Złożoność obliczeniowa algorytmów

ALGORYTM – przepis na rozwiązanie problemu, wyrażony w skończonej liczbie poleceń

Wykonawca algorytmu musi rozumieć polecenia.

dawniej – wykonawcą był na ogół człowiek (matematyka, inżynieria, rzemiosło)

dziś – w centrum zainteresowania są algorytmy komputerowe; wykonawcą jest komputer, który rozumie polecenia wydawane w specjalnym języku programowania

Algorytm komputerowy do program.

Informatyka zajmuje się zarówno tworzeniem, jak i badaniem algorytmów komputerowych.

Interesują ją takie zagadnienia jak:

• istnienie algorytmów dla pewnych problemów (gdy algorytm nie istnieje, to problem nazywamy nierozstrzygalnym)

• poprawność algorytmów

- czy algorytm robi to co powinien?
- czy algorytm się zatrzyma?

• efektywność algorytmów

- czasowa
- pamięciowa

- Czy istnieje algorytm lepszy od danego?

- Czy istnieje algorytm najlepszy?

Problematyką tą zajmuje się dział informatyki zwany analizą algorytmów

Przykład problemu nierozstrzygalnego

Dziesiąty problem HILBERTA

Równaniem diofantycznym nazywa się każde równanie o współczynnikach całkowitych, którego rozwiązań szukamy w zbiorze liczb całkowitych lub naturalnych, np.:

$$2x + 3y + 5z - 11 = 0,$$

$$x^2 + x - 1560 = 0$$

Przykład problemu nierozstrzygalnego

Czy dla równania diofantycznego z dowolną liczbą niewiadomych istnieje algorytm pozwalający w skończonej liczbie kroków stwierdzić, czy istnieją liczby całkowite spełniające to równanie?

Odpowiedź: NIE

Czas wykonania programu

Podczas rozwiązywania problemu jesteśmy często zmuszeni do wyboru jednego z wielu algorytmów. Czym się powinniśmy w takim wypadku kierować. Są dwa sprzeczne cele:

1.Chcielibyśmy mieć algorytm łatwy do zrozumienia, zakodowania w języku programowania i śledzenia.

2.Chcielibyśmy mieć algorytm wykorzystujący efektywnie zasoby komputera, a przede wszystkim działający tak szybko, jak tylko to jest możliwe.

Złożoność obliczeniowa algorytmów

W przypadku gdy tworzony algorytm (program) będzie używany jednokrotnie lub najwyżej parę razy, wystarczy spełnienie warunku 1, bo ewentualne koszty optymalizacji przekroczą zyski wynikające z szybkości działania programu.

Jeśli natomiast projektujemy algorytm wielokrotnego użytku, szybkość staje się istotna i warto ponieść koszt optymalizacji (szczególnie, jeśli przyjdzie uruchamiać program na dużych zbiorach danych). Ale nawet wtedy warto stworzyć prosty algorytm i w odniesieniu do niego ocenić korzyści, jakie przyniesie optymalizacja.

Czas wykonania programu komputerowego

Czas wykonania programu zależy od następujących czynników:

1. Wielkości i rodzaju wejścia (input) dla programu.
2. Jakości kodu wygenerowanego przez kompilator, którego użyliśmy do stworzenia programu wynikowego.
3. Rodzaju i szybkości instrukcji maszynowych użytych do wykonania programu.
4. Złożoności czasowej algorytmu, na podstawie którego powstał program.

Dlatego też nie możemy wyrażać czasu wykonania programu w standardowych jednostkach, takich jak sekundy. Możemy jedynie mówić, że „czas wykonania tego algorytmu jest proporcjonalny do n^2”. Stałej proporcjonalności nie będziemy określać, gdyż zależy ona ściśle od kompilatora, maszyny i innych czynników.

