Computational Complexity
Výpočetní složitost

Doc. Ing. Roman Šenkeřík, Ph.D.

Obsah prezentace

- Analýza algoritmů
- Výpočetní složitost (účinnost)
- Asymptotická složitost
- Jak zajistit rychlejší běh algoritmů?

Analýza algoritmů

- V Informatice představuje analýza algoritmů určení výpočetní složitosti algoritmů, tj. množství času, úložného prostoru (paměti) a/nebo dalších zdrojů nezbytných k jejich provedení.

- Obvykle to zahrnuje definování matematické funkce, která určuje **závislost počtu kroků** (*časovou složitost algoritmu – time complexity*) nebo **velikost potřebné paměti**, kterou algoritmus použije (*paměťová/prostorová složitost – space complexity*) **na velikosti vstupu algoritmu**.

- Algoritmus považujeme za efektivní, když jsou výstupy takovéto funkce nízké.

- Protože různé vstupy stejné délky mohou způsobit, že se algoritmus chová odlišně, funkce popisující jeho složitost je obvykle horní hranicí skutečného výkonu algoritmu (upper bound), určeného od nejhorších možných vstupů do algoritmu (worst case scenario).

Analýza algoritmů

- **Časová a prostorová složitost / Time and space complexity:**

- Obecně řečeno: ***časová složitost*** udává, **jak dlouho** bude algoritmus běžet **v závislosti na velikosti vstupních dat.**

- ***Prostorová (paměťová) složitost*** definuje, **kolik paměti** je potřeba k provedení algoritmu **v závislosti na velikosti vstupních dat.**

- Tyto dvě složitosti spolu úzce souvisí. Výpočet můžeme urychlit tzv. předpočítáním některých dat.

- Příkladem je program, který na vstupu dostane n, kde $1 \geq n \geq 50$ a má odpovědět, pokud lze šachovnici velikosti n x n „proskákat“ pomocí šachového koně, tak abychom přesně navštívili každé políčko jednou. Všechny odpovědi lze předem vypočítat. Samotný program se pak jen podívá do tabulky a vrátí odpověď. Užitečnějším příkladem může být předpočítání všech prvočísel (z potřebného rozsahu).

- Musíme však ale uložit do paměti všechny předem vypočítané výsledky, a to může zvýšit prostorovou složitost. Proto se často stává, že rychlejší algoritmy mají vyšší prostorovou složitost a naopak.

Analýza algoritmů

- Můžeme také uvážit další metriky:
  - Množství použité komunikace (communication complexity)
  - Množství procesorů/vláken (parallel computing)
  - …
- Teorie vypočítatelnosti (Computability theory) - úzce souvisí s teorií složitosti, ale řeší obecnou otázku o všech možných algoritmech, které by bylo možné použít k vyřešení stejného problému.

Teorie výpočetní složitosti

- A co je nejdůležitější, výpočetní složitost zahrnuje kategorizaci problémů, zda jsou „snadné“ nebo „těžko“ řešitelné.

- Jednou z rolí teorie výpočetní složitosti je určit praktické limity toho, co počítače mohou a nemohou dělat, tj. problém P versus NP, což je jeden ze sedmi „Millenium Prize Problems“.

- Toto klasifikační schéma zahrnuje dobře známé třídy P a NP; termíny „NP-kompletní a NP-těžký, které se vztahují obecně k třídě NP ..

- V teoretické analýze algoritmů je běžné odhadovat jejich **složitost v asymptotickém smyslu**, tj. ***odhadnout funkci složitosti pro libovolně velký vstup***. (A toto je téma této prezentace).

Teorie výpočetní složitosti

- Abychom plně porozuměli složitosti algoritmu, musíme nejprve definovat:
  - algoritmy,
  - problémy,
  - “instance” problému.
- Musíme také pochopit, jak se „měří“ velikost instance problému a dále postupy nebo kroky v algoritmu.

Teorie výpočetní složitosti

- Problém je abstraktní popis spojený s otázkou, která vyžaduje řešení.

- Například problém obchodního cestujícího (TSP) zní: „Pokud vezmeme v úvahu graf s uzly a hranami a „náklady cesty“ spojené s hranami, jaká je nejlevnější uzavřená cesta obsahující každý z uzlů přesně jedenkrát?“

- Instance problému v tomto případě zahrnuje přesnou specifikaci dat. Například „graf obsahující uzly 1,2,3,4,5 a 6 a hrany (1,2) s cenou 10 (1,3) s cenou 14, ...“ atd.

