Paradygmaty Programowania

Marcin Bączyk (prowadzący) Adam Wojtasik(autor prezentacji)

Wykład 2

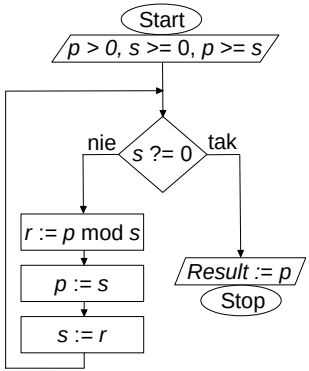
1 marca 2021

Treść dzisiejszego wykładu

- algorytm
  - definicja
  - złożoność
- problem obliczeniowy
  - definicja
  - funkcja celu
  - problem optymalizacyjny
  - problem decyzyjny
  - maszyna Turinga
  - klasy problemów decyzyjnych

Algorytm

Przepis postępowania prowadzący do rozwiązania danego zadania, określający sekwencję czynności elementarnych, które należy w tym celu wykonać



Graficzna reprezentacja algorytmu znajdowania największego wspólnego dzielnika dwóch liczb. Algorytm ten został zdefiniowany przez Euklidesa w IV w. p.n.e.

Cechy poprawnego algorytmu

- Istnieje skończony ciąg jednoznacznych symboli, którymi można go wyrazić.
- Może być przedstawiony w różnych postaciach (postać graficzna, język naturalny – mówiony czy pisany, pseudokod, język programowania, kod maszynowy, itd.).
- Jest realizowalny (można go wykonać w dowolny sposób bez i/lub z użyciem jakiegoś aparatu abstrakcyjnego, narzędzia mechanicznego czy elektronicznego).
- Może być wykonywany wielokrotnie i dla różnych danych.
- Jest poprawny semantycznie, tzn. dla każdych poprawnych danych wejściowych będzie znaleziony wynik i działanie algorytmu się zakończy (tzw. własność stopu).
- Jest kompletny, tzn. uwzględnia wszystkie możliwe przypadki, które mogą wystąpić w trakcie jego realizacji.
- Jest jednoznaczny, tzn. dla tych samych danych wejściowych musi dawać zawsze ten sam wynik.

Algorytm Euklidesa

- Język „ludzki”:
  Weź dwie liczby. Jeżeli druga jest równa zero, rozwiązaniem jest liczba pierwsza. Jeżeli nie jest równa zero, oblicz resztę z dzielenia pierwszej liczby przez drugą. Druga liczba staje się pierwszą liczbą, reszta z dzielenia staje się drugą liczbą. Przejdź do porównania drugiej liczby z zerem.
- Pyton:

```
def nwd(p, s):
    while s:
        p, s = s, p%s
    return p
```

- C, C++, C#, Java:

```
int nwd(int p, int s){
    return s ? nwd(s,p%s) : p;
}
```

Algorytm numeryczny

Algorytm numeryczny to metoda rozwiązywania problemów matematycznych za pomocą operacji na liczbach.

- Zazwyczaj uzyskujemy tylko wyniki przybliżone.
- Dokładność obliczeń może być z góry określona i można ją dobrać w zależności od potrzeb.
- Zazwyczaj generuje pewien ciąg liczb, którego kolejne wyrazy są coraz bliższe dokładnego rozwiązania.
- Najczęściej ma postać iteracyjną, tzn. polegają na wielokrotnym wykonywaniu tego samego zestawu operacji.
- Zazwyczaj ma prostą postać, choć często stoi za nim zaawansowana matematyka.
- Obliczenia za jego pomocą najczęściej wykonują komputery.

Algorytm nienumeryczny

Algorytm nienumeryczny operuje na obiektach nieliczbowych (np. książki, strony w książce, wyrazy w tekście, litery w tekście itd., itp.).
Ale:

- Obiekty te mogą być jednak reprezentowane przez liczby (np. numery stron).
- Zastosowane w algorytmie operacje mogą być (i najczęściej są) operacjami matematycznymi (np. przekręcenie kartki w książce => dodanie 1 do numeru aktualniej strony).

Złożoność algorytmu

Złożoność obliczeniowa

Funkcja zależności liczby operacji potrzebnych do uzyskania wyniku od rozmiaru danych wejściowych

Złożoność czasowa

Funkcja zależności czasu potrzebnego do uzyskania wyniku od rozmiaru danych wejściowych

Złożoność pamięciowa

Funkcja zależności wielkości pamięci potrzebnej do przeprowadzenia algorytmu od rozmiaru danych wejściowych

---

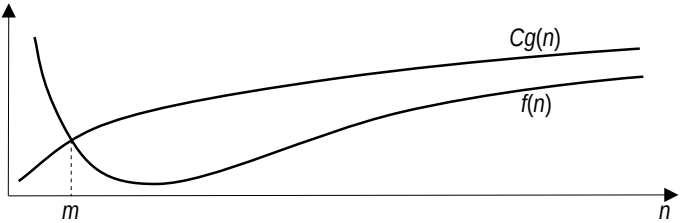
Złożoność algorytmu określa się stosując notację asymptotyczną.

