Wstęp do informatyki

Algorytmy i struktury danych

Piotr Fulmański

Wydział Matematyki i Informatyki,
Uniwersytet Łódzki, Polska

26 października 2018

Spis treści

Dlaczego potrzebne są nam algorytmy?

Potrzeba ustalenia wspólnego języka, pozwalającego na precyzyjne przekazywanie myśli od nadawcy do odbiorcy w taki sposób, aby minimalizować ryzyko pomyłek związanych z interpretacją przekazywanych komunikatów.

Niestety zgodnie z tym co możemy przeczytać w powiastce filozoficznej Mały Książe[1]

Mowa jest źródłem nieporozumień

a w szerszym kontekście nie tylko mowa, ale ogólnie *komunikacja*.

[1] Antoine de Saint-Exupéry, *Mały Książę* (fr. *Le Petit Prince*), wydana po raz pierwszy w 1943 w Nowym Jorku w wydawnictwie Reynal and Hitchcock.

Zdanie:

The spirit is willing, but the flesh is weak

przetłumaczono automatycznie z j. angielskiego na j. rosyjski
a następnie tak otrzymane zdanie przetłumaczono z j. rosyjskiego na j. angielski

The vodka is good, but the meat is rotten

https://en.wikipedia.org/wiki/Literal_translation

Zdanie:

The spirit is willing, but the flesh is weak

przetłumaczono automatycznie z j. angielskiego na j. rosyjski
a następnie tak otrzymane zdanie przetłumaczono z j. rosyjskiego na j. angielski

The vodka is good, but the meat is rotten

https://en.wikipedia.org/wiki/Literal_translation

Zdanie:

The spirit is willing, but the flesh is weak

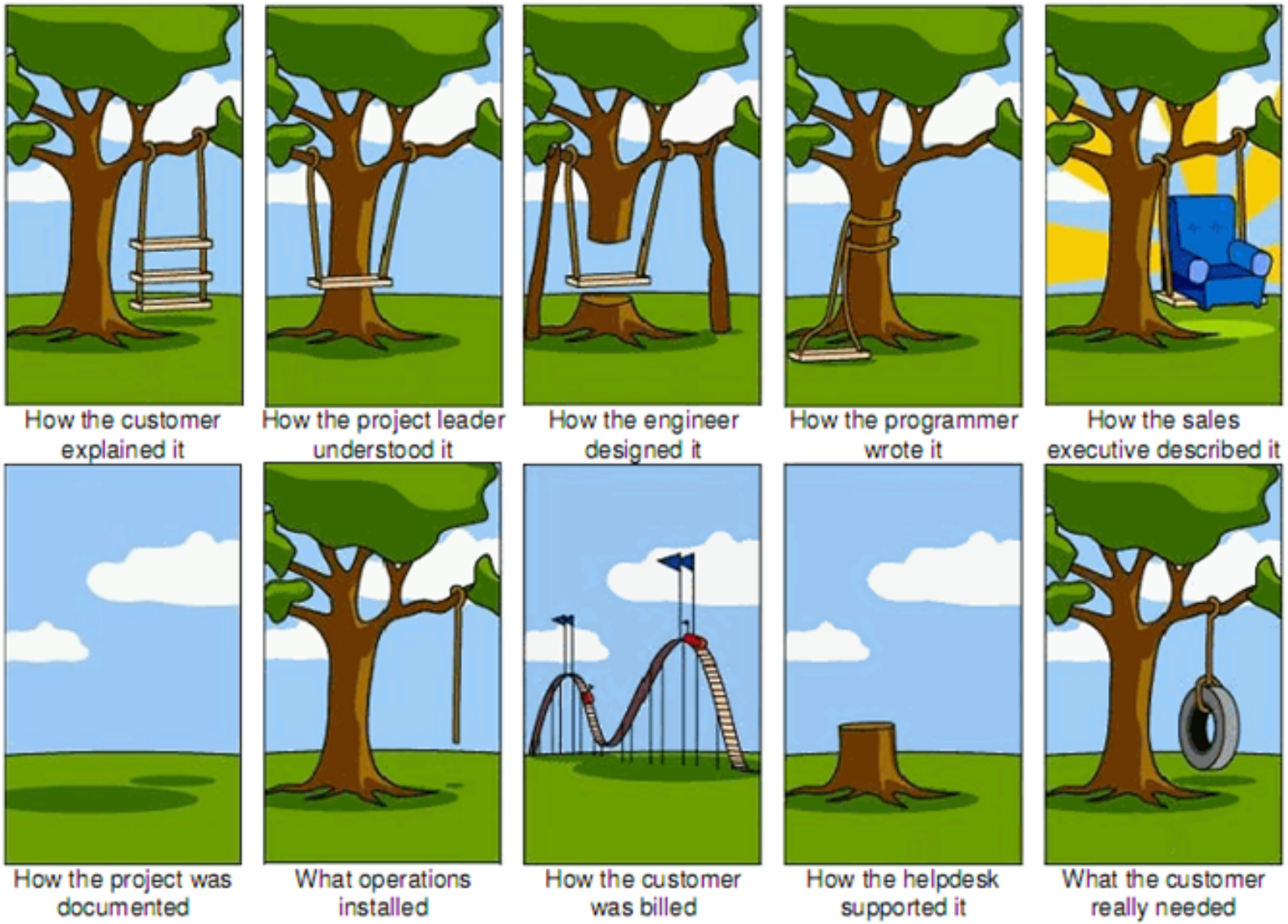
przetłumaczono automatycznie z j. angielskiego na j. rosyjski
a następnie tak otrzymane zdanie przetłumaczono z j. rosyjskiego na j. angielski

The vodka is good, but the meat is rotten

https://en.wikipedia.org/wiki/Literal_translation

Wstęp

Przykłady nieporozumień w komunikacji



https://blog.lansa.com/rapid-application-development/ambiguity-in-requirements

Nazwa

Termin **algorytm** pochodzi od nazwiska perskiego astronoma i matematyka żyjącego na przełomie VIII i IX w n.e. W 825 roku Muhammad ibn Musa al-Chorezmi (al-Khawarizmy) napisał traktat, w którym podał wiele precyzyjnych opisów dotyczących różnych matematycznych reguł (np. dodawania czy mnożenia liczb dziesiętnych). W XII wieku dzieło to zostało przetłumaczone na łacinę jako *Algoritmi de numero Indorum*, co należało rozumieć następująco: *Algoritmi o liczbach Indyjskich*. Pojawiające się tutaj po raz pierwszy słowo *Algoritmi* było oczywiście inaczej zapisanym nazwiskiem matematyka. Większość ludzi rozumiała jednak tytuł bardziej jako *Algorytmy o liczbach Indyjskich* a stąd już blisko do *Algorytmy na liczbach indyjskich (arabskich)*. W ten oto sposób precyzyjnie opisaną metodę obliczeniową zaczęto nazywć algorytmem (łac. *algorismus*).

Jak więc widać, sam termin *algorytm* jest wynikiem nieporozumienia komunikacyjnego.

Algorytm

Nieformalna definicja

Nie ma jednej uniwersalnej definicji algorytmu. W potocznym tego słowa znaczeniu, algorytm oznacza sposób postępowania, przepis na coś, schemat działania.

