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(1))


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 $f(n) \geq cg(n)$

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

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.

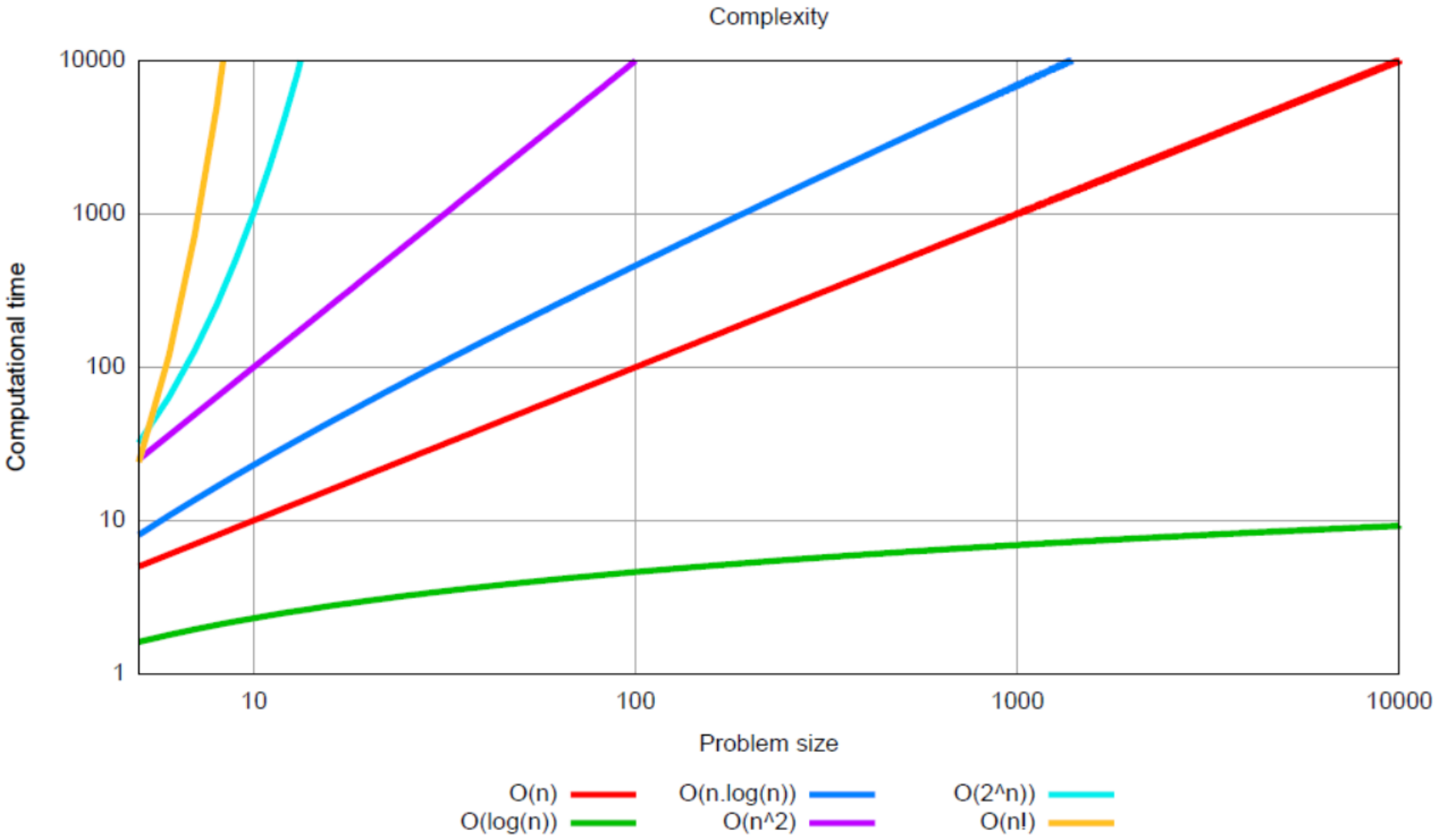
Asymptotická složitost

- Pro každý algoritmus můžeme analyzovat jeho složitost

Přehled například na http://bigocheatsheet.com/

$T(n)$	Name	Problems
$O(1)$	constant	
$O(logn)$	logarithmic	
$O(n)$	linear	easy-solved
$O(nlogn)$	linear-logarithmic	
$O(n^2)$	quadratic	
$O(n^3)$	cubic	
$O(2^n)$	exponential	
$O(n!)$	factorial	hard-solved

http://agafonovslava.com/post/2011/01/14/Algorithm-theory-and-complexity-introduction


Complexity
10000
1000
100
10
1
Computational time
10
100
1000
10000
Problem size
O(n)
O(log(n))
O(n.log(n))
O(n^2)
O(2^n))
O(n!)