Notacja asymptotyczna – mniejszość

Funkcja $f(n)$ jest jest asymptotycznie mniejsza od funkcji $g(n)$, jeżeli dla każdego rzeczywistego $C > 0$ istnieje takie m (naturalne), że dla każdego $n > m$ zachodzi $f(n) < C \cdot g(n)$.

Zbiór wszystkich funkcji asymptotycznie mniejszych od $g(n)$ oznaczamy przez $o(g(n))$.

Mówimy, że $f(n)$ jest niższego rzędu $g(n)$, co oznaczamy: $f(n) = o(g(n))$



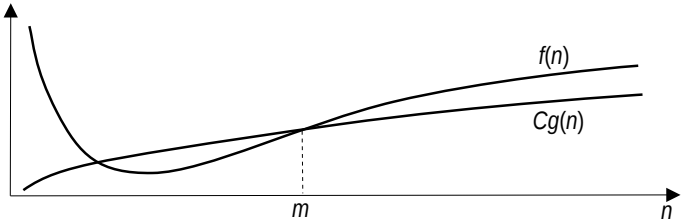
Twierdzenie o granicach: $f(n) = o(g(n)) \Leftrightarrow \lim_{x \to \infty} \frac{f(n)}{g(n)} = 0$

Notacja asymptotyczna – większość

Funkcja $f(n)$ jest jest asymptotycznie większa od funkcji $g(n)$, jeżeli dla każdego rzeczywistego $C > 0$ istnieje takie m (naturalne), że dla każdego $n > m$ zachodzi $C \cdot g(n) < f(n)$.

Zbiór wszystkich funkcji asymptotycznie większych od $g(n)$ oznaczamy przez $\omega(g(n))$.

Mówimy, że $f(n)$ jest wyższego rzędu niż $g(n)$, co oznaczamy: $f(n) = \omega(g(n))$



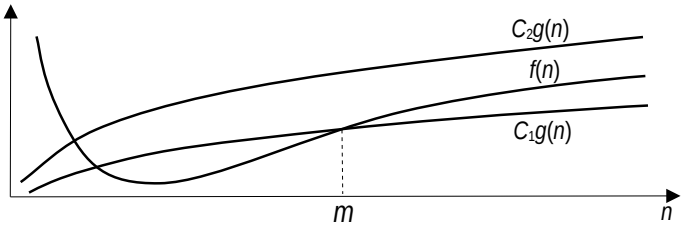
Twierdzenie o granicach: $f(n) = \omega(g(n)) \Leftrightarrow \lim_{x\to\infty} \frac{f(n)}{g(n)} = \infty$

Notacja asymptotyczna – dokładność (podobieństwo)

Funkcja $f(n)$ jest jest asymptotycznie dokładna z funkcją $g(n)$, jeżeli istnieją takie rzeczywiste $C_1, C_2 > 0$ i takie m (naturalne), że dla każdego $n > m$ zachodzi $C_1 \cdot g(n) \leq f(n) \leq C_2 \cdot g(n)$.

Zbiór wszystkich funkcji asymptotycznie dokładnych z $g(n)$ oznaczamy przez $\Theta(g(n))$.

Mówimy, że $f(n)$ jest tego samego rzędu co $g(n)$, oznaczamy to jako $f(n) = \Theta(g(n))$



Twierdzenie o granicach: $f(n) = \Theta(g(n)) \Leftrightarrow \lim_{x \to \infty} \frac{f(n)}{g(n)} = G > 0$

Zmieniając w definicjach na mniejszość i większość wyrażenie „dla każdego $C > 0$” na „istnieje takie $C > 0$” oraz zastępując nierówności ostre nierównościami nieostrymi otrzymamy definicje na asymptotyczną niewiększość i asymptotyczną niemniejszość z oznaczeniami odpowiednio $O(g(n))$, $\Omega(g(n))$.

W przypadku złożoności algorytmów najbardziej interesuje nas rząd funkcji opisującej tę złożoność. Jest podstawowym wyznacznikiem jakości tego algorytmu.

Wyznaczanie złożoności algorytmu – przykład

Niech pewien algorytm polega na wykonaniu obliczeń w dwóch pętlach (jedna zawiera się w drugiej), z których zewnętrzna wykonywana jest n razy, a wewnętrzna $\lg n$ razy.

$$f(n) = a_3 + \underbrace{n(a_2 + a_1 \underbrace{\lg n}_{\text{wew}})}_{\text{zew}} = a_3 + a_2 n + a_1 n \lg n$$

Dla $g(n) = n$ $\quad \lim_{x\to\infty} \frac{f(n)}{g(n)} = \lim_{x\to\infty}(\frac{a_3}{n} + a_2 + a_1 \lg n) = \infty$

Dla $g(n) = n^2$ $\quad \lim_{x\to\infty} \frac{f(n)}{g(n)} = \lim_{x\to\infty}(\frac{a_3}{n^2} + \frac{a_2}{n} + \frac{a_1 \lg n}{n}) = 0$