Bardziej formalnie

Algorytm - w matematyce oraz informatyce to skończony, uporządkowany ciąg jasno zdefiniowanych czynności, koniecznych do wykonania pewnego zadania.

- wyróżniony początek i koniec
- warunek jednoznaczności
- warunek dyskretności
- warunek uniwersalności
- warunek efektywności

Algorytm

Nieformalna definicja

Nie ma jednej uniwersalnej definicji algorytmu. W potocznym tego słowa znaczeniu, algorytm oznacza sposób postępowania, przepis na coś, schemat działania.

Bardziej formalnie

Algorytm - w matematyce oraz informatyce to skończony, uporządkowany ciąg jasno zdefiniowanych czynności, koniecznych do wykonania pewnego zadania.

- wyróżniony początek i koniec
- warunek jednoznaczności
- warunek dyskretności
- warunek uniwersalności
- warunek efektywności

Algorytm

Nieformalna definicja

Nie ma jednej uniwersalnej definicji algorytmu. W potocznym tego słowa znaczeniu, algorytm oznacza sposób postępowania, przepis na coś, schemat działania.

Bardziej formalnie

Algorytm - w matematyce oraz informatyce to skończony, uporządkowany ciąg jasno zdefiniowanych czynności, koniecznych do wykonania pewnego zadania.

- wyróżniony początek i koniec
- warunek jednoznaczności
- warunek dyskretności
- warunek uniwersalności
- warunek efektywności

Algorytm

Nieformalna definicja

Nie ma jednej uniwersalnej definicji algorytmu. W potocznym tego słowa znaczeniu, algorytm oznacza sposób postępowania, przepis na coś, schemat działania.

Bardziej formalnie

Algorytm - w matematyce oraz informatyce to skończony, uporządkowany ciąg jasno zdefiniowanych czynności, koniecznych do wykonania pewnego zadania.

- wyróżniony początek i koniec
- warunek jednoznaczności
- warunek dyskretności
- warunek uniwersalności
- warunek efektywności

Algorytm

Nieformalna definicja

Nie ma jednej uniwersalnej definicji algorytmu. W potocznym tego słowa znaczeniu, algorytm oznacza sposób postępowania, przepis na coś, schemat działania.

Bardziej formalnie

Algorytm - w matematyce oraz informatyce to skończony, uporządkowany ciąg jasno zdefiniowanych czynności, koniecznych do wykonania pewnego zadania.

- wyróżniony początek i koniec
- warunek jednoznaczności
- warunek dyskretności
- warunek uniwersalności
- warunek efektywności

Algorytm

Nieformalna definicja

Nie ma jednej uniwersalnej definicji algorytmu. W potocznym tego słowa znaczeniu, algorytm oznacza sposób postępowania, przepis na coś, schemat działania.

Bardziej formalnie

Algorytm - w matematyce oraz informatyce to skończony, uporządkowany ciąg jasno zdefiniowanych czynności, koniecznych do wykonania pewnego zadania.

- wyróżniony początek i koniec
- warunek jednoznaczności
- warunek dyskretności
- warunek uniwersalności
- warunek efektywności

Algorytm

Nieformalna definicja

Nie ma jednej uniwersalnej definicji algorytmu. W potocznym tego słowa znaczeniu, algorytm oznacza sposób postępowania, przepis na coś, schemat działania.

Bardziej formalnie

Algorytm - w matematyce oraz informatyce to skończony, uporządkowany ciąg jasno zdefiniowanych czynności, koniecznych do wykonania pewnego zadania.

- wyróżniony początek i koniec
- warunek jednoznaczności
- warunek dyskretności
- warunek uniwersalności
- warunek efektywności

Miejsce

Miejsce algorytmu na przykładzie implementacji oprogramowania rozwiązującego postawiony problem.

- problem
- komputer (czas, wewnętrzna reprezentacja danych, oprogramowanie)
- język (dostępne konstrukcje i typy danych)
- **algorytm**
- program

Miejsce

Miejsce algorytmu na przykładzie implementacji oprogramowania rozwiązującego postawiony problem.

- problem
- komputer (czas, wewnętrzna reprezentacja danych, oprogramowanie)
- język (dostępne konstrukcje i typy danych)
- **algorytm**
- program

Miejsce

Miejsce algorytmu na przykładzie implementacji oprogramowania rozwiązującego postawiony problem.

- problem
- komputer (czas, wewnętrzna reprezentacja danych, oprogramowanie)
- język (dostępne konstrukcje i typy danych)
- **algorytm**
- program

Miejsce

Miejsce algorytmu na przykładzie implementacji oprogramowania rozwiązującego postawiony problem.

- problem
- komputer (czas, wewnętrzna reprezentacja danych, oprogramowanie)
- język (dostępne konstrukcje i typy danych)
- **algorytm**
- program

Miejsce

Miejsce algorytmu na przykładzie implementacji oprogramowania rozwiązującego postawiony problem.

- problem
- komputer (czas, wewnętrzna reprezentacja danych, oprogramowanie)
- język (dostępne konstrukcje i typy danych)
- **algorytm**
- program

Miejsce

Miejsce algorytmu na przykładzie implementacji oprogramowania rozwiązującego postawiony problem.

- problem
- komputer (czas, wewnętrzna reprezentacja danych, oprogramowanie)
- język (dostępne konstrukcje i typy danych)
- **algorytm**
- program

Przetwarzane informacje

Ograniczona informacja

- Informacja przechowywan w komputerze i przez niego przetwarzana stanowi pewien niewielki wycinek rzeczywistości zawierający dane niezbędne do rozwiązania postawionego problemu.
- Musimy się zastanowić jakie informacje są nam niezbędne, jakie mogą pomóc a jakie są całkiem niepotrzebne.
- Musimy się zastanowić jak będziemy reprezentować wybrane przez nas informacje.

Ostatni punkt prowadzi nas do pojęcia *typu danych* (*struktury danych*).

Przetwarzane informacje

Ograniczona informacja

- Informacja przechowywan w komputerze i przez niego przetwarzana stanowi pewien niewielki wycinek rzeczywistości zawierający dane niezbędne do rozwiązania postawionego problemu.
- Musimy się zastanowić jakie informacje są nam niezbędne, jakie mogą pomóc a jakie są całkiem niepotrzebne.
- Musimy się zastanowić jak będziemy reprezentować wybrane przez nas informacje.

Ostatni punkt prowadzi nas do pojęcia *typu danych* (*struktury danych*).

Przetwarzane informacje

Ograniczona informacja

- Informacja przechowywan w komputerze i przez niego przetwarzana stanowi pewien niewielki wycinek rzeczywistości zawierający dane niezbędne do rozwiązania postawionego problemu.
- Musimy się zastanowić jakie informacje są nam niezbędne, jakie mogą pomóc a jakie są całkiem niepotrzebne.
- Musimy się zastanowić jak będziemy reprezentować wybrane przez nas informacje.

Ostatni punkt prowadzi nas do pojęcia *typu danych* (*struktury danych*).

Przetwarzane informacje

Ograniczona informacja

- Informacja przechowywan w komputerze i przez niego przetwarzana stanowi pewien niewielki wycinek rzeczywistości zawierający dane niezbędne do rozwiązania postawionego problemu.
- Musimy się zastanowić jakie informacje są nam niezbędne, jakie mogą pomóc a jakie są całkiem niepotrzebne.
- Musimy się zastanowić jak będziemy reprezentować wybrane przez nas informacje.