Notacja Ơ

Aby mówić o szybkości wzrostu funkcji używamy tzw. „duże-o” notacji. Na przykład, gdy mówimy, że czas wykonania T(n) jakiegoś programu jest Ơ(n^2), to znaczy to, że istnieją dodatnie stałe c i n_0, takie że dla $n \geq n_0$: $T(n) \leq cn^2$

PRZYKŁAD

Załóżmy, że $T(0) = 1$, $T(1) = 4$ i w ogólnym przypadku $T(n) = (n+1)^2$. Widzimy, że $T(n)$ jest $O(n^2)$, bo przyjmując $n_0 = 1$ i $c = 4$ mamy dla $n \geq 1$, $(n+1)^2 \leq 4n^2$. Oczywiście nie możemy przyjąć $n_0 = 0$, bo $T(0) = 1 > c0^2 = 0$ dla każdej stałej c.

Algorytm sortowania przez proste wstawianie

```
a[0], a[1], a[2] ... a[i-1], a[i], a[n]

for i := 1 to n do
  begin
    j := i – 1; x := a[i]; a[0] := x;
    while x < a[j] do
      begin
        a[j+1] :=: a[j];
        j := j – 1;
      end;
    a[j+1] := x
  end;
```

```
    ↓
5 5 2 3 7 1 8
      ↓
2 5 2 3 7 1 8
        ↓
3 2 5 3 7 1 8
          ↓
1 2 3 5 7 1 8
              ↓
8 1 2 3 5 7 8
```

Postać danych dająca minimum operacji

1, 2, 3, 4, 5, 6 $T(n) = n$

Postać danych dająca maksimum operacji

6, 5, 4, 3, 2, 1 $T(n) = n^2/2 + n/2$

Złożoność obliczeniowa algorytmu (pesymistyczna)

jest funkcją:

$$T_A : N \to R$$

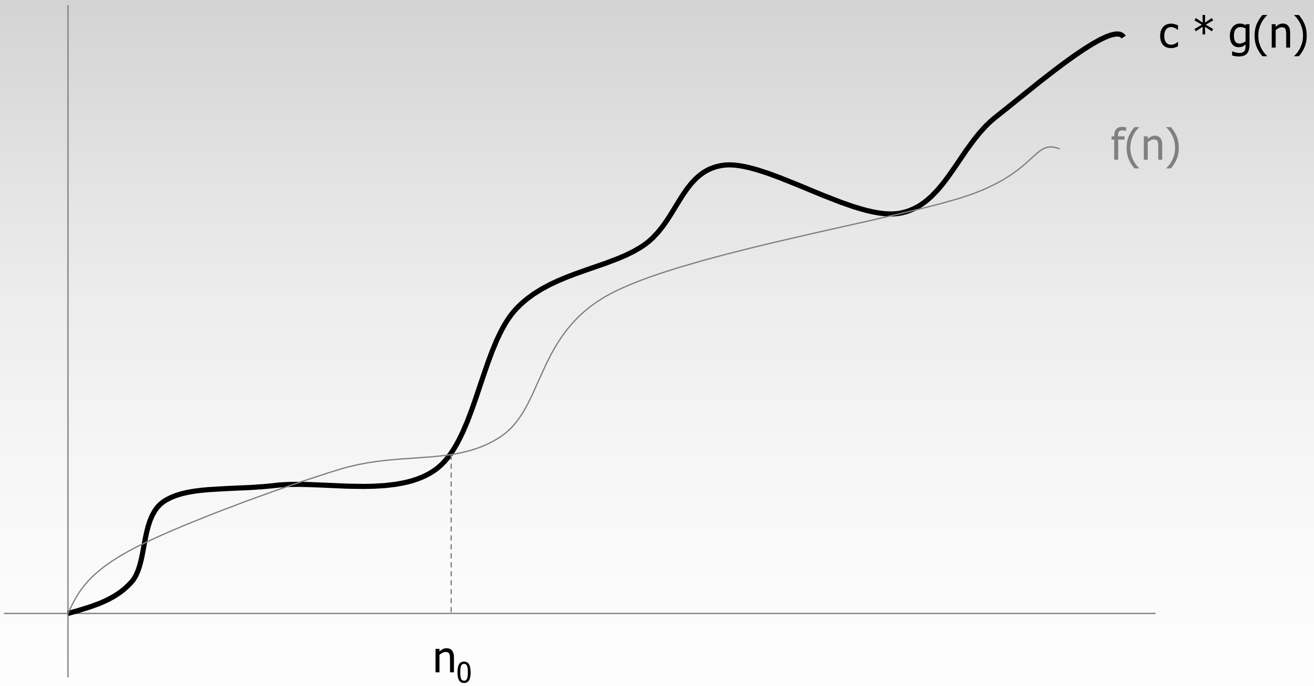
$$T_A(n) = \max \{\text{czasy wykonania algorytmu A dla wejść rozmiaru n}\}$$

