Analýza efektivity algoritmu.

Analýza efektivity algoritmu. Asymptotická složitost. Třídy složitosti P a NP.

Tomáš Bayer | bayertom@natur.cuni.cz

Katedra aplikované geoinformatiky a kartografie. Přírodovědecká fakulta UK.

Obsah přednášky

1. Efektivita algoritmu

Efektivita algoritmu je důležitou *výkonostní charakteristikou*.
Výkonnostní charakteristiky algoritmů nelze ignorovat.
Časové a materiální úspory.

Efektivní algoritmus řeší problém s minimálními nároky na HW.
Strategie ”Time Is Money”.
Optimální využití existujících prostředků”.
Pozor na zdánlivě nepodstatné detaily.

Nejrychlejší vs. optimální řešení:
Nejjednodušší zpravidla nejpomalejší, ale implementačně jednoduché.
Nejrychlejší velmi náročné na implementaci (časově kritické aplikace).

V praxi používáno optimální, kompromis (běžné aplikace).

Hodnocení efektivity empiricky nebo matermatickou analýzou.

2. Analýza efektivity algoritmu

Efektivita algoritmu funkcí velikosti dat a jejich typu.
Stanovena na základě jeho **analýzy:**

- *Empirická*
  Srovnáním běhu > algoritmů, 3 množiny:
  1) Random: ověření funkcionality.
  2) Worst: schopnost zpracovat libovolná data.
  3) Best: nejlepší případ.
- *Exaktní*
  Matematická analýza řad, hledání asymptotických funkcí.

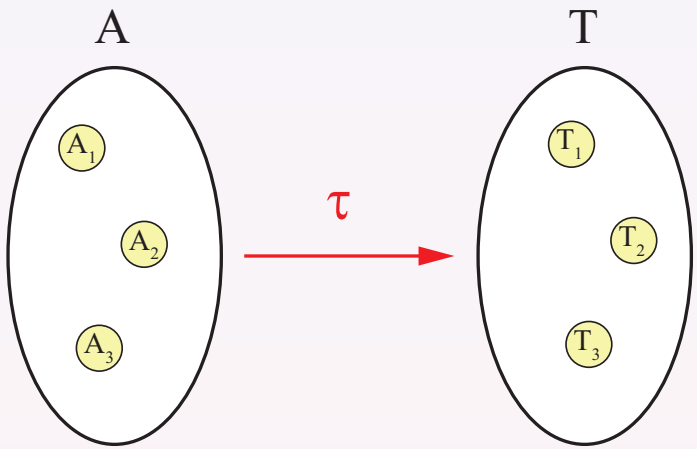
Cíle analýzy algoritmů:

- Porovnání > algoritmů řešící problém: výběr optimálního.
- Odhad výkonnosti algoritmu: lze ho použít pro problém & data?
- Nastavení parametrů algoritmu: co nejefektivnější běh.

Chybou soustředění se pouze na výkonnostní charakteristiky.
Implementace a odladění rychlého algoritmu pro libovolný vstup složité.
Lépe pomalejší, ale univerzálnější algoritmus.

3. Složitost algoritmu

Problém řešitelný více algoritmy.
Mohou se výrazně lišit dobou běhu (i v řádech).



Měřitelné charakteristiky: doba běhu,výpočtu, počet instrukcí, množství paměti...

- **Časová složitost algoritmu (Time Complexity)**
  Doba zpracování vstupních dat D algoritmem A v čase T

$$T = \tau(A(D)).$$

- **Paměťová složitost algoritmu (Space Complexity)**
  Množství paměti M pro zpracování vstupních dat D algoritmem A

$$M = \mu(A(D)).$$

Vykonání instrukce trvá určitou dobu.
Nejrychlejší je přiřazení, nejpomalejší násobení/dělení.
Vyhýbat se mocninám a odmocninám, velmi pomalý výpočet.

4. Posuzování složitosti algoritmu

Složitost funkcí velikosti vstupních dat a jejich hodnot

$$A = (D, n, ...), \qquad n = |D| .$$

A) Velikost vstupu n

Složitost **funkcí velikosti vstupu** n**.**
Exaktním tvaru složitý (např. $4n^3 - 9n^2 + 20n + 27$).
Používán asymptotický (limitní) odhad (např. $O(n^3)$).

B) Charakteristika vstupních dat

Pro vstup n se složitost závisí na hodnotách vstupu.
Např. náhodná setříděná nebo reverzně setříděná data.

Složitost lze posuzovat **empiricky**:

- dle nejhoršího možného případu (Worst Case).
- dle průměrné doby běhu (Average Case).
- dle nejlepšího možného případu (Best Case).

- Pouze relativní porovnání různých přístupů.
- Závislost na datech i HW.

5. Doba běhu nezávislá na datech

Porovnání dvou posloupností:
$a = \{a_1, ..., a_n\}, b = \{b_1, ..., b_n\}$.

Procházení prvek po prvku:

```
equal = true;
for i in range(n):
    for j in range(n):
        if a[i]!= b[j]:
            equal = false;
return equal;
```

Počet iterací: n^2.
Elementární operace: t, pak

$$T = t^2.$$

6. Worst Case

Nejhorší možný případ.

Maximum z dob běhu algoritmu pro všechny vstupy velikosti n

$$T_{WORST} = \max(T_1(n), T_2(n), ..., T_n(n)).$$

Nevhodná konfigurace vstupních dat).

