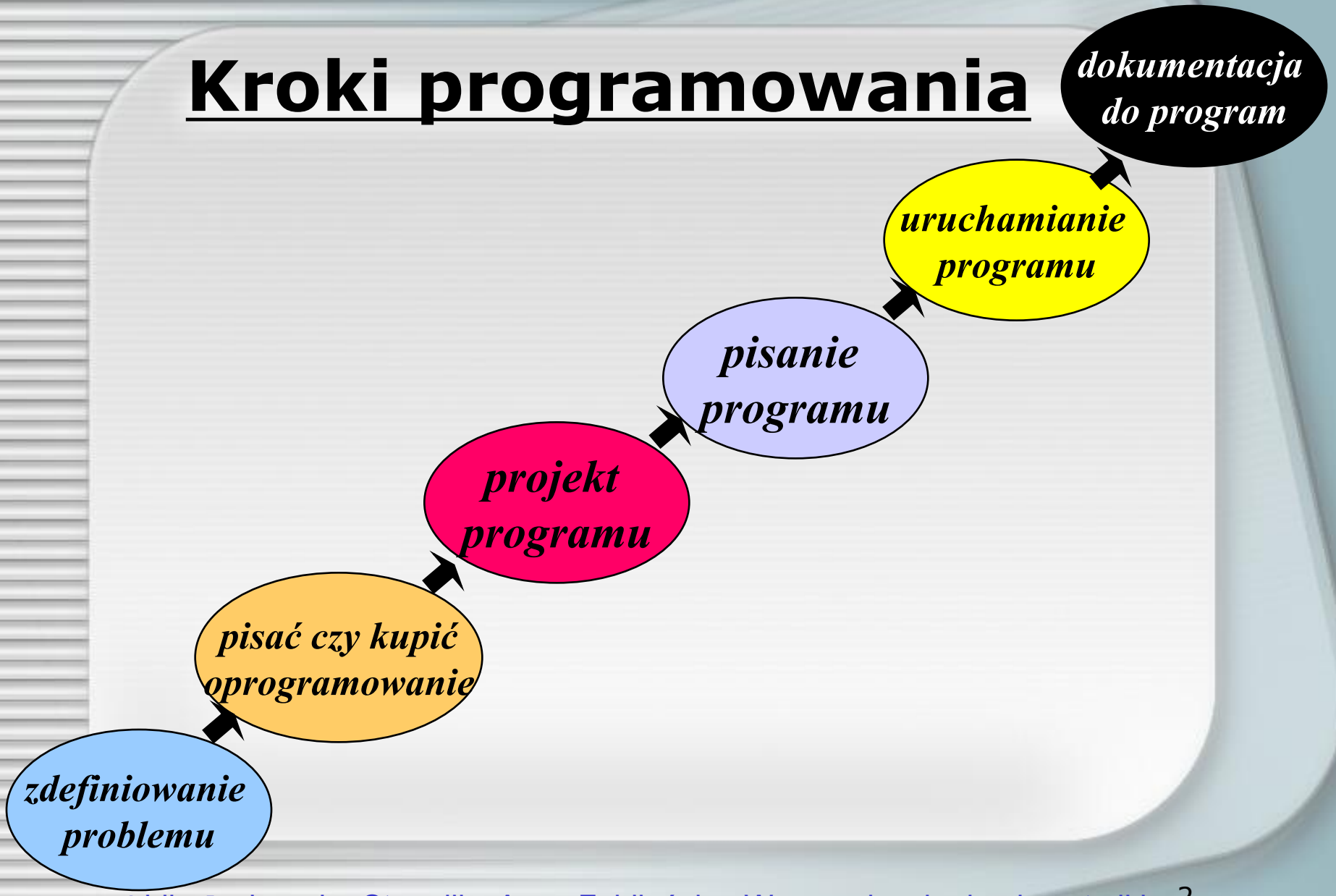
WPROWADZENIE DO ALGORYTMIKI

dr hab. inż. Lidia Jackowska-Strumiłło, prof. PŁ

dr hab. inż. Anna Fabijańska, prof. PŁ

Kroki programowania



Etapy budowy systemu informatycznego dla przedsiębiorstwa

1 *To, co klient zamówił*

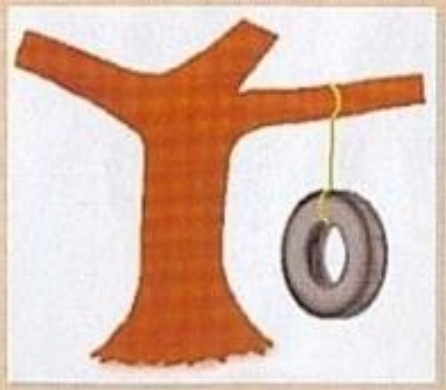
2 *To, co analityk zrozumiał.*

3 *To, co opisywał projekt.*

4 *To, co wykonali programiści.*

5 *Projekt po uruchomieniu i wdrożeniu.*

6 *To, za co klient zapłacił.*



7 *A to, czego klient potrzebował*

8 *Praktyczne zastosowanie projektu.*

Algorytm

Algorytm jest to sposób rozwiązania zagadnienia, podany w formie przepisu określającego skończoną liczbę operacji oraz kolejność w jakiej operacje te powinny być wykonywane.

Algorytm jest podstawowym pojęciem informatyki.

Cechy algorytmu

- **Uniwersalność** – zapewnienie rozwiązania każdego zadania należącego do określonego typu zadań.
- **Jednoznaczność** – prezentacja metody postępowania w postaci skończonej listy prostych i jednoznacznych rozkazów.
- **Zbieżność** – dla każdego dopuszczalnego zbioru danych początkowych liczba operacji prowadzących do poszukiwanego wyniku jest skończona.
- **Powtarzalność** – dla analogicznych danych algorytm, uzyska analogiczne wyniki.

Metody zapisu algorytmów

- **Zapis słowny** w postaci ciągu kroków (języka potocznego)
- **Zapis w notacji matematycznej**
- **Zapis w postaci graficznej** (schemat blokowy)
- **Zapis w języku symbolicznym** (pseudokod)
- **Zapis w języku programowania** (program)
- **Zapis w języku formalnym** (np. UML)
- **Strukturogramy**

Zapis słowny

- Najbardziej rozpowszechniony w życiu codziennym.
- **Polega na logicznym i zrozumiałym przedstawieniu listy kolejnych kroków, które należy wykonać, aby osiągnąć efekt**.
- Zalety: prostota, bez konstrukcji formalnych, szeroki sposób możliwego do użycia słownictwa.
- Wady: mała precyzja opisu, możliwość błędnej interpretacji, mała zwięzłość.

Przykład opisu słownego

Oblicz czas pracy dla pracownika A

1. Wyzeruj godziny pracy i godziny nadliczbowe.
2. Pobierz czas rozpoczęcia pracy i jej zakończenia.
3. Jeśli pracował po godzinie 17:00, oblicz godziny nadliczbowe.
4. Oblicz godziny pracy.
5. Dodaj godziny pracy do całkowitej liczby godzin pracy.
6. Dodaj godziny pracy nadliczbowej do całkowitej liczby godzin pracy nadliczbowej.
7. Jeśli są jeszcze kolejne godziny pracy tego pracownika, powróć do punktu 2 i powtórz kroki (2-7).

Pseudokod

Pseudokod to sposób zapisu algorytmu, który zachowując strukturę charakterystyczną dla kodu zapisanego w języku programowania, rezygnuje ze ścisłych reguł składniowych na rzecz prostoty i czytelności.

Pseudokod nie zawiera szczegółów implementacyjnych. Często też pomija opis działania podprocedur, zaś nietrywialne kroki algorytmu opisywane są z pomocą formuł matematycznych lub zdań w języku naturalnym.

Pseudokod

Zadanie proste:

przypisz średniej **wartość** zero
czytaj x
pisz wynik

Zadanie decyzyjne

jeżeli warunek **to** zadanie
jeżeli warunek1 **to** zadanie1
w przeciwnym przypadku zadanie2

zadanie iteracyjne podczas gdy

podczas gdy warunek **wykonuj** zadanie

Pseudokod

Zadanie iteracyjne powtarzaj
    powtarzaj zadanie **aż** warunek

Zadanie iteracyjne dla
    dla lista sytuacji **wykonuj** zadanie

zadanie wybierz
    wybierz przełącznik **z**
    wartość1: zadanie1
    ...
    w innym przypadku zadanie domyślne

Zadanie grupujące {}
    {zadanie1 zadanie2 zadanie3}

Notacja matematyczna

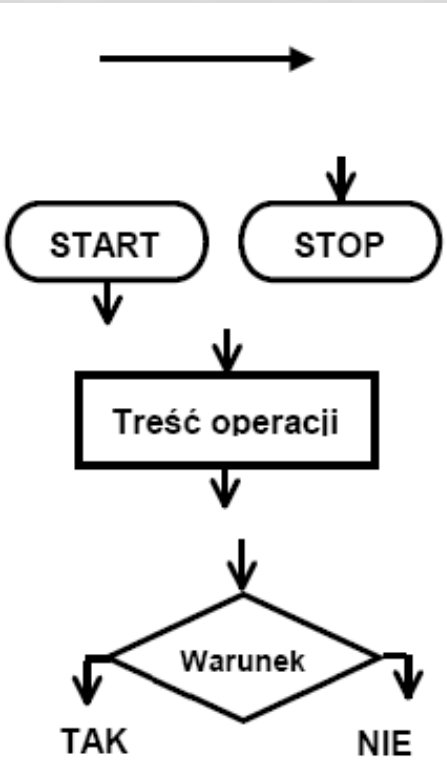
- Wykorzystuje symbole matematyczne
- Oszczędna i precyzyjna
- Użyteczna dla algorytmów przeznaczonych do obliczania wartości wyrażenia

$$\sum_{i=1}^{100} i^2$$

Zapis graficzny

- **Schemat blokowy** (ang. *flowchart, block diagram, block scheme, flow diagram*).
- **Przedstawia algorytm w postaci graficznej z wykorzystaniem bloków operacyjnych oraz wskazaniem przepływu sterowania.**
- Zalety: formalizuje zapis algorytmów.
- Wady: problem z przedstawieniem dużych algorytmów.

