1. Časová a prostorová složitost

V minulé kapitole jsme zkusili navrhnout algoritmy pro několik jednoduchých úloh. Zjistili jsme přitom, že pro každou úlohu existuje algoritmů více. Všechny fungují, ale jak poznat, který z nich je nejlepší? A co vlastně znamenají pojmy „lepší“ a „horší“? Kritérií kvality může být mnoho. Nás v této knize budou 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. Kdybychom chtěli svým kolegům popsat vlastnosti určitého algoritmu, jen stěží nám postačí „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 způsob, jak měřit dobu běhu algoritmu a jeho paměťové nároky nezávisle na technických podrobnostech – konkrétním stroji, jazyku, operačním systému. Těmto mírám budeme říkat *časová a prostorová složitost* algoritmu.

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. Co mají společného? My se přidržíme všeobecně uznávané definice, kterou v roce 1946 vyslovil vynikající matematik John von Neumann.

Von neumannovský počítač se skládá z pěti funkčních jednotek:

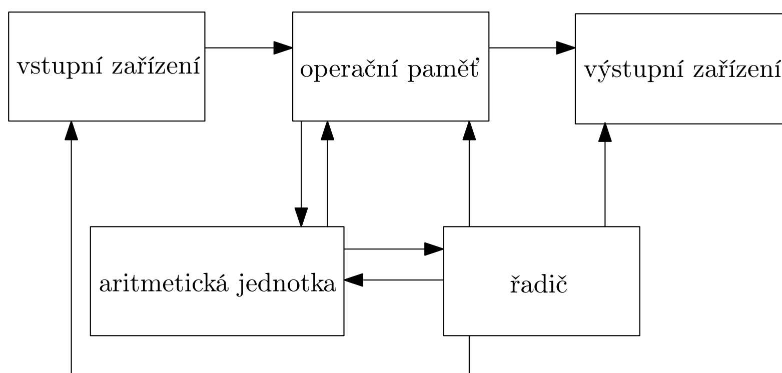
- *řídicí jednotka (řadič)* – koordinuje činnost ostatních jednotek a určuje, co mají v kterém okamžiku dělat,
- *aritmeticko-logická jednotka (ALU)* – provádí numerické výpočty, vyhodnocuje podmínky, ...,
- *operační paměť* – uchovává data a program,
- *vstupní zařízení* – zařízení, odkud se do počítače dostávají data ke zpracování,
- *výstupní zařízení* – do tohoto zařízení zapisuje počítač výsledky své činnosti.

Struktura počítače 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.⟨1⟩ Paměť je rozdělená na stejně velké *buňky,* které jsou průběžně očíslované; pomocí čísla buňky (*adresy*) se dá přečíst nebo změnit obsah buňky. V každé buňce je uloženo číslo. Všechna data i instrukce programu kódujeme pomocí čísel. Kódování a dekódování zabezpečují vhodné logické obvody v řídicí jednotce.

Po sobě jdoucí instrukce programu se nacházejí v po sobě jdoucích paměťových buňkách. Instrukcemi skoku se dá odklonit od zpracování instrukcí v uloženém pořadí. Existují následující typy instrukcí:

- aritmetické instrukce (sčítání, násobení, ukládání konstant, ...)
- logické instrukce (porovnání, AND, OR, ...)
- instrukce přenosu (z paměti do ALU a opačně, na vstup a výstup)
- podmíněné a nepodmíněné skoky
- ostatní (čekání, zastavení, ...)



Obr. 1.1: Schéma von Neumannova počítače (po šipkách tečou data a povely)

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.

⟨1⟩ Tím se liší von Neumannova architektura od architektury zvané *harvardská,* jež program a data důsledně odděluje. Výhodou von Neumannova počítače je kromě jednoduchosti i to, že program může modifikovat sám sebe. Výhodou harvardské pak možnost přistupovat k programu i datům současně.

```
#include <stdio.h>

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 řídicí jednotka mohla program provést, musíme nejdříve spustit *kompilátor* neboli *překladač*. To je nějaký jiný program, který náš program z jazyka C přeloží do takzvaného *strojového kódu*. Tedy do posloupnosti jednoduchých instrukcí kódovaných pomocí čísel, jimž už počítač přímo rozumí. Na rozdíl od původního příkladu, 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 AMD64. Tato architektura kromě běžné paměti pracuje ještě s *registry* – těch jsou řádově jednotky, také se do nich ukládají čísla a jsou přístupné rychleji než operační paměť. Můžeme si představit, že jsou uložené uvnitř aritmeticko-logické jednotky.

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:   pushq   %rbx            # uschovej registr RBX na zásobník
        xorl    %ebx, %ebx      # vynuluj registr EBX
LOOP:   movsbl  str(%rbx), %edi # ulož do EDI RBX-tý znak řetězce
        incq    %rbx            # zvyš RBX o 1
        call    putchar         # zavolej funkci putchar z knihovny
        cmpq    $13, %rbx       # už máme v RBX napočítáno 13 znaků?
        jne     LOOP            # pokud ne, skoč na LOOP
        xorl    %eax, %eax      # vynuluj EAX: nastav návratový kód 0
        popq    %rbx            # vrať do RBX obsah ze zásobníku
        ret                     # vrať se z podprogramu
STR:    .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.

