1. Časová a paměťová složitost

Cílem každého programátora je bezesporu navrhovat co nejlepší algoritmy. Navrhnout dobrý algoritmus dá však často značnou práci; nabízí se totiž otázka, jak vůbec poznáme, zda určitý algoritmus je kvalitní. Když máme dva algoritmy řešící stejnou úlohu, jak rozhodneme, který z nich je lepší? A co vlastně znamenají pojmy „lepší“ a „horší“?

Kritérií kvality může být mnoho. Například můžeme za lepší algoritmus považovat ten, který nás napadne jako první, nebo ten, jehož naprogramování nám dá méně práce. Obvykle však posuzujeme algoritmy na základě vlastností výsledného programu. Budou nás zajímat časové a paměťové nároky programu, tzn. rychlost výpočtu a velikost potřebné operační paměti počítače.

Jako první srovnávací metoda nás nejspíš napadne srovnávané algoritmy naprogramovat v nějakém programovacím jazyce, spustit je na větší množině testovacích dat a měřit se stopkami v ruce (nebo alespoň s těmi zabudovanými do operačního systému), který z nich je lepší. Takový postup se skutečně v praxi používá, z teoretického hlediska je však nevhodný. Když bychom chtěli v literatuře popsat vlastnosti určitého algoritmu, asi jen stěží napíšeme „na mém stroji doběhl do hodiny“. A jak bude fungovat na jiném stroji, s odlišnou architekturou, naprogramovaný v jiném jazyce, pod jiným operačním systémem, pro jinou sadu vstupních dat?

V této kapitole vybudujeme a vysvětlíme míru doby běhu algoritmu a jeho paměťových nároků, která bude nezávislá na technických podrobnostech stroje, jazyka a operačního systému, na němž bychom jinak algoritmus museli analyzovat. Těmto mírám budeme říkat *časová složitost* pro měření rychlosti algoritmu, a analogicky *paměťová složitost* pro měření paměťových nároků.

1.1. Jak fungují počítače uvnitř

Definice pojmu „počítač“ není samozřejmá. V současnosti i v historii bychom jistě našli spoustu strojů, kterým by se tak dalo říkat. My se přidržíme všeobecně uznávané definice, kterou v roce 1946 vyslovil vynikající matematik John von Neumann. Ta obsahuje následující body:

- Stroj má 5 funkčních jednotek:
  - řídící jednotka (řadič) – koordinuje činnost ostatních jednotek
  - aritmeticko-logická jednotka – provádí numerické výpočty, vyhodnocuje podmínky, ...
  - operační paměť – uchovává číselně kódovaná data a program⟨1⟩

⟨1⟩ Zde stojí za zdůraznění fakt, že data i program jsou ve stejné operační paměť

    - vstupní zařízení – zařízení, odkud se do počítače dostávají data k zpracování
    - výstupní zařízení – do tohoto zařízení zapisuje počítač výsledky své činnosti
- Struktura je nezávislá na zpracovávaných problémech, na řešení problému se musí zvenčí zavést návod na zpracování (program) a musí se uložit do paměti; bez tohoto programu není stroj schopen práce.
- Programy, data, mezivýsledky a konečné výsledky se ukládají do téže paměti.
- Paměť je rozdělená na stejně velké buňky, které jsou průběžně očíslované; přes číslo buňky (adresu) se dá přečíst nebo změnit obsah buňky.
- Po sobě jdoucí instrukce programu se uloží do paměťových buněk jdoucích po sobě, přístup k následující instrukci se uskuteční z řídící jednotky zvýšením instrukční adresy o 1.
- Instrukcemi skoku se dá odklonit od zpracování instrukcí v uloženém pořadí.
- Existují následující typy instrukcí:
    - aritmetické instrukce (sčítání, násobení, ukládání konstant, ...)
    - logické instrukce (porovnání, not, and, or, ...)
    - instrukce přenosu (z paměti do řídící jednotky a opačně, na vstup a výstup)
    - podmíněné a nepodmíněné skoky
    - ostatní (čekání, zastavení, ...)
- Všechna data (instrukce, adresy, ...) jsou číselně kódovaná, správné dekódování zabezpečují vhodné logické obvody v řídící jednotce.

Přesné specifikaci počítače, tedy způsobu vzájemného propojení jednotek, jejich komunikace a programování, popisu instrukční sady, říkáme *architektura*.

Nezabíhejme do detailů fungování běžných osobních počítačů, čili popisu jejich architektury. V každém z nich se však jednotky chovají tak, jak popsal von Neumann. Z našeho hlediska bude nejdůležitější podívat se co se děje, pokud na počítači vytvoříme a spustíme program.

Algoritmus zapíšeme obvykle ve formě vyššího programovacího jazyka. Zde je příklad v jazyce C.

```
#include <stdio.h>
```

---

ti. Počítač tak vlastně pojem „data“ a „program“ nerozlišuje, povoluje tedy třeba program měnit „pod rukou“ a podobně. Existují však i jiné architektury, například tzv. harvardská, které program a data důsledně oddělují.

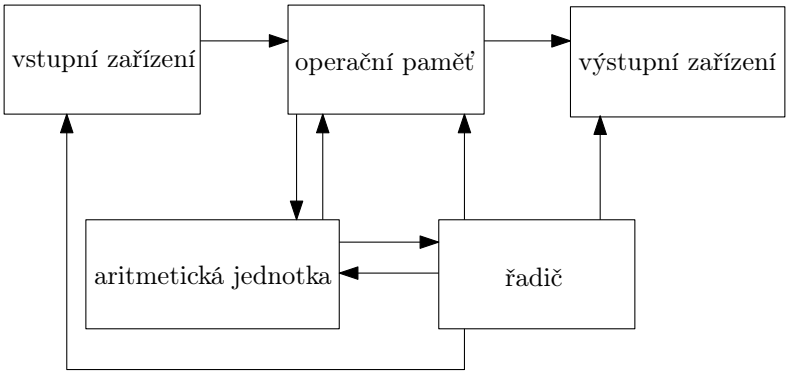

Schéma von Neumannova počítače

```
int main(void)
{
   static char s[] = "Hello world\n";
   int i, n = sizeof(s);
   for (i = 0; i < n; i++)
      putchar(s[i]);
   return 0;
}
```

Aby řídící jednotka mohla program provést, musí místo předchozího programu dostat posloupnost jednoduchých instrukcí, které jsou číselně kódovány. K tomu (a mnoha dalším věcem) slouží proces překladu (kompilace) zdrojového programu do *strojového kódu*.

