Úvod do teoretické informatiky

Zdeněk Sawa

Verze: 5. ledna 2021

VŠB–Technická Univerzita Ostrava

EVROPSKÁ UNIE
Evropské strukturální a investiční fondy
Operační program Výzkum, vývoj a vzdělávání

Obsah

1 Úvod **1**

I Algoritmy **13**

2 Algoritmy **15**

2.1 Popis algoritmů pomocí pseudokódu . . . 15

2.2 Řídící tok . . . 18

2.3 Výpočet algoritmu . . . 25

3 Korektnost algoritmů **31**

3.1 Invarianty . . . 32

3.2 Konečnost výpočtu . . . 41

4 Výpočetní složitost algoritmů **45**

4.1 Velikost vstupu . . . 46

4.2 Doba výpočtu algoritmu . . . 48

4.3 Časová složitost algoritmu . . . 50

A Symboly **55**

Kapitola 1

Úvod

Cílem této úvodní kapitoly je popsat, co je náplní teoretické informatiky, čím se tento vědní obor zabývá a jaké jsou typické otázky, které se zde zkoumají.

Když se řekne například fyzika, biologie nebo třeba geometrie nebo metalurgie, tak každý ví, co si pod těmito slovy představit a čím se zhruba tyto obory zabývají. Když se řekne teoretická informatika, tak asi jen menšina lidí má jasnou představu, co to vlastně je a co je náplní tohoto oboru. Tento úvodní text by měl dát alespoň rámcovou představu o teoretické informatice jako celku, o alespoň některých jejích podoblastech a o typických problémech, které se v rámci teoretické informatiky řeší.

* * *

Teoretická informatika je značně široký obor ležící na pomezí mezi informatikou, matematikou a logikou. Teoretická informatika studuje nejrůznější typy problémů, které se objevují v informatice, přičemž pro zkoumání a analýzu těchto problémů používá matematické prostředky. To, co odlišuje teoretickou informatiku od jiných oblastí informatiky a co je pro ni charakteristické, je rigorózní matematický přístup. Důraz je kladen především na důkazy a přesné formální definice pojmů, se kterými se pracuje. Většina výsledků v teoretické informatice, ať už jsou publikovány na odborných konferencích, v časopisech či knihách, má podobu důkazu nějakého tvrzení.

Jedním z ústředních pojmů zkoumaných v teoretické informatice je pojem ***algoritmu***. Tím se teoretická informatika odlišuje od jiných oblastí matematiky. V matematice se obecně zkoumá, ***co*** platí, co je pravda, zatímco informatika se soustředí především na to, ***jak*** to spočítat, tj. soustředí se na samotný proces výpočtu.

Algoritmem se rozumí nějaký mechanický postup, jak něco vypočítat, přičemž tento postup je typicky realizován ve formě počítačového programu. Slovo „vypočítat“ je zde ovšem chápáno v značně obecném smyslu, neboť nemusí jít o počítání v pravém slova smyslu, tak, jak se tomuto slovu obvykle rozumí v běžné řeči, ale je jím myšlena libovolná transformace nějakých vstupních dat na data, která vydá algoritmus jako výstup.

Algoritmy obecně slouží k řešení nějakých ***problémů***. Slovo „problém“, které má v běžné řeči značně široký význam, je zde použito ve specifickém smyslu, který se od významu slova „problém“ v běžné řeči poněkud liší. Z toho důvodu se též někdy mluví o ***algoritmickém problému***, aby se tento rozdíl zdůraznil. V informatice se totiž pod pojmem problém myslí specifikace toho, ***co*** má daný algoritmus vypočítat. Konkrétně sestává popis takového algoritmického problému z popisu toho, co je vstupem (tj. jaký typ dat má algoritmus očekávat na vstupu), popisu toho, co je výstupem (tj. jaký typ dat má algoritmus vyprodukovat jako svůj výstup) a popisem toho, co má daný výstup splňovat vzhledem ke vstupu.

Pro ilustraci uveďme příklady několika takových typických algoritmických problémů. Příkladem velmi jednoduchého problému je třeba problém sečíst dvě čísla:

> VSTUP: Přirozená čísla x a y.
>
> VÝSTUP: Přirozené číslo z takové, že $z = x + y$.

Algoritmus pro řešení tohoto problému čtenář jistě zná, protože postup, jak sečíst dvě čísla je něco, co se učí děti na prvním stupni základní školy.

O něco složitější je třeba problém třídění:

VSTUP: Posloupnost přirozených čísel $a_1, a_2, \ldots, a_n$.

VÝSTUP: Čísla $a_1, a_2, \ldots, a_n$ seřazená od nejmenšího po největší.

S různými algoritmy pro řešení tohoto problémů se asi setká každý, kdo začne studovat programování a algoritmy. Ze známých algoritmů uveďme třeba třídění přímým vkládáním, třídění přímým výběrem, Bubblesort, Quicksort, Heapsort či Mergesort. (Pro jednoduchost bylo ve výše uvedeném popisu problému třídění uvedeno, že se třídí přirozená čísla. Toto však není pro tento problém podstatné, protože třídit se dají prvky libovolného typu. Stačí, pokud je pro daný typ prvků definováno příslušné uspořádání, tj. musí být specifikováno, kdy je jeden prvek menší nebo naopak větší než jiný prvek.)

Jako třetí příklad problému uveďme problém z oblasti teorie grafů — problém hledání nejkratší cesty v orientovaném grafu:

VSTUP: Orientovaný graf G, ve kterém jsou hrany ohodnoceny přirozenými čísly, vyjadřujícími délku jednotlivých hran, a dvojice vrcholů tohoto grafu u a v.

VÝSTUP: Nejkratší cesta z vrcholu u do vrcholu v, případně informace, že žádná taková cesta neexistuje. (Pokud je nejkratších cest víc, může být výstupem libovolná z nich.)

Příkladem algoritmu, který řeší tento problém, je třeba Dijkstrův algoritmus, se kterým se čtenář možná již někdy setkal v nějakém jiném kurzu.

Algoritmické problémy tedy specifikují, ***co*** se má vypočítat, neurčují ale jak. Rovněž je důležité, že algoritmické problémy jsou formulovány ***obecně***. Například u problému sčítání dvou přirozených čísel není tento problém formulován tak, že se má určit, kolik je součet čísel 125 a 451, nýbrž chceme postup, který spočítá součet pro ***libovolná*** dvě čísla x a y. Podobně problém třídění nespočívá v tom, umět setřídit posloupnost $5, 1, 8, 3, 2, 6$, ale chceme, aby daný algoritmus fungoval pro ***jakoukoliv*** posloupnost $a_1, a_2, \ldots, a_n$.

Konkrétní vstupy (jako dvojice čísel 125, 451 u problému sčítání nebo jako posloupnost

$$5, 1, 8, 3, 2, 6$$

u problému třídění) se označují jako ***instance*** problému. U algoritmu tedy nejde o to, umět najít odpovídající řešení (tj. odpovídající výstup) jen pro nějaké konkrétní instance problému, ale o postup, který funguje pro ***všechny*** přípustné instance.

Algoritmus musí popisovat konkrétní postup, jak transformovat vstupní data na výstupní, přičemž se tento postup skládá z nějakých jednoduchých elementárních kroků. Tyto kroky musí být navíc takového typu, aby je bylo možno vykonávat zcela mechanicky, bez nutnosti rozumět tomu, co se počítá, jinými slovy, aby tento postup bylo možno popsat instrukcemi nějakého programovacího jazyka.

* * *

O algoritmu se řekne, že ***řeší*** daný problém, jestliže platí, že pro ***každý*** vstup, který je přípustným vstupem pro daný problém, se algoritmus po konečném počtu kroků zastaví a vydá výstup, který odpovídá tomu, co je popsáno ve specifikaci daného problému. O takovém algoritmu se řekne, že je ***korektní***. Pojem korektnosti algoritmu se tedy vždy vztahuje k nějakému problému, který má daný algoritmus řešit. Není to vlastnost algoritmu jako takového.

Speciálně tedy algoritmem, který by řešil daný problém, není postup, který by vyžadoval provedení nekonečného počtu kroků. Algoritmus také není korektním řešením daného problému,

jestliže existuje ***alespoň jeden*** vstup, pro který se algoritmus nikdy nezastaví nebo pro který se zastaví, ale nevydá správný výstup (protože buď nevydá žádný výstup nebo vydá nějaký chybný).

To, že nějaký daný konkrétní algoritmus řeší určitý problém, nemusí být v řadě případů vůbec zřejmé. Nepostačuje ani algoritmus implementovat a otestovat na několika (i velmi mnoha) vstupech. Samozřejmě, pokud se najde nějaký alespoň jeden vstup, pro který algoritmus nevrací očekávaný výsledek, tak je jasné, že tento algoritmus korektní není. To, že algoritmus funguje správně na testovacích vstupech, však nevylučuje, že může existovat nějaký jiný vstup, pro který algoritmus vrací chybný výsledek nebo vůbec neskončí.

Množina všech možných vstupů bývá téměř vždy nekonečná nebo alespoň velmi velká, takže nepřichází v úvahu možnost vyzkoušet algoritmus pro všechny možné vstupy. Typicky se tedy očekává, že pokud někdo navrhne nějaký algoritmus pro řešení určitého problému, tak kromě toho, že popíše, jak tento algoritmus pracuje, tak podá také ***důkaz korektnosti*** tohoto algoritmu. U těch nejprimitivnějších algoritmů, kde je očividné, že algoritmus dělá to co má, sice takový takový důkaz asi potřeba není, ale u jakéhokoliv trochu složitějšího algoritmu je nezbytný, abychom si mohli být alespoň trochu jisti, že algoritmus funguje správně a že neprodukuje nesmyslné výsledky.

Důkaz korektnosti algoritmu samozřejmě nezaručuje se stoprocentní jistotou, že algoritmus je skutečně korektní, protože v tomto důkaze může být chyba. Důkaz může být proveden na různých úrovních podrobnosti. Čím má být důkaz podrobnější, tím je pracnější ho vytvořit a zkontrolovat. Na druhou stranu u podrobnějšího důkazu je větší šance, že neobsahuje nějakou zásadní chybu (tj. chybu, která by se nedala snadno opravit, naopak různých triviálních lehce opravitelných chyb bude asi v podrobnějším důkazu více než v méně podrobném).

Pokud si vezmeme nějakou konkrétní implementaci daného algoritmu (tj. program zapsaný v nějakém programovacím jazyce, který se dá spustit na počítači), mohou být chyby také v této konkrétní implementaci, i když algoritmus jako takový je v pořádku. Důkaz korektnosti algoritmu tedy sám o sobě ještě nezaručuje, že určitý program bude fungovat správně, ale dává alespoň určitou minimální jistotu ohledně toho, že daný algoritmus není úplně špatně.

* * *

Pokud by počítače pracovaly nekonečně rychle, stačilo by pro daný problém vždy použít nějaký libovolný algoritmus, který by tento problém řešil. Pokud bychom měli k dispozici více algoritmů řešících tento problém, asi by bylo nejlepší použít nějaký co nejjednodušší algoritmus, u kterého bychom si mohli být více jisti ohledně jeho korektnosti. I když počítače nepracují nekonečně rychle, tak v případě, kdy předpokládáme, že příslušnou implementaci algoritmu budeme používat jen pro malá množství vstupních dat, dá se postupovat přesně tímto způsobem.

Pokud například budeme chtít třídit posloupnosti prvků, které budou mít maximálně 10 prvků, je úplně jedno, jestli pro toto třídění použijeme třídění přímým vkládáním, Bubblesort, Mergesort nebo nějaký jiný třídící algoritmus, za předpokladu, že je tento algoritmus korektní. Jiná situace však nastane, pokud budeme chtít třídit posloupnosti, které mají 1 000 000 nebo třeba 1 000 000 000 prvků. V takovém případě mohou být mezi různými algoritmy obrovské rozdíly v době jejich běhu a ve množství spotřebované paměti, kterou algoritmus ke svému běhu potřebuje. To, co jeden algoritmus počítá hodinu, může jiný spočítat ve zlomku vteřiny. Nějaký jiný neefektivní algoritmus by mohl stejnou věc naopak počítat třeba miliardu let (ovšem dávno předtím, než by takový výpočet skončil, by se počítač, na kterém by výpočet běžel, rozpadl na prach).

Konkrétní doba výpočtu je ovlivněna mnoha různými faktory jako je použitý hardware, taktovací frekvence procesoru, použitý programovací jazyk, použitý překladač nebo interpretr tohoto programovacího jazyka, nastavení voleb pro optimalizaci u tohoto překladače, atd. Jedním z nejdůležitějších faktorů je však použitý algoritmus. Volba efektivního nebo naopak nevhodného algoritmu může celkovou dobu výpočtu ovlivnit v mnohem větší míře než tyto ostatní faktory.

U algoritmů se tedy posuzuje nejen jejich korektnost (která by měla být považována za samozřejmost, pokud má být algoritmus vůbec použitelný), ale především také to, jak jsou efektivní, tj. jaké jsou jejich časové a paměťové nároky. Časová a paměťová náročnost algoritmů se typicky neposuzuje z hlediska nějakých absolutních jednotek (jako třeba doba běhu v sekundách, počet

bytů v paměti, apod.), protože ty jsou ovlivněny příliš mnoha jinými faktory než jen samotným algoritmem, a také proto, že tyto hodnoty závisí na konkrétním vstupu a pro různé vstupní hodnoty se mohou (a pravděpodobně budou) lišit.

Ukazuje se, že mnohem důležitější je to, jak doba výpočtu či množství použité paměti roste s tím, jak se zvětšuje velikost vstupních dat. Tato závislost doby běhu algoritmu na velikosti vstupních dat se označuje jako ***časová složitost*** algoritmu a závislost množství použité paměti na velikosti vstupních dat jako ***paměťová*** nebo též ***prostorová složitost*** algoritmu. Z formálního hlediska jsou časová a paměťová složitost funkce, které vyjadřují, jak roste doba výpočtu nebo množství použité paměti v závislosti na velikosti vstupu.

Časová ani paměťová složitost algoritmu se prakticky nikdy neurčuje zcela přesně, protože by to jednak bylo velice pracné (a u složitějších algoritmů téměř nemožné) a navíc by takový výsledek nebyl příliš užitečný, protože konkrétní hodnoty by závisely na nespočtu různých detailů a při jakékoliv drobné změně v implementaci algoritmu by bylo nutné provést analýzu znovu. Proto se složitost (ať už časová či paměťová) vyjadřuje vždy ve formě nějakého více či méně přesného ***odhadu***, přičemž tento odhad se vyjadřuje většinou ve formě zápisu pomocí tzv. ***asymptotické notace***. Příkladem použití asymptotické notace je například tvrzení, že Bubblesort má časovou složitost $O(n^2)$, zatímco Mergesort $O(n \log n)$. Například to, že Bubblesort má časovou složitost $O(n^2)$, znamená, že doba výpočtu tohoto algoritmu je zhruba úměrná druhé mocnině počtu prvků, které je třeba setřídit. (Pro jistotu upozorněme na to, že toto je poněkud nepřesné a zjednodušené vyjádření toho, co přesně časová složitost $O(n^2)$ znamená.)

Poznamenejme, že slovo odhad zde neznamená, že se něco hádá nebo odhaduje, ale to, že se příslušné funkce vyjadřující složitost neurčují přesně. Místo toho se určuje určitá třída funkcí, do které budou spadat všechny funkce vyjadřující skutečné složitosti v případě všech možných jednotlivých implementací příslušného algoritmu. V tomto smyslu jsou tedy odhady složitosti algoritmů zcela přesná a konkrétní matematická tvrzení, která se matematicky dokazují.

Analýzu složitosti algoritmu je možné provádět na různé úrovni podrobnosti. Hrubý odhad složitosti je často rutinní záležitostí, kterou lze provést relativně rychle a snadno projitím kódu algoritmu, spočítáním úrovní zanoření cyklů, určením toho, s kolika prvky se v určité instrukci maximálně pracuje, apod. U efektivních algoritmů je však často jejich skutečná složitost o dost menší než by se podle takové hrubé naivní analýzy mohlo zdát. Přesnější určení složitosti tak může být někdy značně netriviální a v některých případech může vést na komplikované matematické problémy.

Přímočaré jednoduché algoritmy většinou nemívají příliš dobrou složitost. Algoritmy s lepší (tj. menší) složitostí mohou být (někdy velmi výrazně) komplikovanější a náročnější na implementaci. U efektivnějších algoritmů také často není na první pohled zjevná jejich korektnost a tuto korektnost je nutné dokazovat a podrobněji zdůvodňovat.

Ukazuje se také, že snažit se o co nejmenší časovou složitost může znamenat větší paměťové nároky algoritmu (nejčastěji proto, že si algoritmus ukládá v paměti velké množství různých mezivýsledků, které opakovaně používá). Naopak je možné vytvořit algoritmy, které jsou velmi paměťově úsporné, ale za cenu delší doby výpočtu (algoritmus si v paměti ukládá jen nezbytné minimum dat a mnohé věci počítá opakovaně, aby si je nemusel pamatovat).

Většinou se určuje složitost ***v nejhorším případě***, tj. hledá se určité omezení shora na dobu výpočtu či množství použité paměti. Zde se zaručuje, že pro ***žádný*** vstup nebude doba výpočtu delší či množství spotřebované paměti větší než to, co bylo odvozeno. Pro mnoho vstupů může být však skutečná doba výpočtu menší, i když třeba existují nějaké vstupy, pro které výpočet může trvat tak dlouho, jak odpovídá odvozené časové složitosti v nejhorším případě. Z toho důvodu se někdy také analyzuje složitost ***v průměrném případě***, kdy se předpokládá nějaké pravděpodobností rozdělení na množině vstupů a počítá se asymptotický odhad střední hodnoty doby výpočtu. (Připomeňme, že pojem ***střední hodnota*** označuje hodnotu, ke které by se blížil aritmetický průměr jednotlivých dob výpočtu, kdybychom mnohokrát vybírali náhodná vstupní data podle příslušného pravděpodobnostního rozdělení.)

Jako příklad algoritmu, který má nižší časovou složitost v průměrném případě než v nej-

horším případě je možné uvést třeba Quicksort, který má časovou složitost v průměrném případě $O(n \log n)$ zatímco v nejhorším $O(n^2)$.

Jak se dá očekávat, analýza složitosti v průměrném případě je často výrazně komplikovanější a náročnější než analýza složitosti v nejhorším případě.

Náhoda a pravděpodobnost hrají při návrhu a analýze efektivních algoritmů roli nejen z hlediska vstupních dat, ale je také možné je využívat jako prostředku pro vytvoření efektivnějších algoritmů. Tzv. ***randomizované*** algoritmy využívají během výpočtu generátor náhodných čísel a vnášejí tak do výpočtu náhodnost, která tam původně nebyla. Pro stejná vstupní data a stejný algoritmus může výpočet proběhnou různě. V některých případech je tato náhodnost využita čistě k tomu, aby se zaručilo, že program s velkou pravděpodobností skončí svůj výpočet rychle. S určitou malou pravděpodobností může trvat výpočet déle, ale program vždy skončí a výsledek bude korektní.

Někdy se však používají i randomizované algoritmy, které nejsou korektní v tom smyslu, jak bylo popsáno výše. Tyto algoritmy vracejí s určitou pravděpodobností chybný výsledek, ale pravděpodobnost této chyby je omezena a s delší dobou výpočtu se limitně blíží nule. Pokud se program nechá běžet kratší dobu, bude pravděpodobnost chybného výsledku větší, když se nechá běžet déle, bude pravděpodobnost chyby velmi rychle klesat. Typickým příkladem takových algoritmů jsou algoritmy pro testování velkých prvočísel, které mají velký praktický význam v kryptografii (pro generování šifrovacích klíčů apod.).

* * *

Jak vyplývá z předchozího, jedním z důležitých témat, kterými se teoretická informatika zabývá, je návrh a analýza efektivních algoritmů pro řešení problémů z nejrůznějších oblastí. (Analýzou algoritmů je zde myšleno především dokazování korektnosti těchto algoritmů a analýza jejich výpočetní složitosti.)

Teoretická informatika se zde překrývá s řadou jiných oborů matematiky a informatiky. Namátkou uveďme alespoň některé z nich:

- ***Teorie grafů*** — tato oblast je bohatým zdrojem nejrůznějších algoritmických problémů. Grafy jsou v informatice všudypřítomné, navíc velké množství problémů z jiných oblastí se často dá přeformulovat jako problémy na grafech.

- ***Teorie čísel*** — jedná se o oblast matematiky zabývající se především výpočty na velmi velkých (např. tisícimístných) přirozených číslech. Typické problémy z této oblasti se často týkají prvočísel, dělitelnosti, řešení různých typů rovnic v modulární aritmetice (kde se počítá se zbytky po dělení) apod. Jedná se o poměrně náročnou disciplínu, řada nejobtížnějších matematických výsledků pochází právě z této oblasti (např. důkaz Velké Fermatovy věty), která se však může jevit jako poměrně vzdálená běžné realitě. Ve skutečnosti mají ovšem výsledky z této disciplíny velký praktický význam pro kryptografii.

- ***Kryptografie*** — velká většina šifer a dalších kryptografických mechanizmů je založena na tom, že pro určité typy problémů nejsou známy efektivní algoritmy (příkladem takového problému je třeba problém rozkladu velkých čísel na prvočísla či problémy týkající se nalezení řešení určitého typu rovnic v modulární aritmetice) a pro jiné naopak ano (např. nalezení největšího společného dělitele dvou čísel). Bez těchto mechanizmů by například nebyl možný elektronický podpis. (Poznamenejme, že analýza kryptografických algoritmů patří k těm nejnáročnějším a nejzrádnějším, protože tyto algoritmy musí být odolné nejen proti řešení hrubou silou, ale i například proti různým sofistikovaným randomizovaným algoritmům.)

- ***Výpočetní geometrie*** — je to oblast informatiky, zabývající se řešením geometrických úloh pomocí algoritmů. Tyto algoritmy samozřejmě nepracují s geometrickými objekty jako takovými, ale s jejich reprezentacemi pomocí čísel, která reprezentují například souřadnice bodů, jejich vzdálenosti apod. Algoritmy z této oblasti mají velké využití například v počítačové grafice.

- ***Vyhledávání v textu, komprese dat*** — existuje velké množství různých algoritmů, které provádějí různé operace s řetězci symbolů. Důležité jsou například algoritmy pro hledání zadaného řetězce v delším textu. Nemusí se hledat jen výskyt konkrétního řetězce, ale to co se hledá, může být zadáno pomocí složitějších kritérií. Algoritmy se také liší podle toho, jestli prohledávaný text může být nějak předzpracován. S touto problematikou úzce souvisí hledání efektivních algoritmů používaných ke kompresi dat. Další oblast, kde se algoritmy pracující s řetězci symbolů používají, je biologie, kdy se tyto algoritmy používají například při analýze DNA.

- ***Teorie her*** — jedná se o oblast aplikované matematiky zabývající se analýzou konfliktních situací, kdy se dva nebo více hráčů snaží pomocí určitých akcí dosáhnout nějakých navzájem protikladných cílů. Typické problémy z této oblasti se týkají například hledání optimálních strategií pro jednotlivé hráče, zjišťování, zda má některých z hráčů vítěznou strategii, která mu zaručí vítězství bez ohledu na to, jak hrají ostatní hráči, apod. Řada problémů z jiných oblastí matematiky a informatiky, které se na první pohled her vůbec netýkají, se dá zformulovat jako problémy týkající se určitých specifických her.