Dla $g(n) = n \lg n$ $\quad \lim_{x\to\infty} \frac{f(n)}{g(n)} = \lim_{x\to\infty}(\frac{a_3}{n \lg n} + \frac{a_2}{\lg n} + a_1) = a_1$

Czyli: $f(n) = \omega(n)$; $\quad f(n) = o(n^2)$; $\quad f(n) = \Theta(n \lg n)$

Obliczanie złożoności – wnioski

1. Złożoność określana jest przez najbardziej znaczący człon funkcji i nie zależy od rodzaju mniej znaczących członów.

$$n^3 + n^2 + n \quad \text{oraz} \quad n^3 - \log n \quad \text{są rzędu} \quad \Theta(n^3)$$

2. Wszelkie stałe i współczynniki nie mają znaczenia dla określenia rzędu złożoności.

$$5n^2 + n - 1 \quad \text{oraz} \quad 2n^2 + 57n - 244 \quad \text{są rzędu} \quad \Theta(n^2)$$

Nie wyklucza to jednak stwierdzenia, że algorytm o złożoności $5n^2 + n - 1$ jest mniej sprawny od algorytmu o złożoności $2n^2 + 57n - 244$ (na co wskazują wartości przy najbardziej znaczącym członie).

Złożoność obliczeniowa algorytmu

Ma charakter teoretyczny – dla danego algorytmu można ją matematycznie obliczyć bądź oszacować, udowodnić jej formalne właściwości itp.

Uwaga: Pojęcie operacji, które występuje w definicji złożoności obliczeniowej, nie jest pojęciem jednozanacznym. W zależności od algorytmu może to być operacja elementarna (np. mnożenie), nieco bardziej złożona (np. przestawienie danych w komórkach pamięci z użyciem bufora), a nawet bardzo skomplikowana (implementacja całego innego algorytmu, traktowanego jako pomocniczy).

Złożoność czasowa algorytmu

Ma charakter praktyczny (empiryczny), związany z implementacją – zwykle złożoność tę wyznacza się doświadczalnie.

- Nie wszystkie operacje wykonywane w trakcie działania algorytmu są istotne dla złożoności czasowej. Czas działania algorytmu zależy przede wszystkim od tzw. operacji dominujących.
- Różne operacje mogą być dominujące dla tego samego algorytmu. Może to zależeć od:
  - sposobu implementacji,
  - użytego sprzętu.

Złożoność czasowa algorytmu

Przykład (dość skrajny, ale jednak teoretycznie możliwy przypadek): Niech w pewnym algorytmie wystpuje wielokrotne w każdej operacji pobieranie liczb z pamięci zewnętrznej (ze złożonością obliczeniową n^3) w celu ich pomnożena (n^2), a następnie zapisywanie wyniku.

- Implementacja A algorytmu korzysta tylko z pamięci zewnętrznej o wolnym dostępie.
- Implementacja B algorytmu umożliwia przechowywanie pobranej liczby w pamięci podręcznej o szybkim dostępie (nie trzeba liczby pobiera drugi raz z pamięci zewnętrznej).
- Mamy do dyspozycji dwa komputery:
  - K1, w którym czas dostępu do pamięci zewnętrznej jest pomijalnie mały w porównaniu z czasem mnożenia,
  - K2 z relacją odwrotną.

Jaką będzie złożoność czasowa różnych wersji algorytmów uruchamianych na różnych komputerach?

Złożoność bardzo dużej liczby algorytmów zależy od konfiguracji danych wejściowych

Na przykład algorytm służący do wyszukiwania konkretnego elementu w tablicy (przegląd bezpośredni tablicy):

- jeżeli element ten jest na pierwszym miejscu: $\Theta(1)$,
- jeżeli element ten jest na ostatnim miejscu: $\Theta(n)$,
- w pozostałych przypadkach (statystycznie): $\Theta(\frac{n}{2})$.

Można mówić zatem o:

- złożoności optymistycznej (czyli dla najlepszego przypadku),
- złożoności pesymistycznej (czyli dla najgorszego przypadku),
- złożoności oczekiwanej (typowej, prawdopodobnej).

Klasy złożoności

Rząd funkcji definiującej złożoność obliczeniową algorytmu decyduje o przynależności tego algorytmu do jakiejś klasy złożoności

1. Algorytm o złożoności $f(n) = \Theta(g(n))$ należy do klasy złożoności $g(n)$
2. Klasa $g_1(n)$ jest niższa od klasy $g_2(n)$, jeżeli $g_1(n) = o(g_2(n))$
3. Sprawność dwóch algorytmów należących do tej samej klasy porównać można na podstawie relacji ich współczynników C w ich zależnościach $\Theta(Cg(n))$

Klasy złożoności

k	stała[1]	
$\lg n$	logarytmiczna	Grupa
n	liniowa	algorytmów
$n \lg n$	liniowo-logarytmiczna	wielomianowych
n^k	wielomianowa	
$n^{\lg n}$	podwykładnicza	Grupa
k^n	wykładnicza	algorytmów
$n!$	silniowa	ponadwielomianowych
n^n	nadwykładnicza	(wykładniczych)
∞	nierozwiązywalne[2]	

[1] W praktyce nie ma „normalnych" algorytmów należących do tej klasy. Dotyczy ona elementarnych przypadków – algorytmów składających się z jednej prostej (nawet jeśli składa się z kilku czynności) operacji (np. algorytm płacenia bezgotówkowego: wyjmij kartę, zbliż do terminala, wstukaj pin). Wykorzystuje się tę klasę także przy określaniu „najlepszego przypadku".