Ostatni punkt prowadzi nas do pojęcia *typu danych* (*struktury danych*).

Przetwarzane informacje

Ograniczona informacja

- Informacja przechowywan w komputerze i przez niego przetwarzana stanowi pewien niewielki wycinek rzeczywistości zawierający dane niezbędne do rozwiązania postawionego problemu.
- Musimy się zastanowić jakie informacje są nam niezbędne, jakie mogą pomóc a jakie są całkiem niepotrzebne.
- Musimy się zastanowić jak będziemy reprezentować wybrane przez nas informacje.

Ostatni punkt prowadzi nas do pojęcia *typu danych* (*struktury danych*).

Struktury danych

Struktura danych jest takim sposobem przechowywania danych w komputerze aby ułatwiać ich wykorzystanie. Często rozsądny wybór struktury danych pozwala na wykorzystanie efektywniejszych algorytmów.

Typ danej

Najpopularniejszy podział rozróżnia typy proste, nazywany też typami wbudowanymi i typy złożone — typy złożone z typów prostych.
Do typów prostych zaliczamy zwykle:

- typy liczbowe (np. całkowity, zmiennoprzecinkowy, stałoprzecinkowy)
- typ znakowy (znaki alfanumeryczne)
- typ logiczny

Do typów złożonych (nazywanych też strukturami) zaliczamy zwykle:

- tablicę/listę
- słownik
- zbiór
- rekord
- plik
- kolejkę
- stos
- drzewo

Typ danej

Najpopularniejszy podział rozróżnia typy proste, nazywany też typami wbudowanymi i typy złożone — typy złożone z typów prostych.
Do typów prostych zaliczamy zwykle:

- typy liczbowe (np. całkowity, zmiennoprzecinkowy, stałoprzecinkowy)
- typ znakowy (znaki alfanumeryczne)
- typ logiczny

Do typów złożonych (nazywanych też strukturami) zaliczamy zwykle:

- tablicę/listę
- słownik
- zbiór
- rekord
- plik
- kolejkę
- stos
- drzewo

Typ danej

Najpopularniejszy podział rozróżnia typy proste, nazywany też typami wbudowanymi i typy złożone — typy złożone z typów prostych.
Do typów prostych zaliczamy zwykle:

- typy liczbowe (np. całkowity, zmiennoprzecinkowy, stałoprzecinkowy)
- typ znakowy (znaki alfanumeryczne)
- typ logiczny

Do typów złożonych (nazywanych też strukturami) zaliczamy zwykle:

- tablicę/listę
- słownik
- zbiór
- rekord
- plik
- kolejkę
- stos
- drzewo

Typ danej

Najpopularniejszy podział rozróżnia typy proste, nazywany też typami wbudowanymi i typy złożone — typy złożone z typów prostych.
Do typów prostych zaliczamy zwykle:

- typy liczbowe (np. całkowity, zmiennoprzecinkowy, stałoprzecinkowy)
- typ znakowy (znaki alfanumeryczne)
- typ logiczny

Do typów złożonych (nazywanych też strukturami) zaliczamy zwykle:

- tablicę/listę
- słownik
- zbiór
- rekord
- plik
- kolejkę
- stos
- drzewo

Typ danej

Najpopularniejszy podział rozróżnia typy proste, nazywany też typami wbudowanymi i typy złożone — typy złożone z typów prostych.
Do typów prostych zaliczamy zwykle:

- typy liczbowe (np. całkowity, zmiennoprzecinkowy, stałoprzecinkowy)
- typ znakowy (znaki alfanumeryczne)
- typ logiczny

Do typów złożonych (nazywanych też strukturami) zaliczamy zwykle:

- tablicę/listę
- słownik
- zbiór
- rekord
- plik
- kolejkę
- stos
- drzewo

Typ danej

Najpopularniejszy podział rozróżnia typy proste, nazywany też typami wbudowanymi i typy złożone — typy złożone z typów prostych.
Do typów prostych zaliczamy zwykle:

- typy liczbowe (np. całkowity, zmiennoprzecinkowy, stałoprzecinkowy)
- typ znakowy (znaki alfanumeryczne)
- typ logiczny

Do typów złożonych (nazywanych też strukturami) zaliczamy zwykle:

- tablicę/listę
- słownik
- zbiór
- rekord
- plik
- kolejkę
- stos
- drzewo

Typ danej

Najpopularniejszy podział rozróżnia typy proste, nazywany też typami wbudowanymi i typy złożone — typy złożone z typów prostych.
Do typów prostych zaliczamy zwykle:

- typy liczbowe (np. całkowity, zmiennoprzecinkowy, stałoprzecinkowy)
- typ znakowy (znaki alfanumeryczne)
- typ logiczny

Do typów złożonych (nazywanych też strukturami) zaliczamy zwykle:

- tablicę/listę
- słownik
- zbiór
- rekord
- plik
- kolejkę
- stos
- drzewo

Typ danej

Najpopularniejszy podział rozróżnia typy proste, nazywany też typami wbudowanymi i typy złożone — typy złożone z typów prostych.
Do typów prostych zaliczamy zwykle:

- typy liczbowe (np. całkowity, zmiennoprzecinkowy, stałoprzecinkowy)
- typ znakowy (znaki alfanumeryczne)
- typ logiczny

Do typów złożonych (nazywanych też strukturami) zaliczamy zwykle:

- tablicę/listę
- słownik
- zbiór
- rekord
- plik
- kolejkę
- stos
- drzewo

Typ danej

Najpopularniejszy podział rozróżnia typy proste, nazywany też typami wbudowanymi i typy złożone — typy złożone z typów prostych.
Do typów prostych zaliczamy zwykle:

- typy liczbowe (np. całkowity, zmiennoprzecinkowy, stałoprzecinkowy)
- typ znakowy (znaki alfanumeryczne)
- typ logiczny

Do typów złożonych (nazywanych też strukturami) zaliczamy zwykle:

- tablicę/listę
- słownik
- zbiór
- rekord
- plik
- kolejkę
- stos
- drzewo

Typ danej

Najpopularniejszy podział rozróżnia typy proste, nazywany też typami wbudowanymi i typy złożone — typy złożone z typów prostych.
Do typów prostych zaliczamy zwykle:

- typy liczbowe (np. całkowity, zmiennoprzecinkowy, stałoprzecinkowy)
- typ znakowy (znaki alfanumeryczne)
- typ logiczny

Do typów złożonych (nazywanych też strukturami) zaliczamy zwykle:

- tablicę/listę
- słownik
- zbiór
- rekord
- plik
- kolejkę
- stos
- drzewo

Typ danej

Najpopularniejszy podział rozróżnia typy proste, nazywany też typami wbudowanymi i typy złożone — typy złożone z typów prostych.
Do typów prostych zaliczamy zwykle:

- typy liczbowe (np. całkowity, zmiennoprzecinkowy, stałoprzecinkowy)
- typ znakowy (znaki alfanumeryczne)
- typ logiczny

Do typów złożonych (nazywanych też strukturami) zaliczamy zwykle:

