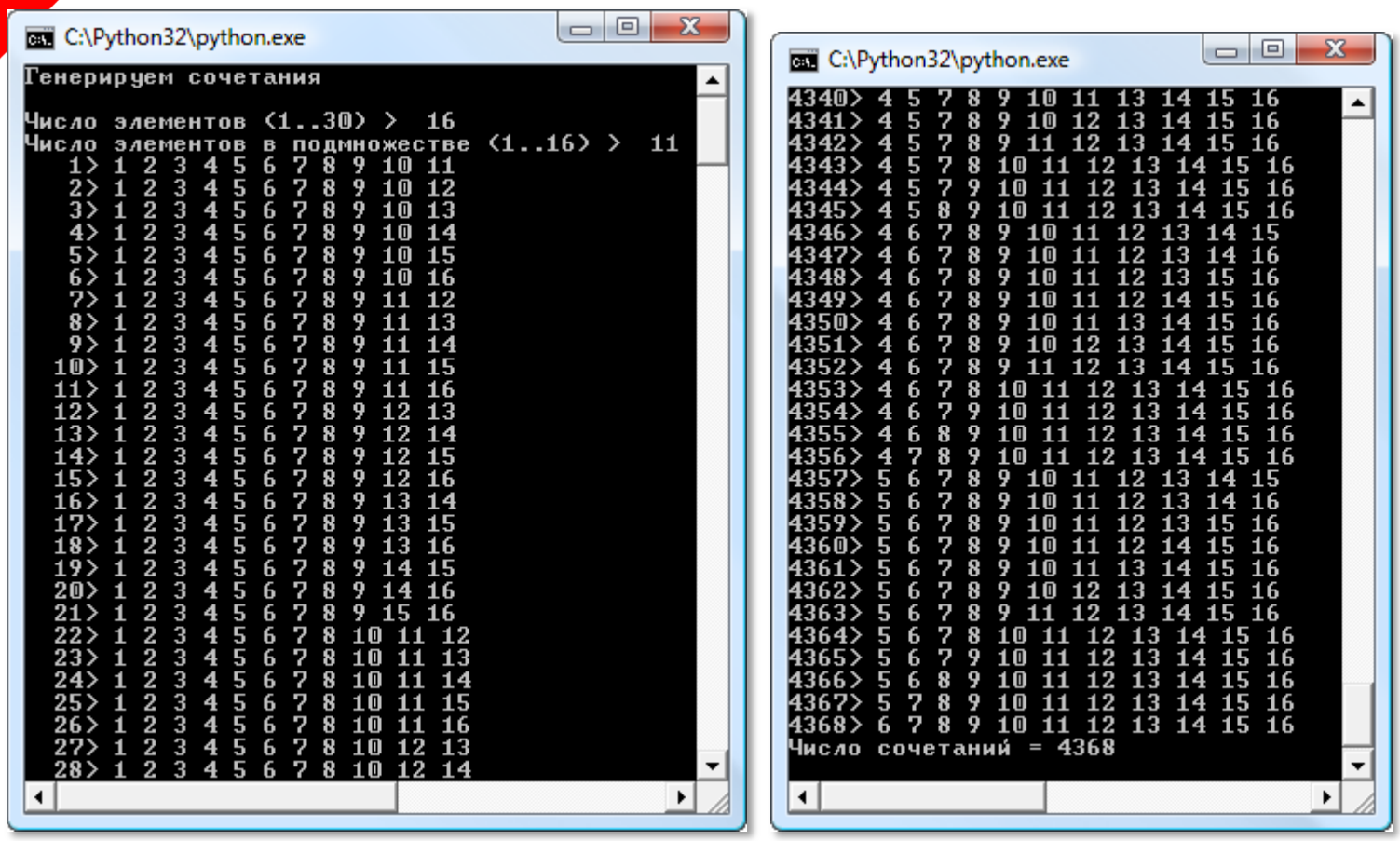


Рис. 9.7. Футбольный комбинатор

Иногда полезно заранее знать, сколько же получится сочетаний при тех или иных исходных данных - не печатать же весь список подмножеств, если нам интересно узнать только их число! На этот случай в комбинаторике припасена несложная **формула**:

$$C_n^k = \frac{n!}{k!(n-k)!}$$

> В нашей программе *n = nElem*.