Až o několik řádů vyšší než Average Case.

Algoritmus se bude většinou chovat “lépe” než pesimistický odhad.

V praxi k takové situaci nemusí dojít (nebo jen ve velmi řídkých případech).

Vlastnosti algoritmu mohou být tímto odhadem zkresleny.

7. Best Case

Nejlepší možný případ.

Minimum z dob běhu algoritmu pro všechny vstupy velikosti n.

$$T_{BEST} = \min(T_1(n), T_2(n), ..., T_n(n))$$

Ideální konfiguraci vstupních dat.

Až o několik řádů lepší než Average Case.

Algoritmus se bude většinou chovat “hůře” než optimistický odhad.

V praxi k takové situaci nemusí dojít (nebo jen ve velmi řídkých případech).

Vlastnosti algoritmu mohou být tímto odhadem zkresleny.

8. Average Case

Průměrný případ.
Průměrná doba běhu na (běžných) datech velikosti n.
Může být o několik řádů lepší/horší než Worst/Best Case .
Problematické určení, nejčastěji medián.

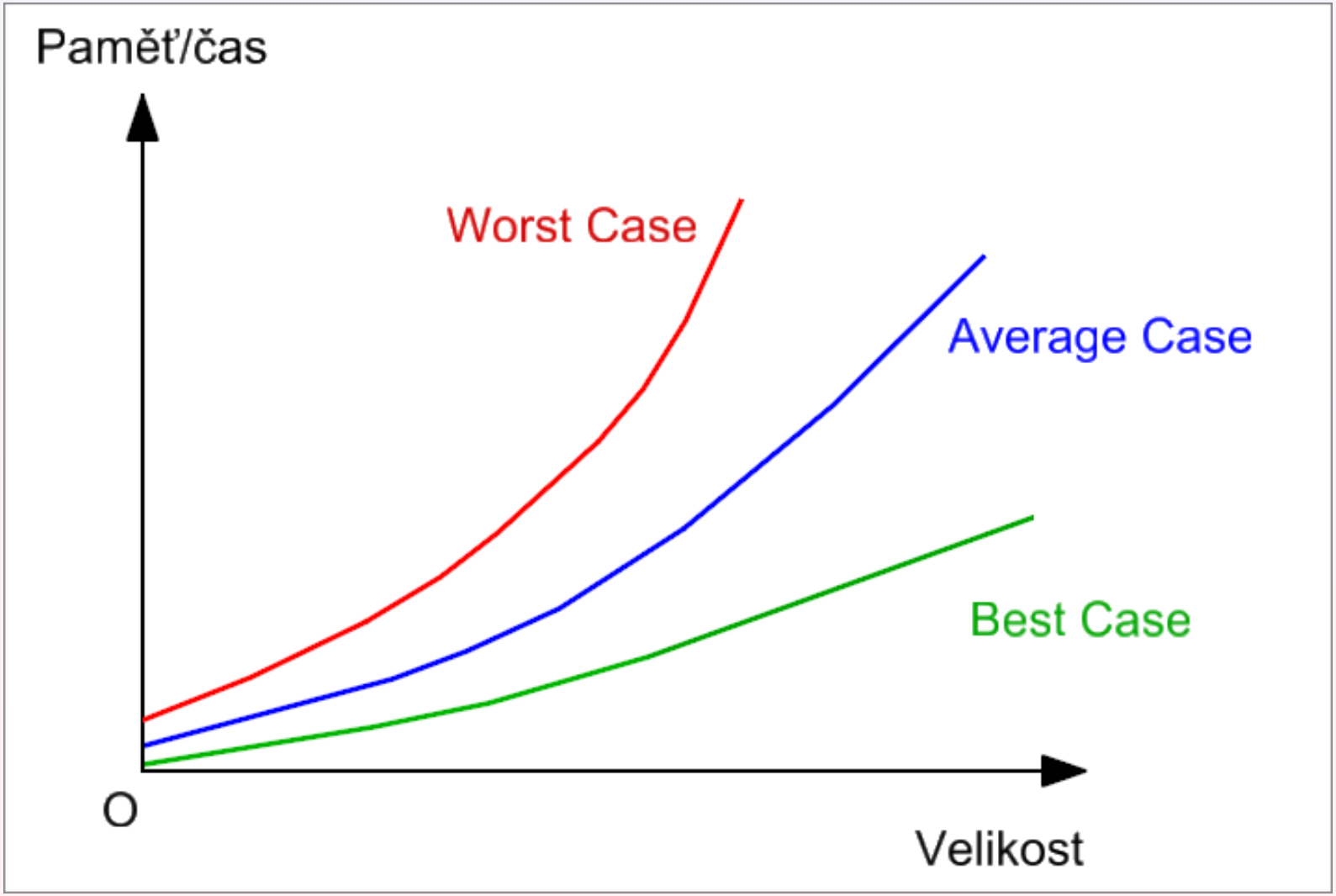
$$T_{AVERAGE} = E(T_1(n), T_2(n), ..., T_n(n)).$$

Běžná konfigurace vstupních dat.
Nemusí reprezentovat skutečná data, pouhý matematický konstrukt.

U dobře navržených algoritmů rozdíl mezi Best Case a Worst Case malý.
Optimalizace algoritmu: snížení rozpětí mezi Best Case a Worst Case.
Pokud Best Case = Worst Case, nelze dále optimalizovat.