Teorie výpočetní složitosti

- Algoritmus pro jakýkoli problém je sada instrukcí/příkazů, která zaručeně najde správné řešení jakékoli instance v konečném počtu procedur nebo kroků. Jinými slovy, pro problém T představuje algoritmus konečný počet kroků pro výpočet T(*x*) pro jakýkoli daný vstup *x*.

- V teorii, matematičtí a teoretičtí vědci „modelují“ algoritmy pomocí matematické konstrukce zvané Turingův stroj „Turing Machine“.

- V jednoduchém modelu výpočetního zařízení „krok“ sestává z jedné z následujících operací, jako je sčítání, odčítání, násobení, dělení s konečnou přesností a srovnání dvou hodnot. Pokud algoritmus pro některou instanci vyžaduje 100 sčítání a 220 srovnání, v tomto případě říkáme, že algoritmus potřebuje v této instanci přibližně 320 kroků.

Teorie výpočetní složitosti

- Aby všechny tyto hodnoty měly smysl (počet kroků), bylo by dobré je vyjádřit jako funkci velikosti odpovídající instance.

- **Ale určit přesnou funkci by bylo nepraktické!**

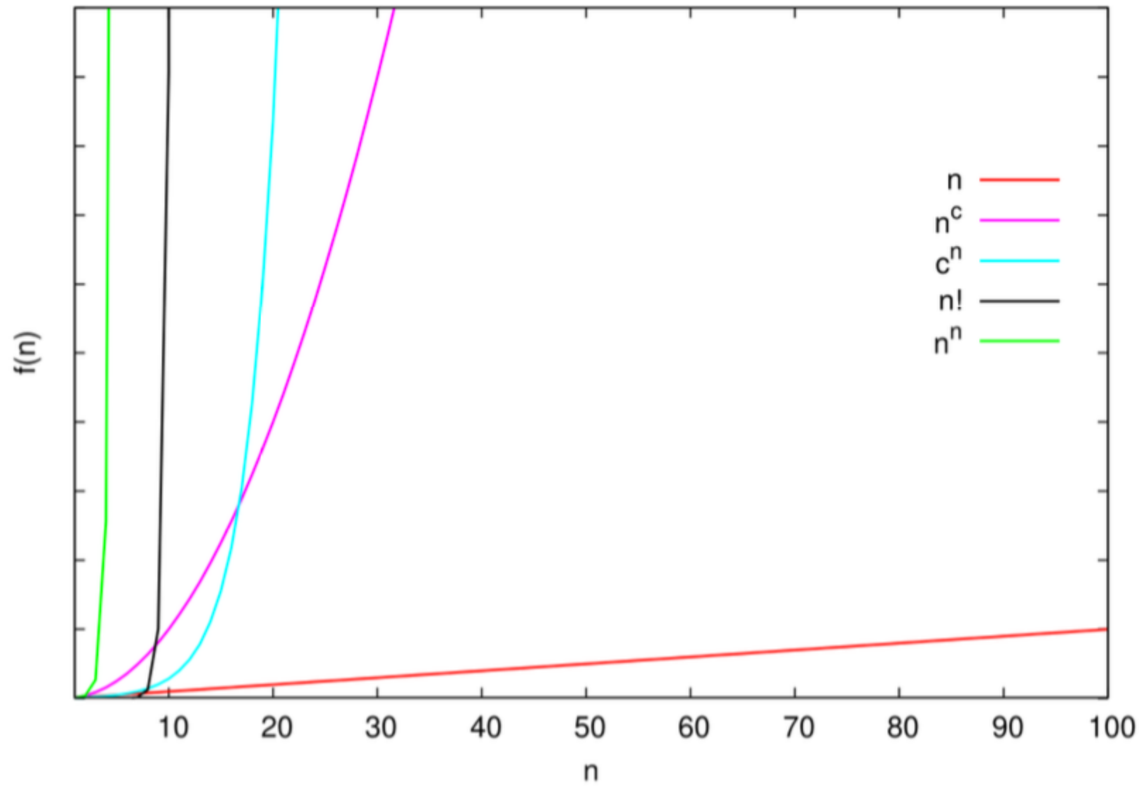
- Alternativně, protože se pozornost zaměřuje na to, kolik času algoritmus vyžaduje (*v tom nejhorším případě*), vyjadřuje se funkce asymptoticky, tedy jak se velikost instance zvětšuje, formulujeme jakousi jednoduchou funkci velikosti vstupních dat, která je přiměřeně těsnou horní hranicí přesného počtu kroků.

