Časová zložitosť

Laický pohľad skutočne môže naznačovať, že efektívne algoritmy vôbec nepotrebujeme. Veď predsa každý rok sa výrobcovia počítačov predbiehajú v tom, kto vyrobí ešte výkonnejší procesor, ešte rýchlejšiu pamäť, ... Tento trend je dokonca natoľko výrazný, že má aj svoje meno: Moorov zákon. Pomenovaný je podľa jedného zo zakladateľov Intelu, Gordona E. Moora, ktorý v roku 1965 predpovedal, že každé dva roky sa zdvojnásobí počet tranzistorov, ktoré sa podarí umiestniť na integrovaný obvod danej veľkosti. V súčasnosti sa názov „Moorov zákon“ používa všeobecnejšie a asi najznámejšia jeho formulácia znie: „približne každých 18 mesiacov sa zdvojnásobí výpočtová sila počítačov bežne dostupných v obchodoch“.

Uvedieme jeden príklad: pred 15 rokmi (27. marca 1995) prišiel Intel na trh s najnovším modelom procesoru Pentium s taktovacou rýchlosťou 120 MHz. V roku 2010 už boli bežne dostupné procesory, ktoré mali taktovaciu rýchlosť vyše 3 GHz (vyše 25-násobný nárast), štyri nezávislé jadrá, dve úrovne vyrovnávacej pamäte a iné vylepšenia, ktoré dokopy skutočne spôsobili, že počítače v roku 2010 boli približne tisíckrát výkonnejšie ako tie o 15 rokov staršie.

Keď teda máme k dispozícii takto výkonné počítače (a nádej, že v budúcnosti budú ešte lepšie), potrebujeme teda na niečo efektívne algoritmy? Áno, potrebujeme, ba dokonca viac ako kedykoľvek predtým. Spolu s rastom výpočtovej sily bežných počítačov rastie totiž aj objem dát, ktoré naša spoločnosť potrebuje denne spracovať.

Opäť si porovnajme roky 1995 a 2010. V roku 1995 sa kapacita bežne dostupných pevných diskov blížila k 1 GB a dáta sme bežne prenášali na 3.5-palcových disketách. V roku 2010 už boli bežné pevné disky s kapacitou výrazne presahujúcou 1 TB. (Opäť ide o vyše tisícnásobný nárast.)

Podľa The Official Google Blog prvý index webstránok, ktorý algoritmy vyhľadávača Google zostrojili v roku 1998, obsahoval 26 miliónov webstránok. Okolo roku 2000 tento počet dosiahol miliardu a v lete 2008 už Google poznal dokonca bilión (10^{12}) rôznych webstránok. Množstvo webstránok teda len za 10 rokov narástlo takmer 50 000-krát – omnoho výraznejšie ako výpočtová sila počítačov! A to nehovoríme o tom, že dnešné webstránky sú často plné multimediálneho obsahu, a teda sú neporovnateľne väčšie ako tie z roku 1998.

A nielen samotné množstvo spracúvaných dát nás núti hľadať čo najefektívnejšie algoritmy. Druhým dôvodom je obtiažnosť problémov, ktoré sa snažíme pomocou počítačov riešiť. Častokrát je počet všetkých možných riešení problému natoľko obrovský, že keď hľadáme to najlepšie z nich, nemôžeme si ani zďaleka dovoliť vyskúšať ich všetky.

Žartík

Skôr, než opustíme Moorov zákon a pozrieme sa na ťažké problémy, si ešte uveďme jedno vtipné pozorovanie.

Predpokladajme, že máme *fakt veľký* výpočtový problém, o ktorom odhadujeme, že ho náš počítač vyrieši za sedem rokov. Čo máme robiť, ak potrebujeme výsledok čo najskôr? Samozrejme, že čo najskôr spustiť náš program, však že?

Nuž, nie. Ak dôverujeme Moorovmu zákonu, existujú aj lepšie riešenia. Najlepšie vychádza nasledovné: treba najskôr počkať približne 2.54 roka, vtedy kúpiť nový počítač a spustiť program na tom. Keďže ten nový počítač bude niekoľkokrát rýchlejší od súčasného, výsledok spočíta za niečo vyše dvoch rokov. Od súčasnosti budeme teda na výsledok namiesto siedmich rokov čakať len približne 4.7 roka. Dobré, nie? :)

(Viete vypočítať ako vyzerá riešenie vo všeobecnosti? Kedy sa oplatí čakať a ako dlho?)

