POPRAWNOŚĆ PROGRAMÓW

Na poprawność programu komputerowego składa się:

- poprawność algorytmu,
- poprawność zapisu w języku programowania.

Jeśli uważasz, że jakiś program komputerowy jest bezbłędny, to się mylisz – po prostu nie zauważyłeś jeszcze skutków błędu, który jest w nim zawarty. ☺ 😐 ☹

Na poprawność algorytmu składa się:

- poprawność logiczna (założenia, rozumowanie, metoda, ...)
- poprawność algorytmiczna (struktura, sterowanie procesem, ...)

Na poprawność zapisu składa się:

- poprawność składniowa,
- poprawność semantyczna.

Gdzie można popełnić błąd?

1. **Błędy składniowe**

   Powstają w wyniku naruszenia reguł składni języka programowania, którego używamy do zapisania algorytmu

   Np. zamiast **for** *K* := 1 **to** *N* **do** *X*[*K*] := *K*

   napisaliśmy **for** *K* := 1 **to** *N* **to** *X*[*K*] := *K*

   Możliwe skutki i znaczenie:

- zatrzymanie kompilacji lub interpretacji z komunikatem lub bez,
- przerwanie realizacji programu nawet jeśli kompilator błędu nie wykrył,
- błędy nieprzyjemne, ale zwykle niezbyt poważne

– są względnie łatwe do poprawienia.

2. **Błędy semantyczne**

   Wynikają z niezrozumienia semantyki używanego języka programowania

   Np. sądziliśmy, że po zakończeniu instrukcji wykonania iteracji ograniczonej **for** *K* := 1 **to** *N* **do** *X*[*K*] := *K*

   zmienna *K* ma wartość równą zmiennej *N*, a nie *N* + 1, jak wynika z reguł semantycznych języka *MJP*, i wykorzystywaliśmy to w dalszej części programu

   Możliwe skutki i znaczenie:

- program nie realizuje poprawnie algorytmu,
- błędy trudne do przewidzenia i potencjalnie groźne, ale są do uniknięcia przy wnikliwym sprawdzaniu znaczenia używanych instrukcji.

3. **Błędy logiczne**

   Mogą pochodzić z przyjęcia fałszywych założeń, wad rozumowania lub źle dobranej metody wyznaczania wyniku.

   Np. w algorytmie zliczania zdań zawierających słowo „algorytm” nie zauważyliśmy, że sekwencja dwóch znaków „**.** ” może występować także wewnątrz zdania:

   „Na Rys. 2 pokazano schemat...”,

   a używaliśmy jej do wyszukiwania jego końca.

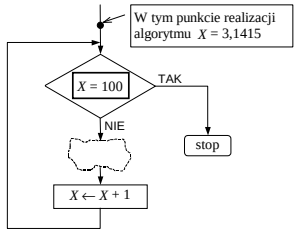
Możliwe skutki i znaczenie:

- algorytm przestaje być poprawnym rozwiązaniem zadania algorytmicznego,
- dla pewnych zestawów danych wejściowych algorytm podaje wyniki niezgodne z oczekiwanymi,
- procesor może nie być w stanie wykonać pewnych instrukcji (np. żądamy dzielenia przez 0),
- błędy bardzo groźne – mogą być trudne do znalezienia i pozostawać długo w ukryciu, nawet w trakcie wielokrotnego używania programu w postaci kodu.

4. **Błędy algorytmiczne**

   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).

   Np. pętla nieskończona:



Możliwe skutki i znaczenie:

- algorytm dla pewnych dopuszczalnych danych wejściowych daje niepoprawny wynik,
- wykonanie programu realizującego algorytm jest przerywane w trybie awaryjnym,
- program realizujący algorytm nie kończy w normalnym trybie swego działania,
- błędy groźne, ale możliwe do uniknięcia poprzez staranne opisanie struktury algorytmu przed jego zapisaniem w jakimkolwiek języku oraz zachowanie zasad „dobrej praktyki" programowania.