Samostatnou oblastí algoritmů je studium ***datových struktur***. Datové struktury se týkají toho, jakým způsobem mohou být v paměti počítače uloženy kolekce dat tak, aby se s nimi dalo co nejlépe pracovat. Typické datové struktury zahrnují například různé typy stromů (např. RB-stromy, AVL-stromy, B-stromy, trie, apod.), rozličné typy hashovacích tabulek, různé druhy spojových seznamů apod. Různé datové struktury se liší časovou složitostí jednotlivých operací, které lze s prvky provádět (přidání prvku, nalezení prvku, odebrání prvku, sekvenční procházení všech prvků apod.), a paměťovými nároky. Neexistuje nějaká jedna nejlepší datová struktura, která by byla vhodná pro všechny účely. Pokud je nějaká datová struktura efektivní pro provádění nějaké jedné operace, je to často vykoupeno vyšší časovou složitostí nějaké jiné operace, nebo vyššími paměťovými nároky nebo podstatně vyšší komplikovaností a náročností na implementaci.

Použité datové struktury mají často zásadní vliv na celkovou výpočetní složitost algoritmů, ve kterých jsou použity. Nejefektivnější známé algoritmy pro určité problémy (například to často platí u problémů z teorie grafů) jsou mnohdy založeny na použití různých rafinovaných datových struktur (různé speciální typy stromů apod.).

Nestudují se jen algoritmy prováděné sekvenčně na jediném procesoru. Důležitou oblastí je studium ***paralelních algoritmů*** a ***distribuovaných algoritmů***, u kterých se předpokládá běh na mnoha současně běžících procesorech. U těchto algoritmů je důležité, jakým způsobem spolu jednotlivé procesory komunikují (sdílená paměť, posílání zpráv), jak se synchronizují, atd. Paralelní algoritmy často vyžadují zcela odlišný přístup než algoritmy sekvenční a důkazy korektnosti takových algoritmů jsou také většinou komplikovanější než v případě sekvenčních algoritmů.

* * *

Při studiu algoritmů a algoritmických problémů je někdy výhodné se zaměřit na problémy v jejich co nejjednodušší podobě. Proto se často uvažují místo obecnějších problémů, které chceme v praxi řešit, tzv. ***rozhodovací problémy***, což jsou problémy, ve kterých je výstup omezen na odpověď typu Ano/Ne.

Vezměme si například problém (vrcholového) barvení grafu:

Vstup: Neorientovaný graf G.

Výstup: Přiřazení barev vrcholům grafu G tak, aby žádné dva sousední vrcholy nebyly obarveny stejnou barvou a byl použit nejmenší možný počet barev, kterým je vrcholy grafu takto možno obarvit.

Místo tohoto obecnějšího problému můžeme uvažovat následující rozhodovací problém:

Vstup: Neorientovaný graf G, číslo k.

Výstup: Ano — pokud je možné obarvit vrcholy grafu G pomocí k barev tak, aby žádné dva sousední vrcholy nebyly obarveny stejnou barvou; Ne — pokud to možné není.

Rozhodovací problémy bývají často specifikovány tak, že místo toho, aby bylo napsáno, co je výstupem, je tam uvedena otázka týkající se vstupu, na kterou se očekává odpověď Ano nebo Ne. To, že možným výstupem je Ano nebo Ne se tedy ve specifikaci rozhodovacího problému explicitně neuvádí a bere se to za něco, co je automaticky dané.

Výše uvedený problém se například dá formulovat následovně:

Vstup: Neorientovaný graf G, číslo k.

Otázka: Je možné obarvit vrcholy grafu G pomocí k barev tak, aby žádné dva sousední vrcholy nebyly obarveny stejnou barvou?

Jedním z důvodů, proč se místo obecnějších problémů často uvažují jejich rozhodovací varianty, je to, že tyto problémy i algoritmy pro jejich řešení jsou často o něco jednodušší z hlediska jejich formulace, popisu a analýzy. Pokud je nalezen (efektivní) algoritmus pro řešení daného rozhodovacího algoritmu, většinou se dá tento algoritmus relativně snadno rozšířit o to, aby produkoval odpovědi i na původní obecnější problém.

Rozhodovací problémy tedy v sobě často koncentrují to, co je možné považovat za „jádro“ původního obecnějšího problému, které když se vyřeší, tak vyprodukovat zbytek už je poměrně snadné.

* * *

Se studiem algoritmů a algoritmických problémů souvisí zkoumání obecnějších otázek, které v souvislosti s algoritmy vyvstávají. Jsou to otázky týkající se toho, jaké typy problémů se vůbec dají pomocí algoritmů řešit.

O problému se řekne, že je ***algoritmicky řešitelný***, jestliže existuje nějaký algoritmus, který tento problém řeší (tj. pro každý možný vstup se tento algoritmus po konečném počtu kroků zastaví a vydá správný výstup). U rozhodovacích problémů se většinou místo pojmu algoritmicky řešitelný používá pojem ***algoritmicky rozhodnutelný*** či jen ***rozhodnutelný***. Daný rozhodovací problém je tedy rozhodnutelný, jestliže existuje algoritmus řešící tento problém.

Naopak pokud algoritmus pro daný problém neexistuje, řekne se o takovém problému, že ***není algoritmicky řešitelný***. V případě rozhodovacího problému se pak o něm řekne, že je ***nerozhodnutelný***.

Zde je třeba zdůraznit, co se ve formulaci pojmů jako „algoritmicky řešitelný problém“, „rozhodnutelný problém“ nebo „nerozhodnutelný problém“ rozumí pojmy „existuje“ nebo „neexistuje“. Pokud se řekne, že „existuje algoritmus“, není tím myšleno, že takový algoritmus už někdo vymyslel, popsal a zdůvodnil o něm, že skutečně daný problém řeší. Slovu „existuje“ je zde třeba rozumět ve smyslu v jakém se toto slovo používá v matematice, například, když se třeba o nějaké rovnici o jedné neznámé řekne, že existuje (nebo naopak neexistuje) její kořen. Pro danou rovnici, buď existuje nějaké číslo, které, když se dosadí za neznámou, tak rovnice platí, nebo žádné takové číslo neexistuje. Existence nebo neexistence kořene dané rovnice je tedy objektivní fakt, který je zcela nezávislý na tom, zda daný kořen už někdo vypočítal.

Podobně, když se řekne, že existuje nějaký algoritmus, který něco splňuje (např. řeší určitý problém), myslí se tím existence takového algoritmu mezi všemi možnými algoritmy, které je možné potenciálně vytvořit, ne jen mezi těmi, které už někdo skutečně vymyslel a popsal. To, zda existuje algoritmus řešící určitý algoritmický problém, je tedy objektivní fakt nezávislý na aktuálních lidských znalostech. Samozřejmě, nejčastěji se o existenci algoritmu pro určitý problém přesvědčíme tím, že někdo takový algoritmus popíše a dokáže, že tento algoritmus skutečně řeší daný problém. Není to ovšem tak, že předtím, než tento algoritmus někdo vymyslel, nebyl daný problém algoritmicky řešitelný a pak se algoritmicky řešitelným stal (nebo byl nerozhodnutelný a

stal se rozhodnutelným). Takový problém byl algoritmicky řešitelný (resp. rozhodnutelný) vždy, akorát jsme to předtím nevěděli.

Pokud se tedy o nějakém problému řekne, že je nerozhodnutelný, je tím myšleno, že vůbec nelze vytvořit algoritmus, který by daný problém řešil.

Přirozená otázka je, zda vůbec jsou nějaké nerozhodnutelné problémy. Překvapivě se ukazuje, že ano. Existuje celá řada různých problémů, které se dají přesně (v matematických pojmech) definovat a popsat a které na první pohled mohou vypadat, že by pro ně nějaký algoritmus mohl existovat, ale pro které se dá dokázat, že žádný algoritmus, který by je řešil, neexistuje.

Mnoho takových problémů se týká chování programů. Asi není nic překvapivého na tom, že můžeme uvažovat o problémech, kde vstupem je kód nějakého programu. Kód programu je prostě posloupnost znaků (ne ovšem úplně libovolná, ale vyhovující nějakým pravidlům), se kterou se dá pracovat jako s jakoukoliv jinou posloupností znaků. Typickým příkladem programů, které zpracovávají kód jiných programů jako svůj vstup, jsou třeba překladače.

Spousta různých problémů týkajících se kódu programů se určitě algoritmicky řešit dá. Například se určitě dá algoritmicky zjistit, zda daný kód obsahuje nějakou nadeklarovanou proměnnou (což je jen jedna ze spousty věcí, kterou zjišťuje překladač při překladu zdrojového kódu). Čtenář jistě sám přijde se spoustou dalších příkladů toho, co se dá se zdrojovými kódy programu dělat a co se dá realizovat pomocí nějakého algoritmu.

Pomocí algoritmu však například nelze pro libovolný kód programu určit, zda se tento kód vždy po konečném počtu kroků zastaví, podobně nelze pro libovolný kód a určitý příkaz v tomto kódu algoritmem určit, zda existuje nějaký vstup, pro který se tento příkaz provede. Také například nelze pro libovolné dva kódy programů určit, zda se chovají stejně v tom smyslu, že pro stejné vstupy dávají vždy stejný výstup. Takových a podobných problémů, které se nedají řešit pomocí algoritmů, existuje velké množství.

Je zde ovšem potřeba mít pořád na paměti, co to znamená algoritmické řešení problému. Připomeňme, že algoritmus řeší problém, jestliže se pro každý vstup zastaví a vydá správný výstup. Pokud tedy nějaký problém nemá algoritmické řešení, znamená to, že pro každý algoritmus existuje alespoň jedna instance problému, pro kterou se tento konkrétní algoritmus nikdy nezastaví nebo pro kterou se zastaví, ale vydá chybný výstup.

Jednak tedy to, že nějaký problém není algoritmicky řešitelný, neznamená, že se pro žádný vstup tohoto problému nedá najít odpovídající výstup. I u problému, který není algoritmicky řešitelný, může existovat spousta instancí, pro které se dá nějakým algoritmem najít správný výstup. Existují však také nějaké instance, pro které tento algoritmus správný výstup nenajde.

To, že problém není algoritmicky řešitelný také neznamená, že existuje nějaká konkrétní instance tohoto problému, se kterou si žádný algoritmus neporadí. Ke každému algoritmu však existuje nějaká instance (ve skutečnosti nekonečně mnoho instancí), pro kterou se tento algoritmus nezastaví nebo pro kterou vrací chybný výstup. Tyto instance jsou však obecně pro různé algoritmy různé.

Pokud se tedy o nějakém problému ukáže, že není algoritmicky řešitelný, neznamená to ještě, že nemá smysl se snažit o jeho řešení pomocí algoritmů. Znamená to pouze, že nemůžeme očekávat algoritmus, který bude fungovat pro všechny vstupy.

Mimo jiné z výsledků týkajících se nerozhodnutelnosti některých problémů také vyplývá, že nelze zcela automatizovat proces dokazování korektnosti algoritmů, že to není něco, co bychom mohli zcela přenechat počítačům. Při návrhu a analýze algoritmů tedy hrají lidská inteligence a vhled do problému nezastupitelnou úlohu.

Poznamenejme, že oblast teoretické informatiky, která se zabývá tím, které problémy jsou a nejsou algoritmicky řešitelné, se nazývá ***teorie vyčíslitelnosti***.

* * *

To, že pro nějaký problém existuje algoritmus, který ho řeší, ještě z praktického hlediska příliš neznamená. V definici toho, že algoritmus řeší problém, se nic neříká o množství času a paměti, které má tento algoritmus k dispozici. Implicitně se předpokládá, že algoritmus má k dispozici

neomezené množství času a neomezené množství paměti, což ovšem v praxi nikdy neplatí. Z praktického hlediska je program, který se dopočítá výsledku pro určitý vstup za miliardu let, stejně neužitečný jako program, kde se výpočet pro tento vstup nikdy nezastaví.

Z praktického hlediska jsou tedy důležité ***efektivní algoritmy***, které skončí v nějakém „rozumném“ čase a bude jim stačit nějaké „rozumné“ množství paměti. Co je rozumný čas nebo rozumné množství paměti závisí na konkrétním typu úlohy, kde bude daný algoritmus použit, na typu hardwaru, na kterém program poběží a na velikosti vstupních dat, která se budou zpracovávat. Například při řízení nějakého stroje může být zpozdění 10 ms příliš velké. Naopak jiné typy výpočtů se mohou nechat běžet třeba několik dní. Při běžné interaktivní práci na počítači očekáváme výsledky maximálně v řádu sekund nebo nanejvýš minut (například, když spustíme kompilaci nějakého programu).

Pro řadu problémů existují velice efektivní algoritmy s velmi malou výpočetní složitostí. Například není problém na běžném PC nalézt během 1 sekundy nejkratší cestu mezi dvěma vrcholy v grafu, který má 100 000 vrcholů, přičemž nejdéle na celém výpočtu bude trvat načtení vstupu.

Oproti tomu ale existuje také mnoho problémů, které sice jsou algoritmicky řešitelné, ale u kterých se nedaří nelézt efektivní algoritmy. Příkladem takových problémů je třeba dříve uvedený problém barvení vrcholů grafu minimálním počtem barev nebo následující problém, známý pod názvem ***problém obchodního cestujícího*** (angl. ***travelling salesman problem***, často se také označuje zkratkou ***TSP***):

Vstup: Neorientovaný graf G, kde hrany jsou ohodnoceny přirozenými čísly.

Výstup: Nejkratší okružní cesta, která projde všemi vrcholy grafu a skončí ve stejném vrcholu, ve kterém začíná. (Délka cesty je součet ohodnocení hran na této cestě.)

Oba problémy se dají vyřešit ***hrubou silou*** (***brute force***), tj. systematickým zkoušením všech možných potenciálních řešení, kterých je jen konečný (i když velmi velký) počet. Například u problému barvení grafu je jasné, že stačí nanejvýš tolik barev, kolik je vrcholů, takže se dají zkoušet všechny možnosti, jak přiřadit jednotlivým vrcholům různé barvy z této množiny barev. Těchto možností je jen konečně mnoho a je možné je systematicky probrat a vybrat z nich tu, kde bude použito nejméně barev. Podobně u problému obchodního cestujícího má smysl zkoušet okružní cesty jen do určitého omezeného počtu hran. Takových cest je jen konečně mnoho, takže se opět dají probrat všechny a vybrat z nich tu nejkratší.

Takováto řešení hrubou silou se určitě dají použít třeba na grafy, které mají 10 vrcholů. Zde se výsledku dočkáme maximálně v řádu sekund. S narůstajícím počtem vrcholů však doba výpočtu takových algoritmů velmi prudce stoupá a pro 100 vrcholů pak výpočet může trvat třeba miliardy let.

Tím samozřejmě není řečeno, že takové řešení hrubou silou je tím nejlepším možným přístupem, a že se nic chytřejšího vymyslet nedá. Pro oba problémy existuje řada podstatně chytřejších algoritmů, které však mají buď tu nevýhodu, že nejsou korektní, protože nenajdou nutně vždy to nejoptimálnější řešení (nejmenší počet barev, nejkratší okružní cestu), nebo tu nevýhodu, že na některých instancích nejsou výrazně rychlejší než řešení hrubou silou (i když na některých jiných instancích skončí podstatně rychleji).

Ukazuje se, že mnohé algoritmické problémy jsou ze své podstaty výrazně obtížnější než jiné v tom smyslu, že každý algoritmus, který je řeší, vyžaduje mnohem větší množství času a paměti. U některých problémů se dokonce dá dokázat, že existuje nějaká určitá nejmenší časová nebo paměťová složitost, jakou musí mít každý algoritmus, který daný problém řeší.

U mnohých problémů se také ukazuje, že složitost algoritmů, které je řeší, je určitým způsobem svázaná se složitostí algoritmů řešících nějaký jiný problém. Tyto složitosti mohou být například provázány v tom smyslu, že dá ukázat, že složitost každého algoritmu, který řeší jeden problém, nemůže být nějak výrazně menší než složitost nejefektivnějších algoritmů, které řeší druhý problém.

Podle takových a podobných kritérií je možné klasifikovat problémy do různých tzv. ***tříd složitosti***. Pro problémy patřící do stejné třídy platí vždy podobná omezení na složitost algoritmů, které je řeší. Oblast teoretické informatiky, která se zabývá klasifikací problémů podle

jejich složitosti, vztahy mezi jednotlivými třídami a dalšími souvisejícími otázkami, se nazývá ***teorie složitosti***.

Zvláště důležitou třídou problémů jsou NP-úplné problémy. Výše uvedené problémy vrcholového barvení grafu nebo problém obchodního cestujícího jsou příklady NP-úplných problémů. V literatuře je však popsáno velké množství dalších NP-úplných problémů z nejrůznějších oblastí informatiky. Pro žádný problém z této třídy není znám efektivní algoritmus, ale na druhou stranu se ani zatím nikomu nepodařilo dokázat, že takový algoritmus nemůže existovat. NP-úplné problémy mají tu zajímavou vlastnost, že pokud by se podařilo najít efektivní algoritmus pro jeden jediný NP-úplný problém (je úplně jedno pro který), tak by tím okamžitě byly nalezeny efektivní algoritmy pro všechny ostatní NP-úplné problémy. Pokud by se naopak alespoň pro jeden NP-úplný problém podařilo dokázat, že se tento problém nedá žádným efektivním algoritmem řešit, znamenalo by to, že se nadá efektivně řešit ani žádný jiný NP-úplný problém.

* * *

U algoritmu se předpokládá, že je (nebo může být) vykonáván nějakým typem stroje. Nejčastěji je tímto strojem počítač nějakého typu. Tento počítač má procesor, paměť, nějaká periferní zařízení, pomocí kterých komunikuje s okolím. Různé druhy počítačů se mohou lišit v mnoha ohledech, jinak vypadá superpočítač pro vědecké výpočty, jinak počítač, který řídí automatickou pračku. Přesto však má většina počítačů celou řadu společných rysů.

Z hlediska algoritmů je důležité zejména to, s jakým typem dat se pracuje, jakým způsobem je organizována paměť a jaké typy instrukcí je možné provádět. Co se týká typu dat, se kterými počítače pracují, zde panuje téměř univerzální shoda v tom, že prakticky veškeré typy dat jsou na nízké úrovni reprezentovány jako sekvence bitů, tj. počítače v podstatě nedělají nic jiného, než že určitým způsobem transformují posloupnosti nul a jedniček na jiné posloupnosti nul a jedniček.

Doba výpočtu, počet kroků provedených algoritmem během výpočtu a do určité míry i množství použité paměti, jsou do značné míry ovlivněny vlastnostmi stroje, na kterém algoritmus běží. Z toho důvodu není výpočetní složitost algoritmu vlastností algoritmu jako takového, ale vztahuje se (často jen implicitně) k určitému typu stroje.

V teoretické informatice se z toho důvodu zavádějí různé ***výpočetní modely***, což jsou různé typy idealizovaných strojů, které mohou provádět algoritmy. Některé tyto výpočetní modely více či méně zhruba odpovídají tomu, jak vypadají skutečné počítače, jiné však mohou vypadat i velmi odlišně.

Oproti skutečným počítačům jsou tyto výpočetní modely často výrazně zjednodušené, je v nich ponecháno jen to, co je z hlediska algoritmů skutečně podstatné. U většiny výpočetních modelů se například ignoruje fakt, že skutečné počítače mívají jen konečné množství paměti, a pro jednoduchost se předpokládá, že paměť nemůže nikdy dojít.

Z hlediska programování je samozřejmě výhodné, když má programátor k dispozici různé konstrukce programovacího jazyka, kterými může vyjádřit jednotlivé kroky algoritmu jednodušším způsobem, které mu pomáhají program organizovat tak, aby se v kódu dalo vyznat, apod. Z hlediska výpočetní složitosti nebo z hlediska toho, jaké typy problémů se dají pomocí algoritmů řešit, je však většina těchto konstrukcí nepodstatných, protože se dají realizovat pomocí primitivnějších operací. Instrukce vykonávané procesorem jsou většinou velmi primitivní. Programátor tak píše v nějakém vyšším programovacím jazyce, ale tento kód se přeloží do instrukcí na mnohem nižší úrovni.

Většina výpočetních modelů pracuje jen s malým počtem typů instrukcí, které jsou schopny vykonávat. Tyto instrukce navíc bývají velmi jednoduché. To, co se na vyšší úrovni dá realizovat jedinou instrukcí, může vyžadovat velké množství jednodušších instrukcí na nižší úrovni.

Jednou z oblastí teoretické informatiky je zkoumání vztahů mezi různými výpočetními modely. Konkretně se například dá zkoumat, zda a jak je možné jeden model simulovat pomocí jiného modelu, jak při této simulaci narůstá počet provedených instrukcí apod. Ukazuje se, že veškeré algoritmy je možné realizovat i na velmi primitivních typech strojů, které používají instrukce, které jsou ještě mnohem jednodušší než instrukce používané skutečnými procesory.

Pro ilustraci uveďme jako příklad stroj, který má konečný počet čítačů, kde každý z těchto čítačů může obsahovat libovolně velké přirozené číslo. Jediné operace, které je možné s těmito čítači provádět, je zvýšit hodnotu čítače o jedna, snížit hodnotu čítače o jedna a otestovat, zda čítač obsahuje nulu. I když se to může zdát neuvěřitelné, těchto několik typů instrukcí postačuje na to, aby se pomocí nich dal implementovat libovolný algoritmus.

Různé výpočetní modely většinou neslouží k tomu, aby se pomocí nich skutečně algoritmy popisovaly, protože by to bylo velmi pracné a nepříliš užitečné. Výpočetní modely slouží především jako prostředek používaný při důkazech a i z tohoto hlediska je vhodné, aby byly co nejjednodušší.

Potřeba takových výpočetních modelů, které mají na jedné straně stejné vyjadřovací schopnosti jako libovolný programovací jazyk (tj. cokoliv, co se dá napsat v nějakém programovacím jazyce se dá realizovat v rámci daného modelu, i když třeba mnohem pracněji) a na druhé straně jsou velmi jednoduché, vyvstává zejména při dokazování toho, že něco nejde (že se daný problém nedá algoritmicky řešit nebo že každý algoritmus, který ho řeší, musí mít nějakou minimální výpočetní složitost).

Kromě modelů, které pracují sekvenčně existuje i velké množství různých modelů, které umožňují reprezentovat paralelní výpočty. Výpočetní modely také nemusí mít jen podobu strojů, které se podobají (byť velmi vzdáleně) klasickým počítačům. Existuje také mnoho modelů, kde stav výpočtu algoritmu je reprezentován určitým výrazem (pro jednoduchost si můžeme představit třeba aritmetický výraz) a provádění jednotlivých kroků algoritmu je realizováno jako přepisování tohoto výrazu podle určitých pravidel. Dalším typem výpočetních modelů jsou různý typy logických obvodů, kde je algoritmus realizován pomocí soustavy logických hradel propojených pomocí vodičů.

Velmi specifickým výpočetním modelem jsou ***kvantové počítače***.

Kromě výpočetních modelů, které jsou schopny realizovat libovolný algoritmus, se zkoumají také různé typy strojů, které mají typy instrukcí omezené natolik, že se pomocí nich nedá realizovat libovolný algoritmus, ale jen určitá specifická omezená třída algoritmů. Takovéto stroje se často označují jako ***automaty*** a oblast teoretické informatiky, která se zkoumáním těchto automatů zabývá, se nazývá ***teorie automatů***. Důvod, proč se různými omezenými typy strojů zabývat, je ten, že mnoho otázek, týkajících se algoritmů, které se v obecnosti řešit nedají, se pro určité typy automatů vyřešit dá.

