Teoretyczne podstawy informatyki

Wykład 2
Struktury danych i algorytmy
Metody programistyczne

Struktury danych i algorytmy

Struktury danych to **narzędzia** do reprezentowania informacji która ma być przetworzona przez program komputerowy, a **algorytmy** to **przepisy** wykonania czynności niezbędnych do jej przetworzenia.

Wybór algorytmu do rozwiązania konkretnego problemu programistycznego pomaga w ustaleniu, jaką strukturę danych należałoby użyć, ale i odwrotnie – wybrana struktura danych ma ogromny wpływ na szczegóły realizacji i efektywności algorytmu.

Struktury danych i algorytmy

Badanie modelów danych, ich własności oraz sposobów właściwego wykorzystania, stanowi **jeden z filarów informatyki.**

Drugim, **równie ważnym filarem** jest uważna analiza algorytmów i powiązanych z nimi struktur danych.

- Musimy znać najlepsze sposoby wykonywania najczęściej spotykanych żądań, musimy także nauczyć się podstawowych technik projektowania dobrych algorytmów.
- Musimy zrozumieć w jaki sposób wykorzystywać struktury danych i algorytmy tak, by pasowały do procesu tworzenia przydatnych programów.

Typy danych i struktury danych

Są to „obiekty” którymi manipuluje algorytm.
Te obiekty to nie tylko dane wejściowe lub wyjściowe (wyniki działania algorytmu), to również obiekty pośrednie tworzone i używane w trakcie działanie algorytmu.

Dane mogą by różnych typów, do najpospolitszych należą liczby (całkowite, dziesiętne, ułamkowe) i słowa zapisane w rozmaitych alfabetach.

Typy danych i struktury danych

- Interesują nas sposoby w jaki algorytmy mogą organizować, zapamiętywać i zmieniać zbiory danych oraz sięgać do nich.
  - Zmienne czyli „pudelka” w których chwilowo przechowujemy jakąś wartość
  - Wektory, czyli listy takich zmiennych
  - Tablice czyli tabele (macierze), możemy odwoływać się do indeksów w tabeli.
  - Kolejki i stosy czyli „wektory” w których nie wykorzystujemy całej mocy zawartej w mechaniźmie indeksów.
  - Drzewa czyli hierarchiczne ułożenie danych.
  - Zbiory…. Grafy…. Relacje….

Typy danych i struktury danych

W wielu zastosowaniach same struktury danych nie wystarczają. Czasami potrzeba bardzo obszernych zasobów danych, stanowiących dla wielu algorytmów potencjalne dane wejściowe, a więc mające ustaloną strukturę i nadające się do odszukiwania i manipulowania nimi. Nazywa się je bazami danych (relacyjne i hierarchiczne).

Kolejny krok to bazy wiedzy, których elementami są bazy danych, a które zawierają również informacje o związkach pomiędzy danymi.

Algorytmika

- Algorytm to „przepis postępowania” prowadzący do rozwiązania konkretnego zadania; zbiór poleceń dotyczących pewnych obiektów (danych) ze wskazaniem kolejności w jakiej mają być wykonane”.
- Jest jednoznaczną i precyzyjną specyfikacją kroków które mogą być wykonywane „mechanicznie”.
- W matematyce algorytm jest „pojęciem służącym do formułowania i badania rozstrzygalności problemów i teorii”
- Algorytm odpowiada na pytanie „jak to zrobić” postawione przy formułowaniu zadania. Istota algorytmu polega na rozpisaniu całej procedury na kolejne, możliwie elementarne kroki.

➢ **Algorytmiczne myślenie można kształtować niezależnie od programowania komputerów, chociaż każdy program komputerowy jest zapisem jakiegoś algorytmu.**

Sposoby zapisu algorytmu

1. Najprostszy sposób zapisu to **zapis słowny**
   - pozwala określić kierunek działań i odpowiedzieć na pytanie, czy zagadnienie jest możliwe do rozwiązania.
2. Bardziej konkretny zapis to **lista kroków**

   Staramy się zapisać kolejne operacje w postaci kolejnych kroków które należy wykonać

   Budowa listy kroków obejmuje następujące elementy
   - sformułowanie zagadnienia (zadanie algorytmu)
   - określenie zbioru danych potrzebnych do rozwiązania zagadnienia (określenie czy zbiór danych jest właściwy)
   - określenie przewidywanego wyniku (wyników): co chcemy otrzymać i jakie mogą być warianty rozwiązania
   - zapis kolejnych ponumerowanych kroków, które należy wykonać, aby przejść od punktu początkowego do końcowego
3. Bardzo wygodny zapis to **zapis graficzny**
   - Schematy blokowe i grafy