Následujícími příkazy v operačním systému Linux jsme spustili překladač jazyka C, přeložili vzorový program do strojového kódu a spustili ho.

```
$ gcc hello.c -o hello
$ ./hello
Hello world
$
```

Přeskočíme-li všechny podrobnosti překládání programu, konečným produktem překladu je obvykle spustitelný program, to jest posloupnost strojových instrukcí pro daný stroj. Narozdíl od našeho příkladu zapsaném v jazyce C, který na všech počítačích s překladačem jazyka C bude vypadat stejně, strojový kód se bude lišit architekturu od architektury, operační systém od operačního systému, dokonce překladač od překladače.

Ukážeme příklad úseku strojového kódu, který vznikl po překladu našeho příkladu v operačním systému Linux na architektuře Intel x86. Aby se lidem jednotlivé

instrukce lépe četly, mají přiřazeny své symbolické názvy. Tomuto jazyku symbolických instrukcí se říká *assembler*. Kromě symbolických názvů instrukcí dovoluje assembler ještě pro pohodlí pojmenovat adresy a několik dalších užitečných věcí.

```
main:   leal    4(%esp), %ecx
        andl    $-16, %esp
        pushl   -4(%ecx)
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ecx
        subl    $20, %esp
        movl    $13, -8(%ebp)
        movl    $0, -12(%ebp)
        jmp     L2
L3:     movl    -12(%ebp), %eax
        movzbl  L1(%eax), %eax
        movsbl  %al,%eax
        movl    %eax, (%esp)
        call    putchar
        addl    $1, -12(%ebp)
L2:     movl    -12(%ebp), %eax
        cmpl    -8(%ebp), %eax
        jl      L3
        movl    $0, %eax
        addl    $20, %esp
        popl    %ecx
        popl    %ebp
        leal    -4(%ecx), %esp
        ret
L1:     .string "Hello world\n"
```

A pro zajímavost ještě na architektuře PPC64:

```
main:   stwu 1,-32(1)
        mflr 0
        stw 31,28(1)
        stw 0,36(1)
        mr 31,1
        li 0,13
        stw 0,12(31)
        li 0,0
        stw 0,8(31)
L2:     lwz 0,8(31)
        lwz 9,12(31)
        cmpw 7,0,9
        blt 7,L5
```

```
        b L3
L5:     lis 9,s0@ha
        la 9,s0@l(9)
        lwz 0,8(31)
        add 9,9,0
        lbz 0,0(9)
        rlwinm 0,0,0,0xff
        mr 3,0
        bl putchar
        lwz 9,8(31)
        addi 0,9,1
        stw 0,8(31)
        b L2
L3:     li 0,0
        mr 3,0
        lwz 11,0(1)
        lwz 0,4(11)
        mtlr 0
        lwz 31,-4(11)
        mr 1,11
        blr
s0:     .string "Hello world\n"
```

Každá instrukce je zapsána posloupností několika bytů. Věříme, že čtenář si dokáže představit přechozí kód zapsaný v číslech a ukázku vynecháme.

Programátor píšící programy v assembleru musí být perfektně seznámen s instrukční sadou procesoru, vlastnostmi architektury, technickými detaily služeb operačního systému a mnoha dalšími věcmi. ⟨2⟩

1.2. Rychlost konkrétního výpočtu

Dejme tomu, že chceme změřit dobu běhu našeho příkladu „Hello world“ z předchozího oddílu. Začneme nejjednodušší metodou – se stopkami v ruce. Jak přesné stopky a rychlé ruce bychom na to museli mít nyní ponechme stranou. Spustíme-li na operačním systému program několikrát, nejspíš pokaždé naměříme o něco rozdílné časy. Může za to aktivita ostatních procesů, stav operačního systému, obsahy nejrůznějších vyrovnávacích pamětí a desítky dalších věcí. A to ještě ukázkový program nečte žádná vstupní data. Co kdyby se doba jeho běhu odvíjela podle nich?

⟨2⟩ Z vlastní zkušenosti můžeme jako assembler doporučit GNU Assembler, který je dobře zdokumentován. Detaily jednotlivých instrukcí, fungování procesoru a pomocných systémů je třeba hledat v dokumentaci výrobce, která bývá k dispozici na jeho webových stránkách.

Takový přístup se tedy hodí pouze pro testování kvality konkrétního programu na konkrétním hardware a konfiguraci. Nezatracujeme ho, velmi často se používá pro testování programů určených k nasazení v těch nejvypjatějších situacích. Ale naším cílem v této kapitole je vyvinout prostředek na měření doby běhu obecně popsaného algoritmu, bez nutnosti naprogramování v konkrétním programovacím jazyce a architektuře. Navíc zohledňující závislost na množství vstupních dat.

Dejme se tedy do postupné tvorby takové metody. Zapomeňme odteď na detaily překladu programu do strojového kódu, zapomeňme dokonce na detaily nějakého konkrétního programovacího jazyka. Algoritmy začneme popisovat *pseudokódem*. To znamená, že nebudeme v programech zabíhat do technických detailů konkrétních jazyků či architektury, nicméně s jejich znalostí bude už potom snadné pseudokód do programovacího jazyka přepsat. Operace budeme popisovat slovně, případně matematickou symbolikou.

Nyní spočítáme celkový počet provedených tzv. *elementárních operací*. Tímto pojmem rozumíme především operace sčítání, odčítání, násobení, porovnávání; také základní řídící konstrukce, jako jsou třeba skoky a podmíněné skoky. Zkrátka to, co normální procesor zvládne jednou nebo nejvýše několika instrukcemi. Elementární operací rozhodně není například přesun paměťového bloku z místa na místo, byť vypadá zapsaný jediným příkazem, nebo třeba většina operací s textovými řetězci.