1.1 Šachové programy

Dobre si to môžeme ilustrovať na príklade programov, ktoré hrajú šach. Súčasné šachové programy sú neuveriteľne komplikované (obsahujú napríklad samostatné knižnice rôznych otvorení a koncoviek), ale všetky majú spoločné jadro: Keď sa takýto program rozhoduje, aký ťah má zahrať, spraví to tak, že začne prezerať možné priebehy nasledujúcich ťahov oboch hráčov. Keď mu vyprší čas, vyberie si ten ťah, ktorý na základe možných vývojov partie vyhodnotí ako najlepší.

Koľko toho stihne takýto program prezrieť? Nech trebárs v každej pozícii existuje v priemere 10 rozumných možností, ako potiahnuť. Ak sa budeme pozerať na nasledujúce dva polťahy, je $10 \times 10 = 100$ možností, ako budú vyzerať – na každý z ťahov počítača môže jeho protihráč zareagovať 10 spôsobmi. Ak by sme chceli prezrieť všetky možné priebehy nasledujúcich troch polťahov, už ich počet narastie na 10^3 a tak ďalej.

Počet priebehov partie teda rastie exponenciálne. Už napríklad pri deviatich polťahoch dostávame odhadom miliardu možných priebehov. Tento počet je tak na hranici toho, čo dnešný počítač zvládne prezrieť za jednu minútu. Zjednodušene teda môžeme povedať: Ak by sme nášmu šachovému programu dovolili minútu počítať, stihol by vyhodnotiť, čo všetko sa môže stať v nasledujúcich deviatich polťahoch.

Ako veľmi nám pri hraní šachu pomôže, ak si kúpime nový počítač? Odpoveď je jednoduchá: na to, aby sa nový počítač stihol za minútu pozrieť čo i len o jediný polťah ďalej, musí byť 10-krát rýchlejší od toho, ktorý máme teraz. Ak sa aj v budúcnosti bude výpočtová sila počítačov správať podľa Moorovho zákona, dočkáme sa takého počítača až približne o 5 rokov.

S podobnou situáciou sa často stretneme pri spracúvaní dát v praxi: Ak nepoznáme efektívny algoritmus, ktorý by náš problém riešil, samotný nárast výpočtovej sily počítačov nás nemá ako zachrániť. A práve tu prichádza k slovu návrh efektívnych algoritmov: ak v praxi narazíme na ťažký problém, potrebujeme ho vedieť analyzovať a odhaliť čo najlepší spôsob, ako ho algoritmicky riešiť.

1.2 Iné ťažké problémy

Problém batoha (https://en.wikipedia.org/wiki/Knapsack_problem) je optimalizačný problém, v ktorom chceme spomedzi daných n vecí vybrať podmnožinu ktorú zvládneme naraz odniesť a ktorá má v súčte čo najväčšiu cenu. Tento problém vieme riešiť hrubou silou: vyskúšame postupne všetkých 2^n podmnožín vecí a o každej si spočítame, koľko dokopy váži a akú má dokopy cenu.

Problém obchodného cestujúceho (https://en.wikipedia.org/wiki/Travelling_salesman_problem) je optimalizačný problém, pri ktorom chceme nájsť najlacnejší spôsob ako postupne navštíviť všetky mestá v danej krajine. Aj tento problém vieme riešiť hrubou silou: tentoraz treba vyskúšať všetkých $n!$ poradí, v ktorých mestá navštíviť.

Tieto riešenia hrubou silou sú však v praxi použiteľné len pre veľmi malé hodnoty n. Aj takéto problémy nás preto motivujú hľadať efektívnejšie algoritmy.

Napríklad tým, že kúpim $10\times$ rýchlejší počítač, zväčším pri probléme batoha n, ktoré zvládnem spracovať hrubou silou, približne o 3: ak som na svojom starom počítači vedel za hodinu vyriešiť vstup v ktorom $n = 35$, na novom počítači to bude $n = 38$. Toto pozorovanie platí pre ľubovoľný problém, na ktorého riešenie potrebujeme exponenciálny počet krokov: ak *niekoľkokrát* zväčším rýchlosť počítača, len *o niekoľko* narastie veľkosť dát, ktoré zvládnem spracovať. (Respektíve naopak: ak *o niekoľko* narasie veľkosť dát, tak *niekoľkokrát* narastie potrebný počet krokov výpočtu.)

