NPRG062 Algoritmizace 2/1 Z+Zk 4 kr.

Pavel Töpfer
Katedra softwaru a výuky informatiky MFF UK
MFF Malostranské nám., 4. patro, pracovna 404

pavel.topfer@mff.cuni.cz
https://ksvi.mff.cuni.cz/~topfer

Algoritmizace
- metody řešení úloh na počítači
- algoritmy, datové struktury, programovací techniky

Správnost algoritmu
- konečnost
- parciální korektnost (správné výsledky, když výpočet skončí)

Efektivita algoritmu (složitost algoritmu)
- časová (počet vykonaných operací)
- prostorová (paměť potřebná na uložení dat)

Učivo

- algoritmus, časová a paměťová složitost
- dělitelnost čísel, Eukleidův algoritmus
- test prvočíselnosti, Eratosthenovo síto
- rozklad čísla na cifry
- aritmetika s vyšší přesností („dlouhá čísla“)
- Hornerovo schéma, poziční číselné soustavy
- algoritmy vyhledávání v poli (sekvenční, binární, zarážka)
- třídění čísel v poli - přímé metody, heapsort, mergesort, quicksort
- složitost problému třídění
- přihrádkové třídění
- reprezentace dat v paměti
- zásobník, fronta, slovník, halda
- spojový seznam

Učivo (pokračování)

- rekurze – princip, příklady, efektivita
- binární a obecný strom – reprezentace, průchod, použití
- binární vyhledávací strom, princip vyvažování
- notace aritmetického výrazu – vyhodnocení, převody
- hešovací tabulka
- prohledávání stavového prostoru do hloubky a do šířky
- metody zrychlení backtrackingu – ořezávání, heuristiky
- programování her, algoritmus minimaxu
- metoda Rozděl a panuj
- dynamické programování
- reprezentace grafu
- prohledávání grafu, základní grafové algoritmy

Studijní zdroje

Prezentace z přednášek
- aktuálně vždy po přednášce na webu přednášejícího
- často rozšířené a doplněné oproti přednášce
- ukázkové programy

Pavel Töpfer: Algoritmy a programovací techniky
Prometheus Praha 1995, 2. vydání 2007
tištěná v knihovnách, nyní k zakoupení pouze jako e-kniha
https://www.prometheus-eknihy.cz/

Studijní zdroje (pokračování)

Programátorské kuchařky KSP
krátké studijní texty k jednotlivým tématům algoritmizace a programování
http://ksp.mff.cuni.cz/kucharky/

Martin Mareš, Tomáš Valla: Průvodce labyrintem algoritmů
CZ.NIC Praha 2017
text pdf zdarma ke stažení
http://pruvodce.ucw.cz/
https://knihy.nic.cz/

Zápočet

- uděluje cvičící
- domácí úkoly
- případné další požadavky cvičícího (práce na cvičeních)

Zkouška

- společné požadavky a zkušební termíny pro obě paralelky přednášky
- přihlašování prostřednictvím SISu
- k účasti na zkoušce je nutné ***předchozí získání započtu***
- formu zkoušky včas upřesníme podle aktuální epidemiologické situace

Algoritmus

„Konečná posloupnost elementárních příkazů, jejichž provádění umožňuje pro každá přípustná vstupní data mechanickým způsobem získat po konečném počtu kroků příslušná výstupní data.“

Vlastnosti:

- konečnost
- hromadnost
- resultativnost
- jednoznačnost
- determinismus

Formální modely algoritmu

rekurzivní funkce (Kurt Gödel, 1934)
Turingův stroj (Alan Turing, 1936)
lambda kalkul (Alonzo Church, 1941)
RAM počítač

Popis a zápis algoritmu

slovní popis v přirozeném jazyce
pseudokód
program (zjednodušené konstrukce programovacího jazyka)

Největší společný dělitel

X, Y – dvě kladná celá čísla → určit největší společný dělitel NSD(X, Y)

Algoritmy:

1. NSD(X, Y) = největší z celých čísel od 1 do *min(X*, Y),
které je dělitelem obou čísel X a Y

– postupně zkoušet, nejlépe v pořadí od *min(X*, Y) dolů k 1,
do nalezení prvního takového společného dělitele

2. Určit prvočíselné rozklady čísel X a Y
– jejich maximální společná část určuje NSD(X, Y)

Např. 396 = 2.2.3.3.11, 324 = 2.2.3.3.3.3

$\rightarrow$ NSD(396, 324) = 2.2.3.3 = 36

3. **Eukleidův algoritmus**

Eukleidés: Základy (řecky Stoicheia, 13 knih), cca 300 př.n.l.