Čas vykonání jedné elementární operace prohlásíme za jednotkový a zbavíme se tak jakýchkoli jednotek ve výsledné době běhu algoritmu. V zásadě je za elementární operace možné zvolit libovolnou rozumnou sadu – doba provádění programu se tak změní nejvýše konstanta-krát, na čemž, jak za chvíli uvidíme, zase tolik nezáleží.

Pozorný čtenář se nyní jistě zeptá, jak počítat počet provedených operací u algoritmu, jehož doba běhu závisí na vstupu a může probíhat mnoha různými větvemi. V takovém případě vždy *vezmeme maximální možný počet provedených operací*. Tento počet navíc vyjádříme vzhledem k vstupu, nejčastěji vzhledem k jeho velikosti.

Jednoduché výpočty časových nároků programu

Než pokročíme dále, zkusme určit počet provedených operací u jednoduchých algoritmů. Konkrétně, budeme nejdříve místo počtu operací počítat počet vypsaných hvězdiček. Ze vstupu všechny algoritmy nejprve na úvod přečtou přirozené číslo N. Čtenář nechť zkusí nejdříve u každého algoritmu počet hvězdiček spočítat sám a teprve potom se podívat na náš výpočet.

Algoritmus Hvězdičky 1
Vstup: číslo N

1. Pro i od 1 do N opakuj:
2. Pro j od 1 do N opakuj:
3. Tiskni $*$

V algoritmu 1 vidíme, že vnější cyklus se provede N-krát, vnořený cyklus také N-krát, dohromady tedy N^2 vytištěných hvězdiček.

Algoritmus Hvězdičky 2
Vstup: číslo N

1. Pro i od 1 do N opakuj:
2. Pro j od 1 do i opakuj:
3. Tiskni $*$

Rozepišme, kolikrát se provede vnitřní cyklus v závislosti na i. Pro $i = 1$ se provede jedenkrát, pro $i = 2$ dvakrát, a tak dále, až pro $i = N$ se provede N-krát. Dohromady se vytiskne $1 + 2 + 3 + \ldots + N$ hvězdiček, což například pomocí vzorce na součet aritmetické posloupnosti sečteme na $N(N + 1)/2$.

Algoritmus Hvězdičky 3
Vstup: číslo N

1. Dokud $N \geq 1$, opakuj:
2. Tiskni $*$
3. $N \leftarrow \lfloor N/2 \rfloor$

V každé iteraci cyklu se N sníží na polovinu. Provedeme-li cyklus k-krát, sníží se hodnota N na $\lfloor N/2^k \rfloor$, neboli klesá exponenciálně rychle v závislosti na počtu iterací cyklu. Chceme-li určit počet iterací, vyřešíme rovnici $\lfloor N/2^k \rfloor = 1$ pro neznámou k. Výsledkem je tedy zhruba dvojkový logaritmus N. Čtenáře odkážeme na cvičení **??**, aby výsledek určil přesně.

Algoritmus Hvězdičky 4
Vstup: číslo N

1. Dokud je $N > 0$, opakuj:
2. Je-li N liché:
3. Pro i od 1 do N opakuj:
4. Tiskni $*$
5. $N \leftarrow \lfloor N/2 \rfloor$

Zde se již situace začíná komplikovat. V každé iteraci vnějšího cyklu se N sníží na polovinu a vnořený cyklus se provede pouze tehdy, bylo-li předtím N liché. To, kolikrát se vnořený cyklus provede, tedy nepůjde úplně snadno vyjádřit pouze z velikosti čísla N. V souladu s našimi pravidly tedy počítejme nejdelší možný průběh algoritmu, kdy test na lichost N pokaždé uspěje. Tehdy se vytiskne $h = N + \lfloor N/2 \rfloor + \lfloor N/2^2 \rfloor + \ldots + \lfloor N/2^k \rfloor + \ldots + 1$ hvězdiček. Protože není na první pohled vidět, kolik h přepsané do jednoduchého vzorce vyjde, spokojíme se alespoň s horním odhadem na h.

Označme symbolem s počet členů v součtu h. Hodnotu h shora odhadneme jako

$$h = \sum_{i=0}^{s} \frac{N}{2^i} \leq \sum_{i=0}^{\infty} \frac{N}{2^i} = N \sum_{i=0}^{\infty} \frac{1}{2^i}$$

přidáním dalších členů do řady až do nekonečna. Jak víme z matematické analýzy, řada $k = \sum_{i=0}^{\infty} 1/2^i$ konverguje, tj. nasčítá se na pevné konečné číslo. Dostáváme, že

počet vytištěných hvězdiček nebude vyšší, než kN, kde k je pevná konstanta nijak nezávisející na N. Číslo k lze spočítat i přesně, viz cvičení 1.2.3.

Protože se v této kapitole snažíme vybudovat míru doby běhu algoritmu a nikoli počtu vytištěných hvězdiček, ukážeme u našich příkladů, že z počtu vytištěných hvězdiček vyplývá i řádový počet všech provedených operací. V algoritmu 1 na vytištění jedné hvězdičky provedeme maximálně dvě operace: změnu proměnné j a možná ještě změnu proměnné i. V algoritmech 2 a 3 je to velmi podobně – na vytištění jedné hvězdičky potřebujeme maximálně dvě další operace.

Algoritmus 4 v případě, že všechny testy lichosti uspějí, pro tisk hvězdičky provede změnu proměnné i, maximálně jeden test lichosti, maximálně jednu aritmetickou operaci s N a podmíněný skok. Co však je-li někdy v průběhu N sudé? Co když test na lichost uspěje pouze jednou nebo dokonce vůbec? (K rozmyšlení: kdy se to může stát?) Může se tedy přihodit, že se vytiskne jen velmi málo hvězdiček (třeba jedna) a algoritmus přesto vykoná velké množství operací. V tomto algoritmu tedy počet operací s počtem hvězdiček nekoresponduje. Čtenáře odkážeme na cvičení **??**, aby zjistil přesně, na čem počet vytištěných hvězdiček závisí.

Pojďme shrnout počty vykonaných kroků (nebo alespoň jejich horní odhady) našich čtyř algoritmů: $2N^2$, $2N(N+1)/2$, což je po úpravě N^2+N, $2\log_2 N$ a kN. Podíváme se na chování algoritmu pro gigantická N, řekněme v řádu bilionů.