Symbole graficznej prezentacji algorytmów



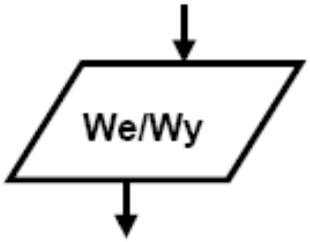
Poszczególne elementy schematu łączy się za pomocą strzałek

Skrzynki START i STOP wskazują początek i koniec każdego algorytmu.

W skrzynce instrukcyjnej umieszcza się polecenia do wykonania (instrukcje) - podstawienie, obliczenie, wprowadzenie wartości.

W skrzynce warunkowej umieszcza się warunek, który decyduje o wyborze dalszej drogi postępowania. Ze skrzynki wychodzą dwa połączenia: TAK (wybierane, gdy warunek jest spełniony), NIE (gdy warunek nie jest spełniony).

Symbole graficznej prezentacji algorytmów

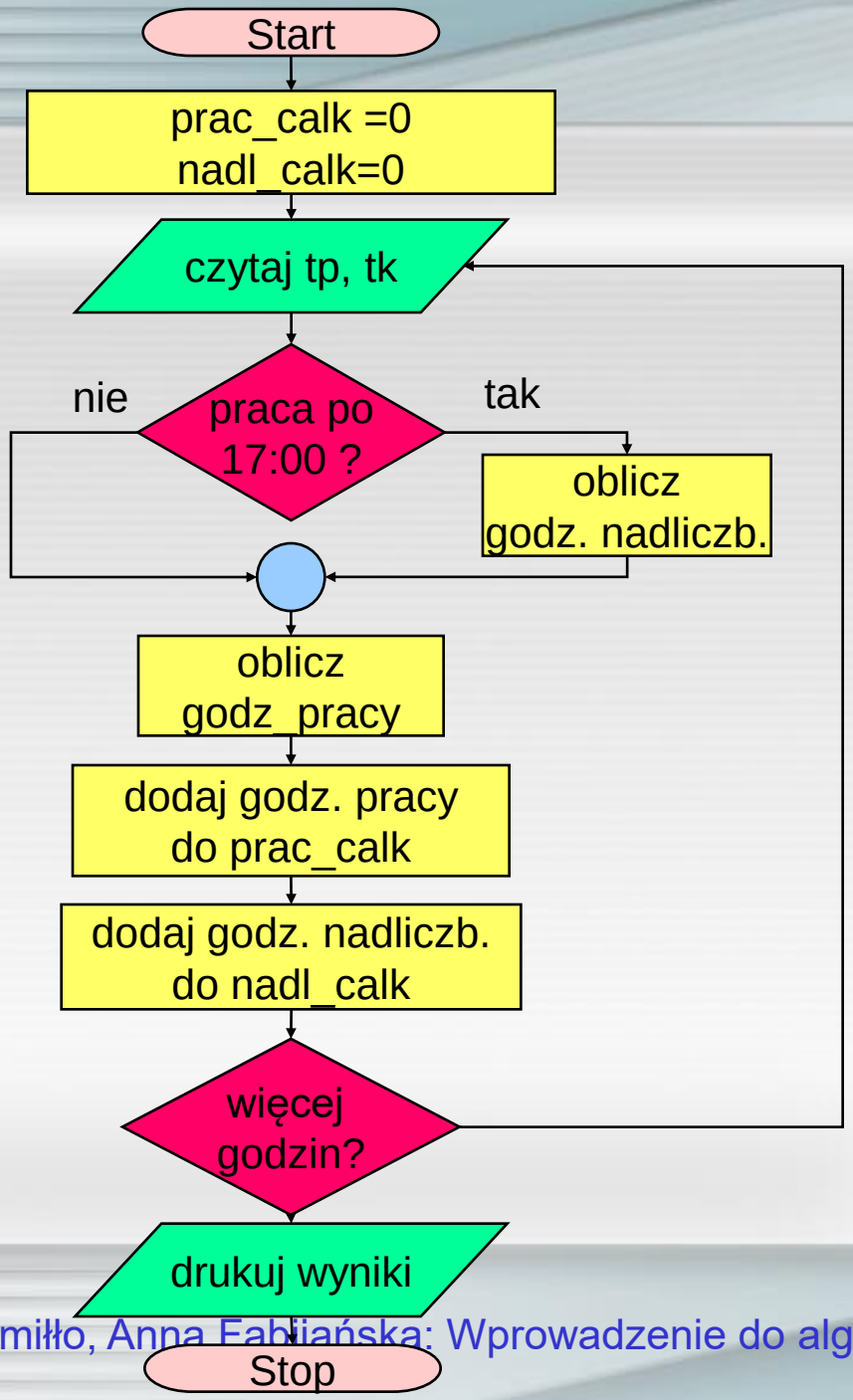


W skrzynce wejścia/wyjścia umieszcza się wprowadzane dane lub wyprowadzane wyniki.

Figura symbolizuje proces, który został już kiedyś zdefiniowany.

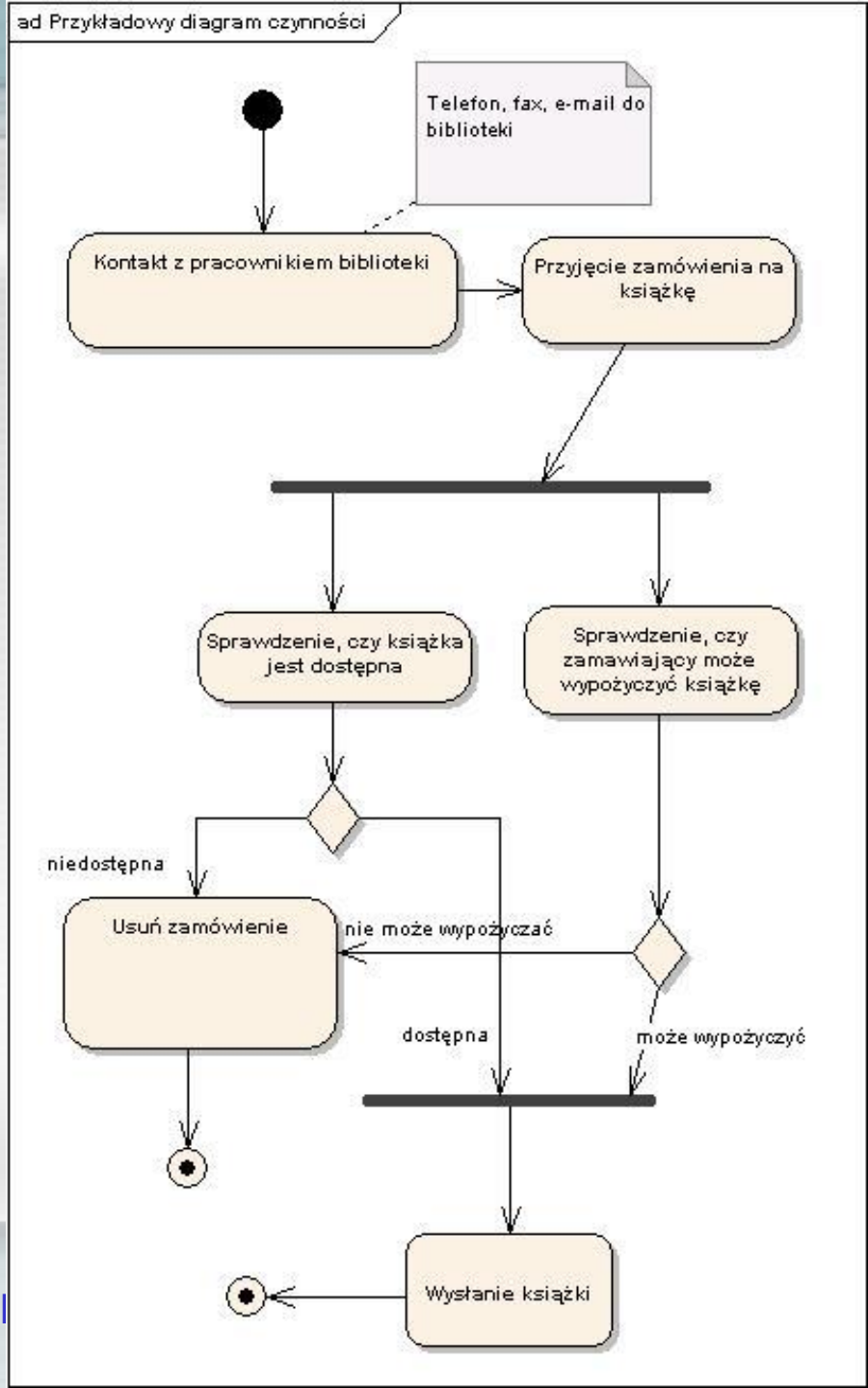
Koło symbolizuje tzw. łącznik stronicowy.

