Introduction to Computer Science

Piotr Fulmański

Piotr Fulmański[1]

Faculty of Mathematics and Computer Science,
University of Łódź
Banacha 22, 90-238, Łódź
Poland

e-mail 1: fulmanp@math.uni.lodz.pl

Last modification: **2 października 2008**

Spis treści

Spis treści **3**

1 Numeral systems **5**
1.1 Numerals and their systems 5
1.2 Types of numeral systems 6
1.2.1 Unary numeral system 6
1.2.2 Abbreviated unary numeral system 6
1.2.3 Nowadays systems 6
1.3 Binary system 9
1.3.1 Conversion from base-2 (binary) into base-10 (decimal) system 9
1.3.2 Conversion from base-10 (decimal) into base-2 (binary) system 9
1.3.3 Binary arithmetic 11
1.3.4 Zapis liczby rzeczywistej w systemie dwójkowym . . . 12
1.3.5 Kod szesnastkowy 15
1.3.6 Inne pozycyjne systemy liczbowe 16
1.4 Kod BCD 20
1.5 Zadania 24

2 Algorithms and data structures **27**
2.1 Algorithm 27
2.2 Data structures 29
2.2.1 The notion of data type 30
2.2.2 Array 31
2.2.3 Dictionary 33
2.2.4 Set 33
2.2.5 Record 33
2.2.6 Klasa 35
2.2.7 Plik 35
2.3 Metody opisu algorytmów 35
2.3.1 Język naturalny 36
2.3.2 Schemat blokowy 36

2.3.3 Schemat zapisu algorytmu za pomocą pseudojęzyka . 37
2.4 Podstawowe algorytmy 44
2.4.1 Algorytmy obliczeniowe 44
2.4.2 Algorytmy sortowania 45
2.4.3 Algorytmy wyszukujące 47
2.5 Rekurencja a iteracja 49
2.6 Analiza złożoności 53
2.7 Zadania 55

Rozdział 1

Numeral systems

1.1 Numerals and their systems

A **number** is an abstract idea used in counting and measuring. A symbol or a word in natural language which represents a number is called a **numeral** and symbols used in this numeral are called **digits**. Numerals differ from numbers just as words differ from the things they refer to. The symbols „*11*", „*eleven*" and „*XI*" are different numerals but representing the same number.

In common usage the word *number* is used for both the idea and the symbol. In addition to their use in counting and measuring, numerals are often used e.g. for labels (telephone numbers), for ordering (serial numbers) or for codes (ISBNs).

To be more precise, term *number* without any adjectives is inexact, because mathematicians not defines *numbers* but *natural numbers*, *integer numbers*, etc. Each kind of numbers is defined by axioms or is derived from more (less) complicated numbers or other elementary terms like set.

Having the term *numeral* a **numeral system** concept can be introduced:

Definition 1.1. *A* **numeral system** *(or* **system of numeration***) is a framework where a set of numbers are represented by numerals in a consistent manner. It can be seen as the context that allows the numeral „*11*" to be interpreted as the binary numeral for three, the decimal numeral for eleven, or other numbers in other system.*

Ideally, a numeral system will:

- Represent a useful set of numbers (e.g. all whole numbers, integers, or real numbers).
- Give every number represented a unique representation (or at least a standard representation).
- Reflect the algebraic and arithmetic structure of the numbers.

1.2 Types of numeral systems

1.2.1 Unary numeral system

The simplest numeral system is the **unary numeral system**, in which every natural number is represented by a corresponding number of symbols. If the symbol `|` is chosen, for example, then the number *seven* would be represented by `|||||||`. Tally marks represent one such system which is still in common use.

This type of system, when the number is a sum of symbols, we can also called **additive numeral system**.

1.2.2 Abbreviated unary numeral system

The unary notation can be abbreviated by introducing different symbols for certain new values. Very commonly, these values are powers of 10; so for instance, if `|` stands for one, `-` for ten and `+` for 100, then the number 304 can be compactly represented as `+++ ||||` and number 123 as `+ -- |||` without any need for zero. This is called sign-value (sign-magnitude) notation. The ancient Egyptian system is of this type and the Roman system is a modification of this idea.

More useful still are systems which employ special abbreviations for repetitions of symbols; for example, using the first nine letters of our alphabet for these abbreviations, with `A` standing for „one occurrence", `B` „two occurrences", and so on, we could then write `C+ D|` for the number 304. The numeral system of English is of this type („three hundred [and] four"), as are those of virtually all other spoken languages, regardless of what written systems they have adopted.

1.2.3 Nowadays systems

Nowadays, the most commonly used system of numerals is known as Hindu-Arabic numerals, and two great Indian mathematicians could be given credit for developing them. Aryabhatta of Kusumapura who lived during the 5th century developed the **place value notation** and Brahmagupta a century later introduced the **symbol zero**.

To understand nowadays systems let's try to realize what is going on when WE write some numbers and when ancient Romans did this.

```
Nowadays   Romans
111=?!     III=?!
|||        |||
||1        ||1
||         ||
|10        |1
```

```
|           |
100         1
```

Consider two numbers: 111 and III. In each case we use only one symbol: „1" in the first number and „I" in the second. The number of occurrence is exactly the same. Notice that in case of Roman number each „I" mean exactly the same: value equal to 1. In case of nowadays number each „1" mean something different: value equal to 1 or 10 or 100. So the first observation is

Observation 1.1. *In nowadays system the position of digit is important.*

```
Nowadays    Romans
115=?!      IIV
|||         |||
||5         ||5
||          ||
|10         |-1
|           |
100         1
```

Now consider two other numbers: 115 and IIV. Notice different meaning of the second symbol „I" in Roman number. Meaning of each „1" symbol in 115 number stays the same as for 111 number.

Observation 1.2. *In nowadays system interpretation of each digit does not depend on context (other sorrounding symbols).*

Taking number 304 we conclude that

$$304 = 300 + 0 + 4 = \dots$$

which is equal to

$$\cdots = 10^2 \cdot 3 + 10^1 \cdot 0 + 10^0 \cdot 4.$$

We use three different digits (0,3 and 4) and the position of a digit to signify the power of 10 that the digit is to be multiplied with. Note that zero, which is not needed in the other systems, is of crucial importance here, in order to be able to „skip" a power. Because we use 10 as a base (we calculate powers of 10) such a systems is called **base-10 systems**. The Hindu-Arabic numeral system, borrowed from India, is a positional base-10 system; it is used today throughout the world.

Generally all n-digit base-10 numbers can be express as

$$d_{n-1}...d_1d_0,$$

where each d_i, $i \in n-1, \dots, 1, 0$ is a base-10 system digit so belongs to set $\{0, 1, ..., 9\}$. Value of such a number is equal to

$$v = d_{n-1} \cdot 10^{n-1} + \cdots + d_1 \cdot 10^1 + d_0 \cdot 10^0.$$

Uwaga 1.1. *Zupełnie inna sytuacja występuje w zapisie liczby w systemie rzymskim. Przypomnijmy, że w systemie tym kolejne liczby od* 1 *do* 9 *mają postać*

I, II, III, IV, V, VI, VII, VIII, IX

Widać, że w takim zapisie pozycja „cyfry" – o ile w ogóle można mówić w tym wypadku o cyfrze – nie jest związana z wyznaczaniem jej wartości, lecz istotna jest postać całej liczby. Taki system zapisu nazywamy ***addytywnym systemem liczbowym****.*

Example of base-10 system leads us to general definition of place-value numeral system of any base.

Definition 1.2. *A* **positional** *or* **place-value numeral system** *is a pair* (b, D)*, where* b *is a natural number called* **base** *or* **radix** *of that numeral system,* D *is a finite set of* b *symbols* $\{s_0, s_1, \ldots, s_{b-1}\}$*, called* **digits**[1]*. Such a system is called* **base-**b **system***.*

In such a system each number is represented as a sequence of digits and its value depends both on digits and position of digit it that sequence. Value of sequence

$$d_k d_{k-1} \ldots d_1 d_0$$

is equal to

$$v = d_k b^k + d_{k-1} b^{k-1} + \ldots + d_1 b + d_0 \tag{1.1}$$

where $d_0, \ldots, d_k \in D$*.*

If $b = 10$ *the system is called decimal, if* $b = 2$ *– binary, if* $b = 8$ *– octal, etc.*

In the other words a place-value numeral system is a system in which each position is related to the next by a constant multiplier, a common ratio, called the base of that numeral system. Each position may be represented by a unique symbol or by a limited set of symbols. The resultant value of each position is the value of its symbol or symbols multiplied by a power of the base. The total value of a positional number is the total of the resultant values of all positions.

Using at one time several different positional systems, we have to write numbers with an information about the base in order to operate right value. Consider followin examples:

- 11_{10} – number of decimal value 11 written in the base-10 (decimal) system,
- 11_2 – number of decimal value 3 written in the base-2 (binary) system,
- 11_5 – number of decimal value 6 written in the base-5 system,
- 11_{25} – number of decimal value 26 written in the base-25 system.

[1]Typically the set D consist of symbols $0, 1, \ldots, 8, 9$ and if it is needed of letters: $A, B, \ldots$.

1.3 Binary system

Because nowadays computers are build base on digital circuit ruled by two-elementary Boolean algebra (see **??**), that is why in computer science binary system is of special importance. Now this system, conversion from decimal to binary and from binary to decimal and arithmetic in this system will be described.

1.3.1 Conversion from base-2 (binary) into base-10 (decimal) system

Based on positional numeral system definition (definition 1.2) we conclude that the base of binary system is number 2, and set of digits D consist of two symbols, by definition denoted as 0 and 1.

Taking as an example number x as follow

$$x = 1011110110_{(2)} \tag{1.2}$$

and using (1.1), we obtain

$$\begin{aligned} v &= 1 \cdot 2^9 + 0 \cdot 2^8 + 1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + \\ &+ 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0. \end{aligned}$$

Repleacing powers of 2 by coresponding values expressed in decimal system, we obtain

$$\begin{aligned} v &= 1 \cdot 512 + 0 \cdot 256 + 1 \cdot 128 + 1 \cdot 64 + 1 \cdot 32 + \\ &+ 1 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 1 \cdot 2 + 0 \cdot 1 = 758_{(10)}. \end{aligned}$$

It can be seen, that conversion from binary to decimal come down to use formula (1.1) and calculate value described by it. The only thing to remember is that all calculations have to be done in decimal system.

1.3.2 Conversion from base-10 (decimal) into base-2 (binary) system

While conversion from binary system to decimal system comes down to addition and multiplication, an opposite conversion require more attention. If some decimal number x is given, than conversion follows such an algorithm:

1. Begin.
2. Let $v = x$.
3. Divide v by 2 and denote it as r.

4. If the result r is an integer, write 0.
5. If the result r is not an integer, write 1.
6. Take as v integer part of result r.
7. If v is not equal to 0, go to step 3.
8. If v is equal to 0, go to step 9.
9. End.

In this way a sequence of 0's and 1's is obtained. This sequence write down from the right to the left (in order we obtain each digit) is a binary number we are looking for.

To illustrate this algorithm, consider following example. Say that we are looking for binary representation ofdecimal number 758. So the first step is to divide number 758 by 2

```
758 | 2 * 379 + 0 (v=379, r=0)
```

Now the number 379 is divided by 2

```
379 | 2 * 189 + 1 (v=189, r=0)
```

and by analogy we obtain Obtained in this way right column is a binary

```
189 | 2 * 94 + 1
 94 | 2 * 47 + 0
 47 | 2 * 23 + 1
 23 | 2 * 11 + 1
 11 | 2 * 5 + 1
  5 | 2 * 2 + 1
  2 | 2 * 1 + 0
  1 | 2 * 0 + 1
  0 |
```

representation of decimal number 758. This column write down from the right to the left (in order we obtain each digit) is a binary number we are looking for.

$$1011110110_{(2)}.$$

As could be notice result is identical to (1.2) which is, as we checked above, binary representation of (decimal) value 758.