- V tomto případě může být taková funkce označována jako složitost nebo doba běhu algoritmu.

Teorie výpočetní složitosti – Potřebujeme ji?

- **Proč je nutné zabývat se výpočetní složitostí algoritmů?**

- **Čas programátora je dražší než čas uživatele. Kromě toho výpočetní výkon stále roste a roste. Nebylo by tedy lepší zanedbávat složitost a nezvyšovat náklady na vývoj? Následující obrázek je odpovědí…**



Teorie výpočetní složitosti – Potřebujeme ji?

n / Funkce	10	50	100	300	1000
Polynomiální					
5n	50	250	500	1500	5000
$n \log_2 n$	33	282	665	2469	9966
n^2	100	2 500	10 000	90 000	1 milion (7 cifer)
n^3	1000	125 000	1 milion (7 cifer)	27 milionů (8 cifer)	1 bilion (10 cifer)
Exponenciální					
2^n	1024	16ciferné číslo	31 ciferné číslo	91 ciferné číslo	302 ciferné číslo
$n!$	3,6 milionů (7 cifer)	65 ciferné číslo	161ciferné číslo	623 ciferné číslo	obrovské číslo
n^n	10 bilionů (11 cifer)	85 ciferné číslo	201ciferné číslo	744 ciferné číslo	obrovské číslo

Pro srovnání: počet protonů ve viditelném vesmíru má 79 cifer. Počet mikrosekund od „velkého třesku" má 24 cifer.

Teorie výpočetní složitosti – Potřebujeme ji?

Rozměr vstupu	$\log_2 n$	n	$N . \log_2 n$	n^2	n^3	n^4	2^n	n!
2	10 µs	20 µs	20 µs	40 µs	80 µs	160 µs	40 µs	20 s
5	23,1 µs	50 µs	116 µs	250 µs	1,25 ms	6,25 ms	320 µs	1,2 ms
10	33,2 µs	100 µs	332 µs	1 ms	10 ms	100 ms	10,2 ms	1,17 s
15	39,1 µs	150 µs	587 µs	2,25 ms	33,8 ms	507 ms	328 ms	15,1 dní
20	43,2 µs	200 µs	864 µs	4 ms	80 ms	1,6 s	10,5 s	771000 let
25	46,4 µs	250 µs	1,16 ms	6,25 ms	156 ms	3,91 s	5,59 min	∞
30	19,1 µs	300 µs	5,73 ms	9 ms	270 ms	37,5 s	2,98 h	∞
50	56,4 µs	500 µs	28,2 ms	25 ms	1,25 s	1,04 min	357 let	∞
100	66,4 µs	1 ms	6,64ms	100 ms	10 s	16,7 min	∞	∞
200	76,4 µs	2 ms	15,3 ms	400 ms	1,34min	4,47 hod	∞	∞
500	89,4 µs	5 ms	44,4 ms	2,5 s	4,17 min	13,9 hod	∞	∞

Růst doby výpočtu pro typické třídy časové složitosti

Přípustný rozsah vstupu pro n-rychlejší počítač

	Původní	Rychlejší 2-krát	Rychlejší 5-krát	Rychlejší 10-krát	Rychlejší 100-krát	Rychlejší 1 000-krát
$\Theta(n)$	100	200	500	1 000	10 000	100 000
$\Theta(n^2)$	100	141	223	316	1 000	3 162
$\Theta(n^3)$	100	125	170	215	464	1 000
$\Theta(2^n)$	100	101	102	103	106	109
$\Theta(n!)$	100	100	100	100	100	101

Teorie výpočetní složitosti (Asymptotická)

- Časovou složitost lze vypočítat určením “nákladů” a počtem opakování pro každou operaci algoritmu (viz následující příklad):