Symbol łącznika między stronicowego.

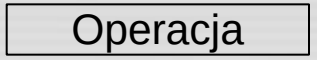

Start
prac_calk =0
nadl_calk=0
czytaj tp, tk
nie
praca po 17:00 ?
tak
oblicz godz. nadliczb.
oblicz godz_pracy
dodaj godz. pracy do prac_calk
dodaj godz. nadliczb. do nadl_calk
więcej godzin?
drukuj wyniki
Stop

Język UML

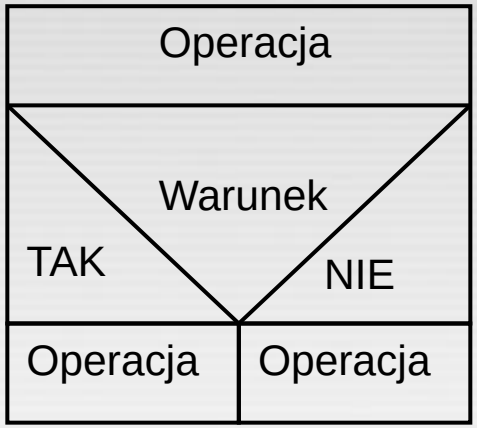
- Diagramy stanów



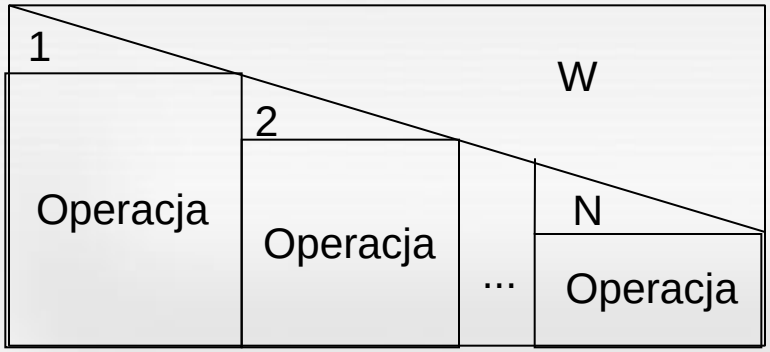
Strukturogramy



Blok operacyjny

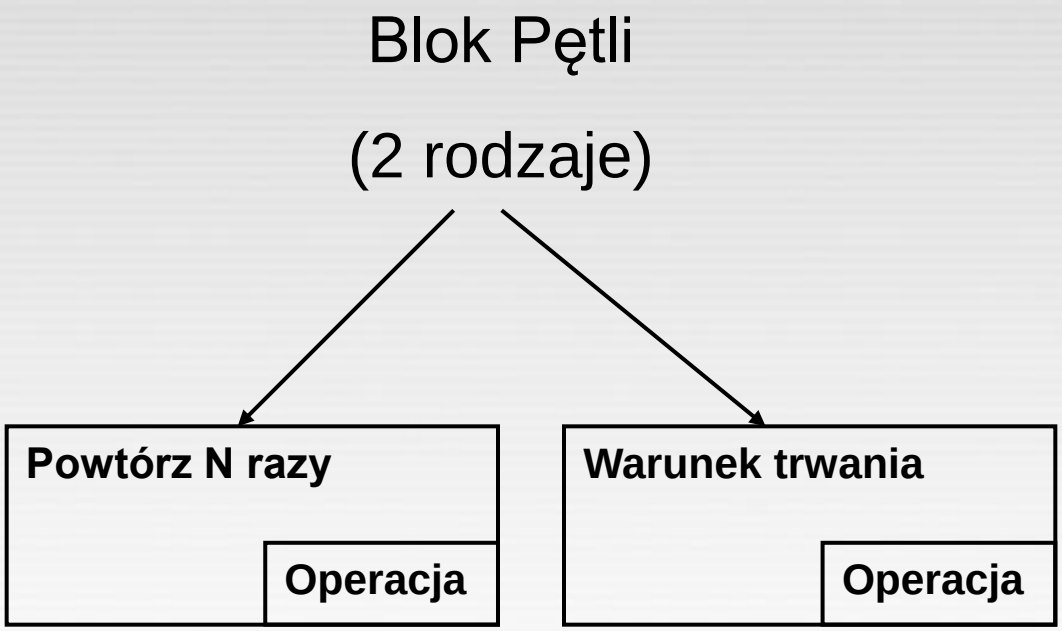


Blok warunkowy



Blok wielokrotnego wyboru

Strukturogramy



Strukturogramy

Implementacja

Zapis (sposób zapisu) algorytmu w konkretnym języku programowania.

Zapis w języku programowania

- Zapis formalny
- Bardzo precyzyjny
- Mało przejrzysty
- Zrozumiały wyłącznie dla programisty

Kodowanie programu

Pascal

```
Begin
prac_calk := 0;
nadl_calk := 0;
while not eof(plik) do
    begin
    readln(plik, g_pocz, min_pocz, g_kon, min_kon); {odczyt z pliku}
    if(g_kon>=17) then
        nadliczbowe := (g_kon - 17) + (min_kon/60)
    else
        nadliczbowe := 0;
    g_pracy := (g_kon - g_pocz) + (min_kon - min_pocz)/60
                                                    -nadliczbowe;
    prac_calk := prac_calk + g_pracy;
    nadl_calk := nadl_calk + nadliczbowe;
    end;
End.
```

Pascal

```
Program Czas_pracy;
var     plik : text; {zmienna plikowa typu tekst}
        prac_calk, nadl_calk, g_pracy, nadliczbowe : real;
        g_pocz, min_pocz, g_kon, min_kon : integer;
Begin
assign (plik,'time.txt'); {skojarzenie zmiennej plikowej z plikiem na dysku}
reset (plik); {otwarcie pliku do czytania}
prac_calk := 0;     nadl_calk := 0;
while not eof(plik) do
    begin
    readln(plik, g_pocz, min_pocz, g_kon, min_kon); {odczyt z pliku}
    if(g_kon>=17) then nadliczbowe:=(g_kon - 17)+(min_kon/60)
                          else nadliczbowe:= 0;
    g_pracy:=(g_kon-g_pocz)+(min_kon-min_pocz)/60-nadliczbowe;
    prac_calk:=prac_calk+g_pracy;
    nadl_calk:= nadl_calk+nadliczbowe;
    end;
close(plik); {zamknięcie pliku}
writeln('Godziny pracy = ', prac_calk); {wydruk wyników na ekranie}
writeln('Godziny nadliczbowe = ', nadl_calk);
End.
```

C

```c
#include <stdio.h>                    //plik czas_pracy.c
int main(){
 FILE *f;
 char znak;
 int g_pocz, min_pocz, g_kon, min_kon;
 float prac_calk, nadl_calk, g_pracy, nadliczbowe;
 f=fopen("time.txt","r");  //otwarcie pliku do czytania
 if(f!=NULL) {  //jesli plik istnieje
  prac_calk = 0; nadl_calk = 0;
  while(znak!=EOF) {
   fscanf(f,"%d %d %d %d", &g_pocz, &min_pocz, &g_kon, &min_kon); //odczyt
   if(g_kon>=17) nadliczbowe = (g_kon - 17) + (min_kon/60.0);
   else nadliczbowe = 0;
   g_pracy = (g_kon - g_pocz) + (min_kon - min_pocz)/60.0 - nadliczbowe;
   prac_calk = prac_calk + g_pracy;
   nadl_calk = nadl_calk + nadliczbowe;
   znak=fgetc(f);
  }
  fclose(f); //zamknięcie pliku
  printf("Godziny pracy = %f, nadliczbowe = %f\n", prac_calk, nadl_calk);
 }
 else printf("Blad odczytu  z pliku\n");
 return 0;
}
```