Nejprve si všimněme, že algoritmus 3 bude nejrychlejší ze všech. I pro N řádově bilion se vykoná pouze několik málo kroků. Algoritmus 4 vykoná kroků maximálně bilion krát pevná konstanta k. Úplně nejpomalejší budou algoritmy 1 a 2, neporovnatelně více než algoritmy 3 a 4.

Jak to, že jsme schopni předpovědět rychlost algoritmů, aniž bychom je spustili? To je právě smyslem určování časové složitosti. Vyjádřili jsme počet kroků matematickou funkcí (a když jsme to nesvedli, tak jsme ji aspoň co nejlépe odhadli) a na základě ní jsme schopni řádově předpovídat vlastnosti algoritmu.

Další postřeh se bude týkat algoritmů 1 a 2. Pro, řekněme, $N = 10^{10}$ vykoná první algoritmus $2{\cdot}10^{20}$ hvězdiček a druhý algoritmus zhruba 10^{20} kroků, což je skoro tolik co první algoritmus, tedy alespoň řádově. Když se zadíváme na vzorce s počty kroků, v obou figurují členy N^2. Tato funkce roste dominantně vzhledem k ostatním členům a pro obrovská N bude v podstatě jediná důležitá, pomaleji rostoucí členy se „ztratí“.

Stejný osud potká multiplikativní konstanty, v případě algoritmu 1 konstantu 2. Multiplikativní konstanta není důležitá, protože různé počítače jsou různě rychlé. Řádový počet operací již však zanedbatelný není.

Cvičení:

1. Určete počet vytištěných hvězdiček u algoritmu 3 naprosto přesně, jednoduchým vzorcem.
2. Na čem u algoritmu 4 závisí počet vytištěných hvězdiček?

3. Dokažte, že

$$\sum_{i=0}^{\infty} \frac{1}{2^i} = 2.$$

1.3. Zavedení časové a paměťové složitosti

Časová složitost

Zanechme nadále ilustračních příkladů tisknoucích hvězdičky a počítejme u algoritmů množství provedených elementárních operací. Vyzbrojeni těmito poznatky, popíšeme „kuchařku“, jak určit řádově dobu běhu algoritmu, kterou už můžeme nazývat *časovou složitostí*. Tomuto způsobu odhadování růstu funkcí říkáme také *asymptotické odhady* a *asymptotická složitost*.

V naprosté většině případů nás bude zajímat doba běhu algoritmu v nejhorším možném případě měřená v závislosti na velikosti vstupních dat a pokud ji neumíme spočítat přesně, tak alespoň stanovíme co nejlepší horní odhad. Důležité je chování algoritmu pro obrovské vstupy. Pro malé množství zpracovávaných dat dobře poslouží doslova každý správný algoritmus. Jenže na obrovských vstupech bude opravdu znát efektivita, bude rozdíl, jestli náš algoritmus poběží vteřinu, hodinu nebo 100 let.

1) Určíme počet $f(N)$ provedených elementárních operací algoritmu v závislosti na vstupu o velikosti N, v nejdelším možném průběhu algoritmu. Pokud neumíme určit počet operací přesně, najdeme alespoň co nejlepší horní odhad na $f(N)$.
2) Ve výsledné formuli $f(N)$, která je součtem několika členů, ponecháme pouze nejrychleji rostoucí funkci, ostatní zanedbáme, tj. vypustíme.
3) Seškrtáme multiplikativní konstanty. Ale jen ty! Nikoli ostatní čísla ve vzorci.

Jak bychom podle naší kuchařky postupovali u ukázkového algoritmu 2: Už jsme spočetli, ze se vykoná $N(N+1)/2 = N^2/2 + N/2$ elementárních operací. Škrtneme člen $N/2$ a zbude nám tedy $N^2/2$. Na závěr škrtneme multiplikativní konstantu $1/2$. Pozor, dvojka v exponentu není multiplikativní konstanta.

Funkci $g(N)$, která zbude, nazveme časovou složitostí algoritmu a tento fakt označíme výrokem „algoritmus má časovou složitost $\mathcal{O}(g(N))$“. Naše ukázkové algoritmy tudíž mají po řadě časovou složitost $\mathcal{O}(N^2)$, $\mathcal{O}(N^2)$, $\mathcal{O}(\log N)$ a $\mathcal{O}(N)$. Přemýšlivému čtenáři necháme v cvičení 1.3.1 k rozmyšlení, proč jsme u třetího příkladu zcela vynechali fakt, že se jednalo o dvojkový logaritmus.

Složitosti algoritmů mohou být velmi komplikované funkce. Nejčastěji se však setkáváme s algoritmy, které mají jednu z následujících složitostí. Složitosti $\mathcal{O}(N)$ říkáme *lineární*, $\mathcal{O}(N^2)$ *kvadratická*, $\mathcal{O}(N^3)$ *kubická*, $\mathcal{O}(\log N)$ *logaritmická*, $\mathcal{O}(2^N)$

exponenciální a $\mathcal{O}(1)$ (tedy že se provede pouze konstantně mnoho kroků) *konstantní*.

Časová složitost hraje zásadní roli v kvalitě algoritmu. Pro ilustraci v následující tabulce uveďme, jak rychle rostou určité funkce, které se často vyskytují jako časové složitosti algoritmů.

Funkce	$N = 10$	$N = 100$	$N = 1000$	$N = 10\,000$
$\log N$	1	2	3	4
N	10	100	1000	10 000
$N \log N$	10	200	3000	40 000
N^2	100	10 000	10^6	10^8
2^N	1024	$\sim 10^{31}$	$\sim 10^{310}$	$\sim 10^{3100}$

Vidíme, že algoritmus s exponenciální časovou složitostí by pro velká N vykonal skutečně enormní množství operací. Zkusme odhadnout, jak dlouho by běžel algoritmus s exponenciální časovou složitostí na současném počítači typu PC pro $N = 70$. Uvážíme jako průměr procesor s frekvencí 2 GHz, který v jednom tiku zpracuje jednu instrukci, čili operaci. Za rok by tedy tento počítač byl schopný vykonat maximálně $365 \cdot 24 \cdot 60 \cdot 60 \cdot 2 \cdot 10^9 \approx 63 \cdot 10^{15}$ operací. Protože celkem je k vykonání $2^{70} \approx 10^{21}$ operací, nedočkali bychom se výsledku ani za 10 000 let.