Insertion Sort(A)	cost	amount
for j=2 to A.lenght do	c1	n
key←A[j]	c2	n-1
i←j-1	c3	n-1
while i > 0 ∧ A[i] > key do	c4	$\sum_{j=2}^{n} t_j$
A[i+1] ← A[i]	c5	$\sum_{j=2}^{n}(t_j - 1)$
i←i-1 od	c6	$\sum_{j=2}^{n}(t_j - 1)$
A[i+1] ← key od	c7	n-1

- Takže v nejhorším případě:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(\tfrac{n(n-1)}{2} - 1) + c_5(\tfrac{n(n-1)}{2}) + c_6(\tfrac{n(n-1)}{2}) + c_7(n-1)$$

- Zjednodušeně:

$$= (\tfrac{c_4}{2} + \tfrac{c_5}{2} + \tfrac{c_6}{2})n^2 + (c_1 + c_2 + c_3 + \tfrac{c_4}{2} - \tfrac{c_5}{2} - \tfrac{c_6}{2} + c_7)n - (c_2 + c_3 + c_4 + c_7)$$
$$= an^2 + bn + c$$
$$= \Theta(n^2)$$

Teorie výpočetní složitosti (Asymptotická)

- Asymptotická složitost nebo analýza algoritmu se týká určení matematických hranic nebo rámců jeho běhové výkonnosti.

- Pomocí asymptotické analýzy můžeme velmi dobře definovat nejlepší případ, průměrný případ, včetně nejhoršího scénáře algoritmu.

- Kromě toho je asymptotická analýza vázána na vstup, to znamená, že pokud v algoritmu není žádný vstup, lze dospět k závěru, že funguje v konstantním čase. Kromě vstupu jsou všechny ostatní klíčové faktory považovány za konstantní.

- Využívají se:

  - **O notace (Big O notation/Omicron notation)**

  - **Omega notace (Big Omega notation)**

  - **Theta notace (Big Theta notation)**

Teorie výpočetní složitosti (Asymptotická)

- Asymptotická složitost se týká stanovení doby běhu jakékoli operace v matematických jednotkách výpočtu. Například doba běhu jedné operace je vypočítána jako f(*n*) a může být dokonce i jiná pro jinou operaci, která je vypočítána jako g(*n*^2). To znamená, že doba běhu první operace se zvýší lineárně s nárůstem n a doba běhu druhé operace se exponenciálně zvýší, když se *n* zvýší. Podobně však můžeme určit, že doba běhu obou operací bude téměř stejná, pokud *n* je výrazně malé.

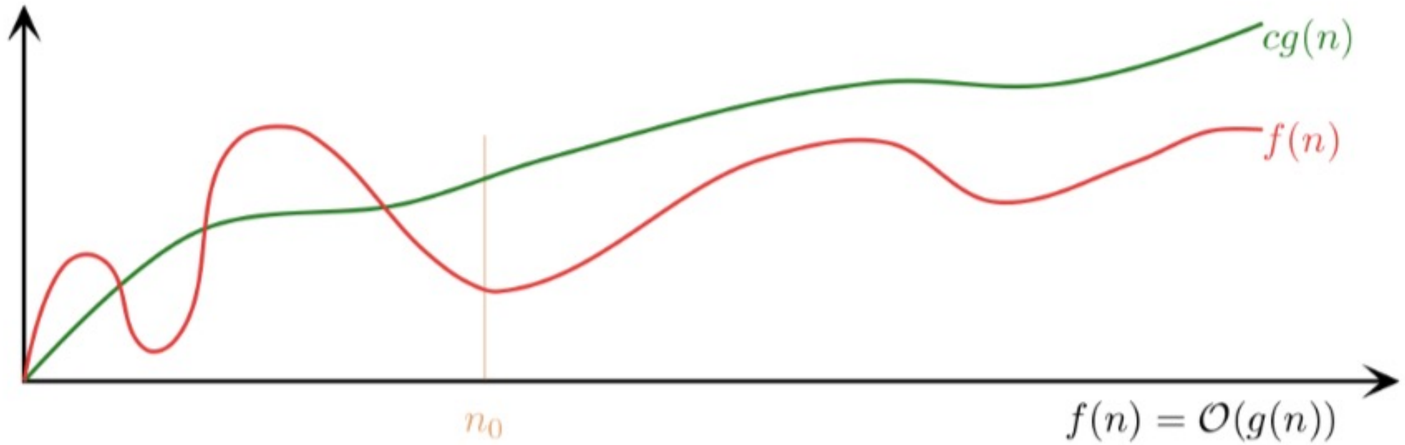
- Čas (nebo prostor) požadovaný algoritmem obvykle spadá do tří kategorií, konkrétně ;
  - Best Case – Minimální čas potřebný pro provedení programu.
  - Average Case – Průměrný čas potřebný pro provedení programu.
  - Worst Case – to je maximální čas potřebný k provedení programu.

