

Kostanay State Pedagogical Institute
Science and Mathematical Faculty
Informatics and Computer Technologies Department

A.D.Tsyganova

V.V. Danilova

ALGORITHMIZATION AND PROGRAMMING

Study Manual
for students of the specialty
5B011100 «Informatics»

Kostanay

2018

UDC 004 (075.8)
BBC 32.973-018 я 73
T 94

Authors: *Tsyganova A.D.*, senior lecturer of the department of informatics and computer technologies, docent of KSPI
Danilova V.V., doctor of philosophy (PhD), candidate of pedagogical sciences, senior lecturer of the Department of the foreign languages, KSPU

Reviewers: *Sukhov M.V.*, candidate of technical sciences, the dean of the Science and Mathematical Faculty, KSPU
Ismailov A.O., candidate of technical sciences, docent of the department of software, KSU after A.Baytursynov

Tsyganova A.D.

T 94 Algorithmization and programming: study manual for students of specialties 5B011100 Informatics. A.D. Tsyganova, V.V. Danilova – Kostanay.: KSPU, 2018. – 99 p.

ISBN 978-601-7934-52-1

The manual is written in accordance with the requirements of the State Program for the Development of Education and Science of the Republic of Kazakhstan for 2016-2019 - a phased transition of the formation of the Republic of Kazakhstan to a trilingual education and is intended for students of polyglot groups of specialty 5B011100 "Informatics".

This manual is a guide to mastering the skills of developing and implementing algorithms for solving problems in the programming environment in English. At the beginning of each topic, the manual contains the necessary theoretical material; an analysis of the solution of demonstration examples is conducted, the methodology for developing algorithms for solving these examples accompanied by the listing of ready-made programs or basic program fragments, is considered; made a selection of interesting practical tasks for the independent decision on each topic in the required volume given guidance, methodological recommendations for solving tasks. Particular attention is paid to the topics: procedural-modular programming technology, input-output data files, algorithms on graphs.

UDC 004 (075.8)

BBC 32.973-018 я 73

Approved by the Academic Council of the Kostanay State Pedagogical Institute
Protocol No7 dated 31.05.2018.

ISBN 978-601-7934-52-1

©KSPU, 2018
©Tsyganova A.D., Danilova V.V., 2018

CONTENTS

Introduction.....	4
1 General information on the programming language Pascal	6
1.1 Structure of the program in Pascal language	6
1.2 Programming environment	8
1.3 Standard data types. Variables, constants	11
1.4 Debugging the program	15
1.5 Standard numeric functions	16
2 Fundamentals of programming in the environment of Pascal	18
2.1 Programming linear algorithms	18
2.2 Programming of branching algorithms	20
2.3 Programming of cyclic algorithms	25
2.3.1 Cycle with parameter.....	25
2.3.2 Cycles with the condition	28
2.4 Procedural-modular technology for solving problems	31
2.4.1 Procedures	31
2.4.2 User functions	34
2.4.3 Recursion	36
3 Types and Structures of Data	38
3.1 File data type	38
3.2 Linear arrays	40
3.2.1 Search in an array	44
3.2.2 Shifts, reversal, and sorting of an array.....	45
3.3 Two-dimensional arrays	52
3.3.1 Square matrix	53
3.3.2 Processing two-dimensional arrays.....	57
3.4 Line data type	61
3.5 Enumerated data type	66
3.6 Sets	67
3.7 Graph model of the algorithm	70
4 Standard Modules	79
4.1 Text screen mode	79
4.2 Graphical display mode	80
4.2.1 Text output in graphical mode	86
4.2.2 Creating function graphs	87
4.2.3 Modeling the movement of objects	96
The list of references	99

INTRODUCTION

Algorithmization and programming are the most important sections of computer science, allowing to form logical thinking, system, operational approach to solving problems of a different nature and complexity.

The programs for the first computers were written in the language of the machine instructions and represented binary numbers. The set of such commands was determined only by the capabilities of the processor used. Although the commands were quite primitive, they could be used to describe any algorithm. Of course, to solve a complex problem, recording a program becomes very cumbersome.

Simplifying the text of the program means to simulate a more powerful language. This direction has led to the abundance of programming languages that we currently have.

Initially, programmers often simply rewrote repetitive actions in a notebook for further use. Thus, even in the forties, the idea of using subprograms began to appear. Over time, there were special programs (linkers), which, according to a special code, could write to the specified place of RAM the required subroutine stored in the library of subroutines.

At the same time, the presentation of programs changed. Instead of machine codes, the symbols developed for them began to be written down. Instead of numbers of memory cells, where the data was stored, they began to use symbolic names - the notion of a variable appeared.

Over time, subroutines began to give meaningful names, it became possible to write numerical expressions in a form familiar to everyone. The first high-level languages appeared. Their main distinguishing features from low-level languages were the description of algorithms quite close to the natural language.

Although there are currently thousands of programming languages, they all have something in common. Each language consists of three components: alphabet, syntax and semantics.

An alphabet is a set of symbols used in a given language.

Syntax is a set of rules for constructing constructs in a language. Syntax sometimes includes the alphabet of the language. There are different ways of describing the language. The most common of these are syntax diagrams and the extended Backus-Naur form (EBNF).

Semantics is the semantic interpretation of each syntactic construction in a language. If syntactic errors can be detected by a language translator, then the responsibility for the semantic errors lies entirely with the programmer.

Currently, there is no generally accepted strict classification of programming languages. Some conditionally share all languages in levels - low and high. High-level languages can be declarative (Prolog, Lisp) and procedural-oriented (C / C ++, Pascal, Ada). Procedural languages evolve into object-oriented languages. The classification of programming languages into educational and professional languages is also highlighted. Now such a classification is very conventional, since most languages have become universal. Programming languages continue to evolve. One of the programming languages

used in education is Pascal. It was developed in 1968-1970 by the Swiss scientist Niklaus Wirth. Pascal initially was not widely used, although it served as the basis for the development of other languages (Modula-2, Ada). Only with the advent of its expansion in the 1980s - the Turbo-Pascal language for IBM, it gained popularity. The first version of Turbo Pascal appeared in 1983, and already in 1984 it was replaced by the second version, by the autumn of 1986 a third version appeared, more convenient in operation. The fourth version (1988) introduced Turbo Pascal in a new form (the emergence of a new environment, the compiler became built-in). In the autumn of the same year, the fifth version was developed, which had a built-in debugger, and in 1989, version 5.5 appeared, which allowed switching to object-oriented programming. The sixth version provided multi-window and multi-file mode, the use of "mouse", the use of object-oriented programming, had built-in assembler. In 1992, Borland International released two programming packages in Pascal language: Borland Pascal 7.0 and Turbo Pascal 7.0. It was declared for the official programming language for high school students in the United States intending to specialize in computer technology and programming at American universities, has been and remains one of the most popular programming languages used in programming contests at various levels.

The first programming language must be required to be studied by the student. It is necessary that the student has a clear idea of what his program does at each step, and be able to write down algorithms in a strict formal language. The more error messages students see from the compiler, and the more of these messages they understand, the more fundamental knowledge about programming they will receive. Pascal has these properties. It is especially important that in it there is a check on whether the array index belongs to an admissible set of values. This is very useful for schoolchildren. But Pascal is rarely used in practice, and the specialist in pascal is not much in demand in the labor market; for real work it is necessary to learn a more popular language (Java, C / C ++, PHP, etc.).

1 General information on the language of Pascal

1.1 Structure of the programme on Pascal language

Language Pascal is an algorithmic language, i.e. the correct program in this language is a formal record of some algorithm, a finite sequence of actions leading to the solution of a certain problem. In accordance with this principle, the Pascal program consists of two parts: a description of the sequence of actions to be performed and descriptions of the data with which the operations work.

In general, the structure of the Pascal program is as follows:

```
{I. Title of the programme}  
program <name of the programme>;  
{II. Section indicating the used modules}  
uses <the list of modules>;  
{III. Section of descriptions}  
label <description of labels>;  
type <description of types>;  
const <description of constants>;  
var <description of variables>;  
procedure <description of procedures>;  
function <description of functions>;  
{V. Section of operators}  
begin  
<operators>  
end.
```

The title of the programme in the environment of *Pascal* is not obligatory.

The section for specifying used modules begins with the keyword *Uses* and is also an optional section. It is used in cases when the program requires constants, types, variables, procedures and functions defined in other modules, except the *System* module.

The standard *System* module is used by default everywhere and you do not need to specify it in the *Uses* clause. This module supports tools such as file based Input / Output, handling of lines, floating point operations, dynamic memory allocation, and etc.

The description section, just like the previous sections, is optional in the program.

Tags can precede any operator of the program and are separated from the operators by a colon. Labels are used together with *Goto* jump operators, where the label is written without a colon:

```
label 1, a;  
....  
goto 1;  
1: y:=y+1;
```

Using labels and *Goto* operators in most cases contradicts the principles of structured programming and therefore it is recommended to avoid using these

constructs in programs without special need. Many types of Pascal can be divided into two groups:

- 1) standard (predefined) types;
- 2) user-defined types (user-based types).

The names of *standard types* are predefined by identifiers, which are described in the standard *System* module, which by default is connected to the list of used modules of each program and each user module.

Custom types are the additional abstract types, the characteristics of which can be defined by the programmer independently.

Variables description introduces a set of data with which actions are performed in the operating unit. The variable is denoted by the identifier; its type is associated with each variable, which defines the set of valid values for this variable and the set of permissible operations.

An *identifier* is a name freely chosen by the programmer for program elements (procedures, functions, constants, variables and data types), formed from letters and digits.

The description of a procedure or function defines a part of the program as a separate syntactic unit and associates a name with it. Actions contained in a procedure or function can be performed ("called") by specifying its name. The section of operators is the only mandatory section in the program structure.

The simplest program can look like:

```
Begin
Writeln('Hello, friend!');
End.
```

The Pascal alphabet consists of the following characters:

- upper and lowercase Latin letters and the underscore A, B, C, ..., X, Y, Z, a, b, c, ..., x, y, z, _ - to form identifiers and service words;
- Arabic numerals 0, 1, 2 9 - for writing numbers and identifiers;
- special characters + - * / = < > , . ; : @ ' () [] { } # \$ ^ for constructing the signs of operations, expressions, comments.

An *expression* is a formal rule for computing a certain value. Expressions are constructed from operands, operation characters, and parentheses. Operands are "elementary" values: variables, fields of records, array elements, function calls, etc.

When writing numeric expressions in Pascal, the following arithmetic operations are used:

- + addition
- subtraction
- * multiplication
- / division

Div - integer division

Mod - remainder of the division

When calculating the value of a numerical expression, operations are performed in decreasing order of priority. The brackets in the expression are used to change the normal order of processing operations.

Div and Mod operations are applied only to operands of the whole type. Variables in the Pascal program are information objects designed to store values of a certain type. Within a given type, a variable can have any value that changes during the execution of the program.

Objects that are externally similar to variables, but which cannot change their value, are called constants:

```
const
  a=1;
  b=1000;
  maxreal=1.7e38;
  pi=3.14;
  min=0;
  max=100;
  center=(max-min) div 2;
  message='out of memory';
```

Operators of the Pascal language are divided into two groups:

- 1) simple operators;
- 2) structural operators.

Operators are separated from each other by a delimiter (;) ; it is not part of the operator, therefore, after the last statement of the program and after the last statement in the compound statement, it is not necessary to put a semicolon before the service word End.

Simple operators include:

- operator of assignment;
- operator of a procedure call;
- operator of the transition.

Structural operators include:

- compound operator;
- conditional operator;
- cycle operators;
- connection operator.

In the program, the most commonly used is the assignment operator
variable = expression;

The execution of the assignment operator results in the calculation of the value determined by the expression and the assignment of this value to the variable.

1.2 The programming environment

After launching the Free Pascal programming environment, the appearance of the screen will be as shown in Figure 1.1.

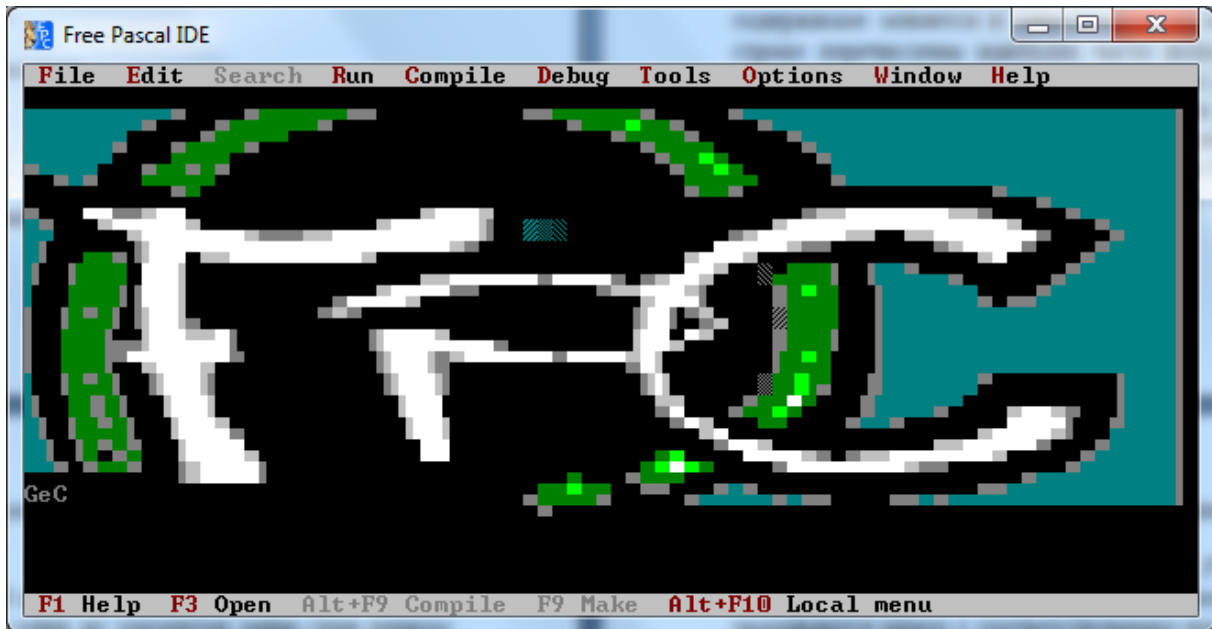


Figure 1.1 Appearance of the screen in the Free Pascal environment

The bottom line of the screen is a string of prompts. Its contents change depending on the actions performed. This line lists the most commonly used keys.

The top line of the screen is the main menu of the integrated environment. Each element of this menu is another menu (submenu). Main menu.

The F10 key is used to exit to the main menu. Clicking it, you will see that one of the menu items (when the first click is the File submenu) acquires a green background. From the main menu, you can move using the left and right arrow keys (or with the mouse).

To select the desired submenu, press Enter. If at some point you work with the menu, you decide to abandon the selected actions, you will need to press the Esc key.

Create a new program. Press F10 to exit to the Main menu. Select the File menu. A submenu will appear, shown in Figure 1.2.

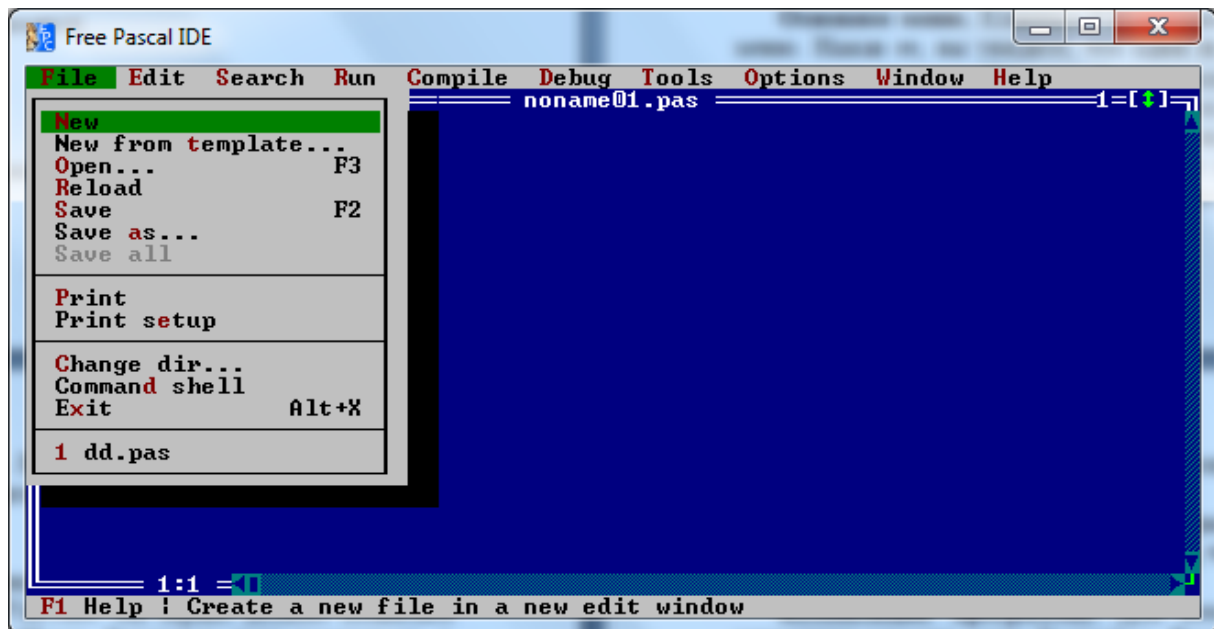


Figure 1.2 The main menu of the program

Then select New (File-New). A new window with the title Noname 00.pas should appear on the screen. The title of the window shows the file in which the program text is stored. In the Pascal environment, you can edit several programs at the same time, the text of each of them will be located in a separate window.

Saving the program. To save the program, select *the File-Save* or *File-Save as ...* menu item. In the first case, the program is saved under a name that coincides with the title of the corresponding window. In the second case, a dialog box appears where under the *Save as ...* line, specify a new file name and press Enter.

Instead of selecting *File-Save*, you can simply press F2. These keys are called "hot". They are usually indicated next to the corresponding menu item and serve for quicker access to the desired menu item.

Loading the program. To download the program, select the menu command *File-Open* (or press the F3 hotkey). In the dialog box that appears, type the name of the file in the *Name* line or move to the file list with the Tab key and select the desired file.

Working with windows. The *Free Pascal* environment allows you to simultaneously edit several programs or even ordinary text files. In order to go to the desired window, you can press the Alt + number keys or move from one window to another until the desired one appears, using the F6 key. In the environment of modern versions of *Pascal*, it is possible to change the configuration of windows. After pressing the key combination Ctrl + F5, you can use the arrow keys to change the size of the current window or, with the same keys while holding down the Shift key, move the window to the desired place on the screen. To return to editing mode, press Esc or Enter.

While working in the environment, some windows or parts of them may overlap with other windows. To make all windows visible, select *Window-Tile* from the menu.

Compiling of the program. To compile the program, you must set the *Destination* menu item. There are two possible values: *Memory* and *Disk*. Each time this menu item is selected, its value changes to the opposite. Next, select the *Compile-Compile* menu item (or press Alt + F9).

Starting the program. To start the program, you need to press Ctrl + F9 or select *Run-Run* from the menu. In this case, the program is compiled first, and then, if there are no errors, it is launched. After the program is finished, the integrated environment returns to the editing mode. In order to view the results of the program execution, you have to press Alt + F5, and to return to edit mode, press any key.

If an error occurs during the compilation process, a corresponding message is displayed and the cursor moves to the intended place of the program where this error is allowed.

Editing the program. When editing, *insert* and *substitute* modes are distinguished (switching with the Insert key).

Exit from Turbo Pascal. To exit the environment, select the *File-Exit* menu item or press Alt + X.

1.3 Standard data types. Variables. Constants

Any variable is characterized by its **type** in Pascal. A type is understood as the set of values that a variable can take, and the set of operations allowed on a given variable.

Pascal is a typed, or static language. This means that the type of the variable is determined when it is described. A variable can only participate in operations allowed by a type.

Pascal has a developed system of *types*. Based on a small number of standard types, a programmer can construct data of arbitrary structure and complexity.

The basic types in the type system are **simple types**. Simple types are divided into *ordinal* and *real* data types. An ordinal type is a data type whose range of values is an ordered countable set. In any ordinal type, for every value except the first, there is a previous value, and for each value, except the last, there is a subsequent value.

There are standard functions that allow determining the corresponding values for a given value in Pascal,:

- the function `Pred (x)` determines the value preceding x;
- the function `Succ (x)` determines the value following x;
- the `Ord (x)` function returns the ordinal number of x.

Whole types

This group of types denotes sets of integers in different ranges.

The integer type	Value range	Memory size
Shortint	-128..127	1 byte
Integer	-32768...32767	2 bytes
Longint	-2147483648...2147483647	4 bytes
Byte	0...255	1 byte
Word	0..65535	2 bytes

The following operations with integer values are allowed:

+	Addition
-	Subtraction
*	Multiplication
/	Division
Div	Integer division
Mod	Calculation of the remainder integer division

When applied to integer values, all the operations listed (except for division) give the whole result. The division operation always yields a real result.