1.2. Rychlost konkrétního výpočtu

Dejme tomu, že chceme změřit dobu běhu našeho příkladu „Hello world“ z předchozího oddílu. Spustíme-li ho na svém počítači několikrát, nejspíš 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 od nich?

Takový přístup se tedy hodí pouze pro testování kvality konkrétního programu na konkrétním hardwaru 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. Zatím předpokládejme, že program dostane nějaký konkrétní vstup.

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 řídicí 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 ho zapíšeme jediným příkazem, třeba při práci 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.

Několik příkladů s hvězdičkami

Než pokročíme dále, zkusme určit počet provedených operací u několika jednoduchých algoritmů. Ty si nejprve na úvod přečtou ze vstupu přirozené číslo n a pak vypisují hvězdičky. Úkol si navíc zjednodušíme: místo počítání všech operací budeme počítat jen vypsané hvězdičky. Čtenář nechť zkusí nejdříve u každého algoritmu počet hvězdiček spočítat sám, a teprve potom se podívat na náš výpočet.

Algoritmus HVĚZDIČKY1

Vstup: Číslo n

1. Pro $i = 1, \ldots, n$ opakujeme:
2.   Pro $j = 1, \ldots, n$ opakujeme:
3.   Vytiskneme $*$.

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

Algoritmus HVĚZDIČKY2
Vstup: Číslo n

1. Pro $i = 1, \ldots, n$ opakujeme:
2. Pro $j = 1, \ldots, i$ opakujeme:
3. Vytiskneme $*$.

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 řady sečteme na $n(n + 1)/2$.

Algoritmus HVĚZDIČKY3
Vstup: Číslo n

1. Dokud $n \geq 1$, opakujeme:
2. Vytiskneme $*$.
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 $\lfloor n/2^\ell \rfloor = 1$ pro neznámou ℓ. Výsledkem je tedy zhruba dvojkový logaritmus n.

Algoritmus HVĚZDIČKY4
Vstup: Číslo n

1. Dokud je $n > 0$, opakujeme:
2. Je-li n liché:
3. Pro $i = 1, \ldots, n$ opakujeme:
4. Vytiskneme $*$.
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. Spočítejme, jak vypadá 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 tohoto jednoduchého vzorce vyjde, spokojíme se alespoň s *horním odhadem* hodnoty h.⟨2⟩

⟨2⟩ Nenechte se mýlit – ač na to nevypadá, *odhad* je naprosto exaktní pojem. V matematice to znamená libovolnou nerovnost, která nějakou neznámou veličinu omezuje shora (horní odhad), nebo zdola (odhad dolní).

Označme symbolem s počet členů v součtu h. Pak můžeme tento součet upravovat následovně:

$$h = \sum_{i=0}^{s} \left\lfloor \frac{n}{2^i} \right\rfloor \leq \sum_{i=0}^{s} \frac{n}{2^i} = n \cdot \sum_{i=0}^{s} \frac{1}{2^i} \leq n \cdot \sum_{i=0}^{\infty} \frac{1}{2^i}.$$

Nejprve jsme využili toho, že $\lfloor x \rfloor \leq x$ pro každé x. Poté jsme vytkli n a přidali do řady další členy až do nekonečna, čímž součet určitě neklesl.

Jak víme z matematické analýzy, *geometrická řada* $\sum_{i=0}^{\infty} q^i$ pro jakékoliv $q \in (-1, 1)$ konverguje a má součet $1/(1-q)$. V našem případě je $q = 1/2$, takže součet řady je 2. Dostáváme, že počet vytištěných hvězdiček nebude vyšší než $2n$.

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 čtyři operace: změnu proměnné j, možná ještě změnu proměnné i a testy, zdali neskončil vnitřní či vnější cyklus. V algoritmech 2 a 3 je to velmi podobně – na vytištění jedné hvězdičky potřebujeme maximálně čtyři 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 2, aby zjistil přesně, na čem počet vytištěných hvězdiček závisí.

Který algoritmus je lepší?

Pojďme shrnout počty vykonaných kroků (nebo alespoň jejich horní odhady) našich čtyř algoritmů:

Hvězdičky1	$4n^2$
Hvězdičky2	$4n(n+1)/2 = 2n^2 + 2n$
Hvězdičky3	$4 \log_2 n$
Hvězdičky4	$8n$

Jakmile umíme počet kroků popsat dostatečně jednoduchou matematickou funkcí, můžeme předpovědět, jak se algoritmy budou chovat pro různá n, aniž bychom je skutečně spustili.

Představme si, že n je nějaké gigantické číslo, řekněme v řádu bilionů. Nejprve si všimněme, že algoritmus 3 bude nejrychlejší ze všech – i pro tak obrovské n se vykoná pouze několik málo kroků. Algoritmus 4 vykoná kroků řádově biliony. Zato algoritmy 1 a 2 budou mnohem, mnohem pomalejší.

