

Kapitola 1

Úvod

1.1 Čo je to Computer Science?

Namiesto úvodu pár myšlienok z úvodu knihy Juraja Hromkoviča: *Theoretical Computer Science – An Introduction to Automata, Computability, Complexity, Algorithmics, Randomization, Communication, and Cryptography*

Na označenie vedy, ktorej sa venuje tento materiál, sa najčastejšie používa označenie Informatika (z anglického computer science, resp. informatics). Každý, kto študuje alebo sa zaoberá touto vednou disciplínou, by sa z času na čas mal zamyslieť nad tým, ako by zdefinoval informatiku, mal by premýšľať o jej prínose pre vedu, vzdelávanie, každodenný život. Je dôležité, aby sme si uvedomili, že získavanie čoraz väčšieho množstva poznatkov o vedeckej disciplíne, prehlbovanie pochopenia jej podstaty vždy vedie k vývoju nášho názoru na úlohu tejto vedy v kontexte všetkých vedeckých disciplín. Je preto najmä pre študentov nesmierne dôležité, aby si neustále premietali svoje vnímanie informatiky ako vedy. Trúfneme si vyprovokovať konflikt medzi vašim terajším vnímaním informatiky a stanoviskom, prezentovaným v tomto úvode; podnietime diskusiu, ktorá môže viesť k vývoju vášho chápania informatiky.

Pokúsme sa najskôr zodpovedať otázku

"Čo je to informatika?"

Je ťažké poskytnúť exaktnú a úplnú definíciu tejto vedeckej disciplíny. Všeobecne akceptovaná definícia je nasledovná:

"Informatika je náuka o algoritmickej spracovávaní, reprezentácii, ukladaní a prenose informácií"

Podľa tejto definície sú hlavnými objektami výskumu informatiky ako vedeckej disciplíny informácia a algoritmus. Táto definícia však zanedbáva úplné odhalenie podstaty a metodológie informatiky. Ďalšou otázkou o podstate informatiky je

"Ku ktorej vedeckej disciplíne informatika patrí? Je to metadisciplína ako matematika a filozofia, prírodná veda alebo inžinierska disciplína?"

Odpoveď na túto otázku slúži nielen na identifikáciu objektu výskumu, musí tiež určiť metodológiu a príspevok informatiky ako vedy. Odpoveďou je, že informatika nemôže byť jednoznačne priradená k žiadnej z týchto disciplín. Informatika v sebe zahŕňa aspekty matematiky, prírodných vied ale aj inžinierskych disciplín. V krátkosti vysvetlíme, prečo.

Podobne ako filozofia a matematika, informatika skúma všeobecné kategórie, ako *determinizmus, nedeterminizmus, náhodnosť, informácia, pravda, nepravda, zložitosť, jazyk, dôkaz, znalosť, komunikácia, aproximácia, algoritmus, simulácia, atď.*

a prispieva k ich pochopeniu. Informatika vrhla nové svetlo na tieto kategórie, priniesla nový význam mnohým z nich.

Prírodná veda, na rozdiel od filozofie a matematiky, skúma konkrétne prírodné objekty a procesy, určuje hranicu medzi možným a nemožným, skúma kvantitatívne vzťahy prírodných procesov. Modeluje, analyzuje a dokazuje vierohodnosť hypotetických modelov experimentmi. Tieto aspekty sú bežné aj v informatike. Objektami sú informácie a algoritmy (programy, počítače) a skúmanými procesmi sú (reálne existujúce) výpočty. Presvedčíme sa o tom sledovaním vývoja informatiky. Historicky prvou dôležitou výskumnou otázkou bolo:

"Existujú dobre definované problémy, ktoré nemôžu byť riešené automaticky (počítačom, bez ohľadu na silu súčasných a budúcich počítačov)?"

Snaha o zodpovedanie tejto otázky viedla k základom informatiky ako nezávislej vedy. Odpoveď na otázku je kladná. V súčasnosti sme si o mnohých praktických problémov, ktoré by sme radi riešili algoritmicky, vedomí toho, že algoritmicky riešiteľné nie sú. Tento záver je založený na čisto matematickom dôkaze algoritmickej neriešiteľnosti (inými slovami, na dôkaze neexistencie algoritmu riešiaceho daný problém) a nie na tom, že sa žiadne algoritmické riešenie doteraz nenašlo.

Po tom, ako bola vyvinutá metóda na klasifikáciu problémov podľa ich riešiteľnosti, začali si ľudia klásť nasledujúcu vedeckú otázku:

"Ako ťažké sú konkrétne algoritmické problémy?"

Pritom obtiažnosť nemeríme obtiažnosťou nájsť algoritmické riešenie, či veľkosťou vytvoreného programu. Obtiažnosť meriame množstvom práce potrebnej a postačujúcej k tomu, aby sme pre daný vstup algoritmicky vypočítali riešenie. Dozvedáme sa o existencii ťažkých problémov, na riešenie ktorých treba energiu presahujúcu energiu celého vesmíru. Existujú také algoritmicky riešiteľné problémy, pre ktoré by výpočet ľubovoľného programu na ich riešenie vyžadoval viac času ako uplynul od "Veľkého tresku". Takže číra existencia programu na riešenie nejakého problému ešte nezaručuje, že je tento problém prakticky riešiteľný.

Pokusy o klasifikáciu problémov na prakticky riešiteľné (tractable) a prakticky neriešiteľné viedli k najfascinujúcejším vedeckým objavom teoretickej informatiky.

Ako príklad si vezmime pravdepodobnostné (randomized) algoritmy. Väčšina programov (algoritmov) tak, ako ich poznáme, je deterministická. Determinizmom rozumíme to, že program a vstup úplne určujú všetky kroky spracovávania problému. V každom okamihu je nasledujúca akcia programu jednoznačne určená a závisí iba od momentálnych údajov. Pravdepodobnostné algoritmy môžu nasledujúcu akciu programu vyberať z viacerých možností. Prácu pravdepodobnostného algoritmu si možno predstaviť tak, že algoritmus z času na čas hádže mincou, aby určil nasledujúci krok, napr. vybral nasledujúcu stratégiu pri hľadaní korektnej odpovede. Pravdepodobnostný program tak môže mať pre jeden vstup niekoľko rôznych výpočtov. Na rozdiel od deterministických programov, ktoré vierohodne vrátia pre každý vstup správny výsledok, pravdepodobnostné programy môžu poskytnúť aj nesprávny výsledok. Cieľom je znižovať/potláčať pravdepodobnosť takýchto nesprávnych výsledkov, čo za určitých podmienok znamená znižovať počet nesprávnych výpočtov.

Na prvý pohľad sa pravdepodobnostné programy môžu zdať, na rozdiel od de-

terministických programov, nespoľahlivé. Prečo ich teda potrebujeme? Existuje veľa reálnych problémov, ktorých riešenie najlepšimi známymi algoritmami vyžaduje viac počítačovej práce ako je realisticky možné. Takéto problémy sú prakticky neriešiteľné. Môže sa však stať zázrak: tým zázrakom môže byť pravdepodobnostný algoritmus, ktorý rieši problém za niekoľko minút s minimálnou pravdepodobnosťou chyby jednej trilióntiny. Môžeme takýto program zavrhnúť ako nespoľahlivý? Deterministický program, ktorého výpočet trvá niekoľko dní, je menej spoľahlivý ako pravdepodobnostný program bežiaci niekoľko minút, pretože pravdepodobnosť výskytu hardverovej chyby počas 24 hodín je oveľa väčšia ako pravdepodobnosť chyby rýchleho pravdepodobnostného programu. Konkrétnym prípadom prakticky nesmierne dôležitého problému je testovanie prvočíselnosti. Vo všadeprítomných kryptografických systémoch založených na verejných kľúčoch je nevyhnutné, aby sa generovali veľké (500 ciferné) prvočísla. Prvé deterministické algoritmy na testovanie prvočíselnosti boli založené na deliteľnosti vstupu n . Už samotný počet prvočísel, menších ako \sqrt{n} , je pre tak veľké čísla väčší, ako počet protónov in the universe. Takéto deterministické algoritmy sú teda prakticky nepoužiteľné. Nedávno sa objavil nový deterministický algoritmus na testovanie prvočíselnosti, ktorého zložitosť je $O(m^{12})$, kde m je dĺžka binárneho zápisu čísla n . Na testovanie 500ciferného čísla však tento algoritmus vyžaduje vykonanie viac ako 10^{32} počítačových inštrukcií; ani na najrýchlejších počítačoch by čas od Veľkého tresku nebol dostatočný na realizáciu tohto výpočtu. Máme však niekoľko randomized algoritmov pomocou ktorých môžeme otestovať prvočíselnosť tak veľkých čísel na bežných PC v priebehu niekoľkých minút, dokonca sekúnd.