Below there are some more examples (example 1.1).

Example 1.1. Conversion from base-10 (decimal) into base-2 (binary) system.

158	2 * 79 + 0	108	2 * 54 + 0	59	2 * 29 + 1
79	2 * 39 + 1	54	2 * 27 + 0	29	2 * 14 + 1
39	2 * 19 + 1	27	2 * 13 + 1	14	2 * 7 + 0
19	2 * 9 + 1	13	2 * 6 + 1	7	2 * 3 + 1
9	2 * 4 + 1	6	2 * 3 + 0	3	2 * 1 + 1
4	2 * 2 + 0	3	2 * 1 + 1	1	2 * 0 + 1
2	2 * 1 + 0	1	2 * 0 + 1	0	end
1	2 * 0 + 1	0	end		
0	end				

The results are:
$158_{(10)} \rightarrow 10011110_{(2)}$, $108_{(10)} \rightarrow 1101100_{(2)}$, $59_{(10)} \rightarrow 111011_{(2)}$

1.3.3 Binary arithmetic

Komputery pracujące w oparciu o układy cyfrowe dokonują większość obliczeń na liczbach w systemie dwójkowych. Pokażemy teraz sposób wykonywania podstawowych działań na liczbach zapisanych w ten sposób.

Dodawanie jest realizowane podobnie jak dla systemu dziesiętnego. Należy jedynie pamiętać, że

$$1_{(2)} + 1_{(2)} = 10_{(2)}.$$

Wynika to z faktu, iż w systemie dwójkowym nie ma cyfry reprezentującej liczbę 2, $1+1$ w tym systemie daje w wyniku 0 na pewnej pozycji, a jedność jest przenoszona na następną pozycję w liczbie. Jest to podoba sytuacja, jak w przypadku dodawania $1 + 9$ w systemie dziesiętnym: otrzymujemy w wyniku 0, a jedność jest przenoszona na następną pozycję. Przypatrzmy się następującym działaniom w przykładzie 1.2.

Example 1.2. Dodawanie w systemie dwójkowym.

```
   11
  00111000     01000001
+ 00010001   + 00010100
----------   ----------
  01001001     01010101
```

Odejmowanie, podobnie jak dodawanie, wykonywane jest według zasady identycznej, jak w systemie dziesiętnym, przy czym w przypadku odejmowania $0 - 1$ w systemie dwójkowym musimy dokonać zapożyczenia 1 na następnej pozycji liczby. Obliczenia zawiera przykład 1.3

Example 1.3. Odejmowanie w systemie dwójkowym.

```
  00111001      00101101
- 00001101    - 00010001
----------    ----------
  00101100      00011100
```

Mnożenie jest wykonywane analogicznie jak mnożenie w systemie dziesiętnym, ilustruje to przykład 1.4.

Example 1.4. Mnożenie w systemie dwójkowym.

```
      1111              10001
     *  101           *    11
-------------     ---------------
       1111             10001
      0000          +   10001
+     1111          ---------------
-------------           110011
    1001011
```

Dzielenie, podobnie jak mnożenie, wykonujemy tak samo jak w przypadku dzielenia w systemie dziesiętnym (przykład 1.5).

Example 1.5. Dzielenie w systemie dwójkowym.

```
    110                       1011
  -----                     -------
  10010:11=00000110         1111001:1011=1011
-  11                     - 1011
-------                   ---------
    11                       10000
-   11                     -  1011
-------                   ---------
     0                          1011
                          -     1011
                          ---------
                                  0
```

1.3.4 Zapis liczby rzeczywistej w systemie dwójkowym

Do tej pory zamiana z i na system dwójkowy była ograniczona wyłącznie do liczb całkowitych. Pozostaje zatem do wyjaśnienia, w jaki sposób należy reprezentować liczbę rzeczywistą w systemie dwójkowym[2]. W celu

[2]Rozważania prowadzone w tym punkcie mają charakter informacyjny, gdyż będą potrzebne w następnym rozdziale. Jednak nie należy postrzegać prezentowanego zapisu liczby rzeczywistej jako tego, który jest stosowany w komputerze w sposób bezpośredni. W następnym rozdziale będzie to wytłumaczone obszerniej.

lepszego zrozumienia zapisu liczb rzeczywistych w systemie dwójkowym, przypomnijmy to, co było omawiane na początku podrozdziału **??**. Zauważmy, że dla liczby całkowitej było powiedziane, że dopiero pozycja cyfry wraz z jej wartością niesie pełną informację. Podobnie ma się sprawa z częścią ułamkową. Jeśli mamy liczbę w systemie dziesiętnym postaci:

$$0, c_{-1}c_{-2}...c_{-n},$$

gdzie $c_{-1}, c_{-2}, \ldots, c_{-n} \in \{0, \ldots, 9\}$ są cyframi, to wartość tej liczby jest wyliczana za pomocą następującego wzoru:

$$w = c_{-1}10^{-1} + c_{-2}10^{-2} + \cdots + c_{-n}10^{-n}.$$

Jeżeli teraz nasze rozważania przeniesiemy do systemu dwójkowego, to wartość będzie wyliczana na podstawie podobnego wzoru (zmieni się jedynie podstawa), zatem:

$$w = c_{-1}2^{-1} + c_{-2}2^{-2} + \cdots + c_{-n}2^{-n}.$$

Oczywiście w tym wypadku cyfry są elementami zbioru $\{0, 1\}$.

Stosując powyższy wzór jesteśmy w stanie bezpośrednio zamienić ułamek zapisany w systemie dwójkowym na system dziesiętny. Rozważmy poniższy przykład:

Example 1.6. Zamiana ułamka dwójkowego na dziesiętny.

$$0, 11_{(2)} = 2^{-1} \cdot 1 + 2^{-2} \cdot 1 = 0, 5 + 0, 25 = 0, 75_{(10)}$$

Do wyjaśnienia pozostaje sposób zamiany w drugą stronę, czyli jak zamieniać ułamek dziesiętny na dwójkowy. Ograniczymy się w tym miejscu wyłącznie do ułamków dodatnich. Schemat postępowania jest podobny do przedstawionych wcześniej. Mamy następujący algorytm (ułamek zapisany dziesiętnie oznaczmy przez x):

1. Start.
2. Niech $w = x$.
3. Mnożymy w przez 2.
4. Jeśli wynikiem operacji mnożenia jest liczba większa od jedności, zapisujemy na boku 1.
5. Jeśli wynikiem operacji mnożenia jest liczba mniejsza od jedności, zapisujemy na boku 0.
6. Ułamkową część wyniku – po odrzuceniu ewentualnej części całkowitej – zapisujemy jako w.
7. Jeśli w jest różne od 0, przechodzimy z powrotem do kroku 2.
8. Jeśli w jest równe 0, kończymy procedurę.

9. Koniec.

Następnie otrzymane zera i jedynki zapisywane w kolejności otrzymywania od lewej do prawej stanowią ułamek zapisany w systemie dwójkowym. Zilustrujmy powyższy algorytm przykładem.

Example 1.7. Zamiana ułamka dziesiętnego na ułamek dwójkowy.

0,75 => ?

```
                  liczba dwójkowa
                  |
0,75 | 2 * 0,75 = 1,5 <- część ułamkową przepisujemy
0,5  | 2 * 0,5  = 1,0
0,0  | koniec
```

Jak widać, w przykładzie 1.7, otrzymaliśmy wartość $0,11$, która odpowiada tej z przykładu 1.6, zatem jednocześnie dokonaliśmy sprawdzenia.

Przedstawmy jeszcze jeden przykład dla liczby $0,40625_{(10)}$, otrzymujemy:

0,40625	2 * 0,40625 = 0,8125
0,8125	2 * 0,8125 = 1,625
0,625	2 * 0,625 = 1,25
0,25	2 * 0,25 = 0,5
0,5	2 * 0,5 = 1,0
0,0	koniec

Zatem $0,40625_{(10)} = 0,01101_{(2)}$, sprawdźmy:

$$2^{-1} \cdot 0 + 2^{-2} \cdot 1 + 2^{-3} \cdot 1 + 2^{-4} \cdot 0 + 2^{-5} \cdot 1 = 0,25 + 0,125 + 0,03125 = 0,40625.$$

Przejdźmy teraz do zamiany liczb rzeczywistych większych od jednego. W takim przypadku stosujemy dla części całkowitej poznany wcześniej proces, a dla części ułamkowej omówiony powyżej.

Example 1.8. Zamiana liczby rzeczywistej dziesiętnej na postać dwójkową.

9,25 => ?

9	2 * 4 + 1	0,25	2 * 0,25 = 0,5
4	2 * 2 + 0	0,5	2 * 0,5 = 1,0
2	2 * 1 + 0	0,0	koniec
1	2 * 0 + 1		
0	koniec		

Zatem otrzymujemy ostatecznie $9,25_{(10)} = 1001,01_{(2)}$.

Na koniec zauważmy, że nie każdą liczbę, którą da się zapisać w postaci ułamka dziesiętnego, da się dobrze przedstawić w postaci ułamka dwójkowego, co ilustruje przykład 1.9.

Example 1.9. Zamiana ułamka dziesiętnego na ułamek dwójkowy okresowy.

0,3	2 * 0,3 = 0,6
0,6	2 * 0,6 = 1,2
0,2	2 * 0,2 = 0,4
0,4	2 * 0,4 = 0,8
0,8	2 * 0,8 = 1,6
0,6	...

Zatem, jak widać, ułamek może mieć „dobrą" postać dziesiętną liczby 0,3, zaś „złą" – dwójkową 0,01001... W powyższym przykładzie ułamek dwójkowy jest okresowy, zatem w którymś momencie musimy zaprzestać procedury, gdyż inaczej prowadzilibyśmy ją w nieskończoność. Jeśli jednak zaprzestaniemy, to będzie to niedokładna reprezentacja wyjściowego ułamka. Dla przykładu ograniczmy się do pięciu cyfr po przecinku, otrzymamy wtedy:

$$0{,}01001_{(2)} = 0{,}25 + 0{,}03125 = 0{,}28125_{(10)}.$$

Jak widać błąd takiego zaokrąglenie wcale nie jest mały, jednak pamiętajmy, że nie jest to „wina" systemu dwójkowego, lecz samego procesu dzielenia w ogóle. Zauważmy, że rozwinięcie dziesiętne ułamka zwykłego 1/3 również będzie przybliżone, jeśli chcemy je przechować na skończonym nośniku danych.

1.3.5 Kod szesnastkowy

Wadą systemu dwójkowego jest „rozwlekłość" zapisywanych w ten sposób liczb. Dla człowieka bywa trudne zapamiętanie ciągu zer i jedynek. Stąd też często w praktyce informatyka spotykamy się z zapisem liczby w **systemie szesnastkowym** lub **kodzie szesnastkowym**. Z definicji pozycyjnego systemu liczbowego (zob. podrozdział **??**) wynika, że w systemie szesnastkowym musimy mieć szesnaście cyfr, ale jak sobie poradzić w sytuacji, gdy mamy do dyspozycji cyfry arabskie $(0, 1, \ldots, 9)$. Problem ten rozwiązano poprzez nazwanie kolejnych cyfr tego systemu, poczynając od dziesięciu, kolejnymi literami alfabetu, zatem $A_{(16)} = 10_{(10)}, B_{(16)} = 11_{(10)}$ itd., dopuszcza się również stosowanie małych liter zamiast dużych.

