Данил Душистов
Решение 50 типовых задач по программированию на языке Pascal

Дата размещения сборника в сети: 31.08.2012
Онлайн-версия сборника находится на сайте http://el-prog.narod2.ru/
Со всеми вопросами и комментариями обращаться на E-mail: danildushistov@gmail.com

Аннотация

Этот сборник содержит подробные решения 50 практических задач, данных в рамках учебного курса «Введение в информатику и программирование», который читается в Адыгейском государственном университете. Он может быть интересен школьникам, студентам и всем, кто изучает основы программирования на языке Pascal.

В качестве дополнительного материала прилагаются тексты решений всех задач для сред PascalABC.NET и Borland Delphi 7.

Предисловие от автора

Этот сборник не может быть использован в качестве учебного пособия. В нем практически отсутствует теория, к тому же предполагается, что его читатель уже знает некоторые базисные понятия в программировании, умеет объявлять переменные и может самостоятельно скомпилировать «пустую» программу. Единственное исключение отводится для элементов синтаксиса – при первом упоминании их смысл раскрывается довольно подробно.

На самом деле, во всем этом нет какой-то особой необходимости. В наше время в Интернете можно найти массу интересных теоретических материалов по программированию на языке Pascal, и по мере надобности читатель, если ему что-то непонятно, может к ним обращаться.

Все, что можно найти в сборнике – это последовательные, максимально подробные разборы задач. Они представлены в достаточно свободном стиле: сначала задача анализируется, затем составляется алгоритм, и для каждого его шага дается детальное описание. В конце разбора каждой задачи для наглядности приводится код всей программы. В некоторых случаях приводится алгоритм на естественном языке, блок-схема какой-то части программы или описание работы алгоритма для определенного варианта исходных данных.

Уровень сложности повышается при увеличении номеров задач, если не сказано обратное. Поэтому если читатель не может понять решение какой-то задачи, то ему стоит подняться выше и попробовать понять более простые задачи.

Стоит учесть, что автор сборника – студент Адыгейского государственного университета, перешедший на 2-ой курс и практически не имеющий педагогического опыта, поэтому он будет признателен за любое указание на присутствующие в сборнике ошибки, недостатки в изложении материала и т. п., как и за любые другие комментарии.

Глава 1. Линейные алгоритмы

Задача № 1. Вывести на экран сообщение «Hello World!»

Формулировка. Вывести на экран сообщение «Hello World!».

Некоторые учебные курсы по программированию рассматривают эту задачу как самую первую при изучении конкретного языка или основ программирования.

Решение. Эта задача включает в себя лишь демонстрацию использования оператора вывода **write** (или **writeln)**, который будет единственным в теле нашей маленькой программы. С помощью него мы будем осуществлять вывод на экран константы `'Hello World!'` типа **string** (или, как допускается говорить, строковой константы). В данном случае будем использовать оператор **writeln**.

Напомню, что при использовании оператора **write** курсор останется в той же строке, в которой осуществлялся вывод, и будет находиться на одну позицию правее восклицательного знака во фразе «Hello World!», а при использовании оператора **writeln** – на первой позиции слева в следующей строке.

Код:

```
1. program HelloWorld;
2.
3. begin
4.   writeln('Hello World!')
5. end.
```

Задача № 2. Вывести на экран три числа в порядке, обратном вводу

Формулировка. Вывести на экран три введенных с клавиатуры числа в порядке, обратном их вводу.

Другими словами, мы ввели с клавиатуры три числа (сначала первое, потом второе и третье), и после этого единственное, что нам нужно сделать – это вывести третье, затем второе и первое.

Решение. Так как с клавиатуры вводится три числа, необходимо завести три переменные. Обозначим их как **a**, **b** и **c**. Ввиду того, что нам ничего не сказано о том, в каком отрезке могут располагаться введенные числа, мы возьмем тип **integer**, так как он охватывает и положительные, и отрицательные числа в некотором диапазоне (от –2147483648 до 2147483647). Затем нам нужно использовать оператор вывода **write** (**writeln**), в списке аргументов которого (напомним, что список аргументов **write** (**writeln**) может содержать не только переменные, но и константы и арифметические выражения) эти переменные будут находиться в обратном порядке. В данном случае будем использовать оператор **writeln**, который после вывода результата переведет курсор на следующую строку:

```
writeln(c, b, a);
```

Однако если мы оставим его в таком виде, то увидим, что при выводе между переменными не будет никакого пробела, и они будут слеплены и визуально смотреться как одно число. Это связано с тем, что при вводе мы использовали пробелы для разделения чисел, а сами пробелы никаким образом не влияют на содержимое переменных, которые будут последовательно выведены оператором **writeln** без каких-либо дополнений. Чтобы избежать этого, нам нужно добавить в список аргументов **writeln** две текстовые константы-пробелы. Проще говоря, пробельная константа – это символ

пробела, заключенный в одиночные апострофы (апостроф – символ «'»). Первая константа будет разделять переменные **a** и **b**, вторая – **b** и **c**. В результате наш оператор вывода будет таким:

```
  writeln(c, ' ', b, ' ', a);
```

Теперь он работает так: выводит переменную **c**, затем одиночный символ пробела, затем переменную **b**, потом еще один символ пробела и, наконец, переменную **a**.

Код:

```
 1. program WriteThree;
 2.
 3. var
 4.   a, b, c: integer;
 5.
 6. begin
 7.   readln(a, b, c);
 8.   writeln(c, ' ', b, ' ', a)
 9. end.
```

Задача № 3. Вывести на экран квадрат введенного числа

Формулировка. Дано натуральное число меньше 256. Сформировать число, представляющее собой его квадрат.

Решение. Для ввода числа нам необходима одна переменная. Обозначим эту переменную как **a**. Так как нам ничего не сообщается о необходимости сохранить исходное число, то для получения квадрата мы можем использовать ту же самую переменную, в которую считывали число с клавиатуры.

В условии задачи дается ограничитель величины вводимого числа – фраза «меньше 256». Это означает, что оно может быть охвачено типом **byte**. Но что произойдет, если в переменную **a** будет введено число 255, и затем мы попытаемся присвоить ей его квадрат, равный 65025? Естественно, это вызовет переполнение типа данных, так как используемой для переменной **a** ячейки памяти не хватит для того, чтобы вместить число 65025. Значит, для ее описания мы должны использовать более емкий числовой тип. При этом типом минимальной размерности, охватывающим данный отрезок (от 1 (это 1^2) до 65025), является тип **word**. Его мы и будем использовать при описании **a**.

Далее нужно сформировать в переменной **a** квадрат. Для этого присвоим ей ее прежнее значение, умноженное само на себя:

```
  a := a * a;
```

Теперь остается вывести результат на экран. Для этого будем использовать оператор **writeln**.

Код:

```
 1. program SqrOfNum;
 2.
 3. var
 4.   a: word;
 5.
 6. begin
 7.   readln(a);
 8.   a := a * a;
 9.   writeln(a)
10. end.
```

Задача № 4. Получить реверсную запись трехзначного числа

Формулировка. Сформировать число, представляющее собой реверсную (обратную в порядке следования разрядов) запись заданного трехзначного числа. Например, для числа 341 таким будет 143.

Давайте разберемся с условием. В нашем случае с клавиатуры вводится некоторое трехзначное число (трехзначными называются числа, в записи которых три разряда (то есть три цифры), например: 115, 263, 749 и т. д.). Нам необходимо получить в некоторой переменной число, которое будет представлять собой реверсную запись введенного числа. Другими словами, нам нужно перевернуть введенное число «задом наперед», представить результат в некоторой переменной и вывести его на экран.

Решение. Определимся с выбором переменных и их количеством. Ясно, что одна переменная нужна для записи введенного числа с клавиатуры, мы обозначим ее как **n**. Так как нам нужно переставить разряды числа **n** в некотором порядке, следует для каждого из них также предусмотреть отдельные переменные. Обозначим их как **a** (для разряда единиц), **b** (для разряда десятков) и **c** (для разряда сотен).

Теперь можно начать запись самого алгоритма. Будем разбирать его поэтапно:

1) Вводим число **n**;
2) Работаем с разрядами числа **n**. Как известно, последний разряд любого числа в десятичной системе счисления – это остаток от деления этого числа на 10. В терминах языка **Pascal** это означает, что для получения разряда единиц нам необходимо присвоить переменной **a** остаток от деления числа **n** на 10. Этому шагу соответствует следующий оператор:
   ```
   a := n mod 10;
   ```
   Получив разряд единиц, мы должны отбросить его, чтобы иметь возможность продолжить работу с разрядом десятков. Для этого разделим число **n** на 10. В терминах **Pascal**, опять же, это означает: присвоить переменной **n** результат от деления без остатка числа **n** на 10. Это мы сделаем с помощью оператора
   ```
   n := n div 10;
   ```
3) Очевидно, что после выполнения **п. 2** в переменной **n** будет храниться двухзначное число, состоящее из разряда сотен и разряда десятков исходного. Теперь, выполнив те же самые действия еще раз, мы получим разряд десятков исходного числа, но его уже нужно присваивать переменной **b**.
4) В результате в переменной **n** будет храниться однозначное число – разряд сотен исходного числа. Мы можем без дополнительных действий присвоить его переменной **c**.
5) Все полученные в переменных числа – однозначные. Теперь переменная **n** нам больше не нужна, и в ней нужно сформировать число-результат, в котором **a** будет находиться в разряде сотен, **b** – десятков, **c** – единиц. Легко понять, что для этого нам следует умножить **a** на 100, прибавить к полученному числу **b**, умноженное на 10 и **c** без изменения, и весь этот результат присвоить переменной **c**. Это можно записать так:
   ```
   n := 100 * a + 10 * b + c;
   ```
6) Далее остается только вывести полученное число на экран.

Код:

```
1. program ReverseNum;
2.
3. var
4.   n, a, b, c: word;
5.
6. begin
```

```
 7.   readln(n);
 8.   a := n mod 10;
 9.   n := n div 10;
10.   b := n mod 10;
11.   n := n div 10;
12.   c := n;
13.   n := 100 * a + 10 * b + c;
14.   writeln(n)
15. end.
```

Проверим работу программы на произвольном варианте введенных данных. Для этого выполним ее «ручную прокрутку», проделав с введенным числом те же действия, которые должен выполнить алгоритм.

Пусть пользователем введено число 514. Покажем в таблице, какие значения будут принимать переменные **после** выполнения соответствующих строк. При этом прочерк означает, что значение переменных на данном шаге не определено, а красным цветом выделены переменные, которые изменяются:

№ строки	n	a	b	c
7	514	—	—	—
8	514	4	—	—
9	51	4	—	—
10	51	4	1	—
11	5	4	1	—
12	5	4	1	5
13	415	4	1	5

Нетрудно понять, что написанная программа будет выводить правильный ответ для любого заданного трехзначного числа, так как в соответствии с алгоритмом заполнение данной таблицы возможно лишь единственным образом. Это значит, что мы можем представить число в виде абстрактного трехзначного числа ***xyz***, (в нем каждая буква должна быть заменена на любое число от 0 до 9, конечно, за исключением тех случаев, когда оно перестает быть трехзначным), и работая с разрядами этого числа, показать, что в результате работы ответом будет число ***zyx***.

Задача № 5. Посчитать количество единичных битов числа

Формулировка. Дано натуральное число меньше 16. Посчитать количество его единичных битов. Например, если дано число 9, запись которого в двоичной системе счисления равна 1001_2 (подстрочная цифра 2 справа от числа означает, что оно записано в двоичной системе счисления), то количество его единичных битов равно 2.

Решение. Нам необходима переменная для ввода с клавиатуры. Обозначим ее как **n**. Так как мы должны накапливать количество найденных битов, то возникает потребность в еще одной переменной. Обозначим ее как **count** («*count*» в переводе с англ. означает «считать», «подсчет» и т. д.). Переменные возьмем типа **byte** (они могут принимать значения от 0 до 255), и пусть в данном случае такой объем избыточен, но это не принципиально важно.

Как же сосчитать количество битов во введенном числе? Ведь число же вводится в десятичной системе счисления, и его нужно переводить в двоичную?

На самом деле все гораздо проще. Здесь нам поможет одно интересное правило:

> *Остаток от деления любого десятичного числа **x** на число **p** дает нам разряд единиц числа **x** (его крайний разряд справа) в системе счисления с основанием **p**.*

То есть, деля некоторое десятичное число, например, на 10, в остатке мы получаем разряд единиц этого числа в системе счисления с основанием 10. Возьмем, например, число 3468. Остаток от деления его на 10 равен 8, то есть разряду единиц этого числа.

Понятно, что такие же правила господствуют и в арифметике в других системах счисления, и в том числе в двоичной системе. Предлагаю поэкспериментировать: запишите на бумаге десятичное число, затем, используя любой калькулятор с функцией перевода из одной системы счисления в другую, переведите это число в двоичную систему счисления и также запишите результат. Затем разделите исходное число на 2 и снова переведите в двоичную систему. Как оно изменилось в результате? Вполне очевидно, что у него пропал крайний разряд справа, или, как мы уже говорили ранее, разряд единиц.

Но как это использовать для решения задачи? Воспользуемся тем, что в двоичной записи числа нет цифр, кроме 0 и 1. Легко убедиться в том, что сложив все разряды двоичного числа, мы получаем как раз таки количество его единичных битов. Это значит, что вместо проверки значений разрядов двоичного представления числа мы можем прибавлять к счетчику сами эти разряды – если в разряде был 0, значение счетчика не изменится, а если 1, то повысится на единицу.

Теперь, резюмируя вышеприведенный итог, можно поэтапно сформировать сам алгоритм:

1) Вводим число **n**;
2) Обнуляем счетчик разрядов **count**. Это делается потому, что значения всех переменных при запуске программы считаются неопределенными, и хотя в большинстве компиляторов **Pascal** они обнуляются при запуске, все же считается признаком «хорошего тона» в программировании обнулить значение переменной, которая будет изменяться в процессе работы без предварительного присваивания ей какого-либо значения.
3) Прибавляем к **count** разряд единиц в двоичной записи числа **n**, то есть остаток от деления **n** на 2:
   ```
   count := count + n mod 2;
   ```
   Строго говоря, мы могли бы не прибавлять предыдущее значение переменной **count** к остатку от деления, так как оно все равно было нулевым. Но мы поступили так для того, чтобы сделать код более однородным, далее это будет видно. Учтя разряд единиц в двоичной записи **n**, мы должны отбросить его, чтобы исследовать число далее. Для этого разделим **n** на 2. На языке **Pascal** это будет выглядеть так:
   ```
   n := n div 2;
   ```
4) Теперь нам нужно еще два раза повторить **п. 3** , после чего останется единственный двоичный разряд числа **n**, который можно просто прибавить к счетчику без каких-либо дополнений:
   ```
   count := count + n;
   ```
5) В результате в переменной **count** будет храниться количество единичных разрядов в двоичной записи исходного числа. Осталось лишь вывести ее на экран.

Код:

```
1. program BinaryUnits;
2.
3. var
4.   n, count: byte;
```

```
 5.
 6. begin
 7.   readln(n);
 8.   count := 0;
 9.   count := count + n mod 2;
10.   n := n div 2;
11.   count := count + n mod 2;
12.   n := n div 2;
13.   count := count + n mod 2;
14.   n := n div 2;
15.   count := count + n;
16.   writeln(count)
17. end.
```

Программа работает правильно на всех вариантах правильных исходных данных, в чем несложно убедиться с помощью простой проверки.

Глава 2. Условные операторы

Задача № 6. Вывести на экран наибольшее из двух чисел

Формулировка. Даны два числа. Вывести на экран то из них, которое больше.

Решение. Собственно, это самая простая задача, с помощью которой можно продемонстрировать использование условного оператора **if**. Напомним, как нужно использовать этот оператор. Мы вводим с клавиатуры числа в переменные **a** и **b** типа **integer**, затем в операторе **if** проверяем булевское выражение «**a > b**»: если оно истинно, то выполняется **then**-блок оператора, если ложно – **else**-блок. Соответственно, если **a** больше **b** (условие в заголовке истинно), то в **then**-блоке мы выводим **a**, а если **a** не больше **b** (условие в заголовке ложно), то выводим **b** (хотя сюда попадает и случай, когда **a** = **b**, что, впрочем, не нарушает решения).

На языке **Pascal** мы можем записать весь оператор с **if**- и **then**-блоками в одну строчку следующим образом:

```
if a > b then writeln(a) else writeln(b);
```

Данная строка легко понятна и читаема по причине того, что мы выполняем столь простой набор операторов в обоих блоках ветвления оператора **if**. Однако в более сложных примерах мы будем с первых же написанных строчек следовать *принципу аккуратного оформления кода*, чтобы не появлялось привычки «вытягивать» операторы ветвлений и другие конструкции в одну строчку, так как в будущем это может сильно сказаться на удобочитаемости и простоте понимания написанного программного кода, особенно при увеличении количества вложенных в блок операторов (которые, например, тоже могут быть операторами ветвления). Не стоит забывать о том, что при вложенности в тело какого-либо оператора хотя бы одного составного оператора или другой сложной конструкции требуется равномерный отступ для подчиненной конструкции с адекватной расстановкой операторных скобок! Например, для оператора **if** это распределение конструкций по мнемонической модели **if-end**, **else-end**, согласно которой эти ключевые слова должны стоять на одном уровне по вертикали, а их содержимое должно быть немного смещено вправо.

Конечно, для простейшей конструкции с условным оператором это вовсе не самоцель, и можно разместить ее в одной строке, если оби ветви оператора (и **if**-блок, и **else**-блок) не содержат составного оператора. В нашем же примере «аккуратное оформление» показывается лишь в качестве введения.

Код:

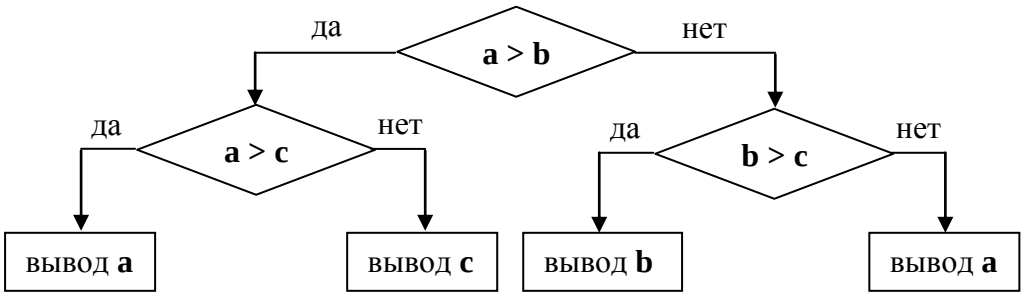
```
 1. program MaxOfTwo;
 2.
 3. var
 4.   a, b: integer;
 5.
 6. begin
 7.   readln(a, b);
 8.   if a > b then begin
 9.     writeln(a)
10.   end
11.   else begin
12.     writeln(b)
13.   end
14. end.
```

При таком оформлении хорошо видно, какой код выполняется при истинности условия, а какой – при его ложности.

Задача № 7. Вывести на экран наибольшее из трех чисел

Формулировка. Даны три числа. Вывести на экран то из них, которое больше.

Решение. Даная задача обобщает предыдущую. В ее решении также нужно использовать условный оператор **if**, однако в данном случае для нахождения максимального числа нам нужно выполнить минимум два сравнения. Сам механизм выбора в виде условного оператора с вложенными в него двумя другими условными операторами можно легко пояснить следующей блок-схемой:



Несмотря на то, что выполняется всего одна инструкция вывода, при написании кода мы все ветвления будем помещать в отдельный составной оператор. Напомним: это значит, что при движении от более общего уровня к частному все конструкции нужно смещать на два пробела относительно родительского блока/оператора.

Код:

```
 1. program MaxOfThree;
 2.
 3. var
 4.   a, b, c: integer;
 5.
 6. begin
 7.   readln(a, b, c);
 8.   if a > b then begin
 9.     if a > c then begin
10.       writeln(a)
```

```
11.     end
12.     else begin
13.       writeln(c)
14.     end
15.   end
16.   else begin
17.     if b > c then begin
18.       writeln(b)
19.     end
20.     else begin
21.       writeln(c)
22.     end
23.   end
24. end.
```

Задача № 8. Вывести название дня недели по его номеру

Формулировка. Вывести название дня недели по его номеру.

Решение. Задача простейшим образом решается с помощью оператора выбора **case**. Напомним, что этот оператор позволяет организовать ветвления в зависимости от значений некоторой переменной, для каждого из которых можно предусмотреть выполнение различных действий. Причем если значению переменной не соответствует ни один вариант, выполняется **else**-блок (если он присутствует). Кстати, не стоит забывать, что после перечисления всех вариантов оператора case необходимо написать ключевое слово end (выходит, ключевое слово **case** является еще и открывающей операторной скобкой).

Для того чтобы воспользоваться оператором **case**, нам необходимо произвести ввод номера дня недели в некоторую переменную **i** типа **byte** и по этому номеру вывести название текущего дня недели. Кстати, благодаря **else**-блоку в этой программке мы впервые предусмотрим сообщение об ошибке, связанной с некорректно введенным номером, которому не соответствует ни один из дней недели.