Iným exemplárnym príkladom je komunikačný protokol pre porovnanie obsahu dvoch databáz, ktoré sú uložené na dvoch vzdialených počítačoch. Matematicky sa dá dokázať, že každý deterministický protokol, ktorý testuje ekvivalenciu ich obsahov, vyžaduje, aby sa vymenilo toľko bitov, koľko ich je uložených v databázach. Pre databázu veľkosti 10^{16} by to bolo únavné. Pravdepodobnostným komunikačným protokolom môžeme testovať ekvivalenciu dvoch databáz výmenou správy dĺžky približne 2000 bitov. Pravdepodobnosť chyby je pritom menšia ako jedna trilióntina.

Ako je to možné? Bez využitia základných znalostí informatiky sa to vysvetľuje ťažko. Hľadanie vysvetlenia sily pravdepodobnostných algoritmov je fascinujúci výskumný projekt, ktorý zachádza do najhlbších základov matematiky, filozofie a prírodných vied. Príroda je náš najlepší učiteľ a náhoda hrá v prírode oveľa dôležitejšiu úlohu, ako si dokážeme pripustiť. Informatici môžu vymenovať mnoho systémov, ktorých požadované charakteristiky a správanie sa dajú dosiahnuť iba pomocou randomizácie. V takýchto prípadoch sú všetky dostupné deterministické systémy zložené z miliónoch podsystemov, ktoré musia interagovať korektne. Tak komplexný systém, vysoko závislý od veľkého počtu komponent, je nepraktický. V prípade výskytu chyby by bolo takmer nemožné ju odhaliť. Netreba ani hovoriť, že cena za vývoj takéhoto systému by bola tiež astronomická. Na druhej strane môžeme skonštruovať malý randomized systém s požadovanými vlastnosťami. Vzhľadom na malú veľkosť sú takéto systémy lacné a funkčnosť ich komponent sa ľahko testuje. Kľúčovým pritom ostáva, že pravdepodobnosť chyby takého systému je tak minimálna, až je zanedbateľná.

Popri doteraz prezentovaných vedeckých aspektoch je pre mnohých vedcov informatika typicky problémovo orientovaná a prakticky inžinierska disciplína. Informatika nielenže zahŕňa technické aspekty ako:

organizácia procesov (fázy, milestones, dokumentácia), formulácia strategických cieľov a obmedzení, modelovanie, popis, špecifikácia, zabezpečenie kvality, testovanie, integrácia do existujúcich systémov, viacnásobné použitie, podporné nástroje,

Zahŕňa tiež aspekty manažmentu ako:

organizácia a vedenie tímu, odhad cien, plánovanie, produktivita, manažment kvality, odhad časových plánov a termínov, product release, podmienky kontraktov a marketing.

Informatici by mali byť aj naozaj pragmatickými praktikmi. Pri konštrukcii komplexného softverového alebo hardverového systému musí človek často robiť rozhodnutia na základe vlastnej skúsenosti, pretože nemá možnosť modelovať a analyzovať komplexnú realitu.

Na základe našej definície informatiky možno nadobudnúť dojem, že štúdium informatiky je príliš obtiažne. Človek potrebuje matematické vedomosti, chápanie uvažovania v prírodných vedách a navyše, schopnosť pracovať ako inžinier. Toto naozaj môžu byť vysoké požiadavky, je to však tiež veľká výhoda tohto typu vzdelania. Hlavnou nevýhodou súčasnej vedy je jej vysoká špecializácia, ktorá vedie k vývoju malých nezávislých disciplín. Každá z nich si buduje svoj vlastný jazyk, ktorý je často nezrozumiteľný dokonca aj pre vedcov z príbuznej oblasti. Zašlo to tak ďaleko, že spôsob štandardnej argumentácie jednej oblasti sa považuje za povrchný a neprípustný v inej oblasti. Spomaľuje to vývoj interdisciplinárneho výskumu. A informatika je v svojej podstate interdisciplinárna. Sústreďuje sa na hľadanie riešenia problémov všetkých oblastí vedy a každodenného života; všade tam, kde je predstaviteľné použitie počítača. Súčasne s tým vyvíja široké spektrum metód, počnúc precíznymi matematickými metódami, končiac inžinierskym "know-how", založenom na skúsenosti. Možnosť súbežného učenia sa rôznych jazykov rôznych oblastí a rôznym spôsobom uvažovania v jednej disciplíne, to je najcennejší dar, ktorý študenti informatiky dostávajú.

1.2 Fascinujúca teória

Kniha je úvodom do základov teoretickej informatiky. Teoretická informatika je fascinujúca vedecká disciplína. Svojimi veľkolepými výsledkami a vďaka svojej veľkej interdisciplinarite prispela veľkým dielom k nášmu pohľadu na svet. Ako by štatistiky potvrdili, nie je teoretická informatika najobľúbenejším predmetom študentov.

Niekoľko dôvodov pre jej štúdium:

filozofická hĺbka

Existujú problémy, ktoré sú algoritmicke neriešiteľné? Ak áno, kde leží hranica medzi riešiteľnými a neriešiteľnými problémami?

Sú nedeterministické a náhodou riadené procesy schopné riešiť viac ako deterministické?

Ako definujeme obtiažnosť(zložitosť) problému?

Kde sú hranice praktickej riešiteľnosti?

Čo je to matematický dôkaz? Je ťažšie algoritmicke nájsť dôkaz alebo algoritmicke overiť jeho platnosť?

Ako definujeme náhodné objekty/náhodnosť?

Bez pojmov teoretickej informatiky by sme tie problémy nemohli ani poriadne sformulovať, ani na ne dať odpoveď.

*aplikovateľnosť
a veľkolepé
výsledky*

TI súvisí s praxou. Poskytuje metodológie, ktoré aplikujeme pri počiatočnom návrhu, ale aj tie, ktoré využijeme počas celého procesu návrhu, dokazovania, implementácie. Existuje nemálo optimalizačných úloh, kde relaxácia požiadaviek vedie k efektívnejšiemu riešeniu. Veríte, že možno niekoho presvedčiť o znalosti tajného hesla bez toho, aby sme ho povedali? Veríte, že dve osoby môžu zistiť, kto je starší bez toho, aby prezradili svoj vek? Veríte, že môžeme overiť platnosť niekoľko tisíc

stránkového dôkazu bez toho, aby sme ho celý čítali, ale videli iba niektoré jeho náhodne zvolené úseky? Všetko toto sa dá...

Zatiaľ čo zhruba polovica vedomostí o software a hardware je po 5 rokoch neaktuálna, metodologické výsledky TI pretrvávajú niekoľko desiatok rokov...

živostnosť vedomostí

TI je vo svojej podstate interdisciplinárna, nachádzame jej uplatnenie v mnohých iných oblastiach - genetika, medicínska diagnostika, optimalizácia v ekonomických a technických disciplínach, automatické rozpoznávanie reči, prehľadávanie priestoru,...

interdisciplinarita

Nie je to jednostranná spolupráca. TI tiež profituje z tých iných disciplín: kvantová fyzika a kvantové počítače; DNA výpočty; kalenie a metóda simulovaného žihania;...