Rodzaje algorytmów

Algorytm liniowy

- Ma postać ciągu kroków które muszą zostać bezwarunkowo wykonane
- jeden po drugim.
- Algorytm taki nie zawiera żadnych warunków ani rozgałęzień: zaczyna się od podania zestawu danych, następnie wykonywane są kolejne kroki wykonawcze, aż dochodzimy do wyniku

Np. dodanie lub mnożenie dwóch liczb

1. **Sformułowanie zadania**: oblicz sumę dwóch liczb naturalnych a,b. Wynik oznacz przez S.
2. **Dane wejściowe**: dwie liczby a i b
3. **Cel obliczeń**: obliczenie sumy S = a + b
4. **Dodatkowe ograniczenia**: sprawdzenie warunku dla danych wejściowych np. czy a, b są naturalne.
   - Ale sprawdzenie takiego warunku sprawia że algorytm przestaje być liniowy

Rodzaje algorytmów

Algorytm z rozgałęzieniem

Większość algorytmów zawiera rozgałęzienia będące efektem sprawdzania warunków. Wyrażenia warunkowe umożliwiają wykonanie zadania dla wielu wariantów danych i rozważanie różnych przypadków.

1. **Sformułowanie zadania**
   Znajdź rozwiązanie równania liniowego postaci a • x + b = 0.
   Wynikiem jest wartość liczbowa lub stwierdzenie dlaczego nie ma jednoznacznego rozwiązania.
2. **Dane wejściowe**
   Dwie liczby rzeczywiste a i b
3. **Cel obliczeń** (co ma być wynikiem)
   Obliczenie wartości x lub stwierdzenie, że równanie nie ma jednoznacznego rozwiązania.
   - gdy a = 0 to sprawdź czy b = 0, jeśli tak to równanie sprzeczne lub tożsamościowe
   - gdy a $\neq$ 0 to oblicz x = -b/a