Как видите, в комбинаторике без факториала не обойтись!

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

Задания для самостоятельного решения

Перестановки

Подумайте, как вместо чисел в перестановках использовать другие элементы, например, буквы.

Сочетания

Напишите программу, которая вычисляла бы и печатала в списке число сочетаний.

Ладейное окончание

Научившись генерировать перестановки, мы, сами того не подозревая, решили знаменитую комбинаторную задачу: **найти все варианты расстановки восьми ладей на шахматной доске так, чтобы они не били друг друга**.

Когда речь идёт о поиске всех вариантов, то обычно для решения задачи используют **метод перебора с возвратами** (по-английски *backtracking*). Однако число всех расстановок ладей 8! = 40320 невелико, и мы можем решить задачу **полным перебором** (*brute force*), учтя, конечно, тот факт, что на одной горизонтали может находиться *единственная* ладья.

Тогда каждая перестановка чисел *1..8* служит решением задачи о ладьях. Например, перестановке **1 2 3 4 5 6 7 8** соответствует решение, представленное на Рис. 9.29.

Действительно, если каждую ладью ставить на *отдельную* горизонталь **Г**, а в ней – на ту вертикаль, номер которой совпадает с числом, стоящим в **Г**-той позиции перестановки, то ни одна ладья не будет угрожать другим ладьям - и задача решена.

Тысячная перестановка, которую выдаёт наша программа *Генерируем перестановки*, такая: **1 4 3 7 2 6 5 8**. Если мы расставим ладьи согласно этой перестановке, то также получим решение задачи (Рис. 9.30).

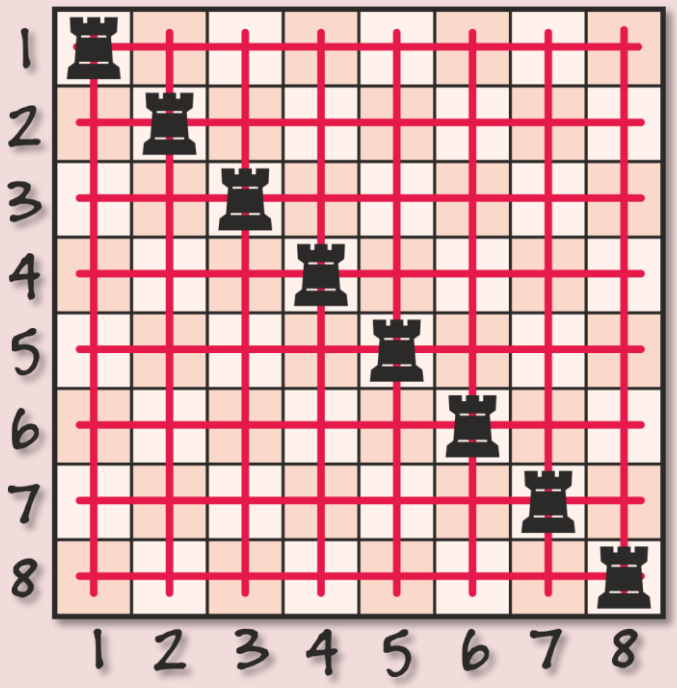


Рис. 9.29. Перестановка ладей

Таким образом, мы умеем находить все решения ладейной задачи, и вам нужно только и всего, что красиво вывести их на экран. Можно ограничиться крестиками **X** в консольном окне, хотя в картинках было бы лучше...