TI podporuje vytváranie a analyzovanie matematických modelov, vytváranie pojmov a metodológií na riešenie problémov.

spôsob myslenia

1.3 A teraz už môj úvod

Pri pohľade dozadu si v súvislosti s efektívnou riešiteľnosťou problémov môže všimnúť niekoľko medzníkov

- Významným krokom bolo formalizovanie pojmu *algoritmus* zhruba v 30-tych rokoch minulého storočia. Vďačíme za to definovaniu *abstraktného* výpočtového modelu - *Turingovho stroja* (TS), ktorý bol všeobecne akceptovaný ako "definícia" pojmu algoritmus. Vďaka tomuto pojmu dochádza k deleniu problémov na *riešiteľné* a *neriešiteľné*.
- Po vymedzení triedy riešiteľných problémov sa pozornosť presúva k vymedzeniu triedy *prakticky riešiteľných* problémov. Praktická riešiteľnosť sa (väčšinou) vzťahuje na časové nároky potrebné na riešenie toho-ktorého problému.

Za prakticky riešiteľný sa dlho považoval problém, ktorý sa dá riešiť *sekvenčne* v deterministickom polynomiálnom čase. Na úrovni abstraktného modelu označujeme túto triedu P . Zatiaľ čo pojem riešiteľného problému je zrejme nemenný, definícia praktickej riešiteľnosti sa posúva. V dnešných dňoch sú už za prakticky riešiteľné považované aj problémy, ktoré sú riešiteľné v polynomiálnom čase pravdepodobnostnými algoritmami, aproximačnými algoritmami, rýchlymi heuristickými algoritmami, paralelnými algoritmami, ...
- Pozornosť sa sústredila na porovnávanie problémov z hľadiska náročnosti ich realizácie, skúmajú sa pritom rôzne miery zložitosti (popisná, čas, pamäť, počet porovnaní,...)

Pozrime sa na problematiku z pohľadu teórie a praxe.

Zložitost' konkrétnych výpočtových úloh sa zaoberá riešením konkrétnych výpočtových problémov. Všimame si zložitost' riešenia, ktoré je vyjadrené/vnímané ako algoritmus v nejakom pseudo-programovacom jazyku. Popri hľadaní čo najlepších algoritmov na riešenie problémov, ktoré nás zaujímajú, sa snažíme o získanie všeobecnejších poznatkov. Medzi takéto rozhodne patria

- metódy tvorby efektívnych algoritmov; stretne sa s metódou rozdeľuj a panuj, dynamickým programovaním, pažravými algoritmami, prehľadávacími algoritmami, ...
- dokazovanie zložitosti konkrétnych algoritmov.

Abstraktná teória zložitosti - sa zaoberá riešením problémov trochu inak. V pozadí stoja abstraktné výpočtové modely. Snažíme sa o získanie takých poznatkov o výpočtoch, ktoré sú invariantné vzhľadom na výber počítača (z danej rozumnej triedy). Definovanie toho, čo je to "rozumný počítač" je samo o sebe nie celkom triviálny problém. *Prvá počítačová trieda* obsahuje všetky výpočtové modely, ktoré sú polynomiálne ekvivalentné¹ deterministickému TS. Uvedomme si, že všetky počítače z prvej počítačovej triedy definujú triedu prakticky riešiteľných problémov—teda problémov, riešiteľných v polynomiálnom čase, rovnako. Problém je riešiteľný v polynomiálnom čase bez ohľadu na výber počítača z tejto triedy. *Druhá počítačová trieda* obsahuje všetky také modely, ktoré vedia efektívne využívať priestor. To znamená, že trieda problémov, ktoré sa dajú riešiť v priestore $f(n)$ sa rovná triede problémov, ktoré sa dajú riešiť v čase exponenciálnom od $f(n)$.

Na problematiku sa môžeme pozrieť aj na základe spracovania informácie. To vedie k vymedzeniu teórie *sekvenčných* a *paralelných* výpočtov.

¹Modely sú polynomiálne ekvivalentné, ak zložitost' riešenia každého problému je na oboch rovnaká až na polynóm.

1.4 Základné pojmy a definície

Začnime opakovaním pojmov, s ktorými ste sa už iste stretli na predchádzajúcich prednáškach, napr. na prednáške "Dátové štruktúry a algoritmy".

Problém

- zobrazenie z množiny vstupných reťazcov do množiny výstupných reťazcov
- konečné problémy (z konečnej množiny do konečnej množiny) - napr. booleovské funkcie
- rozhodovacie problémy (odpoveď je áno-nie - je daný reťazec znakov syntakticky správnym programom v danom programovacom jazyku, existuje v danej množine objekt s danou vlastnosťou, sú dve (potenciálne nekonečné) množiny rovnaké,...)
- ...

Algoritmus

- pôvod slova algoritmus je "al-kawarizmi" v mene Perzského matematika (9. storočie)
- konečný návod na riešenie problému s použitím daných elementárnych operácií. Vyžadujeme, aby každá z elementárnych operácií mala presne špecifikovaný vzťah medzi vstupom a výstupom, aby bola vykonateľná v konečnom čase a konečnej pamäti (*strojovo a jazykovo nezávislé inštrukcie*)

Program

- implementácia algoritmu
- postupnosť korektne zapísaných inštrukcií, ktoré majú byť vykonané na počítači
- tieto inštrukcie sú zapísané v programovacom jazyku

1.4.1 Analýza zložitosti

Väčšinou je našou snahou nájsť *čo najlepší algoritmus* na riešenie skúmaného problému. Čo to ale znamená najlepší? Pozor, nejde o správnosť, kvalitu riešenia². Kvalitou algoritmu budeme rozumieť jeho zložitosť. Lenže na riešenie jedného problému existuje viacero prístupov, ktoré vedú k algoritmom rôznej zložitosti. Aká je teda zložitosť nášho problému?

Analýza algoritmu: využitie matematických metód k predikcii času a pamäte potrebnej na realizáciu algoritmu (pri zbehnutí programu)

Uvažujme *miery zložitosti* X (väčšinou čas, resp. pamäť, počet aritmetických operácií,..) a *model* M (modelom rozumieme vyšpecifikovanie množiny elementárnych inštrukcií, ktoré model poskytuje).

Zložitosť algoritmu A je funkcia veľkosti vstupných dát udávajúca množstvo miery zložitosti X spotrebovanej algoritmom A pri riešení problémov daného rozsahu.

$X_A^M(w)$ — množstvo miery zložitosti X spotrebovanej algoritmom A pri spracovaní vstupu w

²Iná situácia je pri optimalizačných problémoch, keď tzv. aproximačné algoritmy nemusia dávať presné - optimálne - riešenie. Vtedy hovoríme o kvalite riešenia a zložitosti algoritmu

$$X_A^M(n) \begin{cases} \max_{w, |w|=n} \{X_A^M(w)\}, & \text{zložitosť najhoršieho prípadu} \\ \min_{w, |w|=n} \{X_A^M(w)\}, & \text{zložitosť najlepšieho prípadu} \end{cases}$$

Keďže najlepší aj najhorší prípad sú vlastne extrémálnymi hodnotami, nedávajú celkom presnú predstavu o zložitosti pre konkrétny prípad. Preto je lepšou charakteristikou **zložitosť priemerného prípadu**

$$\overline{X_A^M(n)} = \sum_{w_i} p(w_i) X_A^M(w_i), \text{ kde}$$

sumu počítame cez všetky slová w_i dĺžky n a $p(w_i)$ je pravdepodobnosť výskytu konkrétneho vstupu w_i . Problémom však je, že

- často nepoznáme pravdepodobnostné rozdelenie vstupných údajov
- samotná suma sa ťažko počíta.

Preto niekedy predpokladáme rovnomerné rozdelenie a počítame priemernú zložitosť ako normálny aritmetický priemer

$$\overline{X_A^M(n)} = \frac{\sum p(w_i) X_A^M(w_i)}{|W_n|}, \text{ kde } W_n \text{ je množina všetkých vstupov veľkosti } n.$$

Uvedomme si význam parametra M ! Predpokladajme, že X označuje počet aritmetických operácií. Ako sa zmení zložitosť klasického algoritmu násobenia, ak za základnú aritmetickú operáciu budeme považovať násobenie jednociferným číslom v porovnaní s prípadom, keď základnou aritmetickou operáciou je násobenie ľubovoľne veľkých čísel?