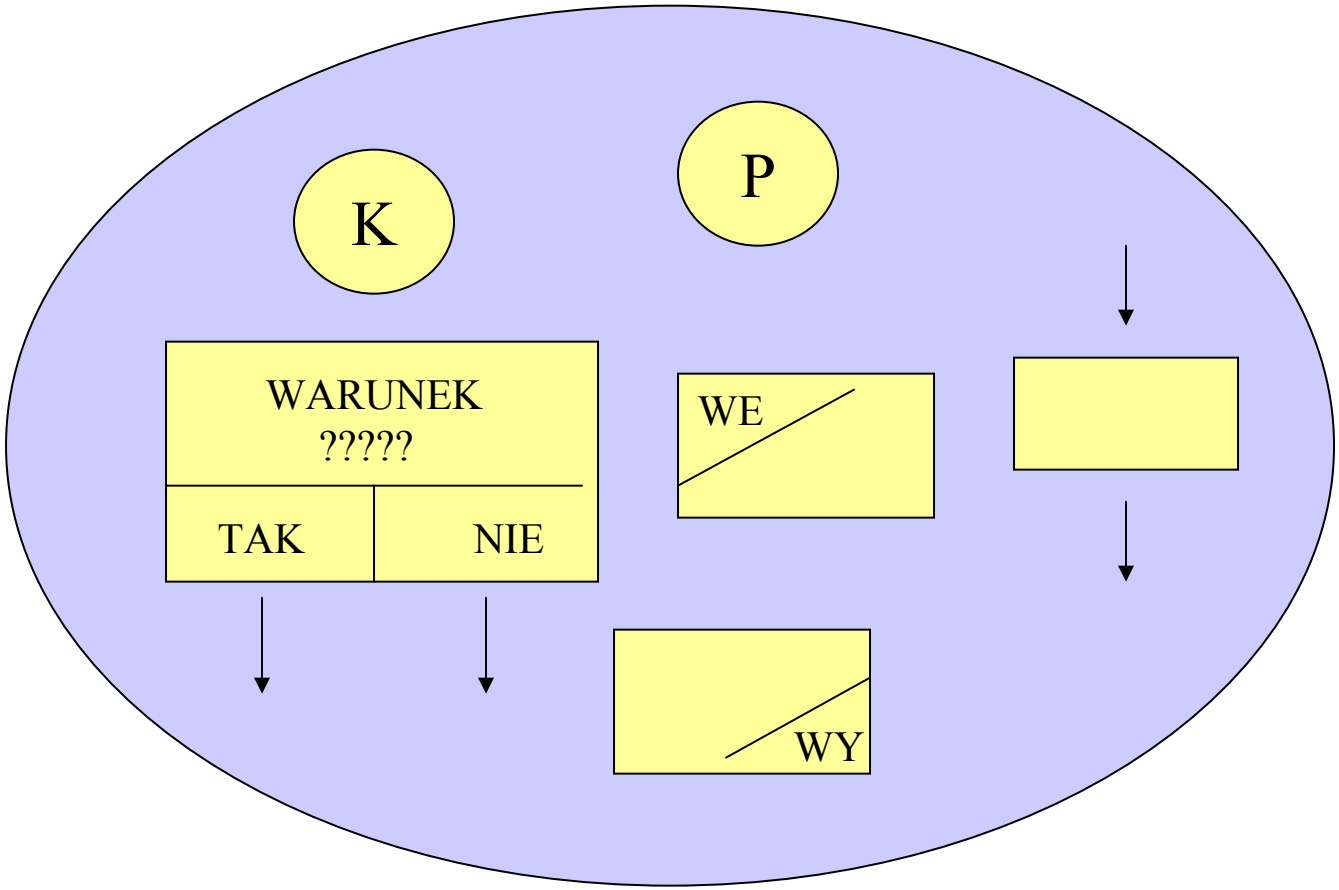
Rodzaje algorytmów

Algorytm z powtórzeniami

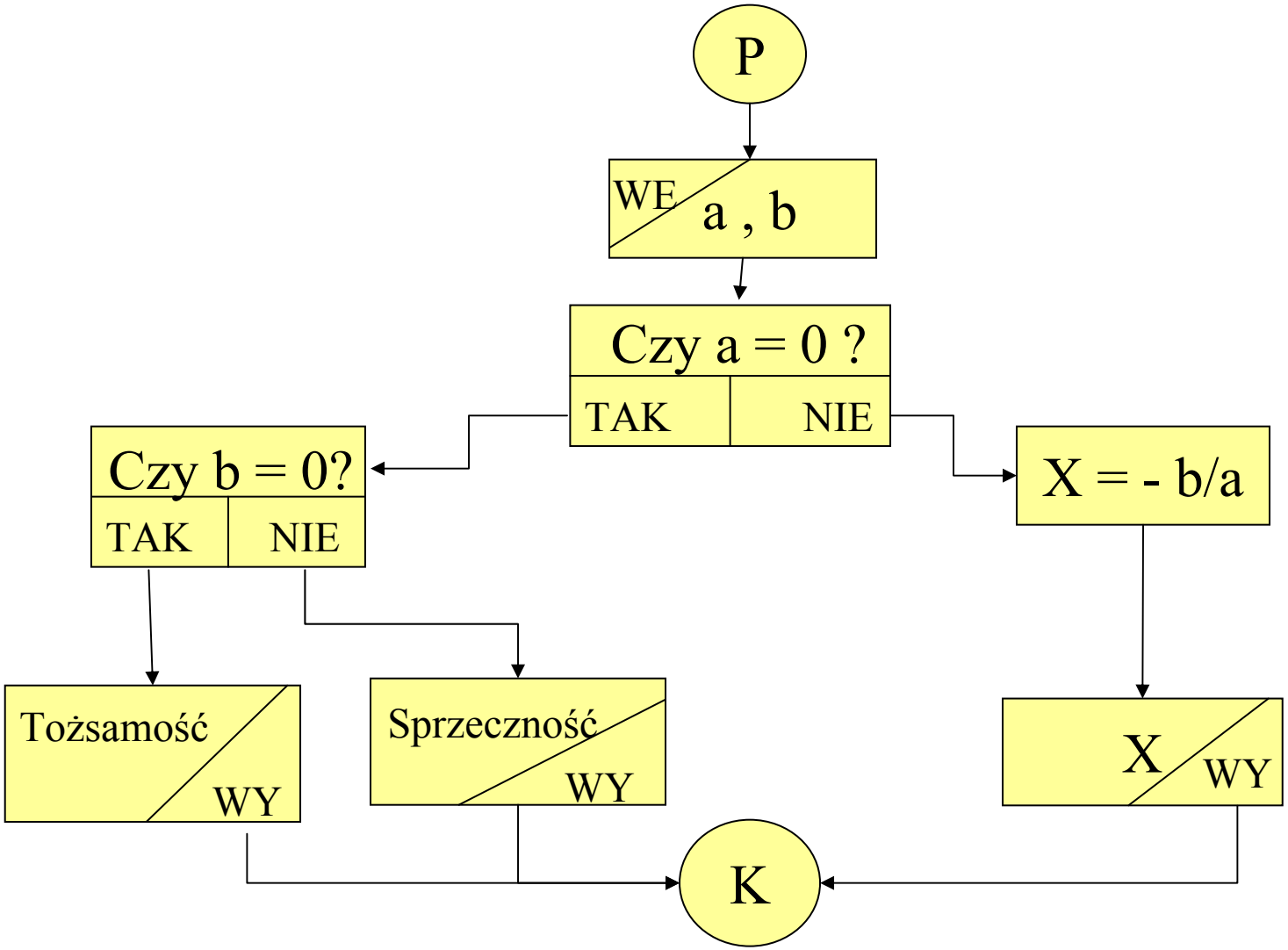
Powtarzanie różnych działań ma dwojaką postać:

- liczba powtórzeń jest z góry określona (przed rozpoczęciem cyklu)
  - najczęściej związany z działaniami na macierzach
- liczba powtórzeń jest nieznana (zależy od spełnienia pewnego warunku)
  - najczęściej związany z obliczeniami typu iteracyjnego

Schematy blokowe i algografy



Schemat blokowy rozwiązania równania liniowego



Grafy

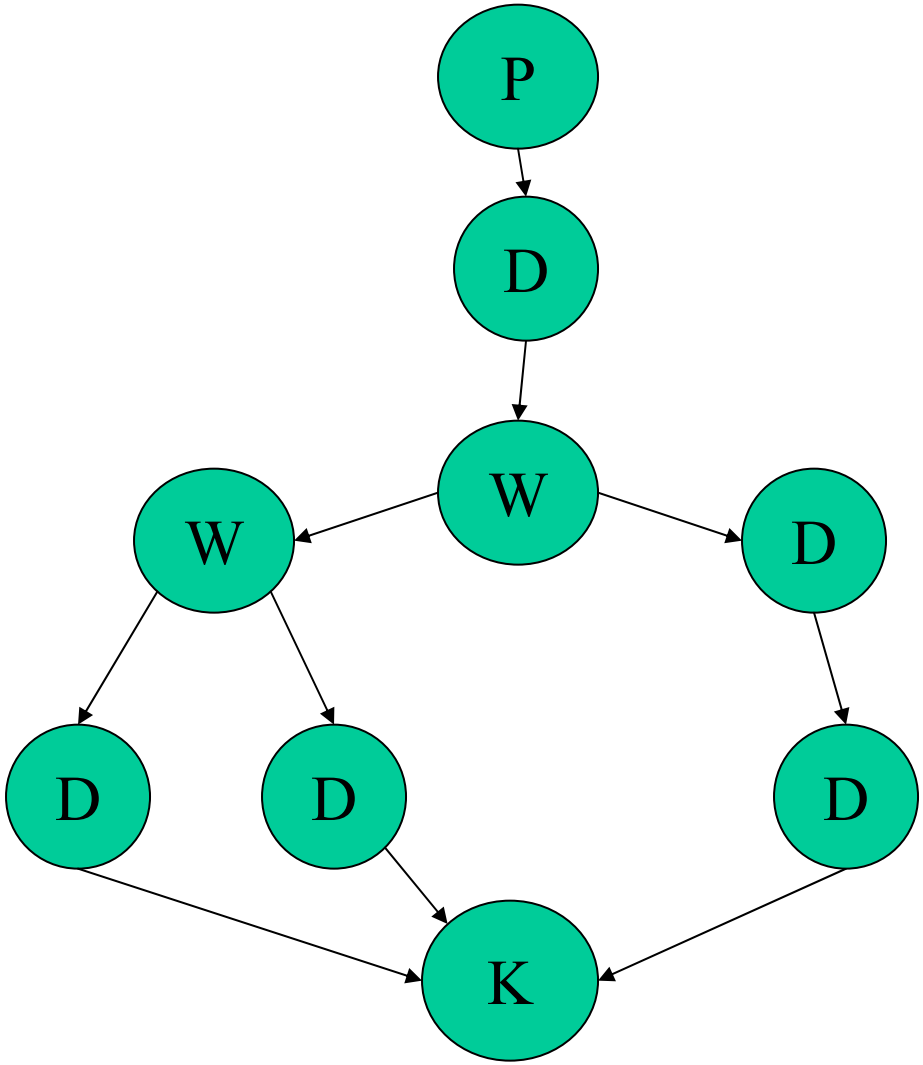
Graf składa się z węzłów i gałęzi.

Graf symbolizuje przepływ informacji.

W przypadku algorytmów graf można wykorzystać aby w uproszczonej formie zilustrować ilość różnych dróg prowadzących do określonego w zadaniu celu.

Graf pozwala także wykryć ścieżki, które nie prowadzą do punktu końcowego, co stanowi podstawową cechę poprawnego algorytmu.