[2] Klasa ta nie jest klasą złożoności algorytmów *sensu stricte* – oznacza, że nie istnieje algorytm rozwiązujący dane zadanie.

Klasy złożoności – ciekawostka

Znanych jest wiele zagadnień nierozwiązywalnych (problemów nierozstrzygalnych) – głównie z obszaru „wyższej" matematyki i logiki.

Dla nas najciekawszym takim zagadnieniem jest tzw. problem weryfikacji własności stopu. Udowodniono matematycznie, że nie istnieje ogólny algorytm pozwalający stwierdzić, czy dowolny algorytm na pewno w każdym przypadku znajdzie rozwiązanie i zatrzyma się (czyli czy algorytm ten ma własność stopu).

Inaczej mówiąc: nie będziemy w stanie odpowiedzieć na pytanie, czy dany program komputerowy nigdy nie wpadnie w wykonywanie nieskończonej pętli.

Klasy złożoności – przykład

Załóżmy, że mamy cztery algorytmy, które rozwiązanie swoich zadań o rozmiarze danych wejściowych = 1 pozwalają uzyskać w ciągu 1 sekundy. Ile trzeba czasu na rozwiązanie zadań o rozmiarze 10 i 100?

Θ	1	10	100
n	1 sek	?	?
$n \lg n$	1 sek	?	?
n^2	1 sek	?	?
2^n	1 sek	?	?

Klasy złożoności – przykład

Załóżmy, że mamy cztery algorytmy, które rozwiązanie swoich zadań o rozmiarze danych wejściowych = 1 pozwalają uzyskać w ciągu 1 sekundy. Ile trzeba czasu na rozwiązanie zadań o rozmiarze 10 i 100?

Θ	1	10	100
n	1 sek	10 sek	?
$n \lg n$	1 sek	ok. 10 sek	?
n^2	1 sek	1,6 min	?
2^n	1 sek	8,5 min	?

Klasy złożoności – przykład

Załóżmy, że mamy cztery algorytmy, które rozwiązanie swoich zadań o rozmiarze danych wejściowych = 1 pozwalają uzyskać w ciągu 1 sekundy. Ile trzeba czasu na rozwiązanie zadań o rozmiarze 10 i 100?

Θ	1	10	100
n	1 sek	10 sek	1,6 min
$n \lg n$	1 sek	ok. 10 sek	3,3 min
n^2	1 sek	1,6 min	2,7 godz
2^n	1 sek	8,5 min	?

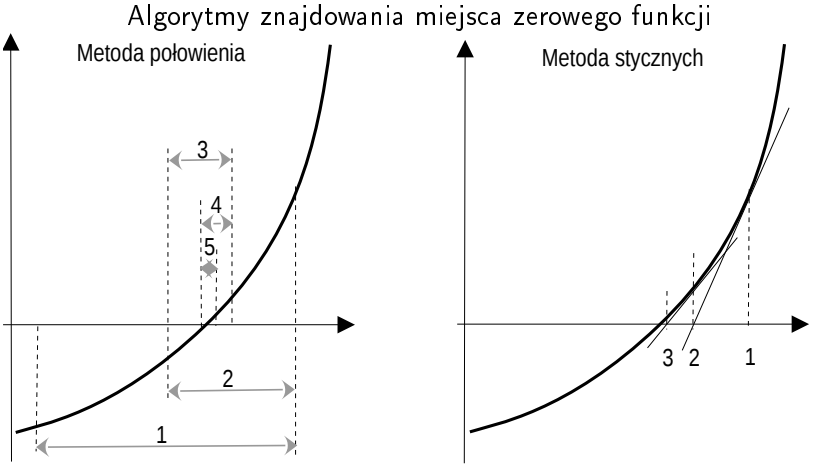
Klasy złożoności – przykład

Załóżmy, że mamy cztery algorytmy, które rozwiązanie swoich zadań o rozmiarze danych wejściowych = 1 pozwalają uzyskać w ciągu 1 sekundy. Ile trzeba czasu na rozwiązanie zadań o rozmiarze 10 i 100?