Notacja Ơ

Mówimy, że f(n) jest Ơ(g(n))

$\Leftrightarrow \exists\ n_0 \in N, c \in R:$

$$n \geq n_0 \Rightarrow f(n) \leq c * g(n)$$



Własności

1. $T_1(n)$ jest $O(f(n))$
   $T_2(n)$ jest $O(g(n))$
   $\Rightarrow T_1(n) + T_2(n)$ jest $O(\max(f(n), g(n))$

2. – ” – $\Rightarrow T_1(n) * T_2(n)$ jest $O((f(n) * g(n))$

Wnioski: $g(n) \leq f(n)$ dla $n \geq n_0 \Rightarrow O(f(n) + g(n)) = O(f(n))$.

$O(c * f(n)) = O(f(n))$. Zatem np. $O(\frac{1}{2}n^2 + \frac{1}{2}n) = O(n^2)$.

Klasy złożoności obliczeniowej

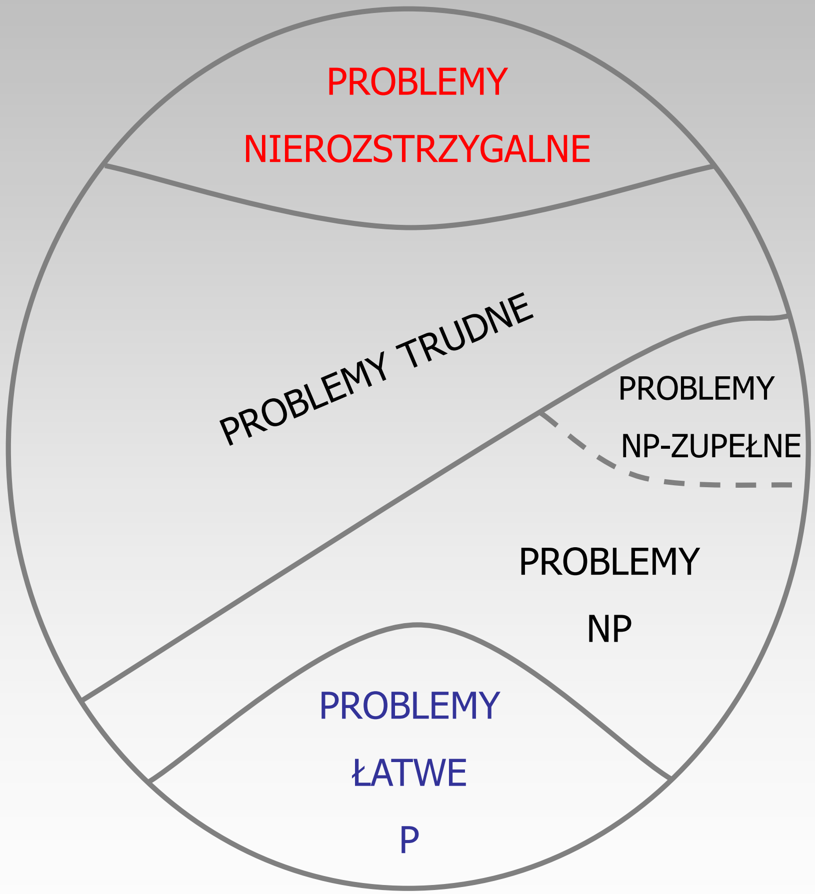
Algorytmy o złożoności czasowej wielomianowej (polynomial), to te których funkcja złożoności T(n) jest Ơ(p(n)), gdzie p(n) – wielomian, a n – rozmiar problemu (wejścia).