Některé algoritmy nepoužitelné pro vysoké T_{WORST}.
QuickSort $T_{AVER} = n \log n$, $T_{WORST} = n^2$!!!.
Nahrazován MergeSortem: $T_{AVER} = T_{WORST} = n \log n$.

9. Ukázka



10. RAM model

Abstrakce od závislosti na HW: různá výkonnost.

Model **abstraktního počítače**, na kterém algoritmy implementovány.
Parametry jsou odvozeny od skutečného počítače.
Příklady: Turingův stroj nebo tzv. RAM (Random Acces Machine).

Vlastnosti:

- Konstantní časová složitost instrukcí.
- Sekvenční zpracování instrukcí.
- Neomezený HW (paměť).
- Konečný počet stavů.

Nezávislost výsledků na HW.

11. Asymptotická složitost

Empirické srovnání algoritmů nepostačuje, relativní.
Algebraické (přesné) vyjádření složitosti matematicky náročné.
Umíme určit jen u jednoduchých problémů.

Nahrazení algebraické složitosti asymptotickým (limitním) odhadem.

Asymptotická složitost pro $n \to \infty$ odpovídá algebraické složitosti.
Zajímá nás chování algoritmu pro velká n.
Popisuje řád růstu funkce, zjednodušení.

Zásady:
1) Zanedbání konstant (ignorujeme aditivní a multiplikativní konstanty).
2) Zanedbání funkcí s nízkým růstem řádu(“rychlé” části algoritmu).

Hodnoty $f_1(n) = 0.1n^2$ pro velká n podobné $f_2(n) = 100n^2 + 90n$.

$$\lim_{n\to\infty} f_1(n) = \lim_{n\to\infty} f_2(n).$$

12. Zanedbání konstant

Předpoklad 1: Zanedbání multiplikativní konstanty c:
Nechť $f(n)$ je libovolná funkce a c libovolná konstanta, $c > 0$. Pak funkce $f(n)$ a $c \cdot f(n)$ jsou označovány jako (asymptoticky) stejně rychle rostoucí.

$$\lim_{n\to\infty} c \cdot f(n) = \lim_{n\to\infty} f(n)$$

Předpoklad 2: Zanedbání aditivní konstanty d:
Nechť $f(n)$ je libovolná funkce a d libovolná konstanta, $d > 0$. Pak funkce $f(n)$ a $f(n) + d$ jsou označovány jako (asymptoticky) stejně rychle rostoucí.

$$\lim_{n\to\infty} f(n) + d = \lim_{n\to\infty} f(n)$$

Důsledek:
Funkce $f(n)$ a $c \cdot f(n) + d$ jsou (asymptoticky) stejně rychle rostoucí.
Stejný řád růstu.

$$\lim_{n\to\infty} c \cdot f(n) + d = \lim_{n\to\infty} f(n).$$

Funkce $f_1(n) = 0.1n^2$ a $f_2(n) = 100n^2 + 90n$ patří do stejné třídy (kvadratické funkce).

13. Přehled asymptotických odhadů

5 asymptotických odhadů složitosti:

1. Asymptotický horní odhad složitosti ostrý $O(g(N))$.
2. Asymptotický dolní odhad složitosti ostrý $\Omega(g(N))$.
3. Asymptotický oboustranný odhad složitosti: $\Theta(g(N))$.
4. Asymptotický horní odhad časové neostrý $o(g(N))$.
5. Asymptotický dolní odhad časové neostrý $\omega(g(N))$.

14. Asymptotický horní odhad $O(g(n))$

Ilustruje nejhorší možný případ doby běhu algoritmu.

Tzv. O-notace (Big O-notation).