Teorie výpočetní složitosti (Asymptotická)
O-notation

- O-notace konkrétně popisuje scénář nejhoršího případu a může být použita k vyjádření potřebného času výpočtu nebo použitého prostoru (např. v paměti nebo na disku) pomocí algoritmu.

- Alternativně je “Big O” notace formálním způsobem, jak vyjádřit horní hranici doby běhu algoritmu.

- **Funkce g se nazývá asymptotická horní hranice funkce (f (n) = O (g (n))). Pokud existují konstanty** c, $n_0 \in N$ takové že: $\forall n \geq n_0$: $f(n) \leq cg(n)$. **Funkce g(n) roste asymptoticky rychleji než f(n)**. Pozn – od určitého zvoleného n_0.

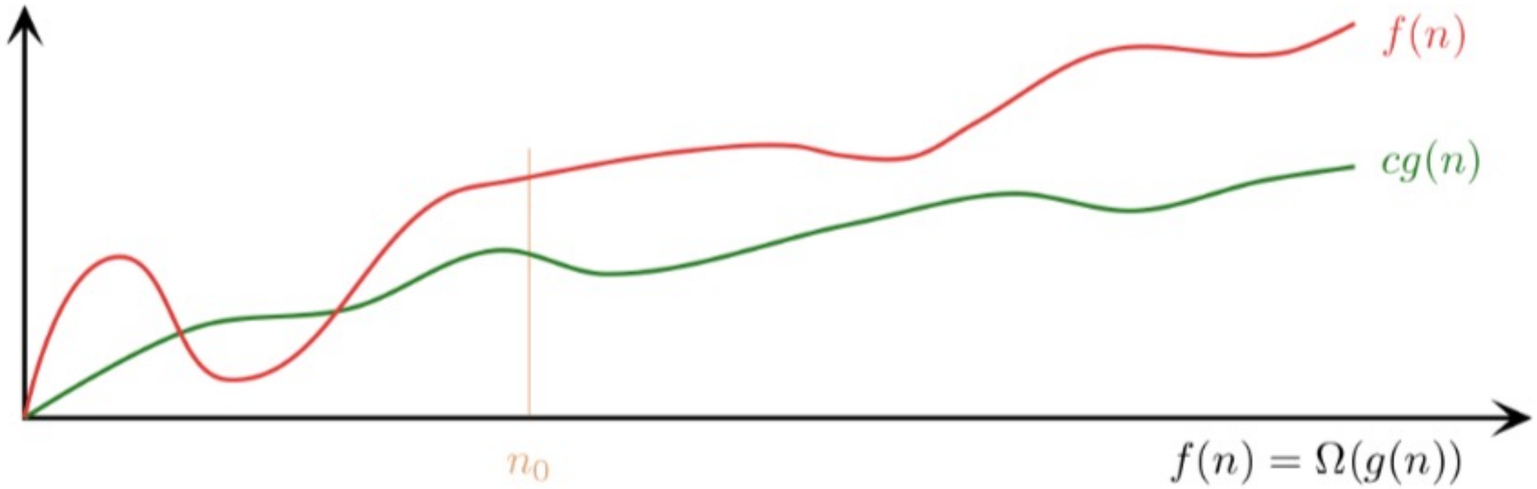


- Pro f(n) = O(g(n)), cg(n) je horní hranicí pro f(n), ***ale neexistuje žádné omezení, jak blízko je limit.***

Teorie výpočetní složitosti (Asymptotická)
Ω-notation

- Ω(n) notace je formální způsob, jak vyjádřit dolní hranici doby běhu algoritmu. Měří nejlepší časovou složitost nebo nejlepší dobu, kterou může algoritmus případně trvat.