Jeśli algorytm nie ma złożoności wielomianowej, to mówimy, że jego złożoność jest wykładnicza (eksponencjalna).

Ơ(n), Ơ(nlogn), Ơ(n^2), Ơ(n^3) – złożoności wielomianowe

Ơ(2^n), Ơ(3^n), Ơ(n!) – złożoności wykładnicze

Klasyfikacja problemów



PROBLEMY ŁATWE (P)
(polynomial)

Istnieje algorytm wielomianowy

PROBLEMY NP.
(nondeterministic polynomial)

Wszystkie łatwe i takie, dla których nie udowodniono, że brak jest rozwiązania wielomianowego, ale go nie znaleziono.

Rozwiązania są typu:
1. zgadnąć rozwiązanie,
2. sprawdzić (wielomianowo) czy jest poprawne.

PROBLEMY NP-ZUPEŁNE

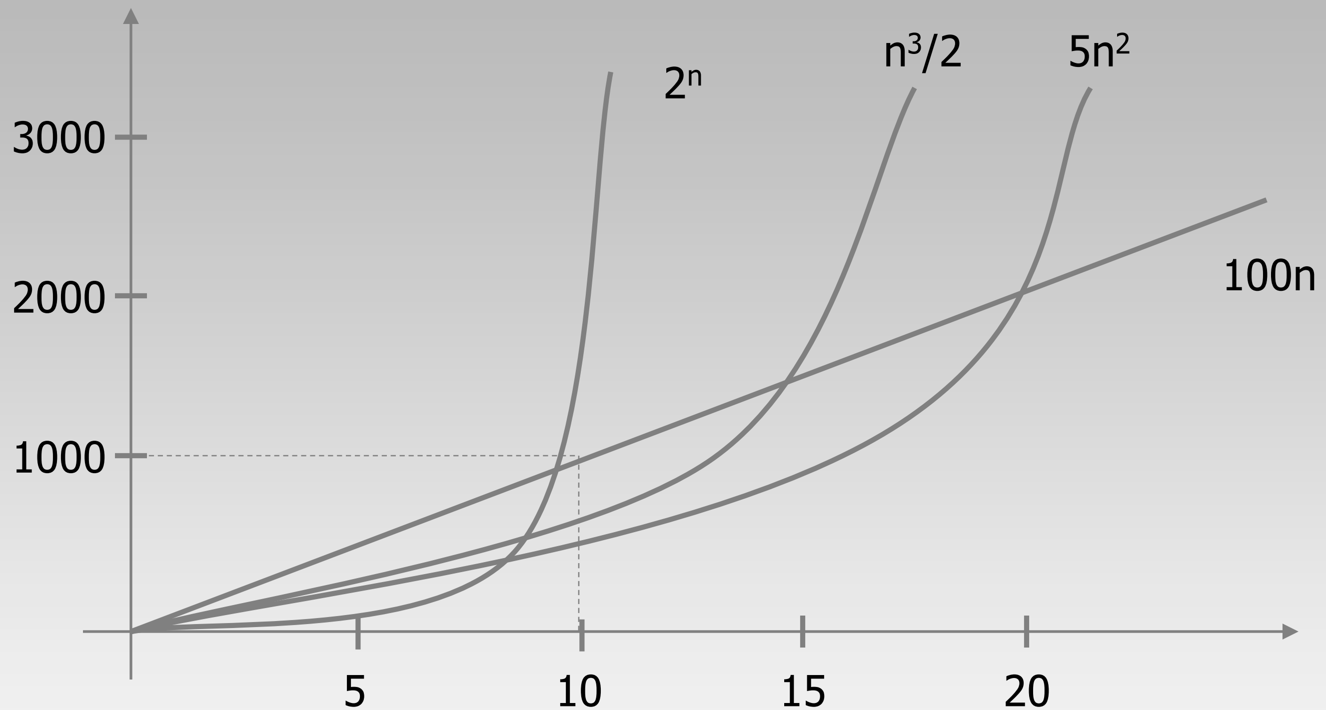
Te problemy NP, dla których znalezienie wielomianowego rozwiązania dla jednego z nich – daje rozwiązanie dla wszystkich.