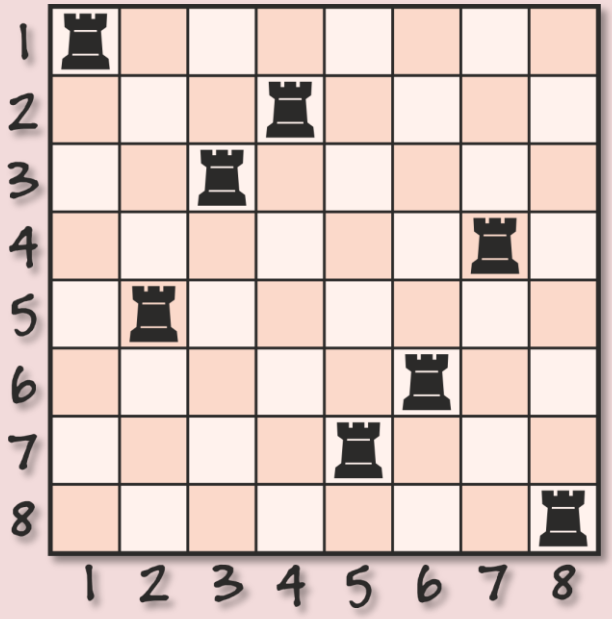


Рис. 9.30. Одно из решений ладейной задачи

Ферзевой гамбит

Если мы заменим ладей **ферзями**, то оба наших решения окажутся неверными. Ферзи – более серьёзные шахматные фигуры и могут больно ударить и по **диагонали**! С другой стороны, совершенно очевидно, что среди 40320 решений для ладей отыщутся и все 92 решения для ферзей. Например, такое (Рис. 9.31).

Так что для решения проблемы с ферзями вам необходимо устроить проверку всех перестановок и отобрать только те, в которых ферзи не угрожают друг другу. Конечно, не лучший способ для решения этой задачи, но вполне пригодный.

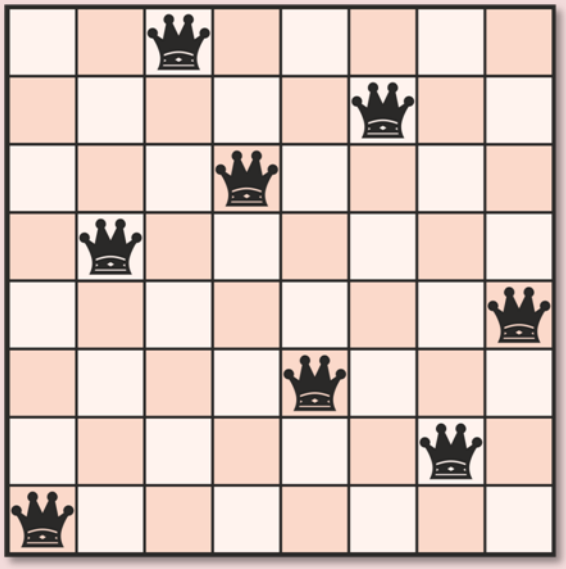
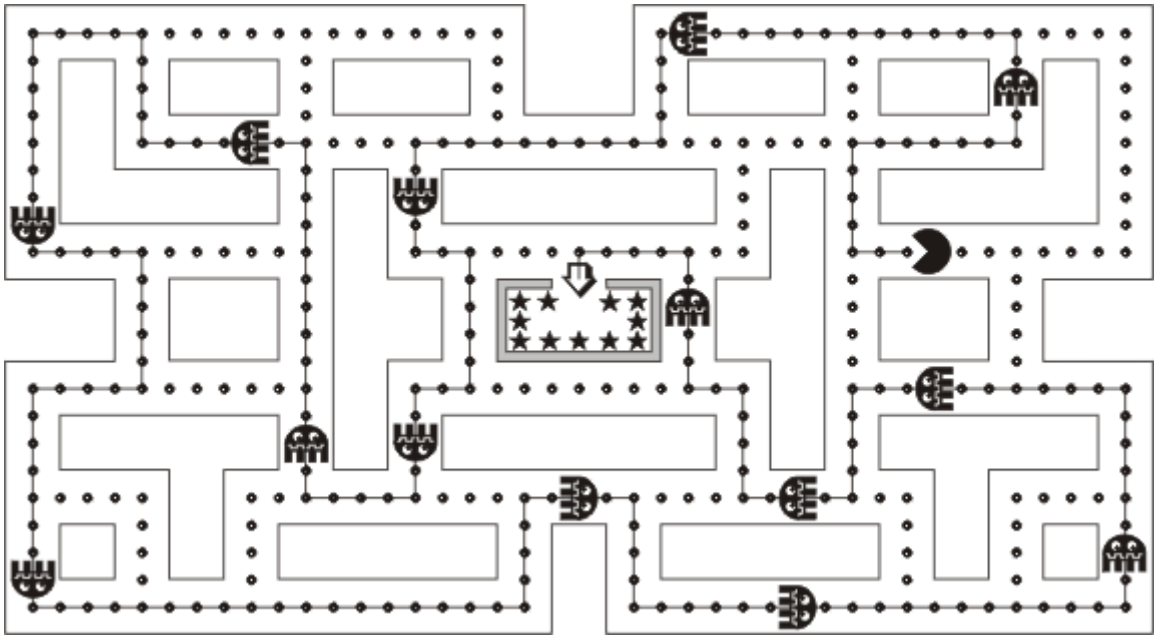


Рис. 9.31. Безобидные ферзи

Ответы

Космический охотник



Литература

[Нагибин88]	
	Нагибин Ф.Ф., Канин Е.С. **Математическая шкатулка** М.: Просвещение, 1988. – 160 с.

