ALGORITMY

Matematické modelovanie

MARTIN TIMOTHY TIMKO

2.9. 2015 – 29.9. 2015

1 ALGORITMY

Ľudia od nepamäti používali určité postupy k tomu, aby vyriešili určité problémy. Samotná požiadavka riešiť problémy bola odjakživa daná inštinktívne. Nebolo potrebné explicitne vymedzovať problémy, pretože problémy boli súčasťou každodenného života. Človek bez ohľadu na formu týchto problémov vedel definovať problém svojím vlastným spôsobom. Už samotná definícia bola čiastočným riešením. A možno práve prienikom viacerých vlastných riešení jednotlivcov získame jediné správne riešenie.

Podstatná časť z toho ako počítače principiálne fungujú kopíruje jednotlivé funkcie ľudského mozgu. Napokon mnoho vecí okolo nás sú odrazom ľudskej činnosti. Samotné vykonávanie postupov určitých algoritmov popisujúcich nejaký problém je činnosť automatizovaná. Ľudia od pradávna riešili tieto problémy svojpomocne, ale až s nástupom počítacích mechanizmov a neskôr moderných počítačov sa stávali tieto problémy riešiteľné v čoraz kratšom čase. Formalizovať tieto problémy pomocou matematiky bolo pomerne jednoduché. Aj riešenie niektorých jednoduchších úloh bolo realizovateľné v dohľadnom čase, aj keď sa niekedy jednalo v časovom horizonte niekoľkých rokov.

Prakticky jediným problémom bolo formalizovanú úlohu vo forme algoritmu prepísať do strojových inštrukcií. Prvým vážnejším pokusom bol realizovaný tzv. **abstraktný počítací stroj** v podaní anglického matematika *Alana Mathisona Turinga*. Stroj z roku 1936 bol zostavený z čítacej a zapisovacej hlavy a z nekonečnej pásky obsahujúcej symboly. Tento abstraktný matematický model položil základy pre dnešné počítače.

V súvislosti s Turingom sa viaže na logo spoločnosti *Apple* jedna historka. V roku 1938 sa uskutočnila britská premiéra Disneyho verzie rozprávky *Snehulienky a sedem trpaslíkov* - film, v ktorom našli zaľúbenie Turing a Gödel. Zvláštne nadšenie v Turingovi vyvolávala scéna, kedy zlá kráľovná ponorí jablko do nápoja s jedom a spieva:

> *"Jablko nořím do jedu, na svět spavou smrt vedu."*

Jed potom vytvorí na povrchu jablka lebku a kráľovná sa obráti na svojho pomocníka havrana:

> *"Hle, zřídlo zla. Kdo zkázu v tobě rozpozná ? Jen rudé buď a líbezné, Ať zalíbíš se princezně."*

Kráľovná ponúkne jablko havranovi, ktorý začne divoko mávať krídlami v snahe uletieť. "Ne tobě, Sněhurce je chci dát," hovorí kráľovná a smeje sa.

> *"Až poruší jemnou slupku, jablko z mé ruky ochutná, krev se jí v žilách zastaví. A nejkrásnější budu já!"*

Táto scéna Turinga uchvátila natoľko, že si neustále pospevoval tieto verše. Možno ho priťahovala morbídna erotickosť tohto výjavu, nehľadiac na to, že scéna je krikľavou narážkou na biblický mýtus - Snehulienka je zvádzaná jablkom podobne ako biblická Eva. Avšak zatiaľ čo Eva ochutná jablko dobrovoľne, Snehulienka je obeťou dobre pripraveného podvodu zo strany kráľovny, ktorá sa snaží trikmi prinútiť Snehulienku, aby sa do jablka zahryzla, keď ju napokon presvedčí, že jablko je "kúzelné" a splní jej želanie. Nakoniec poruší "jemnú šupku", stratí psychické panenstvo a upadne do "spavej smrti".

Turing ovšem nikdy neprezradil, ktorý aspekt psycho-sexuálnej štruktúry filmu ho tak zaujal. V každom prípade je všeobecne známe, že spáchal samovraždu. Spôsob akým vykonal ukončenie života bol v jeho podvedomí prítomný dlhšiu dobu. Raz poslal svoju kamarátovi Jamesovi

Atkinsovi dopis, ktorý obsahoval návod na samovraždu, v ktorom figurovalo jablko a elektrické vedenie. Priateľom často hovoril, že si pred spaním dáva jedno jablko. A v Cambridge si ešte niekoľko týždňov po premiére filmu *Snehulienka a sedem trpaslíkov* pospevoval na chodbách v King's College:

"Jablko nořím do jedu, na svět spavou smrt vedu..."

Dodnes toto jablko priťahuje pozornosť. Je to z veľkej časti dané jeho metaforickými implikáciami. (Jablko smrti, jablko poznania - alebo snáď prílišného poznania ?) Po internete sa šíria informácie, že jablko, ktoré je na logu *Apple Computers* odkazuje na Turinga. Spoločnosť *Apple* však túto súvislosť popiera a tvrdí, že ich jablko je narážkou na *Isaaca Newtona*. Je pravda, že prvé logo Apple Computers (na obrázku vpravo), bolo navrhnuté s Newtonom sediacim pod jabloňou. Avšak prisudzované Newtonové jablko u neho nikdy nefigurovalo ako nakusnuté. Autor tohto loga argumentuje jednoducho tým, že ich nakusnuté jablko je znakom odlíšenia od ostatného ovocia, pretože Apple, okrem prvého loga z roku 1976 nikde nepoužíva logo spolu s názvom tejto značky.