Časovú zložitosť budeme väčšinou označovať T .

Čo je dobrá miera zložitosti?

- **CPU čas**
strojovo závislá (pre rôzne architektúry môže byť rôzna)
- **počet (vykonateľných) príkazov**
závisí od programovacieho jazyka, od programátorovho štýlu,..
- **počet opakovaní cyklu**
závisí od jazyka; dĺžka realizácie jedného behu cyklu sa mení
- + **počet základných operácií**
fixujeme množinu elementárnych inštrukcií(model); toto je strojovo nezávislé

problém	počítané základné operácie
Vyhľadávanie v zozname	Porovnanie
Násobenie matíc	Násobenie (jednotlivých prvkov)
Triedenie	Porovnanie
Prehľadávanie (BS)	Prehľadanie vrchola

Prečo potrebujeme analyzovať algoritmy?

analýza

- lepšie odráža skutočnosť ako experimentovanie
- umožňuje nám porovnávať kvalitu algoritmov bez ich implementácie

- umožňuje odhad času a pamäťových nárokov implementácie
- umožňuje lepšie pochopenie algoritmu, identifikáciu rýchlych a pomalých častí

Algoritmus	T(n)	max. rozsah	problému
		sek	min
A1	n	1000	60 000
A2	nlogn	141	4 893
A3	n^2	31	244
A4	n^3	10	39
A5	2^n	9	15

Predstavme si, že počítač 10-násobne urýchlíme (rýchlosť M je menšia ako rýchlosť N). Ako sa zmení maximálny rozsah problému, ktorý môžeme v danom čase riešiť?

Algoritmus	T(n)	max. na M	max. na N
A1	n	S1	10*S1
A2	nlogn	S2	10*S2
A3	n^2	S3	3.16*S3
A4	n^3	S4	2.15*S4
A5	2^n	S5	S5+3.3

Asymptotická zložitosť

Keďže konštanty (rozumnej veľkosti) nemajú podstatný vplyv na celkovú efektívnosť algoritmu, uspokojíme sa často len s asymptotickým odhadom zložitosti (zanedbáme konštanty). Preto sa v súvislosti so zložitou algoritmu väčšinou stretáme s pojmom asymptotickej zložitosti. Zopakujme si definície asymptotík:

$g(n) = O(f(n))$	$\exists c, n_0 \in \mathbb{N} \forall n \geq n_0 \quad g(n) \leq cf(n)$
$g(n) = \Omega(f(n))$	$\exists c, n_0 \in \mathbb{N} \forall n \geq n_0 \quad g(n) \geq cf(n)$
$g(n) = \Theta(f(n))$	$g(n) = O(f(n)) \wedge g(n) = \Omega(f(n))$
$g(n) = o(f(n))$	$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = 0$

A teraz sa môžeme vrátiť k tomu, aby sme hovorili o **zložitosti problému**. Ako súvisí zložitosť $X_A^M(n)$ algoritmu A riešiaciho daný problém P so zložitou $X_P^M(n)$ daného problému?

- $X_P^M(n) \leq X_A^M(n)$ - každý algoritmus riešiaci daný problém poskytuje **horný odhad zložitosti problému**; pokiaľ máme len tento odhad, nevieme posúdiť kvalitu nami získaného algoritmu.
- $X_P^M(n) \geq f(n) \Leftrightarrow$ neexistuje algoritmus (z danej triedy M) riešiaci problém P so zložitou menšou ako $f(n)$. Ak sa nám také niečo podarí ukázať, potom $f(n)$ je **dolný odhad zložitosti problému**.

Keď v nerovnici $f(n) \leq X_p^M(n) \leq X_A^M(n)$ sú hranice $f(n)$ a $X_A^M(n)$ blízko pri sebe, podarilo sa nám nájsť dobrý algoritmus. Ťažko možno očakávať, že by sme dosiahli úplnú rovnosť.

$$\text{algoritmus } A \text{ je } \begin{cases} \text{optimálny, ak} & f(n) = X_A^M(n); \\ \text{asymptoticky optimálny, ak} & X_p^M(n) = \Theta(X_A^M(n)) \end{cases}$$

1.4.2 Analýza problému triedenia

Uvažujme problém triedenia. Pýtame sa, aká je zložitosť tohto problému v triede tzv. porovnávacích algoritmov (algoritmov, ktoré nevedia "rozoberať" kľúče, ale jedinou informáciu môžu získať pomocou porovnania dvoch kľúčov).

Všimnime si najprv horný odhad - budeme uvažovať jeden z najznámejších algoritmov za predpokladu, že vstupné hodnoty sú rôzne (ako sa algoritmus a nasledujúca analýza zmenia, keď tento predpoklad odstránime?)

Algorithm QSort(S)

vyber $a \in S$;

$S1 = \{b \in S; b < a\}$;

$S2 = \{b \in S; b > a\}$;

return(QSort(S1) . a . QSort(S2))

Čo vieme povedať o zložitosti tohto algoritmu? Meranú počtom porovnaní ju vo všeobecnosti vyjadríme nasledovne:

$$T(|S|) = T(|S1|) + T(|S2|) + n$$

Ľahko vidíme, že zložitosť podstatne závisí od konkrétnych veľkostí množín S1, S2. Takže zložitosť jednotlivých zaujímavých prípadov dostaneme jednoduchou analýzou vzťahu veľkostí množín S1, S2.

Najhorší prípad nastáva, ak pri každom výbere prvku a zvolíme prvok, ktorý je najmenší alebo najväčší v množine; má to za následok, že zložitosť najhoršieho prípadu je $O(n^2)$.

Skúsme sa zamyslieť nad *priemerným prípadom*. Bez ohľadu na to, aké sú skutočné hodnoty triedených prvkov, na zložitosť algoritmu má najväčší vplyv poradie (po usporiadaní množiny) prvku a - tento má totiž vplyv na mohutnosti množín S1, S2. Preto zložitosť priemerného prípadu vyjadríme nasledovne:

$$\overline{T(n)} \leq \frac{1}{n} \sum_{i=1}^n (T(i-1) + T(n-i) + n)$$

$$T(0) = T(1) = b \quad T(i) = 0 \text{ pre } i < 0$$

Ukážeme, že riešením je $T(n) \approx n \log n$:

$$\begin{aligned} \overline{T(n)} &\leq \frac{1}{n} \sum_{i=0}^n (T(i) + T(n-i-1) + n) \\ &= cn + \frac{1}{n} \{T(0) + T(n-1) + T(1) + T(n-2) + \dots + T(n-1) + T(0)\} \\ &= cn + \frac{2}{n} \sum_{i=0}^{n-1} T(i) \end{aligned}$$

Dôkaz spravíme matematickou indukciou.