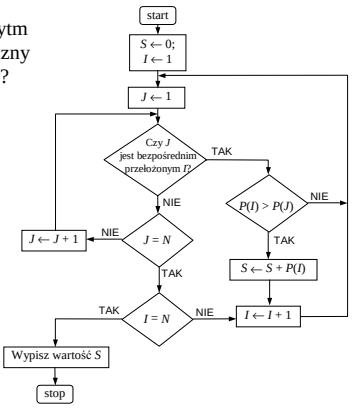
Przykład algorytmu do sprawdzenia, czy nie ma błędów

Konstruujemy algorytm sumowania zarobków pracowników, którzy zarabiają więcej niż ich bezpośredni przełożeni.

Przyjmujemy koncepcję sprawdzania kolejno płac wszystkich pracowników; dla każdego z nich będzie wyszukiwana płaca jego bezpośredniego przełożonego i porównanie tych dwóch płac będzie decydowało o dodaniu kolejnego składnika do wyznaczanej sumy.

Zmienna N ma wartość równą liczbie pracowników, zmienne (indeksowe) I i J wskazują na kolejne elementy tablicy jednowymiarowej $P(X)$, która zawiera płace pracowników, zmienna S kumuluje sumę zarobków; dla każdej pary pracowników wskazanych aktualnymi wartościami zmiennych I i J jest dostępna informacja, która pozwala stwierdzić, czy J jest bezpośrednim przełożonym I.

Czy podany algorytm zawiera błąd logiczny lub algorytmiczny?



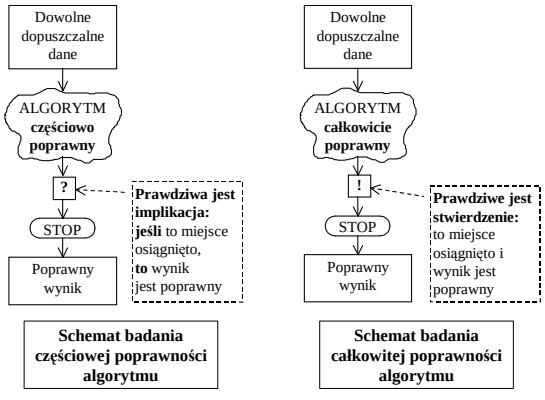
Algorytmem **całkowicie poprawnym** nazywamy algorytm, dla którego **udowodniono**, że nie zawiera on ani błędów logicznych, ani algorytmicznych.

Twierdzenie, które należy udowodnić brzmi:

„Dany algorytm dla każdego zestawu dopuszczalnych danych wejściowych samoistnie przestaje działać po wykonaniu skończonej liczby operacji elementarnych, dając poprawny (spełniający założone warunki) wynik końcowy"

Etapem pośrednim dla wykazania całkowitej poprawności może być **udowodnienie częściowej poprawności** algorytmu.

„Dla każdego zestawu dopuszczalnych danych wejściowych z faktu, że dany algorytm samoistnie przestał działać po wykonaniu skończonej liczby operacji elementarnych wynika, że dał poprawny (spełniający założone warunki) wynik końcowy"



Złożony program komputerowy może zawierać sporą liczbę błędów różnych typów, popełnionych na różnych etapach jego powstawania.

Dla złożonego algorytmu badanie jego całkowitej poprawności może być trudne i czasochłonne

W praktyce opracowywania programów komputerowych pomija się (niestety) etap badania całkowitej poprawności algorytmu i bada się „produkt końcowy", czyli sam program:

- testowanie na licznych zestawach danych wejściowych (dla takiego zestawu powinien być znany wynik końcowy)
- uruchamianie w obecności narzędzi diagnostycznych (badanie stanów pośrednich i końcowych)

Podstawową metodą badania **częściowej poprawności** algorytmu jest **metoda niezmienników**, polegająca na:

- wybraniu w schemacie algorytmu **punktów kontrolnych**,
- związaniu z każdym punktem kontrolnym tzw. **asercji**, czyli warunku logicznego zależnego od stanu realizacji procesu wyznaczania założonego wyniku końcowego,
- ustaleniu w obrębie każdej z iteracji takiej asercji, której prawdziwość będzie można wykazać po dowolnej liczbie powtórzeń tej iteracji (tzw. **niezmiennik** iteracji),
- wykazaniu, że z prawdziwość jednej asercji wynika prawdziwość następnej, że niezmienniki pozostają prawdziwe po kolejnych iteracjach i pociągają za sobą prawdziwość ostatniej asercji, która oznacza osiągnięcie założonego wyniku.

