Základy algoritmizace

10. Složitost a výkon

doc. Ing. Jiří Vokřínek, Ph.D.
Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Základy algoritmizace

- Dnes:
  - Složitost algoritmů
  - Složitost a typy úloh
  - Porovnání složitosti
  - Skutečný výkon, profilování

Algoritmus

- Algoritmus je konečná posloupnost kroků vedoucí k řešení problému

  - Hromadnost a univerzálnost – řešení třídy úloh
  - Determinovanost – jednoznačný výsledek kroku, jak se bude pokračovat
  - Konečnost – skončí po konečném počtu kroků
  - Rezultativnost – vždy vrátí výsledek (třeba chybu)
  - Korektnost – výstup algoritmu je řešením problému (tj. výsledek je správný)
  - Opakovatelnost – stejný vstup vede pokaždé na stejný výstup

Popis algoritmu

- Definice vstupů
- Definice výstupů

- Příklad – Dijkstrův algoritmus
  - Vstup: graf (seznam hran), počáteční a cílový vrchol
  - Výstup: nejkratší cesta z počátečního do koncového vrcholu, délka cesty

  - Definice datových struktur vstupů a výstupů
  - Definice pomocných datových struktur při výpočtu

- Analýza vlastností algoritmu

Správnost algoritmu

- Korektní – pro všechna přípustná data vede ke správnému výsledku
- Parciálně správný – pokud skončí, je výsledek správný
- Konečný – pro všechna přípustná data skončí po konečném počtu kroků

- Věta o zastavení – *Halting theorem*

  Neexistuje algoritmus, který by pro libovolné slovo w a libovolný algoritmus A rozhodl, zda se A při vstupu w zastaví, či ne.

Je možné otestovat algoritmus pro všechny přípustné vstupy?

Ukazatele kvality algoritmů

- Operační složitost
- Paměťová složitost

Příklad – sekvenční hledání

```
def searchLinear(array, val):
    for i in array:
        elem = array[i]
        if elem == val: return elem
```

- Nejhorší případ – algoritmus proběhne *n* krát
- Paměťová náročnost – řídící proměnná *i*

Ukazatele kvality algoritmů

Příklad – hledání binárním půlením

```
def binSearch(array, val):
    l, r = 0, len(array) - 1
    while r>=l:
        i = int((l + r)/2)
        if val == array[i]: return array[i]
        if val > array[i]:
            l = i + 1
        else:
            r = i - 1
```

- Nejhorší případ – cyklus proběhne $log_2(n)$ krát
- Paměťová náročnost – navíc proměnné *l* a *r*
- Vyhledávací pole musí být uspořádané

Ukazatele kvality algoritmů

- Příklad – tisk Fibonacciho posloupnosti

```
def fibonacci(n):
    if n<2: return 1
    return fibonacci(n-1)+fibonacci(n-2)

def printFib1(n):
    for i in range(n): print(fibonacci(i))


def printFib2(n):
    fib = 1
    fibM1 = 0
    for i in range(n):
        print(fib)
        fibM2 = fibM1
        fibM1 = fib
        fib = fibM1 + fibM2
```

Diskutujte rozdíl složitosti algoritmů

Složitost algoritmu

- Časová složitost – doba provádění algoritmu
- Paměťová složitost – rozsah použité paměti

- Jak vyjádřit složitost algoritmu?
  - Doba výpočtu – závisí na rychlosti procesoru
  - Počet instrukcí – závisí na hardware počítače

- Složitost algoritmu se stanovuje teoretickým rozborem činnosti algoritmu

Závisí jen na velikosti vstupních dat

Složitost algoritmu

- Doba výpočtu – velikost vstupních dat *n*
- Složitost algoritmu vyjádříme funkcí

$$T(n) = a \times f(n) + b,$$

kde

  - *a* je multiplikativní konstanta
  - *b* je aditivní konstanta nezávislá na velikosti vstupních dat

- Obvykle je důležitý pouze typ závislosti (tvar funkce *f(n)*), např:
  - Lineární – $f(n) = n$
  - Polynomiální – $f(n) = n^2$
  - Exponenciální – $f(n) = 2^n$
- Efektivní algoritmy mají složitost maximálně polynomiální