Samozrejme v minulosti existovali počítacie stroje aj od iných autorov. Ak nepočítame úplne primitívne počítacie mechanizmy spred tisíc a viac rokov, tak medzi prvé pokusy patrí **Pascalov počítací stroj** z roku 1642. Tento stroj zostrojil *Blaise Pascal*, aby pomohol svojmu otcovi, úradníkovi, na automatizované sčítavanie štátnych príjmov z daní a dávok. Podobný mechanizmus počítania zostrojil aj filozof, matematik a fyzik *Gottfried Wilhelm Leibniz* v roku 1673. Do éry tzv. **diernych štítkov** nás uviedol francúz *Joseph Maria Jacquard*, ktorý vytvoril tkáčsky stroj na základe istého druhu diernych štítkov. Výrobný proces kontroloval *algoritmus*, ktorého kód bol zapísaný v podobe postupnosti vyrazených otvorov vo forme štítkov. Z dierneho štítku bolo možné prečítať dva stavy: *prítomnosť alebo neprítomnosť perforácie štítku*. To slúžilo na indikáciu uskutočnenia alebo neuskutočnenia určitej mechanickej operácie. Tieto dva stavy, *dvojstavová logika*, bola predchodcom súčasných pamätí, kde sa binárne čísla používajú pre zapísanie riadiaceho kódu algoritmu. Neskôr sa pokúsil metódu diernych štítkov prepracovať angličan *Charles Babbage*, keď vytvoril časť prototypu diferenčného stroja na výpočet niektorých matematických úloh. Jeho koncepcia tzv. **analytického stroja** nebola počas jeho života zrealizovaná kvôli problémom s financovaním projektu. Jeho idea sa zrealizovala až v roku 1992 skôr pre zaujímavosť než nejaké vedecké použitie. Prakticky boli použité dierne štítky *Hermanom Hollerithom* vo veľkom projekte na štatistické spracovanie údajov získaných pri sčítaní ľudu.

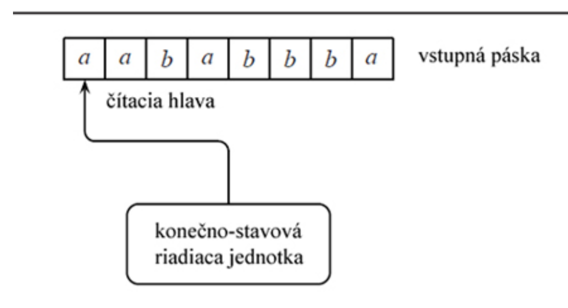
Zariadenia používajúce relé, ktoré sa nazývali "počítačom" bolo ešte niekoľko. Najznámejším elektromechanickým strojom bol **MARK 1** z roku 1944, ktorého tvorcom bol američan *Howard Hathaway Aiken*, pričom vychádzal z asi storočnej Babbageovej koncepcie. Následne vznikol už elektronický počítač **ENIAC** od *Johna Prespera Eckerta* a *Johna Williama Mauchlyho* primárne zostrojený na výpočty *balistických tabuliek*. Avšak skutočný počítač v pravom zmysle slova sa považuje **EDVAC** od *Johna von Neumanna*. Od ostatných vtedajších počítacích strojov, elektromechanických počítačov a počítačov sa zásadne odlišoval v spôsobe spracovania inštrukcií, kedy celý program bol uložený do pamäte počítača spolu so spracovávanými dátami. Táto filozofia počítačov sa ďalej rozvinula pomerne rýchlym tempom, kedy sa na trhu začali objavovať prvé programovateľné kalkulátory od spoločnosti *IBM* ako aj iných, a tiež vznikali veľké vedecké počítače, napr. *UNIVAC*.

Všetky tieto počítacie stroje v zásade používali koncepciu počítacieho stroja podobnú tzv. **konečným automatom**, ktorá bola podobná Turingovmu stroju. Inšpiráciou pre Turinga pri návrhu koncepcie jeho počítacieho stroja bol *písací stroj*, ktorý sa nápadne podobal šifrovaciemu stroju *Enigma*. Napokon v čase zostrojovania rôznych počítacích strojov vznikali aj matematické

teórie, ktoré definovali automaty vo všeobecnosti a položili tak základy modernej (teoretickej) informatiky.

V živote sa často stretávame s automatmi rôzneho druhu. Typickým príkladom je napr. *automat na kávu*. Takýto automat obsahuje display, otvor pre mince, niekoľko tlačidiel pre výber nápoja a samozrejme obsahuje tiež nejaký výdajný systém. Komunikácia prebieha pomocou tlačidiel a pokynov zobrazených na displayi. Počas celej manipulácie automatu nám slúži display ako indikátor **stavu** automatu. Úlohou automatu je vystihnúť, ktoré konkrétne akcie od používateľa vyhovujú algoritmu automatu. Existuje mnoho systémov, ktoré je možné popísať konečným počtom stavov, akcií a prechodov medzi stavmi. Nemusí sa jednať o niečo zložité, napr. spoločenská hra *šachy* obsahuje pravidlá hry, ktoré je možné chápať ako algoritmus a na základe toho algoritmu je možné definovať stavy, ktoré reprezentujú rôzne rozpoloženia figúrok na šachovnici. A keďže máme konečný počet figúrok a políčok na šachovnici, tak aj rozpoložení bude **konečný** počet. Akcie sú v tomto prípade jednotlivé možné ťahy, pričom v každom stave je možné vykonať len tie akcie, ktoré neodporujú pravidlám šachu. Väčšinou sa vždy jedná o konečný počet stavov, tzv. konečno-stavové automaty. Pred nástupom moderných počítačov rôzne teórie popisovali matematické štruktúry v nekonečnom rade, teda prísne všeobecne, ale až s nástupom počítačov sa začalo používať konečné rady. Bolo to dané tým, že počítače používajú pamäť, ktorá má obmedzenú veľkosť, pričom za nekonečnú veľkosť je možné použiť veľmi veľkú časť pamäte. Z toho vyplýva, že aj počítače sú **konečno-stavové automaty**. Ale existujú aj hry, ktoré nie je možné popísať konečno-stavovým automatom, napr. *piškvôrky*. Hracie pole je nekonečné a samotná hra je v zásade nekonečná, teda je možné vykonať nekonečne mnoho akcií.

Abstraktným modelom konečno-stavových systémov sú už spomenuté konečné automaty. Takýto automat je vybavený **konečnou pamäťou**, **čítacou hlavou** a **páskou**, na ktorej je zapísané vstupné slovo.