Można oczywiście zapytać, czemu nie stosujemy w informatyce systemu osiemnastkowego, przyjrzyjmy się poniższej tabelce, a wszystko się wyjaśni.

Zapis dwójkowy	Zapis szesnastkowy	Zapis dziesiętny
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Widać z powyższego zestawienia, że jedna cyfra kodu szesnastkowego odpowiada dokładnie czterocyfrowej liczbie systemu dwójkowego. Ponieważ bajt informacji (zob. dodatek **??**) składa się z 8 bitów, zatem każdy bajt danych daje się w sposób jednoznaczny przedstawić za pomocą dwu cyfr kodu szesnastkowego, co upraszcza kwestię konwersji – nie wymaga ona w praktyce obliczeń, a jedynie podstawienia. Oto przykład

$$00011101_{(2)} = 1D_{(16)},\ 1011111_{(2)} = BF_{(16)},\ 11100011_{(2)} = E3_{(16)}.$$

1.3.6 Inne pozycyjne systemy liczbowe

Punkt ten należy traktować jako informację poszerzającą wiedzę ogólną, gdyż inne systemy liczbowe niż dwójkowy i szesnastkowy praktycznie nie są wykorzystywane w informatyce lub ich obecność jest marginalna. Dla przykładu niekiedy można spotkać zapis liczby w systemie ósemkowym.

Korzystając z definicji pozycyjnego systemu liczbowego, zapiszemy tą samą liczbę w systemach pozycyjnych o różnych podstawach. Przykład 1.10 zawiera ilustrację tego procesu.

Example 1.10. Liczba 758 zapisana w pozycyjnych systemach liczbowych o różnych podstawach.

$$23312_{(4)} = 2 \cdot 4^4 + 3 \cdot 4^3 + 3 \cdot 4^2 + 1 \cdot 4^1 + 2 \cdot 4^0 = 758_{(10)}$$
$$1366_{(8)} = 1 \cdot 8^3 + 3 \cdot 8^2 + 6 \cdot 8^1 + 6 \cdot 8^0 = 758_{(10)}$$
$$2F6_{(16)} = 2 \cdot 16^2 + 15 \cdot 16^1 + 6 \cdot 16^0 = 758_{(10)}$$

Jedną z możliwych metod przejścia od zapisu w systemie o podstawie p_1 do systemu o podstawie p_2 jest zamiana zapisu liczby na system dziesiętny, a dopiero potem na system o podstawie p_2. Metoda ta wydaje się naturalna, gdyż ludzie najsprawniej operują systemem dziesiętnym. Ilustruje to poniższy przykład

Example 1.11. Zamiana liczby z systemu 25 na system 11 poprzez system dziesiątkowy.

Wpierw zamieniamy liczbę z systemu o podstawie 25 na liczbę w systemie dziesiątkowym

$$2K_{(25)} = 2 \cdot 25^1 + K \cdot 25^0 = 2 \cdot 25 + 20 \cdot 1 = 70_{(10)}$$

następnie z systemu dziesiątkowego na jedenastkowy

```
70 | 11 * 6 + 4
 6 | 11 * 0 + 6
```

Ostatecznie więc otrzymujemy

$$2K_{(25)} = 64_{(11)}$$

Zauważmy jednocześnie, że w powyższym przykładzie, przy zapisie liczby w systemie dwudziestopiątkowym, przyjęliśmy podobną notację cyfr, jak w systemie szesnastkowym, zatem wykorzystaliśmy kolejne litery alfabetu.

W szczególnych przypadkach można proces ten jednak znacznie uprościć. Przyjrzyjmy się przykładowym liczbom w zapisie dwójkowym, czwórkowym i dziesiętnym.

Zapis dwójkowy	Zapis czwórkowy	Zapis dziesiętny
00	0	0
01	1	1
10	2	2
11	3	3

$$11011001_{(2)} \to 3121_{(4)},$$
$$3321_{(4)} \to 11111001_{(2)}.$$

Zauważmy, że zamieniając liczbę dwójkową na czwórkową, grupujemy bity po dwa, a następnie te dwójki zapisujemy za pomocą odpowiednich liczb czwórkowych (podobnie czyniliśmy to w zamianie z systemu dwójkowego na szesnastkowy). Zapisując liczbę czwórkową jako dwójkową, każdą jej cyfrę przedstawiamy za pomocą odpowiedniego dwuwyrazowego ciągu zero-jedynkowego.

```
11 01 10 01        3  3  2  1
|  |  |  |        |  |  |  |
3  1  2  1        11 11 10 01
```

Teraz zapiszmy kolejne liczby w systemach: dwójkowym, ósemkowym i dziesiętnym.

Zapis dwójkowy	Zapis ósemkowy	Zapis dziesiętny
000	0	0
001	1	1
010	2	2
011	3	3
100	4	4
101	5	5
110	6	6
111	7	7

W tym przypadku, aby dokonać przeliczenia, możemy grupować cyfry kodu dwójkowego po 3.

```
011 011 001
 |   |   |
 3   3   1
```

Otrzymujemy zatem

$$11011001_{(2)} \to 331_{(8)},$$
$$331_{(8)} \to 11011001_{(2)}.$$

Do konwersji z systemu dziesiętnego na inny pozycyjny system liczbowy, stosujemy analogiczny sposób, jak opisany w pkt **??**, z tą różnicą, że tym razem resztą z dzielenia mogą być nie tylko cyfry zero lub jeden (przykład 1.12).

Example 1.12. Przykład ilustrujący zamianę liczby 758 zapisanej dziesiętnie na system czwórkowy, ósemkowy i szesnastkowy.

758	`4 * 189 + 2`	758	`8 * 94 + 6`	758	`16 * 47 + 6`
189	`4 * 47 + 1`	94	`8 * 11 + 6`	47	`16 * 2 + 15`
47	`4 * 11 + 3`	11	`8 * 1 + 3`	2	`16 * 0 + 2`
11	`4 * 2 + 3`	1	`8 * 0 + 1`		
2	`4 * 0 + 2`				

Zatem otrzymujemy:
$\texttt{758}_{(10)} \to \texttt{23312}_{(4)}$, $\texttt{758}_{(10)} \to \texttt{1366}_{(8)}$, $\texttt{758}_{(10)} \to \texttt{2F6}_{(16)}$

Nic nie stoi na przeszkodzie, aby stosować systemy liczbowe o podstawach różnych od potęgi 2. Niech np. liczba $2K$ będzie liczbą zapisaną w systemie o podstawie 25. Zapiszemy ją teraz jako liczbę systemu o podstawie 11. Najbardziej naturalną drogą, zasugerowaną na początku tego podrozdziału jest konwersja: $(25) \to (10) \to (11)$,

$$2K_{(25)} = 2 \cdot 25^1 + K \cdot 25^0 = 2 \cdot 25 + 20 \cdot 1 = 70_{(10)}.$$

```
70 | 11 * 6 + 4
 6 | 11 * 0 + 6
```

Ostatecznie otrzymujemy, że $2K_{(25)} = 64_{(11)}$.

Innym podejściem jest pominięcie pośredniego kroku w postaci przeliczania na system dziesiętny i operowanie od razu wyłącznie na systemie źródłowym i docelowym. W trakcie konwersji z systemu o mniejszej podstawie do większej nie nastręcza to żadnych trudności. W tym wypadku ma zastosowanie wzór (**??**), jednak należy pamiętać, że wszelkie obliczenia muszą być prowadzone w arytmetyce docelowego systemu liczbowego.

Rozważmy poniższy przykład, niech będzie dana liczba $1111101_{(2)}$, jak widać zapisana w systemie dwójkowym. Załóżmy teraz, że chcemy dokonać konwersji na system szesnastkowy. Napiszmy wzór przedstawiający wartość liczby

$$w = 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0,$$

teraz wszystkie obliczenia po prawej stronie równości należy prowadzić w arytmetyce szesnastkowej. Zatem

$$w = 40 + 20 + 10 + 8 + 4 + 0 + 1 = 70 + D = 7D_{(16)},$$

co jak widać zgadza się ze zwykłą regułą prezentowaną w pkt 1.3.5.

Niestety sytuacja komplikuje się przy dokonywaniu odwrotnej konwersji, wtedy bowiem dzielimy liczbę w systemie źródłowym przez podstawę systemu docelowego, oraz musimy pamiętać o resztach, których liczba, jaką się rozważa, jest zależna od systemu docelowego. Weźmy liczbę $96_{(16)}$ i zamieńmy ją na system trójkowy. W tym celu przeprowadzamy następującą serię dzieleń

```
96 | 3 * 32 + 0
32 | 3 * 10 + 2
10 | 3 * 05 + 1
 5 | 3 * 01 + 2
 1 | 3 * 00 + 1
```

Zatem $96_{(16)} = 12120_{(3)}$. Wykonajmy sprawdzenie powyższej konwersji, zamieniając postacie w obu systemach na system dziesiętny

$$96_{(16)} = 9 \cdot 16^1 + 6 \cdot 16^0 = 144 + 6 = 150_{(10)},$$

$$12120_{(3)} = 1 \cdot 3^4 + 2 \cdot 3^3 + 1 \cdot 3^2 + 2 \cdot 3^1 + 0 = 81 + 54 + 9 + 6 + 0 = 150_{(10)}.$$

1.4 Kod BCD

Do tej pory zapoznaliśmy się z systemem dwójkowym, szesnastkowy oraz innymi. Jak można było zaobserwować, istotnych różnic pomiędzy nimi nie ma i w zasadzie to jesteśmy w stanie operować dowolnym z nich, a nawet kilkoma, dokonując odpowiednich konwersji przy przejściu od jednego do drugiego. Mimo to cały czas odczuwa się jak gdyby wrodzoną skłonność do operowania systemem dziesiętnym, przejawiającą się chociażby w tym, że w konwersji najczęściej stosujemy schemat **podstawa_źródłowa → podstawa_10 → podstawa_docelowa**. Cały kłopot z systemami liczbowymi o podstawach innych niż 10 polega na tym, że ciąg cyfr jest dla nas abstrakcyjnym napisem, który nabiera znaczenia dopiero wtedy, gdy przekształcimy go do postaci dziesiętnej[3].

Zilustrujemy problemy, o których mówimy za pomocą prostego testu. Kto bez zastanowienia odpowie, ile wynosi 1/3 z liczby $111100_{(2)}$? Jak więc widać, systemem dwójkowym operujemy z konieczności, a najchętniej (najsprawniej) używamy dziesiętnego. Problemem jest konwersja z jednego na drugi. Chcąc ułatwić to zadanie wprowadzono system kodowania BCD (ang. *binary-coded decimal*). W tym systemie liczby dziesiętne kodujemy za pomocą ciągu bitów przypisując każdej cyfrze dziesiętnej odpowiadający jej ciąg 4 bitów (patrz tab. 1.1). Ponieważ mamy jedynie 10 możliwości, więc wystarczy zapamiętać tylko tyle, aby móc sprawnie i szybko przechodzić z systemu dziesiętnego do BCD i odwrotnie. Zobaczmy, jak odbywa się ten proces. Zapiszmy liczbę dziesiętną 120 najpierw w systemie dwójkowym, a następnie w formacie BCD. Zatem $120_{(10)} = 1111000_{(2)}$.

[3] Oczywiście wynika to z przyzwyczajenia i wychowania w określonej kulturze. Kiedyś dla przykładu liczono tuzinami czy kopami.