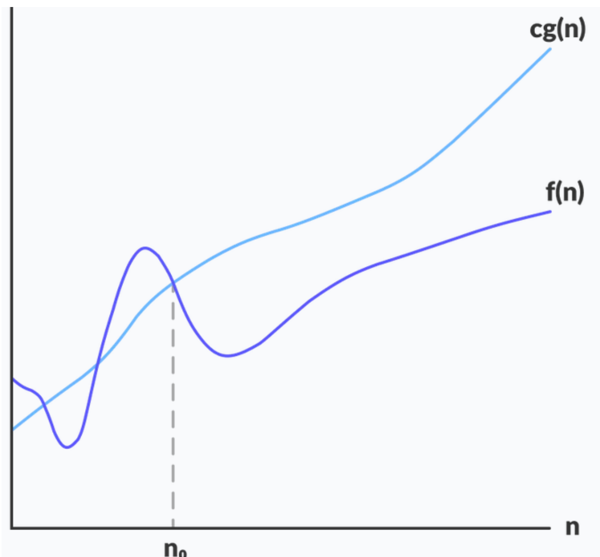
Asymptotická složitost

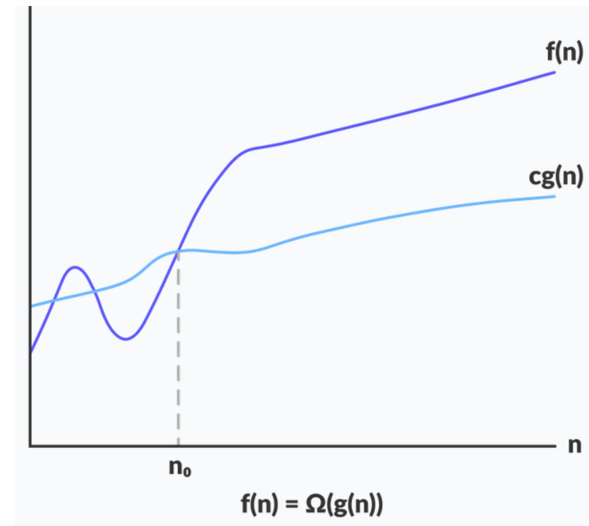
- Popisuje chování funkce *T(n)* pro velká *n*

- Funkce *f(n)* je *O(g)*, jestliže existuje $n_0 \in N$ a $c > 0$ takové, že pro všechna $n \geq n_0$ je $f(n) \leq cg(n)$

Funkce f neroste rychleji než nějaký násobek funkce g

- Funkce *f(n)* je $\Omega(g)$, jestliže existuje $n_0 \in N$ a $c > 0$ takové, že pro všechna $n \geq n_0$ je $f(n) \geq cg(n)$