Θ	1	10	100
n	1 sek	10 sek	1,6 min
$n \lg n$	1 sek	ok. 10 sek	3,3 min
n^2	1 sek	1,6 min	2,7 godz.
2^n	1 sek	8,5 min	$2,06 \cdot 10^{22}$ lat

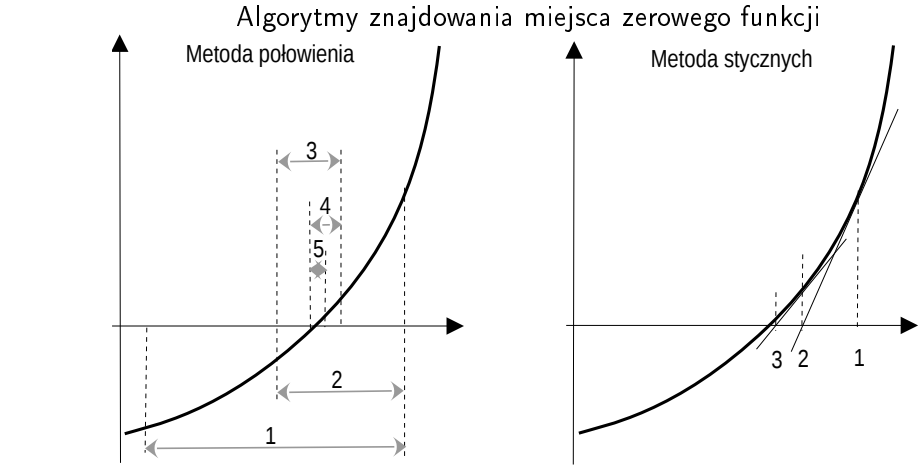
Czyli algorytmy ponadwielomianowe mają praktyczne znaczenie **wyłącznie** dla **małych rozmiarów** danych wejściowych! Dla dużych rozmiarów są **bezużyteczne!**

Złożoność algorytmu numerycznego?



Co tu jest rozmiarem zadania?

Złożoność algorytmu numerycznego?



Mausimy tu porównywać inne własności algorytmów, a nie ich złożoności obliczeniowe (bo takich nie ma).

Porównanie algorytmów numerycznych

- Metoda połowienia wymaga wstępnego zlokalizowania „ujemnego" i „dodatniego" punktu startowego, czyli zastosowania na początku dodatkowego algorytmu.
- Obie metody są zbieżne, ale metoda stycznych szybciej zbiega do rozwiązania, niż metoda połowienia.
- Ale zbieżność metod nie jest dana „raz na zawsze", bo zależy od rodzaju funkcji – metoda stycznych może być rozbieżna, jeżeli między punktem startowym a miejscem zerowym znajduje się punkt przegięcia.
- W metodzie stycznych relacja uzyskanego rozwiązania rzeczywistego miejsca zerowego (większe – mniejsze) określona jest przez punkt startowy, więc jest znana. Przy metodzie połowienia relacja ta jest nieznana – może być albo większe, albo mniejsze.

Kolejne pole do porównań zachowania się metod uzyskalibyśmy dodając założenie, że miejsc zerowych jest więcej, że są ekstrema, nieciągłości itd.

Złożoność algorytmu numerycznego?

Algorytmy rozwiązywania układu równań metodą eliminacji Gaussa
(czyli tzw. dodawanie stronami)

$$
\begin{array}{ccccccccc}
a_{11}x_1 & + & a_{12}x_2 & + & \ldots & + & a_{1n}x_n & = & b_1 \\
a_{21}x_1 & + & a_{22}x_2 & + & \ldots & + & a_{2n}x_n & = & b_2 \\
\ldots & & \ldots & & \ldots & & \ldots & & \ldots \\
a_{n1}x_1 & + & a_{n2}x_2 & + & \ldots & + & a_{nn}x_n & = & b_n
\end{array}
$$

Tu nie ma problemu z dokładnym wyznaczeniem liczby dodawań/odejmowań, mnożeń i dzieleń potrzebnych do otrzymania rozwiązania dla n niewiadomych. Zatem złożoność obliczeniowa tego algorytmu to $\Theta(n^3)$. Nie ma też problemu przy wyznaczeniu rozmiaru potrzebnej pamięci. Złożoność pamięciowa wynosi $\Theta(n^2)$.

Problem obliczeniowy

Formalnie: problem obliczeniowy reprezentowany jest przez zbiór egzemplarzy danych wejściowych, zbiór egzemplarzy rozwiązań oraz zbiór relacji przekształcających pierwszy z tych zbiórów w drugi

$$E = (e_1, e_2, e_3, ...) \quad \xrightarrow{Q(E,S)} \quad S = (s_1, s_2, s_3, ...)$$

Zbiór relacji to zbiór procedur działających z użyciem jakichś algorytmów. Problem obliczeniowy nazywany jest więc także **problemem algorytmicznym**.

Rozwiązanie problemu obliczeniowego

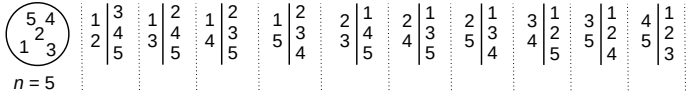
Uwaga: znaczenie słowa „rozwiązanie" jest tu nieco inne niż potoczne znaczenie tego słowa!