* * *

V informatice se objevuje nepřeberné množství problémů, které se týkají práce s textem, tj. s posloupnostmi symbolů. Často nechceme pracovat s úplně libovolnými posloupnostmi symbolů, ale jen s takovými, které vyhovují určitým pravidlům. Zde vstupují na scénu otázky týkající se ***syntaxe***.

Typickým příkladem je třeba syntaxe programovacích jazyků, kde ne každá náhodná posloupnost znaků je dobře vytvořeným program v daném programovacím jazyce, nýbrž se programy musí řídit určitými přesně danými pravidly, která určují, co je a co není dobře vytvořený program, který půjde přeložit.

Nemusí se jednat jen o něco tak komplikovaného jako je programovací jazyk. Ve spoustě aplikací se objevuje potřeba nějakého specifického jazyka pro popis konfigurace nebo pro skriptování aplikace apod. Pokud program pracuje z nějakými textovými daty, která mají nějakou složitější strukturu (tj. není to třeba jen posloupnost čísel, ale jsou tam různé složitější konstrukce), může mít smysl řešit syntaxi těchto dat. Často textové údaje, které zadává uživatel, mohou mít nějakou složitější strukturu (mohou to být například nějaké výrazy, kde mohou být třeba závorky, různé operátory apod.) nebo musí vyhovovat nějakým složitějším kritériím.

Oblast informatiky, která se zabývá otázkami týkajícími se syntaxe, se nazývá ***teorie formálních jazyků***.

Pojem „jazyk“ označuje v informatice něco dost odlišného od toho, co se tímto pojmem označuje v přirozené řeči. V informatice, a speciálně v teorii formálních jazyků, pojem ***jazyk*** označuje libovolnou množinu ***slov***. Pojem „slovo“, zde má ovšem velmi odlišný význam od významu tohoto slova v přirozené řeči. Pojmem ***slovo*** se označuje zcela libovolná (konečná) posloupnost

symbolů (nebo též ***znaků***) z nějaké ***abecedy***, přičemž abecedou se zde myslí nějaká libovolná (většinou se předpokládá, že konečná) množina symbolů. Jako synonymum pojmu „slovo“ se též používá pojem ***řetězec*** (***string***).

Pokud si tedy vezmeme třeba abecedu tvořenou symboly 0 a 1, tak z nich je možné vytvořit třeba slovo 01011 nebo slovo 111. Jazyk je pak nějaká libovolná množina takovýchto slov, třeba množina všech slov, která začínají symbolem 0 a končí symbolem 1.

Takový jazyk asi moc zajímavý z praktického hlediska není. Místo toho ale třeba můžeme uvažovat o abecedě tvořené všemi znaky ASCII tabulky a jazyce tvořeném všemi dobře utvořenými aritmetickými výrazy nebo třeba o jazyce, kde slova tohoto jazyka jsou všechny dobře utvořené programy v C++.

Je asi jasné, že u takových složitějších jazyků vůbec nemusí být snadné přesně popsat, co přesně je a co není slovem daného jazyka. Popis takového jazyka v přirozené řeči může být nepřesný a nejednoznačný.

V teorii formálních jazyků se studují různé typy formálních prostředků pro popis jazyků, z nichž asi nejdůležitější jsou tzv. ***gramatiky***. Těchto gramatik existuje mnoho různých druhů, které se liší svými vyjadřovacími schopnostmi. Obecně se gramatikami myslí různé sady pravidel, která popisují, jak ***generovat*** všechna možná slova daného jazyka, tj. jazyk popsaný danou gramatikou je množina právě těch slov, která se dají pomocí dané gramatiky vygenerovat. V praxi jsou asi nejdůležitější ***bezkontextové gramatiky***, které mají velký význam při popisu syntaxe programovacích jazyků a při tvorbě překladačů pro tyto jazyky.

Dalším, v praxi často používaným, formalismem pro popis jazyků jsou ***regulární výrazy***. Na rozdíl od gramatik se regulární výrazy nepoužívají pro popis komplikovaných jazyků, ale spíše pro takové účely jako je vyhledávání určitého vzorku v textu nebo kontrola toho, že údaj zadaný uživatelem vyhovuje požadovanému formátu. Pomocí regulárních výrazů se třeba snadno dá zapsat něco takového jako vyhledat všechny řádky, které obsahují slovo `xxx`, za kterým následuje libovolný nenulový počet mezer, za kterými je slovo `yyy`, za kterým následuje jedna z číslic `5`, `6`, `7`.

Formální jazyky se také dají popisovat pomocí různých typů automatů. Automat zpracovává jako svůj vstup slovo z určité abecedy a jako svůj výstup vydá odpověď, zda toto slovo přijímá nebo nepřijímá. Jazyk popsaný daným automatem je pak množina právě těch slov, která automat přijímá. Říká se, že automat rozpoznává daný jazyk.

Ukazuje se, že mezi různými druhy gramatik, regulárními výrazy a různými druhy automatů existuje úzká souvislost. Určité gramatiky umožňují generovat právě ty jazyky, které se dají rozpoznávat určitým typem automatů, a naopak.

Toho se v praxi využívá tak, že existují softwarové nástroje, které dostanou jako vstup popis gramatiky určitého jazyka. Podle tohoto popisu pak vygenerují programový kód pro rozpoznávání slov daného jazyka, přičemž tento kód je de facto implementací činnosti určitého automatu.

Podobně knihovny, ve kterých je implementováno vyhledávání a nahrazování v textu podle regulárních výrazů, často pracují tak, že převádí daný regulární výraz interně na reprezentaci odpovídající příslušnému automatu, a při vyhledávání pak v podstatě simulují činnost tohoto automatu.

* * *

V rámci předmětu Úvod do teoretické informatiky budou prezentovány některé základní poznatky a pojmy zejména z následujících dvou oblastí:

- teorie formálních jazyků a automatů,
- algoritmy a výpočetní složitost.

Část I

Algoritmy

Kapitola 2

Algoritmy

2.1 Popis algoritmů pomocí pseudokódu

Při popisu algoritmů se často používá tzv. ***pseudokód***. Jedná se způsob zápisu, který je podobný zápisu programů v nějakém programovacím jazyce. Na rozdíl od programu zapsaného v programovacím jazyce však pseudokód není určen k tomu, aby byl spouštěn na počítači, ale k tomu, aby byl čten člověkem, který se danému algoritmu snaží porozumět. Pseudokód se tedy neřídí žádnou přesně danou syntaxí, není nijak přesně dáno, které operace je nebo není možné používat. Cílem je sdělit čtenáři informace, které jsou důležité z hlediska popisovaného algoritmu. V pseudokódu se často používá běžná matematická notace, někdy i přirozená řeč, pokud je popis nějaké operace jednodušší v přirozené řeči než v podobě kódu.

Často jsou v pseudokódu ignorovány detaily, které nejsou pro daný algoritmus důležité. Příkladem takových detailů je třeba ošetření chybových stavů (např. nedostatku paměti) nebo použité datové typy. Typicky se třeba v pseudokódu pracuje s proměnnými, do nichž je možno přiřadit jako hodnotu libovolně velké celé číslo a neřeší se, že ve skutečném programu napsaném v nějakém programovacím jazyce pak je pro tyto proměnné použit nějaký konkrétní datový typ (jako třeba `int` nebo `long`), kde jsou hodnoty, kterých může proměnná nabývat, omezeny na nějaký konečný interval celých čísel.

Oproti popisu v přirozené řeči má pseudokód výhodu v tom, že je v něm mnohem zřetelněji vidět celková struktura algoritmu, je tam jasně vidět, jak jsou do sebe zanořeny cykly, jak se větví podmínky, apod. Řádky pseudokódu bývají někdy číslovány, aby se na ně dalo snadněji odkazovat. Pseudokód nebývá téměř nikdy uveden jen sám o sobě, ale typicky je součástí nějakého delšího textu (např. knihy nebo článku), ve kterém je podán popis algoritmu normálně v přirozené řeči (s občasným použitím matematické notace), a pseudokód slouží jako pomocný prostředek pro jasnější, přesnější a přehlednější popis toho, co by se slovně popisovalo dost neohrabaně nebo nepřehledně.

* * *

Vezměme si jako příklad algoritmus, který pro libovolné zadané pole zjistí hodnotu největšího prvku v tomto poli. Pro daný algoritmus není podstatné, jakého přesně typu jsou prvky v tomto poli, jediné, co je důležité, je, že tyto prvky je možné vzájemně porovnávat. Pro libovolné dva prvky x a y tedy musí být možné určit, zda je x menší než y (tj. $x < y$), x větší než y (tj. $x > y$) nebo, zda se oba prvky rovnají (tj. $x = y$). Není ale podstatné, zda x a y jsou typu `int`, `double`, `string`, nebo nějakého úplně jiného typu.

(Pozn.: Pokud je určeno, kdy platí $x < y$, kdy $x > y$ a kdy $x = y$, tak je tím automaticky i určeno, kdy platí $x \leq y$ a kdy $x \geq y$.)

Pro konkrétnost budeme zadané pole označovat symbolem A. Dále budeme předpokládat, že pole A obsahuje n prvků a že tyto prvky jsou v poli A indexovány od nuly, tj. že pole A se skládá

z prvků

$$A[0],\, A[1],\, \ldots,\, A[n-1]\,.$$

Vstupem algoritmu je tedy pole A a číslo n udávající počet prvků v tomto poli. Výstupem je pak největší prvek v tomto poli, tj. taková hodnota $A[k]$, kde $0 \leq k < n$ a kde pro každé j takové, že $0 \leq j < n$, platí $A[j] \leq A[k]$.

Pro jednoduchost budeme navíc předpokládat, že pole A je neprázdné, tj. že $n \geq 1$, protože jinak bychom museli specifikovat, co má být výstupem v případě, kdy je pole A prázdné.

Algoritmus pro řešení tohoto problému je velmi jednoduchý, stačí projít prvky pole A jeden po druhém a průběžně si pamatovat, který ze zatím přečtených prvků je největší. (Pravděpodobně ani začínajícím programátorům by nemělo činit problém vymyslet tento jednoduchý postup.)

Tento algoritmus je možné popsat následujícím pseudokódem označeným jako Algoritmus 1.

Algoritmus 1: Algoritmus pro nalezení největšího prvku v poli

```
1  Find-Max (A, n):
2  begin
3      k := 0
4      for i := 1 to n − 1 do
5          if A[i] > A[k] then
6              k := i
7          end
8      end
9      return A[k]
10 end
```

Na řádku 1 je uveden název funkce, která implementuje daný algoritmus pro nalezení největšího prvku v poli. V tomto konkrétním případě má tato funkce název Find-Max. Za názvem funkce následuje v závorkách seznam jejích parametrů. Zde má funkce Find-Max dva parametry — pole A a délku tohoto pole n. Hodnoty těchto parametrů představují vstup daného algoritmu.

Na řádcích 2 až 10 pak následuje tělo funkce Find-Max. Toto tělo začíná klíčovým slovem **begin** na řádku 2 a končí klíčovým slovem **end** na řádku 10. V těle funkce se pak kromě parametrů A a n používají i další lokální proměnné. Zde konkrétně jsou to proměnné k a i. V pseudokódu se obvykle proměnné nijak nedeklarují ani se explicitně neuvádí, jakého jsou typu. V případě funkce Find-Max je z kontextu asi jasné, že proměnné k a i nabývají pouze celočíselných hodnot. (Při bližším prozkoumání je pak jasné, že obě proměnné navíc nabývají jen nezáporných celočíselných hodnot.)

Příkaz $k := 0$ na řádku 3 provede přiřazení hodnoty 0 do proměnné k. Symbol ":=" zde reprezentuje přiřazení. V mnoha programovacích jazycích se používá pro přiřazení symbol "=", v pseudokódu se však často používá pro přiřazení symbol ":=" nebo "←", protože "=" se používá pro rovnost (stejně jako v matematice).

Provedení příkazu přiřazení

$$x := E\,,$$

kde x je název proměnné a E nějaký libovolný výraz, probíhá tak, že se nejprve vyhodnotí výraz E (s aktuálními hodnotami všech proměnných) a poté se tato výsledná hodnota přiřadí do proměnné x. Výraz nalevo od ":= " (zde x) se označuje jako levá strana a výraz napravo od ":=" (zde E) se označuje jako pravá strana přiřazení. Výraz na levé straně může být i složitější než jen jednotlivá proměnná, může to být například označení prvku pole, např. $A[i]$, apod.

Ve funkci Find-Max se kromě přiřazení na řádku 3 nachází ještě jedno přiřazení a to na řádku 6, kde se do proměnné k přiřazuje hodnota proměnné i. Všimněte si, že tyto přiřazovací

příkazy nejsou ukončeny středníkem. V pseudokódu se středníky často neuvádí a to, kde jednotlivé příkazy začínají nebo končí, je dáno formátováním a odsazením těchto příkazů. V tomto textu budou středníky použity v pseudokódu jen výjimečně, například v případech, kdy bude na jednom řádku uvedeno více příkazů za sebou.

Na řádku 4 začíná cyklus **for**. V tomto textu bude používán v pseudokódech způsob zápisu cyklu **for** ve tvaru

$$\textbf{for } i := a \textbf{ to } b\,,$$

za kterým následuje tělo cyklu uzavřené mezi klíčová slova **do** a **end** (ve funkci FIND-MAX začíná tělo cyklu klíčovým slovem **do** na řádku 4 a končí klíčovým slovem **end** na řádku 8). Provedení příkazu **for** probíhá tak, že nejprve se vyhodnotí výrazy a a b a hodnota výrazu a se přiřadí do proměnné i, která představuje řídící proměnnou cyklu. Poté se otestuje, zda platí $i \leq b$. Pokud ne, provádění cyklu **for** končí a pokračuje se dalším příkazem za tělem cyklu. Pokud ano, provede se tělo cyklu. Po provedení těla cyklu se hodnota proměnné i zvětší o jedna (je zde skryto implicitní přiřazení $i := i + 1$) a znovu se vyhodnotí podmínka $i \leq b$. Pokud neplatí, cyklus končí, pokud platí, pokračuje se stejně jako předtím, ale s novou hodnotou i a to tak dlouho, dokud platí $i \leq b$.

Proměnná i tedy při jednotlivých průchodech cyklem nabývá postupně hodnot

$$a,\ a+1,\ a+2,\ \cdots,\ b-1,\ b$$

a po skončení cyklu má hodnotu $b+1$ v případě, kdy $a \leq b$ a kdy tedy tělo cyklu bylo provedeno alespoň jednou, nebo hodnotu a v případě, kdy $a > b$ a kdy tělo cyklu není provedeno ani jednou.

Poznamenejme ještě, že hodnoty výrazů a a b by se neměly při provádění těla cyklu měnit.

Na řádcích 5 až 7 je uveden příkaz **if**. Při provádění příkazu **if** se nejprve vyhodnotí podmínka, následující za klíčovým slovem **if** (v tomto případě $A[i] > A[k]$). Výsledkem vyhodnocení této podmínky je pravdivostní (booleovská) hodnota. Pokud je podmínka splněna (tj. vyhodnotí se jako TRUE), provedou se příkazy mezi klíčovými slovy **then** a **end**, v opačném případě (tj. jestliže se podmínka vyhodnotí jako FALSE, tyto příkazy se přeskočí.

Na řádku 9 výpočet algoritmu končí provedením příkazu **return**, který vrací nalezený maximální prvek jako výsledek.

* * *

Ve výše uvedeném příkladě Algoritmu 1 má tento algoritmus podobu funkce FIND-MAX. Pojmem ***funkce*** je zde myšlen podprogram, který může být součástí nějakého většího kódu, a který je možné z jiných míst v tomto větším kódu zavolat s tím, že při tomto volání jsou mu předány hodnoty vstupních dat jako argumenty. Hodnoty těchto argumentů jsou při volání přiřazeny do parametrů příslušného podprogramu (v případě funkce FIND-MAX do parametrů A a n).

V případě funkce FIND-MAX se tedy nijak neřešilo načítání vstupních dat a ani výpis výsledku. Z hlediska algoritmů jsou operace související se vstupem a výstupem většinou nepodstatné. V konkrétním programu, který má běžet na počítači, se samozřejmě musí tyto operace nějak implementovat, ale je to něco, co je svázáno s konkrétním prostředím (programovacím jazykem, použitými knihovnami, operačním systémem, konkrétním typem aplikace, apod.).

Dalším pojmem souvisejícím s pojmem „funkce“ je ***procedura***. Tímto pojmem se označuje podprogram, který se volá podobně jako funkce, ale nevrací svůj výsledek ve formě návratové hodnoty. Místo toho například zapíše nějaké hodnoty někam do paměti (přičemž místo, kam bude zapisovat, bude určeno buď některým z parametrů nebo třeba nějakou globální proměnnou) nebo je pošle na výstup. (V jazycích, jejíchž syntaxe vychází z jazyka C, jsou takové procedury často reprezentovány jako funkce s návratovou hodnotou typu `void`.)

Poznámka: Pojmy vstup a výstup se myslí nejen údaje zadávané uživatelem z klávesnice a výsledky vypisované na obrazovku, ale může se jednat prakticky o jakýkoliv druh komunikace, například čtení a zápis dat z a do souboru, přijímání a odesílání dat po síti, komunikaci přes nějaký druh portu, poslání dat zvukové kartě, čtení dat z mikrofonu, zápis dat do paměti na grafické kartě, apod.

V případě objektově orientovaných jazyků se místo pojmů funkce nebo procedura většinou používá termín ***metoda***. Metody se od funkcí a procedur liší tím, že metody jsou vždy svázány s nějakým ***objektem***, který je do nich automaticky předáván (většinou implicitně) jako jeden z parametrů. Často je tento imlicitní parametr označován slovy jako `this`, `self`, apod.

V tomto textu budou funkce, procedury, metody a další podobné konstrukce (např. konstruktory a destruktory) označovány souhrnně jako ***podprogramy***.

2.2 Řídící tok

Vezměme si nějaký libovolný algoritmus (například Algoritmus 1 z předchozí sekce) a pro jednoduchost předpokládejme, že celý tento algoritmus je realizován ve formě jediného podprogramu (např. jediné funkce), který sám nevolá žádné další podprogramy.

Instrukce v takovém podprogramu můžeme rozdělit zhruba do dvou skupin. V první skupině budou ty instrukce, které reálně provádějí nějakou činnost, jako například kopírování obsahu jedné proměnné do druhé (např. instrukce jako $k := i$), provedení nějaké aritmetické operace (např. instrukce $i := i + 1$), vyhodnocení toho, zda platí nebo neplatí nějaká podmínka (třeba vyhodnocení podmínky $A[i] > A[k]$), přečtení nějaké hodnoty ze vstupu nebo zápis nějaké hodnoty na výstup, apod.

Kromě těchto instrukcí jsou však v programu i příkazy jako třeba **if**, **while**, **for**, příkazy jsou nějak rozděleny do bloků, apod. Příkazy tohoto druhého typu samy o sobě neprovádějí žádné výpočty, ale nějakým způsobem určují, v jakém pořadí se budou provádět ostatní instrukce, kterým příkazem se v daném místě v kódu bude pokračovat, atd.

Struktura toho, v jakém pořadí jsou vykonávány jednotlivé příkazy v programu, jak na sebe jednotlivé instrukce navazují (jaká instrukce se bude provádět jako následující po provedení určité instrukce), jak se program větví podle vyhodnocení podmínek, apod., se označuje souhrnným názvem ***řídící tok***. Příkazy ze druhé skupiny uvedené výše (**if**, **while**, ...) jsou tedy ty příkazy, které určují řídící tok daného algoritmu.

Konkrétní konstrukce, které určují řídící tok algoritmu, se v různých programovacích jazycích mohou lišit. Ve většině programovacích jsou dispozici základní konstrukce pro strukturované programování jako **if**, **while** a možnost strukturovat příkazy do ***bloků***. Konkrétní syntaxe těchto příkazů se v různých jazycích liší, ale základní příkazy jsou podobné. Určité rozdíly jsou v tom, jaké další příkazy pro řízení toku jsou k dispozici (např. **break**, **continue**, **switch**), či v tom, zda je k dispozici příkaz **goto**, který umožní skok na libovolné místo v rámci podprogramu. Často je k dispozici možnost předčasného ukončení vykonávání podprogramu pomocí příkazu **return**. S řídícím tokem souvisí také například ošetření ***výjimek***.

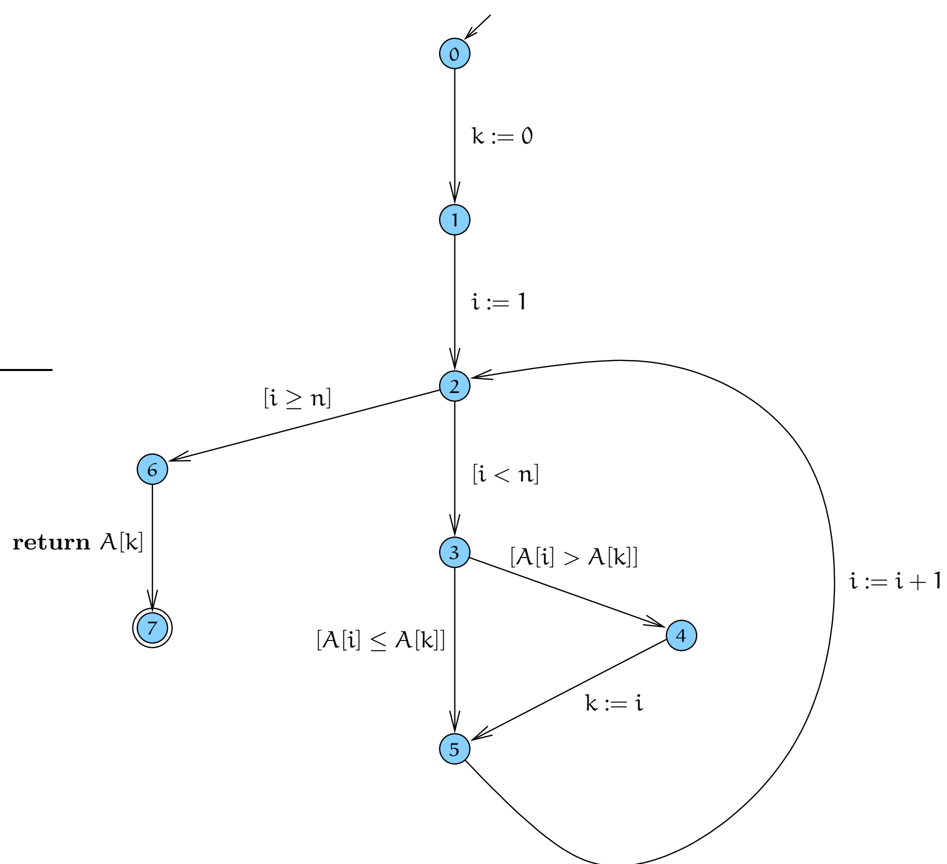
Výše popsané konstrukce se týkají řídícího toku v rámci jednoho podprogramu. Obecně se řídící tok týká i skoků ***mezi*** podprogramy (volání podprogramu, návrat z podprogramu). Pro jednoduchost se však v následujícím výkladu zaměříme především na řídící tok v rámci jednoho podprogramu.

* * *

Řekněme, že máme nějaký algoritmus realizovaný ve formě podprogramu a že tento podprogram nevolá žádné další podprogramy. Jednou možností, jak znázornit řídící tok tohoto algoritmu, je popsat ho pomocí ***grafu řídícího toku***. Jako konkrétní příklad si vezměme Algoritmus 1 (str. 16). Řídící tok tohoto algoritmu se dá znázornit grafem uvedeným na Obrázku 2.1.