Код:

```
 1. program DaysOfTheWeek;
 2.
 3. var
 4.   i: byte;
 5.
 6. begin
 7.   readln(i);
 8.   case i of
 9.     1: writeln('Monday');
10.     2: writeln('Tuesday');
11.     3: writeln('Wednesday');
12.     4: writeln('Thursday');
13.     5: writeln('Friday');
14.     6: writeln('Saturday');
15.     7: writeln('Sunday')
16.     else writeln('This day of the week does not exist!')
17.   end
18. end.
```

Кстати, в каждом из вариантов ветвлений может быть помещен составной оператор, но при описании вариантов мы не стали использовать операторные скобки, так как на этот раз они наоборот испортили бы все оформление кода, которое сейчас является достаточно гармоничным.

Задача № 9. Проверить, является ли четырехзначное число палиндромом

Формулировка. Дано четырехзначное число. Проверить, является ли оно палиндромом.

Примечание: палиндромом называется число, слово или текст, которые одинакового читаются слева направо и справа налево. Например, в нашем случае это числа 1441, 5555, 7117 и т. д.

Примеры других чисел-палиндромов произвольной десятичной разрядности, не относящиеся к решаемой задаче: 3, 787, 11, 91519 и т. д.

Решение. Для ввода числа с клавиатуры будем использовать переменную **n**. Вводимое число принадлежит множеству натуральных чисел и четырехзначно, поэтому оно заведомо больше 255, так что тип **byte** для ее описания нам не подходит. Тогда будем использовать тип **word**.

Какими же свойствами обладают числа-палиндромы? Из указанных примеров легко увидеть, что в силу своей одинаковой «читаемости» с двух сторон в них равны первый и последний разряд, второй и предпоследний и т. д. вплоть до середины. Причем если в числе нечетное количество разрядов, то серединную цифру можно не учитывать при проверке, так как при выполнении названного правила число является палиндромом вне зависимости от ее значения.

В нашей же задаче все даже несколько проще, так как на вход подается четырехзначное число. А это означает, что для решения задачи нам нужно лишь сравнить 1-ю цифру числа с 4-й и 2-ю цифру с 3-ей. Если выполняются оба эти равенства, то число – палиндром. Остается только получить соответствующие разряды числа в отдельных переменных, а затем, используя условный оператор, проверить выполнение обоих равенств с помощью булевского (логического) выражения.

Однако не стоит спешить с решением. Может быть, мы сможем упростить выведенную схему? Возьмем, например, уже упомянутое выше число 1441. Что будет, если разделить его на два числа двузначных числа, первое из которых будет содержать разряд тысяч и сотен исходного, а второе – разряд десятков и единиц исходного. Мы получим числа 14 и 41. Теперь, если второе число заменить на его реверсную запись (это мы делали в **задаче 4**), то мы получим два равных числа 14 и 14! Это преобразование вполне очевидно, так в силу того, что палиндром читается одинаково в обоих направлениях, он состоит из дважды раза повторяющейся комбинации цифр, и одна из копий просто повернута задом-наперед.

Отсюда вывод: нужно разбить исходное число на два двузначных, одно из них реверсировать, а затем выполнить сравнение полученных чисел с помощью условного оператора **if**. Кстати, для получения реверсной записи второй половины числа нам необходимо завести еще две переменные для сохранения используемых разрядов. Обозначим их как **a** и **b**, и будут они типа **byte**.

Теперь опишем сам алгоритм:

1) Вводим число **n**;
2) Присваиваем разряд единиц числа **n** переменной **a**, затем отбрасываем его. После присваиваем разряд десятков **n** переменной **b** и также отбрасываем его:
```
a := n mod 10;
n := n div 10;
b := n mod 10;
n := n div 10;
```
3) Присваиваем переменной **a** число, представляющее собой реверсную запись хранящейся в переменных **a** и **b** второй части исходного числа **n** по уже известной формуле:
```
a := 10 * a + b;
```

4) Теперь мы можем использовать проверку булевского выражения равенства полученных чисел **n** и **a** помощью оператора **if** и организовать вывод ответа с помощью ветвлений:
```
if n = a then writeln('Yes') else writeln('No');
```
Так как в условии задачи явно не сказано, в какой форме необходимо выводить ответ, мы будем считать логичным вывести его на интуитивно понятном пользователю уровне, доступном в средствах самого языка **Pascal**. Напомним, что с помощью оператора **write** (**writeln**) можно выводить результат выражения булевского типа, причем при истинности этого выражения будет выведено слово `'TRUE'` («true» в пер. с англ. означает «истинный»), при ложности – слово `'FALSE'` («false» в пер. с англ. означает «ложный»). Тогда предыдущая конструкция с **if** может быть заменена на
```
writeln(n = a);
```

Код:

```
 1. program PalindromeNum;
 2.
 3. var
 4.   n: word;
 5.   a, b: byte;
 6.
 7. begin
 8.   readln(n);
 9.   a := n mod 10;
10.   n := n div 10;
11.   b := n mod 10;
12.   n := n div 10;
13.   a := 10 * a + b;
14.   writeln(n = a)
15. end.
```

Задача № 10. Проверить, является ли четырехзначное число счастливым билетом

Формулировка. Дано четырехзначное число. Проверить, является ли оно «счастливым билетом».

Примечание: счастливым билетом называется число, в котором: а) при четном количестве цифр в числе сумма цифр его левой половины равна сумме цифр его правой половины; б) при нечетном количестве цифр – то же самое, но с отбрасыванием серединной цифры. Например, рассмотрим число 1322. Его левая половина равна 13, а правая – 22, и оно является счастливым билетом (т. к. 1 + 3 = 2 + 2). Аналогично: 1735 (1 + 7 = 3 + 5), 1111 (1 + 1 = 1 + 1) и т. д.

Примеры других счастливых билетов за рамками условия текущей задачи: 7 (отбросили единственную цифру), 39466 (3 + 9 = 6 + 6, а 4 отбросили), 11 (1 = 1), и т. д.

Решение. Для ввода достаточно одной переменной **n** типа **word**. Все, что необходимо сделать для решения – это последовательно получить все разряды исходного числа, причем из двух младших разрядов (единиц и десятков) сформировать первую сумму, а из двух старших разрядов – вторую, после чего вывести на экран результат булевского выражения равенства полученных сумм. Первую сумму будем хранить в переменной **right**, а вторую – в переменной **left** (выбрано по расположению цифр в записи числа). Значение обоих из них не может превосходить 18 (т. к. для наибольшего допустимого числа 9999 обе суммы равны 9 + 9 = 18), поэтому для их описания используем тип **byte**.

Более детальный разбор алгоритма:

1) Вводим число **n**;

2) Присваиваем переменной **right** значение последней цифры числа **n**, потом отбрасываем эту цифру, затем повторяем то же самое, но на этот раз уже прибавляем добытую цифру к прежнему значению **right**:
```
right := n mod 10;
n := n div 10;
right := right + n mod 10;
n := n div 10;
```

3) Присваиваем переменной **left** последнюю цифру **n**, отбрасываем ее и прибавляем к **right** единственную оставшуюся в переменной **n** цифру:
```
left := n mod 10;
n := n div 10;
left := left + n;
```

4) Выводим на экран результат сравнения накопленных сумм:
```
writeln(left = right);
```

Код:

```
 1. program HappyTicket;
 2.
 3. var
 4.   n: word;
 5.   left, right: byte;
 6.
 7. begin
 8.   readln(n);
 9.   right := n mod 10;
10.   n := n div 10;
11.   right := right + n mod 10;
12.   n := n div 10;
13.   left := n mod 10;
14.   n := n div 10;
15.   left := left + n;
16.   writeln(left = right)
17. end.
```

Задача № 11. Проверить, является ли двоичное представление числа палиндромом

Формулировка. Дано число типа **byte**. Проверить, является ли палиндромом его двоичное представление с учетом того, что сохранены старшие нули. Пример таких чисел: 102 (т. к. 102 = $0110\ 0110_2$, а это палиндром), 129 (129 = $1000\ 0001_2$) и т. д.

Решение. Данная задача частично повторяет **задачу 9**. Сходство состоит в том, что и там, и здесь у проверяемых чисел фиксированная разрядность (длина), ведь здесь нам уже задан тип и получено указание сохранить старшие нули, так что в данном случае двоичные представления всех подаваемых на вход программе чисел будут восьмизначными.

Но как же упростить решение, чтобы не делать сравнение всех разрядов «в лоб»? Для этого нам нужно вспомнить правило, упомянутое в **задаче 5**, на этот раз несколько уточненное и дополненное:

> *– Остаток от деления любого числа **x** в системе счисления с основанием **p** на само число **p** дает нам крайний справа разряд числа **x**.*
>
> *– Умножение любого числа **x** в системе счисления с основанием **p** на само число **p** добавляет числу **x** новый разряд справа.*

Для примера возьмем число 158 в десятичной системе счисления. Мы можем получить его крайнюю цифру справа, которая равна 8, если возьмем остаток от деления 158 на число 10, являющееся в данном случае основанием системы счисления. С другой стороны, если мы умножим 158 на 10, то появляется новый разряд справа, и в результате мы получаем число 1580.

Согласно правилу те же самые арифметические законы актуальны и для двоичной системы счисления. А это в свою очередь означает, что мы можем разработать алгоритм наподобие того, который использовался в **задаче 9** для формирования числа, представляющего собой правую половину исходного числа, которая записана в реверсном порядке. Для этого нам нужно использовать четыре переменных для хранения двоичных разрядов правой половины двоичной записи введенного числа, добыть эти самые разряды с удалением их в исходном числе, сформировать из них двоичную реверсную запись и выполнить сравнение. Обозначим эти переменные типа **byte** как **a**, **b**, **c**, и **d**.

Опишем сам алгоритм:

1) Вводим число **n**;
2) Последовательно получаем 4 крайних справа разряда двоичной записи числа **n**: присваиваем их значение соответствующей переменной, а затем отбрасываем в исходном числе:
```
a := n mod 2;
n := n div 2;
b := n mod 2;
n := n div 2;
c := n mod 2;
n := n div 2;
d := n mod 2;
n := n div 2;
```
3) Теперь нужно подумать, как видоизменится формула, с помощью которой мы получали реверсную запись числа в **задаче 4** и **задаче 9**. Очевидно, что в десятичной системе счисления реверсную запись четырехзначного числа, разряд единиц которого находится в переменной **k**, разряд десятков – в переменной **l**, сотен – в **m** и тысяч – в **n** мы можем найти по следующей формуле (**x** в данной случае – любая переменная типа **word**):
```
x := 1000 * k + 100 * l + 10 * m + n;
```
Можно представить, что мы формируем четыре числа, которые затем складываем. Первое число 1000 * k – это разряд единиц исходного числа, к которому справа приписано три разряда (три нуля), то есть, трижды произведено умножение на основание системы счисления 10, проще говоря, число **k** умножено на 10^3. Аналогично, к **l** нужно приписать два нуля, к **m** – один ноль, а **n** оставить без изменения, так как эта цифра будет находиться в разряде единиц формируемого «перевертыша». Вспомнив правило, высказанное немного выше, преобразуем предыдущую формулу для двоичной системы счисления (будем умножать цифры на двойку в соответствующих степенях). Она получится такой (для формирования числа используется переменная **a**):
```
a := 8 * a + 4 * b + 2 * c + d;
```
4) После применения вышеприведенной строки останется лишь вывести на экран результат сравнения полученных чисел:
```
writeln(n = a);
```

Код:

```
1. program BinaryPalindrome;
2.
3. var
4.   n, a, b, c, d: byte;
5.
```

```
 6. begin
 7.   readln(n);
 8.   a := n mod 2;
 9.   n := n div 2;
10.   b := n mod 2;
11.   n := n div 2;
12.   c := n mod 2;
13.   n := n div 2;
14.   d := n mod 2;
15.   n := n div 2;
16.   a := 8 * a + 4 * b + 2 * c + d;
17.   writeln(n = a)
18. end.
```

Выполним «ручную прокрутку» программы при вводе числа 102. Покажем в таблице, какие значения будут принимать переменные после выполнения соответствующих строк (операторов) кода. Значения переменных для наглядности представлены как в десятичной, так и в двоичной системе счисления (при этом дописаны старшие нули до заполнения тетрады). При этом прочерк означает, что значение переменных на данном шаге не определено, а красным цветом выделены переменные, которые изменяются:

№ строки	Десятичная система					Двоичная система				
	n	**a**	**b**	**c**	**d**	**n**	**a**	**b**	**c**	**d**
7	102	—	—	—	—	0110 0110	—	—	—	—
8	102	0	—	—	—	0110 0110	0000	—	—	—
9	51	0	—	—	—	0011 0011	0000	—	—	—
10	51	0	1	—	—	0011 0011	0000	0001	—	—
11	25	0	1	—	—	0001 1001	0000	0001	—	—
12	25	0	1	1	—	0001 1001	0000	0001	0001	—
13	12	0	1	1	—	0000 1100	0000	0001	0001	—
14	12	0	1	1	0	0000 1100	0000	0001	0001	0000
15	6	0	1	1	0	0000 0110	0000	0001	0001	0000
16	**6**	**6**	1	1	0	**0000 0110**	**0110**	0001	0001	0000

Задача № 12. Решить квадратное уравнение

Формулировка. Даны вещественные числа **a**, **b** и **c**, причем **a** отлично от 0. Решить квадратное уравнение $ax^2 + bx + c = 0$ или сообщить о том, что действительных решений нет.

Решение. Из алгебры известно, что:

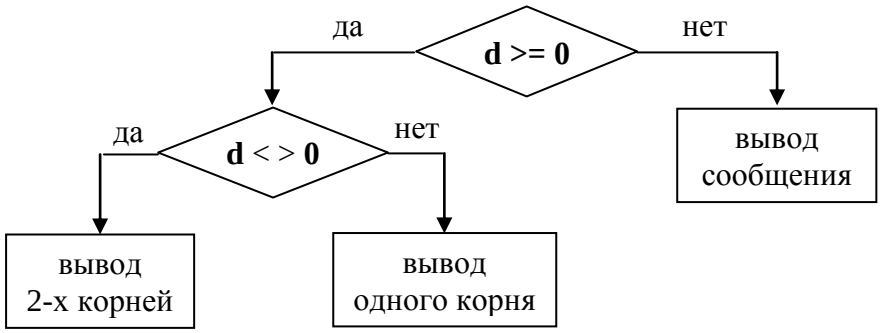
Квадратное уравнение $ax^2 + bx + c = 0$, *выражение* $D = b^2 - 4ac$ – *дискриминант:*

– *если* $D > 0$, *имеет два решения:* $x_1 = \frac{-b+\sqrt{D}}{2a}$, $x_2 = \frac{-b-\sqrt{D}}{2a}$;

– *если* $D = 0$, *имеет единственное решение:* $x_1 = -\frac{b}{2a}$;

– *если* $D < 0$, *не имеет действительных решений.*

Следовательно, нам необходимо вычислить дискриминант (заведем для него вещественную переменную **d** типа **real**) и в зависимости от его значения организовать ветвления. Сначала нужно проверить, имеет ли уравнение действительные решения (для решений заведем переменные **x1** и **x2** типа **real**). Если да, и если дискриминант не равен нулю, то вычисляем оба решения по формулам, а если дискриминант равен нулю, то вычисляем единственное решение. Если же действительных решений нет, выводим текстовое сообщение об этом. Основной алгоритм можно проиллюстрировать следующей блок-схемой:



Три нерасшифрованных блока представляют собой стандартные операторы вывода. Разберем их подробнее:

1) При выводе двух корней выражение будет выглядеть следующим образом:
```
x1 := (-b + sqrt(d)) / 2 * a;
x2 := (-b - sqrt(d)) / 2 * a;
writeln('x1 = ', x1:4:2, ', x2 = ', x2:4:2);
```
При этом выводимое выражение будет выглядеть так: `'x1 = m, x2 = n'`, где синим цветом выделены однозначные текстовые константы, которые берутся из списка аргументов **writeln**, красным – вычисленные значения **x1** и **x2**. Причем корни выведены в форматированном виде: число после первого двоеточия задает ширину поля вывода для переменной вместе с точкой (при нехватке поля она будет расширено программой), а число после второго двоеточия – количество выводимых дробных знаков (его при работе программы изменить нельзя);

2) При выводе одного корня – все то же самое, только выводится один корень:
```
x1 := -(b / 2 * a);
writeln('x = ', x1:4:2);
```

3) При отсутствии действительных корней выводим сообщение:
```
writeln('No real solutions!');
```

В итоге внутренний условный оператор с телом включительно будет выглядеть так:

```
if d <> 0 then begin
  x1 := (-b + sqrt(d)) / 2 * a;
  x2 := (-b - sqrt(d)) / 2 * a;
  writeln('x1 = ', x1:4:2, ', x2 = ', x2:4:2)
end
else begin
  x1 := -(b / 2 * a);
```

```
    writeln('x = ', x1:4:2)
  end;
```

Код:

```
 1. program QuadraticEquation;
 2.
 3. var
 4.   a, b, c, d, x1, x2: real;
 5.
 6. begin
 7.   readln(a, b, c);
 8.   d := b * b - 4 * a * c;
 9.   if d >= 0 then begin
10.     if d <> 0 then begin
11.       x1 := (-b + sqrt(d)) / 2 * a;
12.       x2 := (-b - sqrt(d)) / 2 * a;
13.       writeln('x1 = ', x1:4:2, ', x2 = ', x2:4:2)
14.     end
15.     else begin
16.       x1 := -(b / 2 * a);
17.       writeln('x = ', x1:4:2)
18.     end
19.   end
20.   else begin
21.     writeln('No real solutions!');
22.   end
23. end.
```

Глава 3. Циклы

Задача № 13. Вывести на экран все натуральные числа до заданного

Формулировка. Дано натуральное число. Вывести на экран все натуральные числа до заданного включительно.

Решение. Данная задача решается с использованием оператора цикла **for**. Напомним, что с помощью цикла **for** можно совершить заданное количество итераций (повторений) некоторых операторов, которые синтаксически заключены в содержимое его тела (так называемого тела цикла). При этом некоторая целочисленная переменная изменяется от некоторого стартового значения до некоторого конечного (оба значения включительно), увеличиваясь на единицу с каждым повторением тела цикла.

Так как нам необходимо выводить натуральные числа, это означает, что вывод должен всегда начинаться с единицы, и при этом выводятся все следующие за ней натуральные числа до тех пор, пока значение переменной цикла (обычно используют переменную **i**) не достигнет конечного **n** (на последнем шаге значение переменной цикла будет равно **n**). После этого цикл завершится, и будут выполнены те операторы, которые следуют непосредственно за ним. Кстати, не стоит забывать, что после выхода из цикла **for** его переменная цикла считается неопределенной!

Код:

```
1. program FromOneToN;
2.
3. var
```

```
 4.   i, n: word;
 5.
 6. begin
 7.   readln(n);
 8.   for i := 1 to n do begin
 9.     write(i, ' ')
10.   end
11. end.
```

Пусть введено число 5, например. При входе **i** станет равно 1 и будет проверено существование отрезка в заданных границах. Так как 1 меньше 5, то произойдет вход в цикл, и будут выполняться следующие команды, пока **i** не превысит **n**:

1) Выполнение команд в теле цикла;
2) Увеличение **i** на 1;
3) Возвращение на шаг 1.

Нетрудно понять, что в нашем случае **i** будет принимать значения 1, 2, 3, 4, 5 и будет выведена на экран строка '1 2 3 4 5 '. Здесь красным цветом выделены изменяющиеся значения переменной цикла, а синим – выводящаяся неизменной пробельная константа.

Задача № 14. Найти наибольший нетривиальный делитель натурального числа

Формулировка. Дано натуральное число. Найти его наибольший нетривиальный делитель или вывести единицу, если такового нет.

Примечание 1: делителем натурального числа **a** называется натуральное число **b**, на которое **a** делится без остатка. То есть выражение «**b** – делитель **a**» означает: **a / b = k**, причем **k** – натуральное число.

Примечание: нетривиальным делителем называется делитель, который отличен от 1 и от самого числа (так как на единицу и само на себя делится любое натуральное число).

Решение. Пусть ввод с клавиатуры осуществляется в переменную **n**. Попробуем решить задачу перебором чисел. Для этого возьмем число на единицу меньшее **n** и проверим, делится ли **n** на него. Если да, то выводим результат и выходим из цикла с помощью оператора **break**. Если нет, то снова уменьшаем число на 1 и продолжаем проверку. Если у числа нет нетривиальных делителей, то на каком-то шаге проверка дойдет до единицы, на которую число гарантированно поделится, после чего будет выдан соответствующий условию ответ.

Хотя, если говорить точнее, следовало бы начать проверку с числа, равного **n div 2** (чтобы отбросить дробную часть при делении, если **n** нечетно), так как ни одно натуральное число не имеет делителей больших, чем половина это этого числа. В противном случае частное от деления должно быть натуральным числом между 1 и 2, которого просто не существует.

Данная задача также решается через **for**, но через другую его разновидность, и теперь счетчик будет убывать от **n div 2** до 1. Для этого **do** заменится на **downto**, при позиции начального и конечного значений остаются теми же.

Алгоритм на естественном языке:

1) Ввод **n**;
2) Запуск цикла, при котором **i** изменяется от **n div 2** до 1. В цикле:
    1. Если **n** делится на **i** (то есть, остаток от деления числа **n** на **i** равен 0), то выводим **i** на экран и выходим из цикла с помощью **break**.