Pomalost exponenciálních algoritmů je také možné vidět na následujícím postřehu: kdybychom zdvojnásobili rychlost počítače, umožní nám to ve stejném čase zpracovat pouze o 1 větší vstup. Proto se zpravidla snažíme exponenciálním algoritmům vyhýbat a uchylujeme se k nim, pouze pokud nemáme jinou možnost. ⟨3⟩

Algoritmům se složitostmi $\mathcal{O}(N^k)$ pro pevná konstantní k říkáme *polynomiální* a jsou chápány jako efektivní.

Paměťová složitost

Velmi podobně jako časová složitost se dá zavést tzv. *paměťová složitost* (někdy též *prostorová složitost*), která měří paměťové nároky algoritmu. K tomu musíme spočítat, kolik nejvíce tzv. elementárních paměťových buněk bude v daném algoritmu v každém okamžiku použito. V běžných programovacích jazycích (jako jsou například C nebo Pascal) za elementární buňku můžeme považovat například proměnnou typu `integer`, `float`, `byte`, či ukazatel, naopak elementární velikost rozhodně nemají například pole či textové řetězce.

Opět vyjádříme množství spotřebovaných paměťových buněk funkcí $f(N)$ v závislosti na velikosti vstupu N, pokud to neumíme přesně, tak alespoň co nejlepším

⟨3⟩ Znalec trhu s počítačovým hardware by zde mohl namítnout, že vývoj počítačů jde kupředu tak rychle, že podle empiricky ověřeného Mooreova zákona se každé dva roky výkon počítačů zdvojnásobí. To však znamená pouze to, že algoritmus se složitostí $\mathcal{O}(2^N)$ na o 20 let novějším stroji ve stejném čase zpracuje pouze o 10 větší vstup.

horním odhadem, aplikujeme tříbodovou kuchařku a výsledek zapíšeme pomocí notace $\mathcal{O}(g(N))$. V našich čtyřech příkladech je tedy všude paměťová složitost $\mathcal{O}(1)$, neboť vždy používáme pouze konstantní množství celočíselných proměnných.

Asymptotická notace aneb $\mathcal{O}, \Omega, \Theta$

Matematicky založený čtenář jistě cítí, že poněkud vágní popis určení časové a paměťové složitosti algoritmu (tříbodová kuchařka) je dosti nepřesný a žádá si exaktní definice. Pojďme se do ní pustit.

Mějme dvě funkce $f, g : \mathbb{N} \to \mathbb{N}$. Funkce $f(n)$ bude značit počet elementárních operací (buněk), který jsme nějak spočetli. Řekneme, že funkce $f(n)$ je třídy $\mathcal{O}(g(n))$, jestliže existuje taková kladná reálná konstanta C, že pro všechna přirozená čísla od jistého n_0 počínaje platí $f(n) \leq Cg(n)$. To znamená, že funkce $g(n)$ shora omezuje $f(n)$ až na multiplikativní konstantu. Funkci $g(n)$ se říká *horní asymptotický odhad* funkce $f(n)$.

Technicky vzato je $\mathcal{O}(g(n))$ množina všech funkcí $f(n)$, které splňují předchozí definici, měli bychom tedy správně psát $f(n) \in \mathcal{O}(g(n))$. V literatuře se však v tomto případě formalismy příliš nedodržují, místo formálně správného $f(n) \in \mathcal{O}(g(n))$ se používá značení $f(n) = \mathcal{O}(g(n))$, případně fráze „f je $\mathcal{O}(g(n))$“ a podobně. Přijměme tedy i my tyto zvyklosti a používejme stejné (čistě formálně vzato nesprávné) výrazy a značení.

Nabízí se také otázka, proč jsme zavedli číslo n_0 a splnění nerovnosti požadujeme až od něj dále. Tato konstrukce nám totiž umožní volit za $g(n)$ i funkce, které mají několik počátečních funkčních hodnot nulových či dokonce záporných a nenašli bychom tedy jinak vhodnou konstantu C.

Uvědomíme si nyní, že v předchozím oddílu popsaná tříbodová „kuchařka“ je vlastně důsledkem právě vyslovené definice. V bodu 1 musíme určit největší možný počet provedených elementárních operací daného algoritmu vzhledem k velikosti vstupu, což je přesně určení funkce f. V bodu 2 ponecháme z formule popisující funkci f, která je součtem několika členů, pouze nejrychleji rostoucí člen. Přesněji řečeno, pokud je $f(n) = f_1(n) + f_2(n)$ a f_1 roste rychleji než f_2, člen f_2 vynecháme. Pojem „f_1 roste rychleji než f_2“ však znamená přesně to, že existuje jisté n_0 takové, že $f_1(n) \geq f_2(n)$ pro $n \geq n_0$, čili $f_2(n) = \mathcal{O}(f_1(n))$ a tedy i $f(n) = \mathcal{O}(f_1(n))$. Na závěr, když už zbývá pouze $f(n) = c \cdot f'(n)$ pro nějakou konstantu c, všimněme si, že v definici lze zvolit $C := c$ a tudíž $f(n) = \mathcal{O}(f'(n))$, můžeme tedy škrtat multiplikativní konstantu.

Zavádíme též dolní asymptotický odhad funkce. Mějme dvě funkce $f, g : \mathbb{N} \to \mathbb{N}$. Řekneme, že funkce $f(n)$ je třídy $\Omega(g(n))$, jestliže existuje taková kladná reálná konstanta C, že pro všechna přirozená čísla od jistého n_0 počínaje platí $f(n) \geq Cg(n)$. To znamená, že funkce $g(n)$ zdola omezuje $f(n)$ až na multiplikativní konstantu.

Sluší se také vysvětlit původ slova *asymptotický odhad*. Ten pochází z toho, že zkoumáme chování pro obrovské vstupy, tedy pro n blížící se k nekonečnu. V matematické analýze se zkoumá tzv. asymptota funkce, což je přímka, jejíž vzdálenost

od funkce se s rostoucí souřadnicí zmenšuje a v nekonečnu se dotýkají. Odtud tedy název.