- tablicę/listę
- słownik
- zbiór
- rekord
- plik
- kolejkę
- stos
- drzewo

Typ danej

Najpopularniejszy podział rozróżnia typy proste, nazywany też typami wbudowanymi i typy złożone — typy złożone z typów prostych.
Do typów prostych zaliczamy zwykle:

- typy liczbowe (np. całkowity, zmiennoprzecinkowy, stałoprzecinkowy)
- typ znakowy (znaki alfanumeryczne)
- typ logiczny

Do typów złożonych (nazywanych też strukturami) zaliczamy zwykle:

- tablicę/listę
- słownik
- zbiór
- rekord
- plik
- kolejkę
- stos
- drzewo

```
mojaTablica is array[1..10] of Integer

mojaTablica[5] := 12

wypisz(mojaTablica[9])
```

```
type tablica is array[1..10] of Integer

tablica: mojaTablica_1, mojaTablica_2

mojaTablica_1[5] := 12

wypisz(mojaTablica_2[9])
```

Przykłady zapisu tablic w różnych językach

Ada:

```
-- definicja typu tablicowego
type TableType is array(1 .. 100) of Integer;
-- definicja zmiennej określonego typu tablicowego
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
```

Przykłady zapisu tablic w różnych językach

C:

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

Java:

```
int [] counts;
counts = new int[5];
```

PHP:

```
$pierwszy_kwartal = array(1 =>'Styczeń','Luty','Marzec');
```

Python:

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

Przykłady zapisu słownika

Python:

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

```
var mySet = new Set();

mySet.add(1);
mySet.add(5);
mySet.add("some text");
var o = {a: 1, b: 2};
mySet.add(o);

mySet.has(1); // true
mySet.has(3); // false, 3 has not been added to the set
mySet.has(5);                 // true
mySet.has(Math.sqrt(25));  // true
mySet.has("Some Text".toLowerCase()); // true
mySet.has(o); // true
mySet.size; // 4
mySet.delete(5); // removes 5 from the set
mySet.has(5);    // false, 5 has been removed
mySet.size; // 3, we just removed one value
```

Przykład ze strony https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set

```
silnik is array[1..100] of String
kola is array[1..100] of Integer
kierownica is array[1..100] of Boolean

silnik[17] := ...
kola[17] := ...
kierownica[17] := ...
```

```
silnik is array[1..1000] of String // max 10 silników na auto
kola is array[1..100] of Integer
kierownica is array[1..100} of Boolean

silnik[171] := ...
silnik[172] := ...
kola[17] := ...
kierownica[17] := ...
```

A może lepiej tak?

```
silnik is array[1..1000] of String // max 10 silników na auto
kola is array[1..100] of Integer
kierownica is array[1..100} of Boolean

silnik[17] := ...
silnik[27] := ...
kola[17] := ...
kierownica[17] := ...
```

A może jednak tak?

```
type auto is record
  silnik is array[1..10] of String // max 10 silników na auto
  Integer: kola
  Boolean: kierownica
end

auta is array[1..100] of auto

auta[17].silnik[1] := ...
auta[17].silnik[2] := ...
auta[17].kola := ...
auta[17].kierownica := ...
```

Tak będzie jeszcze czytelniej

```
type auto is record
  silnik is array[1..10] of String // max 10 silników na auto
  Integer: kola
  Boolean: kierownica
end

auta is array[1..100] of auto

autoTomka = auta[17]

autoTomka.silnik[1] := ...
autoTomka.silnik[2] := ...
autoTomka.kola := ...
autoTomka.kierownica := ...
```

```
type osoba is record
  String: imie
  String: nazwisko
  Integer: wiek
end

osoba: o

o.wiek = 12

wypisz(o.imie)
```

Struktura danych a realizacja algorytmu

Uważny i właściwy dobór struktury danych ma zasadnicze znaczenie zarówno dla realizacji algorytmu jak i rozwiązania postawionego zadania. Dobrze dobrana struktura powinna nam ułatwiać rozwiązanie problemu w maksymalnym możliwym stopniu. Czasem wręcz sama w sobie jest tym rozwiązaniem. Przykłady zobaczymy w dalszej części gdy zapoznamy się już z metodami opisu algorytmów.

Metody opisu algorytmów

- język naturalny
  - teoretycznie łatwy do napisania (wypunktowanie czynności)
  - często mała precyzja
  - kłopoty implementacyjne
- schemat blokowy
  - duża przejrzystość i czytelność
  - odzwierciedla strukturę algorytmu wyraźnie zaznaczając występowanie rozgałęzień (punktów decyzyjnych)
  - kłopoty implementacyjne
- pseudojęzyk
  - ułatwia implementację
  - mniejsza przejrzystość

Metody opisu algorytmów

- język naturalny
  - teoretycznie łatwy do napisania (wypunktowanie czynności)
  - często mała precyzja
  - kłopoty implementacyjne
- schemat blokowy
  - duża przejrzystość i czytelność
  - odzwierciedla strukturę algorytmu wyraźnie zaznaczając występowanie rozgałęzień (punktów decyzyjnych)
  - kłopoty implementacyjne
- pseudojęzyk
  - ułatwia implementację
  - mniejsza przejrzystość

Metody opisu algorytmów

- język naturalny
  - teoretycznie łatwy do napisania (wypunktowanie czynności)
  - często mała precyzja
  - kłopoty implementacyjne
- schemat blokowy
  - duża przejrzystość i czytelność
  - odzwierciedla strukturę algorytmu wyraźnie zaznaczając występowanie rozgałęzień (punktów decyzyjnych)
  - kłopoty implementacyjne
- pseudojęzyk
  - ułatwia implementację
  - mniejsza przejrzystość

Metody opisu algorytmów

- język naturalny
  - teoretycznie łatwy do napisania (wypunktowanie czynności)
  - często mała precyzja
  - kłopoty implementacyjne
- schemat blokowy
  - duża przejrzystość i czytelność
  - odzwierciedla strukturę algorytmu wyraźnie zaznaczając występowanie rozgałęzień (punktów decyzyjnych)
  - kłopoty implementacyjne
- pseudojęzyk
  - ułatwia implementację
  - mniejsza przejrzystość

Metody opisu algorytmów

Co potrzebujemy?

???

- ???
- ???
- ???

To może zajmijmy sie pierwszym sposobem. . .

???

- ???
- ???
- ???

To może zajmijmy sie pierwszym sposobem. . .

Algorytm Euklidesa

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

Algorytm Euklidesa

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

Algorytm Euklidesa

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

Algorytm Euklidesa

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

Algorytm Euklidesa

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

Algorytm Euklidesa

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

Algorytm Euklidesa

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

Elementy odnalezione na poprzednim slajdzie to

- ???
- ???
- ???