Visual Basic

```
Imports System.IO  ‘dolaczenie biblioteki we/wy
Module compute_time
   Sub Main()
      Dim g_pocz, min_pocz, g_kon, min_kon As Integer  ‘deklaracja zmiennych
      Dim prac_calk, nadl_calk, g_pracy, nadliczbowe As Single
      Try ‘wyjatki
        Dim sr As StreamReader = New StreamReader("time.txt")  ‘otwarcie pliku
         prac_calk = 0
         nadl_calk = 0
         Do While sr.Peek() >= 0  ‘petla Do While
            Dim tab(4) As String  ‘deklaracja tablicy zawierajacej 4 elementy typu string
            tab = sr.ReadLine().Split(" ")  ‘odczyt danych z pliku i zapisanie ich w tablicy
            g_pocz = Val(tab(0))  ‘konwersja danych z formatu string do integer
            min_pocz = Val(tab(1))
            g_kon = Val(tab(2))
            min_kon = Val(tab(3))
            If (g_kon >= 17) Then nadliczbowe = (g_kon - 17) + (min_kon / 60.0)
                                 Else  nadliczbowe = 0
            End If
            g_pracy = (g_kon - g_pocz) + (min_kon - min_pocz) / 60.0 - nadliczbowe
            prac_calk = prac_calk + g_pracy
            nadl_calk = nadl_calk + nadliczbowe
         Loop ‘koniec petli Do While
         sr.Close() ‘zamkniecie pliku
         Console.WriteLine("Godziny pracy = " & vbTab & "{0,3}", prac_calk) ‘wydruk wynikow na ekranie
         Console.WriteLine("Godziny nadliczbowe = " & vbTab & "{0,3}", nadl_calk)
      Catch e As Exception ‘deklaracja wyjatku e
         Console.WriteLine(„File reading error: {0}", e.ToString()) ‘obsluga wyjatku
      End Try
   End Sub
End Module
```

Techniki programowania

- **Programowanie wstępujące**

  (ang. *bottom-up programming*)

  » rozwiązywanie problemu za pomocą wydzielonych prostszych zadań

  » synteza - od szczegółu do ogółu

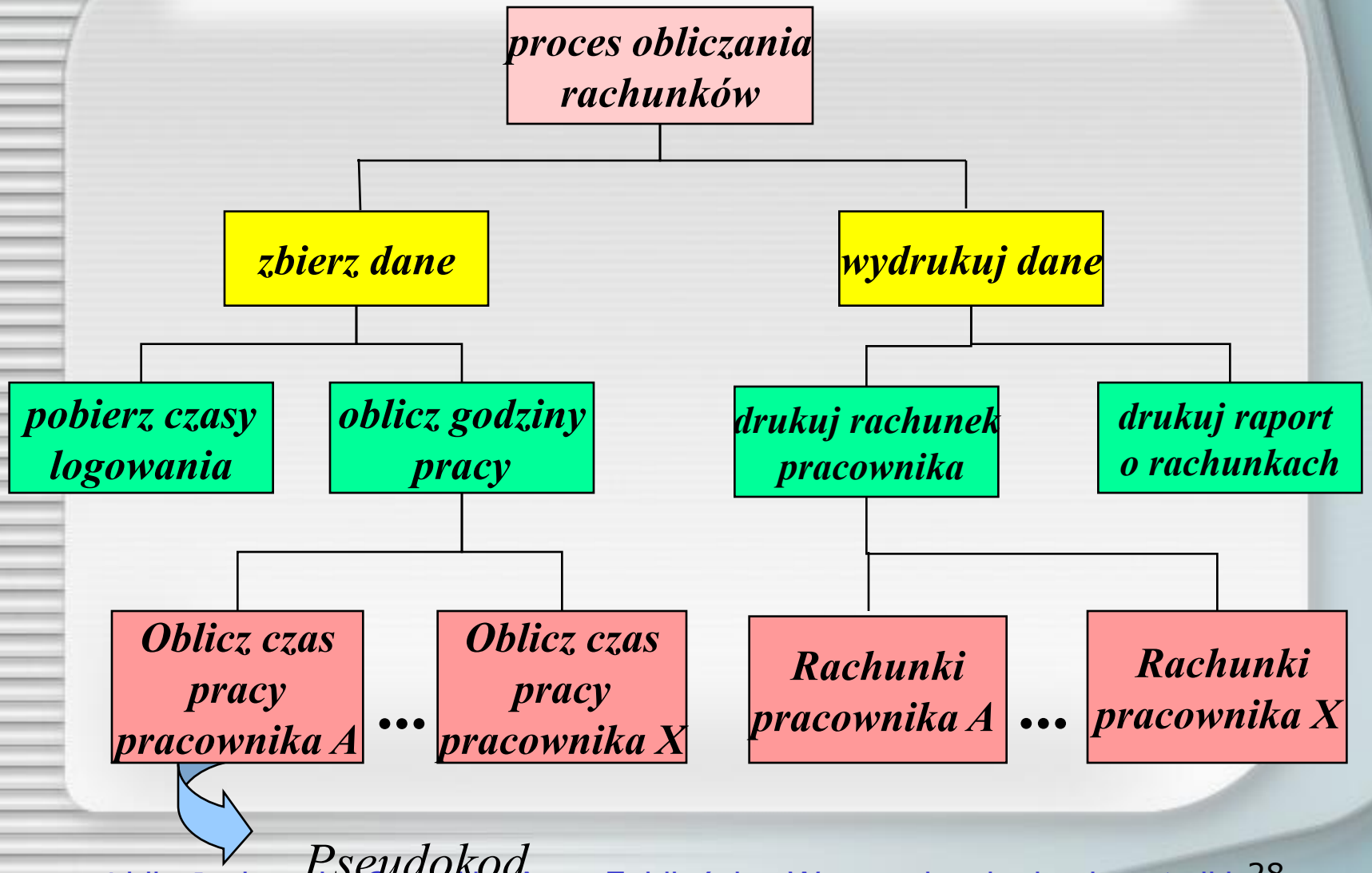
- **Programowanie zstępujące**

  (ang. *top-down programming*)

  » rozwiązywanie problemu w jednym ogólnym, dobrze przemyślanym planie

  » dekompozycja - od ogółu do szczegółu

Projektowanie programu metodą Top-down



Uruchamianie programu

- Sprawdzanie poprawności działania programu
- Poprawianie błędów
- Testowanie programu metodą wstępującą (ang. *bottom-up*) dla reprezentatywnych danych testowych

Poprawność algorytmu

Dowód poprawności algorytmu jest rozumowaniem matematycznym, prowadzącym do formalnego wykazania, że dany algorytm przy poprawnych danych wejściowych da nam wynik spełniający wymagania.

Algorytm definicja

Zadanie algorytmiczne można scharakteryzować zwięźle, jako złożone z:

1. Zbioru **P** dopuszczalnych danych wejściowych
2. Zależności **R** między danymi a żądanymi wynikami **Q**

$$\mathbf{R}: \mathbf{P} \rightarrow \mathbf{Q}$$

Weryfikacja poprawności algorytmu

- **Specyfikacja** wyraża, co algorytm ma robić, i określa związek miedzy jego danymi wejściowymi oraz wyjściowymi.
- **Weryfikacja** poprawności algorytmu opiera się na jego specyfikacji, która jest czymś niezależnym od kodu programu realizującego algorytm.

Poprawność częściowa

- Algorytm **A** jest częściowo poprawny względem **P** i **R**, gdy dla każdego zestawu danych **X** z **P**, jeżeli **A** uruchomiony dla **X** zatrzyma się, to relacja między **X** a otrzymanym zestawem wyników jest spełniona.

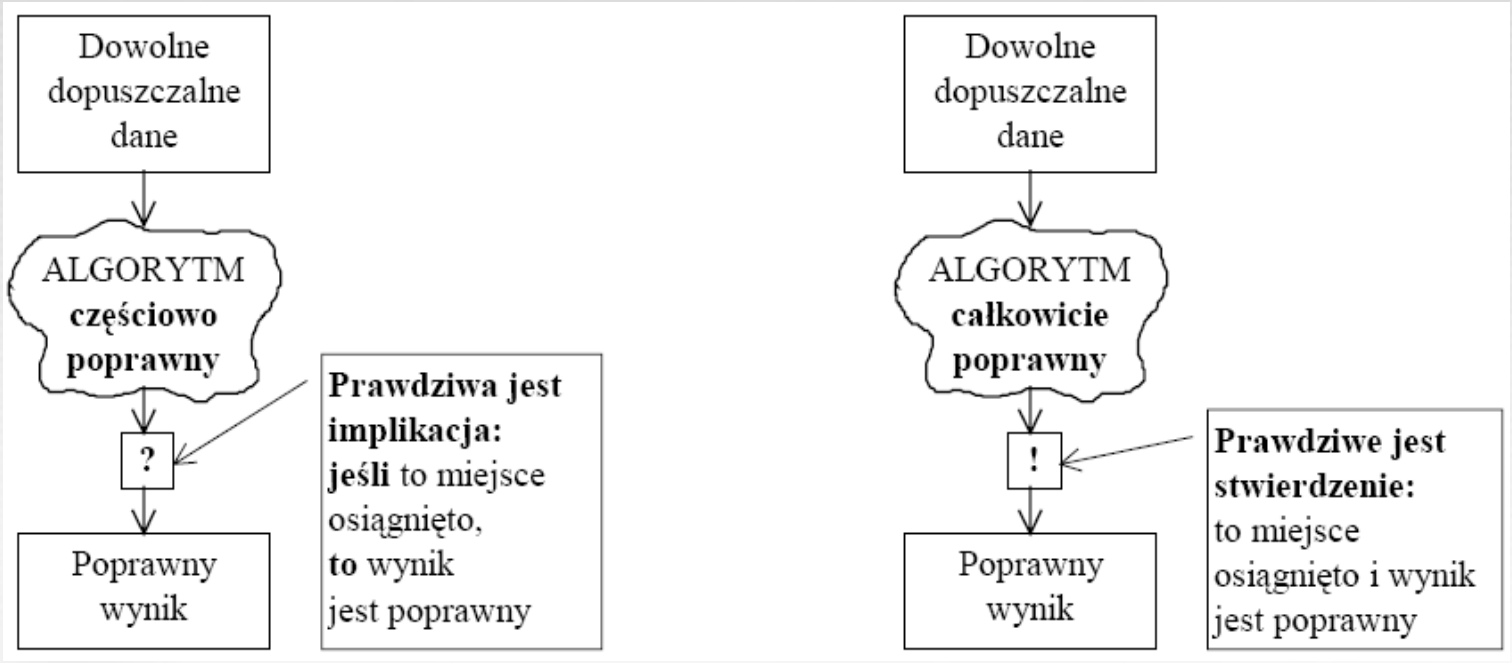
- Algorytm jest częściowo poprawny, wtedy, gdy jeśli się zakończy, to dla poprawnych danych wejściowych daje poprawny wynik. (Specyfikacja algorytmu jest spełniona).