Cyfra dziesiętna	„Cyfra” BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
–	1010
–	1011
–	1100
–	1101
–	1110
–	1111

Tablica 1.1. Cyfry dziesiętne i odpowiadające im „cyfry” BCD

Example 1.13. Zamiana liczby dziesiętnej na jej postać w kodzie BCD.

$1_{(10)}$ $= 0001_{(BCD)}$
$2_{(10)} = 0010_{(BCD)}$
$0_{(10)} = 0000_{(BCD)}$

Zatem: $120_{(10)} = 00010010000_{(BCD)}$

Łatwo można poczynić następujące spostrzeżenia:

- Pomimo, iż kod BCD do zapisu używa cyfr dwójkowych, to otrzymany ciąg różny jest od odpowiadającej liczby dwójkowej.
- Konwersja 10 → BCD lub BCD → 10 odbywa się na podobnych zasadach, jak konwersja 2 → 16 lub 16 → 2 – grupujemy bity po 4.
- Kod BCD wymaga więcej pamięci niż „tradycyjny” zapis dwójkowy.
- Konwersja jest naturalna i odbywa się prawie natychmiast.

W kodzie BCD oczywiście można wykonywać działania arytmetyczne. Ilustruje to przykład 1.14, w którym pokazano dodawanie w kodzie BCD i właśnie do dodawania ograniczymy się przy omawianiu działań.

Example 1.14. Dodawanie bez przeniesienia w kodzie BCD.

```
142 (10) = 000101000010 (BCD)
311 (10) = 001100010001 (BCD) +
-------------------------------
           010001010011
           |   |   |
           |   |   0011 (BCD) = 3 (10)
           |   |
           |   0101 (BCD) = 5 (10)
           |
           0100 (BCD) = 4 (10)
```

Niestety nie zawsze wszystko układa się tak dobrze... Zauważmy, że możemy w wyniku dodawania uzyskać na pewnej pozycji liczbę większą od 9, co spowoduje konieczność dokonania przeniesienia (przykład 1.15).

Example 1.15. Ilustracja wystąpienia przeniesienia w dodawaniu w kodzie BCD.

```
 9 (10) = 1001 (BCD)
 6 (10) = 0110 (BCD) +
----------------------
15 (10) = 1111 (BCD) ????
```

Otrzymany w przykładzie 1.15 wynik nie jest poprawną „cyfrą" kodu BCD! Pojawia się konieczność korekcji otrzymanych wyników. Korekcja polega na dodaniu liczby dziesiętnej 6 (`0110 (BCD)`) do niepoprawnej grupy 4 bitów; ma to miejsce, gdy „cyfra" BCD jest większa od `1001` lub gdy nastąpi przeniesienie z jednej czwórki bitów do następnej, ilustruje to przykład 1.16.

Example 1.16. Dodawanie w kodzie BCD z uwzględnieniem przeniesienia.

```
 60 (10) = 01100000 (BCD)
 55 (10) = 01010101 (BCD) +
----------------------------
115 (10) = 10110101
  korekcja 0110         +
     --------------------
          100010101
       000100010101
```

```
       |   |   |
       |   |   0101 (BCD) = 5 (10)
       |   |
       |   0001 (BCD) = 1 (10)
       |
       0001 (BCD) = 1 (10)
```

```
 99 (10) =  10011001 (BCD)
 99 (10) =  10011001 (BCD) +
----------------------------
189 (10) = 100110010
  korekcja  01100110 +
  ---------------------
           110011000
        000110011000
        |   |   |
        |   |   1000 (BCD) = 8 (10)
        |   |
        |   1001 (BCD) = 9 (10)
        |
        0001 (BCD) = 1 (10)
```

```
 99 (10) = 10011001 (BCD)
 11 (10) = 00010001 (BCD) +
----------------------------
110 (10) = 10101010
  korekcja 01100110 +
 ---------------------
          100010000
       000100010000
       |   |   |
       |   |   0000 (BCD) = 0 (10)
       |   |
       |   0001 (BCD) = 1 (10)
       |
       0001 (BCD) = 1 (10)
```

1.5 Zadania

1. Wykonaj uproszczenie następujących wyrażeń algebraicznych, a następnie sprawdź prawdziwość metodą zero-jedynkową:
   a) $xz + xy + y\overline{z}$,
   b) $xyz + x\overline{y} + y\overline{z}$,
   c) $xyz + x\overline{y}z + xy\overline{z}$,
   d) $xy + x\overline{y} + \overline{x}y$,
   e) $xz + \overline{z}x + yz + y\overline{z} + \overline{x}$.

2. Dokonaj konwersji zapisu następujących liczb z systemu o podstawie 10 na liczby w systemach o podstawach 2, 8 i 16:
   a) 172, b) 517, c) 778, d) 13, e) 76,
   f) 107, g) 300, h) 201, i) 472, j) 802.

3. Dokonaj konwersji zapisu następujących liczb z systemu o podstawie 2 na liczby w systemach o podstawach 8, 16 i 10:
   a) 11100, b) 1011110001,
   c) 10001010001, d) 100100100,
   e) 1110011100, f) 101110001,
   g) 1001001100, h) 101010000,
   i) 10100001, j) 1000101.

4. Dokonaj konwersji zapisu następujących liczb z systemu o podstawie 16 na liczby w systemach o podstawach 2, 8 i 10:
   a) $F2$, b) $11F$, c) AAA, d) 100, e) $1AB$,
   f) 123, g) FF, h) $F0$, i) BAB, j) 307.

5. Dokonaj konwersji zapisu następujących liczb z systemu o podstawie 8 na liczby w systemach o podstawach 2, 16 i 10:
   a) 123, b) 457, c) 177, d) 65, e) 22,
   f) 10, g) 27, h) 55, i) 222, j) 512.

6. Dokonaj konwersji zapisu liczby z systemu o podstawie 7 na system o podstawie 5:
   a) 565, b) 100, c) 62, d) 12, e) 166,
   f) 306, g) 255, h) 32, i) 836, j) 56.

7. Dokonaj konwersji zapisu liczby z systemu o podstawie 5 na system o podstawie 11:
   a) 1234, b) 4222, c) 2131, d) 1421, e) 3123,
   f) 1121, g) 2041, h) 4131, i) 3211, j) 3114.

8. Dokonaj konwersji zapisu liczby z systemu o podstawie 13 na system o podstawie 9:

a) $C99$, b) $2A5$, c) 91, d) 65, e) $3BC$,
f) 910, g) $B7$, h) 18, i) 301, j) $40C$.

Rozdział 2

Algorithms and data structures

Programy stanowią w końcu
skonkretyzowane sformułowania
abstrakcyjnych algorytmów na
podstawie określonej reprezentacji
i struktur danych.
Niklaus Wirth

2.1 Algorithm

As we know, from previous chapter (see. **??**) a term *algorithm* comes from the name of Persian astronomer and mathematician lived between VIII and IX AD. As an example we can consider mentioned in section **??** Euclid's algorithm to determine the greatest common divisor or to determine angle division into two equal parts using only a compass and a ruler, but also we can refere as an algorithm to process of replaceing damaged element in the computer.

The first descriptions, which was later called algorithms, concerned solutions to mathematical problems. In ancient Egypt and Greece had already been created many methods which allowed to solved some problems in algorithmic way. No generally accepted formal definition of algorithm exists yet. As the term is popularly understood, algorithm mean the way of doing sth, recipe for sth or formula for sth. It should be emphasized, that not every method or schema can be called algorithm.

In mathematic and computer science, algorithm mean finite, ordered sequence of clearly defined actions, needed to perform some task. Algorithm should meet the following conditions:

1. Explicitness or uniqueness. Interpretation of each step should be

unambiguous.

2. Standed out the beginning and the end. We have to know when we start (and what we need to start) and when we get a solution.
3. Discreteness. The number of operation needed to finish work shoud be finite. It can be very large but always finite..
4. Versatility. It'll be best if we can use it to whole class of problems but not only to specific task..
5. Effectiveness. Our goal should be reached in acceptable time. What means *acceptable* depend of course on problem[1].

In the past mathematician sometimes tried to solve various problems in algorithmic way, but in general they did not pay to much attention to precise description of subsequents procedure's steps. With the development of mathematic come out problems, that was not sure if it could be solved in algoritmic way or not. That time it had beed started to thinking about precise definition of notion of algorithm.

To show, that given problem can be solved algorithmicly it is enought to give concrete algorithm to solve that problem (it can be any correct algorithm solving that problem). If we want to proof, that given problem can not be solved algorithmicly, we have to proof that there does not exist any algorithm solving given problem. It cause, that there is a need of precise definition of algorithm. In the nineteen twenties problem of precise definition, in a way of mathematic, of algorithm was the one of the main mathematic's problems.

That definitions were done among other by Alan Turing, Emil Post and Alonzo Church. Based on that definitions it was showed, that in mathematic there are a lot of problems which could not be solved algorithmicly.

Interest in algorithms increased with computer science development. In the fiveties appeared a new algorithm executor – computer. It meant from one side an increase in data processing speed and thanks to it more and more algorithms becames solvable in finite (and what is more important: not to distant) time. But on the other hand it required carefully prepared definition of algorithms as well as coming into existence a programming languages in which it could be possible to wrote algorithms.

The place of algorithm in a process of creating program (implementing algorithm) solving given problem is stanted out on below list (please compare this list with text at the begining of the next section).

1. Problem.

2. Computer (time, internal data representation, software).

[1]The last two conditions are not necessar for correctnes of algorithm, but in the other hand who wants to use algorithm which results will be accessible in distant future (e.g. one million years).

3. Programming language (available construction and data types).

4. **Algorithm**.

5. Program.

With the computer science devlopment algorithm beacame a fundamental notion. On one side it can be treated as an computer scientist's tool serve to solve various problems, but on the other it can be treated as an object of investigation. It is interesting not only how to construct an algorithm solving stated problem, but also if this algorithm is correct and realizible i.e. if results are correct and time needed to get results is acceptable.

The issues connected with designing and estimation of quality and correctness of algorithms is known as **analysis of algorithms**.

2.2 Data structures

Today's compters in most cases are used to store and seek out informations, which becames greater from one day to another. Moreover we expect as well a possibility of organizing, grouping and classyfying that informations. Everything is because of beeing able to find data which we need faster and with highter precision.

Any information stored by computers is only a some kind of real word's sample. It can be said, that it is only an specyfic and abstract model of the world. What is more this model or part of our reality can be different depend on our needs. As an example consider an abstract model of a person. It is obvious that we need other informations to describe person treated as an hospital's patient an other when we treat person as a student. In both cases we can dealing with the same person, but the informations we need are different. That is why abstract in that context means ignoring this features of real object which are unnecessary for our problem.

In case of comming to solve a problem with the use of computer one should first think what kind of data one goes to process. After that we have to think how the data will be represent in computer – that is, what kind of **data structures** will store them. Very often choosing representation we have to bare in mind certain machine's limits beeing, for example, a result of nowadays computer's architecture. To make an example, we know that we can represent real numbers only with certain precision. Therefore choice of way we represent our data often is not easy. What is more, very often the same data we can represent in many ways which have it's own pros and cons. Continuing that reasonong we can come to a (correct) conclusion that choice of data structure determine algorithms we can use and by that determine efficiency of all programm.

Summing up above:

- Information stored and processed by computer is a small fragment of reality containing essential data to solve stated problem.
- We have to think which informations are essential, which can help us and which are completely useless.
- We have to think how we will represent choosen informations.