[100]	
	В. А. Дагене, Г. К. Григас, К. Ф. Аугутис **100 задач по программированию** М.:Просвещение, 1993. – 251 с. ISBN: 5-09-003864-3

[КА86] [КА96]

Б.А. Кордемский, А.А.Ахадов

Удивительный мир чисел
(МАТЕМАТИЧЕСКИЕ ГОЛОВОЛОМКИ
И ЗАДАЧИ ДЛЯ ЛЮБОЗНАТЕЛЬНЫХ)

М.: Просвещение, 1986. – 144 с.
М.: Просвещение, 1996. – 159 с

[ОО80]

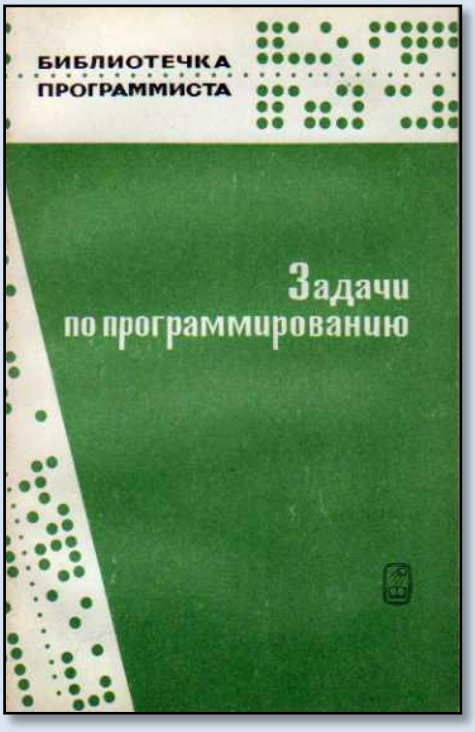
Оре О.

Приглашение в теорию чисел

М.: Наука, 1980 г. - 128 с.

Библиотечка *Квант*, Выпуск 3

[ЗП88]



Абрамов С.А. и др.

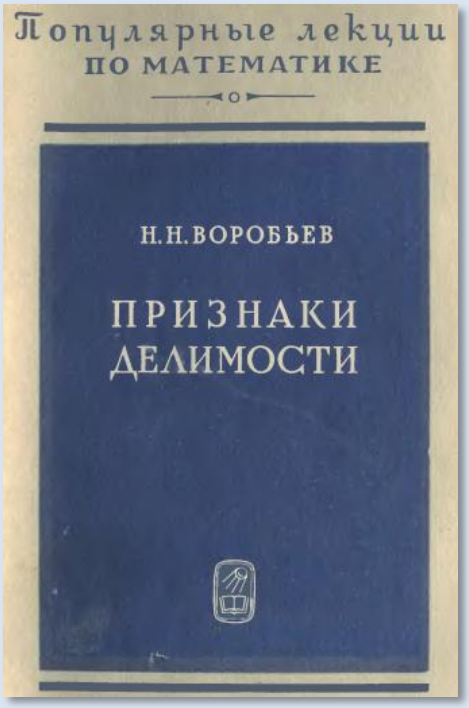
Задачи по программированию

Наука, 1988. – 224 с.

ISBN: 5-02-013774-X

Серия: Библиотечка программиста

[ВНН88]



Воробьёв Н.Н.

Признаки делимости

Наука. - 1988, 96 с.

ISBN 5-02-013731-6

[БК85]

Брудно А. Л. Каплан Л. И.

Олимпиады по программированию для школьников

Наука. - 1985, 96 с.