Vzhledem k naší definici může být vyjádření složitosti algoritmu pomocí symbolu $\mathcal{O}$ dosti hrubé. Kvadratická funkce $2N^2 + 3N + 1$ je totiž třídy $\mathcal{O}(N^2)$, ale podle uvedené definice patří také do třídy $\mathcal{O}(N^3)$, $\mathcal{O}(N^4)$, atd. Proto se zavádí ještě symbol Θ. Řekneme, že funkce $f(n)$ je třídy $\Theta(g(n))$, jestliže $f(n)$ je z $\mathcal{O}(g(n))$ a současně $f(n)$ je z $\Omega(g(n))$. Například naše ukázkové algoritmy 1,2,3,4 mají tedy složitosti po řadě $\Theta(N^2)$, $\Theta(N^2)$, $\Theta(\log N)$ a $\Theta(N)$. Symboly $\Omega(g(n))$ a $\Theta(g(n))$ značí (stejně jako $\mathcal{O}(g(n))$) množiny funkcí splňujících příslušné definice.

Při skutečném srovnávání algoritmů bychom správně měli posuzovat jejich složitost podle tříd složitosti Θ, nikoli podle $\mathcal{O}$. Analýzou algoritmu však obvykle dostáváme pouze horní odhad počtu provedených instrukcí nebo potřebných paměťových míst. Při pečlivé analýze tento odhad zpravidla nebývá příliš vzdálený skutečnosti a je to tedy nejen určení třídy $\mathcal{O}$, ale i třídy Θ. Dokazovat tuto skutečnost formálně pro složitější algoritmy však bývá komplikované, bez důkazu by zase nebylo korektní používat symbol Θ. Budeme proto nadále vyjadřovat složitost algoritmů převážně pomocí symbolu $\mathcal{O}$. Při tom však budeme usilovat o to, aby byl náš odhad asymptotické složitosti co nejlepší.

Cvičení:

1. Proč je časová složitost druhého ilustračního algoritmu $\mathcal{O}(\log N)$ a vynechali jsme fakt, že se jednalo o dvojkový logaritmus?
2. Jaká je složitost následujícího (pseudo)kódu vzhledem k N?

```
while N > 0
    if odd(i) then N--
    else N = N div 2
```

3. Jaká je asymptotická složitost funkce $\log_n(n!)$?

1.4. Výpočetní model RAM

Matematicky založený jedinec stále nemůže být plně spokojen. Doposud jsme totiž odbývali přesné určení toho, co můžeme v algoritmu považovat za elementární operace a elementární paměťové buňky. Naší snahou bude vyhnout se obtížně řešitelným otázkám u věcí jako například reprezentace reálných čísel a zacházení s nimi. Situaci vyřešíme šalamounsky – definujeme vlastní teoretický stroj, který bude mít přesně definované chování, přesně definovaný čas provádění instrukcí a přesně definovaný rozsah a vlastnosti paměťové buňky. Potom dává dobrý smysl měřit časovou a paměťovou náročnost naprogramovaného algoritmu naprosto přesně – nezdržují nás vedlejší efekty reálných počítačů a operačních systémů.

Jedním z mnoha teoretických modelů je tzv. *Random Access Machine*, neboli RAM.⟨4⟩ RAM není jeden pevný model, nýbrž spíše rodina podobných strojů, které sdílejí určité společné vlastnosti.

⟨4⟩ Název lze přeložit do češtiny jako „stroj s náhodným přístupem“. Méně otrocký

Paměť RAMu tvoří pole celočíselných buněk adresovatelné nezápornými celými čísly. Každá buňka pojme jedno celé číslo. Bystrý čtenář se nyní otáže: „To jako neomezeně velké číslo?“ Problematiku omezení kapacity buňky rozebereme v této kapitole později.

Program je konečná posloupnost sekvenčně prováděných instrukcí dvou typů: aritmetických a řídících.

Aritmetické instrukce mají obvykle dva vstupní argumenty a jeden výstupní argument. Argumenty mohou být buďto přímé konstanty (s vyjímkou výstupního argumentu), přímo adresovaná paměťová buňka (zadaná číslem) nebo nepřímo adresovaná paměťová buňka (její adresa je uložena v přímo adresované buňce).

Řídící instrukce zahrnují skoky (na konkrétní instrukci programu), podmíněné skoky (například když se dva argumenty instrukce rovnají) a instrukci zastavení programu.

Na začátku výpočtu obsahuje pamět v určených buňkách vstup a obsah ostatních buněk je nedefinován. Potom je program sekvenčně prováděn, instrukci za instrukcí. Po zastavení programu je obsah paměti interpretován jako výstup programu.

Zmiňme také, že existují „ještě teoretičtější“ výpočetní modely, jejichž zástupcem je tzv. Turingův stroj.

Konkrétní model RAMu

V našem popisu strojů z rodiny RAM jsme vynechali mnoho podstatných detailů. Například přesný čas vykonávání jednotlivých instrukcí, povolený rozsah čísel v jedné paměťové buňce, prostorovou složitost jedné buňky přesné vymezení instrukční sady, zejména aritmetických operací.

V tomto oddílu přesně definujeme jeden konkrétní model RAMu. Popíšeme tedy paměť, zacházení s programem a výpočtem, instrukční sadu a chování stroje.

Procesor v každém kroku provede právě jednu instrukci. K paměti přistupuje procesor přes pseudopole `[i]`, kde `i` je celé číslo. To znamená, že lze použít též indexaci zápornými čísly. Lze použít nejvýše jednonásobnou nepřímou adresaci paměti, tj. lze jako index do pseudopole představujícího paměť použít například jeden z výrazů `[42]`, `[[-13]]`, ale nikoliv `[[[5]]]`. Rovněž nelze použít jako adresu v paměti jakýkoliv aritmetický výraz (např. `[3*[5]]`).

Vstup a výstup stroj dostává a předává většinou v paměťových buňkách s nezápornými indexy, buňky se zápornými indexy se obvykle používají pro pomocná data a proměnné. Prvních 26 buněk se zápornými indexy, tj. `[-1]` až `[-26]` má pro snazší použití přiřazeno aliasy `A`, `B`, až `Z` a říkáme jim *registry*. Jejich hodnoty lze libovolně číst a zapisovat a používat pro indexaci paměti, lze tedy psát např. `[A]`, ale nikoliv `[[A]]`. Využití registrů spočívá v často užívaných pomocných proměnných.