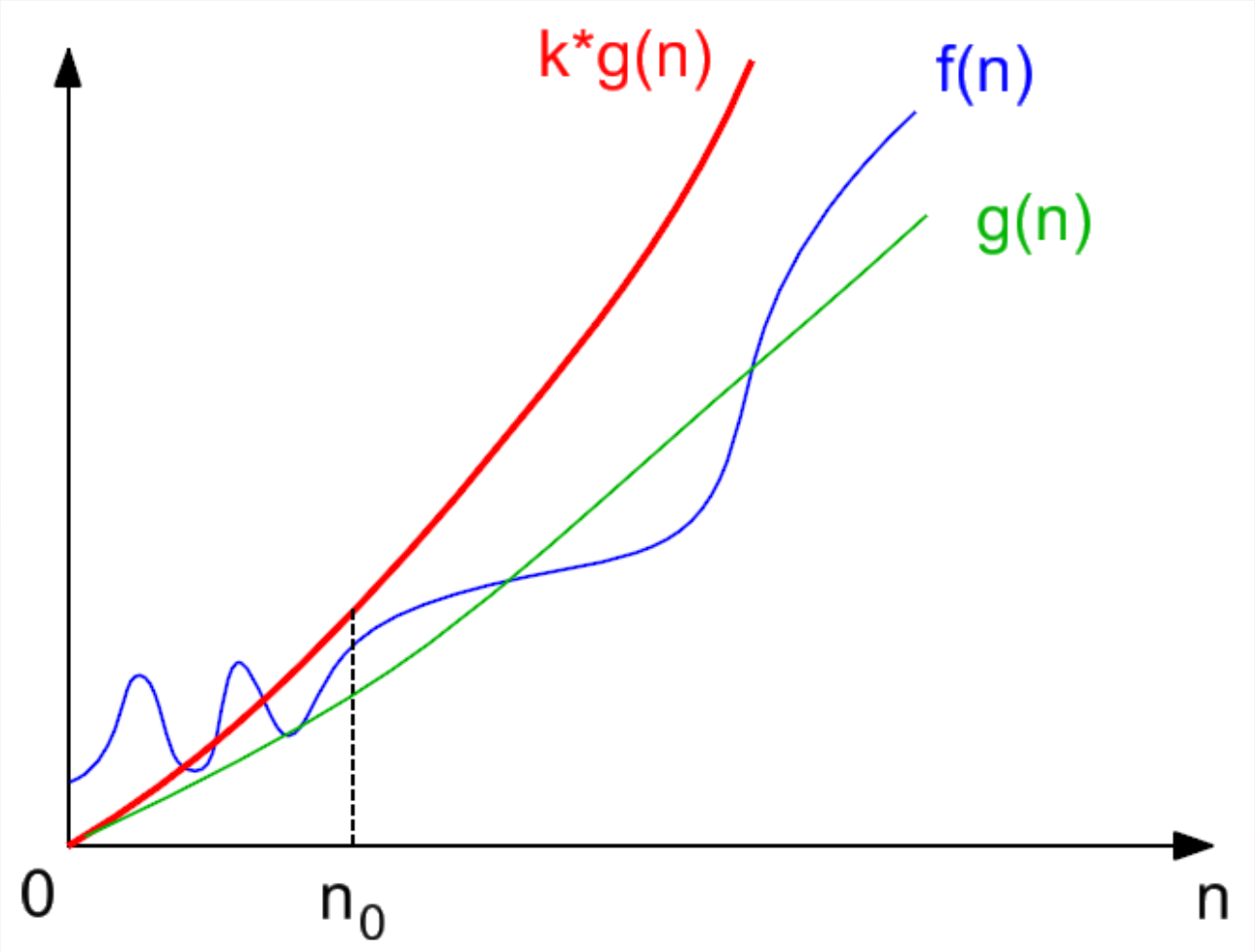
Definice:
Pro libovolné funkce $f, g: \mathbb{N} \to \mathbb{N}$ *platí:* $f(n) \in O(g(n)) \Leftrightarrow \exists k \in \mathbb{R}$, $\exists n_0 \in \mathbb{N}, \forall n > n_0$*:*

$$f(n) \leq k \cdot g(n).$$

Interpretace: f roste nejvýše tak rychle jako g.

Nejčastější typ odhadu.
Informuje o nejpomalejším možném řešení problému.

15. Ukázka asymptotického horního odhadu $O(g(N))$



Příklad: Platí, že $20n^2 \in O(n^2)$?. Řešení: $20n^2 \leq k \cdot n^2$, $k \geq 20$.

16. Asymptotický dolní odhad $\Omega(g(n))$

Ilustruje nejlepší možný případ doby běhu algoritmu.

Tzv. Ω-notace.

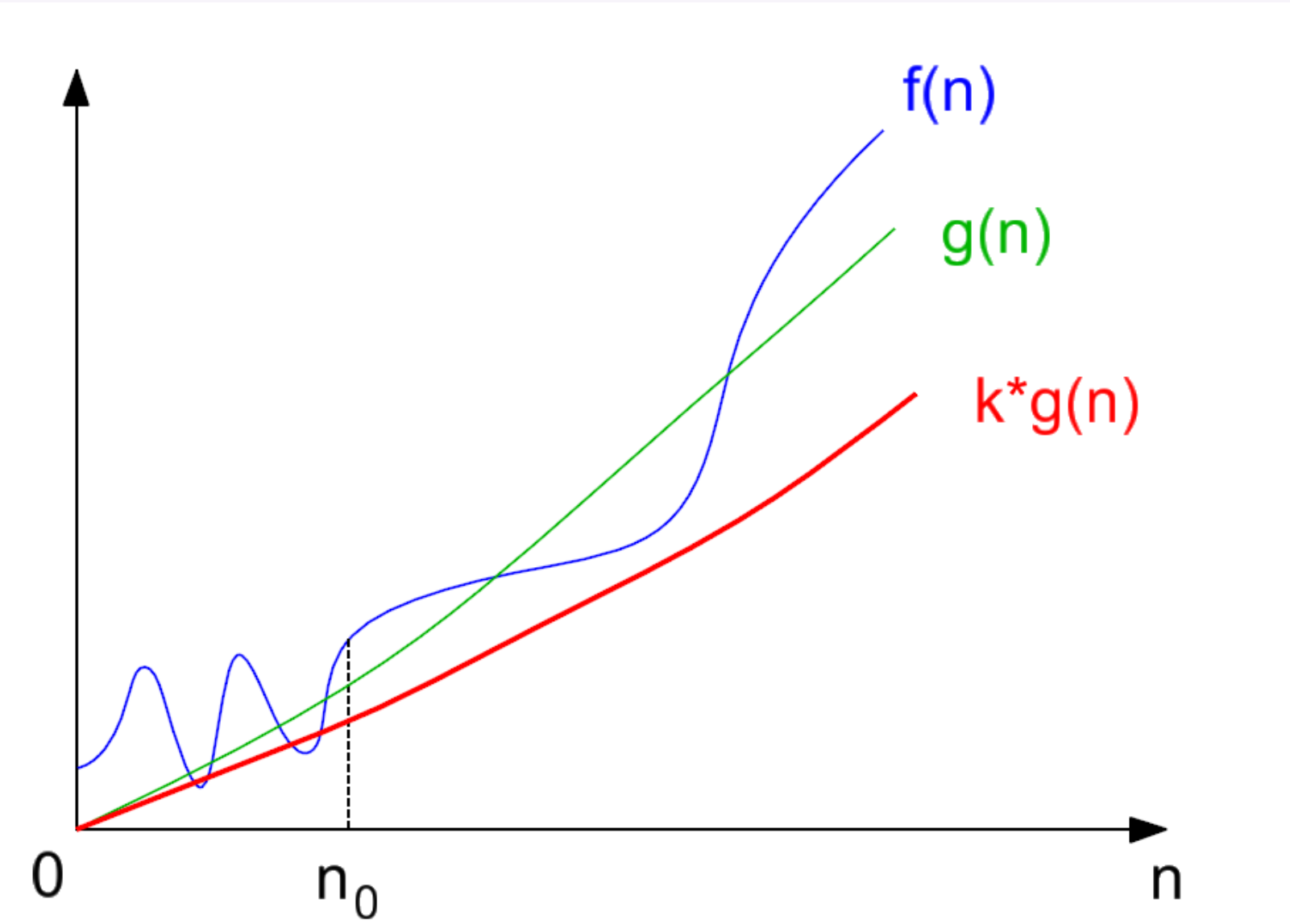
Definice:
Pro libovolné funkce $f, g: \mathbb{N} \to \mathbb{N}$ *platí:* $f(n) \in \Omega(g(n)) \Leftrightarrow \exists k \in \mathbb{R}$, $\exists n_0 \in \mathbb{N}, \forall n > n_0$*:*