Real types

This group of types denotes sets of real values in different ranges:

Type	Value range	Number of digits of mantissa	Memory size
Real	2.9e-39...1.7e38	11-12	6 bytes
Single	1.5e-45...3.4e38	7-8	4 bytes
Double	5.0e-324...1.7e308	15-16	8 bytes
Extend	3.4e-4932...1.1e4932	19-20	10 bytes
Comp	-2e+63...+2e63-1	19-20	8 bytes

Real values can be represented in a fixed-point and floating-point form. The fixed-point number is represented by a decimal number with a fractional part: 127.3, 17.384, 25.0, 0.54.

The floating-point number has the form $m \cdot 10^p$, where 'm' is the mantissa, and 'p' is the order of the number. As 'm', there can be integers and real numbers with a fixed point, as p - only integers. Both the mantissa and the order can contain the symbols like «+» and «-»: $9e-6$, $0.62e + 4$, $-10.8e12$, $20e-3$.

There are many standard functions for working with real numbers in Pascal. The most commonly used ones are:

Abs (x)	Absolute meaning (module) x
Sqr (x)	square x
Sqrt (x)	Square root of x
Sin (x)	sinus x
Cos (x)	cosinus x
Arctan	Arctangens x
Exp (x)	e ^x
Ln (x)	Natural logorythm x
Trunc (x)	Truncation x
Round (x)	Integer round to x (rounding)

Logical type

Variables of a logical type usually get the values *False* and *True* as a result of performing comparison operations (relations): "<" (less), ">" (more), "<=" (less or equal), ">=" (Greater than or equal to) "<>" (not equal to), "=" (equal to).

The result of the relation operation is true if the relation is satisfied for the values of its operands, and false - otherwise.

There are logical operations applied to variables of a logical type in the language of *Pascal*.

Values		The result of the operation			
x	y	not x	x and y	x or y	x xor y
False	False	True	False	False	False
False	True	True	False	True	True
True	False	False	False	True	True
True	True	False	True	True	False

The following priority of operations is accepted:

1. Not
2. And, *, /, Div, Mod
3. Or, Xor, +, -
4. Relation operations.

Symbolic type

The character type data is described using the Char ID. The value of a character-type variable is a character from the set of ASCII (*American Standard Code for Information Interchange*). This set consists of 256 different characters arranged in a certain way.

Limited and enumerated types

A bounded (interval) data type is an interval of values of an ordinal type, called a base type. Type descriptions specify the smallest and largest values that are included in this interval.

Example: `var a:1..15; b:'a'..'z';`

Variables 'a' and 'b' can take values only from the specified interval; The base type for the variable a is the whole type, and for the variable b, the character type. Enumerations allow the programmer to describe new types of data, the values of which are determined by the programmer himself. The description of the enumerated type consists of a list of its elements enclosed in parentheses.

For example:

```
Var WeekDay: (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
```

Defining variables

Any variable used in a Pascal program must be defined, and the variable definition must precede its first use.

For example:

```
Var a:Integer;  
    Index:0..5;  
    s,p:Char;  
    Sinus:Real;
```

Constants

Pascal allows the introduction of objects that look like variables, but which cannot change their meaning. Such objects are called *constants*. The description of constants begins with the service word `const`, followed by a sequence of definitions of constants.

In the simplest case, a constant is a number, a line, a symbol, or a Boolean expression:

```
Const    {number constants}  
    Length=100;  
    Min=-1;  
    Max=-32678;  
    Numb=7.87e-3;  
    {Boolean constants}  
    Boll=true;  
    Bol2=false;  
    {symbolic constants}  
    Char7='7';  
    {line constants}  
    Str1='Turbo';  
    Str2='Pascal';
```

Exercises

1. Two real numbers are entered from the keyboard. Output the result of their work is in the form of equality. For example, for the entered numbers 5 and 7, output is: $5 * 7 = 35$.
2. The values of two integer variables are entered from the keyboard. Exchange the contents of these variables:
 - a) using an additional variable;
 - b) without an additional variable for the case when the sum of the values is less than 32767.
3. From the keyboard century and year in the century are entered. Output the full year designation. For example, for numbers 21 and 4 the output is 2004.
4. A three-digit number is entered from the keyboard. Output the digits of this number on separate lines.

1.4 Debugging the programme

During the development of the program, errors of three types may occur.

Compile-time errors occur at compile time, i.e. before the start of the program. As a rule, they are caused by incorrect recording of syntactic constructions of the Pascal language.

For example, if you try to start the program

```
Program Error1;  
Var  a,b:Integer;  
Begin  
    a=3;  
    b:=a*7;  
    Writeln(b);  
End.
```

The top line of the screen displays a message

" := " expected (expected " := ").

Putting a colon before the equal sign in the expression $a = 3$; and trying to restart the program, you can find the following message:

Unknown identifier (unknown identifier).

And only after changing the letter 'k' to 'l' in the word Writeln the compiler does not report any errors, and the program is successfully executed.

Both errors are related to compile-time errors. They were discovered by the compiler before the program was executed.

Run-time errors are detected after compilation, when the program is already running. For example, compile and run the following program:

```
program Error2;  
var a, b : integer;  
begin  
    write('a=');  
    readln(a);  
    b:=a*7;
```

```
writeln(b);
end.
```

When the message 'a =' appears, enter the number 2.5. You receive an error message

Invalid numeric format and the IDE will return to edit mode. In this case, the program was written correctly, but during its execution an attempt was made to write a real number to the integer variable.

Logical / algorithmic errors cannot be detected by the computer. Most often they are the result of the application of an initially incorrect algorithm or algorithm, incorrectly written in the programming language. Such errors are noticed by the result of the program execution, which does not coincide with the expectation and displays them with the help of test tests.

1.5 Standard numeric functions

Pascal has a number of standard numeric functions that can be used in algebraic expressions.

The syntax for describing the standard function is:

function name (argument name: argument type): the type of the result of the function.

Function	Activity
Abs (x) : «matches with the type	Returns the absolute value of the argument
Artcan (x:real) :real;	Returns the arctangent of the argument - the value of the angle in radians in the range - $\pi/2.. \pi/2$
Cos (x:real) :real;	Returns the value of argument <i>cos</i>
Exp (x:real) :real;	Returns the value of the function of argument e^x , где $e = 2.718281828$
Frac (x:real) :real;	Returns the fractional part of the argument Frac(3.17)= 0.17 Frac(-3.85)= 0.15 Frac(x)= x - Int(x)
Int (x:real) :real;	Returns the integer part of the argument, gives the largest number not exceeding x, i.e. is an N such that $N \leq x < N+1$ Int(2.4)= 2.0 Int(0.99)= 0.0 Int(-1.2)= - 2.0 Int(-3.9)= - 4.0

<code>Ln(x:real):real;</code>	Returns the value of the function of the natural logarithm of the argument
<code>Odd(x:longint):boolean;</code>	Checks whether the argument is an odd number: <code>Odd(70)= False odd(-21)= True</code>
<code>Pi:real;</code>	Returns the value of π , which is equal to 3.1415926535897932385
<code>Pred(x):</code> «matches with the type of the argument»	Returns the previous value of the argument
<code>Random(n:word):</code> «matches with the type of the argument»	Returns a random number from the range $0 \leq x < n$
<code>Round(x:real):longint;</code>	Converts the value of a real type to an integer type with rounding <code>Round(3.18)= 3</code> <code>Round(3.5)= 4</code> <code>Round(-3.5)= -4</code> <code>Round(-3.2)= -3</code>
<code>Sin(x:real):real;</code>	Returns the value of the <i>sin</i> argument
<code>Sqr(x):</code> «matches with the type of the argument»	Returns the value of the square of the argument
<code>Sqrt(x:real):real;</code>	Returns the value of the square root argument
<code>Succ(x):</code> «matches with the type of the argument»	Returns the next argument value <code>Succ(3)= 4</code> <code>Succ(-12)= - 11</code>
<code>Trunc(x:real):longint;</code>	Converts a value of a real type to an integer type by discarding the fractional part of a number <code>Trunc(2.9)=2</code> <code>Trunc(3.1)=3</code> <code>Trunc (-3.1)= -3</code>

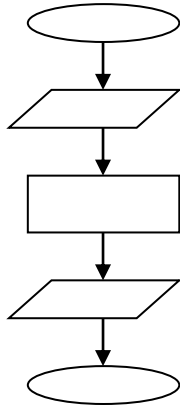
In Pascal there is no operation of erecting a degree. With an integer value of the exponent, it can be replaced by multiplication, in the case of a real exponent, the ratio $xy = \exp(y * \ln(x))$ is used to calculate the power of the number.

2 Fundamentals of programming in the environment of Pascal

2.1 Programming linear algorithms

Basic algorithmic constructions in programming: *follow, fork, cycle*.
An algorithm containing only the construction of a sequence is called *linear*.

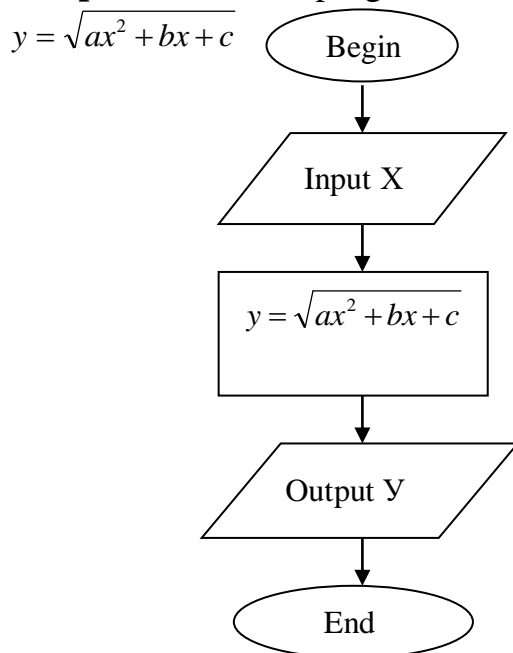
The linear algorithm has the following structure:



Assignment operator in all programming languages is the main operator. The assignment operator is used to calculate the value of a variable and assign it. This operator performs two functions:

1. From the known values of variables, the numerical value of the arithmetic expression to the right of the assignment character is calculated;
2. The computed value is assigned to the name of the variable to the left of the assignment character

Example 1. Create a program for calculating the value of a function:



Solution: As arguments in the task, the values of the variables 'x', 'a', 'b', 'c' are entered, the value of the variable y is output as a result:

```

Var a,b,c,x:integer; y:real;
Begin
  Write('a,b,c=');Readln(a,b,c);
  Writeln('x=');Readln(x);
  y:=Sqrt(a*x*x+b*x+c);
  Writeln('y=', y);
Readln
End.

```

Notes: It is necessary to pay attention to the fact that the value of a numerical value of a real type is given in exponential form. For example, for $a = 1$, $b = 2$, $c = 3$, $x = 1$ the result is given in the form $y = 2.44944897428E + 00$. To output the result of a real type, it is **desirable** to use a formatted output: `Writeln ('y =', y: k: m)`, where the first parameter indicates the number of digits of the number, the second parameter is the number of decimal places of the real number, for example, `Writeln ('y =', y: 8: 2)`.

The `Readln` operator in the program after the result of the output operator arranges the delay on the user's screen before pressing the <Enter> key.

Example 2. Two numbers a and b are given. Give the values of the sum, difference, product, and quotient of these numbers.

Solution: The input parameters are the variables 'a', 'b', the result is the value of the sum, difference, product, and quotient.

Notes: Attention must be paid to the fact that the result of the division operation is always of the *Real* type:

```

Var a,b,s,r,p:Integer;ch:Real;
Begin
  Writeln('enter two numbers:');Readln(a,b);
  s:=a+b; r:=a-b;p:=a*b;ch:=a/b;
  Writeln(a,'+',b,'=',s);
  Writeln(a,'-',b,'=',r);
  Writeln(a,'*',b,'=',p);
  Writeln(a,'/',b,'=',ch:6:2);
Readln
End.

```

Example 3. The seller sold one buyer 25% of the fabric, the second - 30% of the remnant, and the third - 40% of the remaining after the second buyer. How much percent of the fabric is left in the store?

Decision. Let us designate the initial amount of fabric available to the seller-T. After the seller sold 25% of the fabric to the first buyer, the remainder is $T_1 = T - T * 25/100$ ($T - 0.25 * T = 0.75 * T$), after the second and the third buyer we have: $T_2 = T_1 * 30/100$; $T_3 = T_2 * 40/100$;

Note: Note that the value of T in each of the steps in the solution of the problem changes!

```

Var T,P:Real;
Begin

```

```

Writeln(enter the quantity of the
fabric);Readln(T);
P:=T; {remember initial quantity of the fabric}
T:=T-T*25/100; {remnant of the fabric after the first buyer}
T:=T-T*30/100; { remnant of the fabric after the second buyer}
T:=T-T*40/100; { remnant of the fabric after the third buyer}
Writeln('remained =',T/P*100,' % of fabric' );
Readln
End.

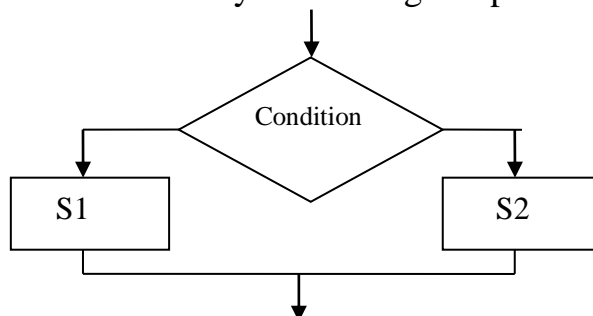
```

Tasks for independent work

1. In one hour the first worker collects N pairs of shoes, the second worker collects L pairs, the third - H pairs. How many pairs of shoes will they collect for T hours?
2. Determine the size of the change after buying goods in the store: gloves - A tenge, a suitcase - B tenge, a tie - C tenge. The initial amount allocated for purchase is D tenge
3. From points A and B two pedestrians with speed V1 and V2 simultaneously came towards each other. The distance between points L0. Make an algorithm for finding the distance between pedestrians through the time interval T.
4. A kid can eat 600 grams of jam for 6 minutes, and Carlson - twice as fast. How long will they have eaten this jam together?
5. To make the necklace, the jeweler took a string of length D and pearls of radius R. Determine how many pearls he can thread on this thread.
6. If the product first went up by 10%, and then went down by 10%, then how much did the price of the product change?
7. If the daughter is 8 years old, and the mother is 38, how many years will the mother be three times as old as her daughter?
8. The dog saw a hare at a distance of 150 m. If the speed of the hare is 250 m, and the speed of the dog is 260 m, how many minutes will the dog catch up with the hare?

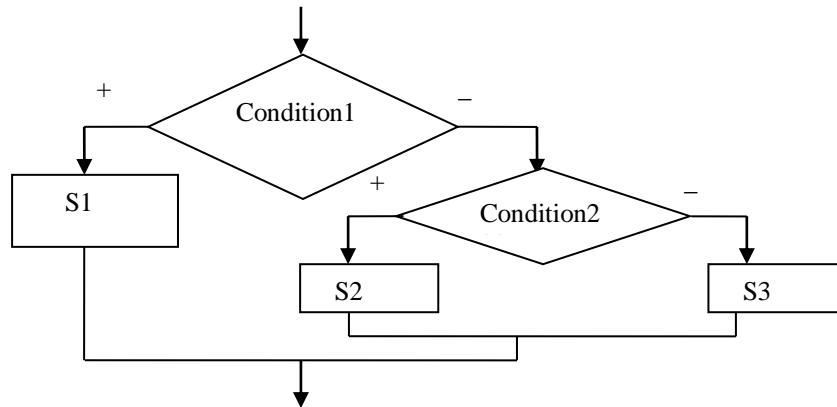
2.2 Programming of branching algorithms

The algorithmic construction of *the fork* is used in algorithms to verify the condition and consists of a logical block and two (or one) function blocks. One of the function blocks may be missing if a partial conditional construction.



An algorithm containing a conditional construction is called a *branching* one.

If both functional blocks are present in the conditional construction, then the construction is called the *complete* conditional, otherwise it is *incomplete*. If in the algorithm there is a need to use additional checking of the condition inside the conditional block, then we are dealing with *nested* conditional constructions



When constructing conditional algorithms, it is necessary to follow the rules of structured programming: there must be one input and one output from the algorithmic construction.

Format of the conditional operator:

```
if condition then
    operator 1
[else operator 2];
```

The execution of the conditional statement begins with the calculation of the values of the logical expression written in the conditional statement. If the condition is true, then the operator 1 is executed, followed by the operator behind the conditional operator. If the result of evaluating the expression is *false*, the statement 2, following the service word *else* is executed, followed by the operator following the conditional word. Thus, as a result of the execution of the conditional statement, one of the operators inside the conditional statement will be selected and executed. If in this case part of the conditional operator, starting with the word *else* is absent, then control is transferred to the operator following the conditional operator. For example, if you find a module, a number (without using standard functions), you can use the complete form of the conditional statement.

```
if x<0 then y:=-x else y:=x;
```

or shortened form

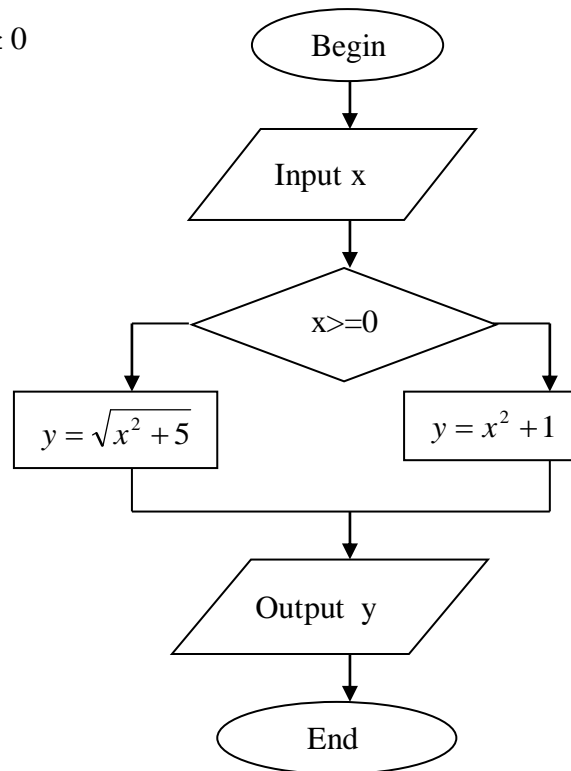
```
if x<0 then x:=-x;
```

The format of nested conditional operators *if*:

```
if expression 1
then
    if expression 2
    then operator 1
    else operator 2;
```

Example 1. Compose an algorithm for calculating the value of an expression

$$y = \begin{cases} x^2 + 1, & x < 0 \\ \sqrt{x+5}, & x \geq 0 \end{cases}$$



The program for calculating the value of Y has the following form:

```

Var x,y:real;
Begin
  Write('x='); Readln(x);
  If x>=0 then
    y:=sqrt(sqr(x)+5)
  Else
    y:=sqr(x)+1;
  Write('y=',y:5:2);
  Readln;
End.
  
```

Example 2. Find the largest of the three given numbers.

Note: This algorithm can be implemented either with the help of a set of 2 conditional statements:

```

if a>b then d:=a else d:=b;
  if d>c then y:=d else y:=c;
  
```

or using a single conditional statement, using nested operators:

```

if a>b then
  if a>c then y:=a else y:=c
else
  if b>c then y:=b else y:=c;
  
```

According to the Syntax of conditional operators

If expression Then operator 1

[Else operator 2];

After keywords like *Then* and *Else* only one operator can be allocated. If, in any of the branches of the alternative (*Then* or *Else*), or both require the execution of several operators, then you should use a compound operator
operator 1 [; operator 2;...]

End;

Which allows interpreting the group of operators as one operator.