Код:

```
 1. program GreatestDiv;
 2.
 3. var
 4.   i, n: word;
 5.
 6. begin
 7.   readln(n);
 8.   for i := n div 2 downto 1 do begin
 9.     if n mod i = 0 then begin
10.       writeln(i);
11.       break
12.     end
13.   end
14. end.
```

Кстати, у оператора ветвления **if** в цикле отсутствует **else**-блок. Такой условный оператор называется оператором ветвления с одной ветвью.

Задача № 15. Найти наименьший нетривиальный делитель натурального числа

Формулировка. Дано натуральное число. Найти его наименьший нетривиальный делитель или вывести само это число, если такового нет.

Решение. Задача решается аналогично предыдущей. При этом необходимо начать обычный цикл с увеличением, при котором переменная цикла **i** изменяется от 2 до **n** (такая верхняя граница нужна для того, чтобы цикл всегда заканчивался, так как когда **i** будет равно **n**, выполнится условие **n mod i = 0**). Весь остальной код при этом не отличается.

Код:

```
 1. program SmallestDiv;
 2.
 3. var
 4.   i, n: word;
 5.
 6. begin
 7.   readln(n);
 8.   for i := 2 to n do begin
 9.     if n mod i = 0 then begin
10.       writeln(i);
11.       break
12.     end
13.   end
14. end.
```

Задача № 16. Подсчитать общее число делителей натурального числа

Формулировка. Дано натуральное число. Подсчитать общее количество его делителей.

Решение. Задача достаточно похожа на две предыдущие. В ней также необходимо провести перебор в цикле некоторого количества натуральных чисел на предмет обнаружения делителей **n**, но при этом необходимо найти не первый из них с какого-либо конца отрезка [1, **n**] (это отрезок, содержащий все числа от 1 до **n** включительно), а посчитать их. Это можно сделать с помощью

счетчика **count**, который нужно обнулить непосредственно перед входом в цикл. Затем в условном операторе **if** в случае истинности условия делимости числа **n** (**n mod i = 0**) нужно увеличивать счетчик **count** на единицу (это удобно делать с помощью оператора **inc**).

Алгоритм на естественном языке:

1) Ввод **n**;
2) Обнуление переменной **count** (в силу необходимости работать с ее значением без предварительного присваивания ей какого-либо числа)
3) Запуск цикла, при котором **i** изменяется от 1 до **n**. В цикле:
    1. Если **n** делится на **i** (то есть, остаток от деления числа **n** на **i** равен 0), то увеличиваем значение переменной **count** на 1;
4) Вывод на экран значения переменной **count**.

Код:

```
 1. program CountDiv;
 2.
 3. var
 4.   i, n, count: word;
 5.
 6. begin
 7.   readln(n);
 8.   count := 0;
 9.   for i := 1 to n do begin
10.     if n mod i = 0 then inc(count)
11.   end;
12.   writeln(count)
13. end.
```

Задача № 17. Проверить, является ли заданное натуральное число простым

Формулировка. Дано натуральное число. Проверить, является ли оно простым.

Примечание: простым называется натуральное число, которое имеет ровно два различных натуральных делителя: единицу и само это число.

Решение. Задача отличается от предыдущей только тем, что вместо вывода на экран числа делителей, содержащегося в переменной **count**, необходимо выполнить проверку равенства счетчика числу 2. Если у числа найдено всего два делителя, то оно простое и нужно вывести положительный ответ, в противном случае – отрицательный ответ. А проверку через условный оператор, как мы уже знаем, можно заменить на вывод результата самого булевского выражения с помощью оператора **write** (**writeln**).

Код:

```
 1. program PrimeTest;
 2.
 3. var
 4.   i, n, count: word;
 5.
 6. begin
 7.   readln(n);
 8.   count := 0;
```

```
 9.   for i := 1 to n do begin
10.     if n mod i = 0 then inc(count)
11.   end;
12.   writeln(count = 2)
13. end.
```

Задача № 18. Вывести на экран все простые числа до заданного

Формулировка. Дано натуральное число. Вывести на экран все простые числа до заданного включительно.

Решение. В решении данной задачи используется решение предыдущей.

Нам необходимо произвести тест простоты числа, который был описан в решении предыдущей задачи следующим кодом (обозначим его как **код 1**):

```
count := 0;
for i := 1 to n do begin
  if n mod i = 0 then inc(count)
end;
writeln(count = 2);
```

Здесь **n** – проверяемое на простоту число. Напомним, что данный фрагмент кода в цикле проверяет, делится ли **n** на каждое **i** в отрезке от 1 до самого **n**, и если **n** делится на **i**, то увеличивает счетчик **count** на 1. Когда цикл заканчивается, на экран выводится результат проверки равенства счетчика числу 2.

В нашем же случае нужно провести проверку на простоту всех натуральных чисел от 1 до заданного числа (обозначим его как **n**). Следовательно, мы должны поместить **код 1** в цикл по всем **k** от 1 до **n**. Также в связи с этим необходимо заменить в **коде 1** идентификатор **n** на **k**, так как в данном решении проверяются на простоту все числа **k**. Кроме того, теперь вместо вывода ответа о подтверждении/опровержении простоты **k** необходимо вывести само это число в случае простоты:

```
if count = 2 then write(k, ' ');
```

Обобщая вышесказанное, приведем алгоритм на естественном языке:

1) Ввод **n**;
2) Запуск цикла, при котором **k** изменяется от 1 до **n**. В цикле:
    1. Обнуление переменной **count**;
    2. Запуск цикла, при котором **i** изменяется от 1 до **k**. В цикле:
        a. Если **k** делится на **i**, то увеличиваем значение переменной **count** на 1;
    3. Если **count** = 2, выводим на экран число **k** и символ пробела.

Код:

```
 1. program PrimesToN;
 2.
 3. var
 4.   i, k, n, count: word;
 5.
 6. begin
 7.   readln(n);
 8.   for k := 1 to n do begin
 9.     count := 0;
```

```
10.     for i := 1 to k do begin
11.       if k mod i = 0 then inc(count)
12.     end;
13.     if count = 2 then write(k, ' ')
14.   end
15. end.
```

Вычислительная сложность. В данной задаче мы впервые столкнулись с вложенным циклом (когда один цикл запускается внутри другого). Это означает, что наш алгоритм имеет нелинейную сложность (при которой количество выполненных операций равно размерности исходных данных (в нашем случае это **n** – количество чисел) плюс некоторое количество обязательных операторов).

Давайте посчитаем количество выполненных операций в частном случае. Итак, пусть необходимо вывести все натуральные простые числа до числа 5. Очевидно, что проверка числа 1 пройдет в 1 + 2 шага, числа 2 – в 2 + 2 шага, числа 3 – в 3 + 2 шага и т. д. (прибавленная двойка к каждому числу – два обязательных оператора вне внутреннего цикла), так как мы проверяем делители во всем отрезке от 1 до проверяемого числа. В итоге количество проведенных операций (выполненных операторов на низшем уровне) представляет собой сумму: 3 + 4 + 5 + 6 + 7 (также опущен обязательный оператор ввода вне главного (внешнего) цикла). Очевидно, что при выводе всех простых чисел от 1 до **n** приведенная ранее сумма будет такой:

$$1 + 3 + 5 + 6 + ... + (\mathbf{n} - 1) + \mathbf{n} + (\mathbf{n} + 1) + (\mathbf{n} + 2),$$

где вместо многоточия нужно дописать все недостающие члены суммы. Очевидно, что это сумма первых (**n** + 2) членов арифметической прогрессии с вычтенным из нее числом 2.

Как известно, сумма **k** первых членов арифметической прогрессии выражена формулой:

$$S_k = \frac{a_1 + a_k}{2} k,$$

где $\mathbf{a_1}$ – первый член прогрессии, $\mathbf{a_k}$ – последний.

Подставляя в эту формулу наши исходные данные и учитывая недостающее до полной суммы число 2, получаем следующее выражение:

$$S_n = \frac{1 + n+2}{2}(n+2) - 2 = \frac{n+2 + (n+2)^2}{2} - \frac{4}{2} = \frac{n+2+n^2+4n+4-4}{2} = \frac{n^2+5n+2}{2} = \frac{1}{2}n^2 + \frac{5}{2}n + 1$$

Чтобы найти асимптотическую сложность алгоритма, отбросим коэффициенты при переменных и слагаемые с низшими степенями (оставив, соответственно, слагаемое с самой высокой степенью). При этом у нас остается член $\boldsymbol{n^2}$, значит, асимптотическая сложность алгоритма – $\mathbf{O(n^2)}$.

Конечно, в дальнейшем мы не будем так подробно находить асимптотическую сложность алгоритмов, а тем более, вычислять количество требуемых операций, что интересно только теоретически. На самом деле, конечно, нас интересует лишь порядок роста времени работы алгоритма (количества необходимых операций), который можно выявить из анализа вложенности циклов и некоторых других характеристик.

Задача № 19. Вывести на экран первых n простых чисел

Формулировка. Дано натуральное число **n**. Вывести на экран **n** первых простых чисел. Например, при вводе числа 10 программа должна вывести ответ: 2 3 5 7 11 13 17 19 23 29

Решение. Эта задача похожа на предыдущую тем, что здесь нам тоже необходимо проверить все натуральные числа на некотором отрезке, на этот раз используя еще один счетчик для подсчета найденных простых. Однако на этот раз мы не можем узнать, сколько придется проверить чи-

сел, пока не насчитается **n** простых. Следовательно, здесь нам не подходит цикл **for**, так как мы не можем посчитать необходимое количество итераций.

Здесь нам поможет цикл **while**. Напомним, что тело цикла **while** повторяется до тех пор, пока остается истинным булевское выражение в его заголовке (поэтому его еще называют циклом с предусловием). Так как **while** не имеет переменной цикла, которая увеличивается на 1 с каждым следующим шагом, то при его использовании необходимо обеспечить изменение некоторых переменных в теле цикла, от которых зависит условие в его заголовке, таким образом, чтобы при достижении требуемого результата оно стало ложным.

Так как мы должны найти и вывести **n** первых простых чисел, подсчитывая их, и с каждый найденным простым числом некоторый счетчик (пусть это будет **primes** с начальным значением 0) увеличивается на 1, то условием в заголовке **while** будет выражение **primes < n**. Иными словами, цикл продолжается, пока число найденных чисел меньше требуемого. На последнем же шаге будет выведено последнее число, **primes** станет равным **n** и следующего шага цикла уже не будет, так как условие в его заголовке станет ложным. На этом работа программы будет закончена.

Проверяться на простоту будет некоторое число **k**, которое с **каждой** итерацией цикла обязательно будет увеличиваться на 1 (таким образом, мы будем двигаться по отрезку натуральных чисел, пока не выберем из него заданное количество простых).

Алгоритм на естественном языке:

1) Ввод **n**;
2) Обнуление переменной **primes**;
3) Присваивание переменной **k** значения 1;
4) Запуск цикла с предусловием **primes < n**. В цикле:
    1. Обнуление переменной **count**;
    2. Запуск цикла, при котором **i** изменяется от 1 до **k**. В цикле:

        а. Если **k** делится на **i**, то увеличиваем значение переменной **count** на 1;
    3. Если **count** = 2, выводим на экран число **k**, символ пробела и увеличиваем значение счетчика **primes** на 1;
    4. Увеличиваем значение **k** на 1.

Код:

```
 1. program FirstNPrimes;
 2.
 3. var
 4.   i, k, n, count, primes: word;
 5.
 6. begin
 7.   readln(n);
 8.   k := 1;
 9.   primes := 0;
10.   while primes < n do begin
11.     count := 0;
12.     for i := 1 to k do begin
13.       if k mod i = 0 then inc(count)
14.     end;
15.     if count = 2 then begin
16.       write(k, ' ');
17.       inc(primes)
```

```
18.      end;
19.      inc(k)
20.    end
21. end.
```

Выполним ручную прокрутку алгоритма, взяв в качестве **n** число 2. При этом будем уже по привычке красным цветом обозначать переменные, изменившиеся после выполнения данной строки, а прочерком те, которые на данном шаге не определены, так как алгоритм до них еще «не дошел». При повторении шагов цикла итерации явно считать не будем (хотя легко увидеть, что их номерам полностью соответствует изменяющаяся после каждого очередного выполнения тела переменная **k**), и в таблице будет указана лишь та строка, которая выполняется. На тех шагах, на которых переменные не изменяются, будем пояснять смысл выполняющихся операторов.

Для наглядности все же отделим друг от друга четные и нечетные шаги основного цикла **while**, при этом его внутренний цикл будем считать самоочевидным и в строке 12-14 будем фиксировать те значения переменных, которые будут получены по выходу из него.

№ строки	n	k	primes	i	count
7	2	—	—	—	—
8	2	1	—	—	—
9	2	1	0	—	—
10	**(primes < n) = true** – входим в цикл				
11	2	1	0	—	0
12-14	2	1	0	от 1 до 1	1
15-18	**(count = 2) = false**				
19	2	2	0	—	1
10	**(primes < n) = true** – входим в цикл				
11	2	2	0	—	0
12-14	2	2	0	от 1 до 2	2
15	**(count = 2) = true**				
16	Вывод числа **k** (то есть 2)				
17-18	2	2	1	—	2
19	2	3	1	—	2
10	**(primes < n) = true** – входим в цикл				
11	2	3	1	—	0
12-14	2	3	1	от 1 до 3	2

15	**(count = 2) = true**				
16	Вывод числа **k** (то есть 3)				
17-18	2	3	2	—	2
19	2	4	2	—	2
10	**(primes < n) = false** – программа завершена				

Как видим, описать действия даже такого элементарного алгоритма и даже при столь малых исходных данных – достаточно трудоемкая и трудно проверяемая задача, поэтому очень важно понимать логику самой программы и уметь представлять себе целостную картину ее поведения с минимальными потребностями в моделировании.

Вычислительная сложность. Так как в данной задаче в главном цикле присутствует вложенный, в котором происходит ровно **k** операций (сложность этой конструкции мы определили в **задаче 18**), то его сложность явно не меньше **O(n^2)** и превышает ее, так как нам явно необходимо выполнить более чем **n** шагов главного цикла. При этом точность расчета сложности зависит от распределения простых чисел, которое описывается с помощью достаточно сложных математических приемов и утверждений, так что мы не будем далее рассматривать эту задачу.

Задача № 20. Проверить, является ли заданное натуральное число совершенным

Формулировка. Дано натуральное число. Проверить, является ли оно совершенным.

Примечание: совершенным числом называется натуральное число, равное сумме всех своих собственных делителей (то есть натуральных делителей, отличных от самого числа). Например, 6 – совершенное число, оно имеет три собственных делителя: 1, 2, 3, и их сумма равна 1 + 2 + 3 = 6.

Решение. Эта задача напоминает **задачу 16**, в которой нужно было найти количество всех натуральных делителей заданного числа. Напомним код ее основной части (назовем его **кодом 1**):

```
count := 0;
for i := 1 to n do begin
  if n mod i = 0 then inc(count)
end;
```

Как видно, в этом цикле проверяется делимость числа **n** на все числа от 1 до **n**, причем при каждом выполнении условия делимости увеличивается на 1 значение счетчика **count** с помощью функции **inc**. Чтобы переделать этот код под текущую задачу, нужно вместо инкрементации (увеличения значения) переменной-счетчика прибавлять числовые значения самих делителей к некоторой переменной для хранения суммы (обычно ее мнемонически называют **sum**, что в пер. с англ. означает «сумма»). В связи с этим оператор

```
    if n mod i = 0 then inc(count);
```

в **коде 1** теперь уже будет выглядеть так:

```
    if n mod i = 0 then sum := sum + i;
```

Кроме того, чтобы не учитывалось само число **n** при суммировании его делителей (насколько мы помним, этот делитель не учитывается в рамках определения совершенного числа), цикл должен продолжаться не до **n**, а до **n – 1**. Правда, если говорить точнее, то цикл следовало бы проводить до **n div 2** (также это обсуждалось в **задаче 14**), так как любое число **n** не может иметь больших делителей, иначе частное от деления должно быть несуществующим натуральным число между 1 и 2.

Единственное, что останется теперь сделать – это вывести ответ, сравнив число **n** с суммой его делителей **sum** как результат булевского выражения через **writeln**:

```
    writeln(n = sum);
```

Код:

```
 1. program PerfectNumbers;
 2.
 3. var
 4.   i, n, count: word;
 5.
 6. begin
 7.   readln(n);
 8.   count := 0;
 9.   for i := 1 to n div 2 do begin
10.       if n mod i = 0 then sum := sum + i
11.   end;
12.   writeln(n = sum)
13. end.
```

Задача № 21. Проверить, являются ли два натуральных числа дружественными

Формулировка. Даны два натуральных числа. Проверить, являются ли они дружественными.

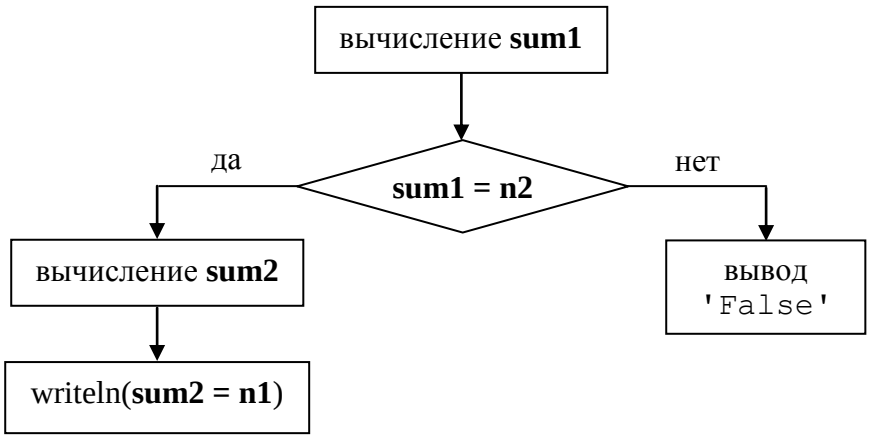
Примечание: дружественными числами называются два различных натуральных числа, для которых сумма всех собственных делителей первого числа равна второму числу и сумма всех собственных делителей второго числа равна первому числу.

Например, 220 и 284 – пара дружественных чисел, потому что:
Сумма собственных делителей 220: 1 + 2 + 4 + 5 + 10 + 11 + 20 + 22 + 44 + 55 + 110 = 284
Сумма собственных делителей 284: 1 + 2 + 4 + 71 + 142 = 220

Решение. Эта задача напоминает **задачу 19**, так как в ней мы тоже считали сумму собственных делителей введенного числа, а затем сравнивали эту сумму с самим числом, проверяя его на предмет совершенности. В данном же случае нам нужно найти не только сумму собственных делителей первого числа (обозначим число как **n1**, а сумму его делителей **sum1**), но и второго числа (возьмем обозначения **n2** и **sum2** соответственно). Тогда ответом в задаче послужит сравнение: **(n1 = sum2) and (n2 = sum1)**. Кстати, здесь впервые в нашем повествовании мы используем логические операции (напомним, что логическое выражение **X1 and X2** принимает значение истины тогда и только тогда, когда истинны булевские выражения **X1** и **X2**, а в остальных случаях оно принимает ложное значение).

Однако предложенную схему можно упростить. Покажем это на примере: пусть даны числа 8 и 4. Считаем сумму собственных делителей числа 8: 1 + 2 + 4 = 7. Это число отлично от 4, поэтому пара уже не соответствует определению дружественных чисел. Можно сразу вывести отрицательный ответ, избежав подсчета суммы делителей второго числа. Если были бы даны числа 8 и 7, то необходимо было бы вычислить сумму собственных делителей числа 7, она равна 1 (так как оно простое). Теперь необходимо сравнить сумму собственных делителей второго с первым числом: так как 1 отлично от 8, числа не дружественные.

Покажем на блок-схеме, как можно разветвить программу (вычисление обоих сумм не изображается):



Таким образом, без логических операций можно и обойтись.

Код:

```
 1. program AmicableTest;
 2.
 3. var
 4.   i, n1, n2, sum1, sum2: word;
 5.
 6. begin
 7.   readln(n1, n2);
 8.   for i := 1 to n1 div 2 do begin
 9.     if n1 mod i = 0 then sum1 := sum1 + i
10.   end;
11.   if sum1 = n2 then begin
12.     for i := 1 to n2 div 2 do begin
13.       if n2 mod i = 0 then sum2 := sum2 + i
14.     end;
15.     writeln(sum2 = n1)
16.   end
17.   else begin
18.     writeln('False')
19.   end
20. end.
```

Задача № 22. Найти наибольший общий делитель двух натуральных чисел

Формулировка. Даны два натуральных числа. Найти их наибольший общий делитель.

Примечание: наибольшим общим делителем (сокращенно пишут НОД) двух натуральных чисел **m** и **n** называется наибольший из их общих делителей. Обозначение: НОД(m, n).

Примечание 2: общим делителем двух натуральных чисел называется натуральное число, на которое натуральное число, которое является делителем обоих этих чисел.