$$f(n) \geq k \cdot g(n).$$

Interpretace: f roste nejméně tak rychle jako g.

Používán méně častěji.
Zpravidla nás nezajímá, jak nejrychleji problém vyřešíme.

17. Ukázka asymptotického dolního odhadu $\Omega(g(N))$



18. Asymptotický oboustranný odhad $\Theta(g(n))$

Popisuje očekávanou složitost (tj. průměrný případ).

Tzv. Θ-notace.

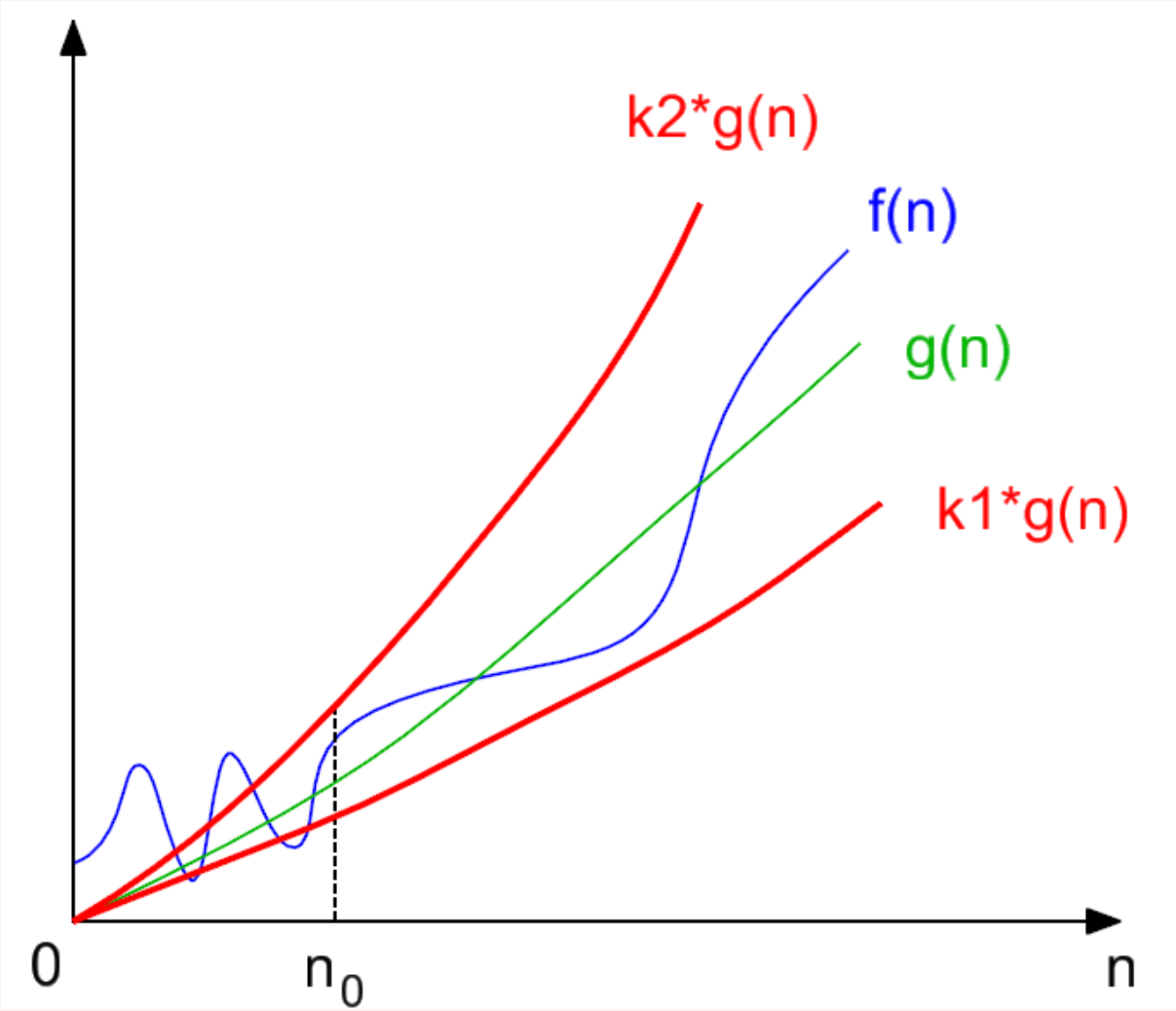
Definice:
Pro libovolné funkce $f, g: \mathbb{N} \to \mathbb{N}$ *platí:* $f(n) \in \Theta(g(n)) \Leftrightarrow \exists k_1 \in \mathbb{R}$, $\exists k_2 \in \mathbb{R}, \exists n_0 \in \mathbb{N}, \forall n > n_0$*:*

