Asymptotická časová složitost

Typické třídy asymptotické časové složitosti algoritmů:
$O(1)$, $O(\log N)$, $O(N)$, $O(N.\log N)$, $O(N^2)$, $O(N^3)$, …, $O(2^N)$, $O(N!)$

Hledáme algoritmus s co nejmenší asymptotickou časovou složitostí, tedy co nejrychlejší pro velká *N*. Pro malá *N* může být třeba i pomalejší než nějaký jiný algoritmus, ale pro malá *N* to nevadí, doba výpočtu je vždy dostatečně krátká.

- polynomiálně omezený čas – obvykle zvládnutelné i pro velká *N*
- exponenciální čas – pro větší *N* časově nezvládnutelné,
  použitelné jen v případě, že vstupní data budou vždy malá

Exponenciální časová složitost

Příklad: Ke zpracování vstupních dat velikosti *N* algoritmus vykoná 2^N operací. Uvažujme rychlost počítače 10^9 operací za sekundu (řádově GHz – dnešní PC).

N	**doba výpočtu**
20	1 ms
30	1 s
40	17 min
50	11 dní
60	31 let
70	3.10^4 let
80	3.10^7 let
90	3.10^{10} let
100	3.10^{13} let

Pro vyšší hodnoty *N* (cca 50) je algoritmus prakticky nepoužitelný.
Nepomůže nám ani rychlejší počítač!

Složitost algoritmu v různých případech

Pro každá vstupní data velikosti *N* nemusí trvat výpočet stejně dlouho, rozdíl může být jen v konstantě, nebo i v asymptotické složitosti.

Časová složitost algoritmu v nejhorším případě
= maximální počet operací vykonaných algoritmem pro nějaká data velikosti *N*

→ nejčastěji používaná pro hodnocení algoritmů

Časová složitost algoritmu v nejlepším případě
= minimální počet operací vykonaných algoritmem pro nějaká data velikosti *N*

→ prakticky se nepoužívá

Časová složitost algoritmu v průměrném případě
= průměrný (očekávaný) počet operací vykonaných algoritmem pro data velikosti *N* (průměr pro všechna možná vstupní data velikosti *N*)

→ dobře charakterizuje kvalitu algoritmu, ale je obtížné odvodit ji

Složitost problému

- složitost nejlepšího algoritmu (z hlediska časové složitosti), kterým lze řešit daný problém

Nelze odvodit ze složitosti nějakého konkrétního algoritmu ani třeba všech běžně známých algoritmů, charakterizuje problém obecně (jaký nejlepší algoritmus řešící problém může v principu existovat)

$\rightarrow$ odvození bývá často obtížné, pro řadu problémů neznáme

Dokázat, že problém má složitost $\Theta(N^2)$ pro nejhorší případ, znamená ukázat, že

- existuje algoritmus, který vyřeší problém s asymptotickou časovou složitostí $O(N^2)$ v nejhorším případě

  to bývá snadné – předvedeme jeden zvolený konkrétní algoritmus

- každý algoritmus, který řeší uvedený problém, pracuje v čase $\Omega(N^2)$ v nejhorším případě

  to bývá obtížné – nemůžeme ukázat na jednom konkrétním algoritmu

Příklady:

1. Nalézt maximum z daných N čísel $\rightarrow$ časová složitost problému $\Theta(N)$

- existuje algoritmus s časovou složitostí $O(N)$ … triviální
- nemůže existovat algoritmus s lepší časovou složitostí

… maximem může být kterékoliv z N čísel, takže na každé se musí algoritmus podívat

2. Seřadit daných N čísel podle velikosti (problém třídění)

$\rightarrow$ časová složitost problému $\Theta(N. \log N)$

- existuje algoritmus s časovou složitostí $O(N. \log N)$ … např. heapsort
- nemůže existovat algoritmus s lepší časovou složitostí

… oboje si ukážeme později

Základní algoritmy s čísly

Rozklad čísla na cifry

Úloha:
Spočítejte ciferný součet zadaného kladného celého čísla.