Graf algorytmu rozwiązania równania liniowego



P – początek
K – koniec
D – działanie
W – warunek

Grafy

Jeżeli w grafie znajduje się ścieżka, która nie doprowadza do węzła końcowego, to mamy do czynienia z niepoprawnym grafem. W programie przygotowanym na podstawie takiego grafu, mamy do czynienia z przerwaniem próby działania i komunikatem o zaistnieniu jakiegoś błędu w działaniu.
Węzeł grafu może mieć dwa wejścia jeżeli ilustruje pętle. Wtedy liczba ścieżek początek-koniec może być nieskończona, gdyż nieznana jest liczba obiegów pętli.

- Graf to tylko schemat kontrolny służący do sprawdzenia algorytmu
- Schemat blokowy służy natomiast jako podstawa do tworzenia programów

Algorytmy „dziel i zwyciężaj"

- Dzielimy problem na mniejsze części tej samej postaci co pierwotny.
- Teraz te pod-problemy dzielimy dalej na coraz mniejsze, używając tej samej metody, aż rozmiar problemu stanie się tak mały, że rozwiązanie będzie oczywiste lub będzie można użyć jakiejś innej efektywnej metody rozwiązania.
- Rozwiązania wszystkich pod-problemów muszą być połączone w celu utworzenia rozwiązania całego problemu.

➢ Metoda zazwyczaj implementowana z zastosowaniem technik rekurencyjnych.

Algorytmy „dziel i zwyciężaj"

Jak znaleść minimum ciągu liczb?

Dzielimy ciąg na dwie części, znajdujemy minimum w każdej z nich, bierzemy minimum z obu liczb jako minimum ciągu.

Jak sortować ciąg liczb?

Dzielimy na dwie części, każdą osobno sortujemy a następnie łączymy dwa uporządkowane ciągi (scalamy).

Algorytmy oparte na programowaniu dynamicznym

Można stosować wówczas, kiedy problem daje się podzielić na wiele pod-problemów, możliwych do zakodowania w jedno-, dwu- lub wielowymiarowej tablicy, w taki sposób że w pewnej określonej kolejności można je wszystkie (a więc i cały problem) efektywnie rozwiązać.

Jak obliczać ciąg Fibonacciego?

$$F(i) = \begin{cases} 1 & \text{jeśli } i = 1 \\ 1 & \text{jeśli } i = 2 \\ F(i-2)+F(i-1) & \text{jeśli } i > 2 \end{cases}$$

- Aby obliczyć F(n), wartość np. F(3) musimy wyliczyć F(n-2) razy
- Liczba ta rośnie wykładniczo
- Korzystnie jest więc zachować wyniki wcześniejszych obliczeń

Jak obliczać liczbę kombinacji?

Liczba kombinacji (podzbiorów) r-elementowych ze zbioru n-elementowego, oznaczana $\binom{n}{r}$, dana jest wzorem:

$$\binom{n}{r} = n!/(r!\,(n-r)!)$$

Możemy użyć wzorów:

$$\binom{n}{r} = 1, \text{ jeśli } r = 0 \text{ lub } n = r$$

$$\binom{n}{r} = \binom{n-1}{r-1} + \binom{n}{r-1} \text{ dla } 0 < r < n$$

- Obliczamy rząd po rzędzie w **trójkącie Pascala**

n \ r	0	1	2	3	4
0	1				
1	1	1			
2	1	2	1		
3	1	3	3	1	
4	1	4	6	4	1

Algorytmy z powrotami

- **Przykładami tego typu algorytmów są gry.**

- Często możemy zdefiniować jakiś problem jako poszukiwanie jakiegoś rozwiązania wśród wielu możliwych przypadków.
- Dana jest pewna przestrzeń stanów, przy czym stan jest to sytuacja stanowiąca rozwiązanie problemu albo mogąca prowadzić do rozwiązania oraz sposób przechodzenia z jednego stanu do drugiego.
- Czasami mogą istnieć stany które nie prowadzą do rozwiązania.

Algorytmy z powrotami

Metoda powrotów

Wymaga zapamiętania wszystkich wykonanych ruchów czy też wszystkich odwiedzonych stanów aby możliwe było cofanie posunięć.

Stanów mogą być tysiące lub miliony więc bezpośrednie zastosowanie metody powrotów, mogące doprowadzić do odwiedzenia wszystkich stanów, może być zbyt kosztowne. Inteligentny wybór następnego posunięcia, **funkcja oceniająca**, może znacznie poprawić efektywność algorytmu.
Np. aby uniknąć przeglądania nieistotnych fragmentów przestrzeni stanów.

Wybór algorytmu

Regułą jest że należy implementować algorytmy najprostsze, które wykonują określone zadanie.

Prosty algorytm to

- łatwiejsza implementacja, czytelniejszy kod
- łatwość testowania
- łatwość pisania dokumentacji,….