Vrcholy tohoto grafu odpovídají pozicím v kódu ***mezi*** jednotlivými příkazy, resp. před a za jednotlivými příkazy. Například vrchol 1 odpovídá pozici mezi řádky 3 a 4, kdy už byl proveden příkaz $k := 0$ a kdy se teprve bude provádět příkaz **for** $i := 1$ **to** $n - 1$, jehož provádění začne tím, že se do proměnné i přiřadí hodnota 1 (tj. provede se přiřazení $i := 1$).

Vrcholy grafu reprezentují určité ***stavy***, ve kterých se algoritmus během výpočtu může nacházet. Těmto stavům budeme říkat ***řídící stavy***. Daný řídící stav vždy určuje, jaká instrukce se bude provádět jako následující.



Obrázek 2.1: Řídící tok funkce Find-Max

Algoritmus se začne provádět ve vrcholu 0, který označuje pozici na začátku těla funkce, tj. na řádku 2 (před instrukcí $k := 0$ na řádku 3). Tento vrchol představuje ***vstupní bod*** daného podprogramu, což je v grafu vyznačeno malou šipkou směřující do vrcholu 0.

Hrany grafu odpovídají jednotlivým instrukcím programu. Pokud vede hrana z vrcholu q do vrcholu q' a tato hrana je označena nějakou konkrétní instrukcí, znamená to, že při provádění algoritmu se přejde provedením této instrukce ze stavu reprezentovaného vrcholem q do stavu reprezentovaného vrcholem q'.

Pokud vede z vrcholu jediná hrana, je tato hrana většinou označena instrukcí, která provádí nějaké přiřazení, čtení hodnoty ze vstupu, zápis hodnoty na výstup, apod. Pokud se tedy algoritmus během výpočtu nachází v takovém stavu, kde z příslušného vrcholu vede jediná hrana, provede se instrukce, kterou je tato hrana označena, a přejde se do stavu, kam tato hrana směřuje.

Na Obrázku 2.1 se jedná například hranu z vrcholu 0 do vrcholu 1 označenou instrukcí $k := 0$. Tato hrana odpovídá instrukcí na řádku 3 Algoritmu 1. Dále pak třeba o hranu z vrcholu 1 do vrcholu 2 označenou instrukcí $i := 1$. Tato hrana odpovídá přiřazení hodnoty 1 na začátku provádění cyklu **for** na řádku 4. Podobně hrana z vrcholu 5 do vrcholu 2 označená přiřazením $i := i + 1$ reprezentuje zvýšení hodnoty proměnné i o jedna po každém provedení těla cyklu **for** na řádcích 4–8. Zde je vidět, že v grafu jsou explicitně uvedeny i některé instrukce, které jsou v kódu pouze implicitně, protože jsou tam skryty ve složitějších příkazech, jako je třeba příkaz **for**.

Někdy se pro přehlednost hodí přidat do grafu hrany, které nejsou označeny žádnou instrukcí. Pokud z nějakého vrcholu taková hrana vede, jedná se o jedinou hranu vedoucí z tohoto vrcholu. Taková hrana pak reprezentuje ***nepodmíněný skok***, kdy projitím touto hranou se pouze změní řídící stav, ale obsah proměnných se nemění ani se neprovádí žádná další akce.

Dále se v grafu kromě vrcholů, ze kterých vede jediná hrana, nachází vrcholy, ze kterých vedou dvě nebo více hran. Takové vrcholy představují ta místa v programu, kde dochází k ***větvení***. V případě vrcholu, ze kterého vede více než jedna hrana, nejsou tyto hrany označeny instrukcemi, které by prováděly nějakou činnost (přiřazení, vstup, výstup). Místo toho je každá taková hrana označena podmínkou, která musí být splněna, aby se danou hranou mohlo projít. Aby se tyto podmínky odlišily od ostatních instrukcí, jsou uváděny v hranatých závorkách.

Pokud výpočet algoritmu dojde do takového vrcholu, kde dochází k větvení, vyhodnotí se podmínky, kterými jsou označeny hrany vedoucí z tohoto vrcholu, a pokračuje se hranou, jejíž podmínka je splněna. Touto hranou se přejde do dalšího řídícího stavu. Předpokládá se, že vždy bude pravdivá právě jedna z podmínek, kterými jsou označeny hrany vedoucí z daného vrcholu.

Nejčastěji vycházejí z vrcholu dvě hrany, přičemž jedna je označena podmínkou B a druhá podmínkou $\neg B$ (na hranách grafu jsou zapsány jako $[B]$ a $[\neg B]$). Pro lepší čitelnost je v následujícím textu někdy místo zápisu $\neg B$ použita ekvivalentní podmínka bez negace. Pokud například podmínka B je podmínka $i < n$, bude jedna hrana označena zápisem $[i < n]$ a druhá zápisem $[\neg(i < n)]$ nebo případně zápisem $[i \geq n]$, neboť $\neg(i < n)$ platí právě tehdy, když $i \geq n$.

Předpokládá se, že výsledkem vyhodnocení každé takové podmínky je booleovská hodnota (tj. TRUE nebo FALSE) a že při jejím vyhodnocení nedochází k žádným vedlejším efektům (jako třeba změně hodnoty nějaké proměnné apod.).

Vrcholy, kde se dochází k větvení, a ze kterých vedou více než dvě hrany, se dají použít například pro konstrukce jako je příkaz **switch**.

Konečně mohou být v grafu také vrcholy, ze kterých nevede žádná hrana. Takové vrcholy odpovídají stavům, ve kterých výpočet algoritmu končí. Na Obrázku 2.1 je to vrchol 7. (Tento vrchol je označen dvojitým kroužkem, aby bylo zdůrazněno, že se jedná o stav, ve kterém výpočet končí.)

* * *

Na Obrázku 2.2 je naznačeno, jak pomocí grafu řídícího toku reprezentovat některé základní konstrukce, které se běžně objevují v programovacích jazycích. Jedná se o konstrukce ***strukturovaného programování***. V případě strukturovaného programování je možné vytvářet složitější

příkazy skládáním jednodušších příkazů. Na Obrázku 2.2 a v následujícím popisu zastupují symboly S, S_1 a S_2 nějaké libovolné příkazy (které mohou být samy složeny z nějakých jednodušších příkazů) a symbol B zastupuje nějakou libovolnou podmínku, výsledkem jejíhož vyhodnocení je booleovská hodnota.

Výhodou strukturovaného programování (alespoň v jeho nejjednodušší podobě) je to, že každý složený příkaz má jeden bod, kde se do něj vstupuje, a jeden bod, kde se z něj vystupuje. To umožňuje programátorovi udržovat lepší přehled o celkové struktuře řídícího toku.

Následuje stručný popis jednotlivých konstrukcí uvedených na Obrázku 2.2:

a) $S_1; S_2$ — Libovolné dva příkazy je možné provést sekvenčně po sobě. Nejprve se provede příkaz S_1 a po skončení provádění příkazu S_1 se provede příkaz S_2. Tímto způsobem je možné za sebou řadit libovolný počet příkazů.

b) **if** B **then** S_1 **else** S_2 — Nejprve se vyhodnotí podmínka B. Pokud tato podmínka platí (tj. pokud má hodnotu TRUE), provede se příkaz S_1 (a příkaz S_2 se neprovádí). V opačném případě, tj. když podmínka B neplatí (protože má hodnotu FALSE), provede se příkaz S_2 (a příkaz S_1 se neprovádí).

c) **if** B **then** S — Jedná se o variantu příkazu **if** popsaného v předchozím bodě, kde příkaz S_2 je prázdný. Pokud tedy podmínka B platí, provede se příkaz S, a jinak se neprovede nic.

d) **while** B **do** S — Pokud je podmínka B splněna, provede se příkaz S, a poté se znovu vyhodnotí podmínka B, pokud opět platí, znovu se provede příkaz S, a zase se znovu vyhodnotí podmínka B, atd. Takto se pokračuje až do doby, než se při vyhodnocení podmínky B zjistí, že neplatí. V tom okamžiku provádění cyklu **while** končí. Pokud tedy podmínka B není splněna hned na začátku (před prvním provedením příkazu S), příkaz S se nikdy neprovede.

e) **do** S **while** B — Tento příkaz pracuje velmi podobně jako cyklus **while** popsaný v předchozím bodě, s tím rozdílem, že na začátku se před prvním provedením příkazu S podmínka B netestuje a příkaz S se tak vždy alespoň jednou provede. Příkaz **do** S **while** B dělá v podstatě to samé, co

$$S;\ \textbf{while}\ B\ \textbf{do}\ S\,.$$

f) **for** $i := a$ **to** b **do** S — Proměnná i nabývá postupně hodnot $a, a+1, \ldots, b$, přičemž pro každou z těchto hodnot se jednou provede příkaz S. Pokud je $a > b$, příkaz S se neprovede ani jednou. Příkaz **for** $i := a$ **to** b **do** S tak dělá to samé, co následující sekvence příkazů:

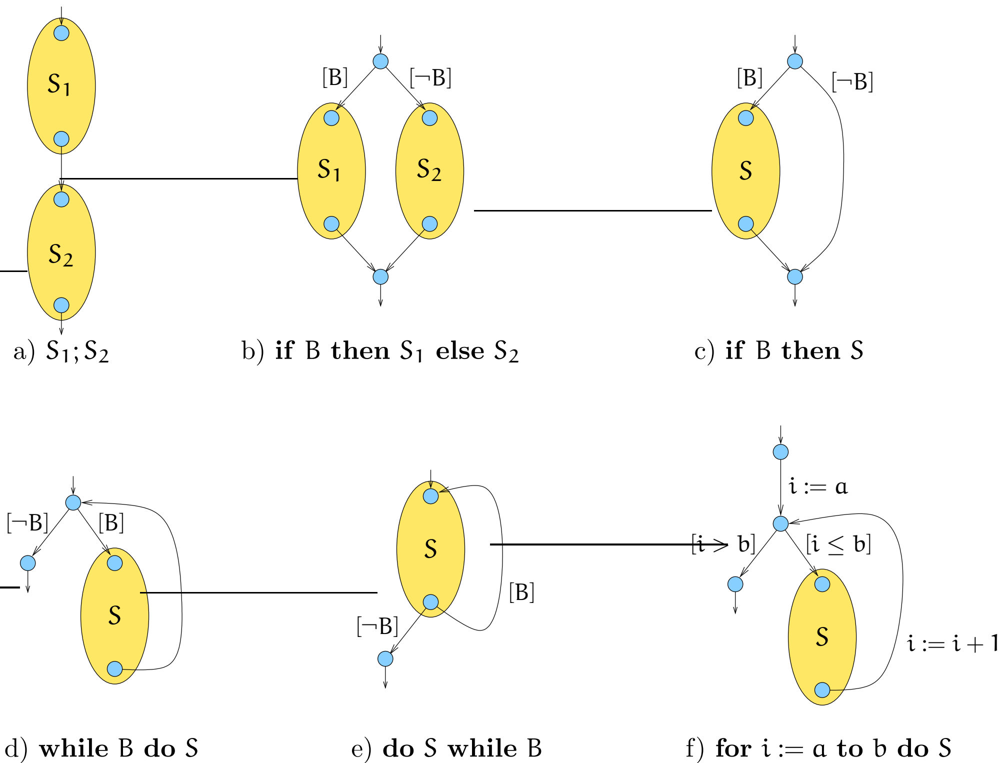
```
i := a
while i ≤ b do
    S
    i := i + 1
end
```

* * *

Podmínky mohou být jednoduché, spočívající například v porovnání hodnot proměnných, apod., jako třeba podmínka $i < n$ nebo podmínka $x \neq 0$. Takové jednoduché podmínky je možné skládat pomocí operátorů **and** a **or** a vytvářet tak z nich složené podmínky, jako třeba podmínku

$$i < n\ \textbf{and}\ x \neq 0\,,$$

která bude platit, jestliže hodnota proměnné i je menší než hodnota proměnné n a zároveň proměnná x neobsahuje nulu. Podobně složená podmínka vytvořená pomocí **or** bude platit, jestliže bude platit alespoň jedna z podmínek, ze kterých je vytvořena.



Obrázek 2.2: Grafy řídícího toku pro různé programové konstrukce

Při vyhodnocování takových složených podmínek se často uplatňuje zkrácený způsob vyhodnocování naznačený na Obrázku 2.3. Na tomto obrázku je zkrácené vyhodnocování ilustrováno na příkazu **if**. U jiných příkazů (**while**, **do** .. **while**, apod.) bude zkrácené vyhodnocení vypadat podobně.

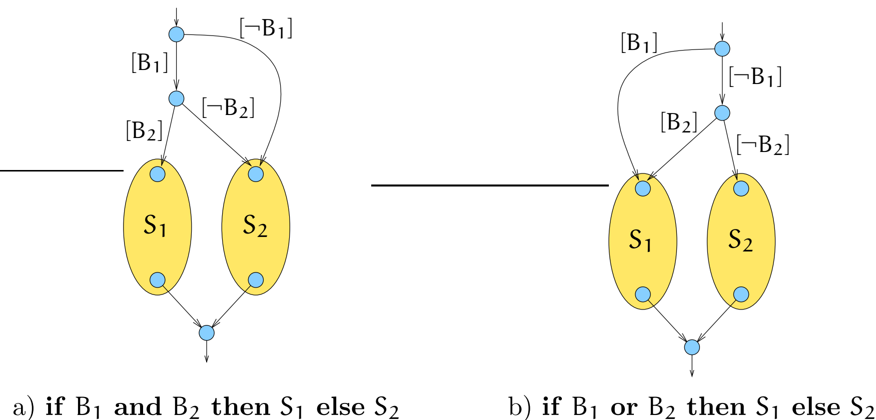
Konkrétně v případě složené podmínky tvaru B_1 **and** B_2 se při zkráceném vyhodnocování nejprve vyhodnotí podmínka B_1. Pokud podmínka B_1 neplatí (má hodnotu FALSE), podmínka B_2 se už nevyhodnocuje a hodnota celé podmínky B_1 **and** B_2 je FALSE. Pokud podmínka B_1 platí (má hodnotu TRUE), podmínka B_2 se vyhodnotí a hodnota celé podmínky je pak dána hodnotou podmínky B_2 (když má B_2 hodnotu TRUE, má celá podmínka hodnotu TRUE, když má B_2 hodnotu FALSE, má celá podmínka hodnotu FALSE). Celý postup vyhodnocení podmínky B_1 **and** B_2 je znázorněn grafem na Obrázku 2.3 (a).

Podobně při vyhodnocování složené podmínky tvaru B_1 **or** B_2 se postupuje tak, že nejprve se vyhodnotí B_1 a pokud platí, tak B_2 se už nevyhodnocuje a celkový výsledek je TRUE. Pokud B_1 neplatí, je celkový výsledek dán hodnotou B_2. Tento postup je znázorněn grafem na Obrázku 2.3 (b).

Operátory **and** a **or** tedy ovlivňují řídící tok. To, že se druhá část podmínky v některých případech nevyhodnocuje, může být důležité. Pokud například budeme mít pole A o n prvcích indexované od nuly (tj. s indexy 0, 1, ..., $n-1$), je možné například napsat

$$\textbf{while } i < n \textbf{ and } A[i] > x \textbf{ do} \ldots$$

Pokud se pak například má provádět tento příkaz v okamžiku, kdy $i \geq n$, vyhodnotí se jen první část podmínky (tj. $i < n$). Protože tato první část v daném okamžiku neplatí, druhá část ($A[i] > x$) se už vyhodnocovat nebude. Pokud by se druhá část vyhodnocovala vždy, bez ohledu na to, zda první část platí nebo ne, došlo by v případě, kdy $i \geq n$, k chybě, neboť by se při testu $A[i] > x$ přistupovalo mimo meze pole A.



a) **if** B_1 **and** B_2 **then** S_1 **else** S_2 b) **if** B_1 **or** B_2 **then** S_1 **else** S_2

Obrázek 2.3: Grafy řídícího toku pro zkrácené vyhodnocování složených podmínek

Je tedy třeba pamatovat na to, že ve složených podmínkách je pořadí jednotlivých dílčích podmínek důležité (tj. B_1 **and** B_2 se může chovat jinak než B_2 **and** B_1).

Pokud budou v dalším textu použity složené podmínky, bude se u nich vždy předpokládat zkrácené vyhodnocování. U grafu řídícího toku se pak bude předpokládat, že jsou tyto složené podmínky rozloženy na postupné vyhodnocení dílčích podmínek a že podmínky u hran grafu jsou jednoduché podmínky, které se vyhodnotí vždy celé.

* * *

Reprezentace algoritmů pomocí grafů řídícího toku se nám bude hodit pro některé účely, jako například pro popis některých postupů, které se používají při dokazování korektnosti algoritmů nebo při analýze jejich časové složitosti. Jednou z výhod grafu řídícího toku je to, že umožňuje abstrahovat od konkrétních řídících konstrukcí daného programovacího jazyka a podstatně snižuje počet různých podpřípadů, které je třeba rozebírat při popisu různých typů analýz algoritmů a programů.

Reprezentace pomocí grafu je také velmi blízká tomu, jak jsou programy většinou implementovány na úrovni strojového kódu nebo bytekódu nějakého virtuálního stroje. Na této nízké úrovni jsou programy realizovány jako posloupnosti jednoduchých instrukcí bez nějaké další struktury. Jednoduchými instrukcemi jsou zde myšleny například přiřazení apod., ale ne žádné složené příkazy. Tyto jednoduché instrukce jsou určitým způsobem číslovány, přičemž čísla instrukcí udávají jejich pozice v paměti. (Tyto pozice v paměti se často označují jako ***adresy*** instrukcí.)

Instrukce se vykonávají postupně v tom pořadí, jak jsou uvedeny v paměti. Při provedení „obyčejné“ instrukce (přiřazení) se vždy pokračuje prováděním instrukce, která je v paměti hned za ní na adrese následující za touto instrukcí. Veškerý řídící tok (v rámci jednoho podprogramu) je pak realizován pomocí následujících dvou typů instrukcí:

- **goto** ℓ — ***nepodmíněný skok*** — po provedení tohoto příkazu se nepokračuje následující instrukcí, ale instrukcí na adrese ℓ,

- **if** B **then goto** ℓ — ***podmíněný skok*** — při provedení tohoto příkazu se nejprve vyhodnotí podmínka B. Pokud platí, pokračuje se na adrese ℓ, pokud ne, pokračuje se následující instrukcí.

Algoritmus 1 realizovaný tímto způsobem vypadá takto:

```
0: k := 0
1: i := 1
2: goto 6
3: if A[i] ≤ A[k] then goto 5
4: k := i
5: i := i + 1
6: if i < n then goto 3
7: return A[k]
```

Čísla na začátku řádků zde udávají adresy instrukcí. Program se začne provádět provedením instrukce $k := 0$ na adrese 0. Poté se provede přiřazení $i := 1$ na adrese 1. Pak se pokračuje provedením instrukce **goto** 6 na adrese 2. Tato instrukce provede skok na adresu 6, kde se provede test, zda platí $i < n$. Pokud ano, pokračuje se instrukcí na adrese 3. Pokud ne, výpočet skončí provedením instrukce **return** $A[k]$ na adrese 7.

Místo adres instrukcí se pro lepší čitelnost používají ***návěští*** (angl. ***label***). Návěští je symbolické pojmenování určitého místa v kódu. Stačí uvádět návěští u těch instrukcí, které jsou cílem nějakého skoku. Adresy ostatních instrukcí nejsou podstatné.

Výše uvedený kód, kde jsou ale místo adres instrukcí použity návěští, vypadá následovně:

```
start: k := 0
       i := 1
       goto L3
   L1: if A[i] ≤ A[k] then goto L2
       k := i
   L2: i := i + 1
   L3: if i < n then goto L1
       return A[k]
```

Poznamenejme, že označování adres v kódu pomocí návěští se používá například při programování v assembleru (správný český, ale nepříliš používaný, termín je ***jazyk symbolických instrukcí***). Číslování řádků a realizaci řídícího toku pomocí příkazů **goto** používaly též různé historické verze programovacích jazyků jako Fortran a Basic.

Čtenáři by jistě nemělo dělat problém transformovat program v této podobě (nestrukturovaná sekvence instrukcí, kde je řídící tok realizován pomocí **goto**) do grafu řídícího toku. Rovněž je snadné realizovat algoritmus popsaný grafem řídícího toku jako takovouto posloupnost instrukcí. Stačí vypsat instrukce na hranách v nějakém libolvolném pořadí a doplnit mezi ně příkazy **goto**, aby se vždy pokračovalo tou správnou instrukcí podle grafu. (Samozřejmě není potřeba přidávat skok na následující řádek, protože následujícím řádkem se pokračuje automaticky. Rovněž je vhodné zvolit takové pořadí instrukcí, aby se tam těch příkazů **goto** muselo přidávat co nejméně.)

Je asi taky jasné, že každý program používající konstrukce strukturovaného programování se dá přeložit do programu, kde řídící tok realizován pomocí **goto**. Pro ilustraci uveďme způsob, jak přeložit strukturované příkazy **if** a **while**. (Další strukturované příkazy se dají přeložit podobným způsobem.) Zápisy $\langle S_1 \rangle$, $\langle S_2 \rangle$ a $\langle S \rangle$ zde zastupují sekvence instrukcí, na které by se přeložily příkazy S_1, S_2 a S.

a) **if** B **then** S_1 **else** S_2:

```
    if ¬B then goto L1
    ⟨S1⟩
    goto L2
L1: ⟨S2⟩
L2:
```

b) **while** B **do** S:

```
    goto L2
L1: ⟨S⟩
L2: if B then goto L1
```

* * *

Zatím jsme nijak neřešili, jak mohou vypadat výrazy v příkazech přiřazení nebo v podmínkách, které se testují. Obecně to ve většině vyšších programovacích jazyků mohou být libovolně složité výrazy, ve kterých se může objevit mnoho různých operátorů a dalších konstrukcí jako třeba přístup k prvku pole, přístup k položce záznamu, dereferencování ukazatele, apod.

Příklad takového komplikovanějšího přiřazení je třeba

$$A[i + s] := (B[3 * j + 1] + x) * y + 8, \qquad (*)$$

kde A a B jsou názvy polí a i, j, s, x, y číselné proměnné.

Pro některé účely se může hodit povolit jen instrukce přiřazení, ve kterých se může vyskytovat nejvýše jedna aritmetická operace, jejíž operandy musí být navíc proměnné nebo konstanty (tj. nemohou to být žádné komplikovanější výrazy). Podobné omezení můžeme dát i na další typy konstrukcí ve výrazech. Například pro přístup k prvkům pole můžeme povolit jen instrukce tvaru $A[i] := x$ a $x := A[i]$, kde i a x musí být proměnné nebo konstanty.

Bez ohledu na to, jaké konkrétní operace a konstrukce jsou ve výrazech povoleny, je asi zřejmé, že každý komplikovaný přiřazovací příkaz je možné transformovat (za cenu přidání nových pomocných proměnných) na sekvenci jednoduchých přiřazení splňující výše popsaná omezení. Například výše uvedené příkaz $(*)$ je možné nahradit následující posloupností jednoduchých přiřazení:

$t_1 := i + s$
$t_2 := 3 * j$
$t_2 := t_2 + 1$
$t_3 := B[t_2]$
$t_3 := t_3 + x$
$t_3 := t_3 * y$
$t_3 := t_3 + 8$
$A[t_1] := t_3$

V této sekvenci příkazů byly t_1, t_2 a t_3 přidány jako nové pomocné proměnné.

Podobným způsobem je možné transformavat libovolnou komplikovanou podmínku na posloupnost přiřazení, za kterou následuje jedna jednoduchá podmínka, kde se například porovnají hodnoty dvou proměnných apod.