Symbole

- początek i koniec
- blok instrukcji
- decyzja/warunek
- łącznik
- zapis/odczyt

Symbole

- początek i koniec
- blok instrukcji
- decyzja/warunek
- łącznik
- zapis/odczyt

Symbole

- początek i koniec
- blok instrukcji
- decyzja/warunek
- łącznik
- zapis/odczyt

Symbole

- początek i koniec
- blok instrukcji
- decyzja/warunek
- łącznik
- zapis/odczyt

Symbole

- początek i koniec
- blok instrukcji
- decyzja/warunek
- łącznik
- zapis/odczyt

Symbole

- początek i koniec
- blok instrukcji
- decyzja/warunek
- łącznik
- zapis/odczyt

Reguły

1. bloki połączone są zorientowanymi liniami
2. wykonywane są wszystkie instrukcje w bloku albo żadna
3. dalsze operacje nie zależą od poprzednich, chyba że zależności zostały przekazane za pomocą danych
4. kolejność wykonania operacji jest ściśle określona przez zorientowane linie łączące poszczególne bloki
5. do każdego bloku prowadzi dokładnie jedna linia
6. linie mogą się łączyć, a punkt połączenia nazywa się punktem zbiegu

Reguły

1. bloki połączone są zorientowanymi liniami
2. wykonywane są wszystkie instrukcje w bloku albo żadna
3. dalsze operacje nie zależą od poprzednich, chyba że zależności zostały przekazane za pomocą danych
4. kolejność wykonania operacji jest ściśle określona przez zorientowane linie łączące poszczególne bloki
5. do każdego bloku prowadzi dokładnie jedna linia
6. linie mogą się łączyć, a punkt połączenia nazywa się punktem zbiegu

Reguły

1. bloki połączone są zorientowanymi liniami
2. wykonywane są wszystkie instrukcje w bloku albo żadna
3. dalsze operacje nie zależą od poprzednich, chyba że zależności zostały przekazane za pomocą danych
4. kolejność wykonania operacji jest ściśle określona przez zorientowane linie łączące poszczególne bloki
5. do każdego bloku prowadzi dokładnie jedna linia
6. linie mogą się łączyć, a punkt połączenia nazywa się punktem zbiegu

Reguły

1. bloki połączone są zorientowanymi liniami
2. wykonywane są wszystkie instrukcje w bloku albo żadna
3. dalsze operacje nie zależą od poprzednich, chyba że zależności zostały przekazane za pomocą danych
4. kolejność wykonania operacji jest ściśle określona przez zorientowane linie łączące poszczególne bloki
5. do każdego bloku prowadzi dokładnie jedna linia
6. linie mogą się łączyć, a punkt połączenia nazywa się punktem zbiegu

Reguły

1. bloki połączone są zorientowanymi liniami
2. wykonywane są wszystkie instrukcje w bloku albo żadna
3. dalsze operacje nie zależą od poprzednich, chyba że zależności zostały przekazane za pomocą danych
4. kolejność wykonania operacji jest ściśle określona przez zorientowane linie łączące poszczególne bloki
5. do każdego bloku prowadzi dokładnie jedna linia
6. linie mogą się łączyć, a punkt połączenia nazywa się punktem zbiegu

Reguły

1. bloki połączone są zorientowanymi liniami
2. wykonywane są wszystkie instrukcje w bloku albo żadna
3. dalsze operacje nie zależą od poprzednich, chyba że zależności zostały przekazane za pomocą danych
4. kolejność wykonania operacji jest ściśle określona przez zorientowane linie łączące poszczególne bloki
5. do każdego bloku prowadzi dokładnie jedna linia
6. linie mogą się łączyć, a punkt połączenia nazywa się punktem zbiegu

Reguły

1. bloki połączone są zorientowanymi liniami
2. wykonywane są wszystkie instrukcje w bloku albo żadna
3. dalsze operacje nie zależą od poprzednich, chyba że zależności zostały przekazane za pomocą danych
4. kolejność wykonania operacji jest ściśle określona przez zorientowane linie łączące poszczególne bloki
5. do każdego bloku prowadzi dokładnie jedna linia
6. linie mogą się łączyć, a punkt połączenia nazywa się punktem zbiegu

Schemat blokowy algorytmu Euklidesa.

Instrukcje

Pseudojęzyk nie ma ścisłej definicji. Zwykle formą zapisu przypomina jeden z wielu występujących dziś języków proceduralnych, stanowiąc mieszaninę konstukcji zapożyczonych z wielu z nich, jak na przykład C, Java, PHP czy Python. Szczegóły nie związane z algorytmem (np. zarządzanie pamięcią) zwykle są pomijane. Często bloki kodu, np. występującego w pętli, zastępowane są krótkim wyrażeniem języka naturalnego.

Na potrzeby zajęć przyjmujemy następujące formy zapisu.

- instrukcja podstawienia (przypisania, ang. *assignment statement*)

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

Instrukcje

Pseudojęzyk nie ma ścisłej definicji. Zwykle formą zapisu przypomina jeden z wielu występujących dziś języków proceduralnych, stanowiąc mieszaninę konstukcji zapożyczonych z wielu z nich, jak na przykład C, Java, PHP czy Python. Szczegóły nie związane z algorytmem (np. zarządzanie pamięcią) zwykle są pomijane. Często bloki kodu, np. występującego w pętli, zastępowane są krótkim wyrażeniem języka naturalnego.

Na potrzeby zajęć przyjmujemy następujące formy zapisu.

- instrukcja podstawienia (przypisania, ang. *assignment statement*)

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

Instrukcje

- blok (ang. *(code) block*), (blok podstawowy; ang. *basic block*)

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

Blok podstawowy to taki ciąg instrukcji, który ma tylko jedno wejście i tylko jedno wyjście. Wejście do bloku znajduje się na jego początku, a wyjście na końcu. **Instrukcje bloku podstawowego wykonują się wszystkie i w kolejności, w jakiej są zapisane.** Bloki są powiązane instrukcjami goto i każdy blok kończy się instrukcją, która przenosi sterowanie do innego bloku.

Instrukcje

- instrukcja warunkowa (ang. *conditional statements*, *conditional expressions*, *conditional constructs*)

```
if (WARUNEK) then
begin
  TRUE
end
```

```
if (WARUNEK) then
begin
  TRUE
end
else
begin
  FALSE
end
```

- `WARUNEK` — wyrażenie zwracające wartość logiczną, np

```
x=7        x>12        x!=18
x>12 and y<3        x=5 and (y=1 or z=2)
```

- `TRUE` (`FALSE`) — instrukcje wykonywane, gdy warunek jest prawdziwy (fałszywy)

Instrukcje

- instrukcja pętli `for` (ang. *for-loop*, *for loop*) (1/2)

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

Powyższa postać pętli to tak zwana **pętla iteracyjna** (**pętla licznikowa**) (ang. *count-controlled loops*).

Instrukcje

- instrukcja pętli `for` (2/2)

```
for i in NAZWA do
begin
  instrukcje
end
```

```
foreach i in NAZWA do
begin
  instrukcje
end
```

- `NAZWA` — nazwa zmiennej reprezentującej listę, słownik, kolejkę, zbiór itp.

Powyższa postać pętli to tak zwana **pętla foreach** (**pętla „po kolekcji”**) (ang. *collection-controlled loops*).

Instrukcje

- instrukcja pętli `do-while` i `while` (ang. *do while loop*, *while loop*)

```
do                          while (WARUNEK)
begin                       begin
  instrukcje                  instrukcje
end                         end
while (WARUNEK);
```

- `NAZWA` — nazwa zmiennej reprezentującej listę, słownik, kolejkę, zbiór itp.