Další postřeh se bude týkat algoritmů 1 a 2. Pro, řekněme, $n = 10^{10}$ vykoná první algoritmus $4 \cdot 10^{20}$ kroků a druhý algoritmus zhruba $2 \cdot 10^{20}$ kroků. Na první pohled se zdá, že je tedy první dvakrát pomalejší než druhý. Zdání ale klame: různé operace, které jsme považovali za elementární, mohou na skutečném počítači odpovídat různým kombinacím strojových instrukcí. A dokonce i jednotlivé strojové instrukce se mohou lišit v rychlosti.

V naší poněkud abstraktní představě o době výpočtu se takto jemné rozdíly ztrácejí. Algoritmy 1 a 2 prostě neumíme porovnat. Přesto můžeme jednoznačně říci, že oba jsou výrazně pomalejší než algoritmy 3 a 4.

Napříště tedy budeme multiplikativní konstanty v počtech operací velkoryse přehlížet. Beztak jsou strojově závislé a chování algoritmu pro velká n nijak zásadně neovlivňují. Podobně si můžeme všimnout, že ve výrazu $2n^2 + 2n$ je pro velká n člen $2n^2$ obrovský oproti $2n$, takže $2n$ můžeme klidně vynechat.

Doby výpočtu našich ukázkových algoritmů tedy můžeme popsat ještě jednoduššími funkcemi n^2, n^2, $\log_2 n$ a n, aniž bychom přišli o cokoliv zásadního.

Cvičení

1. Určete počet vytištěných hvězdiček u algoritmu Hvězdičky3 naprosto přesně, jednoduchým vzorcem.

2.* Na čem u algoritmu Hvězdičky4 závisí počet vytištěných hvězdiček? Najděte přesný vzorec, případně co nejtěsnější dolní a horní odhad.

1.3. Složitost algoritmu

Časová složitost

Už jsme se naučili, jak stanovit dobu běhu algoritmu pro konkrétní vstup. Dokonce jsme v příkladech s hvězdičkami uměli dobu běhu vyjádřit jako funkci vstupu. Málokdy to půjde tak snadno: vstup bývá mnohem složitější než jedno jediné číslo. Přesto obvykle bude platit, že pro „větší“ vstupy program poběží pomaleji než pro ty „menší“.

Pořídíme si proto nějakou *míru velikosti vstupu* a čas budeme vyjadřovat v závislosti na ní. Pokud program pro různé vstupy téže velikosti běží různě dlouho, uvážíme ten nejpomalejší případ – vždy je dobré být připraveni na nejhorší. Tím dostaneme funkci, které se říká *časová složitost* algoritmu.

Zastavme se na chvíli u toho, co si představit pod velikostí vstupu. Pokud je vstupem posloupnost čísel, obvykle za velikost považujeme jejich počet. Podobně za velikost řetězce znaků prohlásíme počet znaků. Tento přístup ale selže třeba pro Euklidův algoritmus z oddílu **??** – ten na vstupu pokaždé dostává dvě čísla a běží různě dlouho v závislosti na jejich hodnotách. Tehdy je přirozené považovat za velikost vstupu maximum ze zadaných čísel.

Vyzbrojeni předchozími poznatky, popíšeme „kuchařku“, jak určit časovou složitost daného algoritmu. Ještě to nebude poctivá matematická definice, tu si necháme na příští oddíl, ale pro téměř všechny algoritmy z této knížky poslouží stejně dobře.

1. Ujasníme si, jak se měří velikost vstupu.
2. Určíme maximální možný počet $f(n)$ elementárních operací algoritmu provedených na vstupu o velikosti n. Pokud neumíme určit počet operací přesně, najdeme alespoň co nejlepší horní odhad.
3. Ve výsledné formuli $f(n)$, která je součtem několika členů, ponecháme pouze nejrychleji rostoucí člen a ty ostatní zanedbáme, tj. vypustíme.
4. Seškrtáme *multiplikativní konstanty* – tedy ty, kterými se zbytek funkce násobí. Ale jen ty! Nikoli ostatní čísla ve vzorci.

Jak bychom podle naší kuchařky postupovali u algoritmu Hvězdičky2? Už jsme spočetli, že se vykoná nejvýše $2n^2 + 2n$ elementárních operací. Škrtneme člen $2n$ a zbude nám $2n^2$. Na závěr škrtneme multiplikativní konstantu 2. Pozor, dvojka v exponentu není multiplikativní konstanta.

Funkci $g(n)$, která zbude, nazveme *asymptotickou* 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 $\mathcal{O}(n)$.

Zde nechť si čtenář povšimne, že jsem ve výrazu $\mathcal{O}(\log n)$ vynechali základ logaritmu. Jednak je v informatické literatuře zvykem, že log bez uvedení základu je dvojkový (takové jsou nejčastější). Mnohem důležitější ale je, že logaritmy o různých základech se liší pouze konstanta-krát a konstanty přeci zanedbáváme (viz cvičení 1).

Běžné složitostní funkce

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

Jak zásadní roli hraje časová složitost algoritmu, je poznat z následující tabulky růstu funkcí (obrázek 1.2).