IP: $T(n) \leq kn \log n, k = 2c + 2b$

$$T(2) \leq cn + \frac{2}{n} \sum_{i=0}^n (T(i)) = 2c + \frac{2}{2}(T(0) + T(1)) = 2c + 2b$$

$$\overline{T(n)} \leq cn + \frac{2}{n} \{T(0) + T(1)\} + \frac{2}{n} \sum_{i=2}^{n-1} k \log i = cn + \frac{4b}{n} + \frac{2k}{n} \sum_{i=2}^{n-1} \log i$$

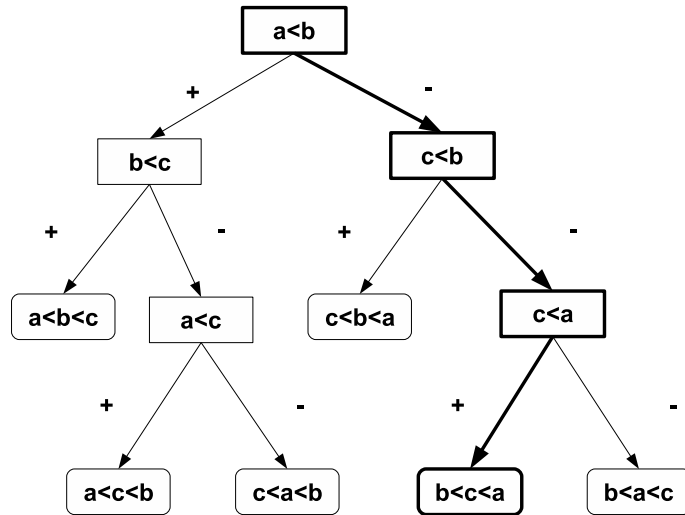
$$\begin{aligned} \text{Keďže } \sum_{i=2}^{n-1} \log i &\leq \int_2^n x \log x dx = \left[\frac{1}{2} x^2 \log x - \frac{x^2}{4} \right]_2^n \\ &= \left(\frac{1}{2} n^2 \log n - 2 - \frac{n^2}{4} + 1 \right) \leq \frac{1}{2} n^2 \log n - \frac{n^2}{4} \end{aligned}$$

$$\text{tak } \overline{T(n)} \leq \frac{4b}{n} + \frac{2k}{n} \sum_{i=2}^{n-1} \log i \leq cn + \underbrace{\frac{4b}{n} - \frac{kn}{2}}_{\leq 0} + kn \log n \leq kn \log n$$

Uvedomte si, že pre $n \geq 2, k = 2c + 2b$ je $cn + \frac{4b}{n} - \frac{kn}{2} \leq 0$

Ostalo nám ešte nájsť **dolný odhad** na zložitosť problému triedenia. Musíme vyargumentovať, koľko porovnaní minimálne musí použiť každý triediaci algoritmus³. Pomôžeme si abstraktným modelom; využijeme rozhodovací strom. Čo to je?

Rozhodovací strom je neuniformný výpočtový model - pre každý rozsah vstupu sa konštruje iný rozhodovací strom. Rozhodovací strom pre triedenie n-prvkovej množiny prvkov označíme RS(n). Opäť predpokladáme, že vstupné hodnoty sú rôzne. Ako sa RS(n) zmení, ak tento predpoklad vynecháme?



Obrázok 1.1: Rozhodovací strom pre triedenie 3 prvkov. Zvýraznená cesta odpovedá vstupným hodnotám $a = 7, b = 2, c = 3$

- vnútorné vrcholy RS(n) obsahujú porovnanie $?a \leq b?$
- listy RS(n) obsahujú permutáciu $a(i_1), \dots, a(i_n)$ vstupných hodnôt (resp. správny výsledok pre daný vstup)

³z triedy porovnávacích algoritmov

Ako prebieha výpočet? "Postavíme sa" do koreňa $RS(n)$. V prípade kladnej odpovede na porovnanie "postupujeme" doľava, v prípade zápornej odpovede doprava. Výsledná permutácia (odpoveď) je v liste, do ktorého sme sa dostali.

Zložitosť konkrétneho prípadu je rovná dĺžke realizovanej cesty v rozhodovacom strome. Najlepší prípad je najkratšia cesta od koreňa do listu, najhorší prípad je hĺbka stromu. Takže sa presunieme k analýze binárnych stromov.

Dolný odhad pre zložitosť problému triedenia:

- počet listov každého rozhodovacieho stromu triediaceho n -prvkovú množinu je aspoň $n!$
- počet listov binárneho stromu hĺbky h je najviac 2^h
- počet porovnaní je pre konkrétny rozhodovací strom rovný jeho hĺbke
- pre ľubovoľný algoritmus A možno pre každé n zostrojiť $RSA(n)$

$$n! \leq \text{počet listov } RSA(n) \leq 2^h, \text{ kde } h \text{ je hĺbka } RSA(n)$$

↓

$$h \geq \log n! \approx c * n \log n$$

Keďže tento vzťah platí pre ľubovoľný algoritmus, je $c n \log n$ dolným odhadom pre zložitosť problému triedenia meranú počtom porovnaní.

□

1.4.3 Dolný odhad pre problém minMax

problém minMax **vstup:** n -prvková množina S s definovaným usporiadaním
výstup: maximálny a minimálny prvok množiny S

Venujme sa dolnému odhadu (zamyslite sa nad efektívnymi algoritmi).

- V prvom rade si treba ujasniť, čo budeme považovať za mieru zložitosti. Bude to opäť počet porovnaní - takéto algoritmy sú nezávislé od skutočnej množiny prvkov; treba len správne realizovať porovnanie.
- Pri dolnom odhade si opäť pomôžeme abstrakciou; tentokrát to bude stavový priestor. V každom okamihu riešenia priradíme rozpracovanej úlohe nejaký stav/charakterizáciu. Množina všetkých potenciálnych stavov potom tvorí stavový priestor.
- Počiatočnú situáciu charakterizuje nejaký stav (resp. množina stavov) a výstupná situácia tiež zodpovedá nejakému stavu, resp. množine stavov.
- Stav úlohy sa mení len pri vykonávaní nejakých operácií - v našom prípade sa stav mení následkom porovnania. Takto konkrétnemu priebehu vykonávania algoritmu odpovedá cesta v stavovom priestore.
- Dolný odhad na počet porovnaní získame argumentáciou o dĺžke cesty z počiatočného stavu do koncového stavu. Každé porovnanie môže zmeniť rozloženie prvkov v množinách, a teda stav. Keďže dĺžka cesty tvorí dolný odhad na počet porovnaní, budeme sa zaujímať o dĺžku cesty z počiatočného stavu do koncového. Ide o "hru" medzi nami a algoritmom. My sa snažíme cestu predlžovať a máme k tomu jediný prostriedok - voľbu vstupu. Naproti tomu algoritmus sa snaží cestu skracovať - robí to voľbou krokov (výberom prvkov, ktoré sa v tom ktorom okamihu porovnajú).

Stav (pre problém minMax) je štvorica $(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d})$, kde

- a** je počet prvkov, ktoré sa ešte neporovnávali, A meno tejto množiny
- b** je počet prvkov, ktoré sa už porovnávali a v každom porovnaní boli väčšie, B meno tejto množiny
- c** je počet prvkov, ktoré sa už porovnávali a v každom porovnaní boli menšie, C meno tejto množiny
- d** je počet prvkov, ktoré sa už porovnávali a boli aj menšie aj väčšie, D meno tejto množiny

Stavový priestor množina $\{(a, b, c, d); a, b, c, d \in \{0, 1, \dots, n\}, a + b + c + d = n\}$

Počiatočný stav je charakterizovaný štvoricou $(n, 0, 0, 0)$

Koncový stav je charakterizovaný štvoricou $(0, 1, 1, n - 2)$

Keďže sa potrebujeme dostať z bodu $(n, 0, 0, 0)$ do bodu $(0, 1, 1, n-2)$, musí

- prvá zložka (mohutnosť množiny A) klesnúť o $n-2$,
- tretia a štvrtá (mohutnosť množín B, C) stúpnuť o 1,
- posledná zložka (mohutnosť množiny D) stúpnuť o $n-2$.