Složitost a typy úloh

Turingův stroj

- Abstraktní matematický model počítače
- Páska – reprezentuje paměťové médium, potenciálně nekonečná, políčko obsahuje symbol nebo prázdný symbol
- Čtecí hlava
- Stroj pracuje v cyklu:
  1. Čtení symbolu z pásky
  2. Zápis symbolu na pásku
  3. Přesun hlavy ($\leftarrow$, $\rightarrow$, stop)
- Posun čtecí hlavy závisí na přečteném symbolu a vnitřním stavu stroje
- Nedeterministický – (náhodný?) výběr z více možností

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 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

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

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

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

- 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

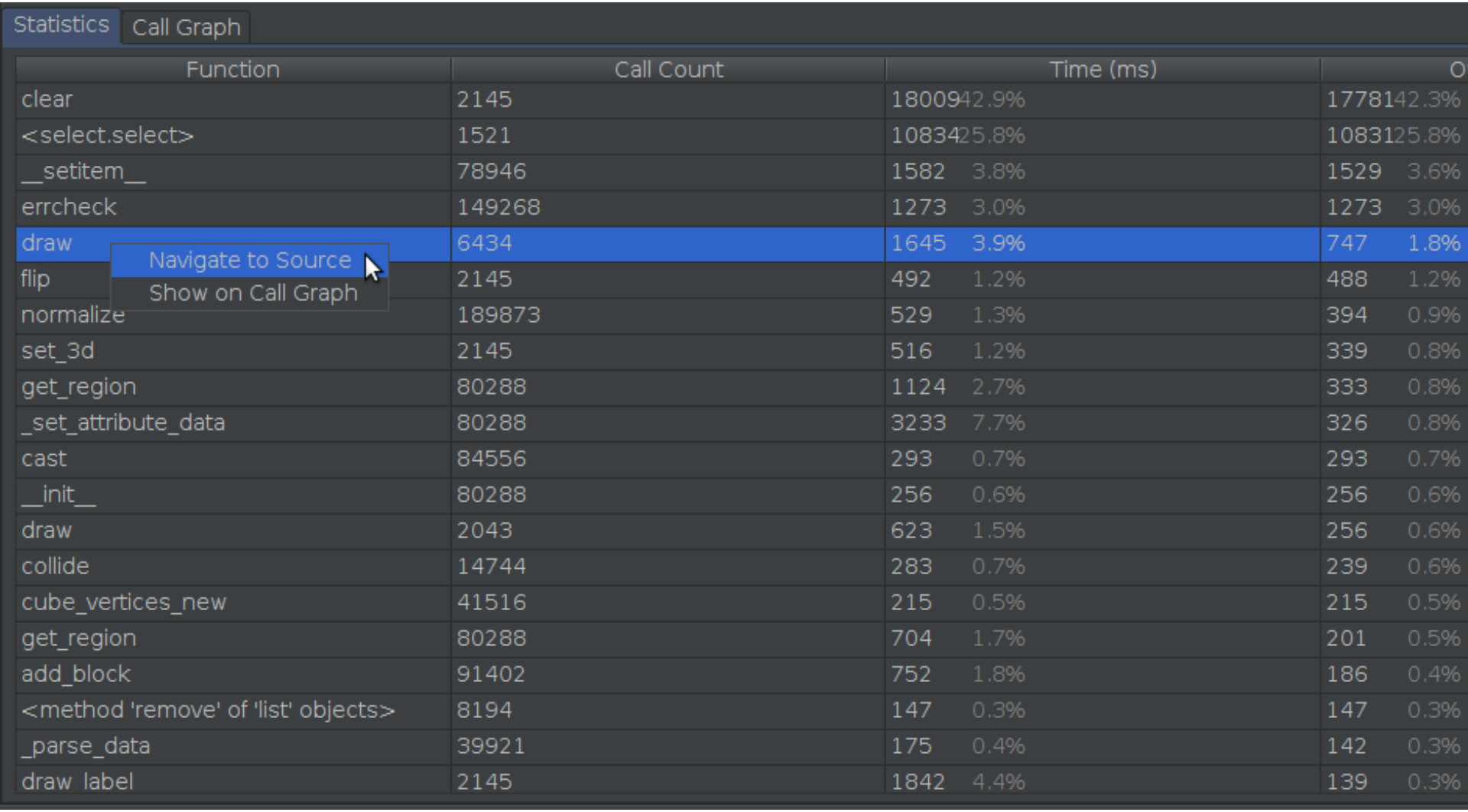
- Čas běhu programu – reálný, uživatelský, systémový
- Vlastní měření času v programu