Na začiatku výpočtu je hlava umiestnená na najľavejšom políčku pásky. Automat na základe prečítaného symbolu a momentálneho stavu svoj stav zmení a posunie čítaciu hlavu o jedno políčko doprava. Výpočet končí, keď sa automat "zablokuje", alebo prečíta celé vstupné slovo. Slovo zapísané na páske je akceptované automatom, pokiaľ je celé prečítané a výsledný stav je niektorý z predom určených koncových stavov. Množina všetkých slov, ktorá daný konečný automat M akceptuje, tvorí **jazyk** akceptovaný automatom M. Formálna definícia pre *ilustratívne účely* vyzerá nasledovne:

Konečný automat M je pätica $(Q, \sum, \delta, q_0, F)$, kde:

- Q je neprázdna konečná množina stavov

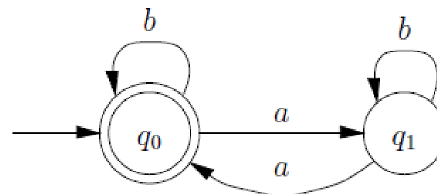
- $\sum$ je konečná množina vstupných symbolov, nazývaná tiež vstupná **abeceda**
- $\delta : Q \times \sum \rightarrow Q$ je parciálna **prechodová funkcia**
- $q_0 \in Q$ je pociatocny stav
- $F \subseteq Q$ je mnozina **koncovych stavov**

Detailnejší popis jednotlivých funkcií tohto automatu bude obsahom ďalších článkov. Pre tieto účely bude stačiť len približná predstava. V uvedenom automate δ-funkcia predstavuje akýsi **program** konečného automatu. Preto hlavnou časťou konštrukcie konečného automatu pre daný jazyk bude práve konštrukcia vhodnej **prechodovej funkcie**. Samotná prechodová funkcia sa zapisuje *prechodovou tabuľkou* alebo *prechodovým diagramom*. Na vyriešenie a pochopenie konkrétneho príkladu pre konečný automat momentálne nedisponujme potrebným teoretickým aparátom, kvôli uvedenému zjednodušeniu, takže len na ilustráciu ukážeme jednotlivé prechodové funkcie.

V nasledujúcom príklade automat zisťuje, či má vstupné slovo **párny počet písmen** a, čiže akceptuje jazyk $L = \{w|w \in \{a, b\}^* \wedge \#_a(w) \ \ je \ parny\}$.

$A:$	a	b
q_0	q_1	q_0
q_1	q_0	q_1

Prechodová tabuľka je intuitívne zrejmá. Stavy automatu sú v záhlaví riadkov a vstupné symboly v záhlaví stĺpcov. Prechodová funkcia je určená obsahom vnútorných polí tabuľky. Za počiatočný stav sa považuje prvý výskyt v riadkoch a za koncový stav sa považuje jeho posledný výskyt v riadkoch.



Prechodový diagram obsahuje jednotlivé stavy automatu vo forme zobrazených kružníc, ktorých obsahom je daný stav, pričom dvojitá kružnica reprezentuje koncový stav automatu. Na začiatočný stav v diagrame ukazuje šípka. Prechodové funkcie sú zobrazené ohodnotenými hranami podobne ako pri grafovej reprezentácii.

Existuje aj reprezentácia pomocou **výpočtového stromu**, avšak z predošlých spôsobov je táto reprezentácia najmenej používaná kvôli jednak obsiahlosti zobrazenia a jednak kvôli dosiahnuteľnosti, tzn. každý stav by mal byť dosiahnuteľný z počiatočného stavu. Jedná sa o podobný strom známy z grafových štruktúr, kde koreň stromu odpovedá počiatočnému stavu, pričom z každého uzlu, ktorý nie je listom vychádzajú ohodnotené hrany vstupnej abecedy. Výpočtový strom nie je úplne jednoznačný, pretože sa môže líšiť podľa toho, či ho konštruujeme *prechádzaním do hĺbky* alebo *šírky*. Aj z toho dôvodu sa pre lepšiu názornosť používajú predchádzajúce spôsoby.

Z doposiaľ uvedeného vyplýva pojem **algoritmus** intuitívne *ako postup pri riešení problémov, zavedený a opakovateľný výpočtový postup s konečným počtom krokov pri riešení nejakého problému*. Slovo *"algoritmus"* je odvodené od mena perzského matematika *Muhammada Ibn Músá al-Chórezmího*, ktorý okolo roku 825 napísal matematický text **Kitab al-jabr wa'l-muqabala** (od spojenia *al-jabr* je odvodené slovo *"algebra"*). Ako príklad algoritmu sa uvádza **Euklidov algoritmus** pre nájdenie *najväčšieho spoločného deliteľa* dvoch čísel. Algoritmus spočíva v tom, že sa z dvoch čísel nájde číslo, ktorým je možné bez zvyšku deliť obidve tieto čísla.

Ak si vezmeme dve čísla, napr. 4782 a 1365, tak začneme postupne deliť väčšie číslo, číslom menším:

4782 : 1365 = 3, zvyšok 687

Teraz vydelíme menšie číslo z obidvoch čísel, teda 1365, zvyškom 687.

1365 : 687 = 1, zvyšok 678

Ďalej uvedený postup opakujeme:

687 : 678 = 1, zvyšok 9, 678 : 9 = 75, zvyšok 3, 9 : 3 = 3, zvyšok 0.

Číslo 3 je teda *najvyšším spoločným deliteľom* obidvoch čísel.

Na *Wikipédii* je možné nájsť podrobnejšiu definíciu tohto algoritmu. Nás bude zaujímať, že z tohto analytického popisu vieme napísať programový kód vo forme jednoduchého pseudokódu.

Pseudokód pre *rekurzívnu implementáciu* Euklidovho algoritmu:

```
function gcd(u, v)
    if v = 0
        return u
    else
        return gcd(v, u mod v).
```

Pseudokód pre *iteratívnu verziu*:

```
function gcd(u, v)
    while v ? 0
        t := v
        v := u mod v
        u := t
    return u.
```

Z uvedeného algoritmu je možné zostrojiť rôzne diagramy, či už forme prechodového diagramu pre konštrukciu konečného automatu, ale aj tzv. *vývojový diagram*.

Existuje hľadanie najväčšieho spoločného deliteľa aj pomocou *prvočíselného rozkladu* oboch čísel ako súčin prvočísel umocnený na najmenší z exponentov pri príslušnom prvočísle v rozkladoch.

Nech $\prod_i p_i^{e_i}$ je prvočíselný rozklad čísla a a $\prod_i p_i^{f_i}$ je prvočíselný rozklad čísla b. Potom:

$$NSD(a, b) = \prod_i p_i^{min(e_i, f_i)} \tag{1}$$

Napríklad najväčšieho spoločného deliteľa čísel 136 a 204 možno nájsť zistením, že $136 = 2^3 \times 17$ a $204 = 2^2 \times 17$. V rozkladoch sa vyskytujú prvočísla 2, 3 a 17 s exponentmi 3, 0, 1 pri menšom čísle a 2, 1, 1 pri väčšom čísle. Výsledný *NSD* je potom súčinom prvočísel vyskytujúcich sa v oboch rozkladoch umocnených na príslušné najmenšie exponenty, teda $2^2 \times 17 = 68$.