Idea: když X < Y NSD(X, Y) = NSD(X, Y-X)
když X > Y NSD(X, Y) = NSD(X-Y ,Y)
když X = Y NSD(X, Y) = X

Příklad:

NSD(396, 324) =
NSD(72, 324) =
NSD(72, 252) =
NSD(72, 180) =
NSD(72, 108) =
NSD(72, 36) =
NSD(36, 36) = 36

Algoritmus (pro kladná celá čísla X, Y):

dokud X ≠ Y dělej
   od většího z čísel X, Y odečti menší z čísel X, Y

```
while x != y:
    if x > y:
        x -= y
    else:
        y -= x
print(x)
```

Možnost urychlení (pro některé vstupní hodnoty):
místo odčítání použít zbytek po celočíselném dělení

```
while y > 0:
    z = x % y
    x = y
    y = z
print(x)
```

Program funguje i v případě X < Y, jenom vykoná o jednu iteraci více a při první iteraci se hodnoty X, Y navzájem prohodí.

```
while y > 0:
    x, y = y, x % y
print(x)
```

Správnost Eukleidova algoritmu

1. Konečnost

= *výpočet pro jakákoliv vstupní data skončí*

- na začátku výpočtu i stále v jeho průběhu je X > 0, Y > 0

- v každém kroku výpočtu se hodnota X+Y sníží alespoň o 1

$\rightarrow$ nejpozději po X+Y krocích výpočet skončí, je tedy konečný

2. Parciální (částečná) správnost
= když výpočet skončí, vydá správný výsledek

- pro X = Y zjevně platí NSD(X, Y) = X

- ukážeme, že pro X > Y platí NSD(X, Y) = NSD(X-Y, Y)

Důkaz sporem:
Nechť N = NSD(X, Y), tedy N dělí X a zároveň N dělí Y.
Proto také N dělí X-Y a je tedy N společným dělitelem X-Y a Y.
Pokud by neplatilo, že N = NSD(X-Y, Y), musí existovat A > 1 tak, že N.A = NSD(X-Y, Y).
Tedy N.A dělí X-Y i Y, takže N.A dělí i jejich součet, což je X.
Jelikož N.A dělí Y a zároveň N.A dělí X, je N.A společným dělitelem čísel X, Y, což je spor s předpokladem, že N = NSD(X, Y).
Proto skutečně N = NSD(X-Y, Y).

Efektivita algoritmu (složitost algoritmu)

- **časová**
  počet vykonaných operací *(kterých?)*
  → rychlost výpočtu programu
- **prostorová (paměťová)**
  velikost datových struktur využívaných algoritmem
  → paměť potřebná na uložení dat při výpočtu programu

Kritéria pro hodnocení kvality algoritmu a programu
(příp. praktické použitelnosti programu).

Obě kritéria mohou mířit proti sobě (nelze najednou optimalizovat paměť i čas), musíme zvolit, čemu dáme přednost (dnes obvykle čas).

„výměna času za paměť“

Funkce vyjadřující počet vykonaných operací (resp. velikost potřebné paměti) v závislosti na velikosti vstupních dat. Jako velikost vstupních dat obvykle stačí uvažovat např. počet zpracovaných čísel, nikoliv konkrétní hodnoty (jedná-li se o hodnoty „standardní“ velikosti). Obecně by se musela uvažovat délka zápisu vstupních dat v bitech.

Funkce časové a prostorové složitosti jsou většinou ***rostoucí*** (nad většími daty bývá výpočet delší).

Jaké operace započítat:
- elementární, vyžadující konstantní čas
- aritmetické, logické, přiřazení
- typické pro řešený problém (převažující)

Kterou paměť započítat:
- do prostorové složitosti nepočítáme paměť potřebnou na uložení vstupních dat, pokud z ní data pouze čteme (neměníme její obsah)

Přesné vyjádření funkce časové složitosti (např. $3.N^2 + 2.N - 4$) je jednak obtížné, jednak většinou zbytečné. Podstatná je řádová rychlost růstu této funkce pro rostoucí N.

Zanedbáme pomaleji rostoucí členy a konstanty
→ **asymptotická časová složitost** $O(N^2)$.

Symbol „velké O“
f, g: $\mathbf{N} \rightarrow \mathbf{R}^+$
$f \in O(g) \Leftrightarrow \exists\, c > 0 \;\; \exists\, n_0 \;\; \forall\, n > n_0 : \;\; 0 \leq f(n) \leq c.g(n)$

tzn. funkce f se dá **shora** odhadnout funkcí g
(až na multiplikativní konstantu a pro dostatečně velká n)

Opačný odhad **zdola**: $f \in \Omega(g)$
Přesný odhad: $f \in \Theta(g) \Leftrightarrow f \in O(g) \;\&\; f \in \Omega(g)$

Poznámka:

Funkce $3.N^2 + 2.N - 4$ patří nejen do třídy $O(N^2)$, ale podle definice také třeba do třídy $O(N^5)$. Odhad složitosti $O(N^5)$ je zde sice správný, ale zbytečně hrubý a nepřesný, vždy se snažíme o co nejlepší (nejnižší) horní odhad složitosti.

Proto je pro nás cennější (ale také obtížnější) odvodit těsný odhad asymptotické časové složitosti algoritmu $\Theta(N^2)$ než jenom horní odhad $O(N^2)$.

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 horší 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!