- **Funkce g se nazývá asymptotická dolní hranice funkce (f(n)=O(g(n)))**. Pokud existují konstanty c, $n_0 \in$ N takové, že: $\forall n \geq n_0$: $f(n) \geq cg(n)$. **Funkce g(n) roste asymptoticky pomaleji než f(n)**. Pozn – od určitého zvoleného n_0.

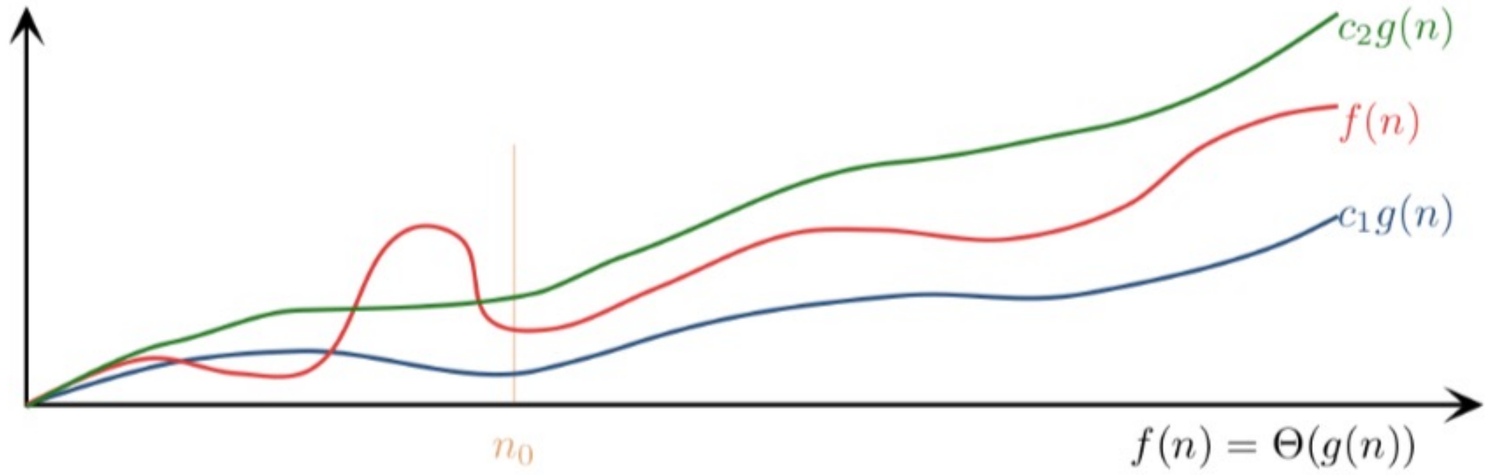


Teorie výpočetní složitosti (Asymptotická)
Θ-notation

- Theta (θ) notace je formální způsob, jak vyjádřit dolní i horní hranici doby běhu algoritmu. Je definována (reptezentována) následovně:

- f(n) = Θ(g(n)), a pozitivní konstanty n_0, c_1 a c_2 takové že: pro všechna $n \geq n_0$ $c_1 g(n) \leq f(n) \leq c_2 g(n)$.

- **Funkce f(n) a g(n) rostou srovnatelně**.



Teorie výpočetní složitosti (Asymptotická)
Vlastnosti

Základní operace:

- $cO(f(n)) = O(f(n))$, kde c je konstanta (ignorujeme konstanty)
- $O(f(n)) + O(f(n)) = O(f(n))$
- $O(O(f(n))) = O(f(n))$
- $O(f(n))O(g(n)) = O(f(n)g(n))$

Transitivita:

- $f(n) = \Theta(g(n))$ a $g(n) = \Theta(h(n))$ implikuje $f(n) = \Theta(h(n))$,
- $f(n) = O(g(n))$ a $g(n) = O(h(n))$ implikuje $f(n) = O(h(n))$,
- $f(n) = \Omega(g(n))$ a $g(n) = \Omega(h(n))$ implikuje $f(n) = \Omega(h(n))$