PROBLEMY TRUDNE

Istnieje algorytm wykładniczy i nie istnieje wielomianowy.

PROBLEMY NIEROZSTRZYGALNE

Algorytm nie istnieje.



Zależność czasu wykonania od rozmiaru danych wejściowych dla czterech algorytmów

T(n)	max we dla 10^3 sek.	max we dla 10^4 sek.	Przyrost wykonywanych obliczeń
100n	10	100	10,0
$5n^2$	14	45	3,2
$n^3/2$	12	27	2,3
2^n	10	13	1,3

Wraz ze wzrostem mocy obliczeniowej ROŚNIE zapotrzebowanie na efektywne algorytmy

Porównanie czasów wykonania algorytmów o różnych złożonościach czasowych

złożoność \ rozmiar we	10	30	50	
n	$10 * 10^{-6}$ s.	$30 * 10^{-6}$ s.	$50 * 10^{-6}$ s.	wiek Ziemi
$n\log_2 n$	$33,2 * 10^{-6}$ s.	$147,2 * 10^{-6}$ s.	$282,5 * 10^{-6}$ s.	ok. $4,6 * 10^9$ lat
n^2	$0,1 * 10^{-3}$ s.	$0,9 * 10^{-3}$ s.	$2,5 * 10^{-3}$ s.	
n^3	$1 * 10^{-3}$ s.	$27 * 10^{-3}$ s.	$125 * 10^{-3}$ s.	
2^n	0,001 s.	17,9 min.	35,7 lat	
3^n	0,059 s.	6,53 roku	$2,3 * 10^8$ wieków	
10^n	2,8 godz.	$3,17 * 10^{14}$ wieków	$3,17 * 10^{34}$ wieków	

Należy podkreślić, że koszt czasowy najgorszego przypadku (złożoność czasowa pesymistyczna) nie jest jedynym i nie zawsze jest najważniejszym kryterium oceny algorytmu. Oto sytuacje, gdy tak nie jest:

1.Jeśli program jest używany jedynie kilka razy, wtedy koszt napisania go i testowania znacznie przewyższa koszt użytkowania. W tym przypadku należy wybrać algorytm najprostszy w implementacji.

2. Jeśli program ma być uruchamiany tylko dla wejść o małych rozmiarach, to szybkość wzrostu czasu wynikająca za złożoności może być mniej istotna od stałej określonej przez rodzaj komputera i kompilatora.

3. Jeśli skomplikowany i efektywny algorytm ma być użyty w programie pisanym przez kogoś innego niż autor algorytmu, to trzeba się liczyć z tym, że z powodu komplikacji będzie on całkowicie bezużyteczny. Lepiej użyć algorytmu mniej wyrafinowanego, ale napisanego zgodnie z rozpowszechnionymi i znanymi technikami.

4. Niektóre szybkie algorytmy zajmują tak dużo pamięci, że niezbędne jest użycie pamięci zewnętrznej (wolnej) – co może postawić pod znakiem zapytania ich efektywność.