Powyższa postać pętli to tak zwana **pętla repetycyjna** (**pętla warunkowa**) (ang. *condition-controlled loops*).

Funkcja

Funkcja (ang. *function*) jako czarna skrzynka wykonująca określone zadanie.

- wywołanie funkcji:

```
nazwaFunkcji(argumenty);
x:=funkcja(arg1, arg2, arg3);
```

- definicja funkcji (ciało funkcji):

```
function nazwaFunkcji(argumenty)
begin
  instrukcje
  return zwracanaWartosc;
end
```

Funkcja

Funkcja (ang. *function*) jako czarna skrzynka wykonująca określone zadanie.

- wywołanie funkcji:

```
nazwaFunkcji(argumenty);
x:=funkcja(arg1, arg2, arg3);
```

- definicja funkcji (ciało funkcji):

```
function nazwaFunkcji(argumenty)
begin
  instrukcje
  return zwracanaWartosc;
end
```

Funkcja

Funkcja (ang. *function*) jako czarna skrzynka wykonująca określone zadanie.

- wywołanie funkcji:

```
nazwaFunkcji(argumenty);
x:=funkcja(arg1, arg2, arg3);
```

- definicja funkcji (ciało funkcji):

```
function nazwaFunkcji(argumenty)
begin
  instrukcje
  return zwracanaWartosc;
end
```

Funkcja – wartość domyślna argumentu

```
nazwaFunkcji(arg1 = 12, arg2 = "text")
```

Możliwe wywołania

```
nazwaFunkcji(3,"test")
nazwaFunkcji(5)
nazwaFunkcji()
```

Funkcja – nazewnictwo

Konwencji związanych ze sposobem zapisywania nazw funkcji (a ogólnie mówiąc wszelkich „obiektów" jak np. zmiennych) jest kilka, ale my będziemy trzymać się chyba najpopularniejszej, tj. tzw. **camelCase**. CamelCase jest systemem zapisu ciągów tekstowych, w którym kolejne wyrazy pisane są łącznie, rozpoczynając każdy następny wielką literą (prócz pierwszego). Na przykład

```
jakasDziwnaNazwaFunkcji
czasMinuty
predkoscPojazduWKilometrachNaGodzine
```

Dbamy o to aby nazwy funkcji a także zmiennych i wszelkich innych „obiektów" oddawały to czym dany obiekt się zajmuje lub co reprezentuje.

Funkcja – zwracanie wartości

Zasadniczon przyjmujemy, że funkcja zwraca jedną wartość. Gdy chcemy zwrócić ich kilka, należy „upakować" je w jakiejś strukturze danych (tablica, lista, słownik, krotka (ang. *tuple*) lub struktura)
W Pythonie możliwy jest następujący zapis

```
def f(x):
  y0 = x + 1
  y1 = x * 3
  y2 = y0 ** y3

  return (y0, y1, y2)

...

y0, y1, y2 = f(2)
```

Funkcja – zwracanie wartości

Używając naszej składni dotyczącej zapisu funkcji napiszemy to samo jako

```
function f(x)
begin
  y0 := x + 1
  y1 := x * 3
  y2 := y0 ** y3

  return (y0, y1, y2)
end
...

y0, y1, y2 := f(2)
```

Funkcja – zwracanie wartości

```
type wynikT is record
  y0
  y1
  y2
end

function f(x)
begin
  wynikT: wynik

  wynik.y0 := x + 1
  wynik.y1 := x * 3
  wynik.y2 := y0 ** y3

  return wynik
end
```

A zatem posługując się przedstawionym pseudojęzykiem, algorym Euklidesa możemy zapisać jak pokazano poniżej

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

  return a
end
```

A zatem posługując się przedstawionym pseudojęzykiem, algorym Euklidesa możemy zapisać jak pokazano poniżej

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

  return a
end
```

Wykorzystując inny pseudojęzyk otrzymamy inny zapis

Algorithm 1 Algorytm Euklidesa przedstawiony za pomocą pseudokodu (pseudojęzyka)

```
1:  function euklides(a, b)
2:      while (b! = 0) do
3:          if a > b then
4:              a := a-b
5:          else
6:              b := b-a
7:          end if
8:      end while
9:      return a
10: end function
```

The terms parameter and argument are sometimes used interchangeably, and the context is used to distinguish the meaning. The term **parameter** (sometimes called formal parameter) is often used to refer to the variable as found in the function definition, while **argument** (sometimes called actual parameter) refers to the actual input passed. For example, in the function definition $f(x) = 2 \cdot x$ the variable x is a parameter; in the function call $f(2)$ the value 2 is the argument of the function. Loosely, a **parameter is a type**, and an **argument is an instance**.

Please keep in mind that for example in the classical *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie in section *5.10 Command-line Arguments* we can find code like this

```
#include <stdio.h>

main(int argc, char *argv[])
{
...
}
```

where we can find not `paramc` but `argc` and not `paramv` but `argv`.

Naming the first parameter `argc` is correct, rather than `paramc`. At run time the value we use is an argument. We reserve the term parameter for when discussing subroutine definitions.
According to authors explanation of this part of code: *When `main` is called, it is called with two arguments. The first (conventionally called `argc`, for argument count) is the number of command-line arguments the program was invoked with; the second (`argv`, for argument vector) is a pointer to an array of character strings that contain the arguments, one per string.*

Formalnie rzecz biorąc, odpowiedź na postawione pytanie jest jedna: tak. Składnia, czy to języka rzeczywistego czy pseudojęzyka, jest nienaruszalną umową. Powyżej opisaliśmy pewną umowę co do zapisu. Ze względów czysto praktycznych (poprawa czytelności kodu na tablicy podczas ćwiczeń) dopuszczalne jest użycie nawiasów klamrowych `{` oraz `}` zamiast, odpowiednio, `begin` oraz `end` (wyjatek: para `begin-end` obejmująca ciało programu/funkcji). **Nie wolno natomiast ich mieszać.**

```
instrukcja          instrukcja {          instrukcja
begin               ...                   {
...                 ...                   ...
end                 }                     }
```

Metody opisu algorytmów

Pseudojęzyk — czy muszę pisać ciągle `begin` i `end`?

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

  return a
end
```

```
function euklides(a, b)
{
  while (b!=0)
  {
    if (a>b) then
    {
      a := a-b
    }
    else
    {
      b := b-a
    }
  }

  return a
}
```

Metody opisu algorytmów

Pseudojęzyk — czy muszę pisać ciągle `begin` i `end`?

```
function euklides(a, b)
{
  while (b!=0)
  {
    if (a>b) then
    {
      a := a-b
    }
    else
    {
      b := b-a
    }
  }

  return a
}
```

```
function euklides(a, b) {
  while (b!=0) {
    if (a>b) then {
      a := a-b
    } else {
      b := b-a
    }

    return a
  }
}
```

Konstrukcja if-elseif-elseif-else może zostać zastąpiona bardziej zwartą switch-case

```
if (decision = 1) then {
    action1()
}
else if (decision = 2) then {
    action2()
}
else if (decision = 3) then {
    action3();
}
```

```
switch (decision) {
    case 1:
        action1()
        break
    case 2:
        action2()
        break
    case 3:
         action3()
         break
}
```

Przy tej okazji zauważmy, że równoważne sobie, na poziomie pseudokodu, instrukcje lub ich ciąg nie muszą być sobie równoważne na poziomie kodu maszynowego.