V pythonu knihovna datetime

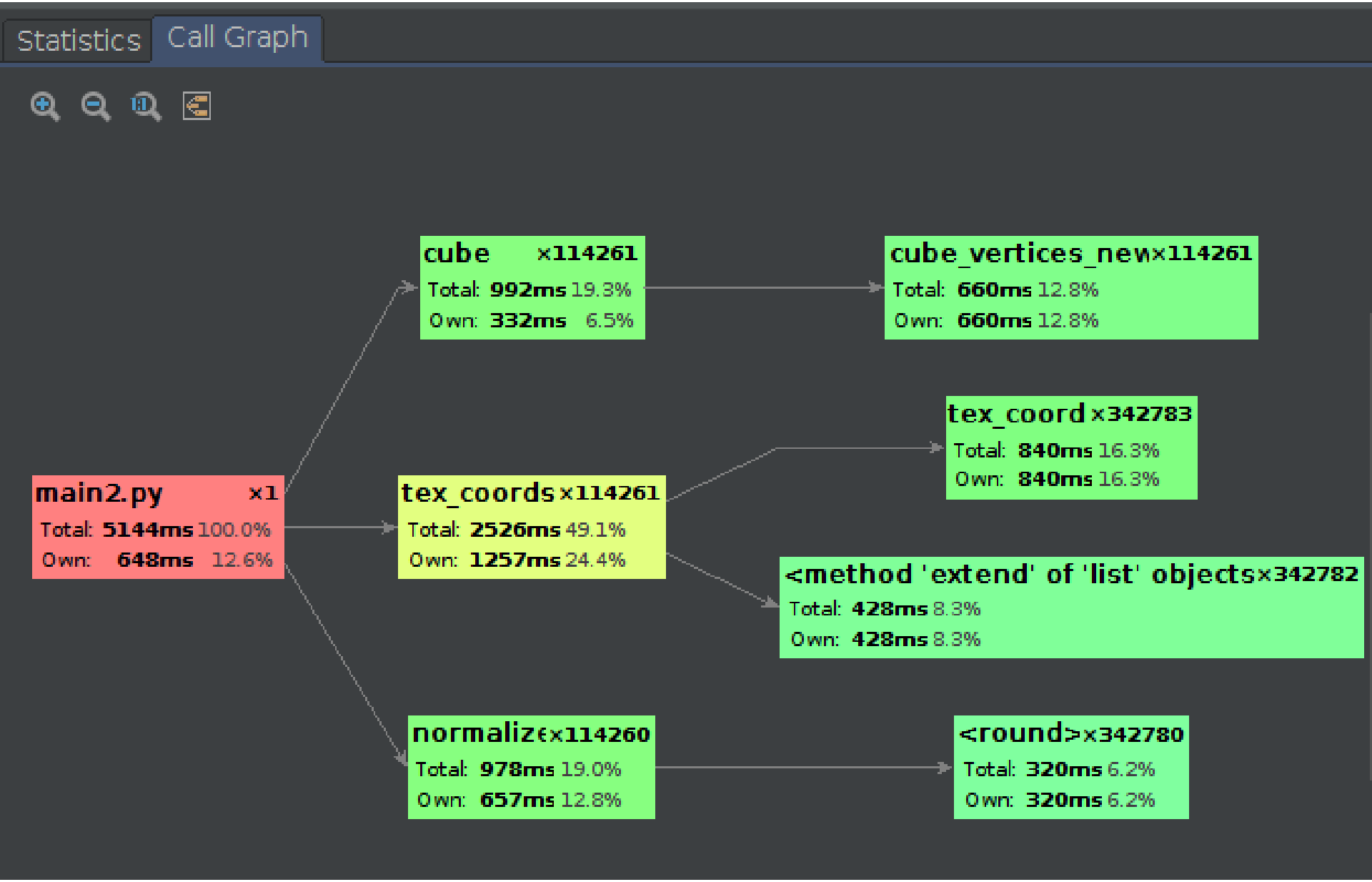
- Profiler – nástroj jak získat profil
  - Transparentnost
  - Efektivita
  - Přesnost

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

Profilování – plošný profil



Profilování – graf volání



Základy algoritmizace

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

Příště přehled programovacích jazyků