Tento výpočet je ľahko pochopiteľný, ale v praxi úplne nepoužiteľný s výnimkou veľmi malých čísel, pretože získanie rozkladu na prvočísla je extrémne náročná operácia. Pre praktické výpočty slúžia výrazne rýchlejšie algoritmy, hlavne tzv. *Euklidov algoritmus*.

Možno sa zdá, že spôsob výpočtu konečným automatom je niečo úplne strojové, niečo v základe obmedzené. Keď si vezmeme krátkodobú, pracovnú ľudskú pamäť, tak takáto pamäť má tiež obmedzenú kapacitu. Sme schopní si zapamätať najviac len asi sedem položiek rôzneho druhu v režime "runtime". Z toho hľadiska môžme abstaktne považovať ľudský mozog ako určitý konečný automat. Funkciu jednoduchého matematického neurónu je možné popísať konečným automatom. Napokon *Church-Turingová* téza hovorí, že ľubovoľný algoritmický alebo fyzikálny proces je možné simulovať pomocou (kvantového) Turingovho stroja. Do akej miery sme schopní mapovať prvky konečných automatov ma ľudský mozog ako aj otázky, či môže stroj (počítač) premýšľať sú záležitosťou *kognitívnej vedy*.

Konečný automat ako taký definuje vlastnosti algoritmov, ktoré môžu byť vykonateľné na tomto automate. Každý algoritmus potom z hľadiska bezproblémovej funkčnosti musí spĺňať určité vlastnosti. Jednou z vlastností je **konečnosť**. Každý algoritmus musí skončiť v konečnom počte krokov, za nejaký konečný čas (aj keď rádovo by sa niekedy mohlo jednať o roky). Navyše každý takýto krok by mal byť jednoduchý, **elementárny**. Písať algoritmus pre jeden problém je síce pohodlnejšie, ale neefektívne, teda, algoritmus by okrem toho mal byť aj **obecný**, riešiť problém vo všeobecnosti ako určitú šablónu, vzor, so širokým spektrom vstupov a výstupov. Zároveň by bolo zvláštne, ak by sme pre rovnaké vstupy dostali zakaždým iné výstupy. Takže v každom časovom okamihu musí byť dodržaná určitá jednoznačnosť, ktorá musí byť jasne definovaná. Stále musí byť zrejmé, čo a ako sa má vykonať, teda algoritmus by mal byť **deterministický**. Mal by byť, to znamená, že nie všetky algoritmy sú deterministické, napr. existujú *pravdepodobnostné algoritmy*, ktoré v sebe majú zahrnutú určitú *náhodu*. Ak by sme navrhli algoritmus, ktorý by síce spĺňal všetky doposiaľ uvedené vlastnosti, ale neposkytoval by nám žiadny výstup, tak by takýto algoritmus nemal veľký zmysel. Každý algoritmus musí okrem iného riešiť nejaký problém, aj keď rôznymi spôsobmi, vo forme aspoň jedného výstupu k prislúchajúcim vstupným parametrom.

Touto problematikou sa bližšie zaoberá vedná oblasť informatiky nazývaná **teória zložitosti**, **algoritmická analýza**, alebo tiež **teória vyčísliteľnosti**. V týchto oblastiach sa analyzujú rôzne druhy algoritmov z hľadiska časovej a priestorovej náročnosti, alebo sa len zisťuje fakt, či je daný problém možné vyriešiť bez ďalších atribútov náročnosti.

Čo sa týka konkrétnych druhov algoritmov, jedná sa o algoritmy, ktoré riešia určitú skupinu problémov. Napr. **rekurzívne algoritmy**, ktoré používajú tzv. *volanie samých seba*, kde je nejaká funkcia, ktorá vo svojom tele opakovane volá funkciu, ktorou je definovaná. Problémom takýchto funkcií je veľká pamäťová náročnosť, keďže sa funkcia musí mnohonásobne volá. Ľahko sa môže stať, že sa daná funkcia zacyklí a vykonávanie programu sa zastaví, alebo ho operačný systém vynútene ukončí po určitom čase dlhšieho zastavenia, alebo, ak sa náhodou dostane program do nejakej neočakávanej pamäťovej chyby, napr. často pretečenie zásobníka. Potom existujú ako už bolo spomenuté aj **pravdepodobnostné algoritmy**, ktoré vo svojom vykonávaní majú určitý *náhodný charakter*, takže sa určitým spôsobom mení stav medzi vstupmi a výstupmi. Umelá inteligencia ako odbor používa aj **genetické algoritmy** na napodobovanie *biologických evolučných procesov*, pričom tieto algoritmy slúžia okrem svojho pôvodného zámeru aj na vyriešenie konvenčných problémov, teda majú širšie uplatnenie. Nie vždy je požiadavka nájsť presné riešenie problému. Na to slúžia tzv. **heuristické algoritmy**. Tieto algoritmy hľadajú *približné riešenie daného problému*, pretože napr. čas nepostačuje na použitie primárnych algorit-

mov. Doposiaľ všetky uvedené typy algoritmov (pričom existuje väčší počet) sa môžu vzájomne kombinovať, čím sa spektrum vyriešenia problému rozšíri.

Pri vývoji moderných počítačových programov sa veľa pozornosti venuje definícii a porozumení riešeného zadania ako aj jeho analýze z hľadiska zložitosti a najmä **dekompozícii**, teda rozdelenie problému na menšie časti, ktoré sa následne prehľadnejšie a ľahšie implementujú. Existujú algoritmy, ktoré z hľadiska svojej zložitosti sa ťažko implementujú s požiadavkou na optimalizáciu systémových zdrojov. Niektoré tieto algoritmy budú obsahom nasledujúcich článkov.