[BE13]

Ehrhard Behrends

Fünf Minuten Mathematik: 100 Beiträge der Mathematik-Kolumne der Zeitung DIE WELT

Springer Spektrum. - 2013, 272 с. 3-е издание

ISBN: 978-3-658-00998-4

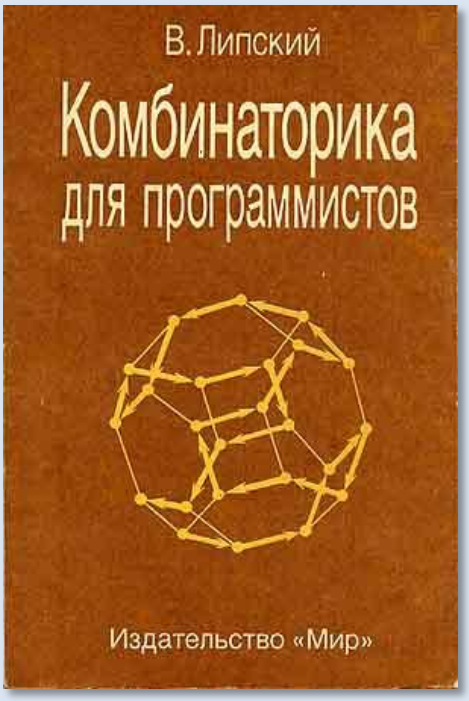
[ГМ72]

Гарднер Мартин

Математические досуги

М.: Мир, 1972. – 495 с.

[ЛВ88]

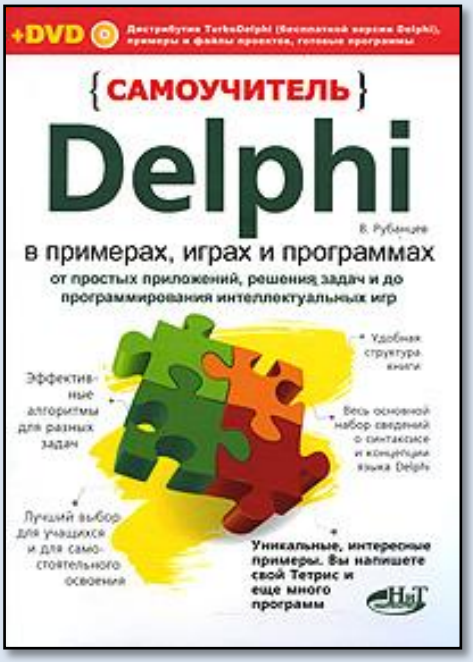


Липский В.

Комбинаторика для программистов

Москва: Мир, 1988. – 200 с.

[РВ11]



Рубанцев Валерий

Delphi в примерах, играх и программах. От простых приложений, решения задач и до программирования интеллектуальных игр

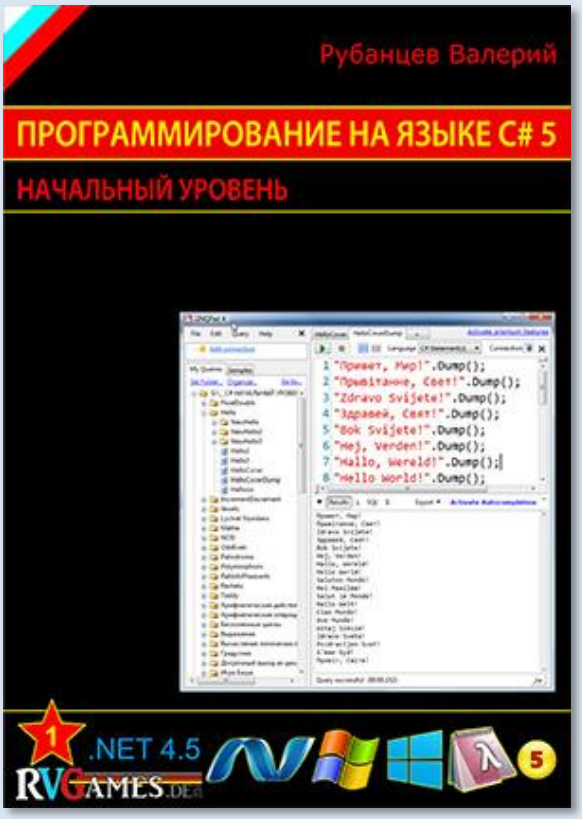
Наука и Техника, 2011. – 672 с.