Všimnime si, ako jedno porovnanie môže zmeniť stav úlohy:

	porovnávané prvky	výsledok porovnania	starý stav	(nový stav
1	$x \in A, y \in A$	$x < y$ $x > y$	(a, b, c, d)	$(a-2, b+1, c+1, d)$ $(a-2, b+1, c+1, d)$
2	$x \in A, y \in B$	$x < y$ $x > y$	(a, b, c, d)	$(a-1, b, c+1, d)$ $a-1, b, c, d+1)$
3	$x \in A, y \in C$	$x < y$ $x > y$	(a, b, c, d)	$(a-1, b+1, c, d)$ $a-1, b, c, d+1)$
4	$x \in A, y \in D$	$x < y$ $x > y$	(a, b, c, d)	$(a-1, b, c+1, d)$ $(a-1, b, c, d+1)$
5	$x \in B, y \in B$	$x < y$ $x > y$	(a, b, c, d)	$(a, b-1, c, d+1)$ $(a, b-1, c, d+1)$
6	$x \in C, y \in C$	$x < y$ $x > y$	(a, b, c, d)	$(a, b, c-1, d+1)$ $(a, b, c-1, d+1)$
7	$x \in B, y \in C$	$x < y$ $x > y$	(a, b, c, d)	$(a, b-1, c-1, d+2)$ (a, b, c, d)
8	$x \in C, y \in D$	$x < y$ $x > y$	(a, b, c, d)	(a, b, c, d) $(a, b, c-1, d+1)$
9	$x \in B, y \in D$	$x < y$ $x > y$	(a, b, c, d)	$(a, b-1, c, d+1)$ (a, b, c, d)

Ak si predstavíme, že porovnávané prvky sa po porovnaní presúvajú do príslušnej množiny, musí sa každý prvok, ktorý skončí v množine D, najprv presunúť do jednej z množín B, C a až potom do množiny D. Prvky, ktoré skončia v B, resp. C, sa zúčastnili aspoň jedného porovnania. Dolný odhad teda získame ako súčet počtu porovnaní, ktorými sa prvky presunú z množiny A plus počet porovnaní, ktorými sa prvky presunú z $B \cup C$ do D.

Uvedomme si, ako súvisí naša "hra"s tabuľkou, ktorá zobrazuje možné zmeny stavu. Algoritmus "vyberá"hrubý riadok, my vhodnou voľbou počiatočných hodnôt môžeme ovplyvniť, ktorá z možností v zvolenom riadku nastala.

- Pretože algoritmu nemôžeme zabrániť, aby prvky z množiny A nepresúval podľa bodu 1, na presun prvkov z A do B alebo C môžeme počítať iba $n/2$ porovnaní - algoritmus vybral pre nás najhoršiu možnosť.
- Koľko porovnaní možno rátať na presuny do množiny D? Vidíme, že posledná zložka sa v jedinom prípade môže zvýšiť o 2 (prípád 7). Nie je ťažké si uvedomiť, že vieme skonštruovať taký vstup, aby každý prvok z množiny B bol väčší ako každý prvok z množiny C (ako?). V takejto situácii prípad 7 nikdy nenastane, a preto na zmenu poslednej súradnice treba minimálne $n-2$ porovnaní.

Dostávame teda

$$\text{počet porovnaní} \geq \frac{n}{2} + n - 2 = \frac{3n}{2} - 2.$$

□

1.5 Amortizovaná zložitosť

Vieme, že zložitosť je funkcia veľkosti/rozsahu vstupu. Pri analyzovaní zložitosti konkrétneho algoritmu väčšinou postupujeme tak, že najprv zanalyzujeme zložitosť jednotlivých operácií (často v najhoršom prípade) a potom sčítame cez všetky realizované operácie. Keď teda máme cyklus, spočítame (väčšinou) zložitosť trvania jednej iterácie a potom pre násobíme počtom opakovaní. Iným prístupom je tzv. *amortizovaná zložitosť*, keď analyzujeme postupnosť realizovaných operácií ako celok.

1.5.1 Metóda zoskupení

Princíp tejto metódy spočíva v tom, že jednotlivé operácie rozdelíme do skupín a potom analyzujeme skupinu ako celok. Aplikujme na príklad manipulácie so zásobníkom a s binárnym počítadlom.

Príklad 1.1 *Uvažujme údajovú štruktúru zásobník s tromi operáciami*

Push(S,x)

vloží do zásobníka S prvok x. Cena/zložitosť realizácie tejto operácie je 1

Pop(S)

(deštruktívne) vráti vrchný prvok zásobníka S. Cena realizácie tejto operácie je tiež 1.

MultiPop(S,k)

vyberie zo zásobníka k prvkov, resp. zásobník vyprázdni, ak v ňom bolo menej ako k prvkov. Cena realizácie operácie je prirodzene k (resp. počet naozaj odstránených prvkov)

Predpokladajme, že máme postupnosť n operácií, ktoré manipulujú so zásobníkom S. Zaujímá nás zložitosť realizácie tejto postupnosti.

Zrejme najhoršia/najťažšia operácia je *MultiPop*, ktorého cena je nanajvyš n . Preto klasicky počítaná zložitosť

$$\text{počet realizovaných operácií} \times \text{zložitosť najťažšej} = n \cdot n = O(n^2)$$

Pri použití metódy zoskupení na problém manipulácie so zásobníkom rozdelíme jednotlivé operácie do dvoch skupín.

- | | | |
|----|---------------|--|
| I | Push | dá sa vždy realizovať, vždy stojí 1, preto je celkový príspevok skupiny n |
| II | Pop, Multipop | nemôžeme vybrať viac prvkov ako sme vložili, preto je sumárny príspevok tejto skupiny tiež n |

Získali sme presnejší odhad zložitosti – $2n$

Príklad 1.2 Uvažujme k -bitové binárne počítadlo realizované ako pole $A[0 \dots k-1]$. Hodnota počítadla reprezentovaného poľom $A[a_0 \dots a_{k-1}]$ je

$$\sum_{i=0}^{k-1} A(i) \cdot 2^i$$

Pripočítanie jednotky možno vyjadriť takto

$INC(A);$

$i \leftarrow 0$

while ($i \leq k - 1$ AND $A(i) = 1$) **do** $A(i) \leftarrow 0; i \leftarrow i + 1$

if $i \leq k - 1$ **then** $A(i) \leftarrow 1$

Ak ako mieru zložitosti uvažujeme počet preklopených bitov, stojí nás jedno pričítanie jednotky zrejme nanejvýš k . Pri realizovaní postupnosti n takýchto inštrukcií sa tak klasicky dostaneme k celkovému počtu

$$n \cdot k$$

Použijeme metódu zoskupení. Vytvoríme k skupín, pričom i -tu skupinu tvorí preklopenie $A(i)$. Uvedomme si, že $A(i)$ sa preklopí v každom 2^i -tom volaní INC . Preto celkovú zložitosť vyjadríme

$$\sum_{i=0}^{\lfloor \log n \rfloor} \left\lfloor \frac{n}{2^i} \right\rfloor \leq n \cdot \sum_{i=0}^{\lfloor \log n \rfloor} \frac{1}{2^i} \leq 2n$$

1.5.2 Metóda súčtov

Okrem prirodzenej ceny dostane každá (sledovaná) operácia kredit⁴. Pri realizácii operácie platíme jej kreditom. Ak je kredit väčší ako prirodzená cena operácie, akoby sme si predplatili nejakú ďalšiu manipuláciu s objektami.

cena \leq **kredit** zvyšné kredity (rozdiel medzi kreditmi a cenou) si uložíme na účet k použitiu v budúcnosti

kredit $<$ **cena** rozdiel musí operácia doplatiť z účtu

Kvôli bezproblémovému realizovaniu postupnosti operácií musíme mať účet v priebehu celého výpočtu nezáporný \Rightarrow **nesmieme ísť do debetu**. Ak tomu tak je, potom súčet kreditov dáva horný odhad na zložitosť.

ZÁSOBNÍK

	cena	kredit
Push	1	2
Pop	1	0
Multipop	$\min\{k, S \}$	0

⁴prepočítaniu na kredity sa niekedy hovorí amortizovaná cena operácie

Všimnime si, že hoci je cena realizácie Pop jednotková, priradili sme tejto operácii 2 kredity – akoby sme si predplatili aj odstránenie práve vloženého prvku. Je zrejmé, že zložitosť je nanajvýš $2n$.

POČÍTADLO

	cena	kredit
nastavenie bitu na 1	1	2
nastavenie bitu na 0	1	0

Hodnota kreditu 2 pri nastavovaní príslušného bitu na 1 opäť počíta "s budúcnosťou" – keď sa nabadúce dostaneme na toto miesto, budeme ho *musieť* preklopiť. Opäť ľahko vidno, že zložitosť realizovania n binárnych pripočítaní je $2n$. Stačí si uvedomiť, že každé pripočítanie jednotky preklopí len jeden bit z 0 na 1 a na 0 preklápan len taký bit, ktorý som predtým nastavila na 1 (a teda som si na túto operáciu ušetrila).