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
n^3	1000	10^6	10^9	10^{12}
2^n	1024	$\approx 10^{31}$	$\approx 10^{310}$	$\approx 10^{3100}$

Obr. 1.2: Chování různých funkcí

Zkusme do další tabulky (obrázek 1.3) zaznamenat odhad, jak dlouho by algoritmy s uvedenými časovými složitostmi běžely na současném počítači typu PC.

Obvyklé PC (v roce 2017) vykoná okolo 10^9 instrukcí za sekundu. Algoritmus s logaritmickou složitostí doběhne v řádu nanosekund pro jakkoliv velký vstup. I ten s lineární složitostí je ještě příjemně rychlý a můžeme čekat, že bude použitelný i pro opravdu velké vstupy. Zato kubický algoritmus se pro $n = 10\,000$ řádně zadýchá a my se na výsledek načekáme přes čtvrt hodiny.

To ale nic není proti exponenciálnímu algoritmu: pro n rovné postupně 10, 20, 30, 40, 50 poběží cca $1\,\mu$s, 1 ms, 1 s, 18 min a 13 dní. Pro $n = 100$ se už výsledku nedočkáme – až Země zanikne a hvězdy vyhasnou, program bude stále počítat a počítat.

Složitost	$n = 10$	$n = 100$	$n = 1000$	$n = 10\,000$
$\log n$	1 ns	2 ns	3 ns	4 ns
n	10 ns	100 ns	1 μs	10 μs
$n \log n$	10 ns	200 ns	3 μs	40 μs
n^2	100 ns	10 μs	1 ms	0.1 s
n^3	1 μs	1 ms	1 s	16.7 min
2^n	1 μs	10^{24} let	10^{303} let	10^{3093} let

Obr. 1.3: Přibližné doby běhu programů s různými složitostmi

Znalec trhu s hardwarem by samozřejmě mohl namítnout, že vývoj počítačů jde kupředu tak rychle, že každé dva roky se jejich výkon zdvojnásobí (tomu se říká Mooreův zákon). Jenže algoritmus se složitostí $\mathcal{O}(2^n)$ na dvakrát rychlejším počítači zpracuje ve stejném čase pouze o jedničku větší vstup. Budeme-li trpělivě čekat 20 let, získáme tisíckrát rychlejší počítač, takže zpracujeme o 10 větší vstup.

Proto se zpravidla snažíme exponenciálním algoritmům vyhýbat a uchylujeme se k nim, pouze pokud nemáme jinou možnost. Naproti tomu *polynomiální* algoritmy, tedy ty se složitostmi $\mathcal{O}(n^k)$ pro pevná konstantní k, můžeme chápat jako efektivní. I mezi nimi samozřejmě budeme rozlišovat a snažit se o co nejmenší stupeň polynomu.

Prostorová složitost

Velmi podobně jako časová složitost se dá zavést tzv. *prostorová složitost* (někdy též *paměťová 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 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

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

Průměrná složitost

Doposud jsme uvažovali takzvanou složitost v nejhorším případě: zajímalo nás, jak nejdéle může algoritmus počítat, dostane-li vstup dané velikosti. Někdy se ale stává, že výpočet obvykle doběhne rychle, pouze existuje několik málo anomálních vstupů, na nichž je pomalý. Tehdy může být praktičtější počítat *průměrnou složitost* (někdy se také říká složitost v průměrném případě). Funkce popisující tuto složitost je definována jako aritmetický průměr časových (prostorových) nároků algoritmů přes všechny vstupy dané velikosti.

Alternativně můžeme průměrnou složitost definovat pomocí teorie pravděpodobnosti. Představíme si, že budeme vstup volit náhodně ze všech vstupů dané velikosti. Potom střední hodnota časových (prostorových) nároků programu bude právě průměrná časová (prostorová) složitost.

Pravděpodobnostní analýzu algoritmů prozkoumáme v kapitole **??** a poskytne nám mnoho zajímavých výsledků.

Složitost problému

Vedle složitosti algoritmu (resp. programu) zavádíme také pojem *složitost problému.* Představme si, že pro daný problém P známe algoritmus, který ho řeší s časovou složitostí $s(n)$, a zároveň umíme dokázat, že neexistuje algoritmus, který by problém P řešil s lepší časovou složitostí než $s(n)$. Potom dává smysl říci, že *složitost problému P* je $s(n)$.

Stanovit složitost nějakého problému je obvykle velice obtížný úkol. Často se musíme spokojit pouze s horní mezí složitosti problému, odvozenou typicky popisem a analýzou vhodného algoritmu, a dolní mezí složitosti problému, odvozenou typicky nějakým matematickým argumentem.

Koncept je hezky vidět například na problému třídění prvků: dostaneme n prvků, které umíme pouze porovnávat a přesouvat, a máme je přerovnat do rostoucí posloupnosti. Tento problém je dobře prostudován: jeho složitost je řádově $n \log n$. To znamená, že existuje algoritmus schopný seřadit n prvků v čase $\mathcal{O}(n \log n)$ a zároveň neexistuje asymptoticky rychlejší algoritmus. Toto tvrzení precizně formulujeme a dokážeme v oddílu **??**.