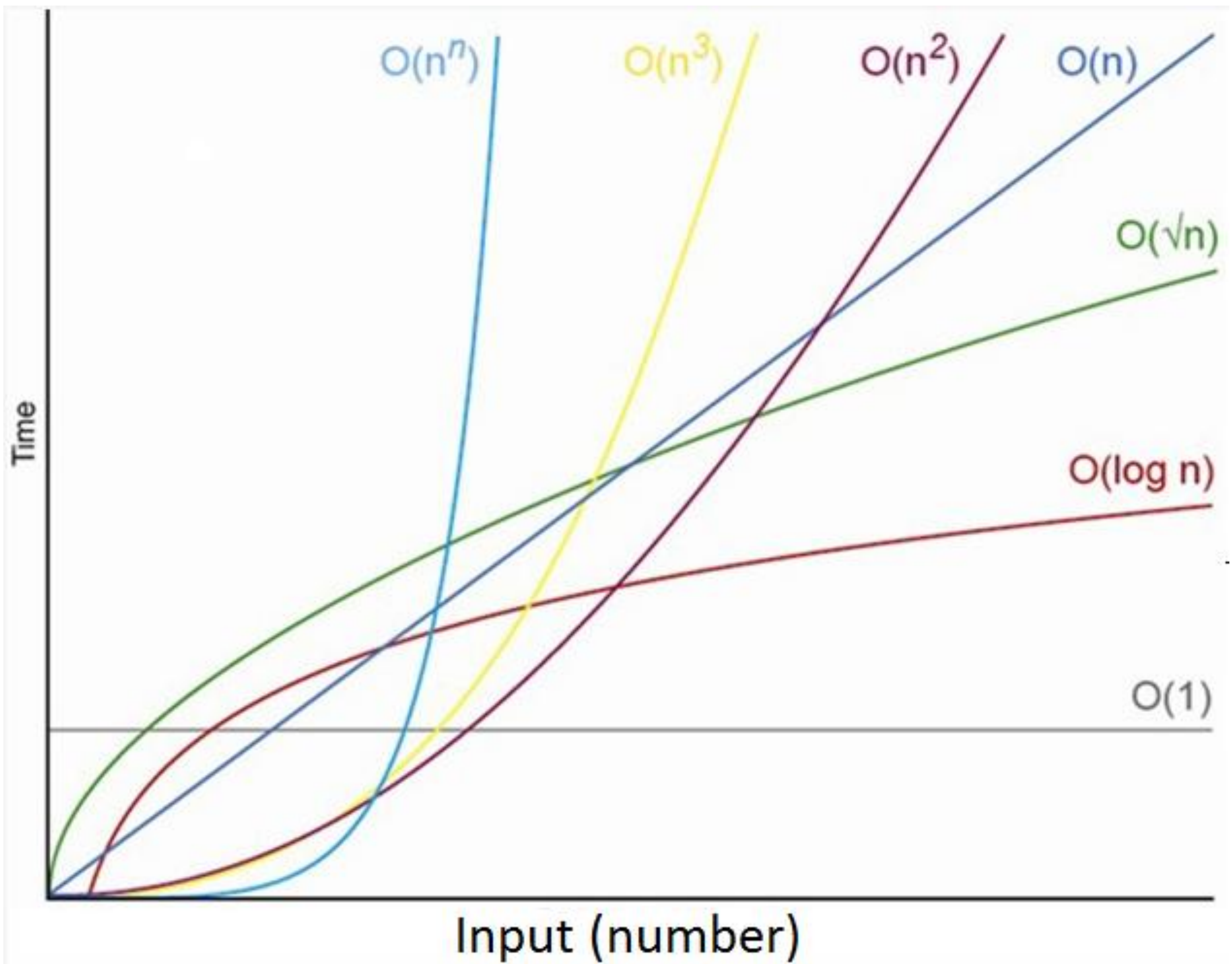




https://www.programiz.com/dsa/asymptotic-notations

Asymptotická složitost

- Horní odhad složitosti $T_{worst}(n)$ udává složitost algoritmu v nejhorším případě
- Algoritmus má složitost $O(f(n))$ pokud funkce $T_{worst}(n)$ je $O(f(n))$
- Dolní odhad složitosti $T_{best}(n)$ je nejkratší čas provedení pro ideální případy vstupních dat
- Střední (průměrný) odhad složitosti $T_{avrg}(n)$ je čas provedení pro „běžné“ případy vstupních dat



https://randerson112358.medium.com/algorithm-analysis-time-complexity-simplified-cd39a81fec71

Asymptotická složitost

- Asymptotická složitost algoritmu *O(f(n))* charakterizuje chování algoritmu pro **velká** *n*
- Není závislá na
  - Hardware počítače
  - Programovacím jazyce
  - Překladači
  - Schopnostech programátora (?)

- Jaká *n* jsou velká?