Napokon, tak ako sme sa pokúšali naznačiť pojem algoritmu, vytváranie programu vo všeobecnosti, tak existuje, na počudovanie, aj **testovanie správnosti programu**. Kedysi veľmi dávno sa písali programy bez výrazných štandardizačných techník. Až s nástupom modernej doby počítačov začali vznikať rôzne metodiky korektného programovania. Cieľom bolo vytvorenie určitých špecifikácií na zabezpečenie konzistencie programu (softwaru). Aj napriek tomu, že sa jedná o pomerne novú disciplínu v softwarovom inžinierstve, ide o najmenej populárnu časť programovania, pričom niektoré metodiky ostávajú aj po prepracovaní diskutabilné.

V nasledujúcich článkoch budú popísané niektoré ďalšie časti o návrhu a analýze algoritmov.

2 ANALÝZA ALGORITMOV

V dnešnej dobe existuje už pomerne obrovské množstvo algoritmov. Kedysi dávno ešte v prvopočiatkoch počítačov záležalo do veľkej miery na tom, ktorý typ algoritmu a aká implementácia sa použije, pretože počítače boli na dnešné pomery výrazne pomalšie a každá snaha o úsporu času sa v konečnom dôsledku oplatila. V súčasnosti je rýchlosť počítačov natoľko vysoká, že nie vždy sa nutne musí zvoliť efektívnejší algoritmus, pretože napr. menej efektívny algoritmus sa jednoduchšie implementuje a je ľahko pochopiteľný. To ale neznamená, že analyzovať algoritmy z rôzneho hľadiska nemá význam. Uplatní sa to najmä pri veľkých a zložitých počítačových programoch.

Pri riešení rôznych úloh sa často stretávame s mnohými odlišnými prístupmi. Pre malé úlohy často nezáleží na tom, aký prístup zvolíme, pokiaľ máme nejaký taký, ktorý daný problém dokáže vyriešiť určitým spôsobom v dohľadnom čase a navyše bezchybne. Ibaže v rozsiahlych úlohách nás pomerne skoro napadne riešiť dané úlohy nejakou efektívnou metódou.

Keď je vyvíjaný veľký alebo zložitý počítačový program je venované pomerne veľké množstvo definícii a porozumeniu riešeného zadania, dekompozícii na menšie úlohy, ktoré môžu byť ľahšie implementované. Často sa vyskytujú algoritmy, ktorých voľba je problematická, pretože sa na ich funkčnosť spotrebováva rôzne množstvo systémových zdrojov. V podstate práve takéto typy algoritmov sú vo veľkej miere predmetom analýzy algoritmov, pretože má zmysel ich nejakým spôsobom optimalizovať. Aj keď ako už bolo spomenuté niekedy zbytočne dochádza k "preoptimalizácii" v tom zmysle, že nie vždy je prvoradé použiť najefektívnejšiu metódu, niekedy stačí použiť jednoduchú a účinnú metódu.

Aby sme mohli použiť algoritmus efektívne za účelom riešenia rozsiahleho problému, ktorý nie je možné riešiť jednoduchou cestou, alebo keď chceme vykonávať efektívnu implementáciu, musíme byť oboznámený s výkonnostnými charakteristikami algoritmov. Vo všeobecnosti môžme algoritmy zapisovať v prirodzenom jazyku alebo v niektorom vyššom programovacom jazyku (napr. jazyk C, Pascal), alebo priamo v operačnom kóde počítača. Pre teoretické účely existujú rôzne modely abstraktných počítacích strojov (napr. Turingov stroj alebo RAM) a algoritmy sa zapisujú v ich strojovom kóde. Okrem toho je možné zapisovať algoritmy aj tzv. psedokódom, čo je akási zjednodušená verzia často niektorého vyššieho programovacieho jazyka, ale samozrejme sa môže jednať o akýkoľvek iný abstraktný programovací jazyk. Dôležité pre všetky tieto programovacie jazyky je *zápis*, ktorým vyjadrujeme určité inštrukcie, ktoré sa majú vykonať v nejakej postupnosti.

Výkonnosť algoritmov najčastejšie posudzujeme jednoduchým porovnaním viacerých algoritmov z hľadiska času ich spracovania. Takýto prístup je **empirický, aposteriórny**, kedy v konečnom dôsledku zvolíme algoritmus s najkratším trvaním výpočtu. Nemusíme sa zamýšľať nad obsahom daných algoritmov, stačí vedieť aké sú vstupy a výstupy, pričom predpokladáme, že implementácia každého zo sledovaných algoritmov adekvátne rieši daný problém. Pokiaľ sa jedná rádovo o časy niekoľko minút, prípadne pár hodín výpočtu, tak môže byť empirická analýza vhodným prediktorom výkonnejšieho algoritmu. Lenže niekedy časová náročnosť výpočtu algoritmov prekračuje rádovo niekoľko hodín až desiatok hodín, a potom nedokážeme adekvátne určiť, ktorý z algoritmov je rýchlejší, alebo efektívnejší (tzn. napr. že môže byť o niečo pomalší, ale pri väčšej záťaži naopak rýchlejší). Teda, keď empirické štúdie začínajú spotrebovávať značné množstvo času a prostriedkov, prichádza na rad **analytický prístup, apriórny**.

Matematická analýza sa stala nástrojom pre optimalizáciu algoritmov. Ľudia vymysleli určité pravidlá algoritmov na základe ich rozsiahleho študovania. Stanovili čas výpočtu algoritmov

Krok	Počet opakovaní
M1	1
M2	n
M3	$n-1$
M4	A
M5	$n-1$

na základe matematických veličín a následne tieto veličiny podrobili matematickej analýze. Zdá sa, že zavedením matematických veličín bude každá analýza zmysluplná a riešiteľná, avšak stáva sa, že analýza vedie k neriešiteľným matematickým problémom, najmä kvôli komplexnosti implementácie alebo vstupno-výstupným parametrom.

Pri analýze algoritmov sa obvykle začína **identifikáciou abstraktných operácií**, na ktorých je daný algoritmus založený, aby sa oddelila analýza od implementácie. Budeme sa napr. pozerať koľko-krát jedna z implementácii algoritmov vykoná časť kódu

```
i = a[i];
```

od analýzy toho, koľko sekúnd bude požadované k vykonaniu určitej časti kódu na určitom počítači. Prvá časť je určená vlastnosťami algoritmu, druhá časť vlastnosťami počítača. Toto oddelenie nám umožní porovnať algoritmy, ktoré nebudú závislé na použitej implementácii, alebo od použitého počítača. Samozrejme počet obsiahnutých operácií v algoritme môže byť veľký, ale výkonnosť algoritmu závisí len na niekoľkých najdôležitejších veličinách, pričom práve tieto veličiny sú pre analýzu pomerne ľahko identifikovateľné. V jazyku C existuje tzv. **profilovací mechanizmus**, ktorý poskytuje počty použitých výskytov inštrukcií a na základe ktorých zistíme najpoužívanejšie časti programu. Okrem toho musíme tiež študovať dáta a modelovať vstupy. Buď budeme predpokladať, že vstupy sú *náhodné* a študovať výkonnosť algoritmu pre akýsi priemer, alebo budeme hľadať *zmiešané dáta* a študovať najhoršiu možnú výkonnosť programu.