Cvičení

1. Dokažte, že $\log_a n$ a $\log_b n$ se liší pouze konstanta-krát, přičemž konstanta závisí na a a b, ale nikoliv na n.
2. Jaká je složitost následujícího (pseudo)kódu vzhledem k n?

   Algoritmus

   1. Opakujeme, dokud $n > 0$:

2. Je-li n liché, položíme $n \leftarrow n - 1$.
3. Jinak položíme $n \leftarrow \lfloor n/2 \rfloor$.

3. Stanovte časovou a prostorovou složitost všech algoritmů z kapitoly **??**.

1.4. Asymptotická notace

Matematicky založený čtenář jistě cítí, že popis „zjednodušování“ funkcí v naší čtyřbodové kuchařce je poněkud vágní a žádá si exaktní definice. Pojďme se do nich pustit.

Definice: Nechť $f, g : \mathbb{N} \to \mathbb{R}$ jsou dvě funkce. Řekneme, že funkce $f(n)$ *je třídy* $\mathcal{O}(g(n))$, jestliže existuje taková kladná reálná konstanta c, že pro *skoro všechna* n platí $f(n) \leq cg(n)$. Skoro všemi n se myslí, že nerovnost může selhat pro konečně mnoho výjimek, tedy že existuje nějaké přirozené n_0 takové, že nerovnost platí pro všechna $n \geq n_0$. Funkci $g(n)$ se pak říká *asymptotický horní odhad* funkce $f(n)$.⟨3⟩

Jinými slovy, dostatečně velký násobek funkce $g(n)$ shora omezuje funkci $f(n)$. Konečně mnoho výjimek se hodí tehdy, má-li funkce $g(n)$ několik počátečních funkčních hodnot nulových či dokonce záporných, takže je nemůžeme „přebít“ jakkoliv vysokou konstantou c.

Poněkud formálněji bychom se na zápis $\mathcal{O}(g)$ mohli dívat jako na množinu všech funkcí f, které splňují uvedenou definici. Pak můžeme místo „funkce f je třídy $\mathcal{O}(g)$“ psát prostě $f \in \mathcal{O}(g)$. Navíc nám to umožní elegantně zapisovat i různé vztahy typu $\mathcal{O}(n) \subseteq \mathcal{O}(n^2)$.

Ve většině informatické literatury se ovšem s $\mathcal{O}$-čkovou notací zachází mnohem nepořádněji: často se píše „f je $\mathcal{O}(g)$“, nebo dokonce $f = \mathcal{O}(g)$. I my si občas takové zjednodušení dovolíme. Stále ale mějme na paměti, že se nejedná o žádnou rovnost, nýbrž o nerovnost (horní odhad).

Zbývá nahlédnout, že instrukce naší čtyřbodové „kuchařky“ jsou důsledky právě vyslovené definice. Čtvrtý bod nás nabádá ke škrtání multiplikativních konstant, což definice $\mathcal{O}$ přímo dovoluje. Třetí bod můžeme formálně popsat takto:

Lemma: Nechť $f(n) = f_1(n) + f_2(n)$ a $f_1(n) \in \mathcal{O}(f_2(n))$. Pak $f(n) \in \mathcal{O}(f_2(n))$.

Důkaz: Z předpokladu víme, že $f_1(n) \leq cf_2(n)$ platí skoro všude pro vhodnou konstantu c. Proto je také skoro všude $f_1(n) + f_2(n) \leq (1 + c) \cdot f_2(n)$, což se jistě vejde do $\mathcal{O}(f_2(n))$. □

Pozor na to, že vyjádření složitosti pomocí $\mathcal{O}$ může být příliš 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 nám bude hodit také obdobné značení pro asymptotický dolní odhad a „asymptotickou rovnost“.

⟨3⟩ Proč se tomuto odhadu říká asymptotický? V matematické analýze se zkoumá asymptota funkce, což je přímka, jejíž vzdálenost od dané funkce se s rostoucím argumentem zmenšuje a v nekonečnu se dotýkají. Podobně my zde zkoumáme chování funkce pro n blížící se k nekonečnu.

Definice: Mějme dvě funkce $f, g : \mathbb{N} \to \mathbb{R}$. Řekneme, že funkce $f(n)$ je třídy $\Omega(g(n))$, jestliže existuje taková kladná reálná konstanta c, že $f(n) \geq cg(n)$ pro skoro všechna n. Tomu se říká *asymptotický dolní odhad.*

Definice: Řekneme, že funkce $f(n)$ je třídy $\Theta(g(n))$, jestliže $f(n)$ je jak třídy $\mathcal{O}(g(n))$, tak třídy $\Omega(g(n))$.

Symboly Ω a Θ mohou opět značit i příslušné množiny funkcí. Pak jistě platí $\Theta(g) = \mathcal{O}(g) \cap \Omega(g)$.

Příklad: O našich ukázkových algoritmech 1, 2, 3, 4 můžeme říci, že mají složitosti po řadě $\Theta(n^2)$, $\Theta(n^2)$, $\Theta(\log n)$ a $\Theta(n)$.