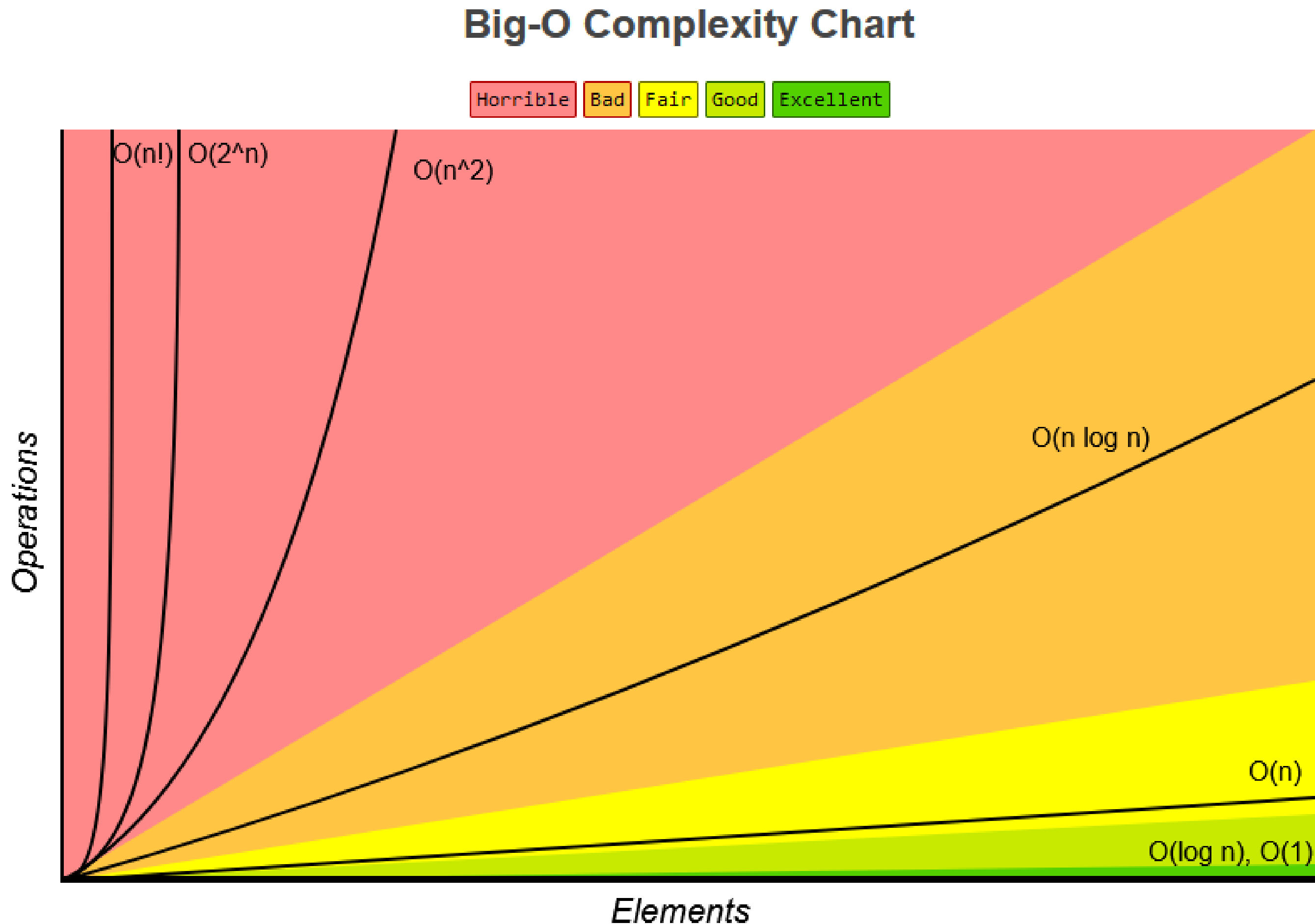
Algoritmus s horší složitostí může fungovat lépe pro malé instance

- Praktický vliv mají faktory uvedené výše, typ a reprezentace dat, atp.