Princip řešení:
zbytek čísla po dělení 10 → poslední cifra
celočíselné vydělení čísla 10 → odstranění poslední cifry

Řešení jako funkce v Pythonu:

```
def cifsoucet(x):
    """ciferný součet kladného celého čísla"""
    y = 0
    while x != 0:
        y += x % 10
        x //= 10
    return y
```

Jiné řešení – převodem přes znakový řetězec:

```
def cifsoucet(x):
    """ciferný součet kladného celého čísla"""
    return sum([int(c) for c in str(x)])
```

Test prvočíselnosti

Úloha: Určete, zda je dané číslo *N* prvočíslem.

Postupy řešení:

1. zkusit všechny dělitele od 2 do *N*-1
   $\rightarrow$ časová složitost **O(*N*)** – cca *N* testů

Poznámka: Popsaný algoritmus má ve skutečnosti exponenciální časovou složitost vzhledem k délce vstupu, pokud za délku vstupu považujeme $[\log_2 N]+1$, což je délka bitového zápisu čísla *N*.

2. stačí zkoušet všechny dělitele od 2 do $N/2$ (větší dělitel neexistuje)
   $\rightarrow$ časová složitost opět **O(*N*)** – cca $N/2$ testů, tedy o něco lepší

3. stačí zkoušet všechny dělitele od 2 do $\sqrt{N}$
   (má-li N vlastního dělitele, musí mít i dělitele z tohoto intervalu)
   $\rightarrow$ časová složitost **O($\sqrt{N}$)** – cca $\sqrt{N}$ testů, asymptoticky lepší

4. stačí zkoušet dělitele od 2 do $\sqrt{N}$, do nalezení prvního
   → asymptotická časová složitost opět **O($\sqrt{N}$),**
   většinou se ale vykoná o dost méně testů

   - v nejhorším případě se vykoná plných $\sqrt{N}$ testů
     → časová složitost v nejhorším případě **O($\sqrt{N}$)**
   - v nejlepším případě stačí pouze jeden test

5. stačí zkusit číslo 2 (jediné sudé prvočíslo)
   a pak jenom liché dělitele lichých čísel
   → další úspora práce

```
from math import sqrt

def prvocislo(n):
    """
      test prvočíselnosti, předpoklad n > 1
    """
    # stačí prověřit dělitele do odmocniny z n
    for d in range(2, int(sqrt(n))+1):
        if n % d == 0:       # pokud d | n
            return False     # n není prvočíslo
    return True
```

```
def prvocislo(n):
    """
      test prvočíselnosti, předpoklad n > 1
    """
    if n % 2 == 0:
        return n == 2        # jediné sudé prvočíslo: 2
    d = 3
    while d * d <= n:        # místo volání sqrt(n)
        if n % d == 0:
            return False
        d += 2
    return True
```

Eratosthenovo síto

řecký matematik Eratosthenes z Kyrény, cca 200 př.n.l.

Úloha: Určete všechna prvočísla od 2 do *N.*

Princip řešení:
- v řadě čísel od 2 do *N* postupně vyškrtáváme
  všechny násobky jednotlivých prvočísel
- co nakonec nebude vyškrtnuto, je prvočíslo

Programová realizace:
- síto v programu reprezentujeme polem prvků typu boolean,
- index pole určuje číslo od 2 do *N*,
- hodnota prvku síta říká, zda je prvočíslem.

```
def eratosth(n):
    """Eratosthenovo síto"""

    sito = [False, False] + [True] * (n-1)
    prvocisla = []

    for i in range(n + 1):
        if sito[i]:
            prvocisla.append(i)
            j = i * i  # stačí začít s násobky od kvadrátu "i"
            while j <= n:
                sito[j] = False
                j = j + i

    return prvocisla
```

Časová složitost:

$$O(N/2 + N/3 + N/5 + \ldots) = O(N.\log(\log N))$$

Dlouhá čísla

Chceme počítat například s kladnými celými čísly s desítkami nebo stovkami cifer (podobně pro čísla se znaménkem, čísla desetinná apod.).

Python to umí sám, většina programovacích jazyků ale nikoliv.

Reprezentace:

- číslo uložíme po cifrách – seznam cifer (pole cifer)
- pro snadnější výpočty použijeme seznam čísel, nikoliv znaků
- nutno zvolit pořadí cifer odpředu nebo odzadu
  (možné je oboje, ale potom důsledně dodržovat v celém programu!)