1.6 Základné výpočtové modely

Rozvoj počítačov spôsobil, že za formálny popis riešenia problémnu môžeme považovať riešenie napísané v programovacom jazyku. Počítač potom realizuje program na jednotlivých vstupoch. Hovoríme teda o algoritmickom riešení problémov.

Ak chceme dokázať, že problém je algoritmicky riešiteľný, napíšeme jeho riešenie v programovacom (algoritmickom) jazyku. Na tejto úrovni nie je nutné jazyk fixovať. Potreba formálnej definície pojmu *algoritmus* vyvstáva, keď chceme hovoriť o *riešiteľnosti/neriešiteľnosti* problémov, resp. o takých aspektoch zložitosti problémov, ktoré sú nezávislé od výberu konkrétneho z rozumnej triedy modelov.

Stretávame sa s viacerými modelmi výpočtov. Potrebujeme, aby bol model jednoduchý, s malým počtom jednoduchých elementárnych inštrukcií, ale pritom aby odrážal naše intuitívne vnímanie pojmu algoritmu/algoritmickej riešiteľnosti/vypočítateľnosti. Za takýto model sa berie *Turingov stroj* (TS). V súvislosti so skúmaním zložitosti konkrétnych problémov sa tiež študuje tzv. RAM.

1.6.1 (M)RAM

V tejto časti sa budeme venovať abstraktnému výpočtovému modelu, ktorý pripomína assembler. (M)RAM je skratkou z anglického Random Access Machine, teda počítač s priamym prístupom do pamäti.

$$(M)RAM = \text{Program} + \text{Dátová štruktúra (pamäť)}$$

(M)RAM je *model* počítača von Neumanovského typu - pamäť, program, čítač inštrukcií. Pri jeho definovaní teda musíme popísať spôsob komunikáciu s okolím – vstupy/výstupy; organizáciu pamäte a spôsob jej adresácie; programovací jazyk, čiže základné inštrukcie. Abstrakcia tohto modelu je najmä v jeho potenciálne nekonečnej pamäti.

Pamäť je pole registrov, v ktorých sa nachádzajú čísla. Pritom predpokladáme

- veľkosť registrov je neobmedzená/operácie nad nekonečne veľkými číslami
- počet registrov je neobmedzený
- priamy prístup do každého registra (indexovanie)

Vstup je konečná postupnosť celých čísel, ktorú čítame zľava doprava. Predpokladáme jednosmerné čítanie, čo znamená, že po vykonaní jednej operácie načítania sa "čítacia hlava" nastaví na ďalšie číslo v poradí.

Program je konečná postupnosť inštrukcií. Nech

- i** označuje číslo zo vstupu
- c(j)** označuje obsah registra $R(j)$
- P** je čítač inštrukcií (číslo práve vykonávaného riadku programu); každá inštrukcia zvýši jeho hodnotu o jedna, ak to skokovou inštrukciou nie je určené inak
- x** je ľubovoľný operand typu
 - =j (konštanta),
 - j (priama adresácia)
 - *j (nepriame adresovanie)

R(0) označuje akumulátor

$$\text{adresné módy} \begin{cases} x \text{ je číslo } j, & \text{obsah registra } R(j) \\ x \text{ je } *j, & \text{obsah registra, ktorého číslo je uložené v registri } R(j) \\ x \text{ je } =j, & \text{priamo číslo } j \end{cases}$$

Inštrukcie	inštrukcia	operand	sémantika
	READ	i	$c(i) \leftarrow$ číslo zo vstupu
	READ	$*i$	$c(c(i)) \leftarrow$ číslo zo vstupu
	STORE	j	$c(j) \leftarrow c(0)$
	STORE	$*j$	$c(c(j)) \leftarrow c(0)$
	LOAD	x	$c(0) \leftarrow x$
	ADD	x	$c(0) \leftarrow c(0) + x$
	SUB	x	$c(0) \leftarrow c(0) - x$
	JUMP	j	$P \leftarrow j$
	JPOS	j	<i>if</i> $c(0) > 0$ <i>then</i> $P \leftarrow j$
	JZERO	j	<i>if</i> $c(0) = 0$ <i>then</i> $P \leftarrow j$
	HALT		ukončenie výpočtu
	WRITE	$= j$	j
	WRITE	j	$c(j)$
	WRITE	$*j$	$c(c(j))$

Na RAM sa môžeme pozeráť dvomi spôsobmi

1. RAM **počíta funkciu** f . Niekedy sa stretneme s tým, že v takomto prípade sa nepoužívajú inštrukcie WRITE, ale za výstup po zastavení výpočtu sa považuje obsah akumulátora.
2. RAM dáva ako výsledok viacero čísel. Vtedy sa používa aj inštrukcia zápisu. Výstup je postupnosť celých čísel. Po zapísaní čísla na výstupe sa hlava posúva ďalej, čo znamená, že výsledkom je postupnosť čísel zapísaná na výstupnej páske.

Ako je to s definovaním zložitosti? Prirodzené by bolo definovať časovú zložitosť ako počet vykonaných inštrukcií. Vzhľadom k tomu, že máme potenciálne nekonečný register a teda potenciálne nekonečne veľké čísla, asi by to niekedy bolo zavádzajúce. Preto sa rozlišujú dva rôzne prístupy

jednotkové kritérium Pri **jednotkovej** cene je manipulácia s číslom/registrom jednotkovej zložitosti

- **čas** je počet vykonaných inštrukcií
- **priestor, resp. pamäť** je počet použitých registrov

Je zrejmé, že jednotková cena odráža zložitosť vtedy, ak veľkosť čísel (dĺžku zápisu) môžeme ohraničiť konštantou. Vtedy manipuláciu s číslom môžeme považovať za konštantu; keďže väčšinou nás zaujíma asymptotické správanie zložitosti, je väčšinou jedno, či uvažujeme $O(f(n))$ alebo $O(cf(n))$, kde c je konštanta.

logaritmicke kritérium **Logaritmicke** cena berie do úvahy veľkosť čísel, s ktorými pracujeme. Cena inštrukcie nie je 1, ale $\lfloor \log i \rfloor + 1$, kde i je najväčšia absolútna hodnota z čísel, s ktorými pri vykonávaní danej inštrukcie manipulujeme (niekedy aj počet bitov, ktoré sa pri vykonaní inštrukcie spracujú; rozdiel v týchto dvoch definíciách je v multiplikatívnej konštante). Analogická úvaha je pre pamäťovú zložitosť, kde veľkosť (cena) použitého registra nie je jednotková, ale je to maximálna dĺžka čísla, uloženého počas výpočtu v tom-ktorom registri.

- **čas** je súčet cien vykonaných inštrukcií
- **pamäť** je súčet cien všetkých registrov, použitých v priebehu výpočtu

Vidíme, že logaritmicke cena odstraňuje – z hľadiska zložitosti – abstrakciu RAMu, ktorou je dvojrozmerná nekonečnosť pamäte - počet registrov aj ich veľkosť môžu

byť nekonečne veľké. Ak pracujeme s nekonečne veľkými číslami, sú všetky povolené operácie RAMu lineárnej zložitosti vzhľadom na dĺžku binárneho zápisu. Teraz už asi vidno, prečo sme pri definovaní aritmetických operácií nedefinovali násobenie a delenie. Súvisí to s tým, že algoritmy násobenia a delenia nie sú lineárne. Je zrejmé, že pridaním násobenia a delenia

- neovplyvníme výpočtovú silu RAMu
- zložitost' nebude úplne zodpovedať tomu, čo očakávame

Model s násobením a delením je však užitočný, preto ho budeme používať a budeme ho označovať MRAM.

MRAM je RAM, ktorý má navyše inštrukcie

MRAM

DIV celočíselné delenie
MULT násobenie

Keďže nie je jednoduché počítať logaritmickú zložitost' presne, postupujeme väčšinou tak, že vypočítame zložitost' pri jednotkovej cene a prenásobíme ju maximálnou dĺžkou čísla, ktoré sa v priebehu výpočtu použilo. Získame tak horný odhad na logaritmickú zložitost'.