Например, найдем НОД(12, 8):

Выпишем все делители числа 12: 1, 2, 3, 4, 6, 12;
Выпишем все делители числа 8: 1, 2, 4, 8;
Выпишем все общие делители чисел 12 и 8: 1, 2, 4. Из них наибольшее число – 4. Это и есть НОД чисел 12 и 8.

Решение. Конечно, при решении мы не будем выписывать делители и выбирать нужный. В принципе, ее можно было бы решить как **задачу 14**, начав цикл с наименьшего из двух заданных

чисел (так как оно тоже может быть НОД, например, НОД(12, 4) = 4). Но мы воспользуемся так называемым алгоритмом Евклида нахождения НОД, который выведен с помощью математических методов. В самом простейшем случае для заданных чисел **m** и **n** он выглядит так:

> *1) Если **m** неравно **n**, перейти к шагу 2, в противном случае вывести **m** и закончить алгоритм;*
>
> *2) Если **m** > **n**, заменить **m** на **m – n**, в противном случае заменить **n** на **n – m;***
>
> *3) Перейти на шаг 1*

Как видим, в шаге 2 большее из двух текущих чисел заменяется разностью большего и меньшего.

Приведем пример для чисел 12 и 8:

a. Так как 12 > 8, заменим 12 на 12 – 8 = 4;

b. Так как 8 > 4, заменим 8 на 8 – 4 = 4;

c. 4 = 4, конец.

Не проводя каких-либо рассуждений над алгоритмом и не доказывая его корректности, переведем его описание в более близкую для языка **Pascal** форму:

Алгоритм на естественном языке:

1) Ввод **m** и **n**;
2) Запуск цикла с предусловием **m** < > **n**. В цикле:
    1. Если **m** > **n**, то переменной **m** присвоить **m – n**, иначе переменной **n** присвоить **n – m**;
3) Вывод **m**:

Код:

```
 1. program GreatestCommonDiv;
 2.
 3. var
 4.   m, n: word;
 5.
 6. begin
 7.   readln(m, n);
 8.   while m <> n do begin
 9.     if m > n then begin
10.       m := m - n
11.     end
12.     else begin
13.       n := n - m
14.     end
15.   end;
16.   writeln(m)
17. end.
```

Задача № 23. Найти наименьшее общее кратное двух натуральных чисел

Формулировка. Даны два натуральных числа. Найти их наименьшее общее кратное.

Примечание: наименьшим общим кратным двух чисел **m** и **n** называется наименьшее натуральное число, которое делится на **m** и **n**. Обозначение: НОК(m, n)

Решение. Из теории чисел известно, что НОК(m, n) связан с НОД(m, n) следующим образом:

$$\text{НОК}\ m,n = \frac{m \cdot n}{\text{НОД}\ m,n}$$

Следовательно, для нахождения ответа нам нужно лишь использовать предыдущую задачу нахождения НОД двух чисел **m** и **n**:

```
while m <> n do begin
  if m > n then begin
    m := m - n
  end
  else begin
    n := n - m
  end
end;
```

Так как исходные переменные будут испорчены в процессе работы алгоритма Евклида, нам нужно вычислить их произведение до входа в описанный выше цикл и присвоить это произведение переменной **prod** (от англ. *product* – «произведение»):

```
prod := m * n;
```

После этого нам остается вывести на экран результат арифметического выражения в правой части нашей формулы. В качестве самого НОД будет использоваться переменная **m**:

```
writeln(prod div m);
```

Кстати, деление в формуле будет целочисленным (через **div**) именно потому, что если два числа делятся на некоторое число, то и их произведение также делится на него.

Код:

```
 1. program LeastCommonMult;
 2.
 3. var
 4.   m, n, prod: word;
 5.
 6. begin
 7.   readln(m, n);
 8.   prod := m * n;
 9.   while m <> n do begin
10.     if m > n then begin
11.       m := m - n
12.     end
13.     else begin
14.       n := n - m
15.     end
16.   end;
17.   writeln(prod div m)
18. end.
```

Задача № 24. Вычислить x^n

Формулировка. Даны натуральные числа **x** и **n** (которое также может быть равно 0). Вычислить x^n.

Решение. Для того чтобы решить эту задачу, вспомним определение степени с натуральным показателем: запись x^n означает, что число **x** умножено само на себя **n** раз.

Сразу из определения видно, что здесь заранее известно количество повторений при вычислении результата, так что задача легко решается через цикл **for**. Выходит, мы копируем исходное

число **x** в некоторую переменную **res** (от англ. *result* – «результат»), а затем просто умножаем его на **x n** раз? Не стоит торопиться с ответом.

Рассмотрим пример: $3^4 = 3 * 3 * 3 * 3 = 81$. Если посмотреть на эту запись, то мы видим, что возведение в четвертую степень как выражение содержит четыре слагаемых, но только три операции, так как мы с первого шага домножаем число 3 на три тройки. Тогда реализация идеи из абзаца выше будет давать число в степени на 1 больше, чем требуется.

Какой можно придумать выход? Например, можно сократить цикл на одну операцию, но что тогда будет при вычислении нулевой степени? Как известно, любое число в нулевой степени дает 1, а здесь при вводе в качестве **n** нуля приведет к тому, что не будет осуществлен вход в цикл (так как не существует целочисленного отрезка от 1 до 0) и в итоге на выход так и пойдет исходное число **x.**

А что, если изменить схему умножения так: $3^4 = 1 * 3 * 3 * 3 * 3 = 81$? Так мы можем сравнять показатель степени и число требуемых операций, да и с нулевой степенью все становится просто, так как при вводе в качестве **n** нуля не будет осуществляться вход в цикл и на выход в программе пойдет число 1!

Теперь алгоритм на естественном языке:

1) Ввод **x** и **n**;
2) Присваивание переменной **res** числа 1;
3) Запуск цикла, при котором **i** изменяется от 1 до **n**. В цикле:
    1. Присваиваем переменной **res** значение **res** * **x**;
4) Вывод переменной **res**.

Код:

```
 1. program Exponentiation;
 2.
 3. var
 4.   x, n, i, res: word;
 5.
 6. begin
 7.   readln(x, n);
 8.   res := 1;
 9.   for i := 1 to n do begin
10.     res := res * x
11.   end;
12.   writeln(res)
13. end.
```

Кстати, стоит понимать, что объявление переменной **res** при использовании типа **word** достаточно условно, так как этот тип принимает значения от 0 до 65535, что на единицу меньше числа 256^2, хотя вводить в программу можно числа, предполагающие возведение в более высокую степень. Так как в условии задачи не сказано ничего о том, в каком числовом промежутке по **x** и **n** она должна выдавать корректный ответ, оставим это в таком виде, достаточном для проверки приложения на работоспособность.

Задача № 25. Вычислить x^n по алгоритму быстрого возведения в степень

Формулировка. Даны натуральные числа **x** и **n**. Вычислить x^n, используя алгоритм быстрого возведения в степень:

$$x^n = \begin{cases} x^{2^{\,n \text{ div } 2}}, & \text{если } n \text{ четно} \\ x \cdot x^{2^{\,n \text{ div } 2}}, & \text{если } n \text{ нечетно} \end{cases}$$

Решение. Разберем этот пока неясный алгоритм, представленный в нашей задаче лишь единственной формулой. Легко понять, что указанное равенство имеет место: например, $3^4 = 3^{4 \text{ div } 2} = (3^2)^2 = 9^2 = 81$. Другой пример: $4^5 = 4 * (4^2)^{5 \text{ div } 2} = 4 * (4^2)^2 = 4 * 16^2 = 4 * 256 = 1024$. Причем если выражение со степенью нельзя в один шаг преобразовать таким образом, то необходимо продолжить преобразование выражения вплоть до получения конечного ответа. Однако как теперь реализовать эту схему?

Рассмотрим пример немного сложнее. Вычислим $4^7 = 4 * (4^2)^3 = 4 * (16)^3 = 4 * 16 * (16^2)^1 = 4 * 16 * 256 = 16384$. Стоит обратить внимание на то, что на каждом шаге при вычислении изменяется только единственное обрабатывающееся число, а остальные множители выносятся однократно и необходимы только для получения ответа в последнем действии.

Чтобы исключить их из рассмотрения, при нечетном текущем числе **n** мы будем домножать на них некоторую промежуточную переменную **r**, поначалу равную 1 (кстати, нечетность числа в **Pascal** определяет функция **odd**), затем (уже независимо от условий) возведем в квадрат текущее **x** и разделим **n** на 2 с отбрасыванием остатка. При повторении этих действий на некотором шаге **n** обязательно станет равным 1. Это происходит потому, что деление любого нечетного числа **a** на 2 с отбрасыванием остатка равносильно делению на двойку числа **(a – 1)**, которое является четным, и при делении, двигаясь по цепочке четных чисел, мы всегда придем к единице. Условие **n = 1** и будет окончанием цикла с предусловием. По выходе из цикла необходимо будет в качестве ответа вывести последнее значение **x**, умноженное на **r**.

Теперь алгоритм на естественном языке:

1) Ввод **x** и **n**;
2) Присваивание переменной **r** числа 1;
5) Запуск цикла с предусловием **n < > 1.** В цикле:
    1. Если **n** нечетно, домножаем **r** на **x**;
    2. Переменной **x** присваиваем значение **x * x**;
    3. Переменной **n** присваиваем результат от деления этой переменной на 2 с отбрасыванием остатка;
3) Вывод выражения **x * r**.

Код:

```
 1. program FastExponentiation;
 2.
 3. var
 4.   x, n, r: word;
 5.
 6. begin
 7.   readln(x, n);
 8.   r := 1;
 9.   while n <> 1 do begin
10.     if odd(n) then r := r * x;
11.     x := x * x;
12.     n := n div 2
13.   end;
14.   writeln(x * r)
```

```
15. end.
```

Из анализа данного алгоритма известно, что его асимптотическая сложность порядка $O(\log_2 n)$.

Задача № 26. Решить квадратное уравнение заданного вида с параметром

Формулировка. Дано натуральное число **n**. Вывести на экран решения всех квадратных уравнений вида $x^2 + 2ax - 3 = 0$ для всех **a** от 1 до **n**.

Решение. Эта задача очень похожа на **задачу 12**. В принципе, ее можно было бы решить, используя код этой задачи, взяв первый и последний коэффициенты равными 1 и –3 соответственно и запустив цикл по всем **a** от 1 до **n**, умножив **a** на 2 во всех формулах.

Однако исследуем это уравнение математически и попытаемся оптимизировать решение:

1) Найдем дискриминант уравнения: $D = 2a\ ^2 - 4 \cdot 1 \cdot -3\ = 4a^2 + 12 = 4\ a^2 + 3$

Очевидно, что найденная величина неотрицательна, и, если быть точнее, то при **a** от 1 до **n** она всегда принимает значение не меньше 16 (так как при **a** = 1 она равна 4 * (1 + 3) = 4 * 4 = 16). Следовательно, наше уравнение всегда имеет решение, причем их два.

2) Найдем формулы корней уравнения:

$$x_1 = \frac{-2a + \sqrt{4\ a^2 + 3}}{2} = \frac{-2a + \sqrt{4}\sqrt{a^2 + 3}}{2} = \frac{-2a + 2\sqrt{a^2 + 3}}{2} = -a + \sqrt{a^2 + 3},\ x_2 = -a - \sqrt{a^2 + 3}$$

Итак, формулы корней уравнения получены, и теперь только осталось вывести в цикле значения корней для всех **a** от 1 до **n**, не забыв сделать вывод форматированным (так как решения будут вещественными).

Код:

```
 1. program MyQuadraticEquation;
 2.
 3. var
 4.   a, n: word;
 5.   x1, x2: real;
 6.
 7. begin
 8.   readln(n);
 9.   for a := 1 to n do begin
10.     x1 := sqrt(a * a + 3) - a;
11.     x2 := -a - sqrt(a * a + 3);
12.     writeln('a = ', a, ', x1 = ', x1:4:2, ', x2 = ', x2:4:2)
13.   end
14. end.
```

Задача № 27. Вычислить значение многочлена в точке

Формулировка. Дано натуральное число **n**, вещественное число **x** и набор вещественных чисел $\mathbf{a_n}$, $\mathbf{a_{n-1}}$, ..., $\mathbf{a_0}$. Вычислить значение многочлена **n**-ной степени с коэффициентами $\mathbf{a_n}$, $\mathbf{a_{n-1}}$, ..., $\mathbf{a_0}$ от одной переменной в точке **x**.

Примечание: многочленом **n**-ной степени от одной переменной **x** называется выражение вида $a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0$, где $\mathbf{a_n}$, $\mathbf{a_{n-1}}$, ..., $\mathbf{a_0}$ – коэффициенты.

Решение. Собственно, в этой задаче требуется возведение переменной **x** (точнее, конкретного ее значения) в некоторую степень **n** – 1 раз, а также **n** операций умножения и **n** операций сложения. В принципе, для полноценного решения она не требует одновременного знания более чем одного коэффициента, так как мы можем в цикле ввести коэффициент $\mathbf{a_n}$ в переменную **a**, умножить его на $\mathbf{x^n}$ и прибавить полученное число к переменной результата **res** (которой перед входом в цикл должно быть присвоено значение 0) и повторить этот шаг для всех коэффициентов. Тогда количество операций: (1 + 2 + ... + **n** + 2**n**), что примерно соответствует асимптотической сложности $\mathbf{O(n^2)}$.

Не занимаясь реализацией этого алгоритма, давайте оптимизируем его. Например, пусть нам дан многочлен третьей степени $a_3x^3 + a_2x^2 + a_1x + a_0$. Вынесем за скобки общий множитель x при слагаемых, в которых это возможно. Получим: $(a_3x^2 + a_2x + a_1)x + a_0$. Вынесем за скобки x также и в полученном выражении со скобками, в итоге получим: $((a_3x + a_2)x + a_1)x + a_0$.

Полученную форму записи называют ***схемой Горнера***. Очевидно, что она существует для многочлена любой степени.

Итак, имея **n** = 3 и коэффициенты $\mathbf{a_3}$, $\mathbf{a_2}$, $\mathbf{a_1}$ и $\mathbf{a_0}$, мы можем посчитать его значение с помощью **n** операций умножения и **n** операций сложения, а это значит, что для вычисления требуется порядка **2n** операций и алгоритм имеет линейную асимптотическую сложность **O(n)**, что демонстрирует очевидное преимущество перед предыдущим решением.

Посмотрим, как может выглядеть цикл, в котором вычисляется значение многочлена в точке. Для этого немного изменим представление в виде схемы Горнера, не меняя при этом значения многочлена: $(((0x + a_3)x + a_2)x + a_1)x + a_0$.

Теперь нам требуется **n** + 1 операций, однако заведя переменную **res** для накопления результата, которая перед входом в цикл будет равна 0, мы можем, вводя коэффициенты $\mathbf{a_3}$, $\mathbf{a_2}$, $\mathbf{a_1}$ и $\mathbf{a_0}$, посчитать значение многочлена в одном цикле:

```
res := 0;
for i := 1 to n + 1 do begin
  read(a);
  res := res * x + a
end;
```

Примечание: оператор **read** нужен нам для того, чтобы можно было вводить коэффициенты через пробел, а не через **Enter**.

Теперь разберем сам цикл. На первом шаге мы получаем в **res** значение выражения $0x + a_3 = a_3$, на втором – $a_3x + a_2$, на третьем – $(a_3x + a_2)x + a_1$, на четвертом – $((a_3x + a_2)x + a_1)x + a_0$. Как видно, формула полностью восстановилась, причем вычисление идет от наиболее вложенных скобок к верхним уровням.

Код:

```
 1. program ValueOfPolynomial;
 2.
 3. var
 4.   i, n: byte;
 5.   x, a, res: real;
 6.
 7. begin
 8.   readln(n, x);
 9.   res := 0;
10.   for i := 1 to n + 1 do begin
11.     read(a);
12.     res := res * x + a
13.   end;
```

```
14.   writeln(res:4:2)
15. end.
```

Задача № 28. Вычислить факториал

Формулировка. Дано натуральное число **n** (которое также может быть равно нулю). Вычислить **n!**

Примечание: **n!** (факториал числа **n**, читается «*эн факториал*») – произведение всех натуральных чисел до **n** включительно.

Решение. Задача очень просто решается через цикл **for** по всем **i** от 1 до **n**, в теле которого мы на каждом шаге домножаем переменную-результат **fact** (которой до входа в цикл присвоено значение 1) на **i**. При этом сохраняется и правило **0! = 1**, так как при вводе нуля программа не войдет в цикл и на выход пойдет неизмененное в переменной **fact** число 1.

Код:

```
 1. program Factorial;
 2.
 3. var
 4.   i, n: byte;
 5.   fact: integer;
 6.
 7. begin
 8.   readln(n);
 9.   fact := 1;
10.   for i := 1 to n do begin
11.     fact := fact * i
12.   end;
13.   writeln(fact)
14. end.
```

Примечание: для накопления результата мы использовали переменную **fact** типа **integer**. Как уже говорилось, этот тип охватывает диапазон целых чисел от –2147483648 до 2147483647 (Borland Delphi 7 и PascalABC). Данная переменная позволит сформировать результаты вплоть до **12!** (= 479001600) включительно.

Задача № 29. Вычислить число сочетаний из n по k

Формулировка. Даны натуральные числа **n** и **k** (**k** не превышает **n**). Вычислить число сочетаний из **n** по **k**.

Примечание: в комбинаторике сочетанием из **n** по **k** называется набор **k** элементов, выбранных из данных **n** элементов; при этом наборы, отличающиеся только порядком следования элементов, считаются одинаковыми. Обозначение числа сочетаний из **n** по **k** элементов: C_n^k. При этом считается, что $C_n^n = 1$, $C_n^0 = 1$ и $C_0^0 = 1$ для любого натурального **n**.

Например, найдем все 2-элементные сочетания 3-элементного множества {1, 2, 3}. Таковыми являются {1, 2}, {1, 3} и {2, 3}. То есть, таковых сочетаний 3. При этом, например, {1, 2} и {2, 1} – одинаковые сочетания, так как они отличаются только порядком следования элементов.

Решение. Из комбинаторики известна формула:

$$C_n^k = \frac{n!}{k!\ n-k\ !} = \frac{n\ \ n-1\ \ \ n-2\ \cdot\ldots\cdot\ n-k+1}{k!}$$

Не интересуясь вопросом ее вывода и корректности, мы будем использовать тот ее вариант, который написан после второго знака равенства (если смотреть слева направо), так как он наиболее оптимален для вычислений и позволит обойтись двумя циклами (для числителя **for** с **downto**, для знаменателя – просто **for**). Для числителя и знаменателя предусмотрим соответственно переменные **num** (от англ. *numerator* – «числитель») и **denom** (от англ. *denominator* – «знаменатель»), которым нужно поначалу присвоить значения 1, чтобы осуществить контроль частных случаев (этот вопрос упомянут в предыдущей задаче):

1) При **k = 0** переменная **num** останется неизменной и будет равна 1, так как невозможен вход в цикл с уменьшением от **n** до **(n + 1)**, переменная **denom** будет равна 1 как **0!**;

2) При **n = k num** и **denom** будут вычислены и при делении дадут единицу;

3) При **n = k = 0** переменная **denom** будет вычислена как **0!**, а переменная **num** не изменится по невозможности входа в цикл с уменьшением от 0 до 1.

Код:

```
 1. program NumOfCombinations;
 2.
 3. var
 4.   i, n, k: byte;
 5.   num, denom: integer;
 6.
 7. begin
 8.   readln(n, k);
 9.   num := 1;
10.   for i := n downto n - k + 1 do begin
11.     num := num * i
12.   end;
13.   denom := 1;
14.   for i := 1 to k do begin
15.     denom := denom * i
16.   end;
17.   writeln(num div denom)
18. end.
```

Задача № 30. Вывести таблицу квадратов и кубов всех натуральных чисел до n

Формулировка. Дано натуральное число **n**, меньшее 256. Используя псевдографику, вывести на экран таблицу квадратов и кубов всех натуральных чисел от 1 до **n** включительно.

Примечание: псевдографика – это совокупность символов для формирования видимых графических примитивов (линий, прямоугольников, рамок, таблиц и т. д.). Она была актуальна в те далекие времена, когда устройства вывода компьютеров не способны были работать с графикой, либо это было проблематично.

Символы, использующиеся для псевдографики, должны быть включены в набор используемого в терминале (консоли) компьютерного шрифта.

Решение. В этой задаче мы впервые займемся графическим оформлением выходных данных программы. Для начала подумаем, как может выглядеть таблица в простейшем случае (**n** = 3):

x	x^2	x^3
1	1	1
2	4	8
3	9	27

Несмотря на то, что кодовые страницы для DOS имеют определенный набор символов для рисования графических примитивов, в частности, таблиц, мы будем пользоваться лишь символами `'-'` и `'|'` для построения линий таблицы, а также `'/'` и `'\'` для формирования ее угловых элементов.

Построим псевдографический эквивалент этой таблицы:

```
/----------------------\
|   x   |  x^2  |  x^3  |
|----------------------|
|   1   |   1   |   1   |
|   2   |   4   |   8   |
|   3   |   9   |  27   |
\----------------------/
```

Примечание: в случае ограниченных возможностей вывода для обозначения возведения выражения в степень используется постфикс «**^k**», где **k** – показатель степени. Кстати, здесь мы выравниваем значения в середине столбцов, сдвигая к середине разряд единиц упорядоченных по правому краю столбцов.