Poprawność całkowita

- Algorytm **A** jest całkowicie poprawny jeżeli rozwiązuje zadanie dla każdego zestawu danych **X** z **P**.

- Innymi słowy, algorytm jest całkowicie poprawny, wtedy, gdy jest częściowo poprawny i posiada własność stopu. Algorytm posiada własność stopu, jeśli zatrzymuje się dla każdego zestawu poprawnych danych wejściowych.

Poprawność całkowita i częściowa



Poprawność całkowita i częściowa

```
s:=0;
i:=1;
while i<>n+2
do
begin
   s:=s + i;
   i:=i+2;
end
```

WP: n>0, n in N

WK: s=1+3+5+...+n, n%2=1

s=1+3+5+...+n-1, n%2=0

Algorytm poprawny częściowo:

dla n parzystego – pętla nieskończona

dla n nieparzystego – pętla skończona

Warunki wstępne i końcowe

Specyfikacja algorytmu składa się z dwóch części:

- **Warunek wstępny** - warunek dla danych wejściowych, który musi być spełniony na początku. Jeżeli dane wejściowe nie spełniają tego warunku, to nie ma gwarancji że algorytm będzie działał poprawnie ani nawet że się zatrzyma.
- **Warunek końcowy** – określa zależność wyników działania algorytmu od danych wejściowych przy założeniu że się on zatrzyma. Jest to warunek wyjścia z algorytmu, który jest zawsze prawdziwy po jego zakończeniu, jeżeli zachodził warunek wstępny.

Dowodzenie poprawności algorytmu

Częściowej poprawności algorytmu można dowieść poprzez

- wybranie punktów kontrolnych;
- związanie z każdym punktem asercji (funkcji logicznej reprezentującej przypuszczenie);
- ustalenie niezmienników w obrębie iteracji;
- dowiedzenie, że z prawdziwości jednej asercji wynika prawdziwość następnej, że niezmiennik pozostaje prawdziwy w kolejnych iteracjach i pociąga za sobą prawdziwość ostatniej asercji.

Całkowitej poprawności algorytmu można dowodzić przez dodatkowe:

- ustalenie zbieżnika (wielkości zależnej od zmiennych i danych, która jest zbieżna)
- dowiedzenie, że po skończonej liczbie iteracji algorytm zatrzyma się w ostatnim punkcie kontrolnym.

Przykład

Algorytm odwracający dowolny napis (procedura **odwrocone**):

odwrocone(„alamakota”)=„atokamala”

Pomocnicze funkcje:

glowa(„alamakota”)=„a”

ogon(„alamakota”)=„lamakota”

Operator łączenia napisów:

„alama”&”kota”=„alamakota”

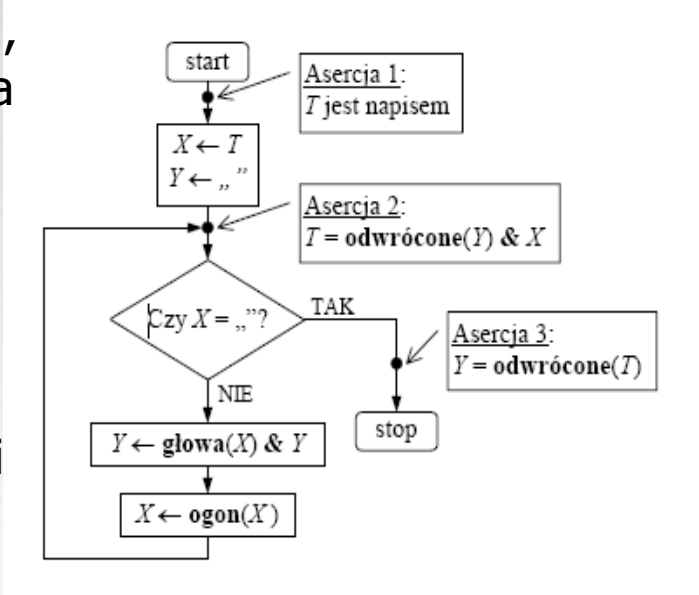
Dla dowolnego napisu T zachodzi zależność:

glowa(T)&ogon(T)=T

Przykład

Aby wykazać częściową poprawność algorytmu należy udowodnić, że:

1. Jeżeli asercja 1 jest prawdziwa, to asercja 2 też jest prawdziwa (przed rozpoczęciem iteracji)
2. Jeżeli w pewnym kroku iteracji asercja 2 jest prawdziwa, to w następnym kroku też jest ona prawdziwa.
3. Jeżeli w ostatnim kroku iteracji asercja 2 jest prawdziwa to asercja3 też jest prawdziwa.

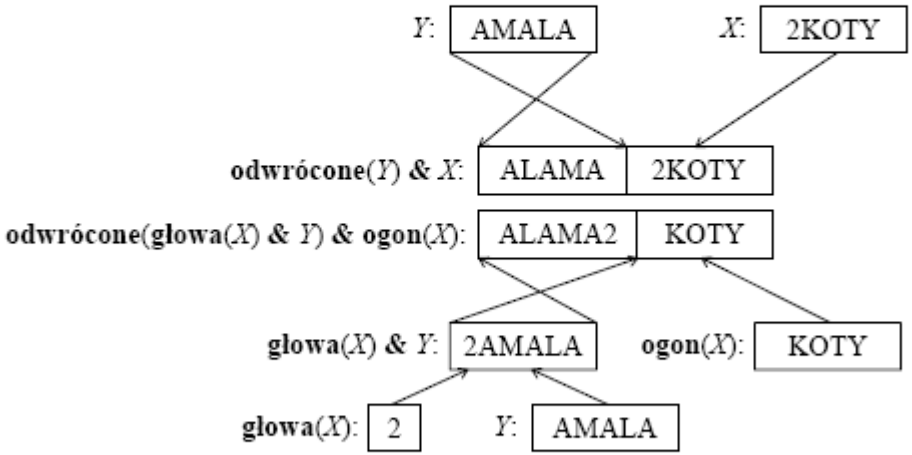


Aby wykazać całkowitą poprawność algorytmu dodatkowo należy udowodnić, że dla każdego napisu T punkt kontrolny 2 jest przechodzony tylko skończoną liczbę razy tzn. 3-ci punkt kontrolny jest zawsze osiągany.

Przykład

Ad 1.: oczywiście zachodzi równość **odwrócone**(„ ”) **&** $T = T$

Ad 2.: trzeba sprawdzić czy
odwrócone(Y) **&** X = **odwrócone**(**głowa**(X) **&** Y) **&** **ogon**(X)
dla każdego Y i $X \neq$ „ ”



Ad 3.: oczywiście zachodzi równość
odwrócone(**odwrócone**(Y) **&** „ ”) = Y

Weryfikacja poprawności algorytmu

Sprawdzenie częściowej poprawności

- Należy dowieść, że jest spełniona specyfikacja algorytmu:

  $$\{P\}\ S\ \{Q\},$$

  gdzie: P – warunek wstępny, S – akcja,

  Q – warunek końcowy.

Weryfikacja poprawności algorytmu

Sprawdzenie częściowej poprawności

- Aby tego dowieść trzeba podzielić specyfikację (tak jak algorytm) na wiele kroków, dla których weryfikacja poprawności jest prosta.