Asymptotická složitost

- Pro každý algoritmus můžeme analyzovat jeho složitost

Přehled například na http://bigocheatsheet.com/



Asymptotická složitost

- Pro každý algoritmus můžeme analyzovat jeho složitost

Přehled například na http://bigocheatsheet.com/

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Asymptotická složitost

- Pro každý algoritmus můžeme analyzovat jeho složitost

Přehled například na http://bigocheatsheet.com/

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Složitost úlohy

- Úloha má horní odhad složitosti $f(n)$, jestliže existuje algoritmus řešící úlohu s časovou složitostí $O(f(n))$
- Úloha má dolní odhad složitosti $g(n)$, jestliže pro každý algoritmus řešící úlohu platí $T_{worst}(n)$ je $\Omega(g(n))$
- Algoritmus je optimální pro danou úlohu, jestliže neexistuje jiný algoritmus, který by úlohu řešil v nejhorším případě s menším počtem základních operací

- Asymptoticky stejná složitost – $f(n)$ a $g(n)$ jsou asymptoticky stejné, pokud

$$\exists c_1, c_2 > 0, \exists n_0 > 0, \forall n \in N, n > n_0 : 0 \le c_1 g(n) \le f(n) \le c_2 g(n)$$

Složitost a typy úloh

Typ a instance úlohy

- Typ úlohy je určen
  - Způsobem, jak je úloha zadána (vstup)
  - Co má být výsledkem (výstup)
  - Jaký je vztah mezi vstupem a výstupem
- Instance úlohy je případ úlohy daného typu
- Pro daný typ úlohy hledáme algoritmus, který jej řeší
- Pro instanci úlohy hledáme implementaci algoritmu (výpočet)

Složitost úloh – P a NP úlohy

- Rozhodovací úloha – řešením je odpověď ano nebo ne
- Každou optimalizační úlohu lze převést na rozhodovací
- Příklad – obchodní cestující (TSP)
  - Optimalizační verze – nalezněte trasu nejkratší délky
  - Rozhodovací verze – existuje trasa délky menší než K?
- Třída P – třída všech rozhodovacích úloh *U*, pro něž existuje polynomiální algoritmus řešící *U*
- Třída NP – třída všech rozhodovacích úloh *U*, pro něž existuje nedeterministický algoritmus pracující v polynomiálním čase
  - Nedeterministická fáze – náhodný řetězec symbolů *s*
  - Deterministická fáze – deterministický algoritmus má na vstupu instanci úlohy a řetězec *s*

NP úlohy

Rozhodovací úloha patří do třídy NP, existuje-li pro ni ověřovací algoritmus s vlastnostmi:

- Vstupem jsou data popisující instanci úlohy délky *d* a *certifikát* jehož velikost je polynomiálně omezena *d*
- Pracuje v polynomiálním čase a vždy dává odpověď „ano“ nebo „nevím“
- Pro instanci úlohy, pro kterou je správná odpověď „ano“ existuje takový certifikát, že ověřovací algoritmus vrátí „ano“
- Pro instanci úlohy, pro kterou je správná odpověď „ne“ dává ověřovací algoritmus vždy „nevím“

NP úlohy

- Příklad – hledání hamiltonovské kružnice
  - Vstupem je graf
  - Certifikát je kružnice
  - Ověřovací algoritmus testuje, zda-li kružnice opravdu prochází přes všechny vrcholy
- Ověřovací algoritmus může být jednoduchý, ale nalézt certifikát je obtížné