Как же сформировать вывод на экран такой таблицы? Понятно, что это нужно сделать построчно. Однако какой ширины сделать таблицу и как организовать вывод строк со степенями? Так как максимальное число, которое может быть подано на вход – 255, и его куб равен 16581375 (он состоит из 8 цифр), то нам нужно сделать колонки ширины 1 + 8 + 8 + 1 = 18 (крайние единицы для отступов) символов, чтобы таблица выглядела равномерно:

```
/------------------------------------------------------\
|         x          |        x^2         |        x^3         |
|------------------------------------------------------|
|         1          |         1          |         1          |
|         2          |         4          |         8          |
|       ...          |       ...          |       ...          |
|       255          |     65025          |  16581375          |
\------------------------------------------------------/
```

Как видим, при постепенном увеличении числа будут «вырастать» справа налево. Чтобы вывести такую строку, нужно вывести константу `'|'`, затем вывести соответствующее число с шириной поля вывода 9, потом вывести константу `'|'` с шириной поля вывода 10 и аналогично вывести оставшиеся колонки:

```
  writeln('|', i:9, '|':10, i * i:9, '|':10, i * i * i:9, '|':10);
```

Схематически с учетом форматирования это будет выглядеть так:

```
'|       255          |     65025          |  16581375          |'
```

Изменение цветов соответствует чередованию аргументов в операторе вывода.

Так как заголовок таблицы один и тот же для всех вариантов исходных данных, мы можем сразу вывести его с помощью трех строковых констант через **writeln**:

```
writeln('/------------------------------------------------------\');
writeln('|         x          |        x^2         |        x^3         |');
writeln('|------------------------------------------------------|');
```

После вывода всех строк нужно вывести нижнюю границу таблицы:

```
writeln('\------------------------------------------------------/');
```

Вообще, все эти константы и правила не взялись «просто так» или из расчетов. Единственный использованный факт – разрядность числа не более 8, поэтому мы и взяли ширину колонок «по максимуму». В остальном нужно было экспериментировать, чтобы найти наиболее легкое и наглядное решение. Конечно, псевдографика – это не алгоритмическое программирование, и в нем тестирование и эксперимент играют чуть ли не самую важную роль.

Код:

```
 1. program MyTable;
 2.
 3. var
 4.   i, n: byte;
 5.
 6. begin
 7.   readln(n);
 8.   writeln('/-----------------------------------------------
    -------\');
 9.   writeln('|        x         |       x^2        |       x^3
    |');
10.   writeln('|-----------------------------------------------
    -------|');
11.   for i := 1 to n do begin
12.     writeln('|', i:9, '|':10, i * i:9, '|':10, i * i * i:9,
    '|':10)
13.   end;
14.   writeln('\-----------------------------------------------
    -------/')
15.  end.
```

Задача № 31. Сформировать реверсную запись заданного числа

Формулировка. Дано натуральное число **n** заранее неизвестной разрядности. Сформировать и вывести на экран число, представляющее собой реверсную запись **n**.

Решение. Это более общий случай **задачи 4**, в которой при случае трехзначного **n** отчетливо видны повторяющиеся фрагменты кода. Попытаемся получить общий алгоритм решения через цикл.

Пусть дано число 25893. Возьмем его последнюю цифру как остаток от деления на 10 – это 3. Очевидно, она должна быть первой. Отбросим ее у числа **n** и возьмем последнюю цифру 9 – она должна быть второй. Чтобы сформировать две цифры реверсного числа, умножим 3 на 10 и прибавим 9, потом добавим третью цифру и т. д.

Так как разрядность числа неизвестна, мы будем использовать цикл с предусловием. Его тело будет выглядеть так:

```
r := r * 10;
r := r + n mod 10;
n := n div 10;
```

Поначалу результат **r** должен быть равен 0, и тогда умножение нуля на 10 в первом шаге не разрушает формирование реверсной записи, которое теперь может быть заключено в один цикл.

Каким же будет условие продолжения? Нетрудно понять, что когда мы будем добавлять последнюю оставшуюся цифру исходного числа **n** к реверсной записи **r**, мы умножим **r** на 10, прибавим к ней как **n mod 10** (в данном случае этот остаток равен **n**) и разделим **n** на 10. Тогда **n** станет равно 0 и цикл должен закончиться, так что условие его продолжения – **n < > 0.**

Код:

```
 1. program ReverseOfN;
 2.
 3. var
 4.   r, n: word;
 5.
 6. begin
 7.   readln(n);
 8.   r := 0;
 9.   while n <> 0 do begin
10.     r := r * 10;
11.     r := r + n mod 10;
12.     n := n div 10
13.   end;
14.   writeln(r)
15. end.
```

Задача № 32. Проверить монотонность последовательности цифр числа

Формулировка. Дано натуральное число **n**. Проверить, представляют его ли цифры его восьмеричной записи строго монотонную последовательность. При этом последовательность из одной цифры считать строго монотонной.

Примечание: в математике строго возрастающие и строго убывающие последовательности называются строго монотонными. В строго возрастающей последовательности каждый следующий член **больше** предыдущего. Например: 1, 3, 4 ,7, 11, 18. В строго убывающей последовательности каждый следующий член **меньше** предыдущего. Например: 9, 8, 5, 1.

Решение. Здесь нам нужно будет последовательно получить разряды восьмеричной записи числа, двигаясь по записи числа справа налево. Как мы уже знаем, последний разряд числа в восьмеричной системе счисления есть остаток от деления этого числа на 8.

Попытаемся определить несколько общих свойств строго возрастающих (обозначим пример как 1) и строго убывающих (обозначим как 2) последовательностей (для наглядности будем сразу брать восьмеричные последовательности):

1) 3, 4, 5, 8, 9, 11.

2) 8, 7, 3, 2, 0.

Для начала введем в рассмотрение некоторую формулу, обозначим ее как (I):

$$delta_i = a_i - a_{i+1},$$

где a_i – член заданной последовательности с индексом i. Нетрудно понять, что эта формула определяет разность между двумя соседними элементами: например, если $i = 5$ (то есть, мы рассматриваем пятую разность), то формула будет выглядеть так: $delta_5 = a_5 - a_6$. При этом стоит учитывать множество всех значений, которые может принимать i. Например, для последовательности (1) i может принимать значения от 1 до 5 включительно, для последовательности (2) – от 1 до 4 включительно. Легко проверить, что для всех других i формула (I) не имеет смысла, так как в ней должны участвовать несуществующие члены последовательности.

Найдем все $delta_i$ для последовательности (1): $delta_1 = 3 - 4 = -1$, $delta_2 = 4 - 5 = -1$, $delta_3 = 5 - 8 = -3$, $delta_4 = 8 - 9 = -1$, $delta_5 = 9 - 11 = -2$.

Как видим, они все отрицательны. Нетрудно догадаться, что это свойство сохраняется для всех строго возрастающих последовательностей.

Выпишем все $delta_i$ для последовательности (2), не расписывая при этом саму формулу: 1, 4, 1, 2. Видим, что все они положительны.

Кстати, весьма примечательно, что в математическом анализе определение монотонной функции дается в терминах, подобных используемым в нашей формуле (I), которая рассматривается там несколько более гибко, а $delta_i$ при этом называется *приращением* и имеет несколько иное обозначение.

Можно обобщить сказанное тем, что последовательность *приращений* показывает, на какую величину *уменьшается* каждый член исследуемой последовательности чисел, начиная с первого. Понятно, что если каждый член числовой последовательности уменьшается на положительную величину, то эта последовательность строго убывает и т. д.

Из всех этих рассуждений делаем вывод о том, что числовая последовательность является строго монотонной (то есть, строго возрастающей или строго убывающей) тогда и только тогда, когда $delta_i$ имеют один и тот же знак для всех i. Таким образом, мы вывели понятие, которое можно проверить с помощью последовательности однотипных действий, то есть, циклической обработки.

Теперь нам необходимо попробовать унифицировать проверку знакопостоянства всех *delta* для последовательностей обоих видов. Для этого рассмотрим произведение каких-либо двух *delta* в последовательностях (1) и (2). Примечательно, что оно положительно как произведение чисел одного знака. Таким образом, мы можем в любой последовательности взять $delta_1$ и получить произведения его со всеми остальными *delta* (обозначим эту формулу как (II)):

$$p = delta_1 * delta_i$$

Если все p положительны, то последовательность строго монотонна, а если же возникает хотя бы одно отрицательное произведение p, то условие монотонности нарушено.

Теперь перенесем эти рассуждения из математики в программирование и конкретизируем их на последовательность цифр восьмеричной записи числа. Обозначим идентификатором **delta** результат вычисления формулы (I) для двух текущих соседних членов последовательности.

Каким же образом двигаться по разрядам числа **n** и обрабатывать все **delta**?

Сначала мы можем получить последнюю цифру числа **n** (назовем ее **b**), отбросить ее и получить предпоследнюю цифру **n** (назовем ее **a**), отбросить ее тоже, а затем вычислить **delta = a – b**. Кстати, отметим, что при таком подходе мы будем для всех i находить $delta_i$, двигаясь по ним справа налево. Начальному фрагменту этих действий соответствует следующий код:

```
b := n mod 8;
n := n div 8;
a := n mod 8;
n := n div 8;
delta:= a - b;
```

Теперь мы можем войти в цикл с предусловием **n < > 0**. В каждом шаге цикла мы должны присвоить переменной **b** число **a**, затем считать следующий разряд в **a** и отбросить этот разряд в **n**. Таким способом мы «сдвигаем» текущую пару: например, на 1-ом шаге в примере (2) мы до входа в цикл использовали бы цифры 2 (в переменной **a**) и 0 (в переменной **b**), затем при входе в цикл скопировали бы 2 в **b** и 3 в **a** – таким образом, все было бы готово для исследования знака по произведению. В связи с этим основной цикл будет выглядеть так:

```
while n <> 0 do begin
  b := a;
  a := n mod 8;
  n := n div 8;
  ...
end;
```

На месте многоточия и будет проверка знакопостоянства произведений. Воспользовавшись формулой (II), мы заменим $delta_i$ в качестве текущего на саму разность **a – b**, чтобы не задействовать дополнительную переменную. В итоге, если теперь **delta * (a – b) <= 0**, то выходим из цикла:

```
if delta * (a - b) <= 0 then break;
```

Теперь рассмотрим развитие событий по завершении цикла:

1) Если произойдет выход по завершению цикла, то есть «закончится» число в связи с превращением его в 0 на некотором шаге, то значения **a**, **b** и **delta** будут содержать значения, подтверждающие строгую монотонность последовательности, что можно проверить с помощью вывода значения булевского выражения на экран.

2) Если в теле цикла произошел выход через условный оператор, то эти переменные будут содержать значения, с помощью которых выявлено условие нарушения строгой монотонности. Это значит, что по выходе из цикла ответ можно выводить в формате:

```
writeln(delta * (a - b) > 0);
```

Код:

```
 1. program OctalSequence;
 2.
 3. var
 4.   n, a, b: word;
 5.   delta: integer;
 6.
 7. begin
 8.   readln(n);
 9.   b := n mod 8;
10.   n := n div 8;
11.   a := n mod 8;
12.   n := n div 8;
13.   delta := a - b;
14.   while n <> 0 do begin
15.     b := a;
16.     a := n mod 8;
17.     n := n div 8;
18.     if delta * (a - b) <= 0 then break
19.   end;
20.   writeln(delta * (a - b) > 0)
21. end.
```

Кстати, что будет при вводе числа **n**, меньшего 8, ведь его восьмеричная запись однозначна? Хотя наша цепочка делений еще до входа в цикл требует двух цифр в записи числа!

Посмотрим, как поведет себя программа при вводе числа **n** из единственной цифры *k*: сначала *k* идет в **b** (строка 9), затем отбрасывается в **n** (строка 10), которое теперь равно 0, затем в **a** идет число 0, так как **0 mod 8 = 0** (строка 11). При этом строка 12 уже ничего не дает, так как **n**, которое сейчас равно нулю, присваивается значение **0 div 8 = 0**. Далее вычисляется **delta = –*k*** (строка 13), затем игнорируется цикл, так как **n = 0** (строки 14 – 19), затем выводится на экран значение выражения **–*k* * –*k* > 0** (строка 20), которое истинно для любого действительного *k* кроме нуля, который и не входит в условие нашей задачи, так как **n** натуральное.

Как видим, вырожденный случай прошел обработку «сам собой» и для него не пришлось разветвлять программу, что выражает несомненный плюс.

Задача № 33. Получить каноническое разложение числа на простые сомножители

Формулировка. Дано натуральное число **n (n > 1)**. Получить его каноническое разложение на простые сомножители, то есть представить в виде произведения простых сомножителей. При этом в разложении допустимо указывать множитель 1. Например, 264 = 2 * 2 * 2 * 3 * 11 (программе допустимо выдать ответ 264 = 1 * 2 * 2 * 2 * 3 * 11).

Решение. Данная задача имеет достаточно красивое решение.

Из *основной теоремы арифметики* известно, что для любого натурального числа больше 1 существует его каноническое разложение на простые сомножители, причем это разложение единственно с точностью до порядка следования множителей. То есть, например, 12 = 2 * 2 * 2 и 12 = 3 * 2 * 2 – это одинаковые разложения.

Рассмотрим каноническую форму любого числа на конкретном примере. Например, 264 = 2 * 2 * 2 * 3 * 11. Каким образом можно выявить эту структуру? Чтобы ответить на этот вопрос, вспомним изложенные в любом школьном курсе алгебры правила деления одночленов, представив, что числа в каноническом разложении являются переменными. Как известно, если разделить выражение на переменную в некоторой степени, содержащуюся в этом выражении в той же степени, оно вычеркивается в ее записи.

То есть, если мы разделим 264 на 2, то в его каноническом разложении уйдет одна двойка. Затем мы можем проверить, делится ли снова получившееся частное на 2. Ответ будет положительным, но третий раз деление даст остаток. Тогда нужно брать для рассмотрения следующее натуральное число 3 – на него частное разделится один раз. В итоге, проходя числовую прямую в положительном направлении, мы дойдем до числа 11, и после деления на 11 **n** станет равно 1, что будет говорить о необходимости закончить процедуру.

Почему при таком «вычеркивании» найденных сомножителей мы не получим делимостей на составные числа? На самом деле, здесь все просто – любое составное число является произведением простых сомножителей, меньших его. В итоге получается, что мы вычеркнем из **n** все сомножители любого составного числа, пока дойдем до него самого в цепочке делений. Например, при таком переборе **n** никогда не разделится на 4, так как «по пути» к этому числу мы вычеркнем из **n** все сомножители-двойки.

Алгоритм на естественном языке:

1) Ввод **n**;
2) Присвоение переменной **p** числа 2;
3) Вывод числа **n**, знака равенства и единицы для оформления разложения;
4) Запуск цикла с предусловием **n < > 1**. В цикле:
    1. Если **m mod p = 0**, то вывести на экран знак умножения и переменную p, затем разделить **n** на **p**, иначе увеличить значение **i** на 1;

Код:

```
 1. program PrimeFactors;
 2.
 3. var
 4.   n, p: word;
 5.
 6. begin
 7.   readln(n);
 8.   p := 2;
 9.   write(n, ' = 1');
10.   while n <> 1 do begin
```

```
11.     if (n mod p) = 0 then begin
12.       write(' * ', p);
13.       n := n div p
14.     end
15.     else begin
16.       inc(p)
17.     end
18.   end
19. end.
```

Задача № 34. Сформировать число из двух заданных чередованием разрядов

Формулировка. Даны два натуральных числа одинаковой десятичной разрядности. Сформировать из них третье число так, чтобы цифры первого числа стояли на нечетных местах третьего, а цифры второго – на четных. При этом порядки следования цифр сохраняются. Например, при вводе 1234 и 5678 программа должна выдать ответ 15263748 (для наглядности разряды обоих чисел выделены разными цветами).

Решение. Так как у чисел (обозначим их **a** и **b**) одинаковая десятичная разрядность, крайняя справа цифра у третьего числа (**c**, которое поначалу должно быть равно 0) всегда будет на четном месте, так как при его формировании мы работаем с длинами **a** и **b** как с числами одной четности, сумма которых всегда четна, и длина **c** как раз и есть позиция крайней справа цифры.

Это значит, что формирование **c** нужно в любом случае начинать с последнего разряда **b**. При этом каждый взятый из **a** или **b** разряд мы должны сместить на необходимую позицию влево, чтобы добавлять разряды **c**, используя операцию сложения. Мы сделаем это с помощью вспомогательной переменной **z**, которая перед входом в цикл будет равна 1. В цикле же она будет умножаться на последний добытый разряд **b** (при этом выражение **z * b mod 10** нужно прибавить к **c**), затем умножить **z** на 10 и проделать то же самое с последним разрядом **a** и снова умножить **z** на 10. Кстати, при этом нужно не забыть своевременно отбросить уже рассмотренные разряды чисел.

Так как разрядность чисел неизвестна, нам нужен цикл с предусловием. В силу одинаковой десятичной разрядности **a** и **b** мы можем сделать условие по обнулению любого из них, так как второе при этом также обнулится. Возьмем условие **a < > 0**.

Таким будет основной цикл:

```
while a <> 0 do begin
  c := c + z * (b mod 10);
  z := z * 10;
  b := b div 10;
  c := c + z * (a mod 10);
  z := z * 10;
  a := a div 10
end;
```

В итоге конечное число **c** будет сформировано в таком виде (все направления справа налево): первая цифра **b**, первая цифра **a**, вторая цифра **b**, вторая цифра **a** и так далее до самых последних разрядов слева. Кстати, скобки в двух операторах нужны для правильного понимания компилятором приоритета выполняемых арифметических операций. Без них **z** умножится на соответствующее число, и остаток от деления именно этого числа прибавится к **c**, что неправильно.

Алгоритм на естественном языке:

1) Ввод **a** и **b**;
2) Обнуление переменной **c**;

3) Присвоение переменной **z** числа 1;
4) Запуск цикла с предусловием **a < > 0**. В цикле:
    1. Прибавляем последний разряд **b** в текущий разряд **c**, определяемый с помощью множителя **z**;
    2. Умножаем **z** на 10;
    3. Избавляемся от последнего разряда в **b**;
    4. Прибавляем последний разряд **a** в текущий разряд **c** с помощью множителя **z**;
    5. Умножаем **z** на 10;
    6. Избавляемся от последнего разряда в **a**;
5) Вывод **c**.

Код:

```
 1. program CombineTwoNums;
 2.
 3. var
 4.   c, z: integer;
 5.   a, b: word;
 6.
 7. begin
 8.   readln(a, b);
 9.   c := 0;
10.   z := 1;
11.   while a <> 0 do begin
12.     c := c + z * (b mod 10);
13.     z := z * 10;
14.     b := b div 10;
15.     c := c + z * (a mod 10);
16.     z := z * 10;
17.     a := a div 10
18.   end;
19.   writeln(c)
20. end.
```

Задача № 35. Вывести на экран x, записанное в системе счисления с основанием n

Формулировка. Даны натуральные числа **x** и **n** (**n** <= 10). Вывести на экран число **x**, записанное в системе счисления с основанием **n**.

Решение. Вспомним правило из **задачи 5**:

> *Остаток от деления любого десятичного числа **x** на число **p** дает нам разряд единиц числа **x** (его крайний разряд справа) в системе счисления с основанием **p**.*

Раньше мы принимали это правило без доказательства, однако сейчас мы коснемся его, так как оно достаточно краткое.

Воспользуемся формулой записи десятичного числа **x** в системе счисления с основанием **p**, состоящего из **r** знаков:

$$x = a_{r-1} * p^{r-1} + a_{r-2} * p^{r-2} + ... + a_2 * p^2 + a_1 * p^1 + a_0 * p^0,$$

где p^{r-1}, p^{r-2}, ..., p^2, p^1, p^0 – основание системы счисления, возведенное в соответствующие степени, a_{r-1}, a_{r-2}, ..., a_2,a_1, a_0 – цифры в записи этого числа в системе счисления с основанием **p**.

Например, число 378 в десятичной системе счисления выглядит так: $378 = 3 * 10^2 + 7 * 10^1 + 8 * 10^0$. Если мы подряд выпишем цифры a_2 (= 3), a_1 (= 7), a_0 (= 8), то исходное число восстановится.

Запишем представление числа в 22 двоичной системе счисления (переведем его с помощью калькулятора, оно равно 10110_2) по этой же формуле: $22 = 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0$. Понятно, что если мы вычислим выражение в правой части равенства, то получим как раз 22.

Теперь покажем то, что если мы возьмем остаток от деления числа 22 на 2, затем разделим его на 2, отбросив остаток, и будем повторять эти действия до обнуления числа, то в итоге получим все его разряды в порядке справа налево. Возьмем его запись $1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 1 * 2^1 + 0 * 2^0$ и разделим ее на 2. Из алгебры известно, что если мы делим сумму чисел на некоторое число, то на него делятся все слагаемые этой суммы. $1 * 2^4$, $0 * 2^3$, $1 * 2^2$ и $1 * 2^1$ делятся на 2, так как в них присутствует множитель 2. $0 * 2^0 = 0 * 1 = 0$ не делится на 2, соответственно, это число будет остатком от деления на 2, и при этом по формуле оно является крайним справа разрядом. Затем мы делим всю эту запись на 2 и отбрасываем остаток, получаем: $1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0$. Очевидно, что при следующем взятии остатка мы получим цифру из крайнего справа слагаемого. Повторяя эту цепочку, мы постепенно получим все цифры числа 22 в системе счисления с основанием 2.