---

a výstižnější překlad by mohl znít „stroj s přímým přístupem do paměti“, což je však zase příliš dlouhé a kostrbaté, stroji tedy budeme říkat prostě RAM. Pozor, hrozí zmatení zkratek s *Random Access Memory*, čili běžným názvem operační paměti počítače typu PC.

Nechť X, Y a Z představují některý z výše uvedených výrazů pro přístup do paměti či registrů, Y a Z mohou být i celočíselné konstanty. Potom instrukce RAMu jsou tvaru:

- Přiřazení: X`:=`Y
- Unární minus: X`:=-`Y
- Součet: X`:=`Y`+`Z
- Rozdíl: X`:=`Y`-`Z
- Součin: X`:=`Y`*`Z
- Celočíselné dělení: X`:=`Y`/`Z, kde Z musí být nenulové.
- Zbytek po celočíselném dělení: X`:=`Y`%`Z, kde Z musí být nenulové.
- Bitová konjunkce: X`:=`Y`&`Z
- Bitová disjunkce: X`:=`$Y|Z$
- Bitová nonekvivalence: X`:=`Y`^`Z
- Bitový posun doleva: X`:=`Y`<<`Z
- Bitový posun doprava: X`:=`Y`>>`Z
- Prázdná instrukce: `nop`
- Ukončení výpočtu: `halt`
- Nepodmíněný skok: `goto LABEL`, kde `LABEL` je návěští, definuje se napsáním `LABEL:` před instrukci.
- Podmíněný příkaz: `if LOGEXPR then INSTR`

`INSTR` je libovolná instrukce mimo podmíněného příkazu a `LOGEXPR` je jeden z následujících logických výrazů:

- Test rovnosti: Y`=`Z
- Negace testu rovnosti: Y`<>`Z
- Test ostré nerovnosti: Y`<`Z, případně Y`>`Z
- Test neostré nerovnosti: Y`<=`Z, případně Y`>=`Z

Doba provádění podmíněného příkazu nezávisí na splnění jeho podmínky a je stejná jako doba provádění libovolné jiné instrukce.

Časovou složitost programu pro vstup velikosti N (čili zabírající N paměťových buněk) definujeme jako maximum z počtu vykonaných instrukcí přes všechny možné legální vstupy velikosti N. Paměťovou složitost programu pro vstup velikosti N definujeme jako maximální rozdíl nejvyššího použitého indexu paměti od nejnižšího použitého indexu paměti, počítaný přes všechny legáalní vstupy velikosti N.

Příklad programu pro RAM

Pro ilustraci přepíšeme algoritmus 2 z předchozích oddílů co nejvěrněji do programu pro náš RAM. Připomeňme algoritmus 2:

Algoritmus Hvězdičky 2
Vstup: číslo N

1. Pro i od 1 do N opakuj:

2. Pro j od 1 do i opakuj:
3. Tiskni *

Zadání pro RAM formulujeme takto: V buňce [0] je uloženo číslo N. Výstup je tvořen posloupností buněk počínaje [1], ve kterých je v každé zapsána jednička (namísto hvězdičky jako v původním programu).

```
                A:=[0]
                C:=1
VNEJSI:         if A=0 then halt
                B:=A
                A:=A-1
VNITRNI:        if B=0 then goto VNEJSI
                [C]:=1
                C:=C+1
                B:=B-1
                goto VNITRNI
```

Omezení kapacity paměťové buňky

Náš model má zatím jednu výrazně nereálnou vlastnost – neomezenou kapacitu paměťové buňky. Toho lze využít k nejrůznějším trikům. Ponechme například čtenáři k rozmyšlení, jak veškerá data programu uložit do konstantně mnoha paměťových buněk (viz cvičení 1.4.2 a 1.4.3).

Dodefinujeme tedy stroj tak, abychom na jednu stranu neomezili kapacitu buňky, ale na druhou stranu kompenzovali nepřirozené výhody z její neomezenosti plynoucí. Možností je mnoho, ukážeme jich tedy několik, ke každé dodáme, jaké jsou její výhody a nevýhody, a na závěr zvolíme tu, kterou budeme používat v celé knize.

Přiblížení první. Omezíme kapacitu paměťové buňky pevnou konstantou, řekněme na 32 bitů. Tím jistě odpadnou problémy s neomezenou kapacitou, lze si také představit, že aritmetické instrukce pracující s 32-bitovými čísly lze hardwarově realizovat v jednotkovém čase. Aritmetiku čísel delších než 32 bitů lze řešit funkcemi na práci s dlouhými čísly rozloženými do více paměťových buněk. Zásadní nevýhoda však spočívá v tom, že vzhledem k naší definici přístupu do paměti si omezíme počet adresovatelných paměťových buněk na 2^{32}. Současné počítače typu PC to tak sice skutečně mají, nicméně z teoretického hlediska je takový stroj nevyhovující, protože umožnuje zpracovávat pouze konstantně velké vstupy.

Přiblížení druhé. Omezíme maximální velikost čísla uložitelného v jedné paměťové buňce polynomem vzhledem k počtu paměťových buněk vstupu. Exaktně řečeno, pro každý program na tomto RAMu bude dán polynom $p(N)$ takový, že pro každý vstup velikosti N paměťových buněk smí program v libovolné paměťové buňce uložit číslo maximální velikosti $p(N)$.

Tento model odstraňuje spoustu nevýhod předchozího, většina zákeřných triků využívajících kombinaci neomezené kapacity buňky a jednotkové ceny instrukce k řešení těžkých problému nepřirozeně rychle na něm neprojde. Model má jedno omezení – pokud je počet bitů buňky nejvýše polynomiálně velký, znamená to, že

nemůžeme použít exponenciálně či více paměťových buněk, protože jich tolik prostě nenaadresujeme. Nemůžeme tedy na tomto stroji používat algoritmy s exponenciální paměťovou složitostí.