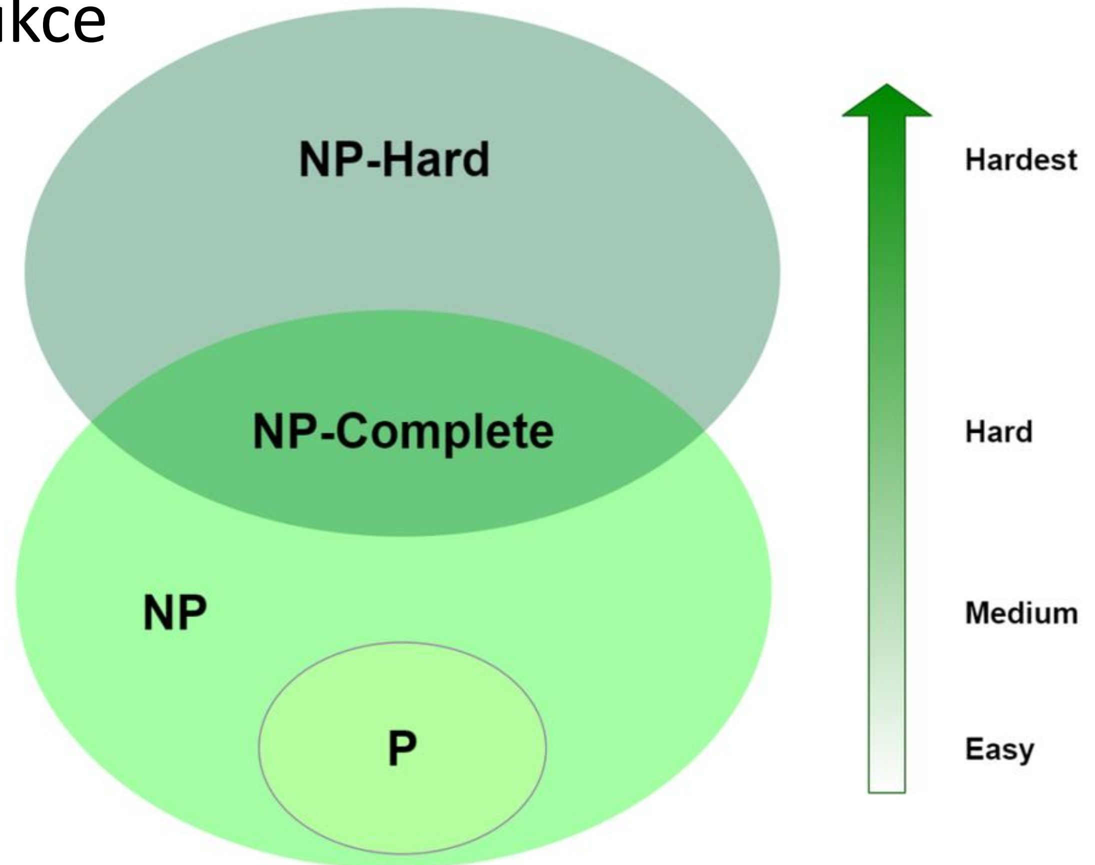
- Důsledek
  - NP úlohy lze ověřit v polynomiálním čase
  - Nevíme zda je lze také v polynomiálním čase vyřešit

NP úlohy

- Redukce úloh
  - Úloha *U* se redukuje na úlohu *V*, jestliže existuje algoritmus *A*, který pro každou instanci úlohy *U* zkontroluje instanci *A(I)* úlohy tak, že *I* je „ano“ instance *U*, právě tehdy když *A(I)* je „ano“ instance *V*
- Polynomiální redukce – *A* je polynomiální
- NP-úplná (complete) úloha je úloha, která je NP úlohou a každá NP úloha se na ní polynomiálně redukuje
- NP-C je třída všech NP-úplných úloh
- NP-těžká (hard) úloha je úloha, na kterou se polynomiálně redukuje každá NP-úloha
- Obecně se má za to, že $P \neq NP$

Složitost úloh – P a NP úlohy

- P – dají se řešit rychle (v rámci možností, viz. „big O“)
- NP – dají se rychle ověřit, těžší na řešení
- NP-C – jako NP + redukce
- NP-těžká
  - Těžko se řeší
  - Těžko se ověřují
  - Redukce



https://www.baeldung.com/cs/p-np-np-complete-np-hard

NP úlohy

- Úlohy ze třídy NP-C jsou z hlediska obtížnosti ekvivalentní
- Existuje-li polynomiální algoritmus, který řeší kteroukoliv úlohu z NP-C, pak P = NP

Do té doby považujeme NP-C úlohy za „polynomiálně neřešitelné“

- NP-těžká úloha nemusí být ve třídě NP – může být těžší
  - NP-těžké optimalizační úlohy mohou mít rozhodovací verze ve třídě NP