$$0 \leq k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n).$$

Odhad stejné rychlosti růstu, f roste stejně rychle jako g.
Nalezení funkce, která roste asymptoticky stejně rychle až na konstantu.
Odpovídá původní definici “efektivity” algoritmu.

Často používaný typ odhadu.

19. Ukázka asymptotického oboustranného odhadu $\Theta(g(N))$



20. Ukázka $\Theta(g(N))$ odhadu

Platí, že $9n^2 - 5n + 6 \in \Theta(n^2)$?

Nalezneme vhodné konstanty k_1, k_2

$$k_1 n^2 \leq 9n^2 - 5n + 6 \leq k_2 n^2,$$

např.: $k_1 = 8$, $k_2 = 9$.
Pak $8n^2 \leq 9n^2 - 5n + 6 \Rightarrow n^2 - 5n + 6 \geq 0$, $n \geq 3$.
Pak $9n^2 - 5n + 6 \leq 9n^2 \Rightarrow 5n \geq 6$, $n \geq \frac{6}{5}$.
Protože $n \geq 3 \wedge n \geq \frac{6}{5} \Rightarrow n \geq 3$, $n_0 = 3$.

21. Charakteristika algoritmů dle časové složitosti

Složitost	**Vyjádření**	**Charakteristika**
Konstantní	1	Konstantní doba běhu programu. Nezávisí na vstupních datech.
Logaritmická	$\log(n)$	Doba běhu se mírně zvětšuje v závislosti na N. Řešení hledáno opakovaným dělení vstupní množiny na menší množiny (hledání v binárním stromu).
Lineární	n	Doba běhu programu roste lineárně s N. Zpracováván každý prvek, např.cyklus.
$n\log(n)$	$n\log(n)$	Doba běhu roste téměř lineárně. Opakované dělení vstupního problému na menší problémy, které jsou řešeny nezávisle (Divide and Conquer, např. třídění).
Kvadratická	n^2	Doba běhu roste kvadraticky, vhodný pro menší množiny dat. Vnořený cyklus.
Kubická	n^3	Doba běhu roste s třetí mocninou, dvojnásobně vnořený cyklus. V praxi snaha nahrazovat algoritmus předchozími dvěma kategoriemi (Greedy algoritmy)
Exponenciální	2^n	Exponenciální doba běhu. Použitelné pro množiny do n=30 Aplikace v kryptografii.

22. Ukázka časové složitosti algoritmů

Vstupní množina $n = 10, 100, 1000$ prvků.
Počet operací nutných pro řešení problému.

Složitost	$n = 10$	$n = 100$	$n = 1000$
Logaritmická složitost	1	2	3
Lineární složitost	10	100	1000
Kvadratická složitost	100	10000	$1.0 \cdot 10^6$
Kubická složitost	1000	$1.0 \cdot 10^6$	$1.0 \cdot 10^9$
Bikvadratická složitost	10000	$1.0 \cdot 10^8$	$1.0 \cdot 10^{12}$
Exponenciální složitost	1024	$1.3 \cdot 10^{30}$	$1.1 \cdot 10^{301}$
Faktoriální složitost	$3.6 \cdot 10^6$	$9.3 \cdot 10^{157}$	$4.0 \cdot 10^{2567}$