Обобщая вышесказанное, приходим к выводу, что для формирования записи числа нам необходимо получить все остатки от деления **x** на основание **n**, при этом деля **x** на **n** после каждого взятия остатка.

Каким образом мы запишем остатки справа налево? Очень просто: умножаем очередной остаток на некоторый множитель **z**, добавляющий необходимое количество нулей, чтобы цифра оказалась в необходимой позиции, и прибавляем к результату **r**. Поначалу **z** будет равен 1, так как мы прибавляем цифру к разряду единиц, затем **z** в каждой итерации будет умножаться на 10.

В итоге мы прибавляем к результату **r** первый остаток, умноженный на 1, второй остаток, умноженный на 10, третий остаток, умноженный на 100 и так далее, пока не будет сформировано искомое число:

```
r := 0;
z := 1;
while x <> 0 do begin
  r := r + z * (x mod n);
  x := x div n;
  z := z * 10
end;
```

Код:

```
 1. program ConvertNotation;
 2.
 3. var
 4.   c, z, r: integer;
 5.   x, z: word;
 6.
 7. begin
 8.   readln(x, n);
 9.   r := 0;
10.   z := 1;
11.   while x <> 0 do begin
12.     r := r + z * (x mod n);
13.     x := x div n;
14.     z := z * 10
```

```
 15.   end;
 16.   writeln(r)
 17. end.
```

Задача № 36. Найти наименьший нетривиальный делитель двух заданных чисел

Формулировка. Даны натуральные числа **m** и **n**. Вывести на экран их наименьший нетривиальный делитель или сообщить, что его нет.

Решение. Задача похожа на **задачу 15**, в которой поиск минимального делителя осуществлялся с помощью цикла:

```
for i := 2 to n do begin
  if n mod i = 0 then begin
    writeln(i);
    break
  end
end;
```

Здесь все просто: проверяем все числа от 2 по возрастанию – если нашли делитель, выводим его на экран и выходим из цикла с помощью **break**. В нашем же случае нужно проверять делимость двух введенных чисел. При этом цикл должен проходить по всем **i** от 2 до минимального из чисел **m** и **n** (назовем его **min**), так как оно может быть наименьшим нетривиальным делителем, когда оно простое. Например, для чисел 17 и 34 таковым является 17.

Найти наименьшее из двух чисел можно так:

```
if n < m then min := n else min := m;
```

Кстати, теперь в цикле мы должны не просто вывести на экран найденный делитель, а сохранить его в некоторую переменную (**mindiv**), которая до входа в цикл будет равна 1 (или 0), чтобы проверить, выполнилось ли условие делимости в цикле. Если да, то необходимо вывести значение наименьшего общего делителя, а если нет, и **mindiv** все еще равно 1 (или 0), то вывести сообщение об отсутствии делителя.

Код:

```
 1. program MinDivisor;
 2.
 3. var
 4.   m, n, i, min, mindiv: word;
 5.
 6. begin
 7.   readln(m, n);
 8.   mindiv := 1;
 9.   if n < m then min := n else min := m;
10.   for i := 2 to min do begin
11.     if (n mod i = 0) and (m mod i = 0) then begin
12.       mindiv := i;
13.       break
14.     end
15.   end;
16.   if mindiv <> 1 then writeln(mindiv) else writeln('No divi-
    sors!')
17. end.
```

Задача № 37. Проверить, является ли натуральное число счастливым билетом

Формулировка. Дано натуральное число **n**. Проверить, является ли оно счастливым билетом.

Примечание: вообще, в математике обычно рассматриваются счастливые билеты с четным количеством цифр, потому что у них можно явно выделить левую и правую половины одинаковой длины, суммы цифр которых и сравниваются. Однако мы несколько расширим это определение, полагая, что если число имеет нечетную длину, его центральную цифру можно отбросить, так как ее логично было бы прибавить к накапливаемым суммам обоих половин, что, собственно, не изменит отношения между ними.

Например, число 14350 – счастливый билет, так как 1 + 4 = 5 + 0, а центральную цифру мы отбросили.

Решение. Задача является общим случаем **задачи 10**. Для ее решения необходимо знать длину числа (то есть его разрядность), вследствие чего нам необходимо скопировать переменную **n** в некоторую другую (например, **a**), чтобы на основе **a** посчитать количество десятичных разрядов **n** и сохранить его в некоторой переменной **digits** (*digits* в пер. с англ. означает «цифры»**)**. Сделать это можно так:

```
a := n;
digits := 0;
while a <> 0 do begin
  a := a div 10;
  inc(digits)
end;
```

Здесь мы в каждой итерации цикла отбрасываем одну цифру от **a** и увеличиваем значение счетчика **digits** на 1. На некотором шаге число **a** будет однозначно и станет равным нулю при делении на 10, и после инкрементации счетчика, который теперь уже будет содержать количество цифр числа, произойдет выход из цикла.

Чтобы посчитать суммы левой и правой половины цифр числа (для накопления которых мы предусмотрим переменные **left** и **right**), мы должны запустить два последовательных цикла от 1 до **digits div 2**, в которых прибавлять каждый полученный разряд к соответствующей переменной и отбрасывать его. Если длина нечетная, нам необходимо отбросить серединную цифру числа без прибавления к какой-либо сумме. Так как в первом цикле мы обработали и отбросили правую половину цифр числа, то по выходе из него серединная цифра как раз будет находиться в разряде единиц. Поэтому необходим следующий шаг:

```
if odd(digits) then n := n div 10;
```

Напомним, что функция **odd(n)** возвращает значение **true**, если **n** нечетно, и **false**, если **n** четно. То есть, написанный выше оператор проверяет счетчик **digits** (в котором хранится длина исходного числа) на нечетность, и если оно нечетно, отбрасывает последнюю его цифру.

Далее необходимо накопить сумму цифр левой половины числа и вывести на экран результат сравнения сумм левой и правой половины.

Код:

```
1. program HappyTicket;
2.
3. var
4.   n, a: longint;
5.   left, right, digits, i: byte;
6.
7. begin
```

```
 8.   readln(n);
 9.   a := n;
10.   digits := 0;
11.   while a <> 0 do begin
12.     a := a div 10;
13.     inc(digits)
14.   end;
15.   left := 0;
16.   right := 0;
17.   for i := 1 to digits div 2 do begin
18.     right := right + n mod 10;
19.     n := n div 10
20.   end;
21.   if odd(digits) then n := n div 10;
22.   for i := 1 to digits div 2 do begin
23.     left := left + n mod 10;
24.     n := n div 10
25.   end;
26.   writeln(left = right)
27. end.
```

Представим, как должен работать алгоритм при вводе числа 14350:

1) Считаем длину числа, она равна 5 (строки 11-14);
2) В цикле из **5 div 2 = 2** повторений прибавляем к **right** крайние справа цифры 0 и 5, после чего отбрасываем их и имеем в **n** 143 (строки 17-20);
3) Так как **odd(digits) = odd(5) = true**, отбрасываем 3, после чего имеем в **n** 14 (строка 21);
4) В цикле из **5 div 2 = 2** повторений прибавляем к **left** оставшиеся цифры 1 и 4, после чего **n** становится равно 0, что, впрочем, нас уже не интересует (строки 22-25);
5) Выводим на экран значение выражения **left = right** – ответ положительный (строка 26).

Задача № 38. Проверить, является ли натуральное число палиндромом

Формулировка. Дано натуральное число **n**. Проверить, представляет ли собой палиндром его десятичная запись.

Решение. Задача является общим случаем **задачи 9**. Чтобы решить ее, необходимо разделить число **n** на две половины одинаковой длины, отбросить серединную цифру в случае нечетной длины **n** и проверить равенство одной из частей реверсной записи другой части.

Так как нам заранее неизвестна десятичная разрядность **n**, мы можем посчитать ее с помощью следующего цикла (подробнее это описывалось в предыдущей задаче):

```
a := n;
digits := 0;
while a <> 0 do begin
  a := a div 10;
  inc(digits)
end;
```

Теперь рассмотрим варианты проверки числа на палиндром вместе с разбором на примере.

Пусть дано число нечетной длины, например, 79597. Мы можем отделить его правую половину 97, проведя ряд последовательных делений с взятием остатка в цикле из **digits div 2** повторе-

ний. При этом необходимо сразу сформировать ее реверс в переменную **right** (мы делали это в **задаче 31**):

```
right := 0;
for i := 1 to digits div 2 do begin
  right := right * 10;
  right := right + n mod 10;
  n := n div 10
end;
```

Так как число нечетно, нужно отбросить его центральную цифру 5, после чего в переменной **n** (равной 79) будет содержаться левая половина числа, а в переменной **right** (также равной 79) – его перевернутая правая половина. Они равны, следовательно, ответ положительный.

Тот же порядок действий применяется и для чисел четной длины, однако теперь нам не нужно ничего отбрасывать после накопления реверсной левой части числа в переменную **right**, так как в числах четной длины нет серединной цифры. Например, дано число 1551: переворачиваем правую половину числа 51 (получим 15) и сравниваем ее с левой половиной: 15 = 15, ответ положительный.

Эти допущения говорят о том, что необходима проверка длины числа **n** на нечетность и, соответственно, отбрасывание серединной цифры в случае нечетности:

```
if odd(digits) then n := n div 10;
```

Код:

```
 1. program CheckPalindrome;
 2.
 3. var
 4.   n, a, right: longint;
 5.   digits, i: byte;
 6.
 7. begin
 8.   readln(n);
 9.   a := n;
10.   digits := 0;
11.   while a <> 0 do begin
12.     a := a div 10;
13.     inc(digits)
14.   end;
15.   right := 0;
16.   for i := 1 to digits div 2 do begin
17.     right := right * 10;
18.     right := right + n mod 10;
19.     n := n div 10
20.   end;
21.   if odd(digits) then n := n div 10;
22.   writeln(n = right)
23. end.
```

Выполним «ручную прокрутку» алгоритма на числе 147741:

1) Считаем длину числа, она равна 6 (строки 11-14);
2) В цикле из **6 div 2 = 3** повторений прибавляем к **right** (формируя реверсную запись) последние три цифры числа **n**, после чего отбрасываем их и имеем в **n** 147, в **right** 147 (строки 16-20);

3) Так как **odd(digits) = odd(6) = false**, ничего не делаем (строка 21);

4) Выводим на экран значение выражения **n = right** – ответ положительный (строка 22).

Задача № 39. Проверить, является ли натуральное число степенью двойки

Формулировка. Дано натуральное число **n**. Проверить, представляет ли оно собой натуральную степень числа 2.

Решение. Проще говоря, нам нужно ответить на вопрос: можно ли возвести число 2 в какую-либо натуральную степень (или в нулевую степень, так как $2^0 = 1$), чтобы получилось число **n**?

Вообще, для решения этой задачи существует достаточно красивое равенство, выполняющееся для всех натуральных степеней числа 2, позволяющее получить ответ с помощью одной единственной логической побитовой операции:

$$\mathbf{n\ and\ (n - 1) = 0}$$

Обозначим его как (1).

Дело в том, что натуральная степень числа 2 с показателем **p** в двоичном виде всегда представляется как единица с **p** нулями справа. Это происходит потому, что двоичная запись этого числа в десятичном виде представляется как $1 * 2^p + 0 * 2^{p-1} + ... + 0 * 2^1 + 0 * 2^0$, где все пропущенные слагаемые имеют коэффициент 0, и из этой записи легко восстановить двоичное представление: 10...00, здесь нулей всего **p**. Поэтому если мы отнимем от любой степени двойки 1, то получим число 1...11, где всего **p** единиц (точнее говоря, это будет число 01...11). В итоге, если мы применим к этим двум числа побитовую конъюнкцию, то всегда будем получать результирующее число, равное 0.

Примечание: побитовая конъюнкция – это бинарная операция, которая эквивалента обычной конъюнкции, примененной к двоичным разрядам операндов (двух исходных чисел), стоящим на одинаковых позициях в двоичных представлениях этих чисел. При этом результатом применения побитовой конъюнкции является некое результирующее число, значение соответствующих битов которого зависит от значений битов исходных чисел: в соответствующем разряде будет находиться 1 тогда и только тогда, когда на этих позициях в обоих исходных числах стояли единичные биты, и 0, иначе.

Пример: выполним поразрядную конъюнкцию двоичных чисел 011001_2 и 101011_2 (при этом выпишем их так, чтобы соответствующие двоичные разряды стояли друг под другом):

```
Первый операнд: 011001₂
Второй операнд: 101011₂
Результат:      001001₂
```

Биты, конъюнкция которых даст 0, выделены красным цветом, а те, конъюнкция которых даст 1 – синим.

Так как 1-й разряд слева у первого числа равен 0, а у второго – 1, то в соответствующий первый разряд результата идет бит 0. 2-е разряды, соответственно, равны 1 и 0, и в результат снова идет бит 0. 3-и разряды у обоих чисел равны 1 (выделены синим цветом), поэтому в 3-й разряд результата идет 1 и так далее.

Кстати, наша формула (1) пропускает число 0 в качестве степени двойки. Так как компиляторы языка **Pascal** (гарантированно называются Borland Delphi 7 и PascalABC) реализуют числовые типы данных в виде кольцевых отрезков (то есть, например, в типе **byte** после числа 255 следует число 0, а перед числом 0 – число 255), то в любом таком типе выражение (0 – 1) имеет некоторое ненулевое битовое представление (так как нулевое битовое представление имеет лишь число 0), а побитовая конъюнкция числа 0 и любого другого числа дает в результате число 0.

Вообще, так как нам данное нам n является натуральным числом, число 0 вводиться не будет. Однако покажем, как отсечь 0 при проверке числа по формуле (1): можно осуществить про-

верку введенного числа на равенство нулю, и в случае равенства заменить его на какое-либо другое число, заведомо не являющееся степенью двойки, чтобы условие формулы (1) отработало правильно:

```
if n = 0 then n := 3;
```

Вообще, формула (1) требует доказательства в обе стороны: мы лишь доказали, что если **n** является степенью двойки, то есть $n = 2^p$ (где **p** – любое натуральное число или 0), то выражение **n and (n – 1)** гарантированно дает результат 0. Покажем это схематически еще раз:

```
Первый операнд: 100...00
Второй операнд: 011...11
Результат:      000...00
```

Однако мы также должны доказать, что никакое другое число **n**, кроме как степень двойки, не может дать 0 в результате выполнения операции **n and (n – 1)**. Однако мы примем это утверждение без доказательства. В итоге тело программки может выглядеть так (для натурального **n**, которое также может быть нулем):

```
readln(n);
if n = 0 then n := 3;
writeln(n and (n - 1) = 0);
```

Однако мы в качестве основного решения возьмем более простую идею: пусть данное число **n** является степенью двойки. Следовательно, его можно представить так: $2^p = 1 * 2 * 2 * ... * 2$ (здесь ровно **p** двоек). Разделив это выражение на 2 определенное количество раз, в результате мы получим число 1.

Если же число **n** не является степенью двойки, то на некотором шаге мы получим остаток при делении на 2. В связи с этим возникает алгоритм:

1) Вводим **n**;
2) В цикле с предусловием **n** > 1 работаем с **n**:
    1. Если остаток от деления **n** на 2 равен 1 (**n mod 2 = 1**), то выходим из цикла;
    2. Делим **n** на 2 (**n := n div 2**);
3) Выводим на экран значение выражения **n = 1** (если цикл завершился, то это условие истинно и **n** – степень двойки, а если нет – то на каком-то шаге мы получили остаток при делении на 2 и вышли через **break**);

Даже если ввести **n**, равное 0, то программа выдаст правильный ответ, так как не будет осуществлен вход в цикл (2) и на шаге (3) будет выведено значение выражения 0 = 1, равное **false**.

Код:

```
 1. program PowerOfTwo;
 2.
 3. var
 4.   n: integer;
 5.
 6. begin
 7.   readln(n);
 8.   while n > 1 do begin
 9.     if n mod 2 = 1 then break;
10.     n := n div 2
11.   end;
12.   writeln(n = 1)
13. end.
```

Задача № 40. Вывести на экран произведение четных элементов заданной последовательности натуральных чисел

Формулировка. Дана последовательность натуральных чисел, ограниченная вводом нуля. Вывести на экран произведение четных элементов этой последовательности. При этом ноль не считается членом последовательности.

Примечание: задачи подобного рода требуют выполнения каких-либо действий в зависимости от некоторого характеристического свойства. Большинство из них математически неинтересны, однако развивают способность совмещать отдельные приемы и методы программирования, поэтому, в основном, способствуют наработке опыта.

Решение. Так как нам заранее неизвестна длина рассматриваемой последовательности, но мы знаем о том, что она ограничивается вводом нуля, и поэтому можем сделать цикл с предусловием **a < > 0**, где **a** – текущий введенный член. Так как нет необходимости работать с несколькими членами одновременно, мы можем следовать данной схеме и в конкретный момент времени работать лишь с одним элементом последовательности.

При всем этом перед циклом нам необходимо считать первый член **a**, чтобы войти в цикл, если последовательность непустая (а если пустая, то есть состоит из одного нуля, то и не нужно входить в цикл и что-либо делать):

```
read(a);
while a <> 0 do begin
  ...
  read(a)
end;
```

Кстати, мы используем оператор ввода **read**, чтобы можно было вводить члены через пробел, а не через **enter**, как при использовании **readln**. Внутри цикла вместо многоточия должна располагаться некоторая последовательность операторов, которые и будут выполнять обработку вводимых данных. После каждой итерации необходимо ввести следующий член последовательности и продолжить обработку, если он не равен нуля, и закончить в противном случае.

Примечательно, что лишь за некоторыми исключениями именно в цикле такого вида будет располагаться основной блок обработки для всех задач на последовательность, ограниченную вводом нуля.

Что же касается текущей задачи, то для ее решения мы в цикле должны проверить каждый элемент на четность, и если он четный, домножить на него некоторую переменную **prod** для накопления результата (которую поначалу нужно сделать равной 1). При этом если по завершении программы переменная **prod** будет по-прежнему равна 1, то это значит, что последовательность либо пуста, либо в ней нет четных элементов, что побуждает сделать в этом случае соответствующую проверку с выводом результата или сообщения об отсутствии элементов. В связи с написанным возникает такой код:

```
read(a);
prod := 1;
while a <> 0 do begin
  if a mod 2 = 0 then prod := prod * a;
  read(a)
end;
if prod <> 1 then writeln(prod) else writeln('No such elements!');
```

При этом проверяется неравенство **prod** единице, так как хотелось бы поместить в **then**-блоке условного оператора вывод «положительного» ответа (который отвечает критерию задачи), а в **else**-блоке – обработку «вырожденного случая». На самом же деле такой порядок не должен быть самоцелью и не считается «хорошим тоном» в программировании, да и делается только по прихоти автора, так что не было бы никакой разницы, если бы в последней строчке было:

```
if prod = 1 then writeln('No such elements!') else writeln(prod);
```

Код:

```
 1. program ProductOfEven;
 2.
 3. var
 4.   a, prod: word;
 5.
 6. begin
 7.   read(a);
 8.   prod := 1;
 9.   while a <> 0 do begin
10.     if a mod 2 = 0 then prod := prod * a;
11.     read(a)
12.   end;
13.   if prod <> 1 then writeln(prod) else writeln('No such ele-
    ments!')
14. end.
```

Задача № 41. Вывести на экран произведение двузначных элементов последовательности натуральных чисел, которые делятся на заданное число

Формулировка. Дано натуральное число **n**, а затем последовательность натуральных чисел, ограниченная вводом нуля. Вывести на экран произведение двузначных элементов этой последовательности, которые делятся на **n**.

Решение. Задача очень похожа на предыдущую, только в этот раз нам необходимо проверять делимость не на 2 (это было условие четности), а на **n**. К тому же, мы должны рассматривать только двузначные члены. Для выявления двузначного числа мы, однако, не будем считать его разрядность и сравнивать ее с двойкой, а воспользуемся тем, что любое двузначное число больше 9 и меньше 100.

В связи с этим условие поиска члена последовательности, отвечающего заданным критериям, будет выглядеть так: **(a > 9) and (a < 100) and (a mod n = 0)**.

Напомним, что **and** – это конъюнкция, причем наше выражение содержит три конъюнктивных члена, так как операция применяется дважды. Оно истинно тогда и только тогда, когда истинны все три члена, и ложно во всех остальных случаях. При этом конъюнктивные члены стоят в скобках, так как логические операции в языке **Pascal** имеют больший приоритет, чем операции сравнения и арифметические операции. То есть, если опустить скобки, то **Pascal**, вычисляя значение слева направо, выполнит сначала операции **9 and a**, **100 and a** и т. д., что в корне неправильно.