Ak však vymyslím lepší algoritmus, ktorý bude napríklad namiesto 2^n možností skúšať len $2^{n/2}$ možností, zlepším tým veľkosť zvládnuteľných dát až na $n = 70$. Kde nákup techniky zlyhal, lepší algoritmus víťazí.

V oboch vyššie uvedených úlohách už skutočne poznáme šikovnejšie algoritmy, ktoré riešenie nájdu efektívnejšie ako hrubá sila. Ešte stále však všetky tieto algoritmy potrebujú spraviť exponenciálne veľa krokov v závislosti od n. A predpokladá sa, že žiadne principiálne lepšie riešenia týchto problémov neexistujú – o oboch totiž vieme dokázať, že sú tzv. NP-ťažké.

1.3 Ako neporovnávať algoritmy

Skôr než sa dostaneme k samotnému návrhu algoritmov, potrebujeme spraviť jednu dôležitú odbočku. Keď totiž vymyslíme k nejakému problému viacero rôznych algoritmov, budeme potrebovať vyhodnotiť, ktorý z nich je lepší. Ako ale algoritmy porovnávať?

Ako prvý asi každému napadne priamočiary prístup: Keď máme vymyslené dva algoritmy, skúsime oba naprogramovať a spustiť. Ktorý skôr skončí, ten je lepší.

Tento prístup je však veľmi nepraktický, a to hneď z viacerých dôvodov:

- Vyžaduje presne to, čomu sa chceme vyhnúť. My nechceme programovať všetky riešenia, ktoré nám napadnú, práve naopak. Chceme vybrať to najlepšie z nich a naprogramovať len to.
- Je nepresný. Rýchlosť behu programu závisí od mnohých faktorov, ako napríklad momentálna záťaž procesora inými úlohami, množstvo voľnej pamäte, architektúra procesora a podobne.

- Nemusí byť použiteľný. Môže sa stať, že dáta sú natoľko veľké, že takéto praktické testy by trvali neúnosne dlho. (Obzvlášť ak ani jeden z našich algoritmov nie je dostatočne dobrý.)
- Je nedostatočný. Tým, že programy spustíme pre nejaké konkrétne vstupné dáta, sa dozvieme len to, ako sa správajú pre tento konkrétny vstup. Čo nám ale zaručí, že aj hocijaké iné dáta zvládne ten program spracovať rovnako rýchlo?

Pri analýze algoritmov sa preto bežne používa iný prístup: Nebudeme používať žiaden konkrétny počítač, budeme sa na vykonávanie algoritmu dívať ako na postupnosť logických krokov. Čím menej krokov potrebujeme spraviť pri vykonávaní daného algoritmu, tým lepší bude.

1.4 Počítanie počtu krokov

Spočítať, koľko presne krokov spraví daný algoritmus (pre dané vstupné dáta), môže byť už aj pre veľmi jednoduché algoritmy veľmi zložitá úloha. Bude preto vhodné začať jednoduchou návodnou úlohou.

> *Na obchodnom stretnutí sa stretlo 20 podnikateľov. Každý z nich si podal ruku so všetkými ostatnými. Koľko podaní rúk sa dokopy uskutočnilo?*

Riešenie tejto úlohy je jednoduché: dvojíc podnikateľov je $\binom{20}{2} = \frac{20.19}{2} = 190$ a každá dvojica si raz podala ruku, preto podaní rúk bolo práve 190.

Nasleduje druhá návodná úloha:

> *Koľko hviezdičiek vypíše nasledujúca časť programu, ak premenná* `n` *obsahuje hodnotu 20?*
>
> ```
> for i := 1 to n do
>   for j := i + 1 to n do
>     write('*');
> ```
>
> *Keby sa hodnota premennej* `n` *načítavala z klávesnice, ako by počet hviezdičiek závisel od hodnoty, ktorú používateľ zadá?*

Toto je vlastne tá istá úloha, len teraz je zamaskovaná v inom šate. Aby sme lepšie videli, čo sa pri vykonávaní programu deje, upravme si ho nasledovne:

```
for i := 1 to n do
  for j := i+1 to n do
    writeln(i,' ',j);
```

Keby sme si podnikateľov z predchádzajúcej úlohy očíslovali od 1 po 20, tento program by práve raz vypísal každú dvojicu ich čísel. Tento upravený program teda vypíše presne 190 riadkov, a teda pôvodný program vypíše 190 hviezdičiek.