```
compare value, 1
    jump_if_zero label1
    compare value, 2
    jump_if_zero label2
    compare value, 3
    jump_if_zero label3
label1:
    call action1
label2:
    call action2
label3:
    call action3
```

```
add program_counter, value
     call action1
     call action2
     call action3
```

xxx

xxx

Opis zadania

Dany jest graf `G`. Wszystkie węzły grafu są ponumerowane, dzięki czemu możemy mówić np. o wierzchołku 5. Możemy też powiedzieć jakie wierzchołki sąsiadują z zadanym wierzchołkiem podając posortowaną od najmniejszej wartości tablicę (a wygodniej kolejkę) wierzchołków. W związku z tym na grafie `G` możemy wykonać następujące operacje

1. pobrać tablicę (kolejkę) wierzchołków sąsiednich dla zadanego wierzchołka `n`

   ```
   G.getNeighbours(n)
   ```

Zadanie polega na sprawdzeniu, czy istnieje możliwość przejścia od zadanego wierzchołka `nStart` do zadanego wierzchołka końcowego `nStop`.

Algorytm

Dane wejściowe:

- graf `G`
- wierzchołek początkowy `nStart`
- wierzchołek końcowy `nStop`

Wynik działania:

- zwrócenie wartości logicznej `TRUE` gdy istnieje w grafie `G` możliwość przejścia od zadanego wierzchołka `nStart` do zadanego wierzchołka końcowego `nStop`; gdy taka ścieżka nie istnieje, zwrócenie wartości `FALSE`

Algorytm

Zmienne użyte w programie:

- `dataStructure` – pewna struktura danych, na której możemy wykonać następujące operacje
  1. pobrać wierzchołek ze struktury

     `dataStructure.pop()`
  2. dodać wierzchołek `n` do struktury

     `dataStructure.add(n)`
  3. sprawdzić czy wierzchołek `n` jest już w strukturze

     `dataStructure.exist(n)`
- `explored` – będzie wyjaśnione na wykładzie

```
 1 function graphSearch(nStart, nStop, G) return TRUE or FALSE
 2 begin
 3   node := nStart
 4   if (node = nStop) then
 5     return TRUE
 6   dataStructure := ???
 7   dataStructure.add(nStart)
 8   explored := an empty set
 9   loop {
10     if (the dataStructure is empty) then return FALSE
11     node := dataStructure.pop()
12
13     explored.add(node)
14
15     for each child in G.getNeighbours(node) {
16       if (child is not in explored or dataStructure) then {
17         if (child = nStop) then {
18           return TRUE
19         }
20         dataStructure.add(child)
21       }
22     } // end foreach
23   } // end main loop
24 end
```

Algorithm 2 Algorithm + datastructure

```
 1: function graphSearch(nStart, nStop, G)
 2:     node := nStart
 3:     if node = nStop then
 4:         return TRUE
 5:     end if
 6:     dataStructure :=???
 7:     dataStructure.add(nStart)
 8:     explored := an empty set
 9:     loop
10:         if the dataStructure is empty then
11:             return FALSE
12:         end if
13:         node := dataStructure.pop()
14:         explored.add(node)
15:         for all child  in G.getNeighbours(node) do
16:             if child is not in explored or dataStructure then
17:                 if child = nStop then
18:                     return TRUE
19:                 end if
20:                 dataStructure.add(child)
21:             end if
22:         end for
23:     end loop
24: end function
```

Opowieść o wywoływaniach funkcji

- Program bez funkcji
- Program z dwukrotnym wywołaniem tej samej funkcji
- Program z dwukrotnym wywołaniem tej samej funkcji, która wywołuje kolejną funkcję
- A jak to wygląda w assemblerze

Iteracja i rekurencja

Iteracja

Iteracja (łac. *iteratio*) to czynność powtarzania (najczęściej wielokrotnego) tej samej instrukcji (albo wielu instrukcji) w pętli.

Rekurencja

Rekurencja (ang. *recursion*, z łac. *recurrere*, przybiec z powrotem) to w logice, programowaniu i w matematyce odwoływanie się np. funkcji lub definicji do samej siebie.

Iteracja i rekurencja

Iteracja

Iteracja (łac. *iteratio*) to czynność powtarzania (najczęściej wielokrotnego) tej samej instrukcji (albo wielu instrukcji) w pętli.

Rekurencja

Rekurencja (ang. *recursion*, z łac. *recurrere*, przybiec z powrotem) to w logice, programowaniu i w matematyce odwoływanie się np. funkcji lub definicji do samej siebie.

Iteracja i rekurencja

Silnia iteracyjnie

```
n! = 1 * 2 * 3 * ... * n
```

Silnia rekurencyjnie

```
n! = n * (n-1)!
```

Silnia

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

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

Iteracja i rekurencja

Silnia iteracyjnie

```
n! = 1 * 2 * 3 * ... * n
```

Silnia rekurencyjnie

```
n! = n * (n-1)!
```

Silnia

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

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

Iteracja i rekurencja

Silnia iteracyjnie