Jeśli program ma działać wielokrotnie, jego wydajność i wykorzystywany algorytm stają się bardzo ważne. W ogólności, efektywność wiąże się z czasem potrzebnym programowi na wykonanie danego zadania. Istnieją również inne zasoby, które należy niekiedy oszczędnie wykorzystywać w pisanych programach.

- ilość przestrzeni pamięciowej wykorzystywanej przez zmienne
- generowane przez program obciążenie sieci komputerowej
- ilość danych odczytywanych i zapisywanych na dysku

Wybór algorytmu

Zrozumiałość i ***efektywność*** to są często sprzeczne cele. Typowa jest sytuacja w której programy efektywne dla dużej ilości danych są trudniejsze do napisania/zrozumienia.

Np. sortowanie przez wybieranie (łatwy, nieefektywny dla dużej ilości danych) i sortowanie przez scalanie (trudniejszy, dużo efektywniejszy).

Zrozumiałość to pojecie względne, natomiast *efektywność można obiektywnie zmierzyć*.

Metodyka: *testy wzorcowe, analiza złożoności obliczeń*

Efektywność algorytmu

Testy wzorcowe

Podczas porównywania dwóch lub więcej programów zaprojektowanych do wykonywania tego samego zadania, opracowujemy niewielki zbiór typowych danych wejściowych które mogą posłużyć jako *dane wzorcowe* (ang. benchmark).

Powinny być one reprezentatywne i zakłada się że program dobrze działający dla danych wzorcowych będzie też dobrze działał dla wszystkich innych danych.

Np. test wzorcowy umożliwiający porównanie algorytmów sortujących może opierać się na jednym ***małym*** zbiorze danych, np. zbiór pierwszych 20 cyfr liczby p; jednym ***średnim***, np. zbiór kodów pocztowych województwa krakowskiego; oraz na ***dużym*** zbiorze takim jak zbiór numerów telefonów z obszaru Krakowa i okolic.

Przydatne jest też sprawdzenie jak algorytm działa dla ciągu już posortowanego (często działają kiepsko).

Efektywność algorytmu

Czas działania

Oznaczamy przez funkcje **T(n)** liczbę jednostek czasu, które zajmuje wykonanie programu lub algorytmu w przypadku problemu o rozmiarze **n**. Funkcje te nazywamy *czasem działania*. Dość często czas działania zależy od konkretnych danych wejściowych, nie tylko ich rozmiaru. W takim przypadku, 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.

Wyidealizowany scenariusz tworzenia oprogramowania

1. *Definicja problemu i specyfikacja*

   Analiza wymagań użytkownika (często nieprecyzyjne i trudne do zapisania), budowa prostego prototypu lub modelu systemu
2. *Projekt*

   Wyróżniamy najważniejsze komponenty, specyfikujemy wymagania związane z wydajnością systemu, szczegółowe specyfikacje niektórych komponentów
3. *Implementacja*

   Każdy implementowany komponent poddajemy serii testów
4. *Integracja i testowanie systemu*
5. *Instalacja i testowanie przez użytkowników*
6. *Konserwacja*

   Niezwykle istotna jakość stylu programowania (w wielu wypadkach to ponad 50% nakładów poniesionych na napisanie systemu)

Metody programistyczne

Przy rozwiązywaniu prostych zadań **podejście „ad hoc”** może dawać szybsze rezultaty. Jednak, gdy mamy do czynienia z bardziej złożonymi problemami efektywniejsze jest **podejście systematyczne**

Projektowanie zstępujące (top-down design)

- Rozpoczynamy od zdefiniowania problemu który chcemy rozwiązać
- Problem dzielimy na główne kroki – pod-problemy
- Pod-problemy są dzielone na drobniejsze kroki tak długo, aż rozwiązania drobnych pod-problemów stają się łatwe .... nazywamy to stopniowym uszczegółowianiem
- W idealnej sytuacji pod-problemy mogą być rozwiązywane niezależnie, a ich rozwiązania łączone w celu otrzymania rozwiązania całego problemu. W odniesieniu do tworzenia kodu oznacza to, że poszczególne kroki powinny być kodowane niezależnie, przy czym dane wyjściowe jednego kroku są używane jako dane wejściowe do innego