Přiblížení třetí. Zavedeme *logaritmickou cenu instrukce*. To znamená, že místo jedné časové jednotky definitoricky zavedeme, že aritmetická instrukce a logický výraz spotřebuje tolik časových jednotek, kolik je součet bitů všech operandů včetně výstupního. Cena se nazývá logaritmická proto, že počet bitů čísla je úměrný logaritmu čísla. Tedy například cena instrukce součinu čísel 3 a 8 bude $2 + 4 + 5 = 11$, protože čísla 3 a 8 mají 2 a 4 bity a výsledek 24 má 5 bitů.

Zde již omezení adresovatelného prostoru nehrozí. Prostorová komprese paměti programu do konstantně mnoha buněk je sice stále možná, je však vykoupena velkými časovými nároky instrukcí. V tom spočívá i nevýhoda modelu. Například třídíme-li zpřeházená čísla $1, \ldots, N$ do rostoucího pořadí, na zápis každého z nich potřebujeme až $\log N$ bitů, každé porovnání dvou čísel tedy bude trvat $\mathcal{O}(\log N)$ času, i nejrychlejší třídící algoritmy by tedy místo času $\mathcal{O}(N \log N)$ potřebovaly čas $\mathcal{O}(N \log^2 N)$.

Existuje také relativně obtížný problém, který na takto definovaném RAMu bude řešitelný nepřirozeně snadno a rychle – násobení dlouhých čísel. Uložíme-li obě násobená čísla do dvou paměťových buněk, postačí na jejich vynásobení jedna aritmetická instrukce a její čas bude lineární vzhledem k počtu cifer obou čísel. V této knize přitom ukazujeme netriviální algoritmus, který na stroji s omezenou kapacitou buňky a jednotkovým časem operace vyžaduje čas přibližně $\mathcal{O}(N^{1.58})$, kde N je součet počtů cifer obou čísel. Model s logaritmickou cenou instrukce tedy nezavrhujeme, ale je nekonzistentní s preferovaným výpočetním modelem této knihy.

Přiblížení čtvrté. Zavedeme *poměrnou logaritmickou cenu instrukce*. Cena aritmetické instrukce a logického výrazu nyní bude dána vztahem

$$\left\lceil \frac{b(X) + b(Y) + b(Z)}{\log_2 N} \right\rceil,$$

kde $b(X), b(Y), b(Z)$ jsou po řadě počty bitů jednotlivých vstupních i výstupních operandů a N značí počet buněk vstupu. V tomto modelu odpadnou problémy například s časem třídících algoritmů, paradoxy při násobení dlouhých čísel však přetrvávají.

Tak který model si vybrat? Jak vidíme, ať jsme navrhli jakékoli omezení paměťových buněk či zvýšení prováděcího času instrukce, vždy jsme odhalili nějaké nevýhody. Ideální model nejspíše ani neexistuje. Pro tuto knihu však hledáme referenční model, pro který půjde každý algoritmus z knihy naprogramovat se zachováním uvedené časové a paměťové složitosti.

Zvolíme tedy model číslo 2, neboli model s polynomiálně omezenou kapacitou paměťové buňky. Algoritmy používající exponenciální paměť se totiž v této knize nevyskytují a pokud by snad na exponenciálně velký prostor přišla řeč, určitě na to upozorníme a zvolíme jiný referenční model.

Naše teoretická práce je nyní u konce. Máme přesnou definici teoretického stroje RAM, pro který je přesně definována časová a paměťová složitost programů na něm běžících. Časovou složitost měříme počtem provedených instrukcí a paměťovou složitost maximálním počtem použitých paměťových buněk.

Cvičení:

1. Naprogramujte na RAMu zbývající algoritmy z oddílu 1.2.
2. Jak zakódovat libovolné množství celých čísel $c_1, \ldots, c_n$ do jednoho libovolně velkého celého čísla C tak, aby se jednotlivá čísla c_i dala jednoznačně dekódovat?
3. Navrhněte postup, jak v případě neomezené kapacity paměťové buňky pozměnit libovolný program RAMu tak, aby používal vždy jen konstantně mnoho paměťových buněk (možná za cenu časového zpomalení). Kolik nejméně buněk je potřeba?
4. Spočítejte přesně počet provedených instrukcí v předchozím vzorovém programu vzhledem k N.
5. Pokud zavedeme logaritmickou cenu instrukce, jaká bude přesná doba běhu programu?
6. Pokud zavedeme poměrnou logaritmickou cenu instrukce, jaká bude přesná doba běhu programu?

1.5. Další složitosti

Doposud jsme vždy pod pojmem časová či paměťová složitost rozuměli složitost v nejhorším možném případě vzhledem k velikosti vstupních dat. Někdy však má smysl určovat i tzv. *složitost v průměrném případě*. Funkce popisující průměrnou složitost je definována jako průměr časových (paměťových) nároků algoritmů pro určitou množinu vstupů.

Alternativní pohled na průměrnou složitost spočívá v tom, že kdybychom náhodně volili jeden vstup z jisté množiny M, potom střední hodnota časových (paměťových) nároků programu měřená přes všechny vstupy z M bude právě průměrná časová (paměťová) složitost. Například algoritmus QuickSort vykazuje v průměrném případě velmi dobré vlastnosti.

Vedle složitosti algoritmu (resp. programu) zavádíme také pojem *složitost problému*. Tento pojem vychází z teoretické představy, že máme k dispozici všechny algoritmy řešící daný problém a porovnáváme jejich složitost. Časová složitost problému je pak rovna časové složitosti nejrychlejšího z algoritmů, který problém řeší. Má tedy význam dolního odhadu složitosti S, kterého lze dosáhnout. Říká nám, že v principu nemůže existovat algoritmus, který by řešil tento problém s menší složitostí než je S, a zároveň říká, že existuje algoritmus řešící problém se složitostí S.

Stanovit složitost nějakého problému je obvykle velice obtížný úkol. Nemůžeme samozřejmě posuzovat všechny algoritmy daný problém řešící, těch je nekonečně mnoho. Odvození je třeba provádět jinou cestou.

Například složitost problému třídění prvků, které mezi sebou umíme pouze porovnávat, je dobře prostudována. Jeho složitost je $\Theta(N \log N)$, což znamená, že existuje algoritmus schopný utřídit N prvků v čase $\mathcal{O}(N \log N)$ a zároveň neexistuje asymptoticky rychlejší algoritmus.