2.2.1 The notion of data type

If we recall some basic concepts from mathematics, among them we will find such kind of elements like real numbers or integer numbers. Then we can talk about real variables or integer variables that is about elements of set consist of integer or real numbers. In case of algorithms we also introduce concpet of **variable** and set of admissible values for that variable[2].

Usually we distinguish **primitive types** also known as **basic types** or **built-in types** and **composite types**. As a primitive types we consider:

- **numerical type** (e.g. integer, floating-point number, fixed-point number);
- **character type** (alphanumeric symbols);
- **bolean type** (logic type for *true* and *false* value).

As a composite types we consider:

- **array**,
- **dictionary**,
- **record**,
- **class**,
- **file**,
- **set**,
- **queue**,
- **stack**,
- **tree**.

[2]There are exist programming languages in which there is no necessity of informing about the type of each variable. However, in our opinion, it can be confusing for begginers, and what i worse, it can teach a bad habits. Moreover, writing in typeless programming languages demand some kind of experience because of easiness of making hard to find mistakes.

Now we will discuss all listed composite types, focusing on most important features, without entering an implementation's niuanses or any wariants and modification of this types. Extended information about data structures and algorithms can be found in many other books, some of them we point out in bibliography.

2.2.2 Array

Array is one of the most often used data structure and it can be found in almost every programming language on implementation level. In array we can store only elements of one choosen type[3].

Let's imagine a box for storing CD. It is impossible to put into it e.g. a video tape. Next characteristic feature of array is **random access** to any element – it means, that we can refer to any element in any order and that time which we need to get any element is constant. Once again refer to CD box: each disc can be taken in the same easy way independent on the disk position in the box. In case of array to take specific element **indexing** is use, i.e. we give the number of element in array we want to use. For example, let `T` be a variable representing an array created in following way:

```
T: Array of Integer;
```

Such defined array `T` allow to store integer numbers. To call an element at index 1 and store into it value 3 we can write:

```
T[1] := 3
```

In this example it was used operator := as an assign operator contrary to operator = used as compare operator.

Let's notice that in some implementation one should give a size of an array. Then we can store only a limited number of elements. Definition of five-element's array of integer object can be written as follow:

```
T: Array[5] of Integer;
```

Above notation is a case of convention and vary depend on accepted symbolic or specyfic programming language. For example it is not obvious, from which number we start indexing array; usually we use 0 or 1. Some programming languages allow as to define a scope of index, e.g. in following way:

```
T1: Array[2..6] of Real;
T2: Array[-6..-2] of Real;
```

[3]There are some programming languages in which we can store in one array many different types, but to make our disquisition clear we want avoid such an array as we want avoid typeles programming languages

In this example we created two array variables allowing us to store five real numbers, where for the first index vary from 2 to 6 and for the second from -6 to -2.

Nothing stand in the way of talking about multidimensional array, when each index can take values from differnt set. Intuitively twodimensional array can be treated as matrix in mathemaics, and as an example of threedimensional array we can consider Rubik's cube. For the higher dimension our immagination fail [4]. However it is not hard to give an example justify such a construction. Consider multistorey building – it's address is a first dimension, storey – second dimension, corrido – the third, room number – fourth etc. Three dimensional array we can define in following way:

```
T: Array[2..6][3..7] of Integer;
```

Now we give some examples of array usage taken from different programming languages.

```
Ada:
-- definition of array type
type TableType is array(1 .. 100) of Integer;
-- variable definition of specyfic array type
MyTable : TableType;

Visual Basic:
Dim a(1 to 5,1 to 5) As Double
Dim MyIntArray(10) As Integer
Dim MySingleArray(3 to 5) As Single

C:
char my_string[40];
int my_array[] = {1,23,17,4,-5,100};

Java:
int [] counts;
counts = new int[5];

PHP:
$first_quarter  = array(1 => 'January', 'February', 'March');

Python:
mylist = ["List item 1", 2, 3.14]
```

[4] At least writing this words authors.

2.2.3 Dictionary

First let's try to understand what is dictionary; after that we will tell how it can be written. Let's imagine a box. This box i intended to story different kind of object. All objects are not sorted as in array but spreaded chaotically. However additionaly to each object it is fixed string ended by small slip of paper. On this slip there is written a text thanks to it we can explicitly identify object at the other end of a string. This text is called a **key**. All slips are sorted in some way, e.g. alphabetically, for the greates to the smallest etc. – it depend on our preferences. Now if we want to get an access to specific object we do not look for it in the box but we pull the string ended by right slip.

Then in dictionary we can story object beeing a pair **key-value**. Key is some kind of identificator or headword pointing a real object that is a value. Objects in dictionary are orderless. Possible order of object depend on the order of key usage.

Below we present some of dictionary operations used in Python programming language.

```
d = {"key1":"val1", "key2":"val2"}
x = d["key2"]
d["key3"] = 122
d[42] = "val4"
```

2.2.4 Set

Next worth mention data structure is a **set**. Set can be treated as equivalent of well known mathematical set. Therefore there are operations like: **intersection**, **union**, **(relative) complement**, **membership**.

Definition of set variable can be written as follow:

```
S = Set of setType;
```

It means that `S` is a set of elements of type setType.

Set is an interesting programming construction, which do not have features of previous. It means that we can not call to third element, because elements in a set do not have an order, and we can not use a name or a key like in dictionary, because elements in a set do not have a name or key. Instead of this we can easily answer if set is empty, if some object is or is not an element of particular set etc.

2.2.5 Record

Jak można było zauważyć pewną wadą tablicy jest konieczność przechowywania elementów tego samego typu[5]. W pewnych sytuacjach jest to

[5]Co prawda konkretne realizacje tablic w istniejących językach programowania dopuszczają wkładanie do tablicy elementów różnych typów, ale tak naprawdę są to nowe

niedogodne, ale dla nich został zaprojektowany **rekord** (ang. *record*), który jest najbardziej ogólną metodą na stworzenie struktury danych. W rekordzie istnieje możliwość zapamiętania dowolnej, ale określonej liczby danych dowolnych typów. Elementy występujące w rekordzie nazywamy jego **polami**. Jako przykład takich konstrukcji może służyć rekord danych osobowych, gdzie będzie zapamiętane imię, nazwisko, wiek itp. Dopuszcza się również, by elementami (składowymi) rekordu były inne typy złożone, czyli np. tablice, rekordy, zbiory itd.

Głównym przeznaczeniem rekordu jest powiązanie różnych informacji w jedną logiczną całość, tak by móc się do niej odwoływać wspólnie. Przykładowo, jeśli znajdziemy poszukiwaną osobę z imienia i nazwiska, z pewnością będą nas interesowały dodatkowe informacje z nią związane, jak np. adres zamieszkania[6].

Typ rekordowy będziemy definiowali w następujący sposób:

```
type osoba = rekord
  imie: napis;
  nazwisko: napis;
  wiek: Integer;
end
```

Powyższy przykład definiuje rekord składający się z imienia, nazwiska, wieku[7]. W przeciwieństwie do tablicy w rekordzie nie ma mowy o 5., czy 2., elemencie, mówimy wyłącznie o polu o określonej nazwie. Zatem odwołanie od konkretnego pola odbywa się poprzez określenie, o jaki rekord nam chodzi i o jakie jego pole, zwykle te nazwy oddzielone są kropką[8]. Stosując definicję rekordu „osoba" z powyższego przykładu, odwołania do jego pól będą miały postać:

```
  o: osoba;
  wypisz(o.imie);
  o.wiek := 12;
```

Powyższy zapis oznacza definicję zmiennej o nazwie „o", która jest typu „osoba", a następnie odwołanie się do imienia z tego rekordu i wypisanie go, po czym wpisanie wartości do pola „wiek" w tym rekordzie.

Wprowadza się również pojęcie rekordu z wariantami. Jest to taka konstrukcja, która może mieć pewne pola alternatywne, np. dla osoby można zapamiętać wiek, lub datę urodzenia. Jednak nie będziemy tej wersji omawiali szerzej – zainteresowany czytelnik może znaleźć opis tej struktury danych np. w książce [**?**].

konstrukcje struktur złożonych, tzw. tablice z wariantami.

[6] Faktycznie adres zamieszkania także może być rekordem.

[7] W definicji użyto typu „napis", który wcześniej nie był zdefiniowany, ale przyjmujemy jego intuicyjne znaczenie.

[8] „Zwykle", ponieważ zależy to od przyjętej notacji, kropka jest stosowana w większości języków programowania.

2.2.6 Klasa

2.2.7 Plik

Omawiane do tej pory struktury danych charakteryzowały się tym, że była znana ich wielkość[9]. Zatem stosunkowo łatwo było je przechowywać w pamięci. Niestety istnieje całą rzesza dynamicznych struktur danych, takich jak kolejki, drzewa, grafy itp., które nie posiadają tej cechy. Przyjrzyjmy się jednej z takich struktur danych, a mianowicie **plikowi**. Plik reprezentuje strukturę danych, która odpowiada koncepcji ciągu. Ciąg jest sekwencyjną strukturą danych, gdzie elementy są ustawione jeden po drugim. Elementami ciągu mogą być rekordy, tablice i inne typy złożone, przy czym ich liczba jest nieograniczona, zatem zawsze można dopisać następny element do ciągu[10]. Podstawowe operacje na ciągach to: pobranie elementu pierwszego, dodanie elementu na końcu, połączenie dwu ciągów w jeden ciąg. Fizycznie ciąg jest reprezentowany poprzez **plik sekwencyjny**, gdyż to pozwala odwzorować charakter ciągu i operacji na nim wykonywanych.

Definicja typu plikowego ma postać:

```
T:file;
```

Dla pliku zdefiniowane są podstawowe operacje, takie jak: **tworzenie pliku**, **otwarcie pliku**, **dopisanie elementu na końcu pliku**, **odczytanie następnego elementu pliku**.

2.3 Metody opisu algorytmów

Jak wspomniano wcześniej, algorytm jest pewnego rodzaju sformalizowanym zapisem pewnej procedury. W celu uniknięcia nieporozumień co do znaczenia pewnych słów, które mogłyby spowodować złą interpretację algorytmu, wprowadza się umowne sposoby zapisu algorytmów. I tak zasadniczo wyróżnia się zapis za pomocą: **języka naturalnego**, **schematu blokowego**, **metajęzyka programowania**. Zanim omówimy pokrótce każdy z tych sposobów, wymieńmy najważniejsze ich cechy.

- język naturalny
  - teoretycznie łatwy do napisania (wypunktowanie czynności)
  - często mała precyzja[11]
  - kłopoty implementacyjne

[9] Oczywiście, o ile znana była wielkość ich elementów.

[10] O ile oczywiście wystarczy nośnika danych.

[11] Będąca wynikiem niejednoznaczności języka naturalnego i brakiem doświadczenia — ten sposób zapisu wybierają najczęściej osoby nie mające na co dzień do czynienia z algorytmami.

- schemat blokowy
  - duża przejrzystość i czytelność
  - odzwierciedla strukturę algorytmu wyraźnie zaznaczając występowanie rozgałęzień (punktów decyzyjnych)
  - kłopoty implementacyjne
- pseudojęzyk
  - ułatwia implementację
  - mniejsza przejrzystość

2.3.1 Język naturalny

Zapis algorytmu w postaci języka naturalnego jest niczym innym jak usystematyzowaniem i wypunktowaniem czynności jakie należy wykonać. Przyjrzyjmy się dobrze znanemu przykładowi — algorytmowi Euklidesa.