Realnie

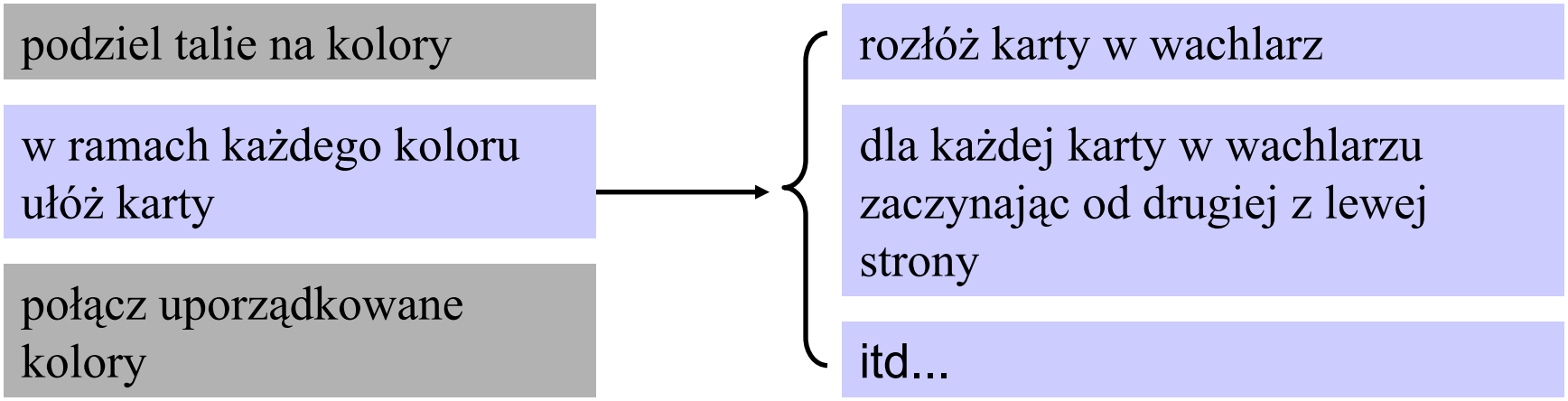
- celem projektowania zstępującego jest **zminimalizowanie tych współzależności**

Metody programistyczne

Zalety projektowania zstępującego

- Systematyczna metoda rozwiązywania problemów
  - rozwiązanie jest **modularne**, poszczególne kroki mogą być uruchamiane, modyfikowane i ulepszane niezależnie od pozostałych, rozwiązanie składa się z klarownych fragmentów które można niezależnie zrozumieć
  - dobrze zaprojektowane fragmenty mogą być **ponownie zastosowane** w innych zadaniach
  - można rozpoznać wspólne problemy na samym początku i **uniknąć wielokrotnego ich rozwiązywania**

Przykład: Uporządkuj talie kart:



Metody programistyczne

Projektowanie wstępujące (bottom-up design)

- Polega na wyjściu od samego języka i wzbogacaniu go nowymi operacjami, dopóki nie będzie można wyrazić rozwiązania problemu w rozszerzonym języku
- Każde oprogramowanie działające na maszynie cyfrowej rozszerza jej możliwości o nowe funkcje. W efekcie, uruchomienie programu na komputerze tworzy nową maszynę, która nie wykonuje swoich operacji bezpośrednio, ale zmienia je na prostsze wykonywalne przez sprzęt
- Nową maszynę nazywamy **maszyną wirtualną**, ponieważ istnieje w świecie abstrakcyjnym a nie fizycznym
- To samo dotyczy sytuacji gdy dodajemy nową operację do języka programowania, pisząc funkcje. Zwiększa ona jego funkcjonalność tworząc nowy, silniejszy język. O zbiorze nowych funkcji można myśleć jako o wirtualnej maszynie zbudowanej na bazie starego języka.

Metody programistyczne

Przykład: Policz wyrażenie (3/28 + 2/7) * 4/3 –1

Aby to wykonać należałoby wzbogacić istniejący język przez dodanie

- funkcji implementujących działania +, -, * , / na ułamkach
- funkcji wczytywania i wypisywania ułamków
- ten pakiet funkcji mógłby też zawierać funkcję skracającą ułamki, chociaż nasz program nie musiałby z niej korzystać

Zazwyczaj programy projektuje się **łącząc metodę wstępująca i zstępująca**

- Zaczynamy od podzielenia problemu na pod-zadania
- Przekonujemy się, że byłby przydatny określony pakiet funkcji
- Rozstrzygamy, jakie funkcje wejdą w skład pakietu i rozszerzą język
- Powtarzamy to iteracyjnie, dzieląc problemy na prostsze i wzbogacają tym samym język.... dopóki rozwiązania wszystkich problemów składowych nie dadzą się zapisać bezpośrednio w rozszerzonym języku

Metody programistyczne

Abstrakcyjne typy danych (ATD)

Jest to istotne ulepszenie metody projektowania programów. Podstawą metody jest **oddzielenie operacji wykonywanych** na danych i sposobów ich przechowywania **od konkretnego typu danych**. Pozwala ona na podzielenie zadania programistycznego na dwie części:

- wybór struktury danych reprezentującej ATD i napisanie funkcji implementujących operacje
- napisanie „programu głównego" który będzie wywoływał funkcje ATD

Program ma dostęp do danych ATD wyłącznie przez wywołanie funkcji; nie może bezpośrednio odczytywać ani modyfikować wartości przechowywanych w wewnętrznych strukturach ATD.