Код:

```
 1. program ProductOfReqNums;
 2.
 3. var
 4.   a, n, prod: word;
 5.
 6. begin
 7.   readln(n);
 8.   read(a);
 9.   prod := 1;
10.   while a <> 0 do begin
11.     if (a > 0) and (a < 100) and (a mod n = 0) then prod :=
```

```
    prod * a;
12.     read(a)
13.   end;
14.   if prod <> 1 then writeln(prod) else writeln('No such ele-
    ments!')
15. end.
```

Задача № 42. Найти количество простых членов последовательности

Формулировка. Дана последовательность натуральных чисел, ограниченная вводом нуля. Вывести на количество простых членов этой последовательности.

Решение. Принцип решения этой задачи напоминает решения обеих предыдущих задач. При этом алгоритм распознавания простых чисел можно взять из **задачи 17**, немного изменив его:

```
s := 0;
for i := 1 to a do begin
  if a mod i = 0 then inc(s)
end;
if s = 2 then inc(count);
```

Здесь мы предварительно поменяли названия переменных и вместо вывода ответа о простоте числа работаем со счетчиком найденных простых чисел. Напомним, что в цикле считается количество всех возможных натуральных делителей числа, и если их 2, то оно простое, и необходимо увеличить счетчик простых чисел **count**. Когда вся числовая последовательность будет обработана, останется только вывести на экран значение переменной **count**.

Код:

```
 1. program NumOfPrimes;
 2.
 3. var
 4.   a, i, s, count: word;
 5.
 6. begin
 7.   read(a);
 8.   count := 0;
 9.   while a <> 0 do begin
10.     s := 0;
11.     for i := 1 to a do begin
12.       if a mod i = 0 then inc(s)
13.     end;
14.     if s = 2 then inc(count);
15.     read(a)
16.   end;
17.   writeln(count)
18. end.
```

Задача № 43. Проверить, начинается ли каждый из членов последовательности с десятичной цифры, на которую оканчивается предыдущий

Формулировка. Дана последовательность натуральных чисел, ограниченная вводом нуля. Проверить, начинается ли каждый из ее членов (со второго) с десятичной цифры, на которую оканчивается предыдущий. Например, таковой последовательностью будет являться 14 47 712 2179 9 9 93 0 (также сохранен ограничивающий ноль).

Решение. Задача решается через цикл с предусловием, что характерно для задач на последовательность с ограничителем. При ее решении мы могли бы не рассматривать «вырожденные варианты», в которых на вход будет подаваться, например, пустая последовательность (то есть состоящая из единственного нуля) или последовательность из одного члена, так как на вопрос о том, удовлетворяют ли такие последовательности заданному критерию, ответить теоретически достаточно трудно. Разумнее всего было бы считать выполнение критерия для таких последовательностей неопределенным, что, однако, в формате даваемого нами ответа не представляется возможным: под «проверкой» характеристического свойства в данном случае понимается ответ на вопрос: либо «да», и последовательность отвечает заданным требованиям, либо «нет», и, соответственно, не отвечает.

Следовательно, нам нужно вместо ответа о «неопределенности» проверяемого свойства дать один из допустимых ответов. Наверное, разумно было бы дать при обработке вырожденных случаев программой ответ «нет». Мы возьмем это на заметку и попытаемся сделать контроль исключений, когда уже будет готово решение задачи для общего случая, чтобы заранее не наделать ошибок. Это необходимо потому, что мы попробуем инициализировать значения переменных при запуске программы таким образом, чтобы обработку вырожденных случаев можно было выполнить с минимальным вложением дополнительного кода – мы уже делали это в **задаче 32.**

Теперь о переменных. Так как нам необходимо проверять выполнение критерия на двойках (парах) элементов, то и хранить в памяти нужно сразу два элемента (**a** и **b**). Первый элемент не имеет предшественника, поэтому после ввода мы не обрабатываем его и можем вводить следующий элемент в одном операторе с ним:

```
read(a, b);
```

Имея два элемента, мы уже можем выполнить проверку нашего свойства. Однако уже на этом этапе мы можем догадаться, что в силу необходимости выполнить проверку для всех пар элементов последовательности следует сразу поместить ее в цикл. Мы будем считывать каждый очередной член последовательности в переменную **b**, и так как последнее вводимое число по условию – 0, то предусловием цикла будет **b < > 0** (так как при **b = 0** цикл должен прекратиться):

```
while b <> 0 do begin
    ...
    a := b;
    read(b)
end;
```

На месте троеточия будет располагаться код проверки каждой пары, полностью охватывающий определение нашего свойства. **Примечание:** в «шаблоне» основного цикла мы выделили также оператор **a := b**, который обеспечивает движение по каждым двум соседним элементам последовательности. Следует обратить внимание на то, что мы должны проверять выполнение нашего свойства для 1-го и 2-го, 2-го и 3-го, 3-го и 4-го и т. д. элементов последовательности.

Когда мы уже выполнили проверку для двух элементов, например, для 1-го (который хранится в переменной **a**) и 2-го (который хранится в переменной **b**), то далее мы присваиваем переменной **a** значение переменной **b** (в которой у нас хранился 2-ой элемент), затем считываем в **b** 3-й элемент, чтобы проверить свойство для 2-го и 3-го элементов и т. п.

При этом нужно четко понимать, что мы не можем считывать в цикле сразу две переменные, так как при таком подходе проверка будет выполняться лишь для 1-го и 2-го, 3-го и 4-го и т. д. элементов, что неверно.

Разберем саму проверку. Так как на каждом шаге цикла программе требуется выяснить, начинается ли следующий член последовательности с десятичной цифры данного, то, имея данный член в переменной **a** и следующий член в **b**, мы должны сравнить последнюю цифру **a** (обозначим ее как **last**) с первой цифрой **b** (обозначим ее как **first**). Сделать это можно так:

```
last := a mod 10;
first := b;
while first > 9 do begin
```

```
    first := first div 10
  end;
```

Здесь мы сначала добыли последнюю цифру **a** (строка 1), затем скопировали в **last** значение **b** (переменную **b** нельзя изменять, так как ее значение понадобится нам на следующем шаге цикла) и во вложенном цикле разделили **last** на 10 столько раз, чтобы в ней осталась лишь одна цифра, которая является его первой цифрой.

Добыв нужные цифры, мы можем выполнить их сравнение и выйти из цикла в том случае, когда они не равны, так как при этом нарушается наше характеристическое свойство, данное в условии, и после этого дальнейшая проверка бессмысленна:

```
  if last <> first then break;
```

Когда цикл завершится, нам останется лишь вывести на экран результат сравнения переменных **last** и **first**: если цикл завершился, то последовательность отвечает заданному свойству (так как не было выхода через **break**), они будут равны и будет выведен ответ *true*; если же был совершен выход через **break**, то переменные неравны и ответ, соответственно, *false*.

Теперь попробуем оптимизировать программу для обработки вырожденных случаев для пустой последовательности (когда вводится единственный 0) и для последовательности из одного члена (когда вводится некоторое число и 0): мы договорились выводить для них ответ *false*.

Очевидно, в данный момент наша программа обрабатывает корректно минимальный случай из двух членов: тогда проходит одно повторение тела цикла, в котором переменные **last** и **first** получают значения, затем может произойти выход по **break** или завершение цикла по вводу нуля, как и должно быть.

Однако если мы введем последовательность из одного члена, то при вводе **a** и **b** в переменную **a** пойдет этот член, а в **b** окажется ограничивающий ноль, что приведет к невыполнению входа в основной цикл и программа перейдет к оператору вывода **writeln(last = first)**, что неверно, так как значение переменных **last** и **first** в данный момент будет не определено и выражение в операторе вывода может дать любой результат. Это значит, что для избегания подобного исхода нам нужно выполнить инициализацию переменных **last** и **first** заведомо неравными значениями, чтобы получить гарантированный ответ *false* при вводе последовательности из одного члена. Это можно сделать так:

```
  first := 1;
  last := 0;
```

Но что будет, если ввести последовательность, состоящую из одного нуля? В нашей программе это невозможно, так как оператор ввода в начале содержит две переменные, и если мы сразу введем 0, то программа «зависнет» в ожидании ввода второго числа. Чтобы избежать этого, мы должны вводить одно число в переменную **a**, и если оно не равно 0, нужно ввести **b**. Вместе с этим необходимо заранее присвоить переменной **b** число 0, так как она была определена в случае последовательности из одного члена, чтобы не осуществился вход в основной цикл:

```
  read(a);
  b := 0;
  if a <> 0 then read(b);
```

Эта конструкция заменит оператор **read(a, b)**, который мы описывали в самом начале решения задачи.

Код:

```
 1. program LastAndFirst;
 2.
 3. var
 4.   a, b, first, last: word;
 5.
```

```
 6. begin
 7.   first := 1;
 8.   last := 0;
 9.   read(a);
10.   b := 0;
11.   if a <> 0 then read(b);
12.   while b <> 0 do begin
13.     last := a mod 10;
14.     first := b;
15.     while first > 9 do begin
16.       first := first div 10
17.     end;
18.     if last <> first then break;
19.     a := b;
20.     read(b)
21.   end;
22.   writeln(last = first)
23. end.
```

Задача № 44. Проверить, является ли последовательность пилообразной

Формулировка. Дана последовательность из трех и более натуральных чисел, ограниченная вводом нуля. Проверить, является ли эта последовательность пилообразной.

Примечание: пилообразной называется последовательность чисел, в которой каждый член, имеющий соседние члены, меньше или больше их.

Пример такой последовательности: 14 12 18 7 10 2. Покажем, что данная последовательность соответствует определению: ее 1-й член (14) мы не рассматриваем, так как он имеет всего один соседний член; 2-й член (12) меньше соседних: 1-го (14) и 3-го (18); 3-й член (18) больше 12 и 7, 7 меньше 18 и 10, 10 больше 7 и 2, а последний элемент 2 мы также не рассматриваем. Эту запись можно формализовать, если между каждыми двумя соседними членами последовательности поставить знак отношения между их величинами («>» или «<»). В связи с этим приведенный выше пример можно проиллюстрировать так: 14 > 12 < 18 > 7 < 10 > 2. При этом характерно направление значков, показывающее, что каждый элемент либо меньше, либо больше соседних. При этом если мы выпишем сами знаки сравнения, то получим символьное сочетание > < > < >. А если выписать эти символы в столбик, становится понятно, почему такая последовательность названа пилообразной.

Решение. Исследуем свойства пилообразной последовательности. В определении сказано, что все ее элементы (кроме двух крайних) меньше либо больше соседних. Конкретизируем это понятие: любую тройку рядом стоящих элементов (левый элемент, центральный элемент, правый элемент) в данной последовательности мы будем называть «зубом».

Например, для указанного выше примера зубьями будут являться тройки 14 12 18, 12 18 7, 18 7 10 и 7 10 2. Каждый зуб удовлетворяет условию, данному в определении, следовательно, последовательность является пилообразной. Очевидно, что если все зубья в некоторой последовательности удовлетворяют данному условию, то эта последовательность является пилообразной.

Найдем некоторые свойства «правильных зубьев», то есть таких, которые отвечают заданному определению. Для этого формализуем рассуждения над элементами зуба, обозначив их как **L** (левый элемент, от англ. *left* – левый), **M** (средний элемент, от англ. *medium* – средний), **R** (правый элемент, от англ. *right* – правый).

Известно, что средний элемент в зубе может быть либо меньше, либо больше крайних, в связи с чем для обоих случаев возникает ряд следующих неравенств:

I случай (средний элемент меньше крайних): $M < L \wedge M < R \Rightarrow \overset{1}{M - L} < 0 \wedge \overset{2}{M - R} < 0$

Здесь знак $\wedge$ обозначает конъюнкцию (логическое «и»), а $\Rightarrow$ обозначает логическое следование (то есть, из выражения, которое стоит левее знака $\Rightarrow$, следует выражение, стоящее правее знака $\Rightarrow$).

II случай (средний элемент больше крайних): $M > L \wedge M > R \Rightarrow \overset{1}{M - L} > 0 \wedge \overset{2}{M - R} > 0$

Итак, в обоих случаях мы составили две разности, которые для удобства обозначили (1) и (2). В I-ом случае (1) и (2) всегда строго больше 0, а во II-ом случае – строго меньше 0.

Составим произведение выражений (1) и (2) и обозначим его как (3): (*M* – *L*) * (*M* – *R*). В I-ом случае оно всегда положительно как произведение отрицательных чисел, во II-ом случае – также всегда положительно как произведение положительных чисел.

Что же следует из этого выражения? А то, что если результат его вычисления – положительное число, то тройка элементов **L**, **M**, **R** образует «правильный зуб». И как мы помним, если все тройки соседних элементов последовательности образуют «правильные зубья», то она является пилообразной.

Так как в условии задачи на вход подается как минимум три элемента, мы можем прочитать их в одном операторе:

```
read(L, M, R);
```

Затем мы входим в цикл с предусловием **R** < > 0. В этом цикле мы должны исследовать каждую тройку L, M, R по свойству (3):

```
while R <> 0 do begin
  if (L - M) * (R - M) <= 0 then break;
  L := M;
  M := R;
  read(R)
end;
```

На каждом шаге цикла мы сначала исследуем уже имеющуюся тройку (оператор 1 в теле цикла), и если выражение (3) оказывается равно или меньше нуля, то выходим из цикла, так как нашли «неправильный зуб», который нарушает условие пилообразной последовательности, в силу чего дальнейшая проверка бессмысленна. Затем нам нужно «сдвинуть» последовательность: например, если в переменных **L**, **M**, **R** у нас соответственно хранились 4-й, 5-й и 6-й элементы последовательности, которые мы уже проверили, то с помощью оператора **L := M** мы переносим 5-й элемент в **L**, а с помощью **M := R** переносим 6-й элемент в **M**. Далее мы вводим 7-й элемент в **R**, после чего в тройке **L**, **M**, **R** у нас содержатся соответственно 5-й, 6-й и 7-й элементы, проверка которых на соответствие условию будет произведена на следующем шаге цикла.

На последнем шаге цикла последовательность будет в очередной раз «сдвинута» и в R будет введено число 0, после чего должен быть выведен результат. В связи с этим возможно два варианта попадания на этот этап:

a) в цикле был осуществлен выход через **break**. А это значит, что в переменных **L**, **M**, и **R** в данном случае как раз находится «неправильный зуб» и можно обойтись выводом значения выражения **(L – M) * (R – M) > 0**, которое как раз даст значение *false*:

```
writeln((L - M) * (R - M) > 0);
```

b) последовательность была проверена до конца, и выход из цикла был осуществлен по вводу в **R** нуля. Но здесь нужно учесть, что при этом в тройке **L**, **M**, **R** теперь находится «несуществующий зуб» (так как 0 не входит в саму последовательность и не должен учитываться при проверке), который может и не быть «правильным».

   Самый простой выход в этом случае – если переменная **R** равна 0, то заменить **R** на **L**. Это приведет к тому, что в выводимом на экран выражении (3) мы перемножаем не (1) и (2), а

(1) и (1), что повлечет вывод *true*, так как мы получаем произведение одинаковых ненулевых чисел, которое всегда положительно. Значит, перед выводом выражения на экран необходима следующая вставка:

```
  if R = 0 then R := L;
```

Код:

```
 1. program Saw;
 2.
 3. var
 4.   L, M, R: integer;
 5.
 6. begin
 7.   read(L, M, R);
 8.   while R <> 0 do begin
 9.     if (L - M) * (R - M) <= 0 then break;
10.     L := M;
11.     M := R;
12.     read(R)
13.   end;
14.   if R = 0 then R := L;
15.   writeln((L - M) * (R - M) > 0)
16. end.
```

К слову, данная задача может быть обобщена и на случай, когда длина вводимой последовательности не имеет ограничений снизу, то есть, последовательность может также быть пустой или содержать один член (для двух членов она будет работать корректно – это легко проверить). Для этого можно выполнить контроль ввода в зависимости от значений первых двух вводимых элементов и провести инициализацию значений для вывода необходимого для вырожденных случаев ответа.

Задача № 45. Проверить, является ли последовательность строго монотонной

Формулировка. Дана последовательность натуральных чисел, ограниченная вводом нуля. Проверить, является ли эта последовательность строго монотонной.

Решение. Эта задача – упрощенный вариант **задачи 32.** Вообще, эти две задачи логично было бы в данном сборнике поменять местами, однако она оказалась на этой позиции из-за тематического распределения задач.

Единственное отличие от **задачи 32** состоит в том, что в нашем случае члены последовательности вводятся с клавиатуры с помощью оператора **read()** – их не нужно добывать как цифры некоторого числа.

Воспользуемся формулой (II) из **задачи 32**:

$$p = delta_1 * delta_i$$

Напомним, что если произведение *p* отрицательно или равно нулю, то последовательность не строго монотонна, так как нашлись два *delta* разных знаков. Мы будем двигаться по последовательности в порядке ввода, слева направо, исследуя при этом знаки всех произведений *p*. Так как $delta_1$ присутствует в формуле в качестве константы, его значение необходимо вычислить заранее:

```
read(a, b);
delta := a - b;
```

В цикле мы будем на каждом шаге считывать один очередной член, поэтому необходимо «сдвигать последовательность» и считывать его в освободившуюся переменную. Так как в переменной **a** хранится левый член каждой пары, а в переменной **b** – правый член, то очередное число

мы будем считывать в переменную **b**. Так как ограничитель последовательности – ноль, то и цикл будет продолжаться до ввода в **b** нуля:

```
while b <> 0 do begin
  if delta * (a - b) <= 0 then break;
  a := b;
  read(b)
end;
```

Здесь в первом операторе тела цикла происходит проверка произведения каждых двух *delta* по уже упомянутой формуле (II) из **задачи 32**. Далее идет «сдвиг» правого члена текущей пары и ввод нового элемента вместо него. Проще говоря, если, например, у нас в **a** находился 1-й член последовательности, а в **b** – 2-й, то данным способом мы переносим 2-й член в переменную **a** и считываем 3-й член в переменную **b**, после чего можно проводить следующее сравнение.

Если введенный элемент – не ноль, то цикл продолжается, и исследуется знак произведения $delta_1$ и следующего вычисленного *delta*. Если условие монотонности нарушается на каком-либо шаге, то дальнейшая проверка бессмысленна, и можно переходить к выводу результата.

Каким же будет развитие событий после выхода из цикла?

1) Если выход был осуществлен через **break**, то есть по условию нарушения строгой монотонности, то можно вывести на экран значение выражения **delta * (a – b) > 0**, что даст ответ *false*;

2) Если цикл завершился по вводу нуля, то последовательность строго монотонна, и нужно, соответственно, выводить ответ *true*. Однако здесь мы сталкиваемся с **проблемой из задачи 45**, связанной с тем, что вводимый ноль не обрабатывается в основном цикле, так как не входит в последовательность, однако он вводится в *обрабатываемую переменную* **b**, чтобы можно было выйти из цикла. Однако из-за этого с помощью оператора **writeln(delta * (a – b) > 0)** мы можем получить неправильный ответ, так как в последовательность обрабатывается с вводимым нулем включительно.

   Например, последовательность 1 2 3 0 строго монотонна, хотя программа выдаст ответ *false*, потому что по выходе из цикла **delta** будет содержать число –1, **a** – число 3, **b** – число 0, и выражение **–1 * (3 – 0) > 0** – неверно.

На этот раз мы справимся с проблемой по-другому. Легко понять, что если после выхода из цикла **b = 0**, то последовательность строго монотонна, так как проверка прошла по всем *delta* вплоть до ввода ограничителя. Если же после выхода **b** отлично от 0, то был совершен выход по **break** в теле цикла и последовательность не является строго монотонной. Поэтому логично оформить вывод ответа так:

```
writeln(b = 0);
```

Кстати, в такой форме можно осуществить вывод и в **задачах 32, 43 и 44**, с некоторым, однако, изменением инициализации входных значений переменных.

Напоследок необходимо позаботиться о правильной обработке вырожденных случаев. Будем считать последовательность из единственного нуля не строго монотонной, в отличие от последовательности из одного члена. Она, кстати, итак уже будет обрабатываться корректно: в **a** вводится некоторое число, а в **b** вводится 0. При этом не осуществляется вход в основной цикл, и программа переходит к выводу выражения **b = 0**, которое верно.

Сделаем корректной обработку пустой последовательности. Во-первых, необходимо дать возможность ввода таковой, так как в нашем наброске требуется ввод минимум двух чисел. Для этого мы будем считывать сначала a, и если оно отлично от нуля, то считывать b:

```
read(a);
if a <> 0 then read(b) else b := 0;
```

При этом если **a = 0**, то мы обязательно присваиваем значение 0 переменной **b**, чтобы она была определена, и гарантированно не было входа в цикл. Однако в этом случае вывод выражения

b = 0 повлечет вывод *true*. Чтобы избежать этого, нужна еще одна проверка: если **a = 0**, то присвоить **b** натуральное число, отличное от 0 (например, 1), что повлечет вывод *false*:

```
if a = 0 then b := 1;
```

Код:

```
 1. program MonotonicSequence;
 2.
 3. var
 4.   a, b: word;
 5.   delta: integer;
 6.
 7. begin
 8.   read(a);
 9.   if a <> 0 then read(b) else b := 0;
10.   delta := a - b;
11.   while b <> 0 do begin
12.     if delta * (a - b) <= 0 then break;
13.     a := b;
14.     read(b)
15.   end;
16.   if a = 0 then b := 1;
17.   writeln(b = 0)
18. end.
```

Стоит отметить, что на первом шаге в основном цикле значение **a** и **b** еще не изменено, и грубо говоря, **delta** в строке 12 умножается само на себя. Это нужно для того, чтобы обеспечить правильную обработку последовательности из двух одинаковых чисел, которая не является строго монотонной. Например, если на вход подается последовательность *k k* 0, *k* – некоторое натуральное число, то **delta** будет равно 0 (строка 10), и при входе в цикл будет осуществлен выход по **break** (строка 12), что повлечет вывод *false*, так как **b** отлично от 0.

