# Wybrane metody projektowania algorytmów (II)

Przyjmujemy następującą definicję algorytmu: algorytm to sposób rozwiązania zadania (problemu) w skończonej liczbie kroków z wykorzystaniem narzędzi informatycznych.

Do najważniejszych cech algorytmów zaliczają się:

■ Poprawność - algorytm powinien zwracać prawidłowe wyniki dla każdego zestawu poprawnych danych (na początku pracy z algorytmami nie musisz dbać o odporność algorytmu na błędy wynikające ze złych danych, ale stopniowo należy wziąć to pod uwagę).

■ Skończoność - rozwiązanie zadania musi być możliwe dla dowolnego zestawu danych w skończonej liczbie kroków.

■ Jednoznaczność - algorytm powinien zwracać te same wyniki dla zestawów takich samych danych wejściowych.

■ Sprawność - ta cecha określa, jak zachowuje się algorytm zarówno pod względem szybkości działania, jak i optymalnego wykorzystania zasobów komputera, w szczególności jego pamięci operacyjnej. Omówimy ją dokładnie nieco później.

Podczas budowania algorytmów należy używać zmiennych, których zadaniem jest przechowywanie wartości, m.in. liczbowych, logicznych bądź tekstowych. Umiejętność zapisywania algorytmów w dowolnej formie jest bardzo ważna, ale równie istotna jest czytelna dokumentacja, która szybko przypomni ci, czego algorytm dotyczy, jak również ułatwi analizę jego działania. Podobnie jak podczas rozwiązywania zadań z innych przedmiotów, przed przystąpieniem do tworzenia algorytmu należy określić, czego on dotyczy, opisać stosowane symbole, wyjaśnić, jakiego typu będą dane wejściowe i wyniki. Dobrym zwyczajem jest umieszczenie w dokumentacji krótkiego opisu zastosowanego algorytmu. Wszystkie informacje zawarte w takiej dokumentacji określa się mianem specyfikacji problemu algorytmicznego. Tworzenie i analiza algorytmów może być zajęciem bardzo ciekawym, zajmującym i twórczym.

Opracowanie algorytmu wraz z poprawną specyfikacją pozwoli łatwo zastosować je w różnych środowiskach programistycznych.

Poprawnie sformułowana specyfikacja problemu algorytmicznego jest punktowana przy ocenie rozwiązań zadań maturalnych z informatyki. Jej brak obniża ocenę nawet poprawnie działającego algorytmu. Zobacz na przykładzie, jak powinna wyglądać taka specyfikacja.

**Algorytm**

Obliczanie k-tej potęgi liczby n, czyli $n^k$, gdzie $n,k \in N_+$.

**Specyfikacja problemu algorytmicznego:**

**Dane:**

$n, k \in N_+$,

n - podstawa potęgi,

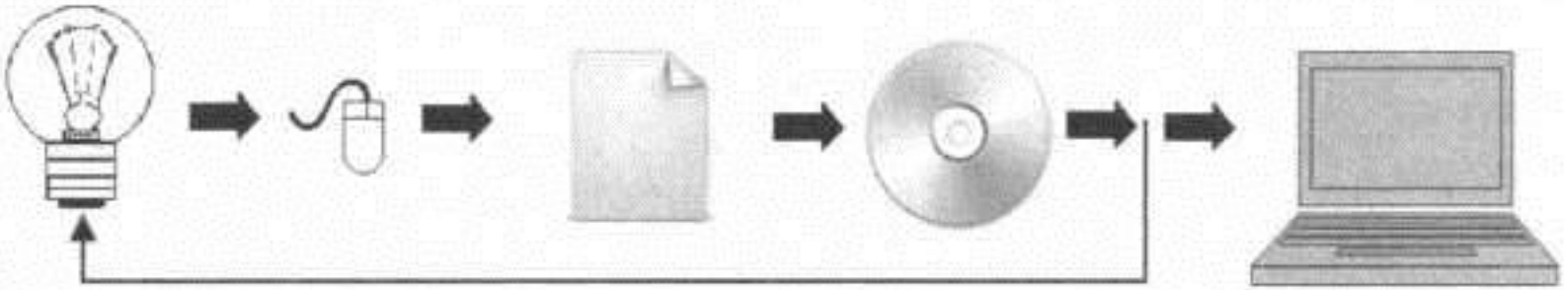
k - wykładnik potęgi.

**Wynik:**

P - obliczana wartość potęgi, $P \in N_+$.