> *Koľko hviezdičiek vypíše nasledujúca časť programu pre* `S = 'abeceda zjedla deda'` *a* `T = 'eden'`*?*
>
> ```
> for i := 1 to 16 do
>   for j := 1 to 4 do begin
>     write('*');
>     if S[i+j-1] <> T[j] then break;
>     if j = 4 then halt;
>   end;
> ```

(Správna odpoveď je 21. Uvedený program hľadá výskyt reťazca T v reťazci S tak, že postupne vyskúša a skontroluje všetky pozície, kde tento výskyt môže začínať. Každý výpis hviezdičky zodpovedá jednému porovnaniu znakov.)

> *Viete nájsť vstup pre predchádzajúci algoritmus, kde budú* `S` *a* `T` *rovnakej dĺžky ako vo vyššie uvedenom príklade a zároveň počet vypísaných hviezdičiek bude najväčší možný?*

1.5 Praktický pohľad

V predchádzajúcej časti sme pre veľmi jednoduché programy dokázali presne spočítať, koľko krokov urobia. Ale potrebuje programátor z praxe niekedy takéto presné výsledky?

Programátor väčšinou nevie presne, aké dáta bude jeho program spracúvať. V lepšom prípade vie približne odhadnúť ich veľkosť, v horšom ani to nie.

Napríklad programátor Andrej píše modul pre prihlasovanie na webový portál, ktorý má dnes 100 000 užívateľov. Počas najrušnejšej hodiny dňa sa ich prihlási okolo 5 000. Andrej teda približne vie, aké veľké dáta bude jeho program spracúvať, a vie, že jeho program musí byť dosť rýchly na to, aby stihol spracovať každého užívateľa za niekoľko desatín sekundy.

Bibiana píše program, ktorý bude simulovať skladanie proteínov. Čím bude jej program rýchlejší, tým viac a väčších simulácií stihne v rozumnom čase spraviť, a tým skôr sa vedci dostanú k dostatočným dátam.

Obaja v skutočnosti nepotrebujú vedieť presný počet krokov, ktorý ich program v danej situácii spraví. Stačil by im spôsob, ako približne zistiť, ako dlho ich program v danej situácii pobeží, resp. či je dostatočne rýchly.

1.6 Ako definovať časovú zložitosť?

V zadaní 2 sme si mohli všimnúť, že počet krokov, ktoré vykonáme pri simulácii dotyčného algoritmu, závisí od hodnoty premennej n. V tomto konkrétnom prípade by sme napríklad mohli povedať, že pre vstupnú hodnotu n daný program spraví $1.5n^2 - 0.5n + 1$ krokov.

Podobnú úvahu vieme spraviť pre ľubovoľný algoritmus A. Vždy sa dá definovať nasledujúca funkcia k_A: Nech v je nejaký vstup, na ktorom môžeme vykonať algoritmus A. Potom $k_A(v)$ je počet krokov, ktoré pri tom spravíme.

Ako sme však uviedli v predchádzajúcej časti, takýto pohľad je zbytočne presný. Všimnime si napríklad zadanie 4. Ako počet vypísaných hviezdičiek, tak aj celkový počet krokov algoritmu závisí nie len od dĺžky reťazcov `S` a `T`, ale aj od toho, ako presne tieto reťazce vyzerajú. Ľahko nahliadneme, že už pre takýto jednoduchý algoritmus neexistuje žiaden pekný jednoduchý matematický zápis funkcie k_A.

Našťastie ho ani nepotrebujeme poznať. To, čo nám o algoritme stačí vo väčšine prípadov vedieť, je jeho najhorší možný prípad. V prípade zadania 3 by nás teda mohla zaujímať odpoveď na otázku: Ak má reťazec `S` dĺžku x a reťazec `T` dĺžku y znakov, koľko najviac krokov spraví uvedený program?

Ak by sme poznali odpoveď na túto otázku, môžeme sa na ňu dívať ako na záruku: nech konkrétne reťazce vyzerajú, ako len chcú, my vieme zaručiť, že náš program sa po takom-a-takom počte krokov skončí.