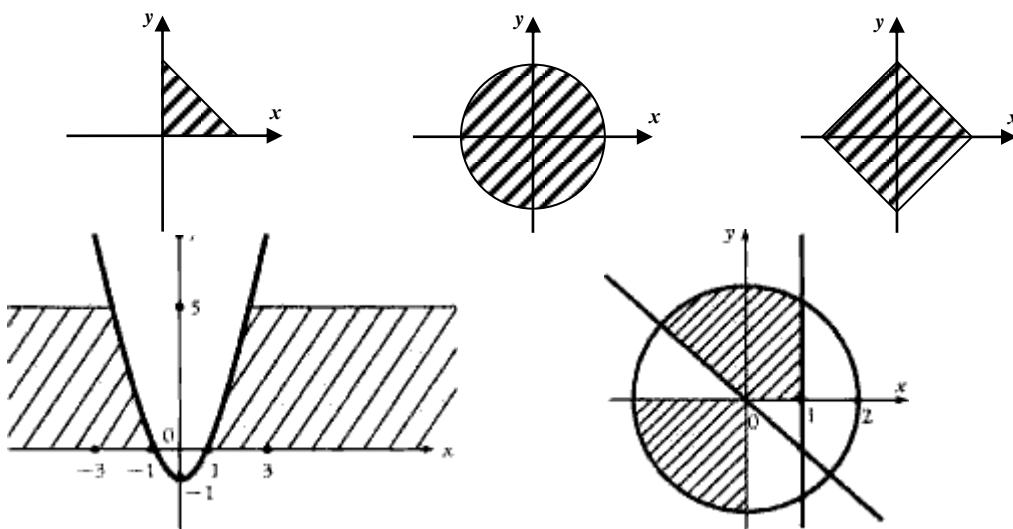
Example 2. Order in growth of two numbers- a and b .

```
If a>b Then
  Begin
    c:=a;
    a:=b;
    b:=c
  End;
```

Using compound operators in most cases makes it possible to clarify the structure of nested operators

Tasks for independent work

1. Real numbers x, y, z are given. You are to find:
 - a) $\max(x, y, z)$;
 - b) $\min(x, y, z)$.
2. Real positive numbers x, y, z are given
 - a) Find out if there is a triangle with sides x, y, z ;
 - b) If a triangle exists, indicate its type (acute, obtuse, rectangular).
3. The triangle is given by the coordinates of its vertices. Determine whether the triangle is equilateral, isosceles, versatile.
4. Create a program that prints the number of days in a month. The month and year are entered from the keyboard. Provide leap year verification.
5. The coordinates of the point $M(x, y)$ are given. Determine whether the point belongs to this plane figure:



Selection operator

The selection operator is a generalization of the conditional operator - it allows one of several actions to be performed, depending on the value of the switch. The switch is an expression, the result of which can only be an ordinal value, the total number of elements of which does not exceed 65535.

The general structure of the selection operator is:

```
Case switch of
    constant 1: operator 1;
    constant 2: operator 2;
    ...
    constant n: operator n;
    [Else operator; ]
End;
```

The selection operator execution begins by calculating the value of the switch. If the result of the calculation is equal to one of the listed constants, then the corresponding operator is executed, then control is passed beyond the selection operator. If the control of the expression does not coincide with any constant, then the operator after the *else* keyword is executed, if it exists, or control is passed to the operator following the *end*.

Example 1. Given the number of the day of the week, give its name.

```
Var N:Byte;
Begin
    Write('N=');Readln(N);
    Case N of
        1: write('Monday');
        2: write('Tuesday');
        3: write('Wednesday');
        4: write('Thursday');
        5: write('Friday');
        6: write('Saturday');
        7: write('Sunday');
    Else Write('No solution'); {or 'There is no such
day' }
```

Example 2. By the number of the entered month, give the number of days in a month.

```
Var n:Byte;
Begin
    Case n of
        1,3,5,7,8,10,12:writeln('31 days' )
        2: writeln ('28/29 days');
        4,6,9,11:writeln('30 days'};
    Else
        Writeln (no month with such a number)
```


End;
End.

Tasks for independent work

1. A natural number n ($n \leq 100$), which determines the age of a person in years, is given. Make a program that displays this number on the screen of the monitor with the name "year", "year" or "years".
2. In the old Japanese calendar was adopted a 60-year cycle, consisting of five 12-year sub-cycles. Subcycles were designated by the name of the color: green, red, yellow, white and black. Within each sub-cycle, years were called animals: rats, cows, tiger, hare, dragon, snake, horse, sheep, monkey, chicken, dog and pig. Write a program that, by the number of the year, determines its name according to the Old Japanese calendar.
3. Make a program that depends on the serial number of the month (1,2, ..., 12), the number of days this month appears on the screen. Consider two cases:
 - a) the year is not a leap year;
 - b) leap year.

A year is a leap year if its number is a multiple of 4, but only multiple of 400 is a multiple of 100 leap years.

2.3 Programming of cyclic algorithms

2.3.1. Cycles with a parameter

With multiple use of the same set of commands, depending on the fulfillment (failure) of the given condition, we are dealing with *cyclic* algorithms.

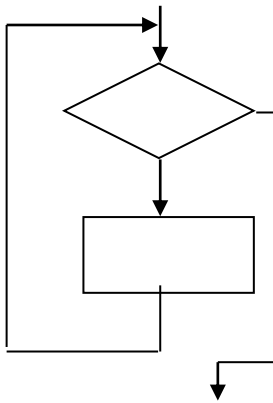
When constructing cyclic algorithms, three types of cyclic constructions are used: a cycle with a precondition, a cycle with a postcondition, a cycle with a parameter.

A cycle with a precondition consists of a conditional block (cycle repeat condition) and a function block containing the commands that make up the body of the loop.

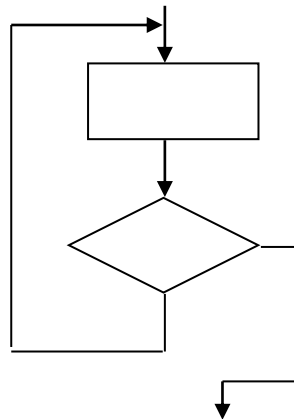
Cycle with postcondition - the condition for the end of the loop (logical block) is located after the function block containing the commands that make up the body of the loop.

A cycle with a known number of repetitions (a repetition command with a parameter) - contains a cycle header where the range of the cycle parameter is specified, as well as the increment of the parameter.

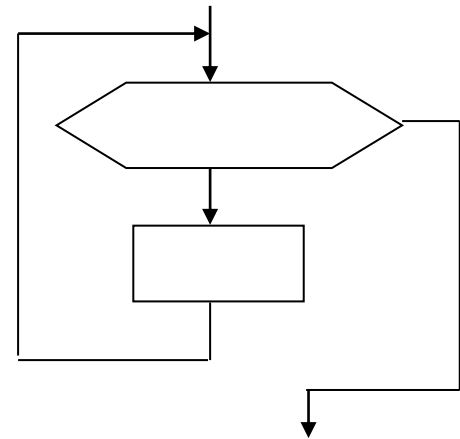
A cycle with a precondition



Cycle with postcondition



A cycle with a known number of repetitions



A cycle operator with a parameter (with a counter) is used to program such cyclic fragments in which the initial and final values of the loop repetition parameter are known before the cycle to be executed:

```
For параметр:=A to B do
    Оператор;
or
For параметр:=A downto B do
    Оператор;
```

where A – initial value of the parameter, B – final value of the parameter.

Let us consider the execution of a cycle operator with a parameter of the form

```
for parameter:=A to B do operator;
```

First, the values of expressions A and B are calculated. The initial value of the loop counter (parameter: = A) is set not in front of the loop header, but directly in the header. If $A < B$, then the parameter is sequential: takes values equal to A, $A + 1$, ..., $B-1$, B and for each of these values the operator representing the cycle body is satisfied. If $A > B$, then the cycle body never fulfill.

In addition, after the end of the body of the cycle, the incrementing of the counter value occurs automatically.

Cycle Operator with Parameter

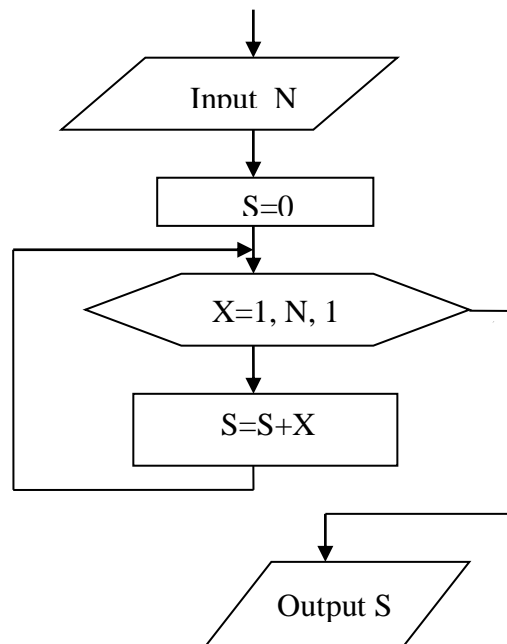
```
for параметр:=A downto B do оператор;
```

is performed in the same way, but the value of the loop parameter changes in steps of -1.

Example 1. Create an algorithm for calculating the sum $S=1+2+3+4+\dots+N$

Solution: This task belongs to the number of exhaustive tasks. One of the solutions to the problem is to use the construction of a loop with a parameter.

Enumeration of the values of the parameter X from the range $[1, N]$ is carried out in the header of the cycle $1 \leq X \leq N$, the body of the cycle consists of one command $S := S + X$; as the initial data, the value of the variable N is entered, the result is the value of S .



```

Var N,X,S:Integer;
Begin
    Write('N=');
    Readln(N);
    S:=0;
    For X:=1 to N do
        S:=S+X;
        Writeln('S=',S);
        Readln
    End.

```

Example 2. Display two-digit numbers whose sum of digits is N ($0 < N \leq 18$)

```

Var k,n,p1,p2,s:Integer;
Begin
    Write ('n=');
    Readln (n);
    For k:=10 to 99 do
        Begin
            {select the highest number of the digit}
            p1:=k div 10;
            {select the lowest number of the digit}
            p2:=k mod 10;
            s:=p1+p2;
            if s=n then write(k, ' ');
        End;
    Readln;

```

End.

Example 3. Find all two-digit numbers that are divisible by n or contain the digit n.

```
Var k,n,p1,p2:Integer;
Begin
    Write('n=');
    Readln(n);
    For k:=10 to 99 do
        Begin
            p1:=k div 10;
            p2:=k mod 10;
            If (p1=n) or (p2=n) or (k mod n=0)
                Then
                    Write(k, ' ');
        End;
    Readln;
End.
```

Tasks for independent work

1. Print the currency exchange table 1,2, ..., of N dollars of the USA into tenge at the current rate (the value of the course is entered from the keyboard).
2. Determine the number of three-digit natural numbers whose sum of digits is equal to the given number N.
3. Write a program for searching for two-digit numbers with the following property: if we add the square of this sum to the sum of the digits of the number, then the given number is obtained again.
4. The squares of some three-digit numbers end in three digits, which constitute the original numbers. Write a program for finding such numbers ($6252 = 390625$).
5. On account in the bank N tenge was given at M percent per annum. How much will in the account in X years be stored?
6. A unicellular amoeba is divided into 2 cells every 3 hours. Determine how many cells will it have in N hours.

2.3.2 Cycles with conditions

A cycle with a precondition is used when the number of repetitions of the loop statement is unknown in advance, and a certain condition is set for the continuation of the cycle: While the condition do is the operator;

Example 1. Find the sum of the digits of a given natural number n.

```
var m,n,s:longint;
```

```

begin
    write('n=');
    readln(n);
    m:=n;
    s:=0;
    while m<>0 do
        begin
            s:=s+m mod 10;
            m:=m div 10;
        end;
    writeln('s=', s);
    readln
end.

```

Example 2. Find the maximum number of a natural number.

```

var max,n,k,p:longint;
begin
    write('n=');
    readln(n);
    max:=n mod 10;
    p:=n;
    while p<>0 do
        begin
            k:=p mod 10;
            if k>max then max:=k;
            p:=p div 10
        end;
    writeln('max ' , n, ' = ',max);
    readln
end.

```

Example 3. Rearrange the digits of a given natural number in the reverse order.

```

var n,k,m:longint;
begin
    write('n=');
    readln(n);
    k:=n;
    m:=0;
    while k>0 do
        begin
            m:=m*10+k mod 10;
            k:=k div 10
        end;
    writeln(n, '-',m)
end.

```

For the software implementation of cyclic algorithms with an unknown number of repetitions, there is one more loop operator - a cycle with a postcondition, which has the following form:

```

Repeat {repeat}
Operator 1;
Operator 2;
Operator 3;
. . .
Until <condition>;

```

This operator differs from the loop with the precondition that the condition check is performed after the next execution of the loop body. This ensures that the loop is executed at least once.

The execution of the statements that make up the body of the loop continues as long as the condition is False and ends when the condition becomes True.

Example 4. Write a program for finding the greatest common divisor of two nonnegative numbers.

```

var x,y,nod:integer;
begin
  write('x,y=');readln(x,y);
  repeat
    if x>y then x:=x mod y else y:=y mod x;
  until (x=0) or (y=0);
  nod:=x+y;
  writeln('NOD = ', nod);
  readln
end.

```

The second version of the algorithm for finding the greatest common divisor of two numbers:

```

while x<>y do
  if x>y then
    x:=x-y
  else y:=y-x;

```

Often when creating programs, it becomes necessary to use nested loops.

Example 5. Find all prime numbers in the interval from A to B. A natural number is said to be simple if it is divisible without residue by only one and self.

```

Var a,b,x,y,k:Integer;
Begin
  Readln(a,b);
  For x:=a to b do
    Begin
      k:=0;
      For y:=1 to x do
        If x mod y=0 then k:=k+1;
      If k=2 then write(x,' ');
    End;
  Readln
End.

```

Tasks for independent work

1. Natural numbers are given M and N.

a) Find NOD (M, N);

b) Find NOK (M, N);

c) reduce the fraction .

2. Create a program for graphical representation of the divisibility of numbers from 1 to N (the value of N is entered from the keyboard). In each line, you need to print the next number and as many "+" characters as there are dividers for this number. For example, if $N = 4$, then the screen should be printed:

1+

2++

3++

4+++

3 A natural number is called perfect if it is equal to the sum of all its regular divisors (except for itself). The number 6 is perfect, since $6 = 1 + 2 + 3$. A natural number N is given. All perfect numbers less than N must be obtained.

4 A natural number N is given. All the numbers of the palindrome that do not exceed a given N are presented on the screen. A natural number is called a palindrome if the record is read equally from the beginning and from the end (for example, 123321, 565, 11, 4).

5 Determine whether a given number is automorphic. An automorphic number is a number whose square ends with itself. For example, the number 6 is automorphic, since its square 36 ends at 6, or the number 25 - its square 625.

2.4 Procedural-modular technology for solving problems

2.4.1 Procedures

When creating programs, there are often situations when some actions need to be performed several times. If these actions follow each other, i.e. they are simply repeated, then they can be written using any of the cycles (the repetition command). But, sometimes, repetitive parts are not located one after another, but scattered throughout the text of the program. In such cases, subroutines are used - parts of the program in which one often writes frequently occurring actions, for further calling them from the main program.

There are two types of subroutines in Pascal: procedures and functions. The main difference is how they are called from the main program. Creating a new procedure, the programmer can say, develops a new command language, which will be used in the future as one of the already existing commands, such as Read () or Write (). The function will be used in the same way as one of the standard functions, for example, Abs () or Sqr ().

Procedures should be described before they are used. The declaration of the procedure is made in the description section of the main program.

The structure of the procedure is similar to the structure of the program and contains the same sections, but instead of the word Program, the word Procedure is specified, followed by parameters (while we are considering procedures without parameters). The rest of the procedure is practically the same as the program. In the description section, there may be descriptions of variables, descriptions of constants, and even declarations of their procedures. After the word End, the procedure is terminated with a semicolon:

```
var
n,m:integer;
i:integer;
fn,fm:integer; {fn=n! fm=m!}
p,nn:integer;
c:real;
procedure factor;
begin
    p:=1;
    for i:=1 to nn do
        p:=p*i;
    end;
```

A global variable is a variable declared in an external block for this procedure.

A local variable is a variable declared in the inner block.

- 1) all declarations of local variables are invisible for the outer block, these variables cannot be used in an external block;
- 2) all declarations of global variables whose names do not coincide with the names of local variables are visible in all internal blocks;
- 3) if the names of global and local variables are the same, then the local variable is used in the inner block.

This is due to the fact that each time the procedure is called, local variables are created anew, and when you return from it, they are destroyed.

The parameters specified in the description of the procedure are called formal.

The parameters supplied during the procedure call are called factual.

For example, a procedure declared with a header

```
procedure add(x: integer; r: real);
```

has two formal parameters: the whole - X and the real one - Y. Inside the procedure, the call to its parameters occurs precisely by these names. When you call a procedure, you can substitute any variables or even expressions as actual parameters. The main thing is to match the type and number of parameters. For example, the procedure can be called as follows: add (i, 3.5);

This greatly simplifies the work with the parameters. However, one more aspect should be considered. In Pascal, there are two ways of transferring parameters:

When passing parameters by **value** (sometimes they say, when using **value parameters**) during the procedure call, local variables are created with the

names of formal parameters. They are assigned the values of the actual parameters. These local variables, as usual, are destroyed during the return from the procedure.

When passing parameters by **reference** (when using **variable parameters**) new variables are not created. Just inside the procedures, the actual parameters are used with new names corresponding to formal parameters. In other words, the variable from the calling block gets a "nickname", by which it can be accessed from the procedure. To declare a parameter passed by reference, the service word `var` must be placed before it.

Example 1. The procedure finds the sum of the digits of a natural number. The input file contains a set of N integers. Output in the output file the sum of the digits of the elements of the sequence.

```
procedure SUM_CIFR(m:integer; var s:integer);
var ml:integer;
begin
    s:=0;
    repeat
        ml:=m mod 10;
        s:=s+ml;
        m:=m div 10;
    until m=0;
var n,l,j,a,s:integer;
    fin,fout:text;
begin
    assign(fin,'1.in');reset(fin);
    assign(fout,'1.out');rewrite(fout);
    readln(fin,n);
    for i:=1 to n do
        begin
            read(fin,a);
            SUM_CIFR(a,s);
            write(fout,s,' ');
        end;
    close(fin);
    close(fout);
end.
```

Example 2. The procedure checks whether the given number is a palindrome. The input file has a sequence of integers. Output the elements of the array that are palindromes in the output file.

```
procedure Palindrom_li(m:integer; var t:Boolean);
var s,ss:string;
    i:integer;
begin
    Str(m,s);
```

```

    ss:='';
    For i:=1 to length(s) do
        ss:=s[i]+ss;
    If s=ss then t:=True
        Else t:=False;
end;
Var n,l,a:integer;
    f:boolean;
    fin,fout:text;
Begin
    assign(fin,'A.in');reset(fin);
    assign(fout,'A.out');rewrite(fout);
    readln(fin,n);
    for i:=1 to n do
    begin
        read(fin,a);
        Palindrom_li(a,f);
        if f=true then write(fout,a,' ');
    end;
    close(fin);
    close(fout);
end.

```

Tasks for independent work

1. Using the procedure of finding the number of natural divisors of a number, determine which of the three numbers of natural divisors is greater.
2. Using the procedure of finding the sum of the digits of a natural number, determine the sum of the digits of which of the three numbers is greater.
3. Using the procedure of finding the greatest common divisor of two natural numbers, find the GCD of three numbers.
4. Using the GCD procedure, reduce the simple fraction. The numerator and denominator of the fraction are entered from the keyboard.
5. Write a procedure for computing the natural degree of an integer.

2.4.2 User functions

The description of the function is similar to the description of the procedure. The word procedure is used instead of the word procedure, followed by a list of parameters in parentheses. After the function name or after the parameter list through the colon, you must specify the type of the value returned by the function.

Example 1.

```
function pow(a,b: integer):integer;
```

```

var f:integer;
    i:integer;
begin
    f:=a;
    for i:=1 to b do
        f:=f*a;
    pow:=f;
end;

```

The function calculates the value of the degree b of the number a specified as parameters of the function.

This function has integers as parameters, and its result will also be an integer value.

To call a function in a program, you need to write its name with parameters, if any, on the right side of the assignment statement or in the expression.

```

s:=pow(2,n);
k:=(s+pow(5,3)) div pow(abs(x-1),8);
writeln('5^4=',pow(5,4));
...

```

Sometimes a user function is used to simplify the writing of an expression. For example, the function of calculating the degree could be described as follows:

```

function pow (a,b: integer): real;
begin
    pow:=exp(b*ln(a));
end;

```

Example 2. To calculate the expression $s = \frac{4f^2(a-1)+3}{f(a-8)+2}$

Where the function f is defined in the following way: $f(x) = \sqrt{5+|x^2-2|}$

```

var s,a:real;
function f(x:real):real;
begin
    f:=sqrt(5+abs(sqr(x)-2));
end;
begin
    write('a=',a);
    readln(a);
    s:=(4*sqr(f(a-1))+3)/(f(a-8)+2);
    writeln('s= ',s);
    readln
end.

```

Tasks for independent work

1. The function calculates the number of natural divisors of a given number. Using this function, determine which of the three numbers of natural divisors is greater.
2. Write a function that checks whether the number is simple or not. Result of the function: TRUE or FALSE.
3. The function finds the number of primes in a given interval. Determine in which of the two intervals there are more prime numbers.
4. The function determines whether the rectangle is a square. The coordinates of the N rectangles are located. Determine how many of these squares the sides of the rectangle are parallel to the coordinate axes.

2.4.3 Recursion

A procedure or function can call another procedure or function if it is in the same block and has been defined in advance. For example, if the function of computing the GCD of two natural numbers with the name GCD was previously determined, then the function of computing the NOD of two natural numbers can be defined as follows:

```
function NOK(a,b:integer):integer;
begin
    NOK:=a*b div NOD(a,b);
end;
```

A subroutine that calls itself is called recursive. The situation when a procedure or function calls itself is called recursion.