5. W algorytmach numerycznych poprawność i stabilność są równie ważne jak szybkość.

Specyfika algorytmów równoległych

Jednostkową realizacją algorytmu jest proces

Poza algorytmami sekwencyjnymi przedmiotem badań są algorytmy równoległe (współbieżne)

Realizacją algorytmów równoległych są biegnące równocześnie procesy, które mogą na siebie współoddziaływać

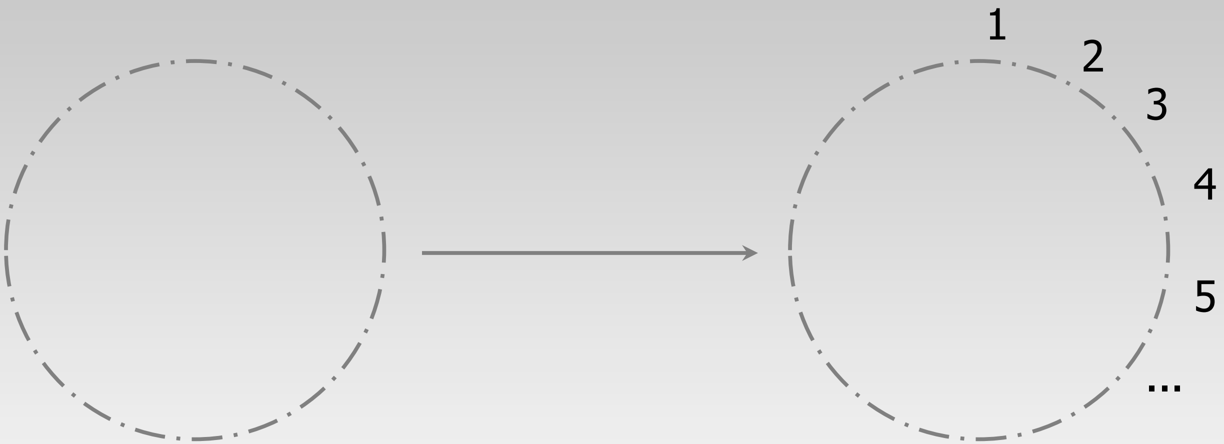
W zależności od sposobu sterowania tym współoddziaływaniem mogą to być procesy:

synchroniczne (wspólne, centralnie sterowane – np. taktowanie)

asynchroniczne (niezależne, „rozproszone” zachowanie procesów)

Problem rankingu

Danych jest n procesorów pracujących sekwencyjnie i tworzących pierścień (każdy procesor ma dwóch sąsiadów, n > 1.



Każdy procesor może przyjąć skończoną liczbę stanów. Praca procesora polega na zmianie stanów. Procesory mogą się ze sobą komunikować. Rezultatem komunikacji jest zmiana stanu.

Procesory pracują w pełni asynchronicznie (nie mają żadnego zegara, który odliczałby wspólny dla nich czas).

Każdy procesor pracuje tak długo, aż osiągnie stan końcowy lub zablokuje się czekając na komunikację, która nigdy nie nastąpi. W przeciwnym wypadku pracuje w nieskończoność.

Problem rankingu (dla pierścienia):

Znajdź taki stan początkowy i taki program (protokół) identyczny dla wszystkich procesorów, że wszystkie one startując z tego stanu i pracując wg tego protokołu w końcu się zatrzymają i ich stany będą różne między sobą.

Problem rankingu jest rozstrzygalny wtedy i tylko wtedy gdy liczba procesorów w pierścieniu jest liczbą pierwszą.

Odmienność zagadnień analizy algorytmów równoległych

- obliczanie lokalne $\rightarrow$ rezultat globalny

- wiedza lokalna a wiedza globalna

(co innego wiedzieć, że się ma numer i się zakończyło swój udział w obliczeniu, a co innego wiedzieć, że obliczenie się zakończyło)