  Dla i-tego kroku:

$$\{P_i\}\ S_i\ \{Q_i\}.$$

```
x:=5
{true}x:=5{x=5}
```

```
x:=5
{x=6, y=100}x:=5{x=5,y=100}
```

```
x:=x+1
{x=15}x:=x+1{x=16}
```

Weryfikacja poprawności algorytmu

Sprawdzenie częściowej poprawności

- Zgodnie z logiką Hoare'a jeśli dla każdego i:

  $P_i = Q_{i-1}$ oraz

  $\{P_{i-1}\}\ S_{i-1}\ \{Q_{i-1}\}$ i $\{P_i\}\ S_i\ \{Q_i\}$,

  to $\{P_{i-1}\}\ S_{i-1}; S_i\ \{Q_i\}$.

- Jeżeli więc dla każdego i=1,2,…,n spełnione jest:

  $\{P_i\}\ S_i\ \{Q_i\}$,

  to jest spełniona specyfikacja algorytmu:

  $\{P\}\ S\ \{Q\}$.

Poprawność częściowa

Krokiem algorytmu może być:

- instrukcja prosta (przypisania),
- instrukcja strukturalna:
  - blok,
  - instrukcja wyboru,
  - pętla,
- podprogram (funkcja),
- cały program.

Poprawność częściowa

Przykłady - instrukcja przypisania:

- dla instrukcji $x:=4$:

  $\{true\}x:=4\{x=4\}$,

  co oznacza, że przy dowolnym stanie przed wykonaniem instrukcji, po wykonaniu instrukcji wartość zmiennej x jest równa 4.
- prawdą będą również formuły:

  $$\{x=6, y=100\}x:=5\{x=5, y=100\}$$

  oraz

  $$\{x=10\}x:=x+2\{x=12\}.$$

Poprawność częściowa

Instrukcje strukturalne i funkcje

- Aby wykazać prawdziwość specyfikacji instrukcji wyboru, należy dowieść, że warunek końcowy wynika z warunku wstępnego i warunku testu w każdym z przypadków instrukcji wyboru.
- Funkcje też mają swoje warunki wstępne i warunki końcowe; musimy sprawdzić że warunek wstępny funkcji wynika z warunku wstępnego kroku wywołania, a warunek końcowy pociąga za sobą warunek końcowy kroku wywołania.
- Wykazanie poprawności programowania dla pętli wymaga użycia **niezmiennika pętli**.

Niezmiennik pętli

- Niezmiennik pętli to wyrażenie logiczne, którego wartość nie zmienia się podczas wykonywania pętli.
- Określa warunki jakie muszą być zawsze spełnione przez zmienne w pętli, a także przez wartości wczytane lub wypisane.
- Warunki te muszą być prawdziwe przed pierwszym wykonaniem pętli oraz po każdym jej obrocie.

Niezmiennik pętli

```
while(w)
{
   …
   instrukcja 1;
   instrukcja 2;
   …
}
```

p – niezmiennik pętli **w** – warunek pętli

zdanie p:

- jest prawdziwe, kiedy wykonuje się treść pętli.
- jest prawdziwe po każdej iteracji pętli
- jest prawdziwe po zakończeniu pętli

zdanie w:

- jest prawdziwe, kiedy wykonuje się treść pętli
- jest fałszywe po zakończeniu pętli

Niezmiennik pętli

Dowód, że pewne zdanie jest niezmiennikiem pętli przeprowadza się wykorzystując **zasadę indukcji matematycznej**. Mówi ona że:

1. jeśli pewne zdanie jest prawdziwe dla n=0,
2. z tego, że jest prawdziwe dla pewnej liczby $n \geq 0$ wynika że musi być prawdziwe dla liczby n+1,
3. to z (1) i (2) wynika, że zdanie to jest prawdziwe dla wszystkich liczb nieujemnych całkowitych.

Niezmiennik pętli

```
int a=5, b=0;
for (int i=0; i<9; i++)
{
     b+=a;
}
```

niezmiennik pętli: a =5

Poprawność algorytmu

Częściowej poprawności algorytmu można dowodzić poprzez:

- wybranie **punktów kontrolnych,**
- związanie z każdym punktem **asercji** (funkcji logicznej reprezentującej przypuszczenie),
- ustalenie **niezmienników** w obrębie iteracji.

Całkowitej poprawności algorytmu można dowodzić poprzez dodatkowe:

- ustalenie **zbieżnika** (wielkości zależnej od zmiennych i danych, która jest zbieżna), co gwarantuje spełnienie warunku stopu.

Reguły wnioskowania dla poprawności częściowej

```
{true}
x := 1; n := N;
WHILE n > 0 DO {x*n! = N!}
BEGIN
x := x*n; n := n-1
END
{x = N!}
```

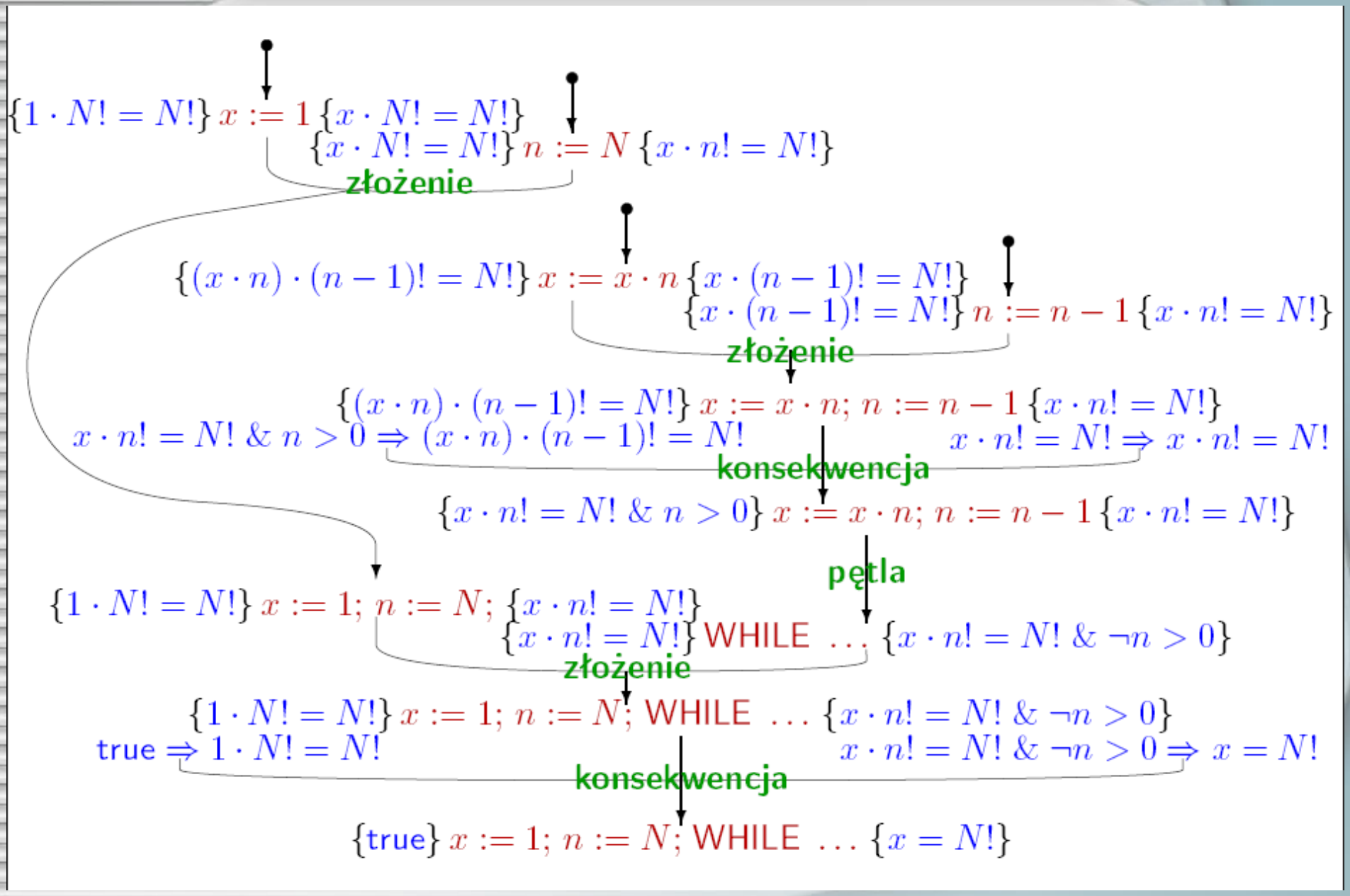
1. Ustalić niezmiennik: $x*n! = N!$
2. Dowieść, że to niezmiennik:

   $x*n! = N! \ \& \ n > 0 \Rightarrow ?\Rightarrow (x*n)*(n-1)! = N!$
3. Dowieść, że inicjalizacja gwarantuje jego spełnienie:

   $true \Rightarrow ?\Rightarrow 1*N! = N!$
4. Dowieść, że niezmiennik gwarantuje spełnienie warunku końcowego:

   $x*n! = N! \ \& \ \neg n > 0 \Rightarrow ?\Rightarrow x = N!$

Reguły wnioskowania dla poprawności częściowej



Reguły wnioskowania dla poprawności częściowej

Logika Hoare'a dla częściowej poprawności jest:

- **poprawna** - wyprowadza wyłącznie prawdziwe fakty,
- **zupełna** - każde prawdziwe stwierdzenie o częściowej poprawności daje się w niej wyprowadzić.