The typical construction of a recursive procedure is:

```
procedure prec(t:integer);
begin
    <actions when entering recursion>;
    if <checking the condition> then rec(t+1);
    <actions on the escape from the recursion>;
end;
```

As an example of a recursive subroutine, let us consider the function of calculating the factorial of a number. The recursive definition of the factorial looks like this:

$$n! = \begin{cases} 1, & n = 1 \\ (n-1)! \cdot n, & n > 1 \end{cases}$$

The corresponding recursive function will look like this:

```
function factor(n:integer):integer;
begin
    if n=1 then
        factor:=1
    else
```

```

        factor:=factor (n-1) *n;
end;

```

The process of calculating the value of a factorial of a number by means of a recursive function can be represented in the form of a table:

		launch		return
exemplar	n	$Factor(n)$	n	$Factor(n)$
1	4	-	4	24
2	3	-	3	6
3	2	-	2	2
4	1	1	1	1

If a procedure is used as a recursive subroutine, it is convenient to return the result of calculations as a var-parameter. Recursive procedure for calculating the factorial of a number:

```

procedure fact(n:integer;var f:integer);
begin
    if n=1 then f:=1
    else
        begin
            fact(n-1,f);
            f:=f*n;
        end;
end;

```

Tasks for independent work

1. Write a recursive function for computing the degree n of a real number a (n is a natural number).
2. Write a recursive function for calculating the k th term of the Fibonacci sequence. The Fibonacci sequence is constructed as follows: $f_0 = f_1 = 1$; $f_n = f_{n-1} + f_{n-2}$ (for $n = 2, 3, \dots$).
3. Write a recursive function that determines whether a given natural number is prime.
4. Write a recursive procedure to display the digits of a natural number in reverse order.
5. Write a recursive procedure for translating a natural number from the decimal number system to an N -like one. The value of N in the main program is entered from the keyboard ($2 \leq N \leq 16$).
6. Write the recursive function of finding NOD (a, b).

3 Types and structures of data

3.1 File-based types of data

The Pascal language allows you to work with files, as with the basic units of the disk operating system. A file is a named (named) area on a magnetic or laser disk.

Files differ in content type and in the way they are accessed:

By type of elements (structure):

- text;
- typed;
- binary.

By ways of access to files:

- sequential access files;
- direct access files.

A text file is a sequential file. This means that when working with it, you can either only read information from it symbol by character (or line by line), or write;

You cannot go back to the data you just read or write (you can close the file and start working with it again).

1. Before you start, you need to associate a variable of type Text with the name of a particular disk file. The standard procedure is for this:

Assign (<file variable>, <file name>);

2. Opening the file for reading is done by the procedure:

Reset (<file variable>);

Opening for writing: Rewrite (<file variable>);

3. In addition, in Turbo Pascal, you can open a file to add. The information will be appended to the existing contents of the file:

Append (<file variable>);

4. After opening the file for reading, you can read the information using the standard read procedure:

Read (<file variable>, <variable>);

If you need to read the data and go to the next line of the file, then the procedure is applied Readln(<filevariable>, <variable>);

5. To write to a file, use the procedure Write (<file variable>, <variable>);

To record with a new line: Writeln (<file variable>, <variable>);

6. When reading from a file, the question arises, how to determine when the file ends (when trying to read after reaching the end of the file, an execution time error occurs). To solve the problem in Pascal there are two logical functions:

Eoln (<file variable>) - checking the end of the line;

Eof (<file variable>) - check for reaching the end of the file.

7. After working with the file, you must close it with the Close procedure `Close(<file variable>)`.

Example 1. A date is given in the format DD.MM.GG in the input file. In the output file print the date in the format: day month year (April 20, 2018 - April 20, 2018).

```
Var S:string;
    Den,mes,god:byte;
    F1,f2:text;
Begin
    Assign(f1,'data.in');reset(f1);
    Assign(f2,'data.out');rewrite(f2);
    Read(f1,s);
    Den:=copy(s,1,2);
    Mes:=copy(s,4,2);
    God:=copy(s,7,4);
    Write(f2,den,' ');
    Case mes of
        1: write(f2,'января ');
        2: write(f2,'февраля ');
        3: write(f2,'марта ');
        4: write(f2,'апреля ');
        5: write(f2,'мая ');
        6: write(f2,'июня ');
        7: write(f2,'июля ');
        8: write(f2,'августа ');
        9: write(f2,'сентября ');
        10:write(f2,'октября ');
        11:write(f2,'ноября ');
        12:write(f2,'декабря ');
    End;
    Write(f2,god,' года');
    Close(f1);
    Close(f2);
End.
```

Example 2. Three numbers are given in the input file. Output the maximum and minimum values of these numbers in the output file.

```
Var a,b,c,max,min:integer;
    F1,f2:text;
Begin
    Assign(f1,'1.in');Reset(f1);
    Assign(f2,'1.out');Rewrite(f2);
    Read(f1,a,b,c);
    If a>b then
        begin
            max:=a;
```

```

        min:=b
    end
else
    begin
        max:=b;
        min:=a
    end;
If c>max then max:=c;
If c<min then min:=c;
Write(f2,'max=',max,' min=',min);
Close(f1);
Close(f2);
End.

```

Tasks for independent work

1. The components of the file are real numbers. Find:
 - a) the sum of the components;
 - b) the product of the components;
 - c) the sum of the squares of the components;
 - d) the modulus of the sum and the square of the product of the components;
 - e) the last component of the file.
2. Components of file f are real numbers. To find:
 - a) the largest of the values of the components;
 - b) the sum of the largest and smallest values of the components;
 - c) the difference between the first and last component.
3. Given a file f whose components are natural numbers. Get all the components of file f in the file g:
 - a) being even numbers;
 - b) dividing by 3 and not divisible by 7;
 - c) are complete squares.
4. Given a file f whose components are integers. Write all the even numbers of the file f into the file g, and all odd numbers in the file h. The sequence of numbers is preserved.
5. Given a character file f. Write the components of file f in file g in reverse order.

3.2 Linear arrays

An array is a data structure that is a homogeneous, fixed by size and configuration set of elements of a simple or composite structure ordered by numbers.

An array is defined by the name and the number of dimensions (coordinates) required to indicate the location of the desired array element.

Arrays can be classified on two different grounds:

- by the number of dimensions, arrays are divided into one-dimensional (vectors), two-dimensional (matrices), and multidimensional ones;
- by type of elements, arrays are divided into arrays of homogeneous data structures, arrays of heterogeneous data structures and file arrays.

One-dimensional array is a fixed number of elements of the same type, united by a common name, each element having its own number, and the element numbers are consecutive.

For example, enter 10 integers from 21 to 30 and combine them with the common name A.

1	2	3	...	10
21	22	23	...	30

A is the common name for all elements of the array. Before processing the array, it must be described in the type declaration section or the variable declaration section:

```
Const n = 10;
```

```
Type massiv = array[1..n] of Integer;
```

```
Var a: massiv;
```

array – functional word («massive», «array»);

[1..n] – the number of the first element is indicated in square brackets, then the number of the last element of the array is indicated;

Integer – the type of all elements of the massive.

Or: Var a:array[1..n] of Integer;

Consequently, one-dimensional array is described as follows:

```
array [n1..n2] of <type of the element>;
```

where n1 is the number of the first element, n2 is the number of the last element.

When solving tasks for processing arrays, the first step is the stage of initializing the array, setting initial values to the elements of the array. Initialization of an array consists in assigning to each element of an array the same value corresponding to the base type.

This can be done by writing a group of assignment operators. For example, a[1]:=0; a[2]:=0; a[3]:=0; a[4]:=0; .

However, with a large number of elements, this initialization method is not rational. It is much more convenient to get the same result using the operator for: for i:=1 to 4 do a[i]:= 0;

Values of array elements can also be assigned using the operator read or readln with the use of the operator of the cycle organisation for:

```
for i:=1 to 4 do readln(a[i]);
```

Let us consider several ways to fill arrays:

1. Enter the elements of the array from the keyboard:

```
write (' Enter the elements of the array:');
```

```
readln (n);
```

```
writeln (' Enter the elements of the array ');
```

```
for i:=1 to n do read (a[i]);
```

2. Filling an array with a random number generator:

```
for i:=1 to n do a[i]:= random(100);
```

3. Reading the values of elements from a file. You can create a text file in advance, and then read the elements into an array from it. The output of an array is performed element by element in a loop:

```
for i:=1 to n do write(a[i], ' ');
```

All tasks related to processing arrays can be conditionally divided into several types:

1. Finding the sum, product, number of elements that have given properties.

2. Check the elements of the array to perform the specified property.

3. Change the values of array elements that have the specified property.

4. Search for items (numbers) that have the specified property.

5. Work with multiple arrays.

6. Shift of array elements.

7. Permutation of array elements.

Example 1. Find the sum of array of the elements that are multiples of a given number.

Decision: It is necessary to look at all the elements of the array and, if the item being watched is divided by the whole number (divisible by a given number), add it to the sum.

```
var a: array[1..100] of integer;
    n,i,s,k:integer;
begin
    write('n=');readln(n);
    for i:=1 to n do read(a[i]);
    write('k= ');
    readln (k);
    s:=0;
    for i:=1 to n do
        if a[i] mod k = 0 then s:=s+a[i];
    writeln('S=',s);
end.
```

Example 2. Check whether the array of the elements form a monotonically decreasing sequence of numbers.

Decision: Elements of the array form a monotonically decreasing sequence of numbers if for each pair of neighboring numbers $a[i]$ and $a[i + 1]$ the condition $a[i] > a[i + 1]$.

```
var a: array[1..100] of integer;
    n,i,s:integer;
    fl:boolean;
begin
    write('n=');readln(n);
    for i:=1 to n do read(a[i]);
```

```

    fl:=true;
    for i:=1 to n-1 do
        if a[i]<=a[i+1] then fl:=false;
    if fl=true then
        write('yes')
        else write('no');
end.

```

Example 3. Given an array. All positive elements of the array are stored in one array, negative - in the other.

Decision: We scan all elements of the array. If the element of the array in question is positive, then we store it in array B, increasing the index of the next element of array B, if negative - in array C, increasing the index of the next element of array C.

```

var a,b,c: array[1..100] of integer;
    n,i,s,ind1,ind2:integer;
begin
    write('n=');readln(n);
    for i:=1 to n do read(a[i]);
    ind1:=0;
    ind2:=0;
    for i:=1 to n do
        if a[i]>0 then
            begin
                ind1:=ind1+1;
                b[ind1]:=a[i];
            end
        else
            begin
                ind2:=ind2+1;
                c[ind2]:=a[i];
            end;
    if ind1>0 then
        begin
            for i:=1 to ind1 do
                write(b[i],' ');
            writeln
        end;
    if ind2>0 then
        begin
            for i:=1 to ind2 do
                write(c[i],' ');
            writeln
        end;
end.

```

Tasks for independent work

1. Replace all even elements of the array with their squares, and odd doubles.
2. Given an array. Obtain two arrays from it: write all the elements with even numbers to one, in the other - elements with odd numbers of the original array.
3. An array of integers consisting of N elements is given. Fill it with the keyboard. You are to find:
 - a) the sum of elements that have an odd value;
 - b) derive indices of those elements whose values are greater than a given number.
4. Given an array of integers consisting of N elements. Fill it with the keyboard, Find the sum of the elements that have odd indexes.

3.2.1 Search in an array

Often, when processing arrays, you have to solve problems related to searching for array elements that satisfy specified conditions, or their numbers.

Example 1. Find the numbers of even elements of the array.

Decision: It is necessary to look through the elements of the array, and if the item being watched is even, output its number.

```
writeln ('numbers of the even elements of the array:')
for i:=1 to n do
  if a[i] mod 2 =0 then write(i:4);
```

Example 2. Find the maximum and minimum elements of an array.

Decision:

```
var a:array[1..100]of integer;
    n,I,max,min:integer;
begin
  write('n=');readln(n);
  for i:=1 to n do read(a[i]);
  max:=a[1];
  for i:=2 to n do
    if a[i]>max then max:=a[i];
  min:=a[1];
  for i:=2 to n do
    if a[i]<min then min:=a[i];
  writeln('max=',max,' min=',min);
end.
```

Note: if you want to find the ordinal number of the maximum (minimum) element, then you can apply the algorithm:

```
max:=a[1];ind:=1;
for i:=1 to n do
  if a[i]>max then
    begin
      max:=a[i];
      ind:=i
    end;
```

```

or:
ind:=1;
for i:=1 to n do
  if a[i]>=a[ind] then
    ind:=i;

```

Tasks for independent work

1. For N points on the plane, find the three points that form the triangle with the largest area.
2. Display the elements of the array that are:
 - a) prime numbers;
 - b) the perfect numbers;
3. Write in the array the first N Fibonacci numbers:
 $a_1=1; a_2=1; \dots a_i=a_{i-2}+a_{i-1}, i>2.$
4. Among the N points on the plane, find a pair of the most distant from each other.

3.2.2 Shifts, reversal, and sorting of array elements

One of the types of array processing is a shift, in which the value of each shifted element is transferred to the previous or next elements of the array. In this connection, the cyclic shifts of the elements to the left and to the right are distinguished.

Consider the shift of array elements in the following example. It is necessary to remove the maximum element from the array (all elements of the array are different). After deleting the maximum element, the array "compact", shifting all the elements following it to the left. In order to solve this problem, it is necessary:

- 1) find the number of the maximal element k;
- 2) move all the elements of the array, starting with k + 1-go, one step to the left;
- 3) assign the last element a value of zero;
- 4) reduce the dimension of the array by 1.

```

var a:array[1..100]of integer;
    n, I, k, max:integer;
begin
  readln(n);
  for i:=1 to n do read(a[i]);
  k:=1;
  for i:=1 to n do
    if a[i]>a[k] then k:=i;
  for i:=k+1 to n do
    a[i-1]:=a[i];
  a[n]:=0;
  for i:=1 to n do write(a[i], ' ');
end.

```

In the considered example, all elements of the array following the maximum moved left one position.

After the cycle is completed, all elements starting from $k + 1$ th will be shifted to the left by 1 step. The maximum element was "pushed" out of the array, the value of the penultimate $a[n-1]$ and the last $a[n]$ array elements were the same. The last element is zeroed, and the number of elements after removing one element is reduced by one. We have considered the so-called ordinary element shift.

For example, if you apply this program to an array A:

<i>A</i>	1	2	3	4	5	6	7	8
	2	7	3	45	17	21	37	9

Then as a result of the program execution, we get the following array:

<i>A</i>	1	2	3	4	5	6	7
	2	7	3	17	21	37	9

Consider a circular shift of the elements of the array to the right by one position.

```
var m:array[1..100] of integer;
    i,k:integer;
begin
    k:=m[n];
    for i:=n downto 2 do
        m[i]:=m[i-1]);
    m[1]:=k;
end;
```

Consider permutations of array elements in the reverse order. If the original array looked like this:

<i>A</i>	1	2	3	4	5	6	7	8
	2	7	3	45	17	21	37	9

then after the treatment it will look like:

<i>A</i>	1	2	3	4	5	6	7	8
	9	37	21	17	45	3	7	2

To reverse an array, proceed as follows. We swap the first element of the array with the last, the second element with the penultimate, and so on.

We will have $A[1] \leftrightarrow A[8]$ on the first step.

The table after the permutation of this pair of elements takes the form:

<i>A</i>	1	2	3	4	5	6	7	8
	9	7	3	45	17	21	37	2

On the second step: $A[2] \leftrightarrow A[7]$:

<i>A</i>	1	2	3	4	5	6	7	8
	9	37	3	45	17	21	7	2

On the third step: $A[3] \leftrightarrow A[6]$:

<i>A</i>	1	2	3	4	5	6	7	8
	9	37	21	45	17	3	7	2

On the fourth step: $A[4] \leftrightarrow A[5]$:

<i>A</i>	1	2	3	4	5	6	7	8
	9	37	21	17	45	3	7	2

It means that after four permutations of pairs of elements standing at the same distance from the center of the array, we obtained an array whose elements are rearranged in the reverse order in comparison with the original one. In general, the algorithm for handling the array will look like this:

```
var m:array[1..100] of integer;
    i,c:integer;
begin
    readln(n);
    for i:=1 to n do read(m[i]);
    for i:=1 to trunc(n/2) do
        begin
            c:=m[i];
            m[i]:=m[n-i+1];
            m[n-i+1]:=c;
        end;
```

Sorting is the process of rearranging the elements in a certain order. There are many methods for sorting arrays. But most of them can be attributed to three main varieties:

- 1) sorting inclusions;
- 2) sorting by choice;
- 3) sorting by exchange.

Let us consider classical variants of these sortings. We will assume that it is required to arrange the array in ascending order of elements.

Sorting by inclusions

Elements of the array will be conditionally divided into an already prepared sequence $a[1] \dots a[i-1]$ and the remaining input sequence. At each step, starting

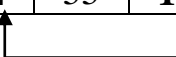
with the second element, we will insert the current element of the array into a suitable place in the already sorted part of the array.

Consider, for example, the initial array:

<i>A</i>	1	2	3	4	5	6	7
	44	55	12	42	94	18	6

In the first step (for $i = 2$), the finished sequence represents only one number 44. The element $a[i] = 55$ should remain in its place, because $55 > 44$. For $i = 3$, the number 12 must be inserted in the sequence {44 55} in the first place.

<i>A</i>	1	2	3	4	5	6	7
	44	55	12	42	94	18	6




That is why the array has the following view:

<i>A</i>	1	2	3	4	5	6	7
	12	44	55	42	94	18	6

In the next step, the number 42 is compared from right to left with each of the previous elements until we find a suitable place for insertion:

<i>A</i>	1	2	3	4	5	6	7
	12	44	55	42	94	18	6

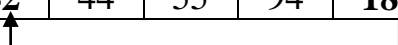


$i=5$:

<i>A</i>	1	2	3	4	5	6	7
	12	42	44	55	94	18	6

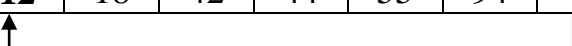
$i=6$:

<i>A</i>	1	2	3	4	5	6	7
	12	42	44	55	94	18	6



$i=7$:

<i>A</i>	1	2	3	4	5	6	7
	12	18	42	44	55	94	6



Sorting finishes when $i > n$:

<i>A</i>	1	2	3	4	5	6	7
	6	12	18	42	44	55	94

Instead of looking for a suitable place for inserting the current element in one cycle, and performing a shift in another cycle, we combine these operations. This procedure is obtained:

```

procedure SortInsert;
Var i,j:integer;
    c:integer;
begin
    for i:=2 to n do
        begin
            x:=a[i];
            j:=i-1;
            while x<a[j] do
                begin
                    a[j+1]:=a[j];
                    j:=j-1;
                end;
            a[j+1]:=x;
        end;
    end;
end;

```

Sorting by choice

Let us find the minimal element of the array and change it by places with the first one. Among the remaining elements, you must find the minimum and swap it with the second and so on until there is a maximum element located at the end. For example, the same array is given:


<i>A</i>	1	2	3	4	5	6	7
	44	55	12	42	94	18	6

After finding the minimum element ($i = 1$) and exchanging it with the first one, we get:

<i>A</i>	1	2	3	4	5	6	7
	6	55	12	42	94	18	44

$i=2$:

<i>A</i>	1	2	3	4	5	6	7
	6	55	12	42	94	18	44



The minimum among the remaining elements (12) is exchanged with the second. The variable i indicates the number of the element where the next minimum value will be placed.

$i=3$:

<i>A</i>	1	2	3	4	5	6	7
	6	12	55	42	94	18	44

i=4:

<i>A</i>	1	2	3	4	5	6	7
	6	12	18	42	94	55	44

i=5:

<i>A</i>	1	2	3	4	5	6	7
	6	12	18	42	94	55	44

i=6:

<i>A</i>	1	2	3	4	5	6	7
	6	12	18	42	44	55	94

The procedure of sorting by choice has the following view:

```

procedure SortSelect;
var i,j,k:integer;
    x:integer;
begin
  for i:=1 to n-1 do
    begin
      k:=i;
      x:=a[i];
      for j:=i+1 to n do
        if a[j]<x then
          begin
            k:=j;
            x:=a[j];
          end;
      a[k]:=a[i];
      a[i]:=x;
    end;
  end;
end;

```

In the proposed procedure, the variable *k* indicates the position of the minimum among the elements considered, and the variable *x* is its value. The inner loop is used to find the minimum value. The exchange of elements occurs after the end of the inner cycle.

Sorting by exchange

The algorithm for sorting by exchange is based on comparing and exchanging a pair of neighboring elements. If two neighboring elements are not arranged in ascending order among themselves, then they exchange places.

At the first viewing, the minimum element is in its place, so the next time you can consider only the remainder of the array.

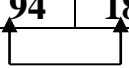
The procedure is as follows:

```
procedure SortChange1;  
var i,j:integer;  
    x:integer,  
begin  
    for i:=2 to n do  
        begin  
            for j:=n downto i do  
                if a[j-1]>a[j] then  
                    begin  
                        x:=a[j-1];  
                        a[j-1]:=a[j]  
                        a[j]:=x;  
                    end;  
            end;  
        end;  
end;
```

Consider the process of executing an inner loop when i=2.