Operace:

- po cifrách – jako sčítání, odčítání, násobení či dělení víceciferných čísel na základní škole
- počítání modulo 10, přenosy do vyšších řádů

Modifikace:
číslo uložíme po skupinách cifer do prvků typu int,
→ úspora paměti, rychlejší výpočet
(počítání modulo 100, nebo 1000, atd.)

Desetinné číslo:
stačí doplnit evidenci polohy desetinné čárky
- speciální hodnota uložená v jednom prvku seznamu
  (méně šikovné)
- proměnná s indexem nulového řádu
- použít dvě pole (celá a desetinná část čísla)

Příklad:

Součet dvou kladných celých čísel s mnoha ciframi

Vstup: a, b – seznamy cifer sčítaných čísel
a[0] = cifra v řádu jednotek
Výstup: c – výsledný seznam cifer součtu čísel a + b

- nejprve sčítání s přenosem v rozsahu cifer kratšího čísla
- dále musíme podobně (s přenosem) ošetřit „přečnívající“ část delšího čísla
- nakonec ještě přidat případný poslední nenulový přenos

```
if len(a) < len(b):
    a, b = b, a          #číslo "a" je nyní delší

prenos = 0
c = []
for i in range(len(b)):
    x = a[i] + b[i] + prenos
    c.append(x%10)
    prenos = x // 10

for i in range(len(b), len(a)):
    x = a[i] + prenos
    c.append(x%10)
    prenos = x // 10

if prenos > 0:
    c.append(prenos)
```

Vyhodnocení polynomu v bodě

$a(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} \ldots + a_1 x + a_0$

n – stupeň polynomu

$a_0, \ldots, a_n$ – koeficienty (reálné konstanty)

x – proměnná, za niž dosazujeme různé hodnoty

Přímý výpočet podle uvedeného předpisu

počet násobení: $n + (n\text{-}1) + (n\text{-}2) + \ldots + 1 = n.(n+1)/2$

počet sčítání: n

časová složitost algoritmu: $\Theta(n^2)$

Hornerovo schéma *(Horner 1819, ale známo již dříve)*

$a(x) = (\ldots((a_n x + a_{n-1}).x + a_{n-2}).x + \ldots + a_1).x + a_0$

počet násobení: n

počet sčítání: n

časová složitost algoritmu: $\Theta(n)$

```
def horner(a,x):
    """
    výpočet hodnoty polynomu Hornerovým schématem
    a: seznam s koeficienty polynomu – a[i] * x^i
    x: bod z definičního oboru
    vrátí: hodnotu polynomu v bodě x
    """
    h = 0
    for i in range(len(a)-1, -1, -1):
        h = h * x + a[i]
    return h
```

Příklad použití algoritmu:

vstup čísla po znacích
- co přibližně dělá procedura „read“ v Pascalu, funkce „scanf“ v C

konverze číselného stringu na integer
- co dělá funkce int() v příkazu a = int(input())

Operace s polynomy

$a(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} \ldots + a_1 x + a_0$
$b(x) = b_m x^m + b_{m-1} x^{m-1} + b_{m-2} x^{m-2} \ldots + b_1 x + b_0$

- součet, součin, …

```
a = [2, -5, 0, 4, 6]      # 6x^4 + 4x^3 - 5x + 2
b = [11, 0, -2]           # -2x^2 + 11

def soucet(a, b):
    c = []
    if len(a) < len(b):
        a, b = b,a
    for i in range(len(b)):
        c.append(a[i]+b[i])
    for i in range(len(b), len(a)):
        c.append(a[i])
    while len(c) > 1 and c[-1] == 0:
        del c[-1]
    return c

print(soucet(a, b))
```

```
a = [2, -5, 0, 4, 6]      # 6x^4 + 4x^3 - 5x + 2
b = [11, 0, -2]           # -2x^2 + 11

def soucin(a, b):
    c = [0] * (len(a)+len(b)-1)
    for i in range(len(a)):
        for j in range(len(b)):
            c[i+j] += a[i] * b[j]
    return c

print(soucin(a, b))
```

Číselné soustavy

- převod **z dvojkové soustavy** na číselnou hodnotu
- algoritmus: Hornerovo schéma

$$110010 \approx 1.2^5 + 1.2^4 + 0.2^3 + 0.2^2 + 1.2^1 + 0.2^0 =$$
$$= ((((1.2 + 1).2 + 0).2 + 0).2 + 1).2 + 0 = 50$$