Na úrovni strojového kódu nebo bytekódu nějakého virtuálního stroje bývá vyhodnocení komplikovaných výrazů realizováno tímto způsobem, takže jednotlivé instrukce pak mohou být velmi jednoduché.

2.3 Výpočet algoritmu

Řekněme, že máme nějaký algoritmus popsaný jako program v nějakém programovacím jazyce, nebo třeba pseudokódem, grafem řídícího toku nebo nějakým jiným způsobem. Konkrétní způsob popisu není v této chvíli podstatný.

Tento algoritmus očekává jako svůj vstup data nějakého určitého typu. Například Algoritmus 1 očekává jako svůj vstup pole hodnot (pro konkrétnost řekněme třeba pole celých čísel) indexované od nuly a přirozené číslo udávající délku tohoto pole. Navíc se předpokládá, že tato délka není nula.

Příkladem vstupních dat pro tento algoritmus je tedy například dvojice

$$([3, 8, 1, 3, 6], 5),$$

kde $[3, 8, 1, 3, 6]$ je pole hodnot a 5 je délka tohoto pole.

Algoritmus je typicky vykonáván nějakým ***strojem***. Slovo „stroj" je zde chápáno ve značně širokém smyslu. Může to být třeba skutečný počítač, který pomocí svého hardware vykonává

program zapsaný ve formě instrukcí strojového kódu, může to být nějaký virtuální stroj (jako je třeba JVM – Java Virtual Machine) vykonávající program ve formě bytekódu, může to být interpretr nějakého programovacího jazyka, který zpracovává programy přímo ve formě zdrojového kódu, apod. Stejně tak může být tímto strojem nějaký idealizovaný matematický model počítače.

Tento stroj může být jednoúčelový, kdy je schopen vykonávat jen jeden jediný konkrétní algoritmus, přičemž tento algoritmus je přímo natvrdo zabudován do struktury tohoto stroje, nebo může být obecnější, kdy je schopen vykonávat mnoho různých algoritmů, přičemž popis konkrétního algoritmu, který má vykonávat, dostane ve formě ***programu***.

Řekněme, že daný stroj (ať už jednoúčelový nebo obecnější, který daný algoritmus dostal ve formě programu) dostane vstupní data pro daný algoritmus a začne tento algoritmus pro tato vstupní data provádět. Začne tato vstupní data nějak zpracovávat, začne provádět určité operace podle instrukcí v popisu algoritmu, přičemž při provádění těchto operací případně generuje nějaký výstup a buď po nějakém konečném počtu kroků skončí nebo pokračuje v provádění těchto operací donekonečna. Posloupnost kroků, které stroj pro daný konkrétní vstup provádí, se označuje jako ***výpočet***. Konkrétní posloupnost operací, které stroj nad daným vstupem provádí, závisí na tomto vstupu. Různým vstupním datům obecně odpovídají různé výpočty. Výpočet může být buď ***konečný*** (provádění algoritmu po nějakém konečném počtu kroků skončí) nebo ***nekonečný*** (stroj donekonečna pokračuje v provádění operací).

Během výpočtu si stroj musí pamatovat, kterou instrukci právě provádí. Kromě toho má typicky k dispozici nějakou ***pracovní paměť***, kam si může ukládat hodnoty, se kterými pracuje, a později je může z této paměti číst. Jakou podobu má tato paměť a jakým způsobem jsou data v této paměti organizována, závisí na konkrétním typu stroje.

Pro jednoduchost a pro konkrétnost si například můžeme představit, že tato paměť má podobu sady ***proměnných*** (pojmenovaných nějakými názvy), do kterých je možno přiřazovat libovolné hodnoty určitých typů. (Pro jednoduchost teď ignorujme věci jako rozdělení proměnných na lokální a globální, možnost dynamické alokace paměti, apod.)

Předpokládá se, že pracovní paměť se chová zcela pasivně, že sama žádné operace neprovádí, a že je spolehlivá. Co do ní stroj zapíše, to z ní později také přečte, bez toho, že by se tato data mohla z paměti sama ztrácet, měnit nebo se tam nějak sama od sebe objevovat. Veškeré změny obsahu paměti jsou důsledkem provedení nějaké operace, kterou stroj vykonal.

Ve skutečném počítači je paměť, kterou program při výpočtu používá tvořena nejen pamětí RAM, ale tvoří ji i registry procesoru, paměti cache ležící mezi procesorem a hlavní pamětí, a někdy může program jako svou paměť využívat i pevný disk.

Z určitého pohledu se i na místo, ze kterého jsou čtena vstupní data, můžeme dívat jako na součást pracovní paměti stroje. Někdy jsou už vstupní data skutečně uložena na začátku výpočtu v paměti. Například v Algoritmu 1 se předpokládá, že pole A obsahující vstupní data už je v paměti uloženo. Někdy se data načítají odněkud „zvenku“ (z disku, ze sítě, z nějakého připojeného čidla apod.). Z hlediska algoritmů však tento rozdíl většinou není podstatný. Na načítaní dat „zvenku“ se můžeme dívat tím způsobem, že zdroj dat je také součást paměti a čtení dat je v podstatě to samé, co kopírování těchto dat z jednoho místa paměti do nějakého jiného místa, kde se s nimi dá dále pracovat.

Podobně se na výstup dat můžeme dívat jako zápis dat někam do paměti. Někdy se skutečně data zapisují jen do paměti, jako třeba v případě, kdy se funkce implementující daný algoritmus vrací výstup jako návratovou hodnotu pomocí příkazu **return**. Jindy se sice data ve skutečnosti posílají na nějakou výstupní periferii, ale z hlediska algoritmu se na to můžeme dívat tak, jako by se tato data zapisovala někam do paměti.

Co se týká vstupu a výstupu, budeme místa odkud se načítají vstupní data a kam se zapisují výstupní data chápat jako součást pracovní paměti stroje, který daný algoritmus vykonává. Z hlediska algoritmu je pouze důležité, zda může tyto části paměti používat pro nějaké další účely. Konkrétně, zda může zapisovat do oblasti paměti, kde má uložena vstupní data, a zda může číst obsah paměti kam ukládá výstupní data.

Ve většině případů je přirozené stanovit omezení, že část paměti, ve které jsou uložena vstupní data, je možné pouze číst (tj. není možné tam nic zapisovat), a do části paměti, kam se zapisují výstupní data, je možné pouze zapisovat (tj. data zapsaná na výstup nemůže algoritmus číst). Ostatní paměť může algoritmus používat libovolným způsobem (tj. může do ní zapisovat i z ní číst).

V některých případech je zase naopak přirozenější to brát tak, že paměť se vstupními a výstupními daty se nijak neliší od zbytku paměti. Například u problému třídění se většinou předpokládá, že data, která se mají třídit, jsou na začátku uložena v poli. Toto pole se používá během výpočtu, kdy algoritmus přemisťuje prvky v tomto poli a mění tak jeho obsah. Po skončení výpočtu jsou pak setříděné prvky uloženy v tomto poli. Stejná paměť (stejné pole) se tak používá jak pro vstupní data, tak pro uložení výsledku, a během během výpočtu se používá jako pomocná paměť, jejíž obsah se průběžně mění.

* * *

Předpokládejme nyní, že máme nějaký konkrétní algoritmus *Alg* a že tento algoritmus je zadaný ve formě grafu řídícího toku. Dále předpokládejme, že tento algoritmus bude vykonáván nějakým konkrétním strojem $\mathcal{M}$, kde navíc budeme předpokládat, že paměť používaná strojem $\mathcal{M}$ má podobu sady proměnných, kterým jsou přiřazeny hodnoty nějakých určitých typů, přičemž hodnoty přiřazené jednotlivým proměnným může stroj $\mathcal{M}$ během výpočtu měnit.

Pokud algoritmus *Alg* vykonávaný strojem $\mathcal{M}$ dostane nějaký konkrétní vstup w (předpokládáme, že w je nějaký vstup z množiny všech přípustných vstupů pro daný algoritmus), začnou se provádět jednotlivé kroky algoritmu. Při těchto krocích se průběžně mění jak aktuální řídící stav (který určuje, jaká instrukce se bude provádět v následujícím kroku), tak obsah paměti.

Představme si, že bychom v nějakém libovolném okamžiku výpočet stroje pozastavili a podívali se, v jakém řídícím stavu se stroj zrovna nachází a jaký je celkový obsah jeho paměti. Takový celkový stav stroje $\mathcal{M}$ v nějakém okamžiku výpočtu se označuje jako ***konfigurace*** stroje $\mathcal{M}$.

Obecně u různých typů strojů mohou mít konfigurace stroje různou podobu. Pokud ovšem předpokládáme, že daný algoritmus je dán ve formě grafu řídícího toku, může být konfigurace reprezentována jako dvojice

$$(q, \mathit{mem})$$

kde

- q — představuje aktuální řídící stav,
- *mem* — představuje aktuální obsah paměti stroje $\mathcal{M}$, tj. určuje jaké hodnoty jsou momentálně přiřazeny jednotlivým proměnným.

Pro konkrétnost si vezměme Algoritmus 1 popsaný grafem řídícího toku, který je uveden na Obrázku 2.4. Tento graf je téměř stejný jako graf na Obrázku 2.1, ovšem oproti grafu na Obrázku 2.1 zde bylo doplněno označení jednotlivých instrukcí symboly I_0, I_1, ..., I_8. Navíc zde byl příkaz **return** $A[k]$ změněn na $\mathit{result} := A[k]$, takže vrácení výsledné návratové hodnoty je zde reprezentováno jako přiřazení do speciální pomocné proměnné *result*.

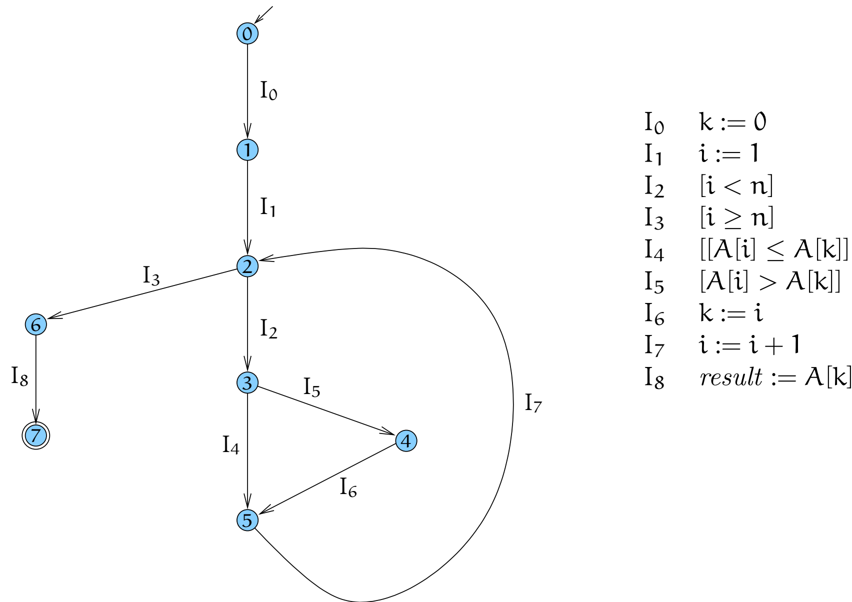
Pokud je například aktuální obsah paměti během výpočtu takový, že pole A obsahuje prvky $[3, 8, 1, 3, 6]$, proměnná n má hodnotu 5, proměnná i hodnotu 1, proměnná k hodnotu 0 a proměnná *result* nebyla dosud inicializována (tj. algoritmus do ní dosud nepřiřadil žádnou hodnotu), může být tento obsah paměti *mem* popsán následovně:

$$\langle A\colon [3, 8, 1, 3, 6],\quad n\colon 5,\quad i\colon 1,\quad k\colon 0,\quad \mathit{result}\colon ?\rangle$$

Otazník u proměnné *result* zde označuje, že tato proměnná nebyla dosud inicializována a že tedy obsahuje nějakou nedefinovanou náhodnou hodnotu, kterou by algoritmus neměl používat.

Pokud se algoritmus právě nachází v řídícím stavu 2 a jeho obsah paměti je výše popsané *mem*, aktuální konfigurace α je dvojice (q, mem), kde q je řídící stav 2, tj. celá konfigurace α vypadá takto:

$$(2,\ \langle A\colon [3, 8, 1, 3, 6],\quad n\colon 5,\quad i\colon 1,\quad k\colon 0,\quad \mathit{result}\colon ?\rangle)$$



Obrázek 2.4: Graf řídícího toku funkce Find-Max

Výpočet algoritmu začíná v ***počáteční konfiguraci***, kde řídící stav je počáteční řídící stav a kde hodnoty vstupních dat jsou uloženy v té části paměti, odkud se čtou vstupní data. Jinak je paměť prázdná, hodnoty proměnných jsou neinicializované, apod.

Pokud například u Algoritmu 1 budeme mít vstupní data w, kde w je dvojice $([3,8,1,3,6], 5)$, počáteční konfigurace bude

$$(0, \ \langle \mathsf{A}\text{: } [3,8,1,3,6], \ \ n\text{: } 5, \ \ i\text{: } ?, \ \ k\text{: } ?, \ \ \textit{result}\text{: } ?\rangle),$$

protože počáteční řídící stav je stav 0. V počáteční konfiguraci jsou tedy do pole pole A přiřazeny příslušné prvky a do proměnné n počet těchto prvků (tj. 5). Otazníky u proměnných i, k a *result* označují, že tyto proměnné jsou na začátku výpočtu neinicializované. Označme tuto počáteční konfiguraci jako α_0.

V grafu na Obrázku 2.4 vede ze stavu 0 jediná hrana. Tato hrana je označena instrukcí I_0, což je instrukce $k := 0$, a vede do stavu 1. V počáteční konfiguraci se tedy provede instrukce $k := 0$, která přiřadí do proměnné k hodnotu 0 a aktuální řídící stav se změní ze stavu 0 na stav 1. Provedením tohoto kroku se tedy z konfigurace α_0 přejde do konfigurace α_1, kde řídící stav je 1 a obsah paměti je stejný jako v konfiguraci α_0 s tím rozdílem, že do proměnné k je teď přiřazena hodnota 0.

Ze stavu 1 opět vede jediná hrana označená instrukcí I_1, což je instrukce $i := 1$. Tato hrana vede do stavu 2. V dalším kroku se tedy do proměnné i přiřadí hodnota 1 a přejde se do stavu 2, čímž se stroj $\mathcal{M}$ dostane do konfigurace α_2, kde řídící stav je stav 2 a obsah paměti je stejný jako v konfiguraci α_0 s tím rozdílem, že do proměnné i je přiřazena hodnota 1 a do proměnné k hodnota 0.

Z řídícího stavu 2 vedou dvě hrany označené podmínkami I_2 a I_3, kde I_2 je podmínka $[i < n]$ a I_3 podmínka $[i \geq n]$. Tyto podmínky se v konfiguraci α_2 vyhodnotí s aktuálními hodnotami proměnných. Protože v konfiguraci α_2 má proměnná i hodnotu 1 a proměnná n hodnotu 5, podmínka $i < n$ platí (protože $1 < 5$) a podmínka $i \geq n$ neplatí. Pokračuje se tedy hranou označenou instrukcí I_2. Tím se přejde z konfigurace α_2 do konfigurace α_3, kde řídící stav je stav 3 a obsah paměti je stejný jako v konfiguraci α_2. (Při vyhodnocování podmínek se obsah paměti nijak nemění.)

Podobným způsobem bychom mohli pokračovat dál. Při zpracování vstupu $([3,8,1,3,6], 5)$

provede Algoritmus 1 celkem 17 kroků, a projde přitom postupně konfiguracemi $\alpha_0, \alpha_1, \ldots, \alpha_{17}$, které vypadají následovně:

$$\begin{array}{rl}
\alpha_0\colon & (0,\ \langle A\colon [3,8,1,3,6],\ \ n\colon 5,\ \ i\colon ?,\ \ k\colon ?,\ \ \mathit{result}\colon ?\rangle) \\
\alpha_1\colon & (1,\ \langle A\colon [3,8,1,3,6],\ \ n\colon 5,\ \ i\colon ?,\ \ k\colon 0,\ \ \mathit{result}\colon ?\rangle) \\
\alpha_2\colon & (2,\ \langle A\colon [3,8,1,3,6],\ \ n\colon 5,\ \ i\colon 1,\ \ k\colon 0,\ \ \mathit{result}\colon ?\rangle) \\
\alpha_3\colon & (3,\ \langle A\colon [3,8,1,3,6],\ \ n\colon 5,\ \ i\colon 1,\ \ k\colon 0,\ \ \mathit{result}\colon ?\rangle) \\
\alpha_4\colon & (4,\ \langle A\colon [3,8,1,3,6],\ \ n\colon 5,\ \ i\colon 1,\ \ k\colon 0,\ \ \mathit{result}\colon ?\rangle) \\
\alpha_5\colon & (5,\ \langle A\colon [3,8,1,3,6],\ \ n\colon 5,\ \ i\colon 1,\ \ k\colon 1,\ \ \mathit{result}\colon ?\rangle) \\
\alpha_6\colon & (2,\ \langle A\colon [3,8,1,3,6],\ \ n\colon 5,\ \ i\colon 2,\ \ k\colon 1,\ \ \mathit{result}\colon ?\rangle) \\
\alpha_7\colon & (3,\ \langle A\colon [3,8,1,3,6],\ \ n\colon 5,\ \ i\colon 2,\ \ k\colon 1,\ \ \mathit{result}\colon ?\rangle) \\
\alpha_8\colon & (5,\ \langle A\colon [3,8,1,3,6],\ \ n\colon 5,\ \ i\colon 2,\ \ k\colon 1,\ \ \mathit{result}\colon ?\rangle) \\
\alpha_9\colon & (2,\ \langle A\colon [3,8,1,3,6],\ \ n\colon 5,\ \ i\colon 3,\ \ k\colon 1,\ \ \mathit{result}\colon ?\rangle) \\
\alpha_{10}\colon & (3,\ \langle A\colon [3,8,1,3,6],\ \ n\colon 5,\ \ i\colon 3,\ \ k\colon 1,\ \ \mathit{result}\colon ?\rangle) \\
\alpha_{11}\colon & (5,\ \langle A\colon [3,8,1,3,6],\ \ n\colon 5,\ \ i\colon 3,\ \ k\colon 1,\ \ \mathit{result}\colon ?\rangle) \\
\alpha_{12}\colon & (2,\ \langle A\colon [3,8,1,3,6],\ \ n\colon 5,\ \ i\colon 4,\ \ k\colon 1,\ \ \mathit{result}\colon ?\rangle) \\
\alpha_{13}\colon & (3,\ \langle A\colon [3,8,1,3,6],\ \ n\colon 5,\ \ i\colon 4,\ \ k\colon 1,\ \ \mathit{result}\colon ?\rangle) \\
\alpha_{14}\colon & (5,\ \langle A\colon [3,8,1,3,6],\ \ n\colon 5,\ \ i\colon 4,\ \ k\colon 1,\ \ \mathit{result}\colon ?\rangle) \\
\alpha_{15}\colon & (2,\ \langle A\colon [3,8,1,3,6],\ \ n\colon 5,\ \ i\colon 5,\ \ k\colon 1,\ \ \mathit{result}\colon ?\rangle) \\
\alpha_{16}\colon & (6,\ \langle A\colon [3,8,1,3,6],\ \ n\colon 5,\ \ i\colon 5,\ \ k\colon 1,\ \ \mathit{result}\colon ?\rangle) \\
\alpha_{17}\colon & (7,\ \langle A\colon [3,8,1,3,6],\ \ n\colon 5,\ \ i\colon 5,\ \ k\colon 1,\ \ \mathit{result}\colon 8\rangle)
\end{array}$$

V konfiguraci α_{17} výpočet končí, protože v této konfiguraci je řídící stav 7 a v grafu na Obrázku 2.4 nevede z tohoto řídícího stavu žádná hrana, takže se jedná o ***koncový stav*** (na obrázku je to znázorněno tím, že stav 7 je označen dvojitým kroužkem). Protože je v konfiguraci α_{17} řídící stav koncový, je konfigurace α_{17} ***koncovou konfigurací***.

Koncové konfigurace jsou ty konfigurace, ve kterých výpočet (korektně) končí. Kromě korektního ukončení v koncové konfiguraci může výpočet skončit také chybou, když během provádění algoritmu dojde k provedení nějaké nepovolené operace, například k přístupu k prvku pole mimo povolený rozsah indexů, k dělení nulou, apod.

Pokud α a α' jsou nějaké dvě konfigurace a I je nějaká instrukce, zápisem

$$\alpha \xrightarrow{I} \alpha'$$

budeme označovat, že provedením instrukce I přejde stroj z konfigurace α jedním krokem do konfigurace α'. Pokud je $\alpha = (q, \mathit{mem})$ a $\alpha' = (q', \mathit{mem}')$, musí vést v grafu řídícího toku hrana ze stavu q do stavu q' a tato hrana musí být označena instrukcí I.

Pokud $\alpha \xrightarrow{I_1} \alpha'$ a $\alpha' \xrightarrow{I_2} \alpha''$, můžeme to zkráceně zapsat jako

$$\alpha \xrightarrow{I_1} \alpha' \xrightarrow{I_2} \alpha''.$$

Podobně můžeme pokračovat, pokud by kroků výpočtu bylo víc.

Například prvních několik kroků výše popsaného výpočtu Algoritmu 1 nad vstupem $([3,8,1,3,6],\ 5)$ je tak možné zapsat následujícím způsobem:

$$\alpha_0 \xrightarrow{I_0} \alpha_1 \xrightarrow{I_1} \alpha_2 \xrightarrow{I_2} \alpha_3 \xrightarrow{I_5} \alpha_4 \xrightarrow{I_6} \alpha_5 \xrightarrow{I_7} \alpha_6 \xrightarrow{I_2} \alpha_7 \xrightarrow{I_4} \alpha_8 \xrightarrow{I_7} \alpha_9 \xrightarrow{I_2} \alpha_{10} \xrightarrow{I_4} \cdots$$

Obecně může být ***výpočet*** algoritmu *Alg* nad vstupem w konečný nebo nekonečný. ***Konečný výpočet*** je (konečná) posloupnost

$$\alpha_0 \xrightarrow{I_0} \alpha_1 \xrightarrow{I_1} \alpha_2 \xrightarrow{I_2} \alpha_3 \xrightarrow{I_3} \alpha_4 \xrightarrow{I_4} \ \cdots\ \xrightarrow{I_{t-2}} \alpha_{t-1} \xrightarrow{I_{t-1}} \alpha_t\,,$$

kde α_i pro $i = 0, 1, \ldots, t$ jsou konfigurace a I_i pro $i = 0, 1, \ldots, t-1$ jsou instrukce takové, že pro každé i, kde $0 \le i < t$, platí $\alpha_i \xrightarrow{I_i} \alpha_{i+1}$. Navíc musí platit, že α_0 je počáteční konfigurace pro vstup w a α_t je koncová konfigurace.

Nekonečný výpočet je definován podobně, akorát, že v tomto případě se jedná o nekonečnou posloupnost

$$\alpha_0 \xrightarrow{I_0} \alpha_1 \xrightarrow{I_1} \alpha_2 \xrightarrow{I_2} \alpha_3 \xrightarrow{I_3} \alpha_4 \xrightarrow{I_4} \cdots ,$$

kde výše uvedené vztahy musí platit pro všechna $i \in \mathbb{N}$. Konfigurace α_0 opět musí být počáteční konfigurace pro vstup w, ovšem na rozdíl od konečného výpočtu v nekonečném výpočtu neexistuje žádná koncová konfigurace.