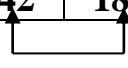
j=6:

1	2	3	4	5	6
44	55	12	42	94	18



j=5:

1	2	3	4	5	6
44	55	12	42	18	94

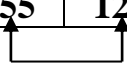


j=4:

1	2	3	4	5	6
44	55	12	18	42	94

j=3:

1	2	3	4	5	6
44	55	12	18	42	94



j=2:

1	2	3	4	5	6
44	12	55	18	42	94

Thus, after all the iterations of the inner loop, the array has the form:

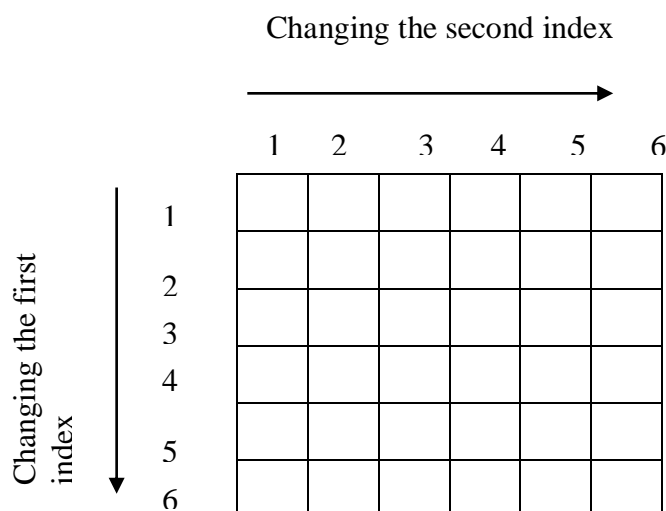
1	2	3	4	5	6
12	44	55	18	42	94

Tasks for independent work:

1. In the table $x[1..n]$, each element is equal to 0, 1 or 2. To rearrange the elements of the array so that all zeros are located first, then all the ones, and finally all the deuces.
2. An array of pairwise distinct numbers is specified. Create an algorithm for rearranging the elements of the array so that the minimum element is the first, the next in ascending order is the last one, etc., and the maximum is the average one.
3. A linear array of N elements is given. Delete those elements of the array that violate the order of elements in ascending order.
4. You are to order all the elements of the linear array preceding the maximal, in ascending order, the ones following it - in descending order.

3.3 Two-dimensional arrays

Arrays can have more than one dimension. In such cases, one speaks of multidimensional arrays. They are widely used in statistics and mathematics (for example, for matrix calculations).



The declaration (description) of a two-dimensional array:

```

const m=8; n=6;
var A: array[1..m, 1..n] of real;
where A – the name of the array,
1..m – borders of changing the first index,
1..n – borders of changing the second index,
real – type of the elements of the array
or:
const m=30; n=50;
type matr=array[1..m,1..n] of real;
var A: matr;

```

The two-dimensional array can be formed in one of three ways: keyboard input, random number generator, read from file.

Entering elements of a two-dimensional array from the keyboard:

```

readln(m,n);
for i:=1 to m do
for j:=1 to n do
read(x[i,j]);

```

Output of elements of a two-dimensional array on the screen:

```

for i:=1 to m do
begin {output of the ith line of the array}
for j:=1 to n do
write (x[i, j]);
writeln;
end;

```

3.3.1 Square Matrices

A two-dimensional array whose number of rows is equal to the number of columns is called a square matrix.

The main diagonal of square matrix:

A ₁₁	A ₁₂	A ₁₃	A ₁₄	A ₁₅
A ₂₁	A ₂₂	A ₂₃	A ₂₄	A ₂₅
A ₃₁	A ₃₂	A ₃₃	A ₃₄	A ₃₅
A ₄₁	A ₄₂	A ₄₃	A ₄₄	A ₄₅
A ₅₁	A ₅₂	A ₅₃	A ₅₄	A ₅₅

A ₁₁	A ₁₂	A ₁₃	A ₁₄	A ₁₅
A ₂₁	A ₂₂	A ₂₃	A ₂₄	A ₂₅
A ₃₁	A ₃₂	A ₃₃	A ₃₄	A ₃₅
A ₄₁	A ₄₂	A ₄₃	A ₄₄	A ₄₅
A ₅₁	A ₅₂	A ₅₃	A ₅₄	A ₅₅

The properties of the indices of the elements of the square matrix relative to the main diagonal:

1. belonging to the main diagonal: $i - j = 0$;
2. located below the main diagonal: $i < j$;
3. located above the main diagonal: $i > j$;
4. located on lines parallel to the main diagonal: $i - j = \text{const}$;
namely, $(n-1) \leq i - j \leq n-1$.

The properties of the indices of the elements of the square matrix relative to the auxiliary diagonal:

1. belonging to the auxiliary diagonal: $i+j=n+1$;
2. located below the auxiliary diagonal: $i+j < n+1$;
3. located above the auxiliary diagonal: $i+j > n+1$;
4. located on lines parallel to the auxiliary diagonal: $i+j = \text{const}$;
namely, $2 \leq i+j \leq 2 \cdot n$.

Auxiliary diagonal of the square matrix:

A_{11}	A_{12}	A_{13}	A_{14}	A_{15}
A_{21}	A_{22}	A_{23}	A_{24}	A_{25}
A_{31}	A_{32}	A_{33}	A_{34}	A_{35}
A_{41}	A_{42}	A_{43}	A_{44}	A_{45}
A_{51}	A_{52}	A_{53}	A_{54}	A_{55}

A_{11}	A_{12}	A_{13}	A_{14}	A_{15}
A_{21}	A_{22}	A_{23}	A_{24}	A_{25}
A_{31}	A_{32}	A_{33}	A_{34}	A_{35}
A_{41}	A_{42}	A_{43}	A_{44}	A_{45}
A_{51}	A_{52}	A_{53}	A_{54}	A_{55}

The condition that the elements of a square matrix belong to one row, column:
The first (last) indices for all elements of a row (column) are the same.

$i = \text{const}$ ($i=3$)

$j = \text{const}$ ($j=3$)

A_{11}	A_{12}	A_{13}	A_{14}	A_{15}
A_{21}	A_{22}	A_{23}	A_{24}	A_{25}
A_{31}	A_{32}	A_{33}	A_{34}	A_{35}
A_{41}	A_{42}	A_{43}	A_{44}	A_{45}
A_{51}	A_{52}	A_{53}	A_{54}	A_{55}

A_{11}	A_{12}	A_{13}	A_{14}	A_{15}
A_{21}	A_{22}	A_{23}	A_{24}	A_{25}
A_{31}	A_{32}	A_{33}	A_{34}	A_{35}
A_{41}	A_{42}	A_{43}	A_{44}	A_{45}
A_{51}	A_{52}	A_{53}	A_{54}	A_{55}

Let us consider various variants of formation of elements of a two-dimensional array on the specified property.

Example 1. Create a program for filling a two-dimensional array $A[m, n]$ by nulls and ones in the staggered order.

```

var a:array[1..10,1..10] of integer;
    n,i,j:integer;
begin
    writeln('n: ');
    readln(n);
    for j:=1 to n do
        if (i+j) mod 2 = 0 then
            a[i,j]:=1
        else
            a[i,j]:= 0;
    for i:=1 to n do
        begin
            for j:=1 to n do write (a[i,j]:2);
            writeln
        end;
    readln

```

end.

Example 2. Create a program for filling a two-dimensional array "on a snake". For example, a rectangular matrix **A[3,4]** has the following type:

```
1  2  3  4
8  7  6  5
9 10 11 12
```

```
var a: array[1..10,1..10] of integer;
    m,n,i,j,k:integer;
begin
    writeln('m,n:');
    readln (m,n);
    for i:=1 to m do
        if i mod 2 <> 0 then
            for j:=1 to n do
                begin
                    k:=k+1; a[i,j]:=k
                end
            else
                for j:=n downto 1 do
                    begin
                        k:=k+1; a[i,j]:=k
                    end;
                for i:=1 to m do
                    begin
                        for j:=1 to n do
                            write(a[i,j]:6);
                            writeln
                        end;
                    end;
                end.

```

You can suggest another way to fill a two-dimensional array "on the snake":

```
for j:=1 to n do
    if i mod 2 <> 0 then
        a[i,j]:=(i-1)*n+j
    else
        a[i,j]:=i*n-j+1;

```

Example 3. A Latin square of order N is a square matrix of size N x N, each row and column of which contains the numbers 1, 2, 3, ..., N. Write a program for constructing a Latin square of dimension N.
Solution: We can suggest the following algorithm for constructing a Latin square:

1. Fill the first line with the numbers 1, 2, 3, ..., N;
 2. Lines, starting with the second, fill the elements of the previous line, shifted along the cycle to the left or right:
- ```
for j:=1 to n do a[i,j]:=j;
for i:=2 to n do

```

```

begin
 a[i,1]:=a[i-1,n];
 for j:=2 to n do a[i,j]:=a[i-1,j-1];
end;

```

**Example 4.** Create a program for constructing Pascal's triangle:

```

 1
 1 1
 1 2 1
1 3 3 1
1 4 6 4 1

```

```

uses crt;
var a: array [1..10, 1..10] of integer;
 n,i,k,j,x,y,l :integer;
begin
 write('n='); readln(n);
 x:=40;
 k:=0;
 clrscr;
 goto(x-10,1);
 writeln('Triangle of Pascal when N=',n,':');
 gotoxy(x,3); writeln('1');
 for i:=1 to n do
 begin
 k:= k+1;
 a[i,1]:=1; a[i,k+1]:=1;
 for j:=2 to k do
 a[i,j]:=a[i-1,j-1]+a[i-1,j];
 x:=x-3; l:=x;
 for j:=1 to k+1 do
 begin
 gotoxy(l,i*2+3);
 write (a[i,j]);
 l:=l+6;
 end;
 end;
 readln
 end.

```

### Tasks for independent work

1. Fill a two-dimensional array A (N×N) by the rule:  
 1 - if the element belongs to the diagonals;  
 0 - if it doesn't belong.



2. An arithmetic square of order  $N$  is a square matrix of size  $N \times N$  whose first row and first column are filled with ones, and each of the remaining elements is equal to the sum of its neighbors on the left and above. Draw up a program for constructing an arithmetic square of dimension  $N$ .

For example when  $N=4$  we have:

|   |   |    |    |
|---|---|----|----|
| 1 | 1 | 1  | 1  |
| 1 | 2 | 3  | 4  |
| 1 | 3 | 6  | 10 |
| 1 | 4 | 10 | 20 |

3. Form the Pythagorean matrix (multiplication table in matrix form) and display it on the screen.
4. Form a square matrix as follows: the elements of the main diagonal are equal to 1, the elements located on lines parallel to the main diagonal (above and below) are equal to 2, 3, ...

For example, for  $N = 4$  we have the following matrix:

|   |   |   |   |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 2 | 1 | 2 | 3 |
| 3 | 2 | 1 | 2 |
| 4 | 3 | 2 | 1 |

5. Fill the square matrix according to the following pattern:

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 2 | 2 | 2 | 1 |
| 5 | 1 | 2 | 2 | 2 | 1 | 3 |
| 5 | 5 | 1 | 2 | 1 | 3 | 3 |
| 5 | 5 | 5 | 1 | 3 | 3 | 3 |
| 5 | 5 | 1 | 4 | 1 | 3 | 3 |
| 5 | 1 | 4 | 4 | 4 | 1 | 3 |
| 1 | 4 | 4 | 4 | 4 | 4 | 1 |

6. The chessboard can be represented as a two-dimensional array of  $8 \times 8$ . Initially, the array is filled with zeros. From the keyboard are entered the coordinates of the queen in the form of column number and line number on the chessboard. Fill in units of the array elements corresponding to the cells threatened by the queen.

### 3.3.2 Processing two-dimensional arrays

You can select several types of tasks for processing two-dimensional arrays.

1. Finding the sum, quantity, product, indices of elements with this property.
2. Arithmetic operations on matrices.
3. Check the elements of the array for the specified condition.
4. Transformation of array elements according to the specified method.
5. Insert and delete elements of the array.
6. Rearrange the elements of the array.

**Example 1.** A two-dimensional array of dimension  $m \times n$  is given. Find the number of negative elements in each row of the array.

Let us consider some variants of the decision of the given problem.

1. You can store the number of negative elements of each row in a one-dimensional array of the corresponding dimension:

```
procedure KOL1(x:dmyarray; var y:omyarray);
 var i,j:integer;
begin
 y[i]:=0;
 for j:=1 to m do
 if x[i,j]<0 then inc(y[i]);
 end;
```

2. You can use the counter, find the number of negative elements of the line and immediately display the found values on the screen:

```
procedure KOL2(x:dmyarray) ;
var i,j,k:integer;
begin
 for i:=1 to n do
 begin
 k:=0;
 for j:=1 to m do
 if x[i,j]<0 then inc(k);
 writeln(i,'-',k);
 end;
```

**Example 2.** Compile a program for computing the product of two rectangular matrices  $A(m, t) * B(t, n)$ .

Elements of the resulting matrix  $C(m, n)$  are determined by the formula:

$$c(i, j) = \sum_{k=1}^t a[i, k] * b[k, j]$$

```
procedure PROIZV(x,y:dmyarray; var z:dmyarray);
 var k,i,j,t:integer;
begin
 for i:=1 to n do
 for j:=1 to n do
 begin
 z[i,j]:=0;
 for k:=1 to t do
 z[i,j]:=z[i,j]+x[i,k]*y[k,j];
 end;
```

**Example 3.** Determine whether a given square matrix is symmetric with respect to the main diagonal.

|        |        |     |        |
|--------|--------|-----|--------|
| a[1,1] | a[1,2] | ... | a[1,n] |
| a[2,1] | a[2,2] | ... | a[2,n] |
| ...    | ...    | ... | ...    |
| a[i,1] | a[i,2] | ... | a[i,n] |

$\begin{matrix} & \dots & & \dots & & \dots & & \dots \\ a[n,1] & & a[n,2] & & \dots & & a[n,n] \end{matrix}$

For a square matrix, the condition  $i=j$  determines the elements of the main diagonal, where  $i < j$  are the elements of above level than the main diagonal,  $i > j$  are the elements of below level than the main diagonal.

If the condition  $a[i, j] = a[j, i]$  holds for all  $i = 1, 2, \dots, n$  and  $j = 1, 2, \dots, n, i > j$ , then the matrix is symmetric. Let us check the square matrix for symmetry as a function:

```
function SIMMETR(x:dmyarray):boolean;
 var i,j:integer;
 t:boolean;
begin
 t:=true;
 i:=2;
 while t and (i<=n) do
 begin
 j:=1;
 while (j<i) and (x[i,j])=x[j,i]) do
 inc(j);
 t:=(j=i);
 inc(i);
 end;
 simmetr:=t;
 end;
```

If there is at least a pair of such elements such that  $x[i, j] \neq x[j, i]$ , the function will return false, if no such pair meets, the value of the function will remain true.

**Example 4.** Given a two-dimensional array of dimension  $m \times n$ , whose elements are integers. Perform a "mirror image" of the matrix elements with respect to the vertical axis of symmetry.

```
procedure OTOBR(var x:dmyarray);
 var i,j,b:integer;
begin
 for j:=1 to n div 2 do
 for i:=1 to m do
 begin
 b:=a[i,j];
 a[i,j]:=a[i,n-j+1];
 a[i,n-j+1]:=b;
 end;
```

**Example 5.** A matrix of dimension  $m \times n$  is given. Among the smallest elements of the rows, find the largest element and determine its location in the matrix.

```

uses crt;
const
 m=10;
 n=15;
type matr=array[1..m,1..n] of integer;
var
 a:matr;
 i,j,k,l:integer;
 min,b:array [1..m] of integer;
begin
 ClrScr;
 readln(m,n);
 {Find the minimal elements of each line and put
 them in an array min[1..m]}
 for i:=1 to m do
 begin
 min[i]:=a[i,1];
 b[i]:=1;
 for j:=2 to n do
 if a[i,j]<min[i] then
 begin
 min[i]:=a[i,j];
 b[i]:=j
 end;
 end;
 {Find the maximum among the elements of the array
 min[1..m]}
 k:=1;
 for i:=2 to m do
 if min[i]>min[k] then k:=i;
 writeln('Maximal among the minimum elements of
 the lines:', min [k], '', its location: line
 number- ', k,' column number - ', b [k]);
 readln
end.

```

**Example 6.** Delete the string with the number  $k$  from the array.

To delete a row with the number  $k$ , you must:

1. Move all lines starting from this one up one line.
2. The last line is "null", that is, assign the value 0 to all elements of the last row.
3. Dimension (number of rows) of the array is reduced by 1.

```

procedure DELETE(k:integer; var x:dmyarray);
var i,j:integer;
begin

```

```

 for i:=k to m-1 do
 for j:=1 to n do
 x[i,j]:=x[i+1,j];
 for j:=1 to n do
 x[m,j]:=0;
 m:=m-1
 end;

```

Using this procedure, you can solve the problem of deleting rows of an array that satisfy the given condition.

### Tasks for independent work

1. Find out if there is a column in the array that has the same elements.
2. Create an algorithm that, for a given square matrix, calculates the sums of elements located on lines parallel to the main (secondary) diagonal.
3. The matrix A (MxN) is given. Get the vector from the minimal column elements of this matrix.
4. Count the number of columns in the array whose elements are sorted in descending order.
5. Reverse the elements of the rows of the array containing duplicate elements.
6. Seryozha and Natasha played tic-tac-toe. If none of them succumbed, then there was always a draw. Then they began to try to play in fields of other sizes. A field of size NxN cells can be represented as a matrix in which 1 corresponds to a cross, and 0 to a zero. Write a program that, according to the filled matrix, determines the winner.

### 3.4 Line data type

The line is a special form of a one-dimensional array of characters, which has a significant difference. An array of characters has a fixed length (the number of elements), which is defined in the description. The line has two varieties of length:

- the total length of the line that characterizes the memory size allocated to the line in the description;
- the current length of the line (less than or equal to the total length), which shows the number of semantic characters of the line at each particular moment in time.

The string can be declared (described) as follows:

```

var s1:string[10];
 s2:string[120];
 s3:string;

```

If the length of the string in the description is not specified, then the default length is 255 characters.

String variables are displayed using standard `Write` and `Writeln` procedures and are entered using standard `Read` and `Readln` procedures. Thus, the lines are not entered element by element, but whole.

### String operations

In Pascal, there are two basic ways of handling variables of type `String`.

The first method involves processing the entire line as a single whole, i.e. single object.

The second method - treat the string as a composite object, consisting of individual characters (`Char` type elements) that are available when processing each separately.

**Gluing** is meant as the consecutive combination of several strings. The operation of gluing the lines is indicated by the symbol "+".

Example: `'infor'+'mation'='information'`.

Gluing of rows can also be organized using the function `concat(S1, S2, ..., Sn)`.

For example:

```
var s1,s2,s3:string;
begin
 s1:='infor';
 s2:='mation';
 s3:=concat(s1,s2);
end.
```

The variable `S3` takes the value 'information'.

Pascal allows you to perform two-line **comparison** operations. The comparison takes place character-by-symbol from left to right: the codes of the corresponding symbols are compared until the equality of the codes is broken or the end of one of the lines is reached. Two lines are called equal if they are equal in length and coincide by character.

For example:

```
'computer' > 'Computer' (ord('c')>ord('C'))
'Algol' > 'Algel' (ord('o')>ord('e'))
'algorithm1'>'algorithm'
(length(str1)>length(str2))
'IBM' = 'IBM'
```

When comparing strings, you can use the relations operations (`>`, `<`, `=`, `>=`, `=`, `=`) in conditional statements. The result of the comparison operation is `True` or `False`.

To work with strings of type `String` in Pascal, standard functions and procedures are used.

To **remove** a fragment from the line, use the procedure `Delete(Str, n, m)`:

For example:

```
Str1:='Programming';
Delete(Str1,4,5);
```

```
Writeln(Str1);
```

```
Str1='Proing'.
```

To **insert** a substring in a string, use the procedure  
`Insert(Str1, Str2, N) :`

For example:

```
Str1:='formati';
```

```
Str2:='Inon';
```

```
Insert(Str1, Str2, 3);
```

```
Str2='Information'.
```

Function `Copy(Str, N, M)` copies M symbols of the string Str, beginning with N-th symbol.

Example:

```
Str:='Information';
```

```
Copy(Str, 3, 6)='format';
```

```
Copy(Str, 3, 5)='forma';
```

Function `Length(Str)` gives the number of symbols in the string.

For example:

```
Length('Information')=11
```

```
Length('true or false')=13
```

Function `Pos(Str1, Str2)` defines the position of entering string Str1 into the string Str2.

Example:

```
Pos('for', 'information')=3
```

```
Pos('the', 'if then else')=4
```

```
Pos('abc', 'ab cd ef')=0
```

```
Pos('ab', 'ab cd ab')=1
```

Procedure `Str(N, Str1)` transfers the numerical type N into string type Str1.

For example:

```
Str(121, Str1); Str1='121'
```

```
Str(25.17, str2); str2='25.17'
```

```
Str(4.25e+2, Str3); Str3='4.25e+2'
```

Procedure `Val(Str, N, K)` transfers the string type meaning Str into numerical N.