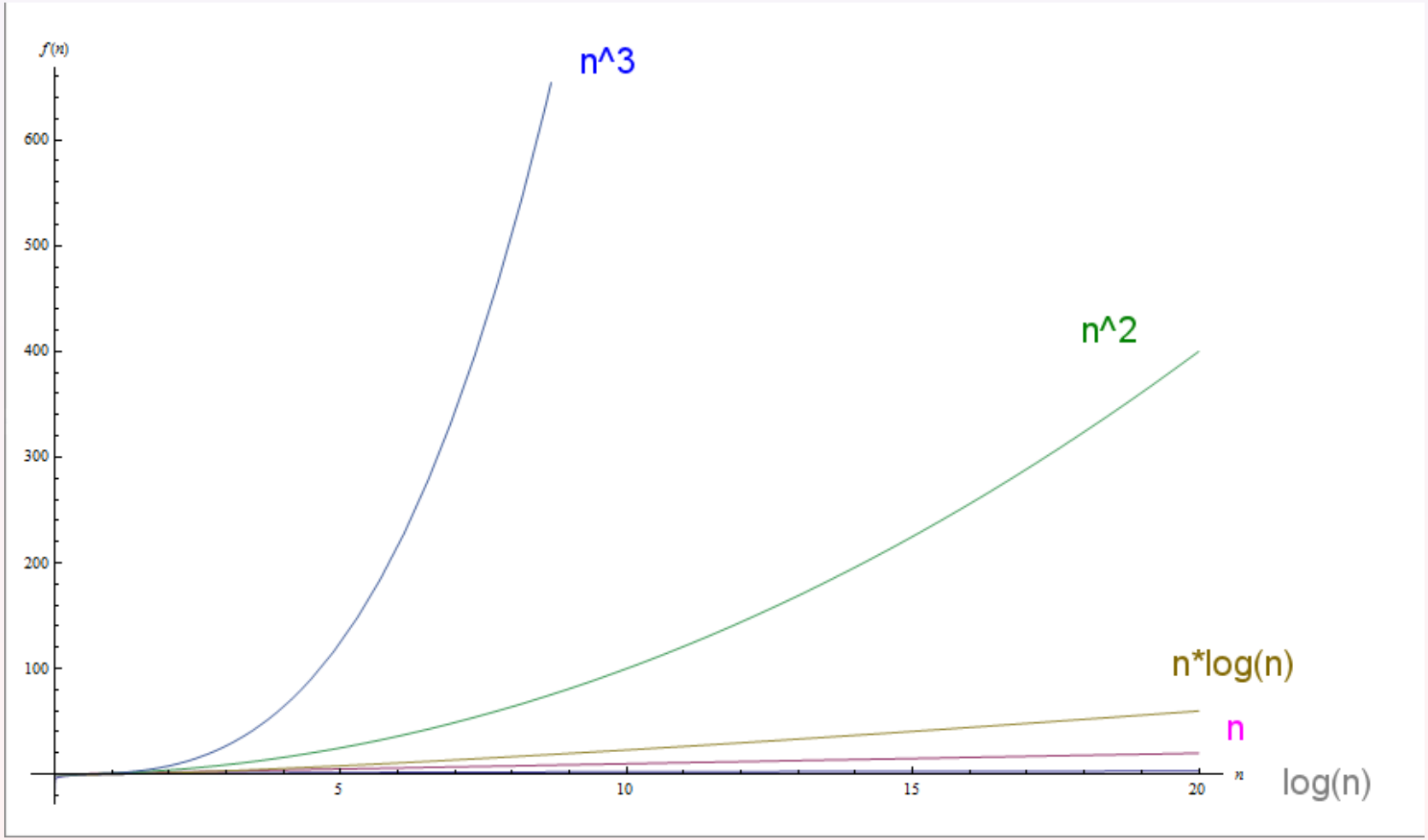
23. Ukázka doby běhu algoritmů

Ukázka doby běhu algoritmu pro $n = 10^9$.
CPU: Počet operací/s $\frac{10^6}{s}, \frac{10^9}{s}, \frac{10^{12}}{s}$.

Složitost	**CPU** $\frac{10^6}{s}$	**CPU** $\frac{10^9}{s}$	**CPU** $\frac{10^{12}}{s}$
Logaritmická složitost	hodiny	vteřiny	okamžitě
Lineární složitost	hodiny	vteřiny	okamžitě
$N \log(N)$	hodiny	vteřiny	okamžitě
Kvadratická složitost	nikdy	roky	týdny
Kubická složitost	nikdy	nikdy	měsíce
Bikvadratická složitost	nikdy	nikdy	roky
Exponenciální složitost	nikdy	nikdy	nikdy
Faktoriální složitost	nikdy	nikdy	nikdy

24. Grafické znázornění algoritmů dle časové složitosti

Vstupní množina: $n \in (0, 20)$



25. Složitost problému vs složitost algoritmu

Problém může být řešen řadou algoritmů s různou složitostí: $O(n^3)$, $O(n^2)$, $O(n \log n)$...
Složitost problému odpovídá složitosti efektivního algoritmu, který problém řeší.

Věty o vztazích složitosti problému a algoritmu:

Věta 1:
Problém P má složitost $O(g(n))$, jestliže pro něj existuje algoritmus řešící ho v $O(g(n))$.

Interpretace:
Pokud se podaří nalézt algoritmus se složitostí $O(g(n))$, *pak má problém i složitost $O(g(n))$.*

Věta 2:
Problém P má složitost $\Omega(g(n))$, jestliže pro něj existuje algoritmus řešící ho v $\Omega(g(n))$.

Interpretace:
Neexistuje algoritmus, který by problém řešil rychleji. Velmi obtížné matematické důkazy!!!

Věta 3:
Problém P má složitost $\Theta(g(n))$, jestliže pro něj existuje algoritmus řešící ho v $\Theta(g(n))$.

Interpretace:
1)+2) Hledán algoritmus se složitostí $O(g(n))$ *+ důkaz, že neexistuje rychlejší algoritmus.*

Tyto algoritmy v konkrétním případě nemusejí být známy.

26. Analýza složitosti algoritmu, Bubble Sort (1/2)

```
def bubbleSort(x):
    for i in range(0, len(x)):
        for j in range(0, len(x) - i - 1):
            if x[j] > x[j + 1]:
                temp = x[j];
                x[j] = x[j + 1];
                x[j + 1] = temp;
```

Hledáme časovou funkci $\tau(f(n))$ a asymptotický horní odhad $O(g(n))$.