1. Weźmy dwie liczby całkowite dodatnie: a i b.

2. Jeśli $b = 0$ idź do **3.**, w przeciwnym razie wykonaj:

2.1. Jeśli $a > b$ to $a := a - b$.

2.2. W przeciwnym razie $b := b - a$.

2.3. Przejdź do **2.**

3. a jest szukanym największym dzielnikiem.

4. Koniec

2.3.2 Schemat blokowy

Schematy blokowe są uniwersalną i dającą odpowiedni stopień ogólności formą zapisu algorytmu. Cechuje je przy tym duża przejrzystość i czytelność. Schematy te odzwierciedlają dobrze strukturę algorytmu, w tym takie elementy, jak **rozgałęzienia** czy **punkty decyzyjne**. Punkt decyzyjny to miejsce, w którym podejmowana jest decyzja, którą z gałęzi (dróg) algorytmu pójść. Decyzja ta jest podejmowana na podstawie pewnego warunku lub ich zespołu. Zatem punkt decyzyjny jest miejscem rozgałęzienia algorytmu. Rysunek 2.1 zawiera podstawowe elementy schematu blokowego, których znaczenie jest następujące:

stan – określa zwykle moment startu i końca;

zapis/odczyt – wskazuje miejsca, w których odbywa się zapis danych (bądź ich odczyt) na nośniki informacji;

instrukcje – blok instrukcji, które mają być wykonane;

Rys. 2.1. Przykładowe elementy graficzne stosowane w schematach blokowych

decyzja – wyliczenie warunku logicznego znajdującego się wewnątrz symbolu i podjęcie na jego podstawie decyzji;

łącznik – połączenie z inną częścią schematu blokowego, np. gdy nie mieści się on na jednej stronie;

dokument – jakaś forma dokumentu (zwykle „papierowa"), która jest generowana przez system (np. zestawienie, raport, lista, podsumowanie).

Schemat blokowy, zwany również **siecią działań**, tworzony jest według pewnych reguł:

1) składa się on z bloków połączonych zorientowanymi liniami;
2) bloki obrazują ciąg operacji;
3) zawsze wykonywane są albo wszystkie instrukcje w bloku, albo żadna;
4) dalsze operacje nie zależą od poprzednich wariantów, chyba że zależności te zostały przekazane za pomocą danych;
5) kolejność wykonania operacji jest ściśle określona przez zorientowane linie łączące poszczególne bloki;
6) do każdego bloku może prowadzić co najwyżej jedna linia;
7) linie mogą się łączyć, a punkty takiego połączenia określane są jako **punkty zbiegu**.

Na rys. 2.2 znajduje się fragment algorytmu do obliczania pierwiastków trójmianu kwadratowego. Łączniki A i B informują, że dalsze części algorytmu znajdują się na osobnym schemacie lub schematach. Wtedy tamte schematy muszą zaczynać się od odpowiedniego łącznika.

2.3.3 Schemat zapisu algorytmu za pomocą pseudojęzyka

Inną metodą przedstawienia algorytmu jest użycie zapisu w pewnym **pseudojęzyku programowania**. Zaletą tego podejścia jest bardzo łatwa

Rys. 2.2. Schemat blokowy – fragment algorytmu znajdowania pierwiastków trójmianu kwadratowego

późniejsza implementacja za pomocą już konkretnie wybranego, istniejącego języka programowania[12]. Wadę stanowi natomiast mniejsza przejrzystość. Dla osób nieprzywykłych do analizowania kodu źródłowego programów dużo czytelniejsze jest przedstawienie w postaci schematu blokowego. Poniżej przedstawiamy typowe operacje użyte do opisu algorytmu za pomocą omawianej metody, przy czym dla porównania i gdzie jest to sensowne przedstawiamy również ich postać blokową.

Opiszemy teraz podstawowe instrukcje służące do opisu algorytmów prezentując również ich graficzne notacje.

instrukcja podstawienia

```
x:=y;
wiek:=12.6;
imie:="Piotr";
```

W wyniku wykonania powyższych instrukcji zmienna `x` przyjmie wartość

[12]Dzieje się tak, ponieważ większość konstrukcji podstawowych, takich jak warunek czy pętla, ma podobną postać w różnych językach.

zmiennej `y`, zmienn `wiek` przyjmie wartość 12.6 a zmienna `imie` przyjmie wartość będącą napisem „Piotr".

blok instrukcji

```
begin
  instrukcje zawarte
  w bloku
end
```

Blok jest sposobem na grupowanie instrukcji. W pewnym sensie odpowiada jemu blok instrukcji ze schematu blokowego, choć tutaj znaczenie bloku jest znacznie szersze. W jego wnętrzu może wystąpić bowiem dowolna instrukcja (np. instrukcja warunkowa). Konsekwencją stosowania bloku jest tak zwany **zasięg nazw**. Otórz zmienna utworzona w bloku istnieje tylko do końca tego bloku. Oznacza to, że w poniższym przykładzie zmienna `x` może być używana w całym bloku pierwszym (także wewnątrz bloku drugiego) natomiast zmienna `y` widoczna (dostępna) jest tylko w bloku drugim.

```
#blok pierwszy
begin
  x:=1;
  #blok drugi
  begin
    y:=2;
    Wypisz(x);
    Wypisz(y);
  end
  #nie można napisać:
  #Wypisz(y);
  Wypisz(x);
begin
```

W powyższym kodzie możemy także zauważyć występowanie symbolu `#` który oznacza komentarz a więc wszystko to co (począwszy od tego symbolu) jest pomijane podczas wykonywania kodu programu.

instrukcja warunkowa

```
if (WARUNEK) then
begin
  instrukcje realizowane, gdy warunek jest spełniony
  (gdy warunek jest prawdziwy)
end
else
begin
```

```
  instrukcje realizowane, gdy warunek jest niespełniony
  (gdy warunek jest fałszywy)
end
```

Podstawowa instrukcja warunkowa: wpierw wyliczany jest warunek w nawiasie[13], następnie, jeśli jest on prawdziwy, wykonywane są instrukcje w pierwszym bloku, w przeciwnym razie w drugim bloku, po słowie `else`. Często używany jest również w uproszczonej wersji, bez bloku „w przeciwnym razie" (czyli bloku b). Ilustracja graficzna znajduje się na rys. 2.3. Przykłady warunków:

```
x=7
x>12
x>12 and y<3
x=5 and (y=1 or z=2)
```

Rys. 2.3. Instrukcja warunkowa `if` zapisana w postaci schematu blokowego: a) warunek `if (W) instrukcje`, b) `if (W) instrukcje1 else instrukcje2`

instrukcje pętli do–while i while

```
do
begin
  ... ciąg instrukcji powtarzanych
  dopóki warunek jest spełniony ...
end
while (warunek);

while (warunek)
```

[13] „Wyliczany" oznacza w tym miejscu tyle, co „wyznaczana jest" jego wartość logiczna.

```
begin
  ... ciąg instrukcji powtarzanych
  dopóki warunek jest spełniony ...
end
```

Są to dwie pętle, umożliwiające wykonywanie cykliczne instrukcji, które znajdują się w ich wnętrzu. Przy czym instrukcje te są wykonywane tak długo, jak długo `warunek` ma wartość logiczną `prawda`. Nie można więc określić ile razy pętla się wykona, ale za to wiemy jaki warunek powoduje, że się ona zakończy. Różnica pomiędzy tymi dwoma pętlami sprowadza się do kolejności wykonania instrukcji wewnętrznych i sprawdzenia warunku. Pierwsza z nich najpierw wykonuje instrukcje, a następnie sprawdza warunek, druga zaś odwrotnie. Ilustracja graficzna omawianych pętli znajduje się na rys. 2.4.

Rys. 2.4. Pętla `while` i `do-while` zapisana w postaci schematu blokowego

instrukcja pętli for

```
for i:=1 to 10 step 1 do
begin
  ... instrukcje powtarzane 10 razy ...
end
```

Pętla ta wykonuje cyklicznie instrukcje zawarte pomiędzy słowani `begin` i `end`. W przykładzie wykonanie zaczyna się od przypisania zmiennej `i` wartości `1` – początek (P), po czym następuje sprawdzenie, czy `i` nie przekracza wartości granicznej `10` – warunek końca (K). Jeśli warunek ten jest fałszywy, wykonane zostają instrukcje, następnie zostaje zmieniona wartość zmiennej `i` o wartość występującą po słowie `step` (S) i na koniec następuje przejście do sprawdzania warunku (K).

A zatem, w odróżnieniu od na przykład pętli *while*, w przypadku pętli *for* zawsze wiemy ile razy się ona wykona. Można również pomijać jawne wskazanie wartości kroku. W takim przypadku przyjmuje się, że jest on równy `1`. Ilustracja graficzna z odpowiednim oznaczeniem literowym znajduje się na rys. 2.5.

Rys. 2.5. Pętla `for` zapisana w postaci schematu blokowego

funkcje **Funkcją** nazwiemy zbiór instrukcji, który posiada nazwę, wtedy nazwa ta stanowi jednocześnie nazwę tej funkcji. Do funkcji można przekazać dane poprzez jej argumenty, można też żądać od niej zwrotu policzonych wartości[14]. Funkcję postrzegać możemy jako zamknięte pudełko, które po dostarczeniu wymaganych argumentów może wykonać dla nas jakieś ściśle określone zadanie. Często traktowana jest jako zamknięta i niezależna całość, do której jeśli ktoś nie chce nie musi zaglądać i o której nie musi wiedzieć jak działa. Wywołanie funkcji polega na podaniu jej nazwy wraz z wymaganymi dla jej działania argumentami umieszczonymi w nawiasie:

```
NazwaFunkcji(argumenty);
x:=Funkcja(arg1,arg2,arg3);
```

Definicja funkcji ma natomiast postać:

```
function NazwaFunkcji(argumenty)
begin
```

[14]To, jakie argumenty funkcja przyjmuje i jaką wartość zwraca, określa jej twórca.

```
  instrukcje
  return zwracanaWartosc;
end
```

Występująca pomiędzy słowami `begin` oraz `end` (a więc w tak zwanym ciele funkcji) instrukcja `return` ma specjalne znaczenie. Po jej napotkaniu kończy się natychmiast działanie funkcji i zwracana jest ewentualnie[15] wartość wymieniona za słowem `return`.

instrukcje nieformalne

Zdarza się również, że w schematach blokowych wstawia się nieformalne instrukcje wyrażone w języku naturalnym. Można spotkać taki oto zapis:

```
if (plik=OTWARTY) then
begin
  Zapisanie danych do pliku
end
else
begin
  Komunikat o błędzie
end
```

gdzie oczywiście komunikaty „`Zapisanie danych do pliku`" i „`Komunikat o błędzie`" są tylko informacjami o tym, co w tym miejscu ma wystąpić, a czego z jakiś przyczyn nie chcemy w danej chwili wyspecyfikowywać. Często są to po prostu odwołania do innych procedur, które są dobrze znane i opisane gdzie indziej.