For example:

```
Val('1234', n, k); n=1234; k=0;
```

```
Val('23.56', n, k); n=23.56; k=0;
```

```
Val('2. 78e+3', n, k); n=2.78e+3; k=0;
```

```
Val('2, 78', n, k); k=2 (error: «,»)
```

`val('75.12c-5', n, k); k=6 (error:symbol «c» is not allowed in the numbers writing).`

**Example 1.** Calculate the number of digits included in this string.

```
var s:string;
```

```
 i,k:byte;
```

```
begin
```

```

write('Input string:');
readln(s);
k:=0;
for i:=1 to length(s) do
 if (s[i]>='0') and (s[i]<='9') then
 k:=k+1;
writeln('k=',k);
readln;
end.

```

**Note:** You can use the procedure Val:

```

k:=0;
for i:=1 to length(s) do
 begin
 val(s[i],n,d);
 if d=0 then
 k:=k+1;
 end;
end;

```

**Example 2.** Delete repeated occurrences of characters in the string.

Variant 1:

```

var i,j:integer;
s,ss:string;
begin
 readln(s);
 ss:='';
 for i:=1 to length(s) do
 if pos(s[i],ss)=0 then
 ss:=ss+s[i];
 end;
end;

```

Variant 2:

```

var i,j,k:byte;
st:string;
begin
 readln(st);
 for i:=1 to length(st)-1 do
 for j:=i+1 to length(st) do
 if st[i]=st[j] then
 begin
 delete(st,j,1);
 end;
 end;
 end;
end;

```

**Example 3.** Two strings of characters are given. Compose a third string of non-repeated characters, simultaneously entering both the first and second strings.

```

var s1,s2,s3:string;
i,k:integer;

```



```

begin
 writeln('s1:');
 readln(s1);
 writeln('s2:');
 readln(s2);
 for i:=1 to length(s1) do
 if (pos(s1[i],s2)<>0) and (pos(s1[i],s3)=0)
 then s3:=s3+s1[i];
 writeln(s3);
 readln;
end.

```

## Tasks for independent work

1. A word is given. Get a copy and reference (a word obtained by reading the original word, beginning with its end) of the given word.
2. Compose a program that forms a string consisting of any given number of any identical characters.
3. There is a string of characters. Add to it at the beginning and at the end a given number of asterisks, i.e. characters "\*".
4. Calculate how many times in a given line the specified character meets?
  1. How many vowels are there in this line?
  2. To check whether the following line is a reversed row (read from left to right and from right to left): after removing all spaces from it:
    - a) "ARGENTINA BACKENS NEGRO" (АРГЕНТИНА МАНИТ НЕГРО);
    - b) "A ROSE FELL TO THE PAW OF AZOR" (А РОЗА УПАЛА НА ЛАПЫ АЗОРА).
5. The text is given. Find the largest number of consecutive identical characters in a string.
6. Three lines are given. Determine whether it is possible to get the third line from the characters of the first two lines.
7. A string of characters is given. All digital symbols are rearranged to the beginning of the line, rearranging them in the reverse order
8. A word is given. Determine how many different letters in it.
9. The text is given. Find the length of its shortest word.

### 3.3 Enumerated data type

Values of this type form an ordered set and are constants.

#### Example:

```

Type Gaz=(Ge,C,O,H):
 Metall=(Na,K,Zi,Ca,Zn);
Var
 G1,G2: Gaz;
 Met1,Met2: Metall;

```

```

Season: Winter, Spring, Summer, Autumn;
Variables of an enumerated type are built-in functions Ord, Pred и
Succ:
Ord(Spring)=1
Pred(Autumn)=2
Succ(Winter)=1

```

The values of these types do not apply to arithmetic operations, nor standard I/O procedures as Read, Readln, Write, Writeln. But you can enter a number that is the sequence number of an element of the enumerated type, using, for example, the selection operator such as Case:

```

writeln('Enter the number of the element');
readln(k);
case k of
 0:writeln('Winter');
 1:writeln('Spring');
 2:writeln('Summer');
 3:writeln('Autumn');
else
 writeln('there is no such an element in the
 task');
end;

```

### Task for independent work

1. Define the enumerated type that describes the days of the week. Write functions to work with this type.
  - getting the value of the listed type by the day of the week number;
  - get a string with the name of the day of the week by the value of the enumerated type.
2. Using these functions, write a program that determines which day of the week the given date is (take only the current year).

### 3.5 Sets

**The set** in Pascal is a set of different elements of the same (basic) type. Sets in Pascal can only include elements of ordinal types.

Sets can be described as follows:

```

Type <name> = set of <type of elements>;
var <the name of the set>:<the name of the type>;
or:
var <the name of the set>: set of <type of
elements>;

```

For example:

```

Type mnog=set of char;
var mnl:set of char;

```

```

mn2: mnog;
mn3: set of 'a'..'z';
a:set of byte;
b:set of 100..200;

```

**Formation (construction) of sets.** In the program, the elements of the set are given in square brackets, separated by a comma. If the elements go successively one after the other, then you can use the range.

For example:

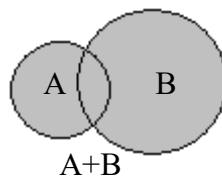
```

type mnog=set of 1..200;
var a,b,c,d:mnog;
begin
 a:=[3,9,67,5];
 b:=[7..60];
 c:=[1..10,80..90];
 d:=[];
end;

```

## Operations over sets

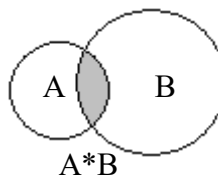
**The union** of two sets is the set of elements belonging to at least one of these sets.



Examples:

- 1) ['A','F']+['B','D']= ['A','F','B','D'];
- 2) [1,2,7,4,6,8] + [3,4,5,6] = [1..8]
- 3) [1..3,5,7,11] + [3..8,10,12,15..20]=  
[1..8,10,11,12,15..20]

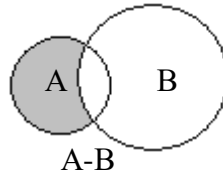
**The intersection** of two sets is the set of elements that belong to both the first and the second set simultaneously.



Examples:

- 1) ['A','F']\*['B','D']=[]
- 2) [1,2,3,4]\*[3,4,5,6]=[3,4]
- 3) [1..3,5,7,11]\*[3..8,10,12,15..20]=[3,5,7]

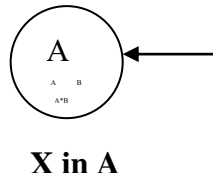
**The difference** between two sets is a set consisting of those elements of the first set that are not elements of the second set.



Examples:

- 1)  $['A', 'F'] - ['B', 'D'] = ['A', 'F']$
- 2)  $[1, 2, 3, 4] - [3, 4, 5, 6] = [1, 2]$
- 3)  $[1..3, 5, 7, 11] - [3, 8, 10, 12, 15..20] = [1, 2, 11]$

The operation of **determining whether an element belongs to a set**. The mathematical notation of the operation is " $\in$ ", in Pascale - "in". The result of the operation is *true* if the element is in the set, and *false* - otherwise.

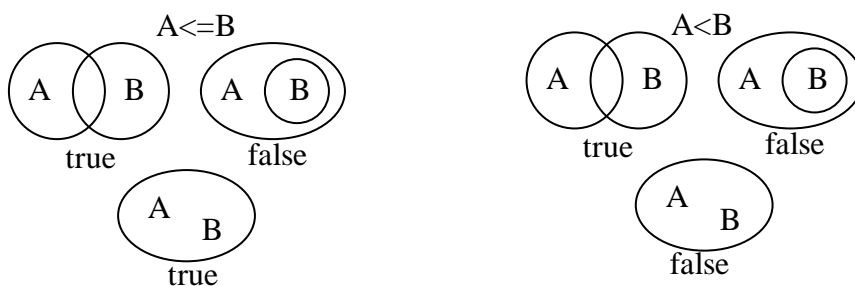


Examples:

- 1)  $4 \text{ in } [3, 4, 6, 8] = \text{true}$
- 2)  $'a' \text{ in } ['A'..'Z'] = \text{false}$

## Comparison of sets

For comparison of sets, the operations of the relation are used:  
 = check for equality (coincidence) of two sets;  
 <> check for the inequality (mismatch) of two sets;  
 <=, < check for the occurrence of the first set in the second;  
 >=, > check for the occurrence of the second set in the first.



Sets are convenient to use when processing strings and texts, as well as when working with ordered sets of numbers.

**Example.** "Sieve of Eratosthenes". Make a program for finding prime numbers in the interval  $[1..n]$ .

*Desicion:*

A simple number is a number that has no other divisors except one and the number itself. The idea of this method is as follows: a set  $M$  is formed in which

all the numbers of a given interval are contained. Then, from the set, we successively remove elements that are multiples of 2, 3, 4...  $n \div 2$ :

```
program Simple;
var m:set of byte;
 i,k,n:integer;
begin
 write('n=');
 readln(n);
 m:=[2..n];
 for k:=2 to n div 2 do
 for i:=2 to n do
 if (i mod k=0) and (i<>k) then
 m:=m-[i];
 for i:=1 to n do
 if i in m then
 write(i:3);
 readln
 end.
```

### Tasks for independent work

1. A natural number N is given. Write a program that prints all the digits that are not in the decimal notation of a given natural number in ascending order.

2. Create a program to print all the characters of the specified text, included in it once.

3. Create a program for calculating the number of vowels and consonants in the given text and determining which letters are larger (vowels or consonants); note that the string may contain other characters, except letters.

4. Solve the rebuses:

a) lob+tri=sam /люб + три = сам

b) iks+isk=ksi /икс + иск = кси

a) abc=ab + bc + ca

b) tochk+krug=konus /точк + круг = конус

c) volvo + fiat = motor

d) mukha+mukha=slon /муха + муха = слон

5. Create a print program in alphabetical order (one time) of all lowercase Russian vowels entering the given text.

### 3.7 Graph model of the algorithm

One way to solve non-standard problems is to use algorithms on graphs.

The main algorithms on graphs are:

– sorting the vertices of the graph in a wide width search ("wave method" or fill method, overflow with return or "trial and error method");

- Dijkstra's algorithm for finding the shortest paths in a graph from a given vertex to all the others;
- Floyd's algorithm for finding the shortest paths in a graph between all pairs of vertices and others.

Graph theory as a mathematical discipline appeared in 1736, when L. Euler (1707 - 1782, a Russian mathematician, a Swiss by birth, an academician of the Petersburg and Berlin Academies of Sciences), solved the widely known at that time task of the Koenigsberg (now Kaliningrad) bridges. This result remained for more than one hundred years unique in graph theory.

Interest in the problems of graph theory revived around the middle of the last century and was concentrated mainly in England. There were many reasons for this revival of the study of graphs: the natural sciences had their influence on this by exploring electrical networks, crystal models and molecular structures. At present, this theory is widely developed and successfully applied.

Graphs can depict schemes of roads, communications, electrical circuits, molecules of chemical compounds, connections between people and groups of people, etc. In terms of graph theory, most problems associated with discrete objects are formulated. Such problems arise in the design of integrated circuits and control schemes, block diagrams of programs, in economics, statistics, biology, scheduling theory, discrete optimization and other fields.

### Basic concepts and definitions of graph theory

A graph is a set of points and a set of segments whose ends belong to a given set of points.

Vertices are called *points* or *nodes*, and lines connecting vertices, *edges* or *arcs*, or *lines*. The edge can have a direction from the vertex A to the vertex B or vice versa; in this case such an edge is called a *directed arc*. A vertex A is called an *initial vertex*, and B is a *finite vertex* of an edge. Vertices are indicated in capital letters or numbers. Vertices that do not belong to any edge are called *isolated vertices*.

Let us consider the problem of the Koenigsberg bridges. The city was located on the banks of the river Pregel and two islands of the river. Different parts of the city were connected by seven bridges. The layout of the bridges in Koenigsberg is shown in Figure 3.1. On Sundays, the townspeople made walks around the city. The task is to pass each bridge one time and return to the starting point.

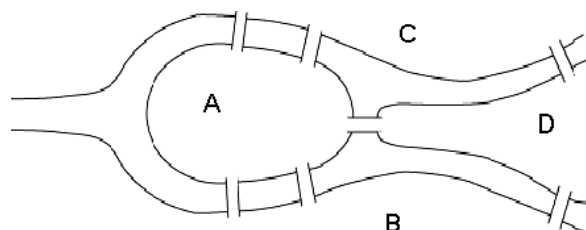


Figure 3.1 Koenigsberg bridges

To prove that the problem does not have a solution, Euler denoted each part of the land by a point (vertex), and each bridge by a line (edge) connecting the corresponding parts of the land. A graph was obtained (Figure 3.2), in which the vertices are marked with the same letters as the four parts of the land in Figure 3.1

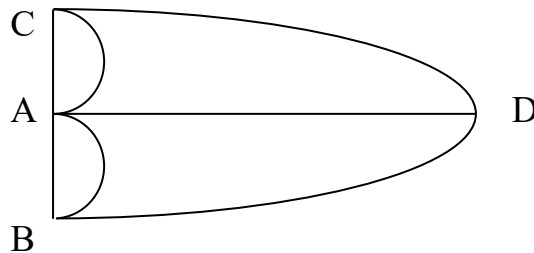


Figure 3.2. Graph model of Koenigsberg bridges

The task was as follows: *to find the route of the passage of all four parts of the land (A, B, C, D), which would begin on any of them, would end on the same part of the land and pass exactly once on each bridge.* Euler proved that this graph does not represent a single cycle; in other words, from whatever vertex we start the bypass, we cannot go around the whole graph and go back without passing any edge twice. If such a cycle existed, then at each vertex of the graph there would be as many edges entering into it as there are leaving it; at each vertex of the graph there would be an even number of edges, however, this condition is not satisfied for the graph representing the map of Koenigsberg. Euler found in a generalized form the criterion for the existence of such a route in the graph.

### Machine Representation of Graphs

A classical way of representing graphs is the *incidence matrix*, which is the matrix  $A$  with  $n$  rows corresponding to the vertices and  $m$  columns corresponding to the edges. For an oriented graph, the column corresponding to the arc  $\langle x, y \rangle \in E$  contains  $(-1)$  in the row corresponding to the vertex  $x$ , and  $(1)$  in the row corresponding to the vertex  $y$ , and zeros in all other rows (it is convenient to represent the loop with a different value in row  $x$ , for example,  $2$ ). In the case of an undirected graph, the column corresponding to the edge  $\{x, y\}$  contains  $1$  in the rows corresponding to  $x$  and  $y$ , and zeros in the remaining rows. From the algorithmic point of view, the incidence matrix is not a very good representation of the graph: first, it requires  $n * m$  memory cells, and most of these cells are generally occupied by zeros. Access to information is also inconvenient. The answer to elementary questions requires, in the worst case, a search of all columns of the matrix, and consequently  $m$  steps.

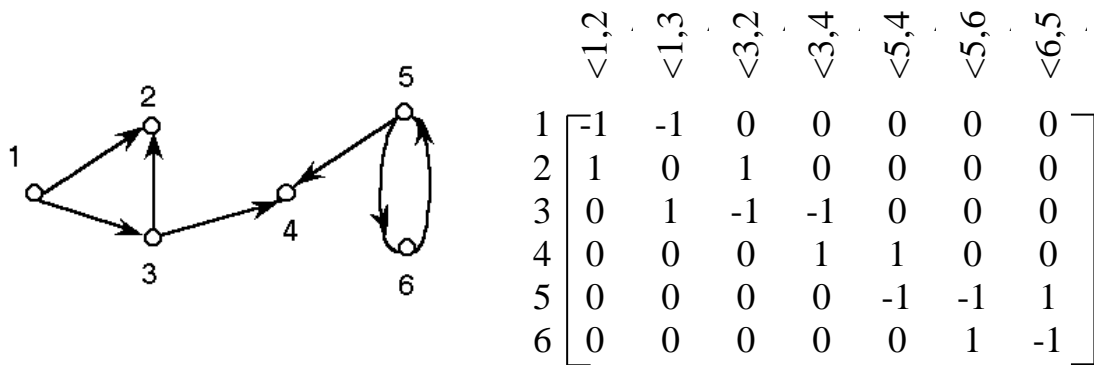


Figure 3.3. Oriented graph and its incidence matrix

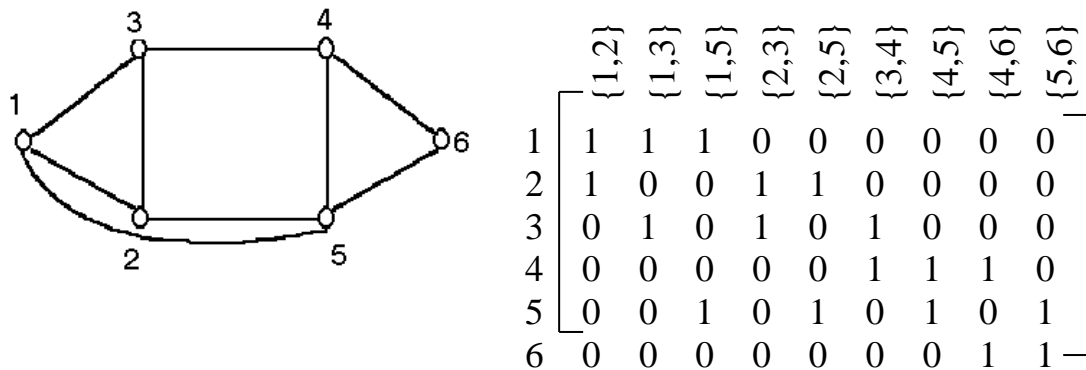


Figure 3.4 An undirected graph and its incidence matrix

A more convenient way to represent a graph is an *adjacency matrix* - a square matrix of size  $n * n$ , where  $b_{ij} = 1$ , if there exists an edge leading from vertex  $x$  to vertex  $y$ , and  $b_{ij} = 0$  - otherwise. Since in an undirected graph the edge  $\{x, y\}$  goes both from  $x$  to  $y$  and from  $y$  to  $x$ , the adjacency matrix of such a graph is always symmetric with respect to the main diagonal. The disadvantage is the fact that, regardless of the number of edges, the amount of occupied memory is  $n^2$ .

It is often required to find vertices in the graph that have certain properties. To do this, you need to bypass the graph, that is, view all its vertices. Therefore, graph bypass algorithms are often called *search* or *browsing algorithms*.

**Task 1.** Create a program to view all the vertices of the graph using the "in-depth" search method (Deep Force Search -DFS).

*Note:* A method called "deep force search - DFS" (retraction) in the graph became one of the main methods of programming and graph algorithms.

Viewing of graph vertices begins with some vertex  $u$ . A vertex  $v$  adjacent to  $u$  is chosen. The process is repeated from the vertex  $v$ . If at the next step there are no vertices adjacent to  $u$  and not previously viewed (new), then we return from vertex  $u$  to the vertex from which we got to  $u$ . If all the vertices of the graph are viewed (returned to the first vertex by stack), the view is finished.

*Solution:* Let the graph be described by the adjacency matrix  $A$ . The view starts from the first vertex. Figure 3.8.5 shows the original graph, and in Figure 3.8.6, the vertices in parentheses indicate the order in which the vertices of the graph were viewed during the depth search.



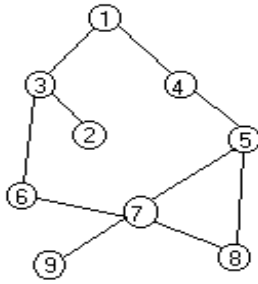


Figure 3.5 Initial graph

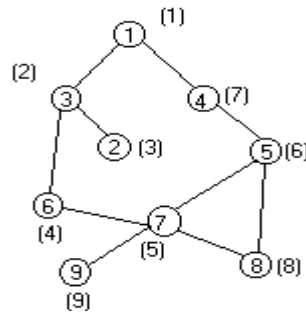


Figure 3.6 Vertices view

Initial file has the following view:

```

N
A[1,1] A[1,2] ... A[1,N]
A[2,1] A[2,2] ... A[2,N]
.
.
.
A[N,1] A[N,2] ... A[N,N] }
```

Implementation of the algorithm for enumerating the vertices of the graph in depth (Deep Force Search - DFS):

```

Var Pom:Array[1..20] of Boolean;
 i,j,yk,n:Integer;
 f:Text;
 r:Char;
 s:String;
 a:Array[1..20,1..20] of Integer;
{ Recursive depth-search procedure }
Procedure DFS(v:Byte);
Var v:Byte;
Begin
 Write(u, ' ');
 Pom[u]:=False;
 For v:=1 To n Do
 If (a[u,v]=1) and Pom[v]
 Then
 DFS(v);
End;
Begin
 Assign(f, 'glub.in');
 Reset(f);
 Readln(f, n);
 For i:=1 To n Do
 For j:=1 To n Do
 Read(f, a[i, j]);
 FillChar(pom, sizeof(pom), True);
 k:=0;
```

```

 For i:=1 To n Do
 If Pom[i] Then
 Begin
 Inc(yk);
 DFSg(i);
 End;
 Writeln(k)
 End.