- převod **z šestnáctkové soustavy** na číselnou hodnotu

$$A1F \approx A.16^2 + 1.16^1 + F.16^0 =$$
$$= 10.16^2 + 1.16^1 + 15.16^0 =$$
$$= (10.16 + 1).16 + 15 = 2591$$

```
def bin_int(s):
    """
      převod binárního zápisu čísla (string s)
      na číselnou hodnotu
    """
    n = 0
    for i in range(len(s)):
        n = n * 2 + int(s[i])
    return n
```

```
def hex_int(s):
    """
      převod hexadecimálního zápisu čísla (string s)
      na číselnou hodnotu
    """
    cifry={'A':10, 'B':11, 'C':12, 'D':13, 'E':14, 'F':15,
           'a':10, 'b':11, 'c':12, 'd':13, 'e':14, 'f':15}
    n = 0
    for i in range(len(s)):
        if s[i] in "0123456789":
            n = n * 16 + int(s[i])
        else:
            n = n * 16 + cifry[s[i]]
    return n
```

Jiné řešení
– kratší zápis, ale jsou povolena pouze velká písmena A, B, C, D, E, F:

```
def hex_int(s):
    """
      převod hexadecimálního zápisu čísla (string s)
      na číselnou hodnotu
    """
    cifry = "0123456789ABCDEF"
    n = 0
    for i in range(len(s)):
        n = n * 16 + cifry.index(s[i])
    return n
```

$110010 \approx 1.2^5 + 1.2^4 + 0.2^3 + 0.2^2 + 1.2^1 + 0.2^0 =$

$= ((((1.2 + 1).2 + 0).2 + 0).2 + 1).2 + 0 = 50$

- převod číselné hodnoty **do dvojkové soustavy**
- algoritmus: Hornerovo schéma využité v opačném směru
  posloupnost zbytků při celočíselném dělení dvěma
  tvoří **odzadu** dvojkový zápis čísla
  → připojování dvojkových cifer do stringu **zleva**

50 : 2 = 25, zb. 0
25 : 2 = 12, zb. 1
12 : 2 = 6, zb. 0
6 : 2 = 3, zb. 0
3 : 2 = 1, zb. 1
1 : 2 = 0, zb. 1

```
def int_bin(n):
    """
      převod čísla do dvojkové soustavy
      - obrácené Hornerovo schéma
    """
    s = ""
    while n > 0:
        s = str(n % 2) + s
        n //= 2
    return s
```

Pozor na časovou složitost – při této implementaci bude kvadratická vzhledem k délce vytvářeného binárního zápisu (kvůli posouvání stringu `s` při jeho prodlužování zleva).
Řešení s lineární složitostí: prodlužovat string zprava a nakonec celý string s binárním zápisem otočit.

```
def int_hex(n):
    """
      převod čísla do šestnáctkové soustavy
      - obrácené Hornerovo schéma
    """
    s = ""
    cifry = "0123456789ABCDEF"
    while n > 0:
        s = cifry[n % 16] + s
        n //= 16
    return s
```

Rychlé umocňování

aplikace dvojkové soustavy

Úloha: spočítat hodnotu x^n, kde

- *n* je velké kladné celé číslo
- x může být reálné číslo (nebo třeba také matice)

Řešení:

1. přímočaře v čase Θ(*n*):

```
def mocnina1(x, n):
    """výpočet x**n lineárně"""
    v = 1
    for i in range(n):
        v *= x
    return v
```

2. rychleji v čase $\Theta(\log n)$:

postupně počítáme hodnoty x, x^2, x^4, x^8, …

a vhodné z nich násobíme do výsledku

Které hodnoty x^k jsou ty vhodné?

Pozorování: $x^{25} = x^{16} \cdot x^8 \cdot x^1$, neboť 25 = 16 + 8 + 1

- to je jednoznačný rozklad čísla *n* na součet mocnin dvojky
- jsou to ty mocniny dvojky, kde je jednička v binárním zápisu čísla *n*

Postup je tedy podobný, jak převod čísla do dvojkové soustavy.

```
def mocnina2(x, n):
    """výpočet x**n rychleji"""
    v = 1
    while n > 0:
        if n % 2 == 1:
            v *= x
        x *= x
        n //= 2
    return v
```

Časová složitost O(log *n*) – počet opakování while-cyklu.