Při skutečném srovnávání algoritmů by tedy bylo lepší zapisovat složitost pomocí Θ, nikoliv podle $\mathcal{O}$. To by zajisté poskytlo úplnější informaci o chování funkce. Ne vždy se nám to ale povede: analýzou algoritmu mnohdy dostáváme pouze horní odhad počtu provedených instrukcí nebo potřebných paměťových míst. Například se nám může stát, že v algoritmu je několik podmínek a nedovedeme určit, které z jejich možných kombinací mohou nastat současně. Raději tedy předpokládáme, že nastanou všechny, čímž dostaneme horní odhad.

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. Nalezněte co nejvíce asymptotických vztahů mezi těmito funkcemi: n, $\log n$, $\log\log n$, $\sqrt{n}$, $n^{\log n}$, 2^n, $n^{3/2}$, $n!$, n^n.
2. Nahlédněte, že $f(n) \in \Theta(g(n))$ by se dalo ekvivalentně definovat tak, že pro vhodné konstanty $c_1, c_2 > 0$ platí $c_1 g(n) \leq f(n) \leq c_2 f(n)$ pro skoro všechna n.
3. Dokažte, že $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(\max(f(n), g(n)))$ pro $f, g \geq 0$.
4. Dokažte, že $n \log n \notin \mathcal{O}(n)$.
5. Dokažte, že $\log n \in \mathcal{O}(n^\varepsilon)$ pro každé $\varepsilon > 0$.
6. Najděte co nejlepší asymptotický odhad funkce $\log_n(n!)$.
7. Najděte funkce f a g takové, že neplatí ani $f = \mathcal{O}(g)$, ani $g = \mathcal{O}(f)$.

1.5. 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.

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

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

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

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

Na začátku výpočtu obsahuje paměť v určených buňkách vstup a obsah ostatních buněk je nedefinován. Potom je program sekvenčně prováděn, instrukci za instrukcí. Po zastavení programu je obsah smluvených míst v 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. Typická instrukce přečte jeden nebo dva operandy z paměti, něco s nimi spočítá a výsledek opět uloží do paměti. Operandem instrukce může být:

- *literál* – konstanta zakódovaná přímo v instrukci. Literály zapisujeme jako čísla v desítkové soustavě.
- *přímo adresovaná buňka* – číslo uložené v paměťové buňce, jejíž adresa je zakódovaná v instrukci. Zapisujeme jako `[adresa]`, kde `adresa` je libovolné celé číslo.

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

- *nepřímo adresovaná buňka* – číslo uložené v buňce, jejíž adresa je v přímo adresované buňce. Píšeme `[[adresa]]`.

Operandy tedy mohou být například `42`, `[16]`, `[-3]` nebo `[[16]]`, ale nikoliv `[[[5]]]` ani `[3*[5]]`. Svůj výsledek může instrukce uložit do přímo či nepřímo adresované paměťové buňky.

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 přiřazeny přezdívky `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]]`. Registry lze použít například jako úložiště často užívaných pomocných proměnných.

Nyní vyjmenujeme instrukce stroje. X, Y a Z vždy představují některý z výše uvedených výrazů pro přístup do paměti či registrů, Y a Z mohou být navíc i konstanty.

- Aritmetické instrukce:
  - Přiřazení: X `:=` Y
  - Negace: X `:= -`Y
  - Sčítání: X `:=` Y `+` Z
  - Odčítání: X `:=` Y `-` Z
  - Násobení: X `:=` Y `*` Z
  - Celočíselné dělení: X `:=` Y `/` Z
    Číslo Z musí být nenulové. Zaokrouhlujeme vždy k nule, takže (-1) `/` $3 = 0 =$ `-`$(1$ `/` $3)$.
  - Zbytek po celočíselném dělení: X `:=` Y `%` Z
    Číslo Z musí být kladné. Dodržujeme, že $(Y$ `/` $Z)$ `*` Z `+` $(Y$ `%` $Z) = Y$, takže (-1) `%` $3 = -1$.
- Logické instrukce:
  - Bitová konjunkce (AND): X `:=` Y `&` Z
    i-tý bit výsledku je 1 právě tehdy, když jsou jedničkové i-té bity obou operandů. Například 12 `&` $5 = (1100)_2$ `&` $(0101)_2 = (0100)_2 = 4$.
  - Bitová disjunkce (OR): X `:=` Y `|` Z
    i-tý bit výsledku je 1, pokud je jedničkový i-tý bit aspoň jednoho operandu. Například 12 `|` $5 = (1100)_2$ `|` $(0101)_2 = (1101)_2 = 13$.
  - Bitová nonekvivalence (XOR): X `:=` Y `^` Z
    i-tý bit výsledku je 1, pokud je jedničkový i-tý bit právě jednoho operandu. Například 12 `^` $5 = (1100)_2$ `^` $(0101)_2 = (1001)_2 = 9$.

- Bitový posun doleva: X `:=` Y `<<` Z
  Doplnění Z nul na konec binárního zápisu čísla Y. Například 11 `<<` 3 $= (1011)_2$ `<<` $3 = (1011000)_2 = 88$.
- Bitový posun doprava: X `:=` Y `>>` Z
  Smazání posledních Z bitů binárního zápisu čísla Y. Například 11 `>>` 2 $= (1011)_2$ `>>` $2 = (10)_2 = 2$.