Nieautomatyzowalne kroki w dowodach częściowej poprawności:

- zgadywanie niezmienników pętli,
- udowadnianie faktów nieinformatycznych potrzebnych regule konsekwencji.

Logika Hoare'a daje się (z pewnym wysiłkiem) uogólnić na bardziej złożone języki programowania

Dowodzenie częściowej poprawności jest zbyt żmudne jak na codzienną praktykę programistyczną.

Logika Hoare'a wywarła wpływ na rozwój języków programowania oraz metod programowania.

Niezmiennik pętli

```
int potega1(int x, int n)
{
  int z,m,y;
  z=x; y=1; m=n;
  while (m!=0)
  {
     if (m%2==1) y=y*z;
     m=m/2;
     z=z*z;
  }
  return y;
}
```

potęgowanie binarne

Niezmiennik: $z^m * y = x^n$

n - liczba naturalna,
x- liczba rzeczywista

$z^m * y = x^n$

$z^{2* m/2} * (y * z) = x^n$ oraz $m \% 2=1$
lub $z^{2* m/2} * y=x^n$ oraz $m \% 2 = 0$

$(z*z)^{m/2} * y = x^n$

$(z*z)^m * y = x^n$

$z^m * y = x^n$

Po wykonaniu pętli:

$z^m * y = x^n$, $m=0$
czyli $y = x^n$

Niezmiennik pętli

Niezmiennik:
NWD(a,b)=NWD(b,a mod b)

```
int NWD(int a, int b)
{
 int c;
 while (b!=0)
 {
   c = a%b;
   a = b;
   b = c;
 }
 return a;
}
```

NWD(a,b)=NWD(b,a mod b)

Po wykonaniu pętli:

NWD(a,b)=NWD(b,a mod b), b=0

NWD(a,b)=NWD(b,a mod b)

algorytm Euklidesa

Niezmiennik pętli

Proste pętle mają zazwyczaj proste niezmienniki

Podział pętli:

- **pętla z wartownikiem:** czyta i przetwarza dane aż do momentu napotkania niedozwolonego elementu
- **pętla z licznikiem:** z góry wiadomo ile razy pętla będzie wykonana
- **pętle ogólne:** wszystkie inne

Podział pętli

- **pętla z wartownikiem**

```
j = 0 ;
L[N] = X ;    //  to jest wartownik
while ( L[j] != X ){
   j = j + 1;
   P = (j < N);
}
```

- **pętla z licznikiem**

```
for(j=0; j<10; j++){
   //instrukcje do wykonania
}
```

Warunek stopu

Warunek stopu określa, kiedy dany program ma zakończyć swoje działanie.

Zakończenie działania programu jest równoznaczne z:

- poprawnym wykonaniem wszystkich instrukcji;
- zakończeniem działania wszystkich pętli programu;
- zakończeniem się wszystkich wywołań funkcji rekurencyjnych.

Warunek stopu

Hipoteza Collatza – nierozstrzygalny przypadek

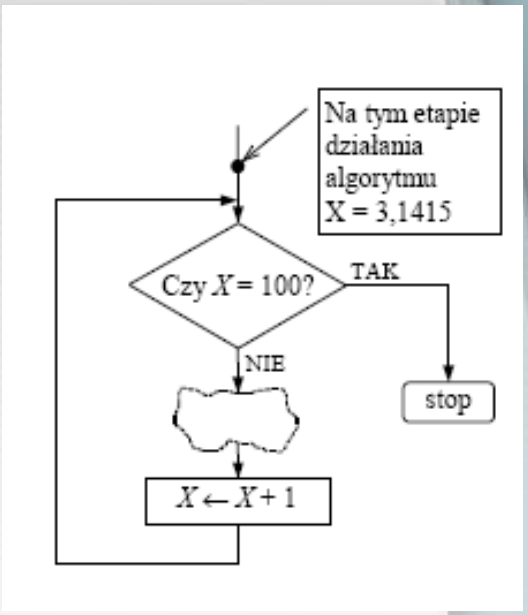
$$c_{n+1} = \begin{cases} \frac{1}{2}c_n & \text{gdy } c_n \text{ jest parzysta} \\ 3c_n + 1 & \text{gdy } c_n \text{ jest nieparzysta} \end{cases}$$

Hipoteza Collatza stwierdza, że niezależnie od jakiej liczby naturalnej c_0 zaczynamy obliczenia, w końcu otrzymamy liczbę 1.

Wykazano prawdziwość hipotezy Collatza dla liczb c_0 do 2^{58} jednak dla ogólnego przypadku problem nadal pozostaje nierozstrzygnięty

Rodzaje błędów w programach

- **<u>językowe</u>**
  powstają w wyniku naruszenia składni języka programowania
- **<u>semantyczne</u>**
  wynikają z niezrozumienia semantyki używanego języka programowania
- **<u>logiczne</u>**
- **<u>algorytmiczne</u>**
  wynikają z wadliwie skonstruowanych struktur sterujących np. niewłaściwych, zakresów iteracji niewłaściwych warunków użytych do zatrzymywania iteracji warunkowych lub przeniesienia sterowania w niewłaściwe miejsce procesu w wyniku zastosowania wyboru warunkowego (lub instrukcji skoku, pętla nieskończona itp).



Poprawność programu

Przy dowodzeniu poprawności programu oprócz poprawności algorytmu należy brać również pod uwagę:

- możliwość przepełnienia wartości całkowitoliczbowych,
- możliwość przepełnienia lub niedomiar dla liczb zmiennopozycyjnych,
- możliwość przekroczenia zakresów tablic,
- prawidłowość otwierania i zamykania plików,
- i inne.

Złożoność algorytmu

Złożoność obliczeniowa algorytmu jest to miara służąca do porównywania efektywności algorytmów.

Określa ona ilość zasobów komputerowych potrzebnych do wykonania algorytmu.

Podstawowe rozważane zasoby to:

- czas działania (złożoność czasowa);
- ilość zajmowanej pamięci (złożoność pamięciowa).

Złożoność algorytmu

- Złożoność czasowa - określa jak zwiększa się czas wykonania algorytmu przy zwiększaniu rozmiaru danych dla tego algorytmu
- Złożoność pamięciowa - określa jak zmienia się zajętość pamięci komputera przy wzroście rozmiaru danych algorytmu.

Złożoność algorytmu

Złożoność obliczeniowa algorytmu jest funkcją rozmiaru danych **n**.

Jednostki złożoności obliczeniowej:

- wykonanie jednej operacji dominującej dla złożoności czasowej;
- słowo pamięci maszyny dla złożoności pamięciowej.

Złożoność asymptotyczna - przybliżona miara efektywności

Złożoność asymptotyczna określa jak zachowuje się funkcja określająca złożoność dla odpowiednio dużych argumentów (ignorując wpływ argumentów małych i nieznaczących).

Złożoność asymptotyczna - przybliżona miara efektywności

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

n – ilość wykonywanych operacji

n	f(n)	n^2	$100 \bullet n$	$\log_{10} n$	1000
1	1 101	0.1%	9%	0.0%	91%
10	2 101	4.8%	48%	0.05%	48%
100	21 002	48%	48%	0.001%	4.8%
10^3	1 101 003	91%	9%	0.0003%	0.09%
10^4		99%	1%	0.0%	0.001%
10^5		99.9%	0.1%	0.0%	0.0000%

Dla dużych wartości n, funkcja rośnie jak n^2, pozostałe składniki mogą być zaniedbane.

Notacja „wielkie O”

Dane są funkcje f i g określone na zbiorze liczb rzeczywistych.

Mówimy, że f jest co najwyżej rzędu g, gdy istnieje takie stałe $n_0>0$ oraz $c>0$, takie że:

$$\forall n \geq n_0,\ f(n) \leq c \cdot g(n)$$

Zapis:

$$f(n) = O(g(n))$$

Notacja „wielkie O”

Przykład:

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

można przybliżyć jako:

$f(n) \sim n^2 + 100*n + O(\log_{10} n)$

albo jako:

$f(n) \sim O(n^2)$

Inne notacje

Analogicznie definiuje się notacje:

- „małe o” (f jest niższego rzędu niż g),

$$\forall n \geq n_0,\ f(n) < c \cdot g(n) \Rightarrow f(n) = o(g(n))$$

- „duże Ω„(f jest co najmniej rzędu g),

$$\forall n \geq n_0,\ f(n) \geq c \cdot g(n) \Rightarrow f(n) = \Omega(g(n))$$

- „małe ω” (f jest wyższego rzędu niż g),

$$\forall n \geq n_0,\ f(n) > c \cdot g(n) \Rightarrow f(n) = \omega(g(n))$$

- „duże Θ” (f jest dokładnie rzędu g)