Po wykazaniu częściowej poprawności algorytmu podstawową metodą badania jego **całkowitej poprawności** jest **metoda zbieżników**, polegająca na:

- ustaleniu dla każdej iteracji **zbieżnika**, czyli takiej zmiennej, której wartości zależą od stanu realizacji algorytmu po wykonaniu kolejnych powtórzeń tej iteracji i tworzą ograniczony ciąg monotoniczny,
- wykazaniu, że każdy ze zbieżników nie może zmieniać się nieskończoną liczbę razy i algorytm po wykonaniu skończonej liczby powtórzeń w każdej z iteracji zatrzyma się w ostatnim punkcie kontrolnym.

Przykład zastosowania metody niezmienników i zbieżnika

Badamy algorytm odwracania dowolnego napisu, którego działanie zapisane w postaci procedury **odwrócone** jest następujące:

odwrócone(„alama2koty”) = „ytok2amala”

Wykorzystamy pomocnicze procedury:

głowa(„alama2koty”) = „a”

ogon(„alama2koty”) = „lama2koty”

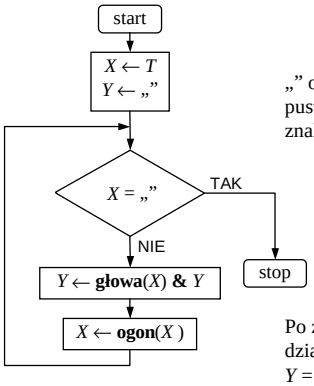
oraz operator konkatenacji **&** (złączenia) dwóch napisów:

„alama” **&** „2koty” = „alama2koty”

Dla dowolnego napisu *T* zachodzi tożsamość:

głowa(*T*) **& ogon**(*T*) = *T*

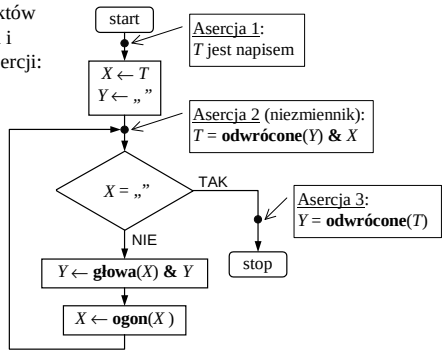
Schemat procedury **odwrócone**



„” oznacza napis pusty (o liczbie znaków równej 0)

Po zakończeniu działania ma być: *Y* = odwrócone (*T*)

Wybór punktów kontrolnych i przydział asercji:



Aby wykazać **częściową poprawność** algorytmu należy kolejno udowodnić prawdziwość następujących implikacji:

1. Jeżeli asercja 1. jest prawdziwa, to asercja 2. przed rozpoczęciem iteracji jest prawdziwa,
2. Jeżeli w pewnym kroku iteracji asercja 2. jest prawdziwa, to w następnym kroku też jest ona prawdziwa (warunek z asercji 2. stanie się wtedy niezmiennikiem iteracji),
3. Jeżeli w ostatnim kroku iteracji asercja 2. jest prawdziwa, to asercja 3. jest też prawdziwa (osiągnięto założony wynik).

Jeżeli T jest napisem (dopuszczalną daną wejściową), to przed rozpoczęciem iteracji asercja 2. jest prawdziwa, bo zachodzi oczywista równość:

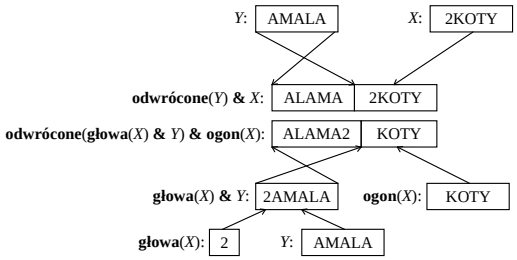
odwrócone(„”) **&** $T = T$, dla Y = „” i $X = T$

Jeżeli założymy, że po wykonaniu pewnej liczby iteracji zachodzi T = **odwrócone**(Y) **&** X dla pewnego Y i $X \neq$ „”, to aby wykazać, że ten warunek nadal będzie spełniony po wykonaniu następnej iteracji, wystarczy pokazać, że

odwrócone(Y) **&** X = **odwrócone**(**głowa**(X) **&** Y) **& ogon**(X) ,

bo nowa wartość Y po wykonaniu iteracji jest równa **głowa**(X) **&** Y, a nowa wartość X jest równa **ogon**(X).