- Řídicí instrukce:
  - Ukončení výpočtu: `halt`
  - Nepodmíněný skok: `goto` *label*, kde *label* je návěští, které se definuje napsáním *label*: před instrukci.
  - Podmíněný příkaz: `if` *podmínka* `then` *instrukce*, přičemž *instrukce* je libovolná instrukce kromě podmíněného příkazu a *podmínka* 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. Doba běhu programu, jterou používáme v naší definici časové složitosti, je tedy rovna celkovému počtu provedených instrukcí.

S měřením spotřebované paměti musíme být trochu opatrnější, protože program by mohl využít malé množství buněk rozprostřených po obrovském prostoru. Budeme tedy měřit rozdíl mezi nejvyšším a nejnižším použitým indexem paměti.

Časovou a paměťovou složitost pak definujeme zavedeným způsobem jako maximum ze spotřeby času a paměti přes všechny vstupy dané velikosti. Roli velikosti vstupu obvykle hraje počet paměťových buněk obsahujících vstup.

Upozorňujeme, že do časové složitosti nepočítáme dobu potřebnou na načtení vstupu – podle naší konvence je vstup při zahájení výpočtu už přítomen v paměti. Můžeme tedy studovat i algoritmy s lepší než lineární časovou složitostí, například binární vyhledávání z oddílu **??**. Pokud navíc program do vstupu nebude zapisovat, nebudeme paměť zabranou vstupem ani počítat do spotřebovaného prostoru.

Příklad programu pro RAM

Pro ilustraci přepíšeme algoritmus Hvězdičky2 z předchozích oddílů co nejvěrněji do programu pro náš RAM. Připomeňme tento algoritmus:

Algoritmus Hvězdičky2
Vstup: Číslo n

1. Pro $i = 1, \ldots, n$ opakujeme:
2. Pro $j = 1, \ldots, i$ opakujeme:
3. Vytiskneme `*`.

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).

```
          I := 1
          Z := 1
VNEJSI:   if I > [0] then halt
          J := 1
VNITRNI:  if J > I then goto DALSI
          [Z] := 1
          Z := Z + 1
          J := J + 1
          goto VNITRNI
DALSI:    I := I + 1
          goto VNEJSI
```

Registry `I` a `J` odpovídají stejnojmenným proměnným algoritmu, registr `Z` ukazuje na buňku paměti, kam zapíšeme příští hvězdičku.

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 (cvičení 2 a 3) a pomocí této „komprese“ programy absurdně zrychlovat (cvičení 9).

Proto upravíme stroj tak, abychom na jednu stranu neomezili kapacitu buňky příliš, ale na druhou stranu kompenzovali nepřirozené výhody plynoucí z její neomezenosti. 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 64 bitů. Tím jistě odpadnou problémy s neomezenou kapacitou, lze si také představit, že aritmetické instrukce pracující s 64-bitovými čísly lze hardwarově realizovat v jednotkovém čase. Aritmetiku čísel delších než 64 bitů lze řešit funkcemi na práci s dlouhými čísly rozloženými do více paměťových buněk. Zásadní nevýhoda však spočívá v tom, že jsme omezili i adresy paměťových buněk. Každý program proto může použít pouze konstantní množství paměti: 2^{64} buněk. Současné počítače typu PC to tak sice skutečně mají, nicméně z teoretického hlediska je takový stroj nevyhovující, protože umožňuje zpracovávat pouze konstantně velké vstupy.

Přiblížení druhé. Abychom mohli adresovat libovolně velký vstup, potřebujeme čísla o alespoň $\log n$ bitech, kde n je velikost vstupu. Omezíme tedy velikost čísla v jedné paměťové buňce na $k \cdot \log n$ bitů, kde k je libovolná konstanta. Hodnota čísla pak musí být menší než $2^{k \log n} = 2^{k \log n} = (2^{\log n})^k = n^k$. Jinými slovy, hodnoty čísel jsme omezili nějakým polynomem ve velikosti vstupu.

Tento model odstraňuje spoustu nevýhod předchozího: většina zákeřných triků využívajících kombinaci neomezené kapacity buňky a jednotkové ceny instrukce

k nepřirozeně rychlému počítání na něm neuspěje. Model má jedno omezení – pokud je velikost čísla v buňce nejvýše polynomiální, znamená to, že nemůžeme použít exponenciálně či více paměťových buněk, protože jich tolik zkrátka nenaadresujeme. Nemůžeme tedy na tomto stroji používat algoritmy s exponenciální paměťovou složitostí. Ty jsou sice málokdy praktické, ale například v oddílu **??** se nám budou hodit.

Přiblížení třetí. Abychom neomezili množství paměti, potřebujeme povolit libovolně velká čísla. Vzdejme se tedy naopak předpokladu, že všechny instrukce trvají stejně dlouho. Zavedeme *logaritmickou cenu instrukce.* To znamená, že jedna instrukce potrvá tolik jednotek času, kolik je součet velikostí všech čísel, s nimiž pracuje, měřený v bitech. To zahrnuje operandy, výsledek i adresy použitých buněk paměti. Cena se nazývá logaritmická proto, že počet bitů čísla je úměrný logaritmu čísla. Tedy například instrukce `[5] := 3*8` bude mít cenu $2+4+5+3 = 14$ jednotek času, protože čísla 3 a 8 mají 2 a 4 bity, výsledek 24 zabere 5 bitů a ukládá se na adresu 5 zapsanou 3-bitovým číslem.