27. Analýza složitosti algoritmu, Bubble Sort (2/2)

Závislost na vstupních datech.

Vnější cyklus for, proměnná i, proběhne n krát.
Vnitřní cyklus for, proměnná j, proběhne nejvýše $n-1$ krát.

Pro setříděnou posloupnost neprovedeno žádné prohození $x[i]$ a $x[i+1]$.
Pro nesetříděnou posloupnost provedeno $n-1$ prohození $x[i]$ a $x[i+1]$.
Pro běžná data bude počet prohození v intervalu $(0, n-1)$.

Časová funkce

$$\begin{aligned} f(n) &= n-1+n-2+\ldots+2+1+0, \\ &= (0+n-1)\frac{n-1}{2}, \\ &= \frac{n^2}{2}-n+\frac{1}{2}. \end{aligned}$$

Zanedbání konstant a nevýznamných členů:

$$g(n) \equiv n^2.$$

Algoritmus náleží do kvadratické třídy $O(n^2)$.

28. Třída složitosti algoritmů

Zaváděny pro kategorizaci problémů dle jejich složitosti.

Rozčlenění problémů do tříd téže složitosti umožňuje:

1. Odhalovat vzájemné podobnosti těchto problémů.
2. Řešení jednoho problému převodem na jiný problém.

Problém se snažíme "zařadit" do třídy s co nejnižší složitostí.

Většina problémů má limitní hranici složitosti, po jejímž dosažení již algoritmus nelze urychlit.

Definice:
Pro funkci $f : N \to N$ *označujeme třídou časové složitosti* $\tau(f(n))$ *množinu problémů* P*, které lze řešit s časovou složitostí* $O(f)$*.*

$$P \in \tau(n) \subseteq \tau(n \cdot \log n) \subseteq \tau(n^2) \subseteq \tau(n^3) \subseteq \tau(2^n).$$

29. Třídy složitosti

Dvě základní třídy složitosti:

- třída složitosti PTIME (problémy rychle řešitelné),
- třída složitosti NPTIME (problémy rychle verifikovatelné).

Třída složitosti PTIME:

Do této třídy patří problémy řešitelné algoritmy s polynomiální složitostí: $O(n^c)$.

$$PTIME = \bigcup_{c=0}^{\infty} \tau(n^c)$$

Problémy, pro něž existuje polynomiální algoritmus.
Výpočetně snadné, zvládnutelné, problémy.

V praxi nejčastěji používány algoritmy se složitostí:
$O(n), O(n\log(n), O(n^2), O(n^3)$.

Nejznámější problémy třídy PTIME: aritmetické operace, třídění, vyhledávání, některé grafové či geometrické algoritmy.

30. Deterministický vs. nedeterministický algoritmus

Stručné připomenutí:

Deterministický algoritmus:
Při opakovaném stejném vstupu $x_i \in X$ dává stejné výsledky $y_i \in Y$, kde $y_i = F(x_i)$

$$F(x_i) = F(x_j), \qquad \forall x_i = x_j.$$

Ke každé hodnotě vstupu definována hodnota výstupu.
Běžný přístup používaný v matematice, ne vždy efektivní.

Nedeterministický algoritmus
Při opakovaném stejném vstupu $x_i \in X$ dává různé výsledky $y_i \in Y$

$$F(x_i) \neq F(x_j), \qquad \forall x_i = x_j.$$

Ke každé hodnotě vstupu definována více variant výstupu.
Algoritmus se rozhoduje náhodně/pseudonáhodně, pomalost.
Existují algoritmy, pro které deterministické řešení není známé.

31. Třída složitosti NPTIME

Třída algoritmů

$$NPTIME = \bigcup_{c=0}^{\infty} \tau(n^c)$$

NP problémy obtížně řešitelné ale verifikovatelné.
Rozhodovatelné nedeterministickými polynomiálními algoritmy v polynomiálním čase $O(n^c)$, $c > 0$.
Existuje verifikační algoritmus pracující v polynomiálně omezeném čase.
Vztah mezi PTIME a NPTIME:

$$PTIME \subseteq NPTIME \text{ nebo } PTIME = NPTIME?$$

Další třídy:

- *NP-complete*
  Nelze nalézt exaktní řešení, lze verifikovat.
- *NP-hard*
  Nelze nalézt exaktní řešení ani verifikovat.

32. Příklady NP-úplných problémů

V současné době existuje přes 1000 NP-úplných problémů.
Není znám polynomiální algoritmus jejich řešení.
Často velmi podobné problémům, u kterých existuje polynomiální algoritmus.

Příklady NP-úplných problémů:

- TSP problem (problém obchodního cestujícího): navštívení všech vrcholů grafu právě jednou, délka cesty nejkratší.
- Problém k barev: obarvení vrcholů grafu tak, aby žádné dva sousední vrcholy nebyly obarveny dvěma stejnými barvami.
- Problém batohu: Jak naskládat do batohu předměty známého tvaru tak, aby zabíraly co nejméně místa?
- Loupežnický problém: rozdělení množiny na dvě podmnožiny se stejným součtem.
- Faktorizace prvočísel.

NP-úplné problémy lze řešit **přibližně**: aproximační algoritmy.