Dla porównania pokażemy teraz sposób zapisu algorytmu Euklidesa przy użyciu pseudokodu.

```
function Euklides(a,b)
begin
   while(b!=0)
   begin
      if(a>b) then
      begin
         a:=a-b;
      end
      else
      begin
        b:=b-1;
      end
   end
```

[15] Ewentualnie, gdyż można pominąc wyrażenie `zwracanaWartość` — w takim przypadku funkcja nic nie zwraca.

```
   return a;
end
```

W przedstawionym kodzie proszę zwrócić uwagę na stosowane wcięcia. Na ogół nie mają one charakteru formalnego i mogą być stosowane dowolnie, ale zwykle ułatwiają czytanie pseudokodu. Stąd dobrą praktyką programistyczną jest wypracownie sobie pewnego sposobu stosowania wcięć i konsekwentnego ich stosowania. Dal porównania pokazujemy też kod zapisany bez użycia wcięc — funkcjonalnie wciąż jest to en sam program choć o znacznie zmniejszonej czytelności.

```
SPOSÓB 1                          SPOSÓB 2
============================================================
function Euklides(a,b)            function Euklides(a,b)
begin                             begin while(b!=0) begin
while(b!=0)                       if(a>b) then begin a:=a-b; end
begin                             else begin b:=b-a; end end
if(a>b) then                      return a; end
begin
a:=a-b;
end
else
begin
b:=b-1;
end
end
return a;
end
```

2.4 Podstawowe algorytmy

Pokażemy teraz kilka algorytmów zapisanych bądź w pseudojęzyku, bądź gdzie to będzie zbyt skomplikowane, ograniczymy się jedynie do nieformalnego zarysowania ich idei. Algorytmy te reprezentują pewne charakterystyczne typy występujące w algorytmice. Z konieczności ograniczymy się do kilku podstawowych grup.

2.4.1 Algorytmy obliczeniowe

Jest to cała gama algorytmów służących do wyliczania pewnych wartości bądź rozwiązywania problemów numerycznych. Przykładami takich algorytmów mogą być NWD, NWW czy obliczanie pierwiastków trójmianu kwadratowego. W przykładzie 2.1 użyto procedury `wypisz`, której działanie, polegające na wyświetleniu podanego w cudzysłowie napisu, przyjmujemy intuicyjnie.

Example 2.1. Algorytm obliczania rzeczywistych pierwiastków równania kwadratowego.

```
if (A=0) then
begin
  wypisz("To nie jest równanie kwadratowe");
end
else
begin
  D:=B^2-4*A*C;
  if (D<0) then
  begin
    wypisz("Nie ma rzeczywistych rozwiązań");
  end
  else
  begin
    if (D>0) then
    begin
      R1:=(-B-Sqrt(D))/(2*A);
      R2:=(-B+Sqrt(D))/(2*A);
    end
    else
    begin
      R1:=-B/(2*A);
      R2:=R1;
    end
  end
  wypisz("Pierwiastkami są:", R1, R2);
end
```

2.4.2 Algorytmy sortowania

Bardzo często czynnością wykonywaną przez każdego z nas, często w sposób nieświadomy, jest procedura **sortowania**. Mówiąc inaczej jest to algorytm porządkowania czy też układania pewnego zbioru elementów według zadanego klucza. Bardzo istotne jest tu wskazanie klucza, który będzie wyróżnikiem, na którym będzie dokonywana operacja porównania. Zauważmy, że jeśli mamy zbiór osób, i teraz ustawimy je rosnąco względem wzrostu, to nie jest to równoznaczne z porządkiem względem wagi, a już na pewno nie względem długości włosów.

Jako przykład prostego algorytmu sortowania opiszemy **sortowanie przez wstawianie**, zwane również sortowaniem przez proste wstawianie. Spójrzmy wpierw na rys. 2.6 i znajdujące się na nim, w linii zatytułowanej START, liczby całkowite w losowej kolejności.

Rys. 2.6. Proces sortowania przez wstawianie

Naszym zadaniem jest poukładanie tych ośmiu liczb, tj, 23, 54, 13, 21, 63, 15, 10 i 61 rosnąco. W tym celu bierzemy pod uwagę wpierw element drugi (54)i zapamiętujemy go w zmiennej pomocniczej (krok 1). Dla tak wyróżnionego elementu sprawdzamy, czy nie można go wstawić gdzieś przed nim, tj. w tym konkretnym przypadku, czy nie jest on mniejszy od pierwszego elementu (23) (krok 2). Zauważamy, że nie jest, zatem pozostawiamy go na miejscu i przechodzimy do następnego – trzeciego elementu (krok 3). Dla niego sprawdzamy wpierw, czy nie możemy go wstawić na drugie miejsce. Możemy, ale najpierw musimy zrobić na niego miejsce, czyli przepisać element drugi na pozycję trzecią (krok 4). Teoretycznie można by teraz wpisać na pozycję drugą ów wyróżniony element. Sprawdzamy jednak dalej, czy przypadkiem wyróżniony element nie jest jeszcze mniejszy od elementu pierwszego. Ponieważ okazuje się mniejszy, zatem przepisujemy element pierwszy na pozycję drugą (zwalniajac tym samym pozycję pierwszą) (krpk 5). Sprawdzanie takie wykonujemy tak długo, aż natrafimy na element mniejszy od elementu wyróżnionego lub rozpatrzymy wszystkie poprzedzające go elementy. W takim przypadku element wyróżniony wstawiamy na ostatnie zwolnione miejsce (krok 6). Całą procedurę powtarzamy kolejno dla pozostałych elementów. Przebieg powyższego procesu przedstawia przykład 2.2. Dla ułatwienia przyjmujemy, że zmienne przechowywane są w tablicy, a rolę zmiennej tymczasowej pełni `a[0]`

Example 2.2. Przebieg procesu sortowania

	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
	--	23	54	13	21	63	15	10	61
krok 1	54	23	54	13	21	63	15	10	61
krok 2	54	23	54	13	21	63	15	10	61
krok 3	13	23	54	13	21	63	15	10	61
krok 4	13	23	54	54	21	63	15	10	61

```
krok 5  13   23   23   54   21   63   15   10   61
krok 6  13   13   23   54   21   63   15   10   61
        ...
```

Spróbujmy zapisać opisany algorytm za pomocą pseudojęzyka. Przy pierwszej próbie zapisania algorytm ten będzie miał następującą postać:

```
for i:=2 to n do
begin
  x:=a[i];
  "wstaw x w odpowiednie miejsce w ciągu a[1]...a[i-1]";
end
```

Teraz sprecyzujemy pseudokod procesu wyszukiwania miejsca wstawienia elementu `x` oraz uzupełnimy powyższy algorytm o ten fragment. W efekcie otrzymamy prostą implementację algorytmu sortowania przez wstawianie zaprezentowaną w przykładzie 2.3. Zauważmy jednocześnie, że algorytm ten działa niejako jak procedura układania książek na półce, wyjmujemy rozważaną książkę, następnie przesuwamy w prawo książki od niej większe, a następnie wsadzamy ją w miejsce zwolnione i dla niej przeznaczone. Teraz, by można było tę książkę wyjąć, należy mieć wolną rękę, która ją przytrzyma. Podobnie w algorytmie: musimy mieć gdzie przechować liczbę, by móc przesunąć na jej miejsce inną. Stąd w przykładzie 2.3 występuje przypisanie `a[0]:=a[i]`, które właśnie zapamiętuje w zerowej komórce tablicy, traktowanej jako zmienna pomocnicza, element, który w danej chwili rozważamy.

Example 2.3. Algorytm sortowania przez wstawianie.

```
for i:=2 to n do
begin
  a[0]:=a[i];
  j:=i-1;
  while (a[0] < a[j])
  begin
    a[j+1]:=a[j];
    j:=j-1;
  end
  a[j+1]:=a[0];
end
```

2.4.3 Algorytmy wyszukujące

Drugim, często spotykanym zagadnieniem jest poszukiwanie informacji o zadanym kluczu, czyli mamy np. zbiór osób, które posiadają jakąś cechę

Rys. 2.7. Proces przeszukiwania połówkowego

(klucz), niech to będzie wzrost. Następnie szukamy osoby o wzroście 178 cm. Możemy spotkać się z dwoma przypadkami. Pierwszy to taki, gdy osoby te są w dowolnym porządku, czyli niepoukładane, oraz drugi, gdy mamy grupę osób ustawioną względem klucza „wzrost" – jak na wojskowej paradzie. Jeśli będziemy rozpatrywali pierwszy przypadek, to nie pozastaje nam nic innego, jak brać po jednym z elementów i sprawdzać, czy pasuje do naszego wzorca. Taki sposób nosi też nazwę **wyszukiwania liniowego**. Jednak w drugim przypadku możemy poradzić sobie lepiej, tzn. efektywnej, jeśli zastosujemy **wyszukiwanie połówkowe** lub **binarne**.

Wyjaśnimy ten sposób realizacji procedury szukania na przykładzie. Spójrzmy na rys. 2.7, gdzie w pierwszym wierszu są przedstawione liczby całkowite posortowane od najmniejszej do największej. Będziemy poszukiwali wartości `15`. Jeśli spróbujemy jej szukać tak, jak w zbiorze nieuporządkowanym i zaczniemy naszą procedurę od pierwszego elementu, to znajdziemy tę wartość w czwartym kroku.

Możemy jednak poszukiwać inaczej. Wpierw wybieramy element środkowy w tym ciągu[16]. Porównujemy wartość tego elementu z wartością szukaną. Jeśli jest to ta wartość to zwracamy numer tego elementu. W przeciwnym razie wiemy, że poszukiwany element jest bądź na prawo, bądź na lewo od środka[17]. I dalej rozważamy tylko ten fragment ciągu, w którym potencjalnie znajduje się nasz element[18]. Cały proces powtarzamy dla okrojonego zbioru. Jeśli znajdziemy element, zwracamy jego pozycję. Jeśli dojdziemy do zbioru pustego, zwracamy informację, że nie znaleźliśmy tego elementu. Proces ten jest właśnie zilustrowany na rys. 2.7.

Zauważmy, że element poszukiwany (`15`) został znaleziony w trzech krokach, zatem o jeden krok szybciej niż w przypadku przeszukiwania liniowego. Możemy oczywiście uznać, że to niewielka oszczędność czasu. Jednak zauważmy, że poszukiwanie jakiegokolwiek elementu w tym zbiorze będzie zajmowało co najwyżej cztery kroki w przypadku wyszukiwania połówko-

[16] Jeżeli liczba elementów jest parzysta, to umawiamy się, że jest to jeden z elementów leżących przy środku. Ważne jest tylko, by zawsze to robić konsekwentnie.

[17] Zauważmy, że to jest miejsce, w którym wykorzystujemy wiedzę o uporządkowaniu zbioru. Gdyby te elementy były ułożone przypadkowo, nie można byłoby wysunąć takiego wniosku.

[18] Potencjalnie, gdyż zaczynając poszukiwania nie wiemy, czy on w ogóle jest w tym zbiorze.

wego, zaś aż 9 w najgorszym przypadku wyszukiwania liniowego. Oczywiście, gdy mamy „pecha", poszukujemy akurat elementu występującego na końcu, ale często to właśnie te najgorsze przypadki bierze się pod uwagę przy analizowaniu sprawności (wydajności) algorytmów. Drugą miarą, jaką się rozważa, jest średnia liczba kroków, jaką trzeba wykonać w przypadku jednego bądź drugiego algorytmu.

2.5 Rekurencja a iteracja

W tym punkcie zasygnalizujemy dwa podstawowe podejścia do realizacji prostych algorytmów: **iteracyjne** i **rekurencyjne**. Zacznijmy od wyjaśnienia pojęcia rekurencji.

Obiekt nazwiemy **rekurencyjnym**, jeśli częściowo składa się z siebie samego lub jego definicja odwołuje się do niego samego. Innym słowem, które jest odpowiednikiem **rekurencji** jest **rekursja**.

Jeden z łatwiejszych do zaobserwowania przykładów rekurencji wziętych z życia polega na ustawieniu naprzeciw siebie dwóch lusterek. Powinniśmy zobaczyć lusterko, a w nim lusterko, a w nim lusterko, a w nim ...