Pri takýchto a podobných analýzach sa často používajú rôzne matematické pojmy, avšak najčastejšie *permutácie*, *faktoriály*, *binomická veta* (kombinácie), *Stirlingová aproximácia* a napokon tzv. *vytvárajúce funkcie* (alebo tiež generujúce). Všetky tieto pojmy ako aj ďalšie sa budú priamo zúčastňovať na analýze algoritmov.

Algoritmus M pre vyšetrenie analýzy je nasledovný. Nech je daných n prvkov $X[1], X[2], \dots, X[n]$. Máme nájsť také m a j, pre ktoré je $m = X[j] = max_{1 \leqslant i \leqslant n} X[i]$, kde j je najväčší index. M1 :[Inicializácia] Priradiť $j \leftarrow n$, $k \leftarrow n-1$, $m \leftarrow X[n]$ M2 :[Sú všetky testy hotové?] Ak je $k = 0$, koniec algoritmu. M3 :[Porovnanie] Ak je $X[k] \leqslant m$, prejdi na M5 M4 :[Zmena k] Priradiť $j \leftarrow k$, $m \leftarrow X[k]$ (hodnota m je nové aktuálne maximum) M5 :[Zníženie k] Znížiť k o jedničku a vrátiť sa na krok M2

Jedná sa o konečný počet vstupných prvkov, takže budeme analyzovať čas potrebný na výpočet týchto prvkov. Každý krok algoritmu posudzujeme podľa toho, koľko-krát sa vykonal.

Ak vieme, koľko-krát sa každý krok vykonal, máme všetky informácie, ktoré potrebujeme pre stanovenie času vykonania daného algoritmu. Máme všetky informácie o počte opakovaní jednotlivých krokov, okrem kroku M4, teda hodnoty A, ktorá vyjadruje počet zmien hodnoty aktuálneho maxima. Pri zisťovaní hodnoty A použijeme metódy matematickej štatistiky, kedy budeme analyzovať minimálnu, maximálnu a priemernú hodnotu, pričom tiež tzv. štandardnú odchýlku hodnoty A, tzn. nakoľko sa hodnota priemeru nachádza v blízkosti skutočnej očakávanej hodnoty.

Situácia	Hodnota A
$X[1] \prec X[2] \prec X[3]$	0
$X[1] \prec X[3] \prec X[2]$	1
$X[2] \prec X[1] \prec X[3]$	0
$X[2] \prec X[3] \prec X[1]$	1
$X[3] \prec X[1] \prec X[2]$	1
$X[3] \prec X[2] \prec X[1]$	2

Minimálna hodnota A je nula:

$$X[n] = max_{1 \leqslant k \leqslant n} X[k] \tag{2}$$

Maximálna hodnota A je $n - 1$, pre prípad:

$$X[1] > X[2] > ... > X[n] \tag{3}$$

Priemerná hodnota je niekde medzi 0 a $n - 1$. Rýchlosť algoritmu nezáleží na presných hodnotách $X[k]$, len na ich relatívnom poradí. Pre $n = 3$ môžme uvažovať nasledujúce možnosti:

Priemerná hodnota A pre $n = 3$ je $(0 + 1 + 0 + 1 + 1 + 2)/6 = 5/6$. Všetky hodnoty sú rovnako pravdepodobné a pravdepodobnosť, že A nadobudne hodnoty k, môžme napísať ako

p_{nk} = (počet permutácii n prvkov, pre ktoré $A = k$) / $n!$

Potom priemernú hodnotu vypočítame podľa vzťahu:

$$A_n = \sum_k k p_{nk} \tag{4}$$

Rozptyl V_n je definovaný ako priemerná hodnota $(A - A_n)^2$, teda:

$$V_n = \sum_k (k - A_n)^2 p_{nk} = ... = \sum_k k^2 p_{nk} - A_n^2 \tag{5}$$

Napokon štandardná odchýlka σ_n je definovaná ako $\sqrt{V_n}$.

Postupnosť vieme efektívne zapísať vo forme funkcie, tzv. vytvárajúcou (generujúcou) funkciou, ktorá v sebe bude obsahovať všetky informácie o danej postupnosti.

Všeobecne sa vytvárajúca funkcia zapisuje nasledovne:

$$G_n(z) = p_{n0} + p_{n1} z + ... + p_{nk} z^k = \sum_k p_{nk} z^k \tag{6}$$

Odtiaľ:

$$p_{nk} = \frac{1}{n} p_{(n-1)(k-1)} + \frac{n-1}{n} p_{n-1} k \tag{7}$$

Následne po dosadení dostávame generujúcu funkciu vo všeobecnom tvare:

$$G_n(z) = \frac{z}{n} G_{n-1}(z) + \frac{n-1}{n} G_{n-1}(z) = \frac{z+n-1}{n} G_{n-1}(z) \tag{8}$$

Napokon konkrétne po dosadení dostávame:

$$G_n(z) = \frac{z+n-1}{n} G_{n-1}(z) = \frac{z+n-1}{n} \frac{z+n-2}{n-1} G_{n-2}(z) = ... = \frac{1}{z+n} \binom{z+1}{n} \tag{9}$$

Efektívnosť tohto algoritmu definujeme ako tzv. *asymptotickú zložitosť*, ktorá charakterizuje ako rastie zložitosť algoritmu s rastúcou dĺžkou vstupných parametrov. Stačí si všímať *počet vnorených cyklov* a *počet jednotlivých porovnávaní*. Avšak nie vždy je to také jednoznačné a priamočiare. V našom prípade potrebujeme vyšetriť každú položku z celkového počtu vstupných parametrov, takže v zásade bude zložitosť lineárna, tzn. s narastajúcim počtom vstupných parametrov začne narastať aj dĺžka výpočtu algoritmu. Avšak napr. na vyhľadávanie položiek v telefónnom zozname je možné použiť lineárne vyhľadávanie, obdoba nášho algoritmu M, tzn. prehľadávanie každej položky v priamej postupnosti, alebo binárne vyhľadávanie, tzn. prehľadávanie položiek v nepriamej postupnosti podľa určitej metodiky, napr. rozdelením a porovnávaním konečného počtu vstupných parametrov na niekoľko častí utriedenej postupnosti. Akékoľvek takéto rozdelenie postupnosti bude mať za následok lepšie časové charakteristiky, presnejšie povedané bude sa jednať o logaritmickú zložitosť. Navyše často sa lineárna a logaritmická zložitosť pre efektívnejšie výsledky vzájomne kombinujú.