Rozwiązanie problemu obliczeniowego to każdy taki zestaw danych wyjściowych (ich wartości, konfiguracji, wzajemnych relacji itp.), który nie jest sprzeczny z założeniami i wymaganiami sformułowanymi w specyfikacji tego problemu

Problemy mogą mieć wiele rozwiązań albo nie mieć żadnego!
Różne wersje tego samego problemu (np. różnice w sformułowaniu) mogą mieć inny zbiór rozwiązań.

Liczność znbioru rozwiązań – przykłady

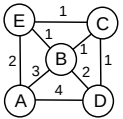
- Problem sortowania zbioru unikatowych liczb naturalnych ma jedno rozwiązanie.
  2,5,6,7,3,1 => 1,2,3,5,6,7
- Problem sortowania zbioru liczb naturalnych, w którym dwie liczby mogą się powtórzyć, ma dwa rozwiązania.
  $2,5,6_a,7,6_b,3,1$ => $1,2,3,5,6_a,6_b,7$ $1,2,3,5,6_b,6_a,7$
- Problem podziału zbioru n-elementowego na dwa podzbiory o licznościach k i $n-k$.
  Liczność zbioru rozwiązań jest równa $\binom{n}{k} = \frac{n!}{k!(n-k)!}$

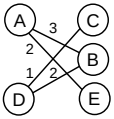


Czy wszystkie rozwiązania są jednakowo „dobre”?

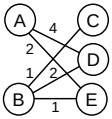
Na tak postawione pytanie można odpowiedzieć tylko wtedy, gdy w sformułowaniu problemu zdefiniowane jest jakieś kryterium, według którego rozwiązania mogą być porównane.

Przykład. Problem podziału zbioru: dane są liczba k oraz n-elementowy zbiór, w którym relacje między każdą parą elementów mają jakąś wagę. Jakością podziału na dwa podzbiory o licznościach k i $n - k$ jest suma wag relacji między elementami należącymi do różnych podzbiorów.

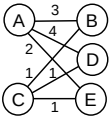




Jakość = 8



Jakość = 10



Jakość = 12

Funkcja celu

Funkcja celu

Funkcja zależności jakości rozwiązania problemu od uzyskanych danych wyjściowych nosi nazwę funkcji celu.

Zwykle naszym celem jest uzyskanie jak najlepszego rozwiązania.

Funkcja celu – definicja matematyczna

Funkcja celu to taka funkcja, której ekstremum się poszukuje.

Złożoność problemu obliczeniowego

Złożoność obliczeniowa **najlepszego** (najbardziej sprawnego obliczeniowo) algorytmu rozwiązującego ten problem dla **najgorszego** przypadku ze zbioru danych wejściowych

- Istnieją problemy, dla których w sposób teoretyczny można wyznaczyć ich złożoność
  - ale możemy nie znać algorytmu mającego taką taką złożoność.
- Istnieją problemy, dla których nie potrafimy teoretycznie wyznaczyć ich złożoności
  - wtedy za „roboczą" złożoność problemu uznajemy złożoność najsprawniejszego znanego algorytmu.

Uwaga: Dla każdego istniejącego problemu znamy natomiast **najmniej sprawny** algorytm – tzw. algorytm siłowy („brute force") polegający na przejrzeniu **wszystkich** możliwych rozwiązań i wybraniu najlepszego.

Klasy złożoności problemów

Klasy złożoności problemów:

k	stała	
$\lg n$	logarytmiczna	Grupa problemów wielomianowych
n	liniowa	
$n \lg n$	liniowo-logarytmiczna	
n^k	wielomianowa	
$n^{\lg n}$	podwykładnicza	Grupa problemów ponadwielomianowych (wykładniczych)
k^n	wykładnicza	
$n!$	silniowa	
n^n	nadwykładnicza	
∞	nierozwiązywalne	

Konsekwencje przynależności problemu do jakiejś klasy:

- Problemy wielomianowe: najlepsze rozwiązanie da się znaleźć w rozsądnym czasie.
- Problemy ponadwielomianowe: znalezienie najlepszego rozwiązania jest **praktycznie niemożliwe**!
  Trzeba stosować algorytmy **heurystyczne**, czyli takie, które nie dają gwarancji znalezienia najlepszego rozwiązania, umożliwiają jednak znalezienie rozwiązania dość dobrego w rozsądnym czasie.

Rodzaje problemów obliczeniowych

Problemy dzieli się ze względu na rodzaj otrzymanego rozwiązania (na rodzaj pytania, na jakie to rozwiązanie odpowiada).
Rodzaje podstawowe (ogólne, teoretyczne):

- Problemy optymalizacyjne.
- Problemy decyzyjne.

Rodzaje nieco bardziej „szczegółowe”:

- Problemy porządkowania (porządkowanie danych według jakiegoś kryterium).
- Problemy segregowania (rozdział danych według jakiegoś kryterium).
- Problemy przeszukiwania (znalezienie danych spełniających jakieś kryterium) .
- Problemy funkcyjne (znalezienie wartości jakiejś zależności – niekoniecznie funkcji).
- itp.