Úloha: Napíšte RAM pre problém triedenia. Začnite s jednoduchým (z hľadiska programovania, nie efektívnosti) algoritmom Bubblesort. Určte zložitost' Vášho programu pri jednotkovej aj logaritmickej cene.

Úloha: Napíšte RAM pre problém Insertsort. Určte zložitost' Vášho programu pri jednotkovej aj logaritmickej cene.

Úloha: Napíšte RAM, ktorý načíta "reťazec"—postupnosť čísel: n, a_1, \dots, a_n , kde $a_i \in \{1, 2\}^*$. Na výstup vypíše:

1. $a_2, a_1, a_3, a_4, \dots, a_n, a_{n-1}$ alebo $a_2, a_1, a_3, a_4, \dots, a_{n-2}, a_{n-1}, a_n$ podľa toho, či je n párne alebo nepárne
2. 1 ak tvorí vstupná postupnosť palindróm ($a_1 \dots a_n = a_n \dots a_1$)
3. 1 ak vstupná postupnosť obsahuje ako súvislý podreťazec palindróm dĺžky aspoň $n/2$
4. 1 ak číslo, ktorého desiatkovým zápisom je vstupná postupnosť, je prvočíslo

Určte zložitost' Vášho programu pri jednotkovej aj logaritmickej cene.

1.6.2 Základný model Turingovho stroja

V tejto časti sa budeme venovať z nejakého pohľadu najjednoduchšej verzii Turingovho stroja (TS) - bude to model, ktorého výpočtová sila je ekvivalentná tomu, čo považujeme za algoritmicky riešiteľné. Analogicky ako v prípade (M)RAMu ide o abstraktný výpočtový model, kde abstrakciou je jednak zúženie množiny elementárnych inštrukcií a na strane druhej uvažovanie neobmedzene veľkej pamäte.

Pamäť TS je (jednosmerne)potenciálne nekonečná páska, tvorená potenciálne nekonečnou postupnosťou políčok. Každé políčko môže obsahovať práve jeden z konečnej množiny symbolov. Vstupné slovo je na začiatku uložené v súvislom bloku po sebe

idúcich políček, čítacia hlava je nastavená na prvom symbole vstupného slova. Ľavý okraj pásky budeme označovať špeciálnym symbolom $\$$. Ostatné políčka obsahujú špeciálny znak, tzv. B (blank), ktorý vyjadruje to, že políčko je prázdne. Prechodová funkcia na základe symbolu pod hlavou (v tomto prípade čítacou i zapisujúcou) a stavu konečnosťovej riadiacej jednotky určuje nový obsah políčka pod hlavou, nový stav a posun hlavy (nanajvýš o jedno políčko doľava, resp. doprava).

Turingov stroj **Definícia 1.1** *Turingov stroj je šestica $T = (\Sigma, \Gamma, K, q_0, \delta, F)$, kde*

$\Sigma; B, \$ \notin \Sigma$, je konečná neprázdna množina symbolov, tzv. vstupná abeceda,
 $\Gamma; \Sigma \subseteq \Gamma$ je konečná neprázdna množina symbolov, tzv. pracovná abeceda
 K je konečná neprázdna množina stavov
 $q_0, q_0 \in K$ je počiatočný stav
 $F, F \subseteq K$, je množina koncových/akceptujúcich stavov
 $\delta :$ $K \times \{\Gamma - \$\} \rightarrow K \times \{\Gamma - B\} \times \{0, 1, -1\} \cup K \times \$ \rightarrow K \times \$ \times \{0, 1, \}$
je prechodová funkcia

konfigurácia TS *Konfigurácia C Turingovho stroja T je reťazec $\alpha q \beta$, kde $\alpha \beta$ je celá neprázdna časť pásky, q je stav, hlava T je umiestnená na prvom políčku-znaku z β . Uvedomme si, že konfigurácia je len dohodnutým popisom kompletnej informácie o stroji T .*

počiatočná konfigurácia *Počiatočná konfigurácia je $C_0 = q_0 a_1 a_2 \dots a_n$, kde $a_1 a_2 \dots a_n$ je vstupné slovo. Je zrejmé, že C_0 popisuje situáciu na začiatku výpočtu.*

krok výpočtu *Elementárnymi inštrukciami stroja T sú zrejmé prvky prechodovej funkcie δ . Aplikovaniu prechodovej funkcie hovoríme *krok výpočtu*. Hovoríme, že konfiguráciu C_{i+1} vieme dostať z konfigurácie C_i v jednom kroku, značíme $C_i \mapsto C_{i+1}$, ak*

$C_i = a_1 a_2 \dots a_j \mathbf{q} a_{j+1} a_{j+2} \dots a_m$ a

$$C_{i+1} = \begin{cases} a_1 a_2 \dots a_j \mathbf{p} b a_{j+2} \dots a_m, & \text{pričom } (p, b, 0) \in \delta(q, a_{j+1}); \text{ hlava stojí} \\ a_1 a_2 \dots a_j \mathbf{b} p a_{j+2} \dots a_m, & \text{pričom } (p, b, 1) \in \delta(q, a_{j+1}); \text{ hlava sa hýbe doprava} \\ a_1 a_2 \dots p a_j \mathbf{b} a_{j+2} \dots a_m, & \text{pričom } (p, b, -1) \in \delta(q, a_{j+1}); \text{ hlava sa hýbe doľava} \end{cases}$$

Symbolom \mapsto^* označujeme reflexívny tranzitívny uzáver \mapsto .

výpočet *Výpočet Turingovho stroja T je postupnosť konfigurácií $C_0, C_1, C_2, \dots, C_m$, pričom C_0 je počiatočná konfigurácia a $C_i \mapsto C_{i+1}$.*

TS ako akceptor *Keď TS používame ako akceptor, inými slovami, ak na TS riešime rozhodovacie problémy, zaujíma nás, či vstupné slovo patrí alebo nepatrí do príslušného jazyka. V takom prípade sa zvykne hovoriť o *akceptujúcom výpočte*. Akceptujúci výpočet je výpočet, ktorý končí konfiguráciou so stavom z množiny koncových stavov (ktorému sa vtedy hovorí aj akceptujúci stav). Potom *jazyk L* akceptovaný TS T je množina slov*

$$L(T) = \{w | q_0 w \mapsto^* \alpha q \beta, q \in F\}^5$$

TS počítajúci funkciu

⁵Uvedomme si, že sme nešpecifikovali, ako sa má skončiť výpočet na takom vstupe, ktorý nepatrí do daného jazyka. Ak k danému jazyku L vieme skonštruovať taký TS, ktorého výpočet na každom vstupe skončí, vtedy je L rekurzívny jazyk; hovoríme tiež, že TS jazyk L rozhoduje/rozpoznáva.

Ak považujeme TS za model počítača, chceli by sme, aby na vstup reagoval výstupom. Keďže máme (zatiaľ) len jednu pásku, musíme aj výstup napísať na ňu. V takomto prípade od *koncovej konfigurácie* C_m vyžadujeme, aby sme na páske videli/našli výstup. Nech teda $C_0, C_1, C_2, \dots, C_m$ je výpočet TS T .

Konfigurácia $C_m = \alpha_m q \beta_m$ je *koncová*, ak

$$q \in F$$

$$\beta_m = \gamma_m \# y, \text{ pričom } y \text{ je výstup/výsledok.}$$

Je zrejmé, že akceptor dostaneme aj tak, že $y \in \{0, 1\}$.

Úloha: Napíšte TS, ktorý rozhoduje jazyk

- $L = \{w c w \mid w \in \{a, b\}^*\}$
- $L = \{w w \in \{a, b\}^*\}$

Ak je príslušný TS taký, že vieme zaručiť konečný výpočet len na slovách z jazyka a pri vstupoch, ktoré z jazyka nie sú sa môže stať, že výpočet je nekonečný, vtedy TS jazyk akceptuje a príslušný jazyk je rekurzívne vyčísliteľný.