Teorie výpočetní složitosti (Asymptotická)
Vlastnosti

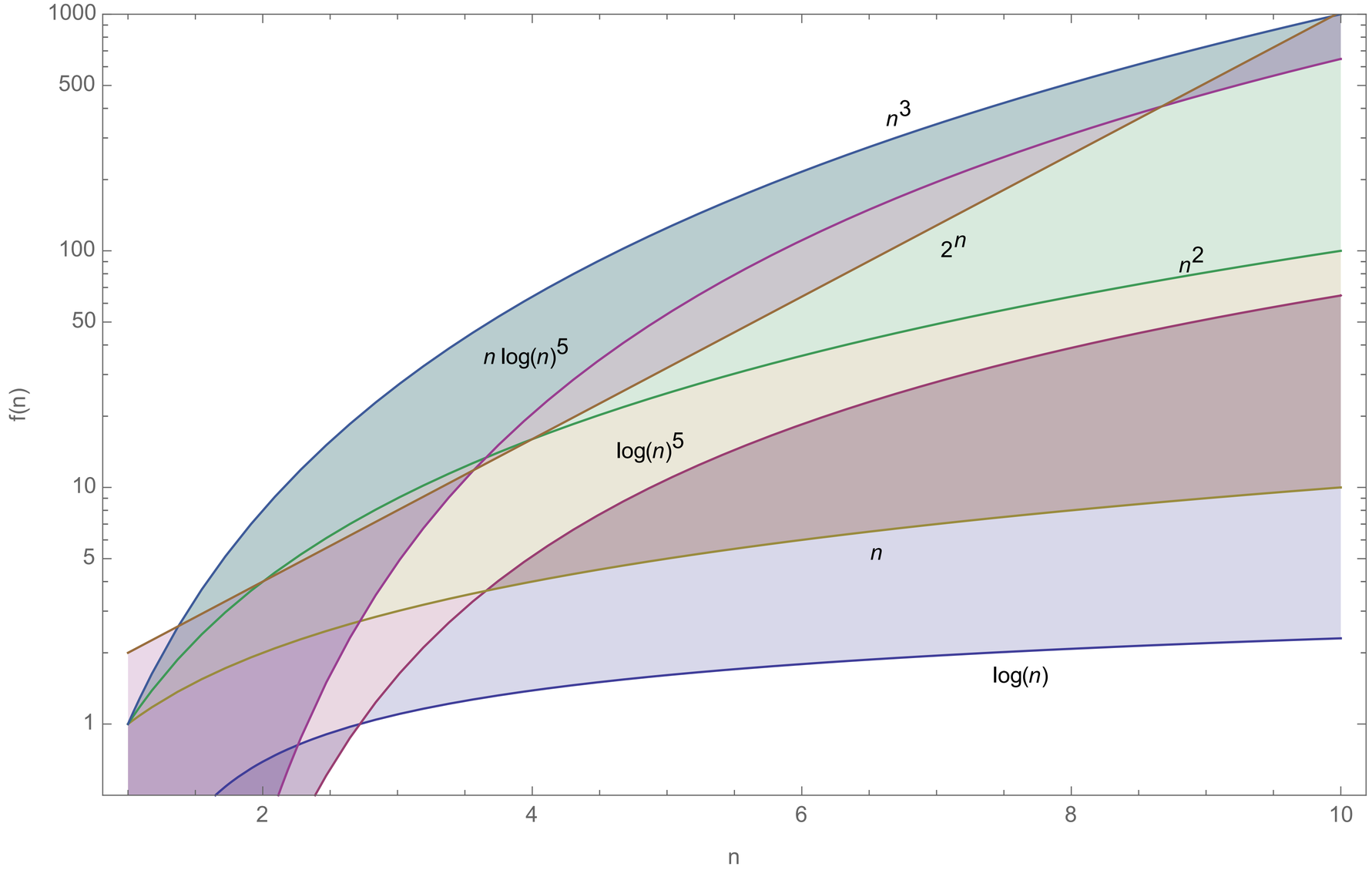
Základní operace/vlastnosti:

- Vyšší mocnina *n* roste rychleji než nižší mocnina *n*.
- Pokud f(n) je polynomiální funkce stupně *k*, pak f(n) = $O(n^k)$.
- Každá exponenciální funkce roste rychleji než každá mocnina.
- Každá kladná mocnina n roste rychleji než ln n.
- n log n roste rychleji než n, ale pomaleji než n^2.
- Všechny logaritmické funkce rostou relativně rychle, tj. $\log_a n = O(\log_b n)$, a $\log_b n = O(\log_a n)$.
- Faktoriál roste rychleji než 2^n.