## Proces koncepcji programów

**Rysunek 1.2.**
*Etapy konstrukcji programu*



Na rysunku 1.2 w sposób uproszczony zobrazowano etapy procesu programowania komputerów. Olbrzymia żarówka symbolizuje etap, który jest od czasu do czasu pomijany przez programistów (dodajmy, że zwykle z opłakanymi skutkami) — REFLEKSJĘ.
Następnie jest tworzony tzw. tekst źródłowy nowego programu, mający postać pliku tekstowego, wprowadzanego do komputera za pomocą zwykłego edytora tekstowego. Większość istniejących obecnie kompilatorów posiada taki edytor już wbudowany, więc użytkownik w praktyce nie opuszcza tzw. środowiska zintegrowanego, grupującego programy niezbędne w procesie

programowania. Ponadto niektóre środowiska zintegrowane zawierają zaawansowane edytory graficzne umożliwiające przygotowanie zewnętrznego interfejsu użytkownika praktycznie bez pisania ani jednej linii kodu.
Efektem pracy programisty jest plik lub zespół plików opisujących w formie symbolicznej sposób zachowania się programu wynikowego. Opis ten jest kodowany w tzw. języku programowania, który stanowi na ogół podzbiór języka naturalnego.
Kompilator wykonuje mniej lub bardziej zaawansowaną analizę poprawności i, jeśli wszystko jest w porządku, produkuje tzw. kod wykonywalny, zapisany w postaci zrozumiałej dla komputera. Plik zawierający kod wykonywalny może być następnie wykonywany pod kontrolą systemu operacyjnego komputera (który to system notabene jest także złożony z wielu pojedynczych programów). Kod wykonywalny jest specyficzny dla danego systemu operacyjnego. W ostatnich latach rozpowszechnił się język Java, który pozwala budować programy niezależne od systemów operacyjnych, ale dzieje się to na zasadzie pewnego „oszustwa”: kod wykonywalny nie jest uruchamiany bezpośrednio przez system operacyjny, tylko przez specjalne środowisko uruchomieniowe, które izoluje go od sprzętu i systemu, wprowadzając jako warstwę pośrednią opóźnienia niezbędne dla dokonania translacji kodu pośredniego na kod finalny.

# Poziomy abstrakcji opisu i wybór języka

Jednym z delikatniejszych problemów związanych z opisem algorytmów jest sposób ich prezentacji zewnętrznej. Można w tym celu przyjąć dwie skrajne pozycje:

♦ zbliżyć się do maszyny (język asemblera: nieczytelny dla nieprzygotowanego odbiorcy);

♦ zbliżyć się do człowieka (opis słowny: maksymalny poziom abstrakcji zakładający poziom inteligencji odbiorcy niemożliwy aktualnie do wbudowania w maszynę).

Wybór języka asemblera do prezentacji algorytmów wymagałby w zasadzie związania się z określonym typem maszyny, co zlikwidowałoby jakąkolwiek ogólność rozważań i uczyniłoby opis trudnym do analizy. Z drugiej zaś strony opis słowny wprowadza ryzyko niejednoznaczności, która może być kosztowna: program, po przetłumaczeniu go na postać zrozumiałą dla komputera, może nie zadziałać!

Aby zaradzić zaanonsowanym wyżej problemom, zwyczajowo przyjęło się prezentowanie algorytmów w dwojaki sposób:

♦ za pomocą istniejącego języka programowania;

♦ przy użyciu pseudojęzyka programowania (mieszanki języka naturalnego i form składniowych pochodzących z kilku reprezentatywnych języków programowania).

Jeśli przykładowo dany algorytm jest możliwy do czytelnego zaprezentowania za pomocą języka programowania, wybór będzie oczywisty! Od czasu do czasu jednak napotkamy sytuacje, w których prezentacja kodu w pełnej postaci, gotowej do wprowadzenia do komputera, byłaby zbędna (np. zbliżony materiał był już przedstawiony wcześniej) lub nieczytelna (liczba linii kodu przekracza objętość jednej strony). W każdym przypadku ewentualne przejście z jednej formy w drugą nie powinno stanowić większego problemu.

Na początku prezentacji pokazano przykład algorytmu Euklidesa (NWD), który — operując na liczbach — dość łatwo daje się przełożyć na instrukcje zrozumiałe przez komputer. Dochodzimy tutaj do momentu, gdy chcemy przejść z fazy abstrakcji do realizacji.

Jak komputer wykonuje taki algorytm? W sytuacji, gdy mamy do czynienia z rzeczywistym systemem komputerowym, jedynym sposobem komunikacji jest język programowania. Możemy wówczas zapisać algorytm do pliku, wykonać tzw. kompilację (przetłumaczyć na instrukcje zrozumiałe dla systemu operacyjnego) i ewentualnie go wykonać.