```
n! = 1 * 2 * 3 * ... * n
```

Silnia rekurencyjnie

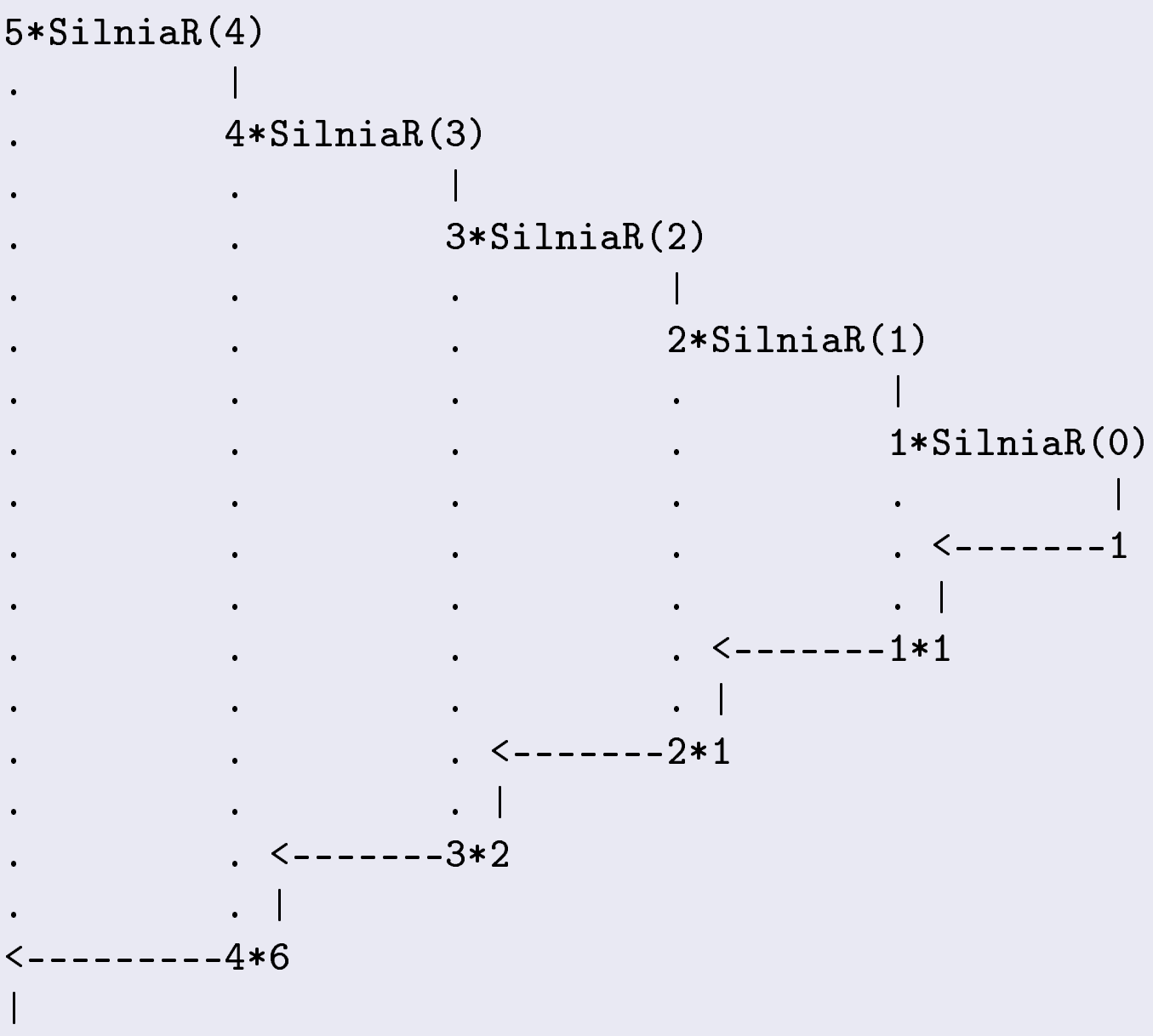
```
n! = n * (n-1)!
```

Silnia

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

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

Drzewo wywołań rekurencyjnych podczas obliczania wartości 4!



Definicja ciągu Fibonacciego

Dla $n > 1$ mamy

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

natomiast wyrazy 1. i 0. przyjmują wartość 1.

Fibonacci rekurencyjnie

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

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

Definicja ciągu Fibonacciego

Dla $n > 1$ mamy

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

natomiast wyrazy 1. i 0. przyjmują wartość 1.

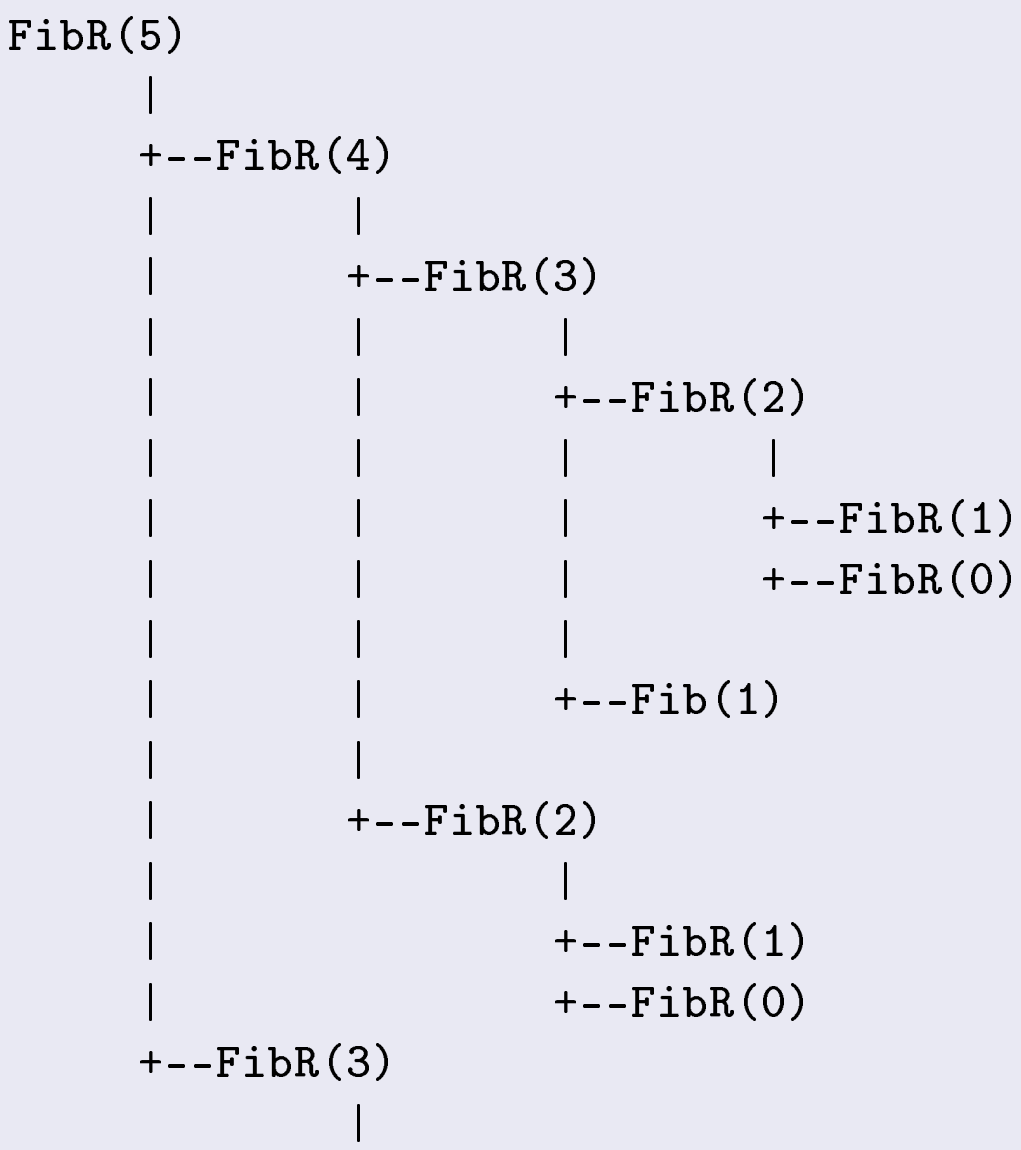
Fibonacci rekurencyjnie

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

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

Czas

Drzewo wywołań rekurencyjnych podczas obliczania 5. wyrazu ciągu Fibonacciego



Liczba wywołań

0	1	0
1	1	0
2	3	2
3	5	2
4	9	4
5	15	6
6	25	10
7	41	16
8	67	26
9	109	42
10	177	68
15	1973	xxx
20	21891	xxx
25	242785	xxx
30	2692537	xxx

Fibonacci iteracyjnie

```
function FibI(n)
begin
  tmp :=0 // zmienna tymczasowa (pomocnicza)
  x:=1; // wyraz n-1
  y:=1; // wyraz n-2

  for i:=1 to n-1 step 1
  begin
    tmp:=y; // zapamiętaj wyraz n-2
    y:=y+x; // przesuń wyraz n-2 na kolejną wartość ciągu
    x:=tmp; // przesuń wyraz n-1 na kolejną wartość ciągu
            // czyli na wartość wyrazu n-1 przed jego
            // przesunięciem
  end

  return x;
end
```