3 ZLOŽITOSŤ ALGORITMOV

Pri analýze algoritmov sa bežne používa pojem zložitosti konkrétneho algoritmu. Vždy sa uvažuje, v akej zložitosti daný algoritmus pracuje. Zložitostí dokonca môže byť niekoľko. Ale, čo to vlastne znamená, keď povieme, že je niečo zložité ? Zložitým alebo náročným by sa mohli javiť aj niektoré umelecké obrazy, napr. *Salvador Dalí* maľoval obrazy, ktoré boli skôr mámením mysle, než zraku, svojím spôsobom obsahovali prvok zložitosti, niekedy vo forme mnohoznačnosti, inokedy v jasnozrivej jednoduchosti, napr. dielo z roku 1938 *Apparition of Face and Fruit Dish on a Beach*.

Pozrieme na obraz a vieme povedať, že je pomerne zložitý. Sledovaním rôznych útvarov na obraze a hľadaním súvislostí medzi nimi sme usúdili, že čím dlhšie sa pozeráme a analyzujeme, tým nielenže vzniká viac asociácií, ale niektoré komponenty nevieme patrične prisúdiť k celkovému obrazu. A možno práve tá neistota, ktorá si vyžaduje viac času je tou povestnou zložitosťou. Neistota vo forme čohokoľvek môže predstavovať akési operácie, ktoré musí mozog vykonať v určitom čase bez ohľadu na správnosť výsledku. Podobné operácie vykonáva počítač v určitom čase z čoho dedukujeme tzv. **zložitosť**.

V tomto zmysle môžme považovať napr. spoločenskú hru šachy za zložitejšiu než hru piškôrky a to aj napriek faktu, že šach má uzavreté hracie pole narozdiel od piškvôriek. Šachy, ktorých pravidlá hry sú determinované rôznymi pravidlami pre každú figúrku zvlásť, ponúkajú väčšiu variabilitu hry ako celku, pričom jednoduchosť a priamočiarosť piškôriek takúto variabilitu neponúka. Z toho je možné usúdiť, že spotrebujeme viac **času** pri narastajúcom počte ťahov figúrok v šachovnici než pri hre piškvorky. Môžme si všimnúť aj **priestor** oboch týchto hier, v zmysle konečnosti a nekonečnosti hracieho poľa.

Práve zložka času a priestoru je dôležitá pri posudzovaní zložitosti nie len z hľadiska antropologickej povahy, ale aj technickej, strojovej, v širšom význame fyzikálnej povahy. Preto moderná informatika rozlišuje vzťahy časovej a priestorovej zložitosti.

V podstate v širšom význame sa zložitosť algoritmov vyjadruje analogicky s popisom výpočtu *priemernej* hodnoty uvedenej v článku *Analýza algoritmov* do tej miery, že počítame nejakú minimálnu hodnotu algoritmu, strednú hodnotu a maximálnu hodnotu.

Doposiaľ je zrejmé, že nosnou zložkou výpočtu zložitosti algoritmov bude čas. Ale vzhľadom k tomu, že rôzne počítače sú rôzne rýchle nie je tento spôsob pre teoretické výpočty vhodný. Lepšie je vyjadrovať zložitosť počtom vykonaných operácií, ktorý bude na všetkých počítačoch rovnaký. Avšak aj toto vyjadrenie má svoje záludnosti, pretože takýto spôsob je závislý na použitom programovacom jazyku a prekladači. Pragmaticky vzaté, úplne presné by bolo počítať operácie na úrovni strojového kódu, ale tento spôsob sa v dnešnej dobe už nepoužíva. V každom prípade čas v spojitosti s počtom operácií rôzneho charakteru bude tvoriť vo všeobecnosti aj naďalej nosnú zložku výpočtu zložitosti.

Zložitosť ako doba výpočtu sa udáva vo forme **funkčnej závislosti**, ktorá závisí od veľkosti vstupných dát. Jedná sa o klasické matematické vyjadrenie funkčných závislostí, napr. **lineárna závislosť** vo forme zápisu

$$T(n) = an + b \tag{10}$$

kde n je veľkosť vstupných dát a a, b sú konštanty, ktoré v sebe ukrývajú konštantný počet operácií v strojovom kóde. Táto závislosť predstavuje zložitosť, ktorá bude rásť lineárne so vzrastajúcou veľkosťou vstupných dát. Ono v zásade nezáleží na konkrétnych hodnotách konštánt ako skôr type funkčnej závislosti. Teda, konštanty a a b sa nebudú meniť v závislosti rozsahu dát n. Z toho vyplýva, že pri veľkých hodnotách n bude hodnota b zanedbateľne malá vzhľadom k členu an, takže sa stačí zaoberať len konštantou a. O konštantách a a b už vieme, že závisia od strojového kódu, a teda môžu nadobúdať rôzne hodnoty v závislosti na použitom prekladači. Ak vyberieme nejakú maximálnu konštantu, ktorú si označíme ako c, môžme tvrdiť, že pre dostatočne veľké n (od istého n_0) bude platiť nerovnosť $T(n) \leqslant cn$, a to sa označuje aj ako $T = O(n)$, čo predstavuje tzv. **asymptotické** (porovnávacie) správanie funkcie T.