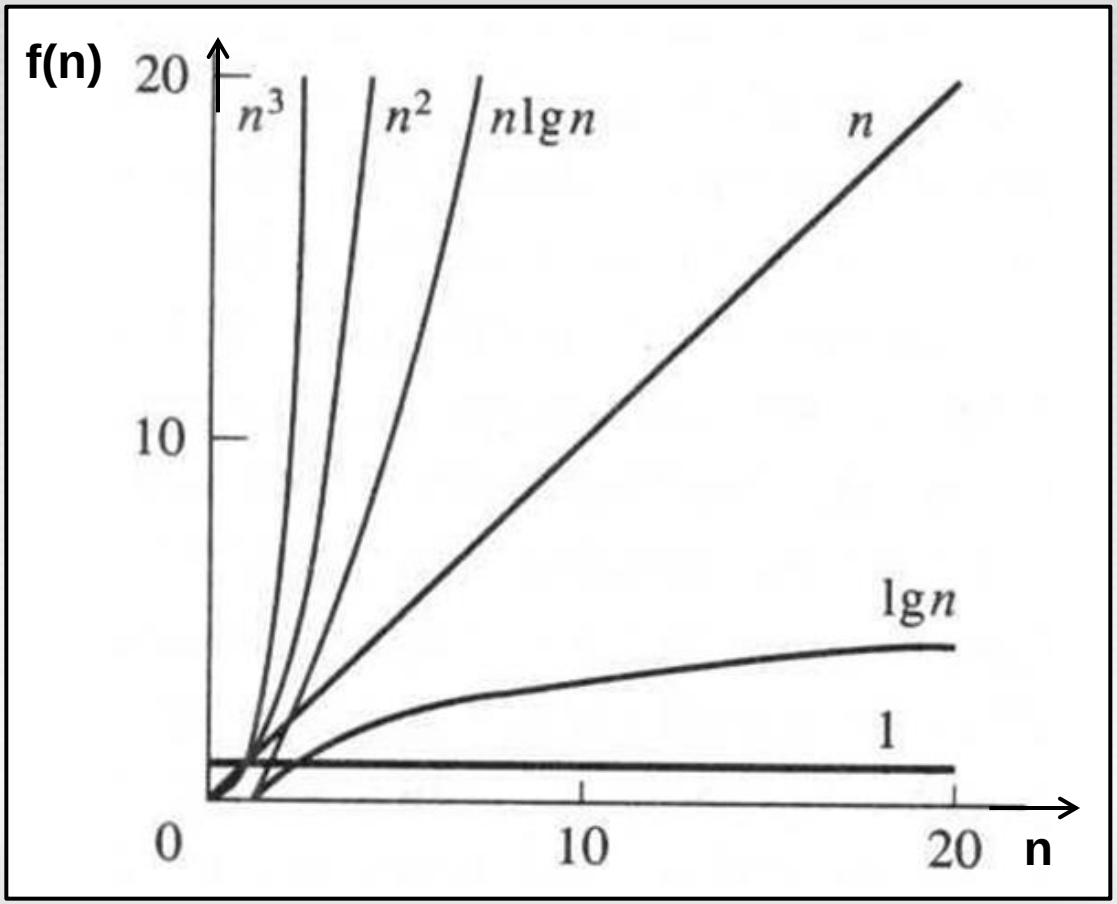
$$\forall n \geq n_0,\ f(n) = c \cdot g(n) \Rightarrow f(n) = \Theta(g(n))$$

Klasy złożoności

Klasy złożoności obliczeniowej:

- **1** - stała
- **$\log_2 n$** - logarytmiczna
- **n** - liniowa
- **$n \log_2 n$** liniowo-logarytmiczna (lub quasi-liniowa)
- **n^2** - kwadratowa
- **n^c** - wielomianowa
- **c^n** - wykładnicza

Klasy złożoności



Klasy złożoności

Klasy algorytmów i ich czasy wykonania na komputerze działającym z szybkością 1 instrukcja na μs

klasa	złożoność	liczba operacji i czas wykonania			
	n	**10**		**10^3**	
stały	$\mathcal{O}(1)$	1	1μs	1	1 μs
logarytm	$\mathcal{O}(\log n)$	3.32	3μs	9.97	10 μs
liniowy	$\mathcal{O}(n)$	10	10μs	10^3	1ms
kwadratowy	$\mathcal{O}(n^2)$	10^2	100μs	10^6	1s
wykładniczy	$\mathcal{O}(2^n)$	1024	10ms	10^{301}	>> 10^{16} lat

Klasy złożoności

Jedną z najważniejszych funkcji przy ocenianiu efektywności algorytmów jest **funkcja logarytmiczna**. Jeżeli można wykazać że złożoność algorytmu jest rzędu logarytmicznego, algorytm można traktować jako bardzo dobry. Istnieje wiele funkcji lepszych w tym sensie niż logarytmiczna, jednak zaledwie kilka spośród nich, jak $O(\log_2 \log_2 n)$ czy $O(1)$ ma praktyczne znaczenie.

Klasy złożoności

Klasy złożoności obliczeniowej algorytmów potęgowania z wykładnikiem naturalnym:

Nazwa algorytmu potegowania	Klasa złożoności
„naiwny”	$O(n)$
binarny	$O(\log_2 n)$

Złożoność algorytmu

Złożoność czasowa algorytmu jest funkcją rozmiaru danych wejściowych.

Zależy ona także od innych parametrów, np. od rodzaju danych wejściowych i ich uporządkowania.

Wyróżnia się:

- złożoność oczekiwaną;
- złożoność optymistyczną;
- złożoność pesymistyczną.

Porównanie klas złożoności

Klasy złożoności obliczeniowej wybranych algorytmów sortujących:

Nazwa algorytmu sortowania	Klasa złożoności		
	optymistyczna	typowa	pesymistyczna
bąbelkowy	$O(n)$	$O(n^2)$	$O(n^2)$
przez wybieranie	$O(n^2)$	$O(n^2)$	$O(n^2)$
przez wstawianie	$O(n)$	$O(n^2)$	$O(n^2)$
szybki	$O(n \log n)$	$O(n \log n)$	$O(n^2)$

Testy wzorcowe

- Porównywania efektywności programów zaprojektowanych do wykonywania tego samego zadania można dokonać też eksperymentalnie za pomocą testów wzorcowych. Wykorzystywany jest do tego celu niewielki zbiór typowych i reprezentatywnych danych wejściowych które traktowane są jako ***dane wzorcowe*** (ang. benchmark).
- Np. test wzorcowy umożliwiający porównanie algorytmów sortujących może opierać się na losowo uporządkowanych danych o różnej wielkości. Przydatne jest też sprawdzenie jak algorytm działa dla ciągu już posortowanego i dla ciągu posortowanego odwrotnie.

Czas działania

Czasem działania nazywamy funkcje **T(n),** które określają liczbę jednostek czasu, które zajmuje wykonanie programu lub algorytmu w przypadku problemu o rozmiarze **n**.

Jeżeli czas działania zależy od konkretnych danych wejściowych, nie tylko ich rozmiaru, funkcje **T(n)** definiuje się jako najmniej korzystny przypadek z punktu widzenia kosztów czasowych.

Inną wyznaczaną wielkością jest też *czas średni*, czyli średni dla różnych danych wejściowych.

Doświadczalne sprawdzenie złożoności obliczeniowej

Na podstawie pomiaru czasów działania algorytmu dla kilku różnych zbiorów wejściowych o różnej liczbie elementów można doświadczalne zweryfikować klasę złożoności algorytmu.

Przykład:

Jeśli pewien algorytm posiada czasową złożoność obliczeniową klasy $O(n^2)$, to czas wykonania algorytmu dla n elementów jest w przybliżeniu proporcjonalny do kwadratu n, czyli:

gdzie: n - liczba przetwarzanych elementów,

$t(n)$ - czas przetwarzania n-elementów w algorytmie

c - stała proporcjonalności pomiędzy $t(n)$ a n^2.

Zależność tę można sprawdzić doświadczalnie.

Doświadczalne sprawdzenie złożoności obliczeniowej

Zawartość pliku wygenerowanego przez program

```
Nazwa: Sortowanie bąbelkowe - Bubble Sort 3
-------------------------------------------
(C)2005 mgr J.Wałaszek      I LO w Tarnowie

------n---------tpo---------tod---------tpp---------tpk---------tnp
   1000     0.000008     0.030413     0.000042     0.002527     0.020769
   2000     0.000014     0.122591     0.000084     0.010772     0.072822
   4000     0.000025     0.487696     0.000164     0.032994     0.292399
   8000     0.000050     1.954120     0.000253     0.095668     1.181462
  16000     0.000119     7.963850     0.000707     0.539036     4.686158
  32000     0.000283    31.464163     0.001363     1.871901    18.937897
-----------------------------------------------------------------------
Koniec
```

Objaśnienia oznaczeń (wszystkie czasy podano w sekundach):

- n - ilość elementów w sortowanym zbiorze
- t_{po} - czas sortowania zbioru posortowanego
- t_{od} - czas sortowania zbioru posortowanego malejąco
- t_{pp} - czas sortowania zbioru posortowanego z losowym elementem na początku
- t_{pk} - czas sortowania zbioru posortowanego z losowym elementem na końcu
- t_{np} - czas sortowania zbioru z losowym rozkładem elementów