Výpočet (ať už konečný nebo nekonečný) je tedy možné popsat dvěma různými způsoby:

(a) jako posloupnost konfigurací $\alpha_0, \alpha_1, \alpha_2, \ldots$,
(b) jako posloupnost provedených instrukcí $I_0, I_1, I_2, \ldots$

V každé konfiguraci α, která není koncová, je touto konfigurací jednoznačně určena instrukce I, která se v této konfiguraci provede. Na druhou stranu, pokud máme konfiguraci α a instrukci I, která se může v této konfiguraci provést, je touto konfigurací a instrukcí jednoznačně určena konfigurace α' taková, že

$$\alpha \xrightarrow{I} \alpha' ,$$

do které se provedením instrukce I v konfiguraci α přejde.

Připomeňme ještě, že v následujícím textu budeme rozlišovat instrukce následujících dvou typů:

- ***přiřazení*** — instrukce tvaru $x := E$.

  Řekněme, že máme konfiguraci $\alpha = (q, mem)$, kde v grafu řídícího toku vede ze stavu q jediná hrana označená instrukcí I tvaru $x := E$ a tato hrana vede do stavu q'. Nejprve se vyhodnotí výraz E při obsahu paměti *mem*. Hodnota tohoto výrazu se poté přiřadí do proměnné x a přejde se do stavu q'. Instrukcí I se tedy přejde z konfigurace α do konfigurace $\alpha' = (q', mem')$, kde obsah paměti mem' je stejný jako obsah paměti *mem* s tím rozdílem, že v mem' je do proměnné x přiřazena hodnota daná hodnotou výrazu E při obsahu paměti *mem*.

  Poznamenejme ještě, že i instrukce pro čtení vstupu a pro zápis na výstup budou chápány jako speciální případy přiřazení, jak bylo popsáno dříve.

- ***test podmínky*** — instrukce tvaru $[B]$.

  Řekněme, že máme konfiguraci $\alpha = (q, mem)$, kde v grafu řídícího toku vedou ze stavu q hrany označená podmínkami $[B_1]$, $[B_2]$, ..., $[B_s]$. Tyto podmínky se vyhodnotí při obsahu paměti *mem*. Podmínky musí být voleny tak, aby vždy byla pravdivá právě jedna z těchto podmínek. Pokud při obsahu paměti *mem* platí podmínka B_i (a ostatní podmínky při tomto obsahu paměti neplatí) a hrana označená touto podmínkou vede ze stavu q do stavu q', pokračuje se ve výpočtu konfigurací (q', mem). Při vyhodnocení podmínky se tedy obsah paměti nemění a provedením instrukce se pouze změní řídící stav.

V grafu řídícího toku na Obrázku 2.4 jsou instrukce I_0, I_1, I_6, I_7 a I_8 instrukcemi přiřazení a instrukce I_2, I_3, I_4 a I_5 jsou podmínky. V řídících stavech 0, 1, 4, 5 a 6 se jako následující instrukce provádí přiřazení, zatímco ve stavech 2 a 3 se vyhodnocují podmínky. Stav 7 je koncový.

Kapitola 3

Korektnost algoritmů

Zhruba řečeno, algoritmus je korektní, jestliže se chová „správně“ a dává takové výsledky, jaké od něj očekáváme. Korektnost algoritmu není vlastnost tohoto algoritmu jako takového, ale vždy se vztahuje k nějaké specifikaci toho, co by daný algoritmus měl dělat.

Nejčastěji má algoritmus za úkol zpracovat nějaký vstup z nějaké dané množiny možných vstupů a pro tento vstup vyprodukovat nějaký výstup, který vzhledem ke vstupu cosi splňuje. V takovém případě bývá to, co má algoritmus dělat, specifikováno ve formě ***algoritmického problému***.

Popis algoritmického problému musí obsahovat následující tři věci:

- ***Popis vstupu*** — Musí být specifikováno, jaký druh dat má algoritmus očekávat jako svůj vstup.
- ***Popis výstupu*** — Musí být specifikováno, jaký druh dat bude algoritmus generovat jako svůj výstup.
- ***Vztah mezi vstupy a výstupy*** — Musí být specifikováno, co musí data, která zapisuje algoritmus na výstup, splňovat vzhledem k datům na vstupu.

Například Algoritmus 1, popsaný v Sekci 2.1, by měl řešit problém nalezení největšího prvku v poli. Nazvěme tento problém názvem MAX-ELEMENT. Problém MAX-ELEMENT je možné podrobněji specifikovat následujícím způsobem:

VSTUP: Pole A indexované od nuly a číslo n udávající počet prvků v tomto poli, přičemž se předpokládá, že $n \geq 1$.

VÝSTUP: Hodnota *result*, která je hodnotou maximálního prvku v poli A, tj. hodnota *result*, pro kterou platí:

- $A[j] \leq \mathit{result}$ pro všechna $j \in \mathbb{N}$, kde $0 \leq j < n$, a
- existuje $j \in \mathbb{N}$ takové, že $0 \leq j < n$ a $A[j] = \mathit{result}$.

Připomeňme, že konkrétní vstupní data, která odpovídají specifikaci vstupu v popisu algoritmického problému se označují jako ***instance*** daného problému.

Například dvojice $([3, 8, 1, 3, 6], 5)$ je tedy instancí problému MAX-ELEMENT.

Definice 3.1 Algoritmus *Alg* ***řeší*** problém P, jestliže pro každou instanci w problému P jsou splněny následující dvě podmínky:

(a) Výpočet algoritmu *Alg* nad vstupem w se po konečném počtu kroků (korektně) zastaví.

(b) Algoritmus *Alg* vygeneruje pro vstup w výstup, který odpovídá podmínkám kladeným na výstup ve specifikaci problému P.

Algoritmus, který řeší problém P, je korektním řešením tohoto problému.

Důvodů, proč nějaký algoritmus *Alg* není korektním řešením problému P, může být několik. Stačí, aby existoval alespoň jeden vstup w, pro který nastane jedna z následujících situací:

(a) Během výpočtu algoritmu *Alg* nad vstupem w dojde k chybě, kdy algoritmus provede nějakou chybnou nepovolenou operaci (např. přístup k prvku pole mimo povolený rozsah indexů, k dělení nulou, apod.).
(b) Algoritmus korektně skončí, ale vygeneruje nějaký výstup, který nesplňuje podmínky specifikované v zadání problému P.
(c) Algoritmus se nikdy nezastaví.

Pokud se má zdůvodnit, že algoritmus je korektním řešením daného problému, musí se prokázat, že pro žádný vstup z množiny všech přípustných vstupů nedojde k žádnému z těchto nežádoucích případů.

Množina všech přípustných vstupů je většinou nekonečná nebo alespoň velmi velká. Například, kdybychom se u problému Max-Element omezili na pole do $1\,000\,000$ prvků a dali omezení, že hodnoty těchto prvků musí být celá čísla v intervalu od $-2\,000\,000\,000$ do $+2\,000\,000\,000$, množina všech takových polí bude sice konečná, ale astronomicky velká. Obecně tedy nepřichází v úvahu možnost systematicky vyzkoušet všechny možné vstupy a pro každý takový vstup otestovat, že za běhu algoritmu nedojde k chybě, že se algoritmus zastaví a vydá správný výsledek.

Samozřejmě je vhodné algoritmus otestovat na nějakých testovacích vstupech. Pokud na některém z testovacích vstupů nefunguje algoritmus správně, tak určitě korektní není. Ovšem to, že algoritmus funguje správně na testovacích vstupech, není zárukou toho, že bude správně fungovat i na všech ostatních vstupech.

Korektnost algoritmu se tedy musí dokazovat jiným způsobem. Při zdůvodňování toho, že algoritmus je korektní, je vhodné si tento úkol rozdělit na dva samostatné podúkoly:

(a) Prokázání toho, že za běhu algoritmu nedojde k žádné chybné nežádoucí operaci. Zdůvodnění toho, že pokud bude algoritmus generovat nějaký výstup, tak tento výstup bude odpovídat zadání, a zdůvodnění toho, že pokud se program zastaví, tak předtím vygeneruje nějaký výstup.
V této fázi analýzy korektnosti se nebude řešit, jestli se program vůbec někdy zastaví, nebo jestli vůbec někdy vygeneruje nějaký výstup.
(b) Prokázání toho, že algoritmus se pro každý přípustný vstup po konečném počtu kroků zastaví.

V Sekci 3.1 bude popsáno použití tzv. ***invariantů***, což je jedna z metod, která se dá použít pro prokazování vlastností algoritmu popsaných v bodě (a). V Sekci 3.2 bude popsán jeden z možných způsobů, jak prokazovat, že se algoritmus pro konečném počtu zastaví, tj. že má vlastnost uvedenou v bodě (b).

3.1 Invarianty

Invarianty jsou prostředkem pro prokázání toho, že algoritmus během svého běhu nikdy neudělá nic „špatného“. Různých „špatných“ věcí, které by algoritmus během svého vykonávání mohl udělat, je hodně — provedení nějaké chybné nepovolené operace, vygenerování chybného výstupu, zastavení se bez toho, aby byl vygenerován výstup, apod.

Invariant je libovolná podmínka, která musí být v určitém místě kódu algoritmu vždy (tj. ve všech možných výpočtech pro všechny možné vstupy) splněna v okamžiku, kdy se tímto místem při vykonávání jednotlivých instrukcí algoritmu prochází.

Předtím, než si tento pojem popíšeme podrobně a než ho budeme ilustrovat na příkladech, je vhodné nejprve zavést několik pomocných pojmů, které se nám budou v dalším výkladu hodit.

Předpokládejme, že máme algoritmus *Alg*, o kterém chceme dokázat, že je korektním řešením problému P. Symbolem *In* si označme množinu všech možných vstupů, které jsou přípustné podle specifikace vstupu v popisu problému P.

Navíc předpokládejme, že algoritmus *Alg* máme reprezentován ve formě grafu řídícího toku.

Každému vstupu w z množiny *In* odpovídá nějaký výpočet algoritmu *Alg*, který můžeme reprezentovat jako (konečnou nebo nekonečnou) posloupnost konfigurací. Konfigurace α je ***dosažitelná***, jestliže existuje nějaký vstup $w \in \mathit{In}$ takový, že α patří do posloupnosti konfigurací reprezentující výpočet algoritmu *Alg* nad vstupem w. Jinými slovy, konfigurace je dosažitelná, jestliže se při alespoň jednom ze všech možných výpočtů může algoritmus do této konfigurace dostat.

Pokud si nyní vezmeme nějaký libovolný řídící stav q z grafu řídícího toku algoritmu *Alg*, je pro tento řídící stav jednoznačně dána množina všech dosažitelných konfigurací, ve kterých je řídícím stavem stav q. Označme tuto množinu jako $\mathcal{S}_q$.

Pokud tedy máme řídící stavy $0, 1, \ldots, n$, množina všech dosažitelných konfigurací se nám tímto rozpadne na vzájemně disjunktní množiny $\mathcal{S}_0$, $\mathcal{S}_1$, $\ldots$, $\mathcal{S}_n$.

Invarianty jsou tvrzení, která se vyjadřují o konfiguracích. Ne každé takové tvrzení je invariant, ale každý invariant má podobu takového tvrzení. Předpokládá se, že takovéto tvrzení, vyjadřující se o konfiguracích, je možné zapsat formulí predikátové logiky a pokud máme dánu nějakou konkrétní konfiguraci, je touto konfigurací jednoznačně určena interpretace a valuace, při které bude daná formule vyhodnocována.

Daná formule tedy pro některé konfigurace může platit a pro jiné neplatit. Pokud máme v konfiguracích reprezentován obsah paměti jako přiřazení hodnot proměnným, mohou tyto proměnné vystupovat v dané formuli jako volné proměnné. Valuace, při které je tato formule vyhodnocována je pak dána přiřazením stejných hodnot proměnným, jako jsou hodnoty těchto proměnných v příslušné konfiguraci.

Příklad 3.1 Předpokládejme, že máme konfigurace, ve kterých se vyskytují proměnné i a k, kterým mohou být přiřazeny jako hodnoty celá čísla. Příkladem formule, kterou v těchto konfiguracích můžeme vyhodnocovat, je třeba formule

$$(1 \leq i) \wedge (i \leq k) .$$

Pokud budeme mít například konfiguraci α, ve které bude mít proměnná i hodnotu 5 a proměnná k hodnotu 14, bude tato formule v konfiguraci α platit, protože $1 \leq 5$ a $5 \leq 14$.

Pokud si oproti tomu vezmeme nějakou konfiguraci α', ve které bude mít proměnná i hodnotu 5 a proměnná k hodnotu 3, výše uvedená formule v této konfiguraci platit nebude, protože není pravda, že $5 \leq 3$.

Vezměme si nyní nějaký řídící stav q z grafu řídícího toku algoritmu *Alg* a nějakou formuli φ, která se vyjadřuje o konfiguracích, a kterou je možné vyhodnotit ve všech možných konfiguracích (ne jen těch dosažitelných), ve kterých je řídícím stavem stav q (tj. předpokládáme, že ve všech těchto konfiguracích je definována pravdivostní hodnota této formule φ).

Formule φ je ***invariantem*** v řídícím stavu q, jestliže φ platí ve všech konfiguracích z množiny $\mathcal{S}_q$. Formule φ je tedy invariantem ve stavu q, jestliže v každém možném výpočtu, který může algoritmus *Alg* provést, platí φ v každé konfiguraci v tomto výpočtu, ve které je aktuální řídící stav q.

Například výše uvedená formule $(1 \leq i) \wedge (i \leq k)$ bude invariantem ve stavu q, jestliže vždy, když se algoritmus dostane do stavu q, budou v daném okamžiku platit pro hodnoty proměnných i a k vztahy $1 \leq i$ a $i \leq k$. Pokud bude existovat alespoň jeden vstup, kde alespoň jednou během výpočtu ve stavu q bude $i < 1$ nebo $i > k$, tato formule invariantem nebude.

* * *

Analýza algoritmů pomocí invariantů je užitečná nejen pro dokazování korektnosti algoritmů, ale mnohdy se hodí i při dokazování dalších vlastností algoritmů.

Je třeba upozornit, že hledání vhodných invariantů, které pro daný algoritmus platí, není vždy snadné, a není to něco, co by se dalo plně automatizovat (i když některé jednoduché typy invariantů se dají hledat a ověřovat algoritmicky). Formulace správných invariantů vyžaduje porozumění tomu, jak se algoritmus chová a na jakých principech pracuje.

Dá se říci, že invarianty slouží k vyjádření určitých očekávání, která má ten, kdo daný algoritmus navrhuje nebo analyzuje, ohledně hodnot proměnných v daném bodě programu, vzájemných vztahů mezi těmito hodnotami, apod. Když jsou tato očekávání explicitně zformulována ve formě určitých konkrétních tvrzení (případně i formalizována pomocí formulí), dá se s nimi dále pracovat. Dá se dokazovat, že platí, nebo se naopak dají hledat příklady vstupů, pro které v nějakém okamžiku výpočtu dané tvrzení neplatí, dají se dále zpřesňovat, dají se použít k přesnější formulaci předpokladů, za kterých algoritmus funguje správně, apod.

* * *

V následujícím textu si ukážeme analýzu algoritmu pomocí invariantů na příkladu Algoritmu 1 popsaného grafem řídícího toku na Obrázku 2.4. Než ukážeme celkový důkaz korektnosti tohoto algoritmu pomocí invariantů, dáme si pro začátek nejprve skromnější cíl, a to ukázat, že daný algoritmus nikdy nepřistupuje k prvkům pole A mimo povolený rozsah indexů $0, 1, \ldots, n-1$.

Algoritmus 1 pracuje s proměnnými n, i a k, které nabývají celočíselných hodnot. Dále se v algoritmu pracuje s polem A a s pomocnou proměnnou *result*, tyto dvě proměnné ale nejsou z hlediska indexů pole A, ke kterým se v algoritmu přistupuje, důležité.

Analýza by měla začít tím, že zkusíme pro každý řídící stav v grafu řídícího toku na Obrázku 2.4 zformulovat, co by mělo platit pro hodnoty proměnných n, i a k vždy, když se algoritmus během výpočtu bude nacházet v daném stavu.

V grafu na Obrázku 2.4 jsou řídící stavy očíslovány čísly $0, 1, \ldots, 7$. Pro každý z těchto stavů tedy zkusíme vytvořit odpovídající formuli, ve které se budou n, i a k vyskytovat jako volné proměnné. Pro stav 0 vytvoříme formuli φ_0, pro stav 1 formuli φ_1, atd. Když tedy máme 8 stavů očíslovaných $0, 1, \ldots, 7$, vytvoříme pro těchto 8 stavů celkem 8 formulí $\varphi_0, \varphi_1, \ldots, \varphi_7$.

Při vytváření těchto formulí můžeme použít následující úvahy:

- Pokud nepředpokládáme, že na začátku výpočtu by byly proměnné automaticky inicializovány na nějakou defaultní hodnotu (např. 0), můžeme toho říci o hodnotách proměnných v počátečním stavu (zde je to stav 0) jen velmi málo. V podstatě o nich můžeme říci jen to, co vyplývá ze specifikace přípustných vstupních hodnot.

  U Algoritmu 1 je ve specifikaci problému Max-Element, který by měl tento algoritmus řešit, řečeno, že se předpokládá, že počet prvků pole A je v proměnné n a že platí $n \geq 1$. Vzhledem k tomu, že se v algoritmu hodnota proměnné n nikde nemění (nikde v kódu není žádné přiřazení do proměnné n), bude vztah $n \geq 1$ platit ve všech stavech.

- Ve stavu 1 bude platit vše, co ve stavu 0. Při přechodu do tohoto stavu ze stavu 0 se provede přiřazení $k := 0$ (instrukce I_0), takže po tomto přiřazení bude mít proměnná k hodnotu 0. Vzhledem k tomu, že do stavu 1 se algoritmus nemůže dostat jinak než tímto přechodem ze stavu 0, bude tedy ve stavu 1 navíc oproti stavu 0 platit, že $k = 0$.

- Při přechodu ze stavu 1 do stavu 2 se provede přiřazení $i := 1$ (instrukce I_1). Nemůžeme ovšem tvrdit, že vždy, když algoritmus bude ve stavu 2 bude platit $i = 1$. Když se podíváme na kód algoritmu a příslušný graf řídícího toku, vidíme, že stav 2 je vstupním bodem do cyklu. Kromě toho, že se do stavu 2 dá dostat ze stavu 1, skáče se na něj rovněž po každém provedení těla cyklu. Stav 2 je také místem, kde se provádí test ukončení cyklu a odkud se dá z cyklu vyskočit.

  Při formulaci toho, co by mělo ve stavu 2 vždy platit, tedy musíme brát v úvahu všechny tyto případy. Prozkoumáním kódu Algoritmu 1 zjistíme, že proměnná i nabývá postupně hodnot $1, 2, \ldots, n$ a že provádění cyklu skončí v okamžiku, kdy $i = n$. Můžeme tedy zkusit

zformulovat invariant pro stav 2 a to, že v tomto stavu by mělo vždy platit, že

$$1 \leq i \leq n\,.$$

(Poznamenejme, že $1 \leq i \leq n$ budeme používat jako zkrácený zápis pro $(1 \leq i) \wedge (i \leq n)$.)

- Do stavu 3 se dá dostat jen ze stavu 2 provedením instrukce I_2, což je otestování podmínky $i < n$. Tímto krokem se nemění hodnoty žádných proměnných. Ve stavu 3 by tedy mělo platit vše, co ve stavu 2, a navíc ještě musí platit podmínka $i < n$. Pokud je tedy výše uvedený invariant $1 \leq i \leq n$ pro stav 2 v pořádku, ve stavu 3 by mělo vždy platit $1 \leq i < n$ (tj. neplatí jen neostrá nerovnost $i \leq n$, ale ostrá nerovnost $i < n$).

  Provedením instrukcí I_4, I_5 a I_6 se hodnoty proměnných n a i nemění. Pokud tedy vztahy $1 \leq i < n$ platí ve stavu 3, měly by platit i ve stavech 4 a 5.

- Do stavu 6 se dá dostat jen ze stavu 2 a při tomto přechodu (instrukcí I_3) se nemění hodnota žádné proměnné. Pokud tedy ve stavu 2 musí platit, že $i \leq n$, a do stavu 6 se ze stavu 2 přejde jen při splnění podmínky $i \geq n$, můžeme z toho vyvodit, že ve stavu 6 by mělo platit $i = n$.

- Přejitím ze stavu 6 do stavu 7 instrukcí I_8 se mění jen hodnota proměnné *result*. Všechny vztahy týkající se proměnných n, i a k, které platí ve stavu 6, by tedy měly platit i ve stavu 7.

- Co se týká hodnoty proměnné k, na začátku je tato hodnota inicializována na nulu a během dalšího výpočtu se mění jen při provedení instrukce I_6, kdy je do ní přiřazena hodnota proměnné i. Proměnná i začíná s hodnotou 1 a v průběhu výpočtu se její hodnota zvětšuje (instrukcí I_7). V průběhu výpočtu by tedy neustále mělo platit, že $k \geq 0$, a ve většině stavů bude navíc platit $k < i$, pouze po provedení instrukce I_6 bude platit $k = i$, takže ve stavu 5 by mělo být $k \leq i$.

Zkusme nyní zapsat výsledek výše popsaných úvah pomocí formulí, tj. zkusme vytvořit příslušné formule φ_0, φ_1, ..., φ_7 pro jednotlivé řídící stavy:

- φ_0: $(n \geq 1)$
- φ_1: $(n \geq 1) \wedge (k = 0)$
- φ_2: $(n \geq 1) \wedge (1 \leq i \leq n) \wedge (0 \leq k < i)$
- φ_3: $(n \geq 1) \wedge (1 \leq i < n) \wedge (0 \leq k < i)$
- φ_4: $(n \geq 1) \wedge (1 \leq i < n) \wedge (0 \leq k < i)$
- φ_5: $(n \geq 1) \wedge (1 \leq i < n) \wedge (0 \leq k \leq i)$
- φ_6: $(n \geq 1) \wedge (i = n) \wedge (0 \leq k < n)$
- φ_7: $(n \geq 1) \wedge (i = n) \wedge (0 \leq k < n)$

Tím, že tyto formule zapíšeme, tak samozřejmě ještě není nijak zaručeno, že tyto formule musí v příslušných řídících stavech skutečně vždy platit. Zatím jsou to pouhé hypotézy o tom, jak by invarianty v daných stavech mohly vypadat. Když je však máme takto explicitně zapsané, můžeme přistoupit k tomu, že zkusíme dokázat, že opravdu platí. Při tomto dokazování se nám může podařit platnost těchto hypotéz dokázat, ale může se také ukázat, že některé formule ve skutečnosti pro některé dosažitelné konfigurace neplatí a je potřeba je upravit.

* * *

Při dokazování toho, že příslušné formule jsou skutečně invarianty, se postupuje tak, že se pro každou jednotlivou instrukci I (v našem případě pro instrukce $I_0, I_1, \ldots, I_8$) prokáže, že jestliže platí příslušný invariant před provedením této instrukce, pak platí i odpovídající invariant po provedení této instrukce.

Vezměme si nějakou instrukci I, kterou je označena hrana vedoucí z řídícího stavu q do řídícího stavu q'. Řekněme, že jsme zformulovali hypotézu, že ve stavu q by měla vždy platit formule φ a ve stavu q' by měla vždy platit formule φ'. Pro instrukci I je třeba dokázat, že pro každou konfiguraci $\alpha = (q, \mathit{mem})$, ve které platí formule φ, bude v konfiguraci $\alpha' = (q', \mathit{mem}')$, do které se algoritmus dostane z konfigurace α provedením instrukce I (tj. $\alpha \xrightarrow{I} \alpha'$), platit formule φ'.