```

In view of the importance of this algorithm, we consider its *non-recursive* realization. Global arrays are the same: Sm is an adjacency matrix, Pom is an array of vertex attributes (new / viewed). The numbers of the viewed vertices are stored in the St stack, yk is the stack pointer.

```

 Procedure DFS1(v:Integer);
 Var St:Array[1..N] of Integer;
 yk:Integer;
 t,j:Integer;
 flag:Boolean;
 Begin
 FillChar(St,sizeof(St),0);
 yk:=0;
 Inc(yk);
 St[yk]:=u1;
 Pom[u1]:=False;
 While yk<>0 do
 Begin
 u:=St[yk]; {extract the vertex from the stack}
 v:=0;
 flag:=false;
 Repeat
 v:=v+1;
 If (Sm[u,v]=1) and Pom[v]
 Then
 flag:=True
 Else
 Inc(v);
 Until flag or (v>=N);
 If flag Then
 Begin
 Inc(yk);
 St[yk]:=v; {add the vertex into the stack}
 Pom[v]:=False; {mark it as a visited}
 End
 Else
 Dec(yk);
 End;
 End;
 End;

```

*Methodical instructions:*

The method of tracing is a powerful tool in achieving an understanding of the essence of methods. Consider the operation of the algorithm using the example of the graph given above.

| Yk | St          | Pom               | u |
|----|-------------|-------------------|---|
| 1  | 1           | F t t t t t t t t | 1 |
| 2  | 3 1         | F t f t t t t t t | 3 |
| 3  | 2 3 1       | F f f t t t t t t | 2 |
| 2  | 3 1         | F f f t t t t t t | 3 |
| 3  | 6 3 1       | F f f t t f t t t | 6 |
| 4  | 7 6 3 1     | F f f t t f f t t | 7 |
| 5  | 5 7 6 3 1   | F f f t f f f t t | 5 |
| 6  | 4 5 7 6 3 1 | F f f f f f f t t | 4 |
| 5  | 5 7 6 3 1   | F f f f f f f t t | 5 |
| 6  | 8 5 7 6 3 1 | F f f f f f f f t | 8 |
| 5  | 5 7 6 3 1   | F f f f f f f f t | 7 |
| 5  | 9 7 6 3 1   | F f f f f f f f f | 9 |
| 4  | 7 6 3 1     | F f f f f f f f f | 6 |
| 3  | 6 3 1       | F f f f f f f f f | 3 |
| 2  | 3 1         | F f f f f f f f f | 1 |
| 1  | 1           | F f f f f f f f f | - |
| 0  |             |                   |   |

**Task 2.** Create a program to search through the graph vertices using the brute force (BFS) method.

Note: When searching in depth, the later the vertex will be visited, the earlier it will be used - more precisely, so it will be under the assumption that the second vertex has been visited before using the first one.

This is a direct consequence of the fact that viewed but not yet used vertices accumulate on the stack.

The width search, roughly speaking, is based on replacing the stack with a queue.

After this modification, the earlier visited vertex is (placed in the queue), the earlier it is used (it is removed from the queue).

The use of the vertex occurs by viewing all the neighbors of this vertex that have not yet been viewed.

At each step, one item is extracted from the start of the queue, and all the associated vertices that are not yet in the queue are added to the end, so the elements are marked as processed at the time they hit the queue, rather than extracting from it.

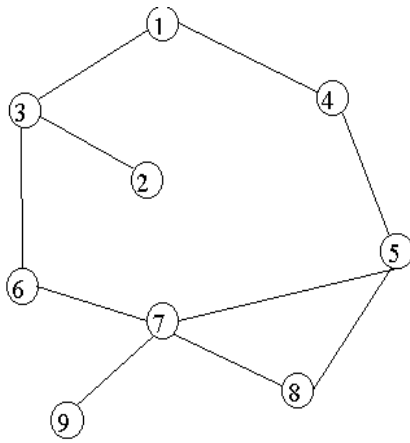


Figure 3.7

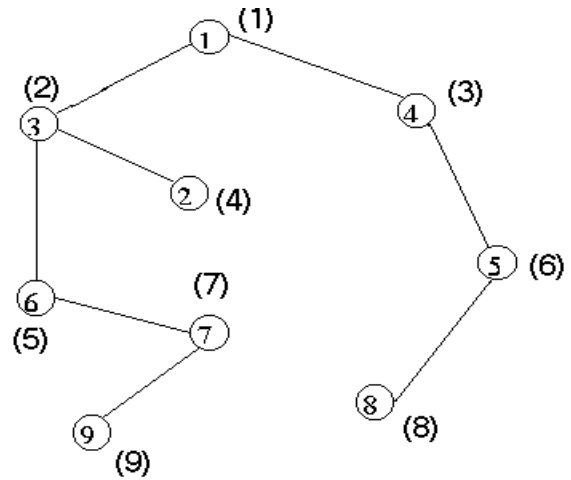


Figure 3.8

**Solution:** use the og array to store the viewed vertices.

```

Var Pom: Array[1..20] of Boolean;
 u,v,n:Integer;
 f:Text;
 Sm:Array[1..20,1..20] of Integer;
Procedure BFS(v:Integer);
Var och:Array[1..20] of 0..20;
 yk1,yk2:Integer;
 u:Integer;
Begin
 FillChar(og,sizeof(og),0);
 och[1]:=v;
 Pom[v]:=False;
 yk1:=1; {pointer to the beginning of the queue}
 yk2:=1; {pointer to the end of the queue}
 While yk1<=yk2 Do
 Begin
 Write(och[yk1], ' ');
 u:=och[yk1];
 For v:=1 To n Do
 If (Sm[u,v]=1) and Pom[v] Then
 Begin
 Pom[v]:=False;
 yk2:=yk2+1;
 och[yk2]:=v;
 End;
 End;
 End;
 End;
End;
Begin
 Assign(f,'BFS.in');
 Reset(f);

```

```

Readln(f,n);
For u:=1 to n do
 For v:=1 to n do
 Read(f,Sm[u,v]);
 FillChar(Pom,Sizeof(Pom),True);
 BFS(1);
 For u:=1 To yk1 Do
 Write(och[u],' ');
End.

```

*Note:* Both types of graph search - DFS and BFS - can be used to find the path between fixed vertices  $u_1$  and  $v_1$ . Start the search at the vertex  $u_1$  and run it until the vertex  $v_1$  is visited. The advantage of searching in depth is the fact that at the moment of visiting the vertex  $v$  the stack contains a sequence of vertices that determines the path from  $u$  to  $v$ . The drawback of the DFS algorithm is that the path thus obtained will not in general be the shortest path from  $u$  to  $v$ . A width search gives the shortest path, but to find it, you need to modify the BFS procedure. It is necessary to use a structure in which the numbers of the graph vertices are remembered, from where we came to the current vertex (numbers of the "ancestors" of the vertices):

```

If (Sm[u,v]=1) and Pom[v]
 Then
 Begin
 Pom[v]:=False;
 yk2:=yk2+1;
 och[yk2]:=v;
 predok[v]:=u;
 End;
 { Restoring the path from the queue }
 v:=v1;
 Repeat
 Write(v,' ');
 v:=predok[v];
 Until v=0;

```

**Task 3.** Make a program for constructing an attainability matrix.

*Note:* In an oriented graph, a path can be defined as a sequence of arcs in which the terminal vertex of any arc other than the last one is the initial vertex of the next.

If there exists a path from vertex  $u$  to vertex  $v$ , then we say that vertex  $v$  is reachable from vertex  $u$ .

The reachability matrix  $D$  is defined as follows:

$D[u, v] = 1$  if the vertex  $v$  is reachable from  $u$  and 0 otherwise

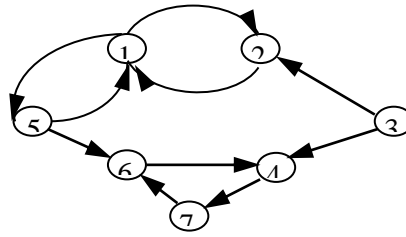


Figure 3.9 Matrix of the graph attainability

```

Var i,n,j:Integer;
 f:Text;
 Sm,Dost:Array[1..20,1..20] of Integer;
Procedure Reach;
Var S,T:Set of 1..20;
 u,v,l:Integer;
Begin
 For u:=1 To n Do
 Begin
 T:=[u];
 Repeat
 S:=T;
 For l:=1 To n Do
 If l in s Then
 For v:=1 To n Do
 If Sm[l,v]=1
 Then
 T:=T+[v];
 Until S=T;
 For v:=1 To n Do
 If v in T
 Then
 Dost[u,v]:=1;
 End;
 End;
 End;
 End;
 End;
 End;
End;

```

### Tasks for independent work:

1. Construct a graph model. Draw matrices Sm, Dost.
2. Elaborate the programs for finding the matrices Sm and Dost for various methods of describing the graph. Enter the description of the graph from the file (consider the case of a weighted graph).

## 4 Standard modules

### 4.1 Text screen mode

The standard module **Crt** includes procedures and functions for working in the text mode of the monitor.

Screen cleaning: `ClrScr`;

After cleaning the screen, the cursor moves to the upper left corner. Moving the cursor to the specified screen position:

`GotoXY (<position>, <string>);`

It is worth recalling that the text screen has 25 lines of 80 character positions in each. The numbering starts from the upper-left corner of the screen, which corresponds to coordinates 1.1. Moving the cursor is used to display a message at the arbitrary part of the screen using the *Write* command. Change the color of characters: `TextColor(<colour>);`

The color of the symbols is denoted by an integer *C* in [0..15]. After executing this procedure, all characters are displayed in the specified color until it is changed again. The default color is 7 (gray).

Change the background color: `TextBackGround (<color>);`

For the background colors with numbers 0..7 are used. The background color clears the screen.

For example, if you run `TextBackGround (1)` and `ClrScr`, the screen will be cleared in blue. The default color is 0 (black).

In order to limit the screen area used for input and output, you can use the procedure for defining a text box: `Window (x1, y1, x2, y2);`

After completing this procedure, all input and output is performed only within the specified window.

The module `Crt` has a delay function for a given number of milliseconds: `Delay(n);`

**Example 1.** Build a Christmas tree on the screen.

```
uses crt;
var x,y,l,j:integer;
begin
 x:=40;
 y:=5;
 k:=0;
 TextColor(2);
 for i:=1 to 5 do
 begin
 for j:=x-k to x+k do
 begin
 gotoxy(i,y);
 write('*');
 end;
 k:=k+2;
 end;
```

```

 y:=y+1;
 end;
 TextColor(6);
 for j:=y to y+4 do
 begin
 gotoxy(x,j);
 write('*');
 end;
 end.

```

### Tasks for independent work

1. Write procedure for constructing a rectangle with given coordinates of opposite angles and color in text mode. Using it, display several rectangles on the given coordinates and color.
2. Add to the result of Example 1 several dancers dancing around the Christmas tree, drawn with the symbols «O», «<», «>», «\|/».

### 4.2 Graphical display mode

The **Graph** module is designed for graphical operation. When using graphics mode, you must:

1. connect the Graph module (uses Graph;)
2. Declare the graphical constants gd, gm, which determine the type of graphics adapter and mode
3. initialize the graphical mode: initgraph (gd, gm, 'path to the BGI folder');

```

uses graph;
var gt, gm: integer;
begin
 initgraph(gd, gm, ' ');
 {graphic procedures}
 readln;
 closegraph;
end.

```

The screen size in the graphical mode is 1024 x 768 pixels (X, Y) - the dot on the graphical screen (1 <= X <= 1024- horizontally, 1 <= Y <= 768 - for Free Pascal)

### Main graphical procedures:

|                                   |                                                                                         |
|-----------------------------------|-----------------------------------------------------------------------------------------|
| MoveTo (x, y)<br>MoveRel (Dx, Dy) | Set the graphic cursor to the specified position (from given increments of coordinates) |
|-----------------------------------|-----------------------------------------------------------------------------------------|



|                                        |                                                                                       |
|----------------------------------------|---------------------------------------------------------------------------------------|
| Arc (X, Y, A, B, R)                    | drawing an arc of a circle                                                            |
| Circle (X, Y, Radius)                  | tracing a circle                                                                      |
| Ellipse (X, Y, A, B, Rx, Ry)           | drawing an ellipse (arc)                                                              |
| FillEllipse (X, Y, Rx, Ry)             | drawing a painted ellipse                                                             |
| Sector (X, Y, A, B, Rx, Ry)            | drawing of the painted sector of the ellipse                                          |
| Line (x1, y1, x2, y2)                  | drawing a line with given coordinates                                                 |
| LineTo (x, y)                          | Drawing a line from the current position of the graphic cursor to the specified point |
| Rectangle (x1, y1, x2, y2)             | drawing an unpainted rectangle with given coordinates of opposite angles              |
| Bar (x1, y1, x2, y2)                   | shaded rectangle                                                                      |
| Bar3D (x1, y1, x2, y2, L, S)           | a three-dimensional parallelepiped S is a sign of the mapping of the upper plane      |
| ClearDevice                            | cleaning the graphic screen                                                           |
| FloodFill (x, y, c)                    | Paint the bounded area using the current placeholder                                  |
| OutText (String)                       | Displays text at the current cursor position                                          |
| OutTextXY (X, Y, String)               | Outputting a string at a point (X, Y)                                                 |
| PutPixel (X, Y, C)                     | The output of the point X, Y by the color C                                           |
| SetBkColor (C)                         | Sets the current background color                                                     |
| SetColor (C)                           | Sets the current color of the drawing                                                 |
| SetFillStyle (шаблон, цвет)            | Sets the type of shading and its color                                                |
| SetLineStyle (type, colour, thickness) | Sets the thickness and type of the line                                               |
| SetTextStyle (font, type, size)        | Sets the font, type and size of the character                                         |

### Colour constants:

|            |    |               |
|------------|----|---------------|
| Black      | 0  | (black)       |
| Blue       | 1  | (blue)        |
| Green      | 2  | (green)       |
| Cyan       | 3  | (cyan)        |
| Red        | 4  | (red)         |
| Magenta    | 5  | (magenta)     |
| Brown      | 6  | (brown)       |
| LightGray  | 7  | (light gray)  |
| DarkGray   | 8  | (dark gray)   |
| LightBlue  | 9  | (light blue)  |
| LightGreen | 10 | (light green) |

|              |     |                 |
|--------------|-----|-----------------|
| LightCyan    | 11  | (light cyan)    |
| LightRed     | 12  | (light red)     |
| LightMagenta | 13  | (light magenta) |
| Yellow       | 14  | (yellow)        |
| White        | 15  | (white)         |
| Blink        | 128 | (blink)         |

Setting the type of filling: `SetFillStyle (<template>, <colour>);`

The template parameter specifies the type of hatching and can use predefined constants:

| Constant       | Meaning | Filling                        |
|----------------|---------|--------------------------------|
| EmptyFill      | 0       | Background filling             |
| SolidFill      | 1       | Solid filling                  |
| LineFill       | 2       | Horizontal lines filling       |
| ltSlashFill    | 3       | Lines tilted to the right      |
| SlashFill      | 4       | Bold lines tilted to the right |
| BkSlashFill    | 5       | Bold lines tilted to the left  |
| LtBkSlash      | 6       | Lines tilted to the left       |
| HatchFill      | 7       | Rare hatching                  |
| XhatchFill     | 8       | Frequent hatching              |
| InterleaveFill | 9       | Dashed filling                 |
| WideDotFill    | 10      | Line of frequent points        |
| CloseDotFill   | 11      | Line of rare points            |
| UserFill       | 12      | User filling                   |

To paint an arbitrary closed loop, use the following procedure:  
`FloodFill (x, y, <border color>);`

The X and Y coordinates indicate the point inside the outline, from which it will occur. The border color indicates the color by which the outline is bordered.

**Example 1.** Make an output program for the "starry sky" screen

```

Uses Crt, Graph;
Var x, y, c, n, I, gd, gm: Integer;
Begin
 Write('n='); Readln(n);
 Initgraph(gd, gm, '');
 Randomize;
 For i:=1 to n do
 Begin
 x:=Random(1200);
 y:=Random(800);
 c:=Random(15);
 PutPixel(x, y, c);
 End
 End

```

```

 End;
 Delay(5000);
 CloseGraph;
End.

```

**Note:** Instead of outputting points, you can output:

1. A circle of radius 2

```

SetColor(c);
Circle(x,y,2);

```

2. A circle of radius R

```

SetColor(c);
R:=Random(100);
Circle(x,y,R);

```

**Example 2.** Create an output program for the display of N rectangles of the same size horizontally.

```

Uses Crt,Graph;
Var x,y,a,b,n,I,c,sh,gd,gm:Integer;
Begin
 Write('n=');Readln(n);
 Initgraph(gd,gm,' ');
 x:=10;
 y:=400;
 a:=20;
 b:=200;
 c:=0;
 sh:=0;
 For i:=1 to n do
 Begin
 c:=c+1;
 If c>15 then c:=1;
 sh:=sh+1;
 If sh>12 then sh:=1;
 SetColor(c);
 Rectangle(x,y,x+a,y-b);
 SetFillStyle(sh,c);
 FloodFill(x+2,y-2,c);
 x:=x+25;
 End;
 Delay(5000);
 Closegraph;
End.

```

**Example 3.** Issue N shaded squares on the diagonal of the screen

```

Uses Crt,Graph;
Var x,y,n,I,c,sh,gd,gm:Integer;
Begin
 Write('n=');

```

```

Readln(n);
Initgraph(gd, gm, ' ');
x:=10;
y:=10;
c:=0;
sh:=0;
For i:=1 to n do
 Begin
 c:=c+1;
 If c>15 then c:=1;
 sh:=sh+1;
 If sh>12 then sh:=1;
 SetColor(c);
 Rectangle(x, y, x+20, y+20);
 SetFillStyle(sh, c);
 FloodFill(x+2, y-2, c);
 x:=x+25;
 y:=y+25;
 End;
 Delay(5000);
 Closegraph;
End.

```

**Example 4.** Display N enclosed Concentric Circles.

```

Uses Crt, Graph;
Var x, y, n, i, c, sh, r, gd, gm: Integer;
Begin
 Write('n=');
 Readln(n);
 Initgraph(gd, gm, ' ');
 x:=600;
 y:=400;
 r:=5;
 For i:=1 to n do
 Begin
 c:=c+1;
 If c>15 then c:=1;
 SetColor(c);
 Circle(x, y, r);
 r:=r+10;
 End;
 Delay(5000);
 Closegraph;
 End.

```

Procedures are used to obtain the effect of animation.  
 Memorizing the screen area:

```
GetImage(x1,y1,x2,y2,<buffer>);
```

The procedure copies the contents of the rectangle with the given coordinates from the video memory to the buffer (in an array or dynamically allocated memory area). The required size of this area (array) is defined by the function: `ImageSize(x1,y1,x2,y2);`

You can call this image using the procedure:

```
PutImage(X,Y,<buffer>,<mask>);
```

Here X and Y are the coordinates of the upper corner of the screen, from which the image from the buffer will be displayed. The parameter *<mask>* specifies the logical operation that is performed on the bit representations of the existing and superimposed images:

| Constant  | Meaning | Operation |
|-----------|---------|-----------|
| NormalPut | 0       | Mov       |
| CopyPut   | 0       | Mov       |
| XorPut    | 1       | Xor       |
| OrPut     | 2       | Or        |
| AndPut    | 3       | And       |
| NotPut    | 4       | Not       |

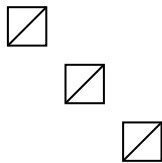
### Example 5.

```
uses graph;
const
z:FillPatternType=(205,205,255,255,215,215,250,250);
var gd,gm:integer;
 size:word;
 P:pointer;
begin
 gd:=Detect;
 InitGraph(gd,gm,'');
 SetColor(10);
 SetFillPattern(z,12);
 Bar(100,100,500,200);
 Rectangle(120,120,480,180);
 Circle(180,250,50);
 Circle(420,250,50);
 Size:=ImageSize(100,180,200,250);
 GetMem(P,Size);
 GetImage(100,180,200,250,P^);
 Readln;
 PutImage(250,250,P^,NormalPut);
 Readln;
 Readln;
 CloseGraph;
end.
```

To store the stored part of the picture, dynamic memory allocated by the procedure `GetMem` was used.

### Tasks for independent work

1. Display on the screen  $n$  circles of arbitrary radius, colors at the points of the screen selected randomly.
2. Create an output program for  $n$  enclosed squares of arbitrary color and size.
3. Display  $n$  rectangles of the same size vertically.
4. Get on the screen a drawing ( of  $n$  objects):



5. Make an output program of a chess field on the screen.

#### 4.2.1 Text output in graphical mode

The `GRAPH` module of the Pascal system allows you to display text in graphical mode. To do this, use the procedure: `OutText (<text>);` or `OutTextXY (x, y, <text>);`. It is usually used instead of the `Write (WriteLn)` procedure.