Je sice stále možné uložit veškerá data programu do konstantně mnoha buněk, ale instrukce s takovými čísly pracují výrazně pomaleji. V tom spočívá i nevýhoda modelu. I jednoduché algoritmy, které původně měly evidentně lineární časovou složitost, najednou poběží pomaleji – například vypsání n hvězdiček potrvá $\Theta(n \log n)$, jelikož i obyčejné zvýšení řídicí proměnné cyklu zabere čas $\Theta(\log n)$. To je poněkud nepohodlné a skutečné počítače se tak nechovají.

Přiblížení čtvrté. Pokusme se o kompromis mezi předchozími dvěma modely. Zavedeme *poměrnou logaritmickou cenu instrukce.* To bude logaritmická cena vydělená logaritmem velikosti vstupu a zaokrouhlená nahoru. Dokud tedy budeme pracovat s polynomiálně velkými čísly (tedy o řádově logaritmickém počtu bitů), poměr bude shora omezen konstantou a budeme si moci představovat, že všechny instrukce mají konstantní ceny. Jakmile začneme pracovat s většími čísly, cena instrukcí odpovídajícím způsobem poroste.

Poměrné logaritmické ceny se tedy chovají intuitivně, neomezují adresovatelný prostor a odstraňují paradoxy původního modelu. Všechny algoritmy v této knize tedy budeme analyzovat v tomto modelu. Navíc použijeme podobný model i pro měření spotřebované paměti: jedna buňka paměti zabere tolik prostoru, kolik je počet bitů potřebných na reprezentaci její adresy a hodnoty, relativně k logaritmu velikosti vstupu. Započítáme všechny buňky mezi minimální a maximální adresou, na níž program přistoupil.

Naše teoretická práce je nyní u konce. Máme přesnou definici teoretického stroje RAM, pro který je přesně definována časová a prostorová složitost programů na něm běžících. Můžeme se pustit do analýzy konkrétních algoritmů.

Cvičení

1. Naprogramujte na RAMu zbývající algoritmy z oddílu 1.2 a z úvodní kapitoly.
2. Mějme RAM s neomezenou velikosti čísel. Vymyslete, jak zakódovat libovolné množství celých čísel $c_1, \ldots, c_n$ do jednoho 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 na RAMu tak, aby používal jen konstantně mnoho paměťových buněk. Program můžete libovolně zpomalit. Kolik nejméně buněk je potřeba?

4. Spočítejte přesně počet provedených instrukcí naší RAMové verze algoritmu Hvězdičky2. Vyjádřete jej jako funkci proměnné n.

5. Pokračujme v předchozím cvičení: jaká bude přesná doba běhu programu, zavedeme-li logaritmickou, případně poměrnou logaritmickou cenu instrukce?

6. Rozmyslete si, jak do instrukcí RAMu překládat konstrukce známé z vyšších programovacích jazyků: podmínky, cykly, volání podprogramů s lokálními proměnnými a rekurzí.

7. Vymyslete, jak na RAMu prohodit obsah dvou paměťových buněk, aniž byste použili jakoukoliv jinou buňku.

8.* Vymyslete, jak na RAMu v konstantním čase otestovat, zda je číslo mocninou dvojky.

9. Mějme RAM s jednotkovou cenou instrukce a neomezenou velikostí čísel. Ukažte, jak v čase $\mathcal{O}(n)$ zakódovat vektor n přirozených čísel tak, abyste z kódů uměli v konstantním čase vypočítat skalární součin dvou vektorů. Z toho odvoďte algoritmus pro násobení matic $n \times n$ v čase $\mathcal{O}(n^2)$.

10. *Interaktivní RAM:* Na naší verzi RAMu dostanou všechny programy hned na začátku celý vstup, pak nějakou dobu počítají a nakonec vydají celý výstup. Někdy se hodí dostávat vstup po částech a průběžně na něj reagovat. Navrhněte rozšíření RAMu, které to umožní.

11. *Registrový stroj* je ještě jednodušší model výpočtu. Disponuje konečným počtem registrů, každý je schopen pojmout jedno přirozené číslo. Má tři instrukce: `inc` pro zvýšení hodnoty registru o 1, `dec` pro snížení o 1 (snížením nuly vyjde opět nula) a `jmpeq` pro skok, pokud se hodnoty dvou registrů rovnají. Vymyslete, jak na registrovém stroji naprogramovat vynulování registru, zkopírování hodnoty z jednoho registru do druhého a vynásobení dvou registrů.

12.* Mějme program pro RAM, jehož vstupem a výstupem je konstantně mnoho čísel (z cvičení 3 víme, že libovolný vstup lze takto zakódovat). Ukažte, jak takový program přeložit na program pro registrový stroj, který počítá totéž. Časová složitost se překladem může libovolně zhoršit.