V matematike a informatike funkcie definované na prirodzených číslach zvykneme porovnávať podľa ich správania v závislosti od n. Tento postup sa nazýva ako tzv. *asymptotická analýza*, alebo *asymptotické správanie*, ktoré predstavuje nejaké porovnanie pre základné funkcie. Hľadáme v zásade určitý **odhad** funkcie, pomocou ktorého určujeme **efektívnosť** algoritmov. Tento odhad zaviedol *Paul Bachmann* v roku 1894 ako tzv. **O-notáciu** pre približné vyjadrenie, čím môžme znamienko $\approx$ vymeniť za znamienko $=$, a následne napísať nasledovný vzťah:

$$H_n = \ln n + \gamma + O\left(\frac{1}{n}\right) \tag{11}$$

Obecne je možné zápis $O(f(n))$ použiť pre každú funkciu $f(n)$ definovanú nad kladným celým číslom n. Vyjadruje veličinu, ktorá nie je explicitne známa, ale o ktorej vieme, že nie je príliš veľká. Zápis $O(f(n))$ znamená nasledovné:

Existujú kladné konštanty M a n_0 také, že číslo x_n vyjadrené funkciou $O(f(n))$ spĺňa podmienku $|x_n| \leqslant M\,|f(n)|$ pre všetky celé čísla $n \geqslant n_0$.

Zo vzťahu $H_n = \ln\, n + \gamma + O\left(\frac{1}{n}\right)$, z uvedeného znamená, že $|H_n - \ln n - \gamma| \leqslant \frac{M}{n}$ pre $n \geqslant n_0$.

Z hľadiska asymptotickej reprezentácie to však znamená, že ak máme dva algoritmy z rovnakej triedy zložitosti násobené nejakou konštantou, tak na dosiahnutie rovnakých výsledkov nám stačia dva rôzne rýchle počítače, čím problém pomerne ľahko vyriešime. Avšak, ak máme dva algoritmy z rôznych tried zložitosti, tak bez ohľadu násobenia konštantami nám na porovnanie výkonu nepomôže ľubovoľne výkonný počítač, pretože dĺžka výpočtu pre určitý objem vstupných dát bude u dvoch algoritmov z rôznych tried odlišná.

V zásade sme počítali približné odhady zložitosti v **najhoršom prípade**, **priemernom prípade** a v **najlepšom prípade**, čo sa konkrétne označuje ako tzv. *horný*, *stredný* a *dolný* odhad.

Určenie zložitosti nejakého algoritmu teda nie je definované len jedným typom zložitosti. Pretože vzhľadom k meniacim sa vstupným dátam sa môže meniť (a často sa aj mení) celkový charakter doby výpočtu algoritmu. Čiže napr. je možné, že nejaký algoritmus bude dobre fungovať pre malý rozsah vstupných dát, avšak pri narastajúcom počte týchto vstupných dát sa charakter fungovania zmení a celkovo môže vykazovať horšiu efektívnosť. Väčšinou, ak sa niekde uvádza zložitosť algoritmu, tak sa jedná o *priemernú (aj tzv. očakávanú) zložitosť*, pretože sa predpokladá, že daný algoritmus bude práve v tejto charakteristike podávať efektívne výkony. Niektoré algoritmy pracujú efektívnejšie v menších objemoch dát, a teda vzájomne sa môžu jednotlivé charakteristiky algoritmov kombinovať, čím vo výsledku môže vzniknúť veľmi efektívne pracujúci algoritmus.

Ako bolo spomenuté, parameter n udáva veľkosť vstupných dát. Samotné vstupné dáta môžu reprezentovať počet znakov v textovom reťazci, veľkosť triedeného súboru, alebo inú abstrakciu uvažovanej veľkosti úlohy. Vo všeobecnosti tento parameter ovplyvňuje dĺžku výpočtu a vyjadruje sa pomocou niektorých matematických vzťahov:

- $O(1)$, väčšina inštrukcií je vykonávaná s jednou alebo malou početnosťou. Ak algoritmus vykazuje takúto vlastnosť, hovoríme, že dĺžka výpočtu je konštantná.

- $O(\log n)$, ak je dĺžka výpočtu logaritmická, potom sa algoritmus mierne spomaľuje tým, ako narastá n. Takýto priebeh sa vyskytuje tam, kde algoritmus rieši veľké zadanie transformáciou na sadu menších tak, že rozdeľuje veľkosť na rovnaké menšie diely pre každý krok.

- $O(n)$, keď je dĺžka výpočtu lineárna, jedná sa o prípad, kedy je pre každý vstupný element vykonané malé množstvo spracovania. Ak sa n zdvojnásobí, potom sa zdvojnásobí aj dĺžka výpočtu. Je to vhodné pre algoritmy, ktoré musia spracovávať n vstupov (alebo vytvárať n výstupov).

- $O(n \log n)$, sa objavuje v prípade, ak algoritmy riešia problém rozštiepením na menšie problémy, ktoré potom riešia nezávisle a následne potom kombinujú riešenia. Ak sa n zdvojnásobí, dĺžka výpočtu sa o niečo viac zdvojnásobí (ale nie príliš).

- $O(n^2)$, ak je dĺžka výpočtu kvadratická, potom je tento algoritmus vhodný len pre malé problémy. Objavuje sa v problémoch, ktoré spracovávajú všetky páry dátových položiek,

napr. v dvojnásobne zahniezdenej slučke. Kedykoľvek sa n zdvojnásobí, dĺžka výpočtu vzrastie štyri-krát.

- $O(n^3)$, sa vyskytuje pri spracovávaní trojice dátových položiek a majú tak kubickú dĺžku výpočtu, napr. trojnásobne zahniezdenej slučke a sú vhodné len pre riešenie malých problémov. Kedykoľvek sa n zdvojnásobí, dĺžka výpočtu vzrastie osemkrát.
- $O(2^N)$, len málo algoritmov s exponenciálnou dĺžkou výpočtu je vhodných pre praktické použitie, aj keby taký algoritmus vznikol, aby vyriešil hrubou silou nejaký problém. Kedykoľvek sa n zdvojnásobí, dĺžka výpočtu vzrastie s druhou mocninou.

Podrobnejší popis použitia jednotlivých tried zložitostí v dátových štruktúrach, triediacich algoritmov, grafových štruktúr a manipulácií z haldami je možné možné nájsť na *Big-O Algorithm Complexity*. Prehľad niektorých tried zložitostí je možné vidieť na nasledujúcom grafe.

Dĺžka výpočtu nejakého algoritmu je konštanta násobená jednou z uvedených podmienok a nejakou ďalšou menšou podmienkou. Hodnoty konštantných koeficientov a zahrnutých podmienok závisia od výsledkov analýzy a na detailoch implementácie.