Weryfikacja poprawności programu

Sprawdzenie częściowej poprawności:

Weryfikacja opiera się o sprawdzenie specyfikacji, która jest czymś niezależnym od kodu programu.
Specyfikacja wyraża, co program ma robić, i określa związek miedzy jego danymi wejściowymi i wyjściowymi.

W specyfikacji definiujemy warunek dotyczący danych wejściowych, który musi być spełniony na początku programu, tzw. ***warunek wstępny***. Jeżeli dane nie spełniają tego warunku, to nie ma gwarancji że program będzie działał poprawnie ani nawet że się zatrzyma.

W specyfikacji definiujemy również ***warunek końcowy***, czyli co oblicza program przy założeniu że się zatrzyma. Warunek końcowy jest zawsze prawdziwy jeżeli zachodzi warunek wstępny.

Weryfikacja poprawności programu

Aby wykazać, że program jest zgodny ze specyfikacją, trzeba podzielić ją na wiele kroków, podobnie jak dzieli się na kroki sam program. Każdy krok programu ma swój warunek wstępny i warunek końcowy.

Prosta instrukcja przypisania

`x = v` , x-zmienna, v-wyrażenie

warunek wstępny: $x \geq 0$
wyrażenie: $x = x+1$
warunek końcowy: $x \geq 1$

Dowodzimy poprawności specyfikacji przez podstawienie wyrażenia x+1 pod każde wystąpienie x w warunku końcowym.
Otrzymujemy formułę $x+1 \geq 1$, która wynika z warunku wstępnego $x \geq 0$

- Zatem specyfikacja tej instrukcji podstawienia jest poprawna

Weryfikacja poprawności programu

Istnieją **cztery sposoby** łączenia prostych kroków w celu otrzymania kroków bardziej złożonych.

1. stosowanie instrukcji złożonych (bloków instrukcji wykonywanych sekwencyjnie)
2. stosowanie instrukcji wyboru
3. stosowanie instrukcji powtarzania (pętli)
4. wywołania funkcji

Weryfikacja poprawności programu

Każdej z tych metod budowania kodu odpowiada metoda przekształcania warunku wstępnego i końcowego w celu wykazania poprawności specyfikacji kroku złożonego:

- 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**.

Niezmienniki pętli

Niezmiennik pętli określa warunki jakie muszą być zawsze spełnione przez zmienne w pętli, a także przez wartości wczytane lub wypisane (jeżeli takie operacje zawiera).

Warunki te muszą być prawdziwe przed pierwszym wykonaniem pętli oraz po każdym jej obrocie.
(mogą stać się chwilowo fałszywe w trakcie wykonywania wnętrza pętli, ale musza ponownie stać się prawdziwe przy końcu każdej iteracji)

Dowodzimy że pewne zdanie jest niezmiennikiem pętli wykorzystując **zasadę indukcji matematycznej**. Mówi ona że:

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

Dowodzenie niezmienników pętli

- Stwierdzenie jest prawdziwe przed pierwszym wykonaniem pętli, ale po dokonaniu całej inicjacji; wynika ono z warunku wstępnego pętli.
- Jeśli założymy, że stwierdzenie jest prawdziwe przed jakimś przebiegiem pętli i że pętla zostanie wykonana ponownie, a więc warunek pętli jest spełniony, to stwierdzenie będzie nadal prawdziwe po kolejnym wykonaniu wnętrza pętli.

Proste pętle maja zazwyczaj proste niezmienniki.

Pętle dzielimy na trzy kategorie

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

Problem „STOP-u"

Aby udowodnić że program się zatrzyma, musimy wykazać, że zakończą działanie wszystkie pętle programu i wszystkie wywołania funkcji rekurencyjnych.

- Dla pętli z wartownikiem wymaga to umieszczenia w warunku wstępnym informacji, że dane wejściowe zawierają wartownika.
- Dla pętli z licznikiem wymaga to dodefiniowania granicy dolnej (górnej) tego licznika.

Uwagi końcowe

Istnieje wiele innych elementów, które muszą być brane pod uwagę przy dowodzeniu poprawności programu:

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

Na wybór najlepszego algorytmu dla tworzonego programu wpływa wiele czynników, najważniejsze to:

- prostota,
- łatwość implementacji
- efektywność