  Záleží na formulaci úlohy, certifikát požadujeme pouze pro kladnou odpověď

  - Speciální případy NP-těžkých úloh mohou být polynomiální

  Např. watchman route problem, touring polygons problem

NP úlohy

- Praktický význam NP úloh
  - Hledání polynomiálního algoritmu?
  - NP-těžké úlohy jsou těžké – proto je nutné volit vhodný kompromis mezi rychlostí nalezení řešení a kvalitou řešení
  - Malé instance NP-těžkých úloh lze řešit metodou hrubé síly (brute force)
  - Aproximace a heuristické algoritmy (AI)

- Paměťová náročnost – třída P SPACE
  - Jestliže existuje deterministický algoritmus řešící úlohu *U* a má paměťovou složitost nejhoršího případu $O(p(n))$ pro nějaký polynom $p(n)$

$$P \subseteq NP \subseteq PSPACE$$

Skutečný výkon

Skutečný výkon programu

- Skutečný výkon programu (implementace, nikoli algoritmu!) závisí na mnoha faktorech:
  - Hardware počítače
  - Programovacím jazyce
  - Překladači
  - Schopnostech programátora
  - Representace dat
- Vstupní data – mohou být předzpracovaná
  - Př. vyhledávání v seřazeném poli
- Vhodná volba pomocných struktur
  - Př. prioritní fronta v hledání nejkratších cest
- Specializace na konkrétní vstup (restrikce úlohy)

Skutečný výkon programu

- V praxi nás typicky zajímá jak rychle daný program běží v závislosti na ostatní činnosti
- Například:
  - Zpracování log. souboru 1x za den může trvat 10 minut
  - Zpracování 100 log. souborů 2x za den nemůže trvat 10 minut
  - Výpočet reakce robotu pohybujícího se 3m/s za 50ms je pomalý (50 ms ≈ 15cm)

- Algoritmus o známé složitosti se dá naprogramovat různě
  - Využití znalosti HW, kompilátoru apod.
  - Efektivní správa paměti, znalost náročnosti systémových volání
  - Měření a optimalizace výkonu v reálných podmínkách (cílové zařízení, reálná vstupní data)

Pravidlo 80/20

- 80% strojového času je stráveno vykonáváním 20% zdrojového kódu
- Postup optimalizace
  1. Identifikace nejčastěji prováděného kódu
  2. Optimalizace kódu
  3. Ověření správné činnosti

Podobně pro využití paměti

- Profilování je porozumění chování programu za běhu

Identifikace jak dlouho která část programu trvá

- „Profile“ je množina frekvencí přiřazená pozorovaným událostem za běhu programu

Profil programu je závislý na konkrétních vstupech

Měření času

- Čas běhu programu – reálný, uživatelský, systémový
- Vlastní měření času v programu

V pythonu knihovna datetime

- Profiler – nástroj jak získat profil
  - Transparentnost
  - Efektivita
  - Přesnost

- Profilovací techniky
  - Hardwarové – není nutné modifikovat zdrojový kód
  - Softwarové – Profiler příslušně modifikuje kód