A presne takto sa v oblasti návrhu a analýzy efektívnych algoritmov časová zložitosť definuje: Časovou zložitosťou algoritmu A je funkcia t_A taká, že hodnota $t_A(n)$ je najmenší počet krokov, ktoré A stačia na spracovanie ľubovoľného vstupu veľkosti n. (Alebo inými slovami, keby sme A vykonali pre všetky možné vstupy veľkosti n a zakaždým si zapísali počet vykonaných krokov, tak by $t_A(n)$ bolo maximum z týchto počtov.)

1.7 Časovú zložitosť stačí odhadnúť

Súčasné procesory zvládajú vykonať približne miliardu inštrukcií za sekundu. Pomocou tohto približného údaja môžeme jednoducho z časovej zložitosti programu odhadnúť, ako dlho by bežal na súčasnom počítači. Napr. program s časovou zložitosťou $5n^2 + 4n$ by pre $n = 10\,000$ bežal približne pol sekundy. (Samozrejme, konkrétna hodnota závisí od konkrétneho počítača.)

Skúsme si teraz položiť otázku opačne: ak vieme, akú má náš program časovú zložitosť a vieme, ako dlho ho sme ochotní nechať bežať, aký najväčší vstup ešte stihne spracovať?

V nasledujúcej tabuľke uvádzame názorný prehľad odpovedí na túto otázku pre niekoľko zaujímavých časových zložitostí.

čas/zložitosť	n	n^2	n^3	2^n	$n!$
milisekunda	1 000 000	1 000	100	20	10
sekunda	1 000 000 000	30 000	1 000	30	12
minúta	∞	250 000	4 000	35	14
hodina	∞	2 000 000	15 000	41	15
deň	∞	9 000 000	44 000	46	16
mesiac	∞	51 000 000	130 000	51	17
rok	∞	170 000 000	310 000	54	18
tisícročie	∞	∞	3 100 000	64	21

Tabuľka 1: Najväčšie n, pre ktoré program s danou časovou zložitosťou skončí v danom čase.
(Veľké hodnoty sú zaokrúhlené, väčšie ako miliarda sú nahradené symbolom ∞.)

Ak sa na algoritmy dívame z tejto perspektívy, ľahko si všimneme, že príliš nezáleží na tom, či má náš algoritmus časovú zložitosť n^2 alebo $n^2 + 100n$. Totiž akonáhle zoberieme dostatočne veľké n, hodnota n^2 bude rádovo väčšia ako $100n$, a teda tých $100n$ krokov navyše môžeme zanedbať. Keby sme do našej tabuľky pridali nový stĺpec pre zložitosť $n^2 + 100n$, vyzeral by skoro identicky ako stĺpec pre n^2.

A takisto nie je žiaden výrazný rozdiel medzi algoritmami s časovou zložitosťou n^3 a $7n^3$. Presný čas behu samozrejme závisí od presnej rýchlosti nášho počítača, ale ešte stále nám naša tabuľka povie, čo rádovo môžeme očakávať. Ak chceme spracovať vstupné dáta veľkosti $n = 40\,000$ a náš program má časovú zložitosť $7n^3$, bude mu to trvať niekoľko dní. Ak by bolo $n = 7\,000\,000$, rovno vieme povedať, že sa odpovede nedožijeme.

Programátor Cyril napísal program, ktorý načíta súradnice stredov a polomery N rôznych kružníc v rovine a následne zistí, koľko majú dokopy priesečníkov. Svoj program vám popísal nasledovne: „Pre každú dvojicu kružníc vypočítam vzdialenosť ich stredov a porovnám ju so súčtom polomerov. Podľa toho, či je väčšia, rovnaká alebo menšia, som našiel 0, 1 alebo 2 priesečníky.“

Odhadnite, ako rýchlo tento program spracuje 1 000 *kružníc. A čo* 100 000 *kružníc?*

Všetkých dvojíc kružníc je $n^2 = n(n-1)/2$. Cyrilov program pre každú dvojicu kružníc spraví konštantný počet krokov: niekoľko matematických operácií a jedno vyhodnotenie podmienky. Časová zložitosť tohto programu je teda zrejme nejakou kvadratickou funkciou premennej n.

Tisíc kružníc by teda tento problém mal bez problémov zvládnuť spracovať za výrazne menej ako sekundu. Pri stotisíc kružniciach bude čas behu rádovo minúta.