ISBN: 978-5-94387-664-6

Серия: Самоучитель

Книги из серии *Программирование на языке C# 5.0: Начальный уровень*

[CS10]



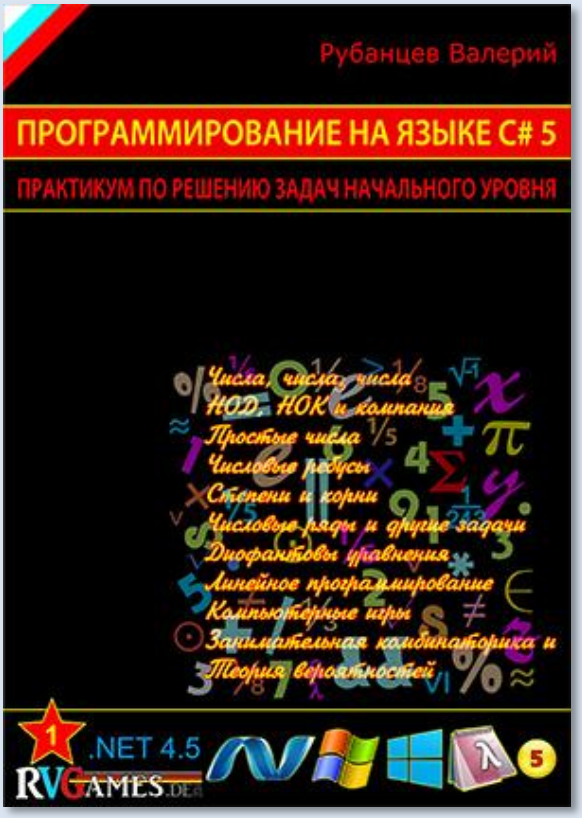
Рубанцев Валерий

Программирование на языке C# 5: Начальный уровень

RVGames, 2014. – 620 с.

Основы программирования на языке *Си-шарп*.

[CS11]



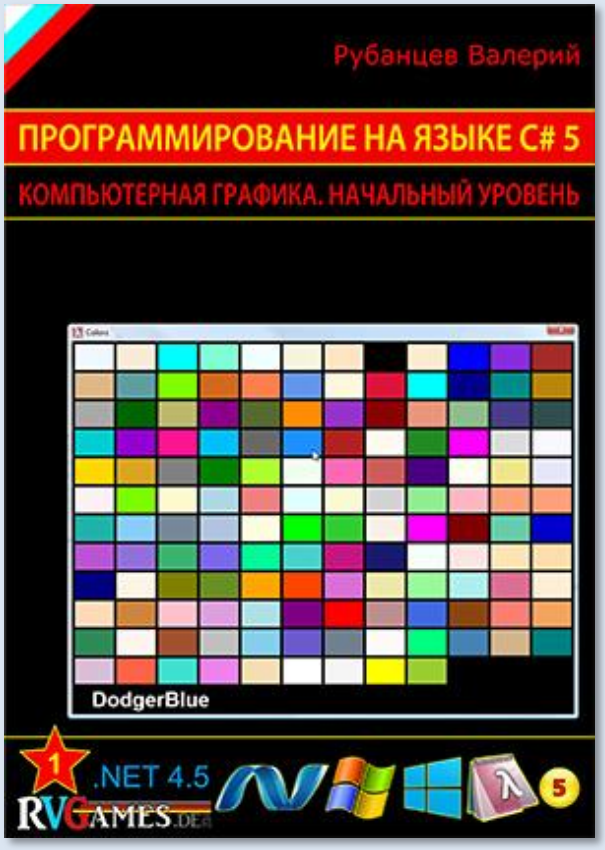
Рубанцев Валерий

Программирование на языке C# 5: Практикум по решению задач начального уровня

RVGames, 2014. – 420 с.

Многочисленные проекты для укрепления навыков программирования на языке *Си-шарп*.

[CS12]



Рубанцев Валерий

Программирование на языке C# 5: Компьютерная графика. Начальный уровень

RVGames, 2014. – 460 с.

Основы компьютерной графики.

[CS13]

Рубанцев Валерий

Программирование на языке C# 5: Тотальный тренинг по *Си-шарпу*. Начальный уровень

RVGames, 2014. – 400 с.

Разнообразные полезные и занимательные проекты на языке *Си-шарп*.