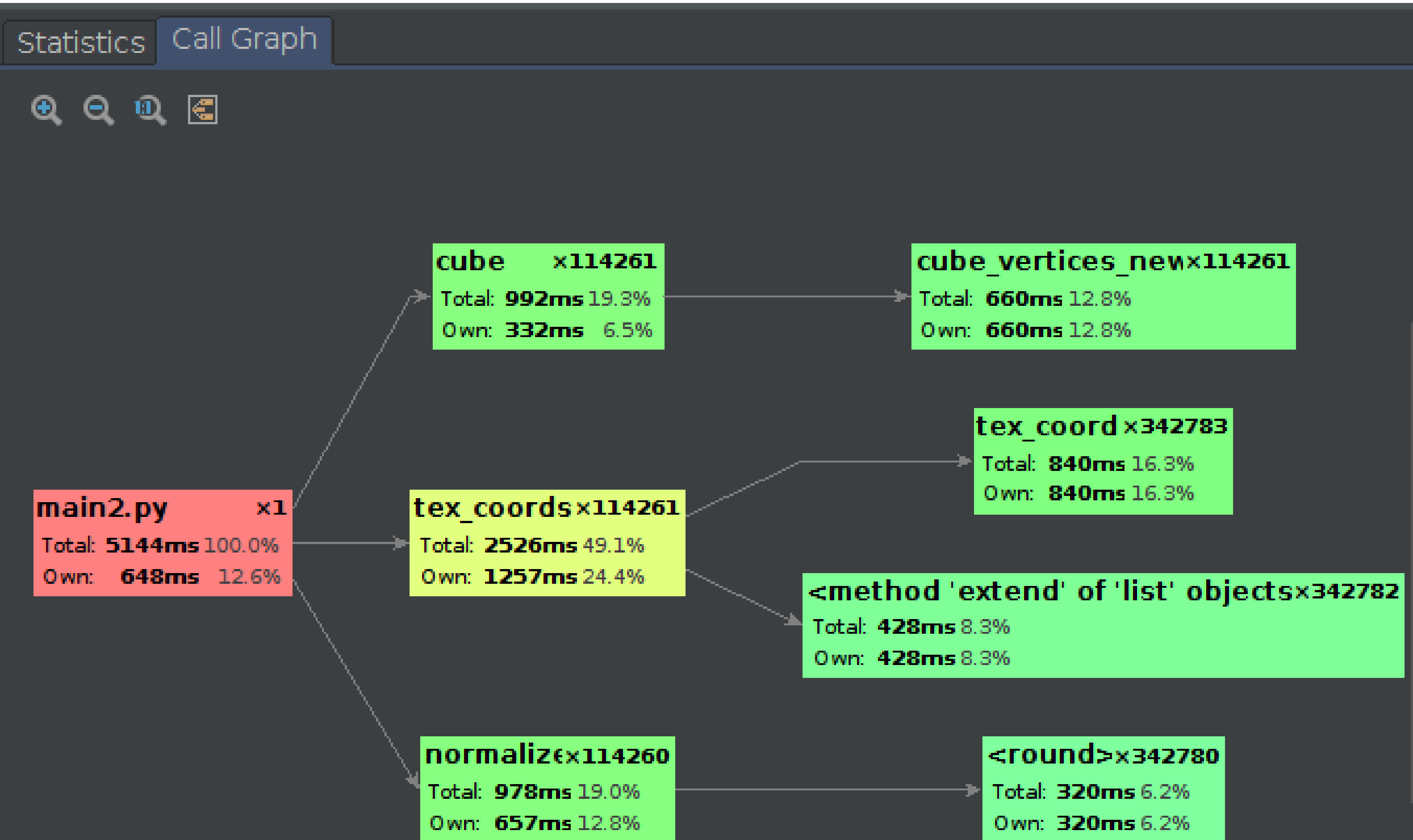
Profilování – plošný profil

Statistics | Call Graph

Function	Call Count	Time (ms)		O	
clear	2145	18009	42.9%	17781	42.3%
<select.select>	1521	10834	25.8%	10831	25.8%
__setitem__	78946	1582	3.8%	1529	3.6%
errcheck	149268	1273	3.0%	1273	3.0%
draw	6434	1645	3.9%	747	1.8%
flip	2145	492	1.2%	488	1.2%
normalize	189873	529	1.3%	394	0.9%
set_3d	2145	516	1.2%	339	0.8%
get_region	80288	1124	2.7%	333	0.8%
_set_attribute_data	80288	3233	7.7%	326	0.8%
cast	84556	293	0.7%	293	0.7%
__init__	80288	256	0.6%	256	0.6%
draw	2043	623	1.5%	256	0.6%
collide	14744	283	0.7%	239	0.6%
cube_vertices_new	41516	215	0.5%	215	0.5%
get_region	80288	704	1.7%	201	0.5%
add_block	91402	752	1.8%	186	0.4%
<method 'remove' of 'list' objects>	8194	147	0.3%	147	0.3%
_parse_data	39921	175	0.4%	142	0.3%
draw_label	2145	1842	4.4%	139	0.3%

Navigate to Source
Show on Call Graph

Profilování – graf volání



Základy algoritmizace

- Dnes:
  - Složitost algoritmů
  - Složitost a typy úloh
  - Porovnání složitosti
  - Skutečný výkon, profilování

Příště přehled programovacích jazyků