Danica vlani dostala od šéfa za úlohu zistiť, či nemajú v databáze zákazníkov nejakého zákazníka omylom viackrát. Túto úlohu vyriešila tak, že napísala program, ktorý porovnal každú dvojicu zákazníkov. Keď ho spustila, program necelé dve minúty bežal a na záver vypísal, že žiadne duplikáty nenašiel.

Dnes má firma trikrát toľko zákazníkov ako pred rokom. Ak by dnes Danica spustila svoj program (na tom istom počítači ako vlani), ako dlho by čakala, kým program dobehne?

O Danicinom programe opäť môžeme rozumne predpokladať len jediné: že jeho časová zložitosť je kvadratickou funkciou od počtu zákazníkov. Teraz už len stačí, keď si uvedomíme, že nič viac ani vedieť nepotrebujeme.

Ak si počet zákazníkov označíme n, tak pre dostatočne veľké n už bude časová zložitosť Danicinho programu takmer presne priamo úmerná n^2.

My síce nepoznáme koeficient tejto priamej úmernosti, ani ho však poznať nemusíme. Stačí si uvedomiť, že počet krokov, ktoré program vykoná pre $3n$ zákazníkov, musí byť s rovnakým koeficientom priamo úmerný číslu $(3n)^2$. A keďže $(3n)^2 = 9n^2$, bude to programu trvať 9-krát dlhšie ako v prvom prípade – čiže približne štvrť hodiny.

1.8 Načo je dobrá asymptotická rýchlosť rastu

Emil a Filoména sú vedci. Obaja nedávno vymysleli nové algoritmy na usporadúvanie čísel a zistili ich časové zložitosti. Časovú zložitosť Emilovho programu označíme e a Filoméninho f. Teda vieme, že Emilov algoritmus

potrebuje na usporiadanie n čísel spraviť v najhoršom prípade $e(n)$ krokov, zatiaľ čo Filoménin algoritmus ich potrebuje spraviť $f(n)$. Ako povedať, ktorý z nich je lepší?

Ak máme na mysli nejaké konkrétne použitie, pri ktorom približne vieme, ako veľa čísel budeme spracúvať, stačilo by porovnať zodpovedajúce hodnoty $e(n)$ a $f(n)$. Čo ak to ale nevieme? Dá sa niekedy povedať, že jeden z týchto algoritmov je „vo všeobecnosti lepší“ ako druhý?

Ukazuje sa, že áno. Pozerajme sa na podiel $e(n)/f(n)$. Táto hodnota nám hovorí, koľkokrát je pre dotyčné n Emilov algoritmus pomalší ako Filoménin. Položme si teraz otázku, čo sa stane, ak budeme n postupne zväčšovať. Ak hodnota podielu $e(n)/f(n)$ porastie cez všetky medze, znamená to, že Emilov algoritmus je čím ďalej, tým výraznejšie horší ako ten Filoménin. A teda až na konečne veľa malých prípadov sa vždy viac oplatí použiť Filoménin algoritmus.

Ak teda navrhujeme algoritmus v situácii, keď nevieme, kto ho kedy použije a na akých veľkých dátach, snažíme sa o to, aby bol vo všeobecnosti čo najlepší – teda aby jeho časová zložitosť rástla čo najpomalšie.

Príklad: Ak má Emilov algoritmus časovú zložitosť $e(n) = n^3 + n^2 + 1$ a Filoménin $f(n) = 100n^2 + 100n$, tak platí $e(n)/f(n) = n/100 + 1/(100n^2 + 100n)$. Pre pár malých hodnôt n je Emilov algoritmus rýchlejší, ale už napr. pre $n = 200$ je približne dvakrát pomalší od Filoméninho a pre $n = 10\,000$ bude už Emilov algoritmus bežať až stokrát dlhšie.

Všimnite si, že v predchádzajúcom príklade by sa nič podstatné nezmenilo, ak by e bola iná kubická funkcia a f iná kvadratická funkcia. Ich podiel by opäť bol približne rovný nejakej lineárnej funkcii, a teda čím väčšie vstupné dáta by sme uvažovali, tým horší by bol Emilov algoritmus v porovnaní s Filoméninym.