Problem optymalizacyjny

To taki problem obliczeniowy, dla którego zdefiniowana jest funkcja celu, a rozwiązanie polega na znalezieniu ekstremum tej funkcji

W problemie optymalizacyjnym należy odpowiedzieć na pytanie **Jaki jest?**

Przykład. Dane są: liczba k oraz n-elementowy zbiór, w którym relacje między każdą parą elementów mają jakąś wagę. Dla jakiego podziału na dwa podzbiory o licznościach k i $n-k$ suma wag relacji między elementami należącymi do różnych podzbiorów jest najmniejsza?

Problem decyzyjny

To taki problem algorytmiczny, w którym rozwiązanie może przyjąć wartości 0 lub 1 (nie lub tak)

W problemie optymalizacyjnym należy odpowiedzieć na pytanie **Czy istnieje?**

Uwaga: Każdy problem można sformułować tak, by był to problem decyzyjny. Odwrotna relacja nie musi być prawdziwa.

Przykład. Dane są: liczba k, pewna wartość w oraz n-elementowy zbiór, w którym relacje między każdą parą elementów mają jakąś wagę. Czy istnieje taki podział na dwa podzbiory o licznościach k i $n-k$, dla którego suma wag relacji między elementami należącymi do różnych podzbiorów jest mniejsza od w?

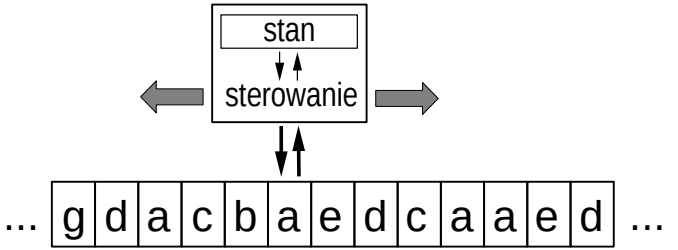
Problem decyzyjny – analiza matematyczna

Problemy decyzyjne są niejako najbardziej ogólne. Zatem rozważania teoretyczne dotyczące złożoności problemów obliczeniowych prowadzi się przede wszystkim właśnie dla problemów zdefiniowanych właśnie w sposób decyzyjny. Wyniki tych rozważań można w większości przypadków przenosić na ocenę złożoności problemów zdefiniowanych niedecyzyjnie.

Podstawy matematyczne analizy algorytmów i problemów obliczeniowych:

Maszyna Turinga – abstrakcyjny model matematyczny maszyny służącej do wykonywania algorytmów

Maszyna Turinga - definicja podstawowa



Na podstawie bieżącego stanu i symbolu znajdującego się pod głowicą układ sterujący podejmuje decyzje o:

- zmianie zawartości komórki taśmy znajdującej się pod głowicą,
- przesunięciu głowicy w lewo lub w prawo,
- zmianie stanu mechanizmu sterującego.

Maszyna Turinga

- deterministyczna maszyna Turinga – algorytmy deterministyczne: w każdym kroku podejmowane są jednoznacznie określone decyzje zdeterminowane przez konkretne kryteria, ale nie ma pewności czy decyzja ta da w ostatecznym rozrachunku dobry rezultat,
- niedeterministyczna maszyna Turinga – algorytmy niedeterministyczne: w każdym kroku podejmowana jest decyzja jak dalej postępować, ale decyzja ta jest zawsze słuszna.

Model maszyny niedeterministycznej jest w pewnym sensie wielowymiarowy – do tego stopnia, że głowica może się przesuwać niejako w wielu kierunkach jednocześnie.

Maszyna Turinga – implementacja

- Współczesny **komputer** jest implementacją deterministycznej maszyny Turinga.

Natomiast:

- Niedeterministyczna maszyna Turinga fizycznie **nie istnieje**. Taki komputer musiałby bowiem umieć bezbłędnie **zgadywać** poprawną odpowiedź na każde zadane pytanie.

Klasy problemów decyzyjnych

1. Klasa P – problemy, które można rozwiązać stosując deterministyczny algorytm o złożoności co najwyżej wielomianowej (w skrócie mówi się: algorytm wielomianowy). Nazwa klasy pochodzi się od słowa *Polynomial*.
2. Klasa NP – problemy, które można rozwiązać stosując niedeterministyczny algorytm o złożoności co najwyżej wielomianowej. Nazwa klasy pochodzi się od słów *Nondeterministic Polynomial*.

P zawiera się w NP, ale istnieje ogromna liczba problemów NP, dla których nie znamy deterministycznych algorytmów wielomianowych (mówi się, że te problemy są trudniejsze od problemów P).

Jeden z problemów milenijnych: czy P = NP?

Gdyby tak było – trudność wszystkich problemów byłyby taka sama!

Ponieważ w praktyce nie dysponujemy komputerami niedeterministycznymi, by na komputerach deterministycznych móc znaleźć rozwiązanie korzystamy z definicji praktycznej:

Klasa NP – definicja praktyczna

Załóżmy, że dany jest problem i jakieś sugerowane rozwiązanie tego problemu (pomińmy na razie skąd to rozwiązanie się wzięło – mogliśmy je zgadnąć albo wylosować z puli wszystkich rozwiązań). Problem należy do klasy NP, jeżeli poprawność sugerowanego rozwiązania da się zweryfikować algorytmem wielomianowym.