Siła rekursji wyraża się w możliwości definiowania nieskończonego zbioru obiektów za pomocą skończonego wyrażenia. Dokładnie na tej samej zasadzie, nieskończoną liczbę obliczeń można zapisać za pomocą skończonego programu rekurencyjnego. Narzędziem umożliwiającym tworzenie programów rekurencyjnych jest, wspomniana przy okazji omawiania pseudokodu, funkcja (można też mówić o procedurze czy podprogramie). Jeśli jakaś funkcja zawiera odwołanie do siebie samej, to nazwiemy ją funkcją rekurencyjną.

Podobnie jak pętle, które pozwalają nam iterować (czyli powtarzać wielokrotnie zestaw instrukcji), funkcje rekurencyjne dopuszczają możliwość wykonywania powtórzeń, z tym wszakże wyjątkiem, że ich ilość będzie nieskończona[19]. Wiąże się z tym konieczność rozwiązania problemu stopu. Zasadniczym wymaganiem w takiej sytuacji jest uzależnienie rekurencyjnego wywołania funkcji od warunku, który w pewnym momencie przestaje być spełniony, co w konsekwencji powoduje zatrzymanie procesu rekurencji. W celu wykazania, że proces wywołań rekurencyjnych się skończy, należy pokazać, że osiągnięty będzie warunek stopu.

Algorytmy rekurencyjne są szczególnie odpowiednie wtedy, gdy rozważany problem lub przetwarzane dane są zdefiniowane w sposób rekurencyjny. Dla przykładu rozważmy dwa sposoby realizacji funkcji obliczającej silnię. Pierwszy z nich, pokazany w przykładzie 2.4, realizuje silnię za pomocą pętli, zatem w sposób iteracyjny.

Example 2.4. Funkcja obliczająca silnię – podejście iteracyjne.

```
function SilniaI(n)
```

[19] Oczywiście ta nieskończona liczba operacji jest wyłącznie teoretyczna, w praktyce przecież realizacja wykonania funkcji odbywa się na komputerze, który nie jest wieczny.

```
begin
  i:=0;
  s:=1;
  while (i<n) do
  begin
    i:=i+1;
    s:=s*i
  end
  return s;
end
```

Drugi sposób realizacji silni jest pokazany w przykładzie 2.5 i jest to podejście rekurencyjne.

Example 2.5. Funkcja obliczająca silnię – podejście rekurencyjne.

```
function SilniaR(n)
begin
  if (n=0) then
  begin
    return 1;
  end

  return n*SilniaR(n-1);
end
```

Example 2.6. Przebieg wywołań rekurencyjnych przy obliczaniu wartości $5!$**.**

```
s_r(5)
.   |
.   5*s_r(4)
.   .     |
.   .     4*s_r(3)
.   .     .     |
.   .     .     3*s_r(2)
.   .     .     .     |
.   .     .     .     2*s_r(1)
.   .     .     .     .     |
.   .     .     .     .     1*s_r(0)
.   .     .     .     .     .     |
.   .     .     .     .     . <---1
.   .     .     .     .     . |
.   .     .     .     . <---1*1
.   .     .     .     . |
.   .     .     . <---2*1
```

```
.   .     .     . |
.   .     . <---3*2
.   .     . |
.   . <---4*6
.   . |
<---5*24
|
120
```

Jednak nie zawsze funkcję, która jest dana wzorem rekurencyjnym powinno się realizować według takiego algorytmu. W istocie istnieją bardziej skomplikowane schematy rekurencyjne, które mogą i powinny być przekształcane do postaci iteracyjnych. Dobry przykład stanowi tutaj zagadnienie obliczania liczb ciągu Fibonacciego. Ciąg Fibonacciego, dla $n > 1$, zdefiniowany jest następująco

$$fib_n := fib_{n-1} + fib_{n-2},$$

natomiast wyrazy 1. i 0. przyjmują wartość 1.

Rekurencyjna implementacja funkcji obliczającej liczby ciągu Fibonacciego jest zaprezentowana w przykładzie 2.7.

Example 2.7. Funkcja obliczająca n-tą liczbę ciągu Fibonacciego – podejście rekurencyjne.

```
function FibR(n)
begin
  if (n=0 or n=1) then
  begin
    return 1;
  end

  return FibR(n-1)+FibR(n-2);
end
```

Podejście rekurencyjne nie sprawdza się w tym przypadku; każde wywołanie powoduje dwa dalsze wywołania, tj. całkowita liczba wywołań rośnie wykładniczo (patrz przykład 2.8, 2.9 oraz rysunek 2.8), co szybko prowadzi do wyczerpania stosu i jednocześnie działa wolno, gdyż funkcja dla tych samych danych jest wyliczana kilkakrotnie[20].

Example 2.8. Przebieg wywołań rekurencyjnych przy obliczaniu szóstego wyrazu ciągu Fibonacciego.

[20] Nawet kilkadziesiąt bądź kilkaset razy.

Rys. 2.8. Zależność ilości wywołań c od czast t.

```
FibR(5)
     |
     +--FibR(4)
     |       |
     |       +--FibR(3)
     |       |       |
     |       |       +--FibR(2)
     |       |       |       |
     |       |       |       +--FibR(1)
     |       |       |       +--FibR(0)
     |       |       |
     |       |       +--Fib(1)
     |       |
     |       +--FibR(2)
     |               |
     |               +--FibR(1)
     |               +--FibR(0)
     +--FibR(3)
             |
             +--FibR(2)
             |       |
             |       +--FibR(1)
             |       +--FibR(0)
             |
             +--FibR(1)
```

Example 2.9. Ilość wywołań funkcji podczas obliczania wyrazów ciągu Fibonaciego od 0 do 35.

```
ilosc wywolan
```

W tym przypadku lepiej użyć funkcji iteracyjnej, jak to zaprezentowano w przykładzie 2.10.

Example 2.10. Funkcja obliczająca n-tą liczbę ciągu Fibonacciego – podejście iteracyjne.

```
function FibI(n)
begin
  i:=1;
  x:=1;
```

```
  y:=1;

  while (i<n)
  begin
    z:=x;
    i:=i+1;
    x:=x+y;
    y:=z;
  end

  return x;
end
```

W celu lepszego uzmysłowienia różnic pomiędzy powyższymi funkcjami dokonajmy porównania czasu wykonania odpowiadających im programów. Poniższa tabela zawiera czasy dla algorytmu rekurencyjnego. Wersja iteracyjna oblicza 1000-ny wyraz w czasie niemierzalnym przy wykorzystaniu stopera. Test przeprowadzono na komputerze 486 DX 4 120 MHz, w środowisku MS-DOS[21].

Wyraz	Czas (s)
30	2
31	2,56
32	3,54
33	5,10
34	7,11
35	12,67
36	18,84
37	29,40
38	48,12
39	75,82
40	>90,0

2.6 Analiza złożoności

Często ten sam problem można rozwiązać za pomocą różnych algorytmów, co było pokazane choćby w podrozdziale 2.5. W takim razie należy się zastanowić, jakie są różnice pomiędzy tymi algorytmami oraz czy różnice te mają wpływ na wybór konkretnego algorytmu do implementacji. Przy porównywaniu algorytmów zwykle bierze się pod uwagę ich **efektywność** (szybkość działania), zapotrzebowanie na zasoby pamięciowe systemu,

[21] Test przeprowadzono specjalnie na starym komputerze, by wyraźniej było widać różnice.

wreszcie ich czytelność. Tak naprawdę czytelność jest rzeczą względną i zależy od długości praktyki osoby czytającej, stąd też rzadko jest brana pod uwagę przy porównywaniach. Istotna z praktycznego punktu widzenia różnica wynika z prędkości działania algorytmu i jego zapotrzebowania na pamięć. Przy czym od razu należy zaznaczyć, że różnice między algorytmami są najczęściej bez znaczenia przy przetwarzaniu małej liczby danych, ale rosną razem z nią i to liniowo, logarytmicznie, z kwadratem, wykładniczo itd.

W celu porównania algorytmów pod względem ich prędkości działania, wprowadzono pojęcie **złożoności obliczeniowej**. Dysponując takim pojęciem, posiadamy pewną miarę, która pozwala porównać różne algorytmy, ocenić efekt poprawienia istniejącego algorytmu, czy wreszcie oszacować czas realizacji danego zadania.

Złożoność obliczeniowa określa, ilu zasobów wymaga użycie danego algorytmu lub jak jest ono kosztowne. Koszt ten można oczywiście mierzyć na wiele różnych sposobów zależnych od postawionego zadania. Stąd czas wyrażany jest za pomocą umownych jednostek.

Rozważmy następujący przykład. Mamy funkcję f zmiennej całkowitej n daną następującym wzorem:

$$f(n) = n^2 + 100n + \log_{10} n + 1000.$$

Dla małych wartości n ostatni czynnik jest największy. Jednak wraz ze wzrostem n maleje jego znaczenie na rzecz pozostałych. Ostatecznie, gdy n przekroczy wartość 100 000, jego znaczenie w ogólnym wyniku jest marginalne. Również pozostałe składniki, oprócz n^2, mają coraz mniejszy wkład. To właśnie głównie od pierwszego składnika zależy wartość funkcji i dla dużych n można przyjąć, że funkcja ta jest postaci

$$f(n) = cn^2,$$

gdzie c jest pewną stałą.

W wyniku takiej obserwacji, jak powyższa, wprowadzono pewne notacje mające na celu określenie złożoności obliczeniowej danego algorytmu. Notacji tych jest kilka. Przedstawimy najprostszą z nich – notację „wielkie O".

Dla danej pary funkcji f i g o dodatnich wartościach rozważmy następujące definicje.

Definition 2.1. *Notacja „wielkie* O*". Powiemy, że funkcja* $f(n)$ *jest rzędu* $\mathrm{O}(g(n))$*, jeśli istnieją dodatnie liczby* c *i* N *takie, że* $0 \leq f(n) \leq cg(n)$ *dla wszystkich* $n \geq N$*.*

Na przykład wyrażenie

$$n^2 + 100n + \log_{10} n + 1000$$

można zapisać jako

$$n^2 + 100n + \mathrm{O}(\log_{10} n),$$

co oznacza, że część „obcięta" jest zawsze mniejsza niż pewna ustalona stała pomnożona przez $\log_{10} n$.

Własności notacji „wielkie O".

Własność 1. Jeśli $f(n)$ jest $\mathrm{O}(g(n))$ i $g(n)$ jest $\mathrm{O}(h(n))$, to $f(n)$ jest $\mathrm{O}(h(n))$. Inaczej: $\mathrm{O}(\mathrm{O}(g(n)))$ jest $\mathrm{O}(g(n))$.

Własność 2. Jeśli $f(n)$ jest $\mathrm{O}(h(n))$ i $g(n)$ jest $\mathrm{O}(h(n))$, to $f(n) + g(n)$ jest $\mathrm{O}(h(n))$.

2.7 Zadania

1. Napisz algorytm do obliczania x^y, gdzie x, y są liczbami naturalnymi, przy czym dopuszczamy, że y może być 0. Napisz wersję iteracyjną i rekurencyjną.

2. Dla iteracyjnej wersji funkcji określonej w poprzednim zadaniu narysuj schemat blokowy.

3. Mając poniższy schemat blokowy zamień go na zapis w pseudojęzyku programowania, zastanów się co robi ten algorytm (rys. 2.9).

4. Napisz funkcję rekurencyjną, która realizuje algorytm wyszukiwania połówkowego opisany w pkt 2.4.3.

Rys. 2.9. Schemat blokowy pewnego algorytmu