Špeciálne si všimnime, že vždy by existovala nejaká hranica n_0, pre ktorú by platilo, že pre všetky veľkosti vstupu väčšie od n_0 už má Filoménin algoritmus lepšiu časovú zložitosť ako Emilov.

1.9 Formálne značenie

Keď rozprávame o časovej zložitosti algoritmov, zväčša chceme zhora ohraničiť, ako rýchlo rastie nejaká funkcia (ktorej presné vyjadrenie často nepoznáme). Za týmto účelom si požičiame notáciu z matematickej analýzy. Najčastejšie používaným symbolom bude veľké písmeno O, ktorého význam je definovaný nasledovne:

Nech f a g sú rastúce funkcie na prirodzených číslach. Potom hovoríme, že f patrí do $O(g)$, ak existuje kladná konštanta c taká, že pre všetky dostatočne veľké n platí $f(n) \leq c \cdot g(n)$. Tento zápis čítame „funkcia f je veľké O od funkcie g“. Preložené z matematickej reči, tento zápis hovorí, že funkcia f rastie **nanajvýš rádovo** tak rýchlo ako funkcia g.

Formálne, veľké O predstavuje triedu (t.j. množinu) funkcií – teda $O(g)$ je trieda všetkých funkcií, ktoré rastú nanajvýš rádovo tak rýchlo ako funkcia g.

Navyše si toto značenie budeme často zjednodušovať: pod O(výraz obsahujúci n) budeme rozumieť „$O(f)$, kde f je funkcia definovaná predpisom $\forall n : f(n) =$ výraz obsahujúci n“. Teda napríklad „$O(n^2)$“ je to isté ako „$O(f)$, kde $\forall n : f(n) = n^2$“. Obe tieto značenia predstavujú *triedu všetkých funkcií, ktoré rastú nanajvýš rádovo tak rýchlo ako kvadratická funkcia.*

Ďalšie bežne používané symboly sú Ω a Θ.

Platí $f \in \Omega(g)$ práve vtedy, keď $g \in O(f)$. Omega teda predstavuje dolný odhad: $f \in \Omega(g)$ nám hovorí, že f rastie **aspoň rádovo** tak rýchlo ako g.

No a $f \in \Theta(g)$ platí vtedy, ak platí $f \in O(g)$ a zároveň $f \in \Omega(g)$ – inými slovami, keď funkcie f a g rastú rádovo rovnako rýchlo.

1.10 Príklady použitia formálneho značenia

Príklady použitia:

- Časová zložitosť Cyrilovho programu patrí do triedy $O(n^2)$.
  Skrátene: Časová zložitosť Cyrilovho programu je $O(n^2)$.
- Každá kubická funkcia je $O(n^3)$.
  (Vedeli by ste toto tvrdenie dokázať?)

- Ak $f(n) = 2n$ a $g(n) = n!$, tak $f \in O(g)$, ale neplatí $g \in O(f)$.
  (Slovne: Funkcia $2n$ rastie nanajvýš tak rýchlo ako $n!$, ale naopak to neplatí.)
- $n^2 + 10n$ patrí do $O(n^{47})$.
  (Všimnite si, že veľké O predstavuje len horný odhad, ten môže byť niekedy veľmi voľný.)
- Každý algoritmus, ktorý vie usporiadať pole n prvkov pomocou porovnávania dvojíc, potrebuje spraviť $\Omega(n \log n)$ porovnaní.

1.11 Zhrnutie

Nájsť horný odhad časovej zložitosti je často výrazne jednoduchšie ako ju určovať presne. Všimnime si napríklad náš starý známy program:

```
for i := 1 to n do
  for j := i + 1 to n do
    write('*');
```

Vidíme, že obsahuje dva vnorené for-cykly, pričom rozsah každého z nich je nanajvýš n. Dokopy teda tento algoritmus vykoná nanajvýš n^2 iterácií, a teda je jeho časová zložitosť nanajvýš kvadratická od n. Toto môžeme matematicky zapísať: „tento algoritmus má časovú zložitosť [ktorá patrí do] $O(n^2)$“.

Takto v praxi bežne vyzerá analýza zložitosti algoritmu. Celý postup si môžeme zhrnúť nasledovne:

- Zistíme, čo a ako ten algoritmus robí.
- Čo najtesnejšie zhora odhadneme počet krokov, ktoré spraví.
- Získaný odhad zapíšeme pomocou matematickej notácie (alebo slovne).