Připomeňme, že instrukce I může být buď nějaké přiřazení nebo to může být test podmínky. Poněkud jednodušší je případ, kdy instrukce I je test nějaké podmínky [B], proto začněme popisem tohoto případu.

- **Případ, kdy I je test podmínky:**

  Předpokládejme, že instrukce I je testem podmínky [B]. Při provedení takovéto instrukce se nemění obsah paměti. Jediné, co se změní, je řídící stav. Z konfigurace $\alpha = (q, \mathit{mem})$ se provedením instrukce I přejde do konfigurace $\alpha' = (q', \mathit{mem})$, přičemž tento krok se provede jedině tehdy, pokud je konfiguraci α splněna podmínka B.

  Vše, co platí v konfiguraci α, musí tedy platit i v konfiguraci α' (protože obsah paměti se nezměnil). Navíc musí v konfiguraci α' platit i podmínka B, protože jinak by se do konfigurace α' z konfigurace α instrukcí I nepřešlo.

  V podstatě stačí tedy dokázat, že pokud splněna podmínka φ a pokud navíc platí B, tak bude platit i φ', tj. je třeba dokázat, že platí implikace

  $$(\varphi \wedge B) \rightarrow \varphi' .$$

  Přesněji řečeno, vzhledem k tomu, že tento vztah musí platit pro všechny možné konfigurace, a tedy pro všechny možné obsahy paměti, a tedy pro všechny všechny možné hodnoty proměnných, je třeba dokázat, že platí formule, která vznikne z výše uvedené formule doplněním univerzálních kvantifikátorů pro všechny možné proměnné, které se v této formuli vyskytují jako volné proměnné.

  Pokud se například vyskytují v našich formulích jako volné proměnné proměnné n, i a k, které mohou nabývat hodnoty z oboru celých čísel $\mathbb{Z}$, je třeba pro instrukci I dokázat, že platí formule

  $$(\forall n \in \mathbb{Z})(\forall i \in \mathbb{Z})(\forall k \in \mathbb{Z})\,(\varphi \wedge B \rightarrow \varphi') .$$

  Všimněme si tedy, že zatímco formule φ a φ' přiřazené stavům typicky obsahují volné proměnné odpovídající proměnným programu, formule, která se dokazuje pro danou instrukci I, je uzavřená.

- **Případ, kdy I je přiřazení:**

  Předpokládejme, že instrukce I je přiřazení $x := E$, kde x je proměnná a E výraz. V konfiguraci $\alpha = (q, \mathit{mem})$ se vyhodnotí výraz E. Výsledná hodnota se přiřadí do proměnné x a řídící stav se změní na q'. Provedením instrukce I se tedy v tomto případě přejde do konfigurace $\alpha' = (q', \mathit{mem}')$, kde obsah paměti mem' je stejný jako mem s tím rozdílem, že proměnné x je přiřazena spočítaná hodnota výrazu E.

  V tomto případě je vhodné od sebe odlišit hodnotu proměnné x v konfiguraci α a hodnotu této proměnné v konfiguraci α'. Z toho důvodu zavedeme pomocnou proměnnou x', která bude reprezentovat hodnotu proměnné x v konfiguraci α, tj. po provedení příkazu I. Symbol x bude reprezentovat hodnotu proměnné x před provedením tohoto příkazu.

  Označme symbolem φ'' formuli, kterou dostaneme z formule φ' nahrazením všech volných výskytů proměnné x symbolem x' (tj. v této formuli přejmenujeme volné výskyty x na x'). Formule φ'' tedy vyjadřuje, co by mělo platit ve stavu q', kdybychom proměnnou x přejmenovali na x'. Hodnota proměnné x' je dána hodnotou výrazu E v konfiguraci α (tj. s původní hodnotou proměnné x).

V tomto případě je tedy třeba ukázat, že pokud platí podmínka φ a do proměnné x' přiřadíme hodnotu výrazu E, bude platit podmínka φ'', tj. je třeba dokázat platnost implikace

$$(\varphi \wedge (x' = E)) \rightarrow \varphi'' ,$$

či lépe řečeno, platnost uzavřené formule, kterou z ní dostaneme, pokud přidáme univerzální kvantifikátory pro všechny volné proměnné (včetně nově přidané pomocné proměnné x').

Pokud tedy budeme mít celočíselné proměnné n, i, k a v instrukci I se bude přiřazovat do proměnné k například hodnota výrazu $3 * k + i + 1$, budeme dokazovat platnost formule

$$(\forall n \in \mathbb{Z})(\forall i \in \mathbb{Z})(\forall k \in \mathbb{Z})(\forall k' \in \mathbb{Z})\,(\varphi \wedge (k' = 3 * k + i + 1) \rightarrow \varphi'') ,$$

kde φ'' bude formule, kterou dostaneme z formule φ' nahrazením volných výskytů proměnné k proměnnou k'.

* * *

Ukažme si nyní výše popsaný postup na příkladu Algoritmu 1 a dříve uvedených formulí φ_0, φ_1, ..., φ_7. Postupně projdeme všechny instrukce $I_0, I_1, \ldots, I_8$ a pro každou z nich dokážeme, že opravdu dané platnost daných formulí zachovává. Pro názornost je u každé instrukce uvedeno, z kterého stavu a do kterého stavu se touto instrukcí přechází. Zápis $q \rightarrow q'$ u dané instrukce znamená, že se přechází touto instrukcí ze stavu q do stavu q'. Pro přehlednost jsou ve formulích vypuštěny univerzální kvantifikátory přes celou formuli.

Pro přehlednost jsou zde zopakovány formule $\varphi_0, \varphi_1, \ldots, \varphi_7$:

- φ_0: $(n \geq 1)$
- φ_1: $(n \geq 1) \wedge (k = 0)$
- φ_2: $(n \geq 1) \wedge (1 \leq i \leq n) \wedge (0 \leq k < i)$
- φ_3: $(n \geq 1) \wedge (1 \leq i < n) \wedge (0 \leq k < i)$
- φ_4: $(n \geq 1) \wedge (1 \leq i < n) \wedge (0 \leq k < i)$
- φ_5: $(n \geq 1) \wedge (1 \leq i < n) \wedge (0 \leq k \leq i)$
- φ_6: $(n \geq 1) \wedge (i = n) \wedge (0 \leq k < n)$
- φ_7: $(n \geq 1) \wedge (i = n) \wedge (0 \leq k < n)$

Začněme nejprve těmi instrukcemi, které jsou testy podmínek:

- I_2 — test $[i < n]$, změna stavu $2 \rightarrow 3$:

  Je třeba dokázat platnost implikace $\varphi_2 \wedge (i < n) \rightarrow \varphi_3$. Formule φ_2 a φ_3 jsou téměř identické. Jediný rozdíl mezi nimi je v tom, že ve formuli φ_3 je uvedeno $i < n$, zatímco ve formuli φ_2 jen $i \leq n$. Pokud tedy platí φ_2 a zároveň $i < n$, tak určitě platí i φ_3.

- I_3 — test $[i \geq n]$, změna stavu $2 \rightarrow 6$:

  Je třeba dokázat platnost implikace $\varphi_2 \wedge (i \geq n) \rightarrow \varphi_6$. Předpokládejme, že platí φ_2 a $i \geq n$. Z platnosti φ_2 plyne, že $n \geq 1$. Podle φ_2 platí $i \leq n$. Z $i \leq n$ a $i \geq n$ dostáváme $i = n$. Podle φ_2 platí $0 \leq k < i$. Protože $i = n$, platí i $0 \leq k < n$.

- I_4 — test $[A[i] \leq A[k]]$, změna stavu $3 \rightarrow 5$:

  Je třeba dokázat implikaci $\varphi_3 \wedge (A[i] \leq A[k]) \rightarrow \varphi_5$. Formule φ_3 a φ_5 jsou téměř identické. Jediný rozdíl mezi nimi je v tom, že ve formuli φ_3 je uvedeno $k < i$ a ve formuli φ_5 je $k \leq i$. Pokud však platí $k < i$, zjevně platí i $k \leq i$.

- I_5 — test $[A[i] > A[k]]$, změna stavu $3 \rightarrow 4$:

  Je třeba dokázat implikaci $\varphi_3 \wedge (A[i] > A[k]) \rightarrow \varphi_4$. Tato implikace zjevně platí, neboť formule φ_3 a φ_4 jsou identické.

Proberme nyní ty instrukce, kde se provádí nějaké přiřazení:

- I_0 — přiřazení $k := 0$, změna stavu $0 \to 1$:

  Je třeba dokázat platnost implikace $\varphi_0 \wedge (k' = 0) \to \varphi_1'$, kde φ_1' je formule $(n \geq 1) \wedge (k' = 0)$ (vznikla nahrazením výskytů proměnné k ve formuli φ_1 symbolem k'). Daná implikace zjevně platí, neboť podle φ_0 je $n \geq 1$.

- I_1 — přiřazení $i := 1$, změna stavu $1 \to 2$:

  Je třeba dokázat implikaci $\varphi_1 \wedge (i' = 1) \to \varphi_2'$, kde φ_2' je formule $(n \geq 1) \wedge (1 \leq i' \leq n) \wedge (0 \leq k < i')$. Předpokládejme, že platí φ_1 a $i' = 1$, tedy že platí $n \geq 1$, $k = 0$, a $i' = 1$. Za těchto předpokladů zjevně platí $n \geq 1$, $1 \leq i' \leq n$ i $0 \leq k < i'$.

- I_6 — přiřazení $k := i$, změna stavu $4 \to 5$:

  Je třeba dokázat implikaci $\varphi_4 \wedge (k' = i) \to \varphi_5'$, kde φ_5' je formule $(n \geq 1) \wedge (1 \leq i < n) \wedge (0 \leq k' \leq i)$. Předpokládejme, že platí φ_4 a $k' = i$. Z platnosti φ_4 plyne $n \geq 1$ a $1 \leq i < n$. Protože $k' = i$, platí i $k' \leq i$. Protože $1 \leq i$, je $1 \leq k'$ a tedy i $0 \leq k'$.

- I_7 — přiřazení $i := i + 1$, změna stavu $5 \to 2$:

  Je třeba dokázat implikaci $\varphi_5 \wedge (i' = i + 1) \to \varphi_2'$, kde φ_2' je formule $(n \geq 1) \wedge (1 \leq i' \leq n) \wedge (0 \leq k < i')$. Předpokládejme, že platí φ_5 a $i' = i + 1$. Platí tedy $n \geq 1$, $1 \leq i < n$ a $0 \leq k \leq i$. Protože $1 \leq i$, je i $1 \leq i + 1$ a tedy $1 \leq i'$. Protože $i < n$, je $i + 1 \leq n$ a tedy $i' \leq n$. Podle φ_5 je $0 \leq k$. Protože $k \leq i$, je $k < i + 1$ a tedy $k < i'$.

- I_8 — přiřazení *result* $:= A[i]$, změna stavu $6 \to 7$:

  Je třeba dokázat implikaci $\varphi_6 \wedge (\mathit{result} = A[k]) \to \varphi_7'$, kde φ_7' je stejná formule jako φ_7, protože proměnná *result* se v ní nevyskytuje. Tato implikace zjevně platí, protože formule φ_6 a φ_7 jsou identické.

Pokud tedy v počáteční konfiguraci bude platit invariant φ_0, tj. $n \geq 1$, bude po provedení každého kroku vždy platit invariant pro příslušný řídící stav. Můžeme tak tedy ověřit, že během výpočtu se nikdy nebude přistupovat mimo meze pole A. K prvkům pole A se přistupuje pouze v instrukcích I_4, I_5 a I_8.

V instrukcích I_4 a I_5 se prvkům pole A přistupuje při vyhodnocování podmínek $[A[i] \leq A[k]]$ a $[A[i] > A[k]]$, které se vyhodnocují v konfiguracích, kde aktuální řídící stav je 3. Ve stavu 3 platí invariant

$$(n \geq 1) \wedge (1 \leq i < n) \wedge (0 \leq k < i)\,,$$

určitě tedy v daném okamžiku platí $0 \leq i < n$ a $0 \leq k < n$, takže přístupy k prvkům $A[i]$ i $A[k]$ jsou v pořádku.

Instrukce I_8 se provádí v konfiguraci, kdy aktuální řídící stav je 6, kdy platí invariant φ_6. Instrukci I_8 se přistupuje k prvku $A[k]$. Podle φ_6 v době vyhodnocování tohoto výrazu platí $0 \leq k < n$, takže tento přístup k prvku pole A je také v pořádku.

* * *

Zatím jsme ukázali, že Algoritmus 1 nikdy během výpočtu neselže kvůli chybnému přístupu k prvku pole. Pokud chceme ale dokázat, že pokud tento algoritmus svůj výpočet skončí, tak vrací správný výsledek, budeme muset přidat další invarianty.

Jak jsme viděli, v daném řídícím stavu může platit více různých invariantů. Pokud máme více takových invariantů, které platí v jednom stavu, platí v tomto stavu samozřejmě i konjunkce těchto invariantů. Takovéto menší invarianty tedy můžeme přidávat postupně, přičemž můžeme využívat dříve dokázané invarianty, s tím, že vždy ze všech těchto dílčích invariantů můžeme vytvořit jednu větší formuli pomocí konjunkce.

Algoritmus 1 je velmi jednoduchý, takže jeho prozkoumáním snadno přijdeme na to, v čem spočívá hlavní myšlenka fungování tohoto algoritmu. Stroj, který tento algoritmus vykonává,

si v proměnné k pamatuje index prvku, který je největší z dosud zpracovaných prvků pole A, tj. z prvků s indexy $0, 1, \ldots, i-1$.

Z využitím tohoto pozorování můžeme zformulovat následující invarianty $\psi_0, \psi_1, \ldots, \psi_7$ jako rozšíření dříve dokázaných invariantů $\varphi_0, \varphi_1, \ldots, \varphi_7$ (invariant ψ_q odpovídá řídícímu stavu q, kde $q = 0, 1, \ldots, 7$):

- ψ_0: φ_0
- ψ_1: $\varphi_1 \wedge (\forall j \in \mathbb{N})(0 \leq j < 1 \rightarrow A[j] \leq A[k])$
- ψ_2: $\varphi_2 \wedge (\forall j \in \mathbb{N})(0 \leq j < i \rightarrow A[j] \leq A[k])$
- ψ_3: $\varphi_3 \wedge (\forall j \in \mathbb{N})(0 \leq j < i \rightarrow A[j] \leq A[k])$
- ψ_4: $\varphi_4 \wedge (\forall j \in \mathbb{N})(0 \leq j < i \rightarrow A[j] \leq A[k]) \wedge (A[i] > A[k])$
- ψ_5: $\varphi_5 \wedge (\forall j \in \mathbb{N})(0 \leq j \leq i \rightarrow A[j] \leq A[k])$
- ψ_6: $\varphi_6 \wedge (\forall j \in \mathbb{N})(0 \leq j < n \rightarrow A[j] \leq A[k])$
- ψ_7: $\varphi_7 \wedge (\mathit{result} = A[k]) \wedge (\forall j \in \mathbb{N})(0 \leq j < n \rightarrow A[j] \leq \mathit{result}) \wedge (\exists j \in \mathbb{N})(0 \leq j < n \wedge A[j] = \mathit{result})$

Tyto invarianty už nebudeme dokazovat tak podrobně jako invarianty $\varphi_0, \varphi_1, \ldots, \varphi_7$ a rozebereme si pouze některé zajímavější případy. Invariant ψ_0 je stejný jako už dříve dokázaný invariant φ_0.

Invariant ψ_1 se dá dokázat i bez toho, abychom rozebírali jednotlivé instrukce. Invariant φ_1 už byl dokázán pro stav 1 dříve a podle tohoto invariantu ve stavu 1 platí $k = 0$. Při dokazování formule

$$(\forall j \in \mathbb{N})(0 \leq j < 1 \rightarrow A[j] \leq A[k])$$

se stačí soustředit na ta $j \in \mathbb{N}$, pro která platí $0 \leq j < 1$. Zjevně existuje jen jedno takové j a to $j = 0$. Protože $A[0] \leq A[0]$, pro $j = 0$ zjevně platí $A[j] \leq A[k]$.

Pro ilustraci si vezměme důkazy pro instrukce I_4, I_6 a I_8. Důkazy pro ostatní instrukce jsou velmi jednoduché a jsou ponechány čtenáři jako cvičení.

- I_4 — test $[A[i] \leq A[k]]$, změna stavu $3 \rightarrow 5$:

  Je třeba dokázat implikaci $\psi_3 \wedge (A[i] \leq A[k]) \rightarrow \psi_5$. Předpokládejme, že platí ψ_3 a $A[i] \leq A[k]$. Formule φ_5 již byla dokázána dřív, takže zbývá dokázat, že pro každé $j \in \mathbb{N}$, kde $0 \leq j \leq i$, platí $A[j] \leq A[k]$. Pokud $0 \leq j \leq i$, tak buď $0 \leq j < i$, a pak $A[j] \leq A[k]$ podle ψ_3, nebo $j = i$, a pak $A[j] \leq A[k]$, protože $A[i] \leq A[k]$.

- I_6 — přiřazení $k := i$, změna stavu $4 \rightarrow 5$:

  Je třeba dokázat implikaci $\psi_4 \wedge (k' = i) \rightarrow \psi_5'$, kde ψ_5' je formule $\varphi_5' \wedge (\forall j \in \mathbb{N})(0 \leq j \leq i \rightarrow A[j] \leq A[k'])$. Předpokládejme, že platí ψ_4 a $k' = i$. Formule φ_5' již byla popsána a dokázána dříve, zbývá tedy dokázat, že pro každé $j \in \mathbb{N}$, kde $0 \leq j \leq i$, platí $A[j] \leq A[k']$. Pokud $0 \leq j \leq i$, tak buď $0 \leq j < i$ nebo $j = i$.

  Rozeberme nejprve případ $j = i$. Protože $j = i$ a $k' = i$, tak $A[j] = A[i] = A[k']$, a tedy $A[j] \leq A[k']$.

  Předpokládejme nyní, že $0 \leq j < i$. Podle ψ_4 je $A[i] > A[k]$, a protože $k' = i$, tak z toho plyne, že $A[k] < A[k']$. Podle ψ_4 pro každé $j \in \mathbb{N}$ takové, že $0 \leq j < i$, platí $A[j] \leq A[k]$. Pro každé takové j tedy platí $A[j] < A[k']$, z čehož plyne, že $A[j] \leq A[k']$.

- I_8 — přiřazení $\mathit{result} := A[i]$, změna stavu $6 \rightarrow 7$:

  Je třeba dokázat implikaci $\psi_6 \wedge (\mathit{result}' = A[k]) \rightarrow \psi_7'$, kde ψ_7' je formule $\varphi_7' \wedge (\forall j \in \mathbb{N})(0 \leq j < n \rightarrow A[j] \leq \mathit{result}') \wedge (\exists j \in \mathbb{N})(0 \leq j < n \wedge A[j] = \mathit{result}')$. Předpokládejme, že platí ψ_6 a $\mathit{result}' = A[k]$. Formule φ_7' již bylo popsána a dokázána dřív. Protože $\mathit{result}' = A[k]$, stačí dokázat, že platí $(\forall j \in \mathbb{N})(0 \leq j < n \rightarrow A[j] \leq A[k])$. Toto ale přímo plyne z ψ_6.

  To, že platí $(\exists j \in \mathbb{N})(0 \leq j < n \wedge A[j] = \mathit{result}')$, plyne z toho, že $\mathit{result}' = A[k]$, takže stačí položit $j = k$.

V koncovém stavu 7 hodnota proměnné *result* splňuje obě podmínky, které má splňovat podle zadání problému MAX-ELEMENT:

- $A[j] \leq$ *result* pro všechna $j \in \mathbb{N}$, kde $0 \leq j < n$,
- existuje $j \in \mathbb{N}$ takové, že $0 \leq j < n$ a $A[j] =$ *result*.

Tím je tedy dokázáno, že pokud Algoritmus 1 dosáhne koncového stavu (stavu 7), bude výsledná hodnota v proměnné *result* odpovídat požadavkům ve specifikaci problému MAX-ELEMENT.

* * *

V praxi se většinou tak podrobná analýza invariantů, jaká byla předvedena na předchozích stránkách, nedělá. Pokud jsou stanoveny invarianty pro jednotlivé stavy, ověření, že tyto invarianty jsou zachovávány při provedení každé jednotlivé instrukce, je už poměrně rutinní i když poněkud pracná záležitost.

Ve většině případů ani není potřeba popisovat invarianty ve všech bodech programu (ve všech řídících stavech), ale pouze v určitých klíčových místech, s tím, že doplnění invariantů pro zbylé řídící stavy je už relativně snadné. Takovými důležitými místy v kódu jsou především ta místa, kde se vstupuje do cyklů a místa, kde se naopak z cyklu vystupuje (tj. tam, kde se provádí test na ukončení cyklu). Například pro Algoritmus 1 je takovým důležitým místem řídící stav 2.

Důkaz toho, že v daném místě příslušný invariant platí, je pak rozdělen na tři části:

- důkaz toho, že invariant platí před prvním provedením cyklu,
- důkaz toho, že jestliže je splněna podmínka pro pokračování v provádění cyklu a invariant platí, tak bude platit po dalším jednom provedení těla cyklu,
- důkaz toho, že jestliže je provádění cyklu ukončeno, protože není splněna podmínka pro provádění cyklu, tak invariant platí.

* * *

V ukázce důkazu toho, že v Algoritmu 1 platí příslušné invarianty, jsme předpokládali, že algoritmus je realizován strojem, kde hodnoty proměnných n, i a k mohou být libovolně velká přirozená čísla. V praxi bude takový algoritmus často implementován v programovacím jazyce, kde jsou maximální hodnoty, kterých mohou nabývat celočíselné proměnné, nějak omezeny. Například může být rozsah hodnot těchto proměnných omezen na 32-bitová nebo 64-bitová čísla.

V takové implementaci výše odvozené invarianty ve skutečnosti neplatí a příslušné podmínky je třeba doplnit o další omezení. Především se pak taková implementace není schopna vypořádat s libovolně velkým vstupem, ale jen se vstupy do určité velikosti (např. maximální délka pole na vstupu může být omezena na hodnotu $2^{31} - 1$, apod.). V praxi však taková omezení většinou nevadí.

Analýza invariantů se dá snadno rozšířit i na tyto případy. Při dokazování korektnosti se pak mohou analyzovat i další typy chyb. Například se dá dokazovat, že pokud data na vstupu budou splňovat příslušná omezení na velikost vstupních dat, tak při provádění jednotlivých instrukcí nikdy nedojde k přetečení obsahu proměnné.

Dokazování invariantů v případech, kdy se berou v úvahu různé implementační detaily, je komplikovanější, protože různé operace pak nemají úplně stejné vlastnosti, které platí na „ideálních“ matematických objektech. Například při použití 32-bitových bezznaménkových čísel (např. typ `unsigned int` v jazyce C) nemusí být pravda, že $i < i + 1$, protože pro $i = 4\,294\,967\,295$ (tj. $i = 2^{32} - 1$) je $i + 1 = 0$.

Určitým specifickým typem invariantů, který jsme v předchozím textu příliš nediskutovali, jsou datové typy přiřazené jednotlivým proměnným. Například na informaci, že proměnná k má v určitém místě programu jako svou hodnotu přirozené číslo, se můžeme dívat jako na invariant. Oproti invariantům popsaným výše jsou navíc invarianty týkající se datových typů často vynucovány přímo překladačem, kdy například překladač nedovolí přiřadit do proměnné hodnotu nesprávného typu.