Dla ilustracji podaję już teraz kod programu C++ który realizuje drugi wariant algorytmu NWD i pokazuje jego użycie.

```
#include <iostream>
using namespace std;

int nwd (int a, int b){
	if (b==0)
		return a;
	else
		return nwd (b, a % b);//	operator % w C++ realizuje funkcję modulo
}					//	(reszta z dzielenia liczb całkowitych)

int main(){
	cout « " nwd(12,3)=" «  nwd ( 12,3) « “.”;
	return 0;
}
```

Należy zwrócić uwagę, że spora część powyższego zapisu nic nie wnosi do istoty działania samego algorytmu i stanowi tylko opakowanie, niezbędne do kompilacji i użycia kodu (dyrektywy kompilatora, wbudowanie algorytmu do funkcji main).

Zaletą kodu C++ jest precyzja zapisu, której brakuje w opisach o charakterze pseudokodu, poza tym możliwość testowania rzeczywistego kodu pozwala na łatwiejsze opanowanie prezentowanego materiału.

# Poprawność algorytmów

Wpisanie programu do komputera, skompilowanie go i uruchomienie jeszcze nie gwarantują że kiedyś nie nastąpi jego załamanie (cokolwiek by to miało znaczyć w praktyce). O ile jednak w przypadku niewinnych domowych aplikacji nie ma to specjalnego znaczenia (w tym sensie, że tylko my ucierpimy), to w momencie zamierzonej komercjalizacji programu sprawa znacznie się komplikuje. W grę zaczyna wchodzić nie tylko kompromitacja programisty, ale i jego odpowiedzialność za ewentualne szkody poniesione przez użytkowników programu.

Błędom w swoich produktach nie są w stanie zapobiec nawet wielkie koncerny programistyczne — w miesiąc po kampanii reklamowej produktu X pojawiają się po cichu „darmowe” (dla legalnych użytkowników) uaktualnione wersje, które nie mają wcześniej niezauważonych błędów. Popularne systemy operacyjne, takie jak np. liczne odmiany Linuksa, Windows lub OS X. posiadają wbudowane mechanizmy automatycznej aktualizacji przez Internet, które służą do naprawiania wadliwych funkcji systemu lub bieżącej reakcji na zagrożenia (np. wirusy).

Zajmijmy się jednak czymś bliższym rzeczywistości typowego programisty: pisze on program i chce uzyskać odpowiedź na pytanie, czy będzie działał poprawnie w każdej sytuacji, dla każdej możliwej konfiguracji danych wejściowych. Odpowiedź jest tym trudniejsza, im bardziej skomplikowane są procedury, które zamierzamy badać. Nawet w przypadku pozornie krótkich w zapisie programów liczba sytuacji, które mogą zaistnieć w praktyce, wyklucza ręczne przetestowanie. Pozostaje więc stosowanie dowodów natury matematycznej, zazwyczaj dość skomplikowanych. Jedną z możliwych ścieżek, którymi można dojść do stwierdzenia formalnej poprawności algorytmu, jest stosowanie *metody niezmienników* (zwanej niekiedy *metodą Floyda*).

Mając dany algorytm, możemy łatwo wyróżnić w nim pewne kluczowe punkty, w których dzieją się interesujące dla danego algorytmu rzeczy. Ich znalezienie nie jest zazwyczaj trudne: ważne są momenty inicjacji zmiennych, którymi będzie operować procedura, testy zakończenia algorytmu, pętla główna itd. W każdym z tych punktów możliwe jest określenie pewnych zawsze prawdziwych warunków — tzw. niezmienników. Można sobie zatem wyobrazić, że dowód formalnej poprawności algorytmu może być uproszczony do stwierdzenia zachowania prawdziwości niezmienników dla dowolnych danych wejściowych.

Dwa typowe sposoby stosowane w praktyce to:

♦ sprawdzanie stanu punktów kontrolnych za pomocą debuggera (odczytujemy wartości pewnych ważnych zmiennych i sprawdzamy, czy zachowują się poprawnie dla pewnych reprezentacyjnych danych wejściowych);

♦ formalne udowodnienie (np. przez indukcję matematyczną) zachowania niezmienników dla dowolnych danych wejściowych.

Zasadniczą wadą powyższych zabiegów jest to, że są nużące i potrafią łatwo zabić całą przyjemność związaną z efektywnym rozwiązywaniem problemów za pomocą komputera.

Każdy program wyprodukowany za pomocą tych metod jest automatycznie poprawny — pod warunkiem, że nie został po drodze popełniony jakiś błąd. Wygenerowanie algorytmu jest możliwe dopiero po jego poprawnym zapisaniu wg schematu:

{warunki wstępne} **poszukiwany program** {warunki końcowe}

Przy pewnej dozie doświadczenia możliwe jest wyprodukowanie ciągu instrukcji, które powodują przejście z „warunków wstępnych” do „warunków końcowych” — wówczas formalny dowód poprawności algorytmu jest zbędny. Można też podejść do problemu z innej strony: mamy dany zespół warunków wstępnych i pewien program, czy jego wykonanie zapewnia ustawienie pożądanych warunków końcowych?