The color of the output text is set by the procedure `SetColor`. In addition, you can change other text options - the ability to select the font and the direction of the output text:

| Const                      | Meaning | Remark             |
|----------------------------|---------|--------------------|
| <code>DefaultFont</code>   | 0       | 8x8 raster         |
| <code>TriplexFont</code>   | 1       | vector             |
| <code>SmallFont</code>     | 2       | vector             |
| <code>SansSerifFont</code> | 3       | vector             |
| <code>GothicFont</code>    | 4       | vector             |
| <code>HorizDir</code>      | 0       | from left to right |
| <code>VertDir</code>       | 1       | down up            |
| <code>UserCharSize</code>  | 0       | user size          |

Procedure `SetTextStyle (<font>, <direction>, <size>);` sets font type, text direction (horizontal or vertical), and character size.

The procedure `SetTextJustify (<Hor.Grid>, <Reverse.exe>);` horizontal and vertical alignment for text:

| Constant   | Meaning | Note               |
|------------|---------|--------------------|
| LeftText   | 0       | left               |
| CenterText | 1       | centered           |
| RightText  | 2       | on the right       |
| BottomText | 0       | on the bottom line |
| TopText    | 2       | on the top line    |

The following program sets the handwritten font and displays an inscription «Hello, boys and girls!».

```

program NewText;
uses Graph;
var
 gd, gm: integer;
 TestFont: integer;
begin
 TestFont := InstallUserFont('SCRI');
 gd := Detect;
 InitGraph(gd, gm, '');
 SetTextStyle(TestFont, HorizDir, 5);
 OutTextXY(120, 180, 'Hi, boys and girls!');
 Readln;
 CloseGraph;
end.

```

### Tasks for independent work:

1. Write a function for entering a line in graphical mode in the specified font. Provide the coordinates on the screen, color and font.
2. Write a procedure for displaying the specified text with a shadow. You can specify the coordinates on the screen, color and font.
3. Display the running line in large letters in graphical mode.

### 4.2.2 Creating function graph

#### Construction of graphs of functions defined in Cartesian coordinates

To construct a function graph, we introduce the following notation:  $x_0, y_0$  are the coordinates of the origin of the Cartesian system,  $x, y$  are the coordinates of the point on the Cartesian plane,  $x_1, y_1$  are the screen coordinates of the point,  $a, b$  is the minimum and maximum abscissa ( $a \leq x \leq b$ ),  $c, d$  are the minimum and maximum values of the function ( $c \leq y \leq d$ ),  $L$  is the scaling factor, which is chosen as follows:

$$L_x = \frac{|ab|_{\text{экp}}}{|ab|_{\text{дек}}}, L_y = \frac{|cd|_{\text{экp}}}{|cd|_{\text{дек}}}, \text{ where } |ab|_{\text{экp}}, |ab|_{\text{дек}} \text{ is the length of the corresponding}$$

segment in the screen (Cartesian) coordinates,  $h = x_{i+1} - x_i$  increment of argument.

### Algorithm for creating a function graph

Set the cursor to the starting point of the curve  $M(a, f(a))$ , then we perform the same actions in the loop for  $x$  from  $a$  to  $b$  through the step  $h$ :

1. calculate the value of the function  $y = f(x)$ ,
2. We translate Cartesian coordinates  $x, y$  into screen coordinates by the formulas:  
 $x1 = x0 + x * Lx, y1 = y0 - y * Ly,$
3. connect the previous point with the received  $(x1, y1)$ .

The program for displaying the graph of the function:

```
uses crt, graph;
const gt:integer=detect;
 gm:integer=0;
function f(t:real):real;
begin
 f:=(expression);
end;
var x,y,h,a,b,c,d:real;
 lx,ly,x0,y0,n,x1,y1:integer;
begin
 write('xmin,xmax:');
 readln(a,b);
 write('Lx,Ly=');
 readln(lx,ly);
 writeln('x0,y0=');
 readln(x0,y0);
 write('h,n=');
 readln(h,n);
 initgraph(gt,gm,'');
 setcolor(n);
 line(x0+trunc(a*lx)-10,y0,x0+trunc(b*lx)+10,y0);
 line(x0+trunc(b*lx)+10,y0,x0+trunc(b*lx)+7,y0-3);
 line(x0+trunc(b*lx)+10,y0,x0+trunc(b*lx)+7,y0+3);
 x:=a; c:=f(x); d:=f(x);
 while x <=b do
 begin
 x:=x+h;
 if f(x)<c then c:=f(x);
 if f(x)>d then d:=f(x);
 end;
 line(x0,y0-trunc(c*ly)+10,x0,y0-trunc(d*ly)-10);
```



```

line(x0,y0-trunc(d*ly)-10,x0+3,y0-trunc(d*ly)-7);
line(x0,y0-trunc(d*ly)-10,x0-3,y0-trunc(d*ly)-7);
x:=a; moveto(x0+trunc(x*lx),y0-trunc(f(x)*ly));
while x<=b do
 begin
 x:=x+h;
 y:=f(x);
 x1:=x0+trunc(x*lx);
 y1:=y0-trunc(y*ly);
 lineto(x1,y1);
 end;
closegraph;
end.

```

*Remark:* The calculation of the scaling factors  $L_x$ ,  $L_y$  can be calculated from formulas:

$L_x = \text{trunc}(600/\text{abs}(b-a))$ ;  $L_y = \text{trunc}(400/\text{abs}(d-c))$ ;

You can also choose as a single scaling factor the value  $L = \min(L_x, L_y)$ ;

### Construction of graphs of functions given in parametric form

Let the function be given by parametric equations:  $\begin{cases} x = \varphi(t) \\ y = \psi(t) \end{cases}$

where  $t$  is parameter whose boundaries change  $a \leq t \leq b$ .

The program for constructing a function graph in the case of a parametric task differs from the one considered earlier only in that the variable  $t$  is used as the loop parameter, and the values of the variables  $x$  and  $y$  are calculated from the formulas given by the condition of the task.

```

uses crt, graph;
const grt:integer=detect;
 grm:integer=0;
function f1(s:real):real;
begin
 f1:=(expression 1)
end;
function f2(s:real):real;
begin
 f2:=(expression 2)
end;
var x,y,h,a,b,c,d,t,xmin,xmax,ymin,ymax:real;
 Lx,Ly,x0,y0,n,x1,y1:integer;
 Grt,grm:integer;
begin
 write('tmin,tmax:');
 readln(a,b);

```

```

 write('x0,y0,h,n:');
 readln(x0,y0,h,n);
 initgraph(grt,grm,'');
 setcolor(n);
xmin:=f1(a);
xmax:=f1(a);
t:=a;
while t<=b do
begin
 t:=t+h;
 if f1(t)<xmin then xmin:=f1(t);
 if f1(t)>xmax then xmax:=f1(t);
end;
ymin:=f2(a); ymax:=f2(a); t:=a;
while t<=b do
begin
 t:=t+h;
 if f2(t)<ymin then ymin:=f2(t);
 if f2(t)>ymax then ymax:=f2(t);
end;
Lx:=trunc(600/abs(xmax-xmin));
Ly:=trunc(400/abs(ymax-ymin));
 {axis construction OX}
line(x0+trunc(xmin*Lx)-
10,y0,x0+trunc(xmax*Lx)+10,y0);
line(x0+trunc(xmax*Lx)+10,y0,x0+trunc(xmax*Lx)+7,y0-
3);
line(x0+trunc(xmax*Lx)+10,y0,x0+trunc(xmax*Lx)+7,y0+3
);
 {построение оси OY}
line(x0,y0-trunc(ymin*Ly)+10,x0,y0-trunc(ymax*Ly)-
10);
line(x0,y0-trunc(ymax*Ly)-10,x0+3,y0-trunc(ymax*Ly)-
7);
line(x0,y0-trunc(ymax*Ly)-10,x0-3,y0-trunc(ymax*Ly)-
7);
{creating the graph of the function }
t:=a;
moveto(x0+trunc(f1(t)*Lx),y0-trunc(f2(t)*Ly));
 while t<=b do
 begin
 t:=t+h;
 x:=f1(t);
 y:=f2(t);
 x1:=x0+trunc(x*Lx);
 y1:=y0-trunc(y*Ly);
 lineto(x1,y1);

```

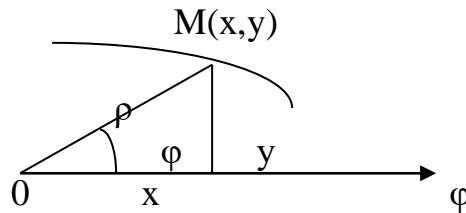
```

end;
readln
end.

```

### Construction of graphs of functions in the polar coordinate system

The equation of the curve in the polar coordinate system has the form:  
 $\rho = \rho(\varphi)$ ,  $\alpha \leq \varphi \leq \beta$  where  $\varphi$  is the polar angle,  $\rho$  is the polar radius-vector.



The formulas for the transition from polar coordinates to Cartesian coordinates:

$$\begin{cases} x = \rho * \cos(\varphi) \\ y = \rho * \sin(\varphi) \end{cases}$$

In the case of specifying a function in a polar coordinate system, the variable plays the role of an independent parameter  $\varphi$  (the cycle parameter), the values of the variable  $\rho$  are calculated by the formula indicated in the condition of the problem, and then we apply the formulas for the transition from polar coordinates to Cartesian ones for calculating the values of the variables  $x$  and  $y$ .

**Example 1.** Program for creating the hyperbola graph.

```

uses crt, graph;
const grt:integer=detect;
 grm:integer=0;
var x, y, h, a, b, ymin, ymax, lx, ly, l:real;
 x0, y0, x1, y1, grt, grm:integer;
begin
 write('a, b=');
 readln(a, b);
 write('x0, y0=');
 readln(x0, y0);
 write('lx, ly=');
 readln(lx, ly);
 initgraph(grt, grm, '');
 x:=a;
 ymin:=1/x;
 ymax:=1/x;
 while x<=b do
 begin

```

```

 x:=x+0.01;
 if abs(x+0.01)<0.00001 then x:=0.05;
 y:=1/x;
 if y<ymin then ymin:=y;
 if y>ymax then ymax:=y;
 end;
 lx:=600/abs(b-a);
 ly:=400/abs(ymax-ymin);
 if lx<ly then l:=lx else l:=ly;
 h:=0.001;
 setcolor(14);
 {construction of coordinate axes}
 line(x0+trunc(a*lx)-10,y0,x0+trunc(b*lx)+10,y0);
 line(x0+trunc(b*lx)+10,y0,x0+trunc(b*lx)+7,y0-3);
 line(x0+trunc(b*lx)+10,y0,x0+trunc(b*lx)+7,y0+3);
 line(x0,y0-trunc(ymin*ly)+10,x0,y0-trunc(ymax*ly)-
 10);
 line(x0,y0-trunc(ymax*ly)-10,x0+3,y0-trunc(ymax*ly)-
 7);
 line(x0,y0-trunc(ymax*ly)-10,x0-3,y0-trunc(ymax*ly)-
 7);
 setcolor(7);
 {creating a function graph}
 x:=a;
 while x<-0.2 do
 begin
while x<-0.2 do
 begin
 x1:=round(x0+lx*x);
 y:=1/x;
 y1:=round(y0-ly*y);
 setcolor(round(7*x));
 pieslice(x1,y1,0,360,1);
 x:=x+h;
 end;
 x:=0.2;
 while x<=b do
 begin
 x1:=round(x0+lx*x);
 y:=1/x;
 y1:=round(y0-ly*y);
 setcolor(round(4*x));
 pieslice(x1,y1,0,360,1);
 x:=x+h;
 end;
 setcolor(15);

```

```

{output of the names of axes, graph}
 settxtstyle(1,0,2);
 outtextXY(327,40,'Y');
 outtextXY(299,200,'O');
 outtextXY(505,210,'X');
 outtextXY(50,21,'Y=1/X');
 setlinestyle(dashedln,0,3);
 setcolor(11);
 rectangle(1,1,638,478);
 readln;
end.

```

Since the hyperbola is a discontinuous function, the creation is done in two cycles - separately for each branch.

### Example 2. Astroid Development Program

$$\begin{cases} x = \cos^3 t \\ y = \sin^3 t, \text{ where } 0 \leq t \leq 2\pi \end{cases}$$

```

Uses crt, graph;
Const grt:integer=detect;
 grm:integer=0;
Var x,y,t,h:real;
 xl,yl,x0,y0,l:integer;
Begin
 Initgraph(grt,grm,'');
 t:=0;
 x0:=320;
 y0:=240;
 l:=150;
 Setcolor(14);
 {construction of coordinate axes}
 Line(320,50,320,405);
 Line(315,55,320,50);
 Line(325,55,320,50);
 Line(20,240,500,240);
 Line(495,235,500,240);
 Line(495,245,500,240);
 Setcolor(7);
 {construction of function graph}
 Moveto(x0+l,y0);
 While t<6.29 do
 Begin
 x:=cos(t)*cos(t)*cos(t);
 y:=sin(t)*sin(t)*sin(t)
 {formulas for the transition from Cartesian
coordinates to screen}

```

```

 xl:=round(l*x+x0);
 yl:=round(y0-l*y);
 Setcolor(7);
 Lineto(xl,y1);
 t:=t+0.01;
 End;
SetColor (15);
{output of the names of axes, graph}
SetTextStyle(1,0,2);
OutTextXY(327,40,'Y');
OutTextXY(299,200,'0');
OutTextXY(505,210,'X');
OutTextXY(50,21,'ASTROID');
SetLineStyle(dashedln,0,3);
Setcolor(11);
Rectangle(1,1,638,478);
Readln;
End.

```

**Example 3.** The program for displaying the cardioid schedule  $R=1-\cos(f)$ , where  $0 \leq f \leq 2\pi$ .

The program for creating the graph in the polar coordinate system has the following form:

```

uses crt,graph;
const grt:integer =detect;
 grm:integer =0;
var r,f,h,a,b:real;
 x,y,l,x0,y0:integer;
begin
 write('a='); readln(a);
 write('b='); readln(b);
 write('h='); readln(h);
 write('L='); readln(L);
 write('x0='); readln(x0);
 write('y0='); readln(y0);
 initgraph (grt, grm, '') ;
{construction of the axes of coordinates}
 setcolor(2);
 line(20,y0,390,y0);
 line (390,y0,385,y0-5);
 line(390,y0,385,y0+5);
 f:=a;
 while f<=b do
 begin
 R:=1-cos{f};

```

```

{formulas for the transition from polar coordinates
to screen}
 x:=round(1*R*cos(f)+x0);
 y:=round(-1*R*sin(f)+y0);
 f:=f+0.01;
 setcolor(round(11*f));
 pieslice(x,y,1,360,1);
end;
settextstyle(triplexfont,0,3);
setcolor(14);
outtextXY(100,5,'CARDIOID');
setlinestyle(dottedln,0,3);
rectangle(0,0,610,470);
outtextXY(380,80,'R=1-COS(f)');
setcolor(15);
outtextXY(4,235,'O');
outtextXY(360,234,'P');
settextstyle(GOTHICfont,0,6);
readln;
closegraph;
end.

```

### Tasks for independent work

Construct graphs of functions given

1. in the Cartesian coordinate system:

$$1.1 \quad y = \begin{cases} 2x + 5, & x < 1 \\ x^2 + 7, & x \geq 1 \end{cases}$$

$$1.2 \quad y = \begin{cases} 1 - x^2, & x \in [-1, 1] \\ x^2 - 1, & x \notin [-1, 1] \end{cases}$$

2. in parametric form:

$$2.1 \quad \begin{cases} x = a(t - \sin t) \\ y = a(1 - \cos t) \end{cases}, \quad 0 \leq t \leq 2\pi$$

$$2.2 \quad \begin{cases} x = a \cos 2t \\ y = a \sin 3t \end{cases} \quad 0 \leq t \leq 2\pi$$

3. in polar coordinates:

$$3.1 \quad r = a \sin(nf), \quad 0 \leq f \leq 2\pi$$

$$3.2 \quad r = a(1 + \cos f), \quad 0 \leq f \leq 2\pi$$

$$3.3 \quad r = a \cos(nf), \quad 0 \leq f \leq 2\pi$$

### 4.2.3 Modeling the movement of objects

#### Algorithm for modeling the movement of an object

1. Determine the coordinates of the current position of the object
2. Determine the color of the object
3. Draw an object
4. Arrange the delay of the object on the screen
5. To paint an object with the background color {erase object}
6. Repeat the algorithm from 1 point

To solve this kind of problem, you just need to add the item "erase" the object (repeat the construction of the object at the point (x, y) with the background color or use the ClearDevice procedure).

#### Example 1. Organize the movement of the circle horizontally.

Uses Crt, Graph;

Var x, y, r, gt, gm: Integer;

Begin

InitGraph(gt, gm, '');

y:=240;

r:=50;

For x:=20 to 1200 do

Begin

SetColor(4);

Circle(x, y, r);

Delay(1000); {drawing and object delay}

SetColor(0);

Circle(x, y, r); {erasing an object with a background color}

End;

Closegraph;

End.

#### Example 2. Simulate the movement of an object along a circle.

uses crt, graph;

var x, y, r1, r2, i: integer;

gt, gm: integer;

t: real;

begin

initgraph(gt, gm, '');

r1:=200;

r2:=20;

n:=200;

t:=0;

SetColor(5);

Circle(320, 240, r1);

Repeat

t:=t+0.01;



```

 x:=trunc(320+r1*cos(t));
 y:=trunc(240-r1*sin(t));
 SetColor(4);
 Circle(x,y,r2);
 Delay (10000);
 SetColor(0);
 Circle (x,y,r2);
 end;
closegraph;
end.

```

Note: organize the painting of a small circle

**Example 3. Simulate the movement of a segment around one of the ends.**

```

uses crt,graph;
var x0,y0,r,x1,y1,c,gt,gm:integer;
 f,x,y:real;
begin
 initgraph(gt,gm,'');
 f:=0;
 x0:=640;
 y0:=480;
 r:=100;
 c:=0;
repeat
 x:=r*cos(f);
 y:=r*sin(f);
 x1:=x0+trunc(x);
 y1:=y0-trunc(y);
 c:=c+1;
 if c>15 then
 c:=1;
 setcolor(c);
 line(x0,y0,x1,y1);
 delay(5000);
 setcolor(0);
 line(x0,y0,x1,y1);
 f:=f+0.01;
until f>6.29;
closegraph;
end.

```

**Example 4. Simulate the production of a cylindrical surface by rotating a segment along two circles.**

Note: use two circles with the centers at the points (x0, y0) and (x00, y00), and use the coordinates of the point (x1, y1) moving along the circle with the coordinates of the center (x0, y0) and the point (x2, y2) moving along the circle with the coordinates of the center (x00, y00).

```

f:=0;
c:=0;
repeat
 x:=r*cos(f);
 y:=r*sin(f);
 x1:=x0+trunc(x);
 y1:=y0-trunc(y);
 x2:=x00+trunc(x);
 y2:=y00-trunc(y);
 c:=c+1;
 if c>1 then
 c:=c+1;
 setcolor(c);
 line(x1,y1,x2,y2);
 f:=f+0.01;
until f>6.29;

```

### **Tasks for independent work**

1. To organize the rotation of the segment around its center.
2. To simulate obtaining the surface of a truncated cone by rotating the ends of a segment along two circles of different radii.
3. To arrange for the production of a conic surface by rotating one end of the segment along the circumference when the second end of the segment is stationary.
4. To stimulate the rotation of the propeller, using the rotation of three segments, whose origin is in the center of the circle, and the ends of the segments rotate along the arc of the circle.

## LIST OF REFERENCES

1. *Algorythmisation and programming languages: Study-manual complex /MES RK, KazSPU after Abay; authors-compilers: O.S.Akhmetova, B.K. Tulbasova.* – Almaty: KazSPU after Abay, 2012. – 164 p.
2. Dauletkulov, A.B. *Olympiads on informatics.* – Almaty; RSPC «Daryn», 1999. – 216p.
3. Faronov, V.V. *Turbo Pascal.* – StPb.:BHV-Peterburg, 2007. – 1056p.
4. Kultin, N. *Turbo Pascal in tasks and examples.* – StPb.:BHV-Peterburg, 2000. – 256p.
5. Okulov, S.M. *Basics of programming.* – M.: UNIMEDIASTYLE, 2002. – 424p.
6. Okulov, S.M. *Programming in algorythms.* - M.:BINOM. Laboratory of knowledge, 2002. – 341p.
7. Tsyganova, A.D. *Algorythms on graphs.* Kostanay.: KSPI, 2007.- 104p.
8. Tsyganova, A.D. *Basics of programming.* – Kostanay.: KSPI, 2014. – 168p.
9. Tsyganova, A.D. *Fundamentals of Pascal programming.* - Kostanay, 2005. -136 p.
10. Tsyganova, A.D. *Mathematical bases of programming.* – Kostanay.: KSPI, 2016. – 128p.
11. Tsyganova, A.D. *Olympic informatics.* – Kostanay.: KSPI, 2013. – 179p.
12. Tsyganova, A.D. *Theoretical bases of informatics: Study manual for students of Informatics specialty.* – Kostanay.: KSPI, 2011. – 86p.
13. Tsyganova, A.D., Gridneva, V.M. *Practice of solving problems in informatics.* Kostanay.: KSPI, 2006. – 183p.