Běžné třídy asymptotické složitosti

- Θ (1) - čas nezávisí na velikosti vstupu (minimální výskyt)
- Θ (logn) - logaritmický růst (hledání dat v binárních stromových strukturách)
- Θ (n) – lineární růst
- Θ (n * logn) – chytré řadící algoritmy
- Θ (n^2) – kvadratický růst (primitivní/naivní řadící algoritmy)
- Θ (n^3) – kubický růst
- Θ (n^k) – polynomiální růst pro jakékoli přirozené číslo k
- Θ (k^n) - exponenciální růst pro k > 1 (nejčastěji k = 2)
- Θ (n!) - factoriální růst

Běžné třídy asymptotické složitosti



Teorie výpočetní složitosti (Asymptotická)
Vlastnosti

Big-O notation je **relativní reprezentace složitosti** algoritmu.

V té větě je několik důležitých slov:

relativní: můžete porovnávat pouze jablka s jablky. Algoritmus pro aritmetické násobení nemůžete porovnávat s algoritmem, který třídí seznam celých čísel. Ale srovnání dvou algoritmů pro aritmetické operace (jedno násobení, jedno sčítání) vám řekne něco smysluplného.

reprezentace: O-notace redukuje srovnání mezi algoritmy na jedinou proměnnou. Tato proměnná je vybrána na základě pozorování nebo předpokladů. Algoritmy řazení jsou například obvykle porovnávány na základě porovnávacích operací (porovnání dvou uzlů k určení jejich relativního uspořádání). To předpokládá, že srovnání je „drahé“ (časově náročné). Ale co když je srovnání „levné“, ale výměna je „drahá“? Mění to pak srovnání .

složitost: pokud mi trvá jednu sekundu seřadit 10 000 prvků, jak dlouho mi bude trvat seřadit jeden milion? Složitost je v tomto případě relativním měřítkem něčeho jiného.

Jak zrychlit algoritmy?

- Existuje několik možností.
- Nejprve ale musíme zjistit, ve které části programu trávíme nejvíce času.
- To nám řekne, která část je “bottleneck” tzv. úzké hrdlo láhve, a zrychlíme jej.
- Pokud zrychlíme část, ve které výpočet stráví například 2% z celkového času, pak si moc nepomůžeme. Je lepší se zaměřit na operace, při kterých trávíme 80% celkového času.

- Možnosti zrychlení jsou následující (viz další slajdy):

Jak zrychlit algoritmy?

- **Předpočítání**: Některé věci by měly být předem vypočítány na úkor použité paměti. Když je potřebujeme, v konstantní čase je vyhledáme v tabulce.
- **Preprocessing**: Připravit data (redukovat) předem (zvláště když spouštíme vícenásobné analýzy).
- **Odstranit rekurzi**: Například nahradíme faktoriální funkci naprogramovanou rekurzí pomocí for-cyklu. Podobného efektu dosáhneme výběrem správného programovacího jazyka, nebo nepoužíváme věci, které nejsou zrovna nutné a potřeba (například objekty) .
- **Eliminace opakovaných výpočtů**.
- **Optimalizace výpočtu pro daný HW**. Nejčastěji se jedná o přesnou práci s pamětí (neplýtvejte – garbage collector, fit in memory), omezení přístupu k souborům (buď na pevný disk nebo jen psaní textu do terminálu, ukládání do vyrovnávací paměti) atd, vhodně napsaný data collector.

Jak zrychlit algoritmy?

- **Optimalizace zdrojového kodu:** Programátor často chce jen splnit jednorázové zadání, a proto píše téměř nečitelný kód (složitost zneužívající programovací jazyk).
- Tím je zdrojový kód nečitelný, pro jiného programátora (nebo dokonce sám po několika měsících kód nechápe).
- Navíc to zvyšuje riziko, že někde uděláte chybu. Na druhou stranu to moc nepomůže (u již splněného zadání), protože překladače optimalizují kód.
- Optimalizace zdrojového kódu má smysl, pokud je optimalizovaným místem bolttleneckem celého programu a pokud dokážeme kód optimalizovat lépe než kompilátor.

Děkuji za pozornost