Inaczej mówiąc: problem wielomianowo weryfikowalny jest problemem klasy NP.

Równoważność problemów

Dwa problemy decyzyjne są równoważne, jeżeli istnieje procedura, która definicję jednego problemu jednoznacznie przekształci w definicję drugiego problemu. Procedura taka zwana jest redukcją problemu

Konsekwencją redukowalności problemów jest również istnienie procedury przekształcającej rozwiązanie jednego problemu w rozwiązanie drugiego. Inaczej mówiąc: jeżeli dwa problemy są równoważne, to znając rozwiązanie pierwszego problemu, można na tej podstawie wydedukować rozwiązanie drugiego.

Problem jest NP-zupełny (NP-Complete, klasa NPC) jeżeli należy do klasy NP oraz istnieje wielomianowy algorytm redukujący go do dowolnego innego problemu NP-zupełnego

Konsekwencja: jeżeli znamy rozwiązanie jakiegoś problemu NP-zupełnego, to jakimś algorytmem wielomianowym możemy wyznaczyć rozwiązanie każdego innego problemu NP-zupełnego (kolokwialnie: jeżeli znamy rozwiązanie pierwszego problemu, to znamy również rozwiązanie problemu drugiego).

Uważa się, że problemy NP-zupełne są jednymi z najtrudniejszych spośród problemów należących do klasy NP – obecnie nie znamy żadnych algorytmów wielomianowych znajdujących rozwiązanie jakiegokolwiek problemu NP-zupełnego.

NP-zupełność przykłady

Znanych jest nieco ponad 1000 problemów NP-zupełnych.
Między innymi:

- problem znajdowania cyklu Hamiltona w grafie,
- problem znajdowania pokrycia wierzchołkowego grafu,
- problem komiwojażera,
- problem plecakowy (problem złodzieja),
- gra Saper,
- gra Tetris,
- gra Sudoku,
- problem rezerwacji koi (zwany też problemem harmonogramowania cumowania).
- problem podziału zbioru z wagami relacji między elementami („nasz" przykład wykladowy).

Uwaga: powyższe przykładowe problemy muszą być sformułowane w sposób decyzyjny (niektóre z nich sformułowane niedecyzyjnie mogą nie należeć do klasy NP-zupełnych).

NP-trudność – definicja praktyczna

Problem jest NP-trudny (NP-Hard, klasa NPH) jeżeli nie istnieje żaden ograniczony wielomianowo algorytm weryfikacji rozwiązania tego problemu, ale daje się w niego przekształcić dowolny problem z klasy NP[1]

Zauważmy, że dla problemu NP-trudnego problem weryfikacji rozwiązania może być problemem NP-zupełnym. Zatem problem NP-trudny może być jeszcze bardziej trudny, niż każdy z problemów klasy NP.

[1] Prawidłowo ta definicja powinna brzmieć: "...może nie istnieć żaden wielomianowy algorytm...". Z formalnej definicji klasy problemów NP-Trudnych (której tu nie przytaczamy, bo wymaga użycia bardzo rozwiniętego aparatu matematycznego) wynika, że wszystkie problemy NP-zupełne (i tylko one z kasy NP) też należą do klasy NP-trudnych (zatem problem NP-trudny może, choć nie musi, należeć do klasy NP). Do celów praktycznych jednak dokonuje się skrótu myślowego nazywając problemami NP-trudnymi tylko „te NP-trudne, które nie są jednocześnie NP".

NP-trudność przykłady

Wiele problemów, których **wersja decyzyjna** należy do klasy NP-zupełnych, w **wersji optymalizacyjnej** jest NP-trudna.

Najbardziej „popularne" to:

- problem komiwojażera,
- problem plecakowy (problem złodzieja).

I oczywiście:

- problem podziału zbioru z wagami relacji między elementami („nasz" przykład wykładowy).

Ale także:

- problem rozprowadzania połączeń na elektronicznej płytce drukowanej czy w układzie scalonym,
- problem rozdziału elementów układu elektronicznego między płyty montażowe urządzenia (skąd my go znamy?),
- problem znajdowania tzw. ścieżek krytycznych w układach elektronicznych i w telekomunikacji,
- itp.

Wniosek generalny z powyższych rozważań. Uważa się, że dla problemów NP-zupełnych i NP-trudnych nie ma algorytmów ograniczonych wielomianowo. W związku z tym, jeżeli udowodnimy, że interesujący nas problem należy do jednej z tych klas, **nie ma sensu poszukiwać idealnego rozwiązania – mamy szansę wyłącznie na znalezienie rozwiązania przybliżonego** (nawet jeżeli przypadkiem znalezione zostanie rozwiązanie idealne, to udowodnienie tego nie jest praktycznie możliwe).

Niestety znakomita większość problemów występujących w elektronice i telekomunikacji to problemy NP-zupełne i NP-trudne