Если бы мы в цикле сначала осуществили сдвиг последовательности и ввод третьего члена (то есть, переместили бы строки 13-14 на одну позицию вверх, а строку 12 поместили бы после них), то по выходе из цикла через **break b** уже было бы равно 0, что повлекло бы вывод *true*, а это неверно.

Задача № 46. Вывести на экран n-ное число Фибоначчи

Формулировка. Дано натуральное **n** (которое также может быть равно 0). Вывести на экран **n**-ное число Фибоначчи.

Примечание: последовательность чисел Фибоначчи задается следующей рекуррентной формулой:

$$F_n = \begin{cases} 0, \text{ если } n = 0 \\ 1, \text{ если } n = 1 \\ F_{n-1} + F_{n-2}, \text{ если } n \geq 2 \end{cases}$$

То есть, нулевой член последовательности – это число 0, 1-й член – число 1, а любой другой член, начиная со 2-го, является суммой двух предыдущих. Например, $F_2 = F_1 + F_0 = 1 + 0 = 1$, $F_3 = F_2 + F_1 = 1 + 1 = 2$ и т. д.

Решение. Найдем несколько первых чисел Фибоначчи:

$F_2 = F_1 + F_0 = 1 + 0 = 1$;

$F_3 = F_2 + F_1 = 1 + 1 = 2$;

$F_4 = F_3 + F_2 = 2 + 1 = 3$;

$F_5 = F_4 + F_3 = 3 + 2 = 5$;

Легко заметить, что при их последовательном вычислении нам не нужно «расписывать» слагаемые по определению, и чтобы получить очередной член последовательности, достаточно на каждом шаге складывать два предыдущих полученных результата.

Так как нулевой и первый члены последовательности не вычисляются и даются как часть определения, будем полагать их заранее известными. Обозначим их идентификаторами **fib0** и **fib1**. По примеру нахождения первых членов последовательности посчитаем количество операций, необходимое для вычисления каждого члена (считая, что предыдущие члены неизвестны). Легко увидеть, что для вычисления 2-го члена (при известном 1-ом и нулевом членах) необходима одна операция сложения, 3-го – две операции сложения и т. д. Видно, что по этим же правилам для вычисления **n**-ного члена необходимо выполнить **(n – 1)** операций.

Теперь можно начать писать программу. Сначала нам необходимо ввести значение **n** и выполнить инициализацию значений нулевого и первого чисел Фибоначчи, так как мы считаем их заранее известными:

```
readln(n);
fib0 := 0;
fib1 := 1;
```

Далее нам необходимо организовать цикл, в котором на каждом шаге переменные **fib0** и **fib1** будут получать следующие значения в последовательности чисел Фибоначчи. То есть, например, если в **fib0** и **fib1** будут находиться значения, соответственно, **(n – 2)**-го и **(n – 1)**-го членов последовательности Фибоначчи, то после одного шага цикла они будут содержать значения **(n – 1)**-го и **n**-го членов. Для этого можно создать некую вспомогательную переменную **fib**, в которую поместить результат сложения **fib0** и **fib1**, после чего в **fib0** у нас будет значение **(n – 2)**-го члена, в **fib1** – **(n – 1)**-го, а в **fib – n**-го. Теперь нужно только скопировать значение **fib1** в **fib0** и **fib** в **fib1**, после чего значение переменной **fib** на этом шаге уже не имеет значения. С учетом того, что мы уже посчитали необходимое количество повторений для получения необходимого результата, цикл будет выглядеть так:

```
for i := 1 to n - 1 do begin
  fib := fib1 + fib0;
  fib0 := fib1;
  fib1 := fib
end;
```

Такой метод решения общей задачи, основанный на использовании в ней решений задач с меньшей размерностью исходных данных (в данном случае под размерностью понимается величина **n**), называется *динамическим программированием*.

Если говорить конкретнее, то мы применили метод *восходящего динамического программирования*, основывающийся на решении задач сначала минимальной размерности с постепенным получением решений более обширных задач. Этот метод наиболее оптимален в плане реализации, быстродействия и используемой памяти.

Однако далеко не для всех задач, решаемых с помощью динамического программирования, можно выяснить, решения каких подзадач потребуются для решения данной. Для этого существует метод *нисходящего динамического программирования*, с которым мы познакомимся позже.

Другой пример нисходящего динамического программирования – вычисление факториала (**задача 28**). Чтобы вычислить **n!**, необходим вычисленный **(n – 1)!** и т. д.

Итак, вернемся к текущей задаче. Ранее было сказано, что по исчерпании **n – 1** шагов цикла в переменной **fib1**, которая пойдет на вывод в программе, будет храниться значение $\mathbf{F_n}$. Проверим корректность обработки граничных значений (в частности, когда **n = 0, 1, 2**):

1) При **n = 0** границы цикла будут в отрезке [1, 0 – 1]. При этом значение правой границы зависит от типа переменной **i**, так как компилятор, дабы избежать ошибок, при вычислении «расширяет» тип выражений, означающих границы цикла.

   Если **i** будет объявлено типа **byte**, то выражение 0 – 1 даст в результате 255 (так как все числовые типы в языке Pascal большинство компиляторов считает кольцевыми), что вызовет длинную цепочку вычислений, а это неверно. Конечно, можно объявить **i** типа **integer**, и тогда границы цикла будут в отрезке [1, –1] и вход не будет осуществлен, но мы поступим иначе, стараясь сохранить память для переменных.

   Так как при вычислении важно лишь количество повторений, мы можем сдвинуть отрезок [1, *n* – 1] на одну позицию правее на числовой прямой, и тогда цикл будет проходить от 2 до **n**, что поможет отсеять вход в цикл при вводе 0 в качестве **n**, так как невозможен цикл по всем **i** от 2 до 0.

   Однако тогда мы столкнемся с другой проблемой: при вводе 0 будет выведено значение **fib1**, которой было присвоено число 1 при инициализации. Справиться с проблемой можно, присвоив **fib1** значение 0, если **n = 0**:
   ```
   if n = 0 then fib1 := 0;
   ```

2) При **n = 1** (с учетом принятых в предыдущем пункте изменений) входа в цикл не будет, и на экран выведется неизменное значение **fib1**, равное 1;

3) При **n = 2** произойдет вход в цикл, где **i** будет изменяться от 2 до 2 (то есть, в этом случае будет выполнен единственный шаг), в котором будет вычислено значение **fib** = **fib1** + **fib0** = 1 + 0 = 1, которое будет скопировано в **fib1** и выведено на экран. Нетрудно понять, что дальнейшая подстановка значений не требуется, так как корректность циклической конструкции мы уже доказали.

Верхние значения мы не проверяем, так как не существует наибольшего номера в последовательности чисел Фибоначчи (хотя понятно, что корректность вычислений ограничена «вместимостью» типа **integer**, и при его превышении в вычислениях числа уже будут неправильными).

Код:

```
 1. program FibonacciNumbers;
 2.
 3. var
 4.   fib0, fib1, fib: integer;
 5.   i, n: byte;
 6.
 7. begin
 8.   readln(n);
 9.   fib0 := 0;
10.   fib1 := 1;
11.   for i := 2 to n do begin
12.     fib := fib1 + fib0;
13.     fib0 := fib1;
14.     fib1 := fib
15.   end;
16.   if n = 0 then fib1 := 0;
17.   writeln(fib1)
18. end.
```

Задача № 47. Вывести на экран сумму чисел Фибоначчи до n-ного включительно

Формулировка. Дано натуральное **n** (которое также может быть равно 0). Вывести на экран сумму чисел Фибоначчи до **n**-ного включительно. Например, при **n = 3** нам необходимо получить сумму 0-го, 1-го, 2-го и 3-го членов последовательности.

Решение. Задача основана на предыдущей, так как здесь нам тоже необходимо найти каждое число Фибоначчи до **n** включительно, однако теперь мы должны прибавлять найденные числа к некоторой переменной суммы (**sum**), которая потом будет выведена на экран.

Используем код предыдущей задачи:

```
readln(n);
fib0 := 0;
fib1 := 1;
for i := 2 to n do begin
  fib := fib1 + fib0;
  fib0 := fib1;
  fib1 := fib
end;
if n = 0 then fib1 := 0;
writeln(fib1);
```

Чтобы переделать этот код по текущему назначению, мы должны добавить в цикл прибавление найденного числа Фибоначчи к переменной **sum**. Например, так:

```
for i := 2 to n do begin
  fib := fib1 + fib0;
  sum := sum + fib;
  fib0 := fib1;
  fib1 := fib
end;
```

Кроме того, следует исправить вывод ответа, так как нам необходимо вывести не последнее найденное число Фибоначчи, а сумму найденных чисел:

```
writeln(sum);
```

Очевидно, что вход в цикл не происходит при **n = 0** и **n = 1**. Следовательно, правильную обработку этих случаев мы должны обеспечить инициализацией значений переменной **sum**, как мы это делали в предыдущей задаче.

Так как сумма нулевого и 1-го чисел Фибоначчи равна 1, то **sum** можно инициализировать значением 1. При входе в цикл первые два числа уже обработаны, поэтому при вводе **n >= 2** накопление суммы также будет верным. Но очевидно, что в случае **n = 0** необходимо инициализировать переменную **sum** значением 0. Реализовать эти два варианта можно так:

```
if n = 0 then sum := 0 else sum := 1;
```

Код:

```
 1. program FibonacciNumbersSum;
 2.
 3. var
 4.   fib0, fib1, fib, sum: integer;
 5.   i, n: byte;
 6.
 7. begin
 8.   readln(n);
 9.   fib0 := 0;
10.   fib1 := 1;
```

```
11.   if n = 0 then sum := 0 else sum := 1;
12.   for i := 2 to n do begin
13.     fib := fib1 + fib0;
14.     sum := sum + fib;
15.     fib0 := fib1;
16.     fib1 := fib
17.   end;
18.   writeln(sum)
19. end.
```

Задача № 48. Вывести на экран все числа Фибоначчи до n-ного включительно

Формулировка. Дано натуральное **n** (которое также может быть равно 0). Вывести на экран все числа Фибоначчи до **n**-ного включительно.

Решение. Задача основана на **задаче 46**. В данном случае нам необходимо лишь выводить каждое найденное число Фибоначчи на экран. Мы можем легко получить решение этой задачи из **задачи 46** или **задачи 47**.

Опишем основные фрагменты программы. Так как нулевой член последовательности выводится при любом возможном **n**, то его можно вывести на экран сразу после ввода **n** (или до, что не имеет значения). Затем, если **n** отлично от нуля, выводим на экран 1-ый член (так как вывод в цикле остальных членов происходит при **n >= 2**):

```
readln(n);
fib0 := 0;
fib1 := 1;
write(fib0, ' ');
if n <> 0 then write(fib1, ' ');
```

Далее необходимо добавить вывод текущего найденного члена в цикл:

```
for i := 2 to n do begin
  fib := fib1 + fib0;
  write(fib, ' ');
  fib0 := fib1;
  fib1 := fib
end;
```

Так как мы выводим все результаты в самом цикле, на этом программа заканчивается.

Код:

```
 1. program FirstNFibonacciNums;
 2.
 3. var
 4.   fib0, fib1, fib: integer;
 5.   i, n: byte;
 6.
 7. begin
 8.   readln(n);
 9.   fib0 := 0;
10.   fib1 := 1;
11.   write(fib0, ' ');
12.   if n <> 0 then write(fib1, ' ');
13.   for i := 2 to n do begin
14.     fib := fib1 + fib0;
```

```
15.     write(fib, ' ');
16.     fib0 := fib1;
17.     fib1 := fib
18.   end
19. end.
```

Задача № 49. Проверить баланс круглых скобок в символьном выражении

Формулировка. Дана последовательность символов длины **n** (**n >= 1**). Проверить баланс круглых скобок в этом выражении. Например, при вводе выражения (())() программа должна сообщить о правильности расстановки скобок, а при вводе выражения ((()) – о неправильности.

Примечание: сбалансированной скобочной записью называется символьное выражение, в котором каждой открывающей скобке соответствует закрывающая скобка правее и наоборот, каждой закрывающей скобке соответствует открывающая скобка левее.

Так как мы вводим последовательность произвольных символов, в которой учитываются только круглые скобки, то между знаками скобок может находиться любая символьная информация, в силу чего корректная программа может проверять баланс скобок в арифметических выражениях, тексте и т. д. Например, выражение $(7y + 1)(17 – (x + 3))$ – правильное, а $(146x + 18(y + 9)$ – неправильное, что сможет распознать программа.

Решение. Представим себе посимвольный ввод скобочного выражения с клавиатуры (когда уже введено некоторое количество символов) и подумаем, какие выводы можно сделать на данном этапе (для простоты восприятия будем рассматривать выражения, состоящие только из скобок):

1) (((

Сейчас «как бы» мы видим начало скобочного выражения и не знаем, какие символы следуют далее. Какие выводы можно сделать на этом этапе? Имеющееся выражение содержит лишние открывающие скобки и его можно как сбалансировать, если дописать две закрывающие скобки, так и нарушить, если оставить в том же виде или применить множество других комбинаций. **Вывод**: если имеются лишние открывающие скобки, то выражение еще может быть сбалансировано.

2) (()))

Это выражение содержит явное нарушение баланса скобок, которое уже не может быть скомпенсировано добавлением любых скобочных комбинаций справа, так как не всем закрывающим скобкам соответствует по одной открывающей скобке левее. Отсюда **вывод**: если в выражении появилась хотя бы одна лишняя закрывающая скобка, то выражение «неправильное» и дальнейшая проверка бессмысленна.

Приступим к реализации этих рассуждений. Заведем счетчик **count** для подсчета открывающих и закрывающих скобок: при вводе открывающей скобки будем увеличивать его на 1, а при вводе закрывающей – уменьшать на 1.

Нам нужно создать переменную **c** символьного типа **char**, в которую мы будем последовательно вводить все символы нашего выражения. Стоит отметить, что в тип **char** также входит пробел, что влияет на значение длины вводимой последовательности. Например, длина **n** вводимого выражения `(7y + 1)(17 – (x + 3))` равна 22 (пробелы выделены красным цветом).

После ввода **n** мы входим в цикл из **n** повторений, в котором вводим в **c** очередной символ. Полагаясь на предыдущие рассуждения, мы увеличиваем **count** на 1, если **c = '('** и уменьшаем на 1, если **c = ')'**:

```
  readln(n);
  count := 0;
  for i := 1 to n do begin
    read(c);
```

```
    if c = '(' then inc(count);
    if c = ')' then dec(count)
  end;
```

Примечание: функция **dec(x)** уменьшает значение переменной **x** числового типа на 1.

Кстати, так как любой любая переменная не может иметь сразу два каких-либо значения, мы можем поместить второй условной оператор цикла в **else**-блок первого, что будет выглядеть логичнее, но мы не будем этого делать, чтобы повысить читаемость кода.

Отметим, что ввод **n** осуществляется с помощью **readln()**, так как он требует ввод **enter**'а в качестве ограничителя. При вводе **n** через **read()** далее следующий пробел или **enter** будет включен непосредственно во вводимую последовательность, что повлечет ошибку. Кроме того, нельзя разделять лишними пробелами или **enter**'ами символы последовательности при вводе, так как они не игнорируются при вводе в переменные типа **char** и должны быть включены в последовательность (при этом каждый пробел добавляет к длине 1, а каждый **enter** – 2).

Вернемся к разбору. Как же быть, если некоторый начальный фрагмент вводимого выражения станет заведомо неправильным, то есть, если в нем появятся лишние закрывающие скобки? Но тогда при появлении лишней («некомпенсируемой») закрывающей скобки переменная **count** станет равна –1, что можно оформить как условие выхода из цикла и поместить после первых двух операторов сравнения:

```
  if count = -1 then break;
```

Какие результаты мы получим по завершении цикла?

1) Цикл прошел по всем символам, но были найдены лишние открывающие скобки (то есть, **count > 0**), компенсирования которых мы ожидали, однако они так и не были скомпенсированы и скобочная последовательность неправильная;
2) Цикл прошел по всем символам (то есть, не было выхода), причем количество скобок обоих видов равно (то есть, **count = 0**) и скобочная последовательность, соответственно, правильная;
3) Был осуществлен выход из цикла (то есть, нашли «некомпенсируемую» закрывающую скобку и **count = –1**) и последовательность неправильная.

Выходит, правильный ответ даст вывод выражения **count = 0** (оно истинно во 2-ом случае и ложно в 1-ом и 3-ем):

```
  writeln(count = 0);
```

Код:

```
 1. program BracketSequence;
 2.
 3. var
 4.   count: integer;
 5.   i, n: byte;
 6.   c: char;
 7.
 8. begin
 9.   readln(n);
10.   count := 0;
11.   for i := 1 to n do begin
12.     read(c);
13.     if c = '(' then inc(count);
14.     if c = ')' then dec(count);
15.     if count = -1 then break
16.   end;
```

```
17.   writeln(count = 0)
18. end.
```

Задача № 50. Вычислить экспоненту с заданной точностью

Формулировка. Дано действительное число **x**. Вычислить значение экспоненциальной функции (то есть, показательной функции e^x, где e – математическая константа, $e \approx 2,718281828459045$) в точке **x** с заданной точностью **eps** с помощью ряда Тейлора:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

Примечание 1: показательными называются функции вида a^x, где a – некоторое действительное число, x – независимая переменная, являющаяся показателем степени.

Примечание 2: ряд Тейлора – это представление функции в виде суммы (возможно, бесконечной) некоторых других функций по особым правилам (требующим детального математического обоснования, что в данном случае нам не нужно).

Решение. Не вникая в теоретическую часть и полагая представленную формулу корректной, попробуем разобраться в том, что же нам необходимо сделать для того, чтобы решить эту задачу:

1) Нам дана некоторая точка на оси *Ox*, и мы должны вычислить значение функции e^x в этой точке. Допустим, если **x = 4**, то значение функции в этой точке будет равно $e^4 \approx 2,71828^4 \approx 54,598$;

2) При этом вычисление необходимо реализовать с помощью заданной бесконечной формулы, в которой прибавление каждого очередного слагаемого увеличивает точность результата;

3) Точность должна составить вещественное число **eps**, меньшее 1 – это означает, что когда очередное прибавляемое к сумме слагаемое будет меньше **eps**, то необходимо завершить вычисление и выдать результат на экран. Это условие обязательно выполнится, так как математически доказано, что каждое следующее слагаемое в ряде Тейлора меньше предыдущего, следовательно, бесконечная последовательность слагаемых – это бесконечно убывающая последовательность.

Теперь разберемся с вычислением самого ряда. Очевидно, что любое его слагаемое, начиная со 2-го, можно получить из предыдущего, умножив его на **x** и разделив на натуральное число, являющееся номером текущего шага при последовательном вычислении (примем во внимание то, что тогда шаги нужно нумеровать с нуля). Значение **x** нам известно на любом шаге, а вот номер текущего шага (будем хранить его в переменной **n**) придется фиксировать.

Создадим вещественную переменную **expf** (от англ. *exponential function* – экспоненциальная функция) для накопления суммы слагаемых. Будем считать нулевой шаг уже выполненным, так как первое слагаемое в ряду – константа 1, и в связи с этим **expf** можно заранее проинициализировать числом 1:

```
expf := 1;
```

Так как мы начинаем вычисления не с нулевого, а с первого шага, то также нужно инициализировать значения **n** (числом 1, так как следующий шаг будет первым) и **p** (в ней будет храниться значение последнего вычисленного слагаемого):

```
n := 1;
p := 1;
```

Теперь можно приступить к разработке цикла. С учетом заданной точности **eps** условием его продолжения будет **abs(p) >= eps**, где **abs(p)** – модуль числа **p** (модуль нужен для того, чтобы не возникло ошибки, если введено отрицательное **x**).

В цикле необходимо домножить **p** на **x** и доделить его на текущий номер шага **n**, чтобы обеспечить реализацию факториала в знаменателе, после чего прибавить новое слагаемое **p** к результату **expf** и увеличить **n** для следующего шага:

```
while abs(p) >= eps do begin
  p := p * x / n;
  expf := expf + p;
  inc(n)
end;
```

После выхода из цикла нужно осуществить форматированный вывод результата **expf** на экран с некоторым количеством цифр после точки, например, пятью. Отметим, что если при этом введенное **eps** содержало меньше 5 цифр после точки, то сформированное значение **expf** будет, соответственно, неточным.

Код:

```
 1. program ExpFunc;
 2.
 3. var
 4.   x, eps, expf, p: real;
 5.   n: word;
 6.
 7. begin
 8.   readln(x, eps);
 9.   expf := 1;
10.   n := 1;
11.   p := 1;
12.   while abs(p) >= eps do begin
13.     p := p * x / n;
14.     expf := expf + p;
15.     inc(n)
16.   end;
17.   writeln(expf:0:5)
18. end.
```

Содержание