Schemat wykazujący, że
odwrócone(Y) **&** X = **odwrócone**(**głowa**(X) **&** Y) **& ogon**(X)



Zatem po wykonaniu kolejnej iteracji zachodzi

T = **odwrócone**(**głowa**(X) **&** Y) **& ogon**(X),

czyli asercja 2. pozostaje prawdziwa bez względu na liczbę powtórzeń iteracji – jest jej niezmiennikiem.

Jeżeli po ostatnim powtórzeniu iteracji prawdziwa jest asercja 2., to

T = **odwrócone**(Y) **&** X dla X = „”, czyli T = **odwrócone**(Y).

Z oczywistej równości

odwrócone(T) = **odwrócone**(**odwrócone**(Y)) = Y

wynika prawdziwość asercji 3.

Wykazana została częściowa poprawność algorytmu:

jeżeli T jest napisem, to Y = **odwrócone**(T) w punkcie kontrolnym 3.

Aby wykazać teraz **całkowitą poprawność** algorytmu trzeba wykazać, że dla dowolnego napisu T punkt kontrolny 2. jest przechodzony tylko skończoną liczbę razy, tzn. 3. punkt kontrolny jest zawsze osiągany.

Wybieramy w tym celu zbieżnik, którym może być długość napisu będącego aktualną wartością zmiennej X.

Przed rozpoczęciem iteracji $X = T$, czyli wartość zbieżnika równa jest długości napisu T (liczba całkowita nieujemna).

Po każdym powtórzeniu iteracji wartość zbieżnika maleje o 1, bo $X \leftarrow$ **ogon**(X).

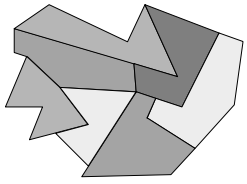
Wartość zbieżnika (długość napisu) nie może być mniejsza od zera, a zatem iteracja może zostać powtórzona tylko skończoną liczbę razy i algorytm osiągnie punkt kontrolny 3.

Wykazana została całkowita poprawność algorytmu:

dla każdego napisu T algorytm znajdzie się w punkcie kontrolnym 3. z wynikiem Y = **odwrócone**(T).

Przykład do czego prowadzą trudności w stosowaniu metody niezmienników i zbieżników

Problem kolorowania mapy płaskiej z 1852 r. (de Morgan)



Rozwiązanie z 1976 r. (Appel i Haken)

Twierdzenie:

Cztery barwy zawsze wystarczą do pokolorowania dowolnej płaskiej mapy, tak aby każdy z dwóch sąsiadujących obszarów różnił się kolorem.

Dowód

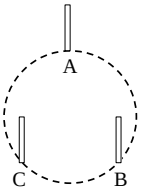
Zbudowano algorytmy rozwiązujące problem kolorowania w szczególnych przypadkach map, które wyczerpują wszystkie możliwe typy map płaskich.

Nikt formalnie nie udowodnił całkowitej poprawności tych algorytmów!

Dla większości algorytmów rekurencyjnych dowód ich całkowitej poprawności może być przeprowadzony po usunięciu najpierw rekurencji z algorytmu – zastąpieniu go równoważnym algorytmem opartym na iteracjach – i zastosowaniu potem metody niezmienników i zbieżników.

Przykład usunięcia rekurencji z algorytmu – wieże Hanoi

Ustaw trzy kołki na okręgu



Iteracyjny algorytm dla problemu wież Hanoi

1. Powtarzaj, co następuje:
   - 1.1. przenieś najmniejszy z dostępnych krążków z kołka, na którym się znajduje, na kołek następny w kierunku ruchu wskazówek zegara,
   - 1.2. jeżeli na kołku C znajdują się już wszystkie krążki, to zakończ działanie,
   - 1.3. wykonaj jedyne możliwe przeniesienie nie zmieniające położenia najmniejszego krążka, który został przeniesiony w kroku 1.1.