3.2 Konečnost výpočtu

V předchozí sekci bylo pro Algoritmus 1 dokázáno, že jestliže tento algoritmus skončí, bude výsledek v proměnné *result* odpovídat zadání. To ale ještě neznamená, že je zaručeno, že tento algoritmus opravdu skončí. Ve skutečnosti je Algoritmus 1 velmi jednoduchý a je u něj téměř očividné, že zaručeně skončí. Způsob dokazování toho, že tento algoritmus opravdu po konečném počtu skončí, který bude popsán v této sekci, je v případě tohoto konkrétního algoritmu použitím kanónu na vrabce. Slouží ale jako ukázka toho, jak se obecně dá při podobných důkazech postupovat.

Obecně se mohou vyskytnout dva následující případy nekonečných výpočtů:

a) Výpočet, ve kterém se nějaká konfigurace zopakuje. Algoritmus pak od tohoto okamžiku začne provádět stejné kroky a dostává se do stejných konfigurací, v jakých už předtím byl.

b) Výpočet, ve kterém se objevují stále nové a nové konfigurace. Vzhledem k tomu, že určitým omezeným počtem bitů se dá reprezentovat jen konečný počet různých konfigurací, musí při takových výpočtech docházet k tomu, že velikost konfigurací stále průběžně narůstá, protože nové konfigurace potřebují stále více a více paměti. V praxi takový výpočet skončí kvůli nedostatku paměti, pokud se nechá běžet dostatečně dlouho, protože skutečné počítače nemají k dispozici neomezené množství paměti.

V řadě případů se dá celkem snadno poznat, že algoritmus zaručeně skončí po konečném počtu kroků, nebo že naopak může běžet donekonečna. Obecně to ale vůbec není snadné. Jako ilustraci toho, že nemusí být vůbec zřejmé, jestli se algoritmus zastaví pro všechny možné vstupy, uveďme následující jednoduchou proceduru (předpokládá se, že proměnná n může jako své hodnoty obsahovat libovolně velká přirozená čísla):

Algoritmus 2: Algoritmus pro vypisování Collatzových posloupností

```
1  COLLATZ (n):
2  begin
3      print n
4      while n > 1 do
5          if n mod 2 = 0 then
6              n := n / 2
7          else
8              n := 3 * n + 1
9          end
10         print n
11     end
12 end
```

Tento algoritmus vypisuje tzv. Collatzovy posloupnosti. Pokud například dostane jako vstup číslo 7, vypíše následující posloupnost:

$$7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1$$

U tohoto algoritmu není znám jediný příklad vstupu, pro který by se tento program nezastavil. Na druhou stranu není znám důkaz toho, že se tento algoritmus zastaví po konečném počtu kroků pro jakékoliv přirozené číslo n, které dostane jako vstup. Jedná se známý otevřený matematický problém. Ačkoliv se obecně považuje za dost pravděpodobné, že se tento algoritmus ve skutečnosti opravdu zastaví pro všechny vstupy, teoreticky je stále otevřená jednak možnost, že existuje nějaké číslo n, pro které se tento algoritmus nezastaví. Jednak je možné, že se pro nějaké číslo n během výpočtu nějaká hodnota větší než 1 zopakuje, a program pak bude vypisovat stejnou konečnou

posloupnost čísel stále dokola. Také je přípustná možnost, že pro nějaké číslo n se budou vypisovat donekonečna stále nová a nová čísla (hodnoty těchto čísel se pak budou muset průběžně stále zvětšovat, protože jinak by se musela začít opakovat stejná čísla).

Algoritmus 2 zde neuvádíme proto, že by měl nějaký speciální praktický význam, ale jako ilustraci toho, že už pro velmi jednoduché algoritmy může být velmi těžké určit, zda se pro každý vstup zastaví nebo ne. Všimněte si, že tento algoritmus má jen pár řádek, používá se v něm jediná proměnná, hodnoty této proměnné jsou přirozená čísla (není to žádný komplikovaný datový typ), se kterými se provádějí jen velmi jednoduché aritmetické operace. Přesto není známo, jestli se tento (zdánlivě) jednoduchý algoritmus zastaví pro libovolný vstup.

* * *

Při dokazování toho, že se algoritmus zaručeně po konečném počtu kroků zastaví, se většinou postupuje tak, že se stanoví nějaká veličina, jejíž hodnota se v průběhu výpočtu snižuje (nebo naopak zvyšuje), přičemž se zdůvodní, že toto snižování nebo zvyšování nemůže pokračovat donekonečna.

Konkrétně to může vypadat tak, že všem (dosažitelným) konfiguracím se přiřadí hodnoty z nějaké vhodně zvolené množiny W, na které definováno nějaké uspořádání $\leq$ takové, že v množině W neexistují vzhledem k tomuto uspořádání žádné nekonečné (ostře) klesající sekvence. Tím, že neexistují žádné nekonečné klesající sekvence je myšleno, že neexistuje žádná nekonečná posloupnost

$$a_0, a_1, a_2, a_3, \ldots,$$

prvků množiny W, kde by pro všechna $i \geq 0$ platilo $a_i > a_{i+1}$, tj.

$$a_0 > a_1 > a_2 > a_3 > \ldots.$$

Příkladem takové množiny je třeba množina přirozených čísel $\mathbb{N}$ s běžným uspořádáním podle velikosti.

Naopak příklady množin, ve kterých mohou existovat nekonečné klesající posloupnosti, jsou třeba celá čísla nebo kladná racionální čísla s běžným uspořádáním podle velikost. V celých číslech existuje například nekonečná klesající posloupnost

$$0, -1, -2, -3, -4, -5, \ldots.$$

V kladných racionálních číslech zase existuje třeba posloupnost

$$1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \frac{1}{32}, \ldots,$$

kde i-tý prvek posloupnosti má hodnotu $\frac{1}{2^i}$. Takovéto množiny tedy nejsou použitelné pro níže popsaný typ důkazu toho, že se algoritmus zastaví.

Předpokládejme tedy, že jsme zvolili nějako množinu W, ve které neexistují nekonečné klesající posloupnosti, a nějaký způsob přiřazení hodnot z této množiny jednotlivým konfiguracím. Označme hodnotu z množiny W přiřazenou konfiguraci α jako $f(\alpha)$. Nyní je třeba ukázat, že pro každou instrukci I a pro každé dvě konfigurace α a α' takové, že

$$\alpha \xrightarrow{I} \alpha'$$

platí $f(\alpha) > f(\alpha')$.

Pokud tedy máme libovolný výpočet tvořený posloupností konfigurací α_0, α_1, α_2, …, bude pro tyto konfigurace platit

$$f(\alpha_0) > f(\alpha_1) > f(\alpha_2) > f(\alpha_3) > \ldots$$

a hodnoty přiřazené jednotlivým konfiguracím tvoří (ostře) klesající posloupnost. Protože v množině W nemůže existovat nekonečná klesající posloupnost, musí být tato posloupnost konečná, což znamená, že příslušný výpočet musí být konečný.

Poznamenejme, že někdy se nepožaduje, že hodnota musí ostře klesnout s každou instrukcí, ale že může zůstat při provedení některých instrukcí stejná. Pokud se ale nějaké instrukce vykonávají v cyklu, je nutné, aby se hodnota s každým průchodem cyklem snižovala.

* * *

Jako příklad množiny W, ve které neexistují nekonečné klesající sekvence, jsme uvedli množinu přirozených čísel $\mathbb{N}$. Přiřazování přirozených čísel konfiguracím tak, aby jejich hodnota s každým krokem klesala, může být dost komplikované.

O něco jednodušší může být pro tento účel použít množinu (konečných) vektorů přirozených čísel, na kterých je definováno ***lexikografické uspořádání***. Přiřazované vektory nemusí být všechny stejně dlouhé (tj. nemusí mít stejný počet položek). Příklady takových vektorů, které můžeme přiřadit konfiguracím jsou třeba $(5, 1, 8, 3)$ nebo $(1, 0)$.

Lexikografické uspořádání je definováno následovně. Vektor $(a_1, a_2, \ldots, a_m)$ je menší než vektor $(b_1, b_2, \ldots, b_n)$, jestliže

- existuje i takové, že $1 \leq i \leq m$ a $i \leq n$, kde $a_i < b_i$ a pro všechna j taková, že $1 \leq j < i$, platí $a_j = b_j$, nebo
- $m < n$ a pro všechna j taková, že $1 \leq j \leq m$, je $a_j = b_j$.

Jinými slovy, pro dvojici $(a_1, a_2, \ldots, a_m)$ a $(b_1, b_2, \ldots, b_n)$ se najde první pozice zleva, na které se hodnoty obou vektorů liší, a porovnají se hodnoty na této pozici. Pokud žádná taková pozice neexistuje, menší je ten vektor, který má méně prvků. Platí tedy například nerovnosti $(5, 1, 3, 6, 4) < (5, 1, 4, 1)$ a $(4, 1, 1) < (4, 1, 1, 3)$.

To, že v množině takovýchto vektorů nemohou při lexikografickém uspořádání existovat nekonečné klesající sekvence, zde nebudeme podrobně dokazovat a budeme to brát jako fakt.

Všimněme si ještě následující zajímavé vlastnosti, kterou se vektory přirozených čísel s lexikálním uspořádáním liší od přirozených čísel. U přirozených čísel, jestliže známe první prvek klesající posloupnosti, je hodnotou tohoto prvku omezena maximální délka této posloupnosti. Například, pokud má první prvek hodnotu 10, nemůže mít tato klesající posloupnost více než 11 prvků $(10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0)$. Oproti tomu, pokud známe hodnotu počátečního vektoru, není tím maximální délka posloupnosti nijak omezena. Pokud máme například vektor $(1, 0)$, může za ním následovat v klesající posloupnosti miliarda dalších vektorů $(0,\ 999\,999\,999)$, $(0,\ 999\,999\,998)$, $(0,\ 999\,999\,997)$, $\ldots (0,\ 1)$, $(0,\ 0)$, a stejně tak za ním může následovat jakýkoliv větší počet vektorů.

* * *

Uveďme nyní příklad důkazu toho, že se Algoritmus 1 popsaný grafem řídícího toku na Obrázku 2.4 pro jakýkoliv vstup zastaví po konečném počtu kroků.

Můžeme si všimnout, že v Algoritmu 1 se v průběhu výpočtu hodnota proměnné i zvětšuje a postupně se blíží hodnotě proměnné n. To znamená s každým průchodem cyklem se postupně snižuje hodnota $n - i$ o jedna a provádění cyklu skončí v okamžiku, kdy tato hodnota klesne na nulu. Následující postup je v podstatě jen formálnější a podrobnější rozpracování této myšlenky.

Dosažitelným konfiguracím přiřadíme jako hodnoty vektory přirozených čísel, na kterých definováno lexikografické uspořádání popsané výše. Hodnotu přiřazenou konfiguraci α budeme označovat $f(\alpha)$.

Na základě řídícího stavu v konfiguraci α bude této konfiguraci přiřazena odpovídající hodnota $f(\alpha)$ následujícím způsobem:

- Stav 0: $f(\alpha) = (4)$
- Stav 1: $f(\alpha) = (3)$
- Stav 2: $f(\alpha) = (2, n - i, 3)$
- Stav 3: $f(\alpha) = (2, n - i, 2)$
- Stav 4: $f(\alpha) = (2, n - i, 1)$

- Stav 5: $f(\alpha) = (2, n - i, 0)$
- Stav 6: $f(\alpha) = (1)$
- Stav 7: $f(\alpha) = (0)$

Hodnoty n a i zde udávají hodnoty těchto proměnných v dané konfiguraci α. Například konfiguraci

$$(2,\ \langle A\colon [3,8,1,3,6],\quad n\colon 5,\quad i\colon 1,\quad k\colon 0,\quad \textit{result}\colon ?\rangle)$$

tak bude přiřazen vektor $(2,4,3)$ (protože v této konfiguraci je $n - i = 5 - 1 = 4$).

Díky tomu, že víme, že ve stavech 2, 3, 4 a 5 platí invariant $i \leq n$ (ve stavech 3, 4 a 5 dokonce $i < n$), je zřejmé, že ve výše popsaném přiřazení jsou všem dosažitelným konfiguracím přiřazeny vektory tvořené přirozenými čísly (tj. žádné dosažitelné konfiguraci není přiřazen vektor, který by měl některou položku zápornou).

Snadno se ověří, že s každou provedenou instrukcí se přiřazený vektor snižuje. U instrukcí I_0, I_1, I_3 a I_8 se snižuje první položka vektoru. U instrukcí I_2, I_4, I_5 a I_6 se snižuje třetí položka vektoru, zatímco první a druhá položka se nemění. (První položka má při těchto instrukcích stále hodnotu 2 a druhá $n - i$, přičemž hodnoty proměnných n a i se při žádné z těchto instrukcí nemění.)

Jediný komplikovanější případ je instrukce I_7. Tato instrukce převádí stroj z konfigurace s hodnotou $(2, n - i, 0)$ do konfigurace s hodnotou $(2, n - i', 3)$, kde $i' = i + 1$. Podobně jako v případě dokazování invariantů jsou zde pro přehlednost použity proměnné i a i' pro označení hodnoty proměnné i před a po příslušném přiřazení (instrukce I_7 mění hodnotu proměnné i). Vzhledem k tomu, že $i' = i + 1$, je $n - i' = n - (i + 1) = (n - i) - 1 < n - i$. Při provedení instrukce I_7 tedy zůstává stejná první položka vektoru, zatímco druhá klesne.

Tím je tedy hotov důkaz toho, že se Algoritmus 1 vždy po konečném počtu kroků zastaví.

* * *

Poznamenejme ještě, že v praxi mohou mít smysl i algoritmy, které se nikdy nezastaví. Jeden typ takových algoritmů jsou algoritmy, které produkují nějaký nekonečný výstup, například všechny prvky nějaké nekonečné posloupnosti. Jako příklady takových algoritmů si můžeme představit třeba algoritmus, který by na výstup vypisoval posloupnost všech prvočísel, nebo třeba algoritmus, který by postupně vypisoval všechny číslice čísla π, tj. začátek výstupu tohoto algoritmu by vypadal takto:

$$3.14159265358979323846\ldots$$

Jiným příkladem takových algoritmů jsou algoritmy, které zpracovávají nějaké požadavky, pro které generují nějaké odpovědi. Příkladem takových systémů jsou třeba operační systém nebo webový server. Můžeme si představit, že takové algoritmy zpracovávají potenciálně nekonečnou frontu požadavků.

U těchto nekonečně běžících algoritmů nedává smysl dokazovat, že se zastaví po konečném počtu kroků, protože to po nich právě nechceme. Má smysl ale například dokazovat, že při vypisování prvků nekonečné posloupnosti bude každý jednotlivý prvek po nějakém konečném počtu kroků vypsán. Například může mít smysl dokazovat, že při vypisování číslic čísla π se nestane, že by se vypsalo prvních 500 číslic a program by pak běžel donekonečna bez toho, aby cokoliv dalšího vypsal. Podobně má smysl dokazovat u systému, který vyřizuje požadavky, že každý požadavek bude po konečném počtu kroků vyřízen, apod.

V tomto textu se ovšem takovými nekonečně běžícími algoritmy příliš podrobně zabývat nebudeme.

Kapitola 4

Výpočetní složitost algoritmů

Počítače pracují rychle, ale ne nekonečně rychle. Provedení každé jednotlivé operace během výpočtu trvá nějakou nenulovou dobu. Z toho důvodu v praxi často není až tak podstatné, jestli se algoritmus pro každý vstup zastaví po konečném počtu kroků, ale zda jsme schopni celkovou dobu výpočtu nějak omezit. Z praktického hlediska se algoritmus, u kterého jsme schopni dokázat, že se sice vždy po konečném kroků zastaví, ale který pro vstupní data, pro která ho chceme použít, bude běžet miliardu let, nijak zvlášť neliší od algoritmu, který se nezastaví nikdy. Ani v jednom případě se spočítání výsledku nedočkáme.

Většinou tedy chceme, aby algoritmus řešící daný problém byl co nejrychlejší. Jak moc úsilí věnovat hledání co nejefektivnějšího algoritmu záleží samozřejmě na tom, jak velké množství dat bychom chtěli tímto algoritmem zpracovávat. Pokud například potřebujeme setřídit pole tvořené deseti prvky, dá se použít v podstatě libovolný třídící algoritmus a pole bude tímto algoritmem setříděno prakticky okamžitě a mezi jednotlivými algoritmy budou jen minimální rozdíly v době jejich běhu. Naprosto jiná bude situace, jestliže budeme chtít třídit pole tvořené milionem nebo třeba miliardou prvků. V takovém případě mohou být mezi jednotlivými algoritmy obrovské rozdíly v tom, jak dlouho výpočet trvá. To, co jeden algoritmus udělá během zlomku sekundy nebo maximálně v řádu sekund, může jinému algoritmu trvat třeba několik minut nebo i desítek minut.

Dalším omezujícím faktorem může být množství paměti, která je pro provedení výpočtu k dispozici. Algoritmus by například byl schopen spočítat výsledek pro daná vstupní data v nějakém „rozumném“ čase, ale jeho výpočet může skončit neúspěšně proto, že na počítači, na kterém tento výpočet poběží, není k dispozici dostatek paměti k provedení všech operací, které je třeba během výpočtu udělat.

Někdy se dá množství paměti, která je k dispozici, zvětšit tím, že si algoritmus bude například odkládat některé údaje na disk. Vzhledem k tomu, že přístup k datům na disku je řádově pomalejší než přístup k datům v paměti RAM, může být pak takový algoritmus mnohem pomalejší, než kdyby ukládal veškerá data v paměti RAM.

U algoritmů se běžně provádí analýza, při které se zkoumají časové a paměťové nároky algoritmů. Zkoumá se, jak dlouho budou výpočty algoritmu trvat a kolik při nich bude bude potřeba paměti.

Je asi zřejmé, že dobu výpočtu algoritmu nebo množství použité paměti není většinou možné omezit nějakým jediným číslem. Obecně závisí doba běhu i množství použité paměti na konkrétním vstupu, který má algoritmus zpracovávat. Jinou dobu bude trvat setřídění pole 10 prvků a jinou dobu pole tvořené milionem prvků. Pokud nedáme na velikost vstupních dat nějaké omezení, tak se asi ve většině případů nedá o algoritmu říct, že na daném počítači poběží pro jakýkoliv vstup maximálně 10 sekund nebo že při výpočtu bude potřebovat maximálně 10 MB paměti.

To, jak konkrétně závisí doba výpočtu na daných vstupních datech, například to, jak rychle roste doba výpočtu s tím, jak se zvětšuje množství vstupních dat, se označuje jako ***časová složitost*** algoritmu. Podobně to, jak konkrétně závisí na vstupních datech množství paměti,

které bude algoritmus při výpočtu potřebovat, se označuje jako ***paměťová*** nebo též ***prostorová složitost*** algoritmu.

Souhrnně se časová a paměťová složitost algoritmu označují jako ***výpočetní složitost algoritmu*** nebo krátce jako ***složitost algoritmu***. Ve skutečnosti tento pojem zahrnuje i další typy složitostí, nejen časovou a paměťovou složitost. Například u distribuovaných algoritmů, které běží paralelně na více počítačích, které spolu komunikují pomocí posílaní zpráv, se může zkoumat tzv. ***komunikační složitost***, kdy se analyzuje počet posílaných zpráv či množství dat, které si musí jednotlivé počítače mezi sebou vyměnit. V tomto textu se jinými typy výpočetní složitosti než časovou a paměťovou složitostí zabývat nebudeme. Časová a paměťová složitost jsou dva nejdůležitější typy složitosti, a proto se zaměříme pouze na ně.

4.1 Velikost vstupu

Jak už bylo řečeno, jak doba výpočtu algoritmu, tak množství použité paměti, většinou závisí na konkrétních vstupních datech. Asi je zřejmé, že se doba výpočtu či množství použité paměti budou lišit v závislosti na množství vstupních dat (vezměme si třeba třídění 10 prvků versus třídění milionu prvků).

Doba výpočtu či množství použité paměti se však mohou lišit i pro vstupy, kde je vstupních dat přibližně stejné množství. Například mnohé třídící algoritmy setřídí dané pole mnohem rychleji, pokud jsou prvky v tomto poli už setříděny, než v případě, kdy jsou tyto prvky nějak náhodně promíchány nebo kdy jsou setříděny v opačném pořadí, než jak je chceme setřídit. Pro dvě taková pole, která mají stejnou velikost (například obě mají $1\,000\,000$ prvků), tak může výpočet trvat velmi rozdílnou dobu v závislosti na tom, jak jsou hodnoty v těchto polích uspořádány.

Přesné určení doby výpočtu či množství použité paměti v závislosti na konkrétních vstupech je i pro relativně jednoduché algoritmy většinou velmi složité a pracné. Dobu běhu algoritmu i množství použité paměti totiž ovlivňuje příliš mnoho různých faktorů, které je třeba vzít při takové analýze v úvahu. Výpočetní složitost algoritmu se proto prakticky nikdy neanalyzuje zcela přesně. Místo toho se při analýze složitosti používá celá řada úvah, které umožní tuto analýzu podstatným způsobem zjednodušit za cenu, že odvozené výsledky této analýzy jsou méně přesné, zato je ale taková analýza prakticky zvládnutelná.

Doba běhu algoritmu a množství použité paměti závisí v mnoha ohledech na detailech stroje, na kterém algoritmus poběží. Jedním z cílů zmíněných zjednodušujících úvah je umožnit odvodit výsledky týkající se výpočetní složitosti, které jsou do značné míry na těchto detailech nezávislé.

První zjednodušení spočívá v tom, že většinou se neanalyzuje závislost doby běhu nebo množství použité paměti na konkrétních jednotlivých vstupech, ale závislost těchto hodnot na ***velikosti vstupu***.

Velikost vstupu je typicky jedno přirozené číslo nebo několik málo přirozených čísel, udávající, jak je daný vstup velký — čím vyšší hodnota (nebo hodnoty), tím je vstup považován za větší.

Pokud máme dán nějaký algoritmický problém a nějaký algoritmus, které tento problém řeší, a chceme analyzovat jeho výpočetní složitost, není obecně nějak apriori dáno, co přesně je velikost vstupu pro tento konkrétní algoritmus nebo problém. Co budeme považovat za velikost vstupu, si můžeme zvolit. Většinou se však pro daný typ problému a daný typ vstupních dat nabízí nějaká přirozená možnost, co konkrétně za velikost vstupu zvolit.

Uveďme několik typických příkladů, co konkrétně se dá u různých typů problémů považovat za velikost vstupu:

- Pokud je vstupem sekvence nějakých hodnot, pole prvků apod., je většinou nejpřirozenější zvolit jako velikost vstupu počet prvků v této sekvenci nebo poli.

  Typickým příkladem je třeba problém třídění, problém hledání maximálního prvku v poli a podobně. Pokud je tedy vstupem sekvence nebo pole o n prvcích, je velikost daného vstupu toto číslo n (např. pokud je vstupem pole o 100 prvcích, je velikost tohoto vstupu 100).

- Pokud je vstupem řetězec znaků, je přirozené zvolit jako velikost vstupu počet těchto znaků. (Jedná se v podstatě o stejný případ, jako v předchozím bodě.)

- Pokud jsou vstupem dva řetězce, z nichž jeden je například (dlouhý) text, který se bude prohledávat, a druhý je (kratší) řetězec, jehož výskyty se budou v daném dlouhém textu vyhledávat, je většinou vhodné velikost tohoto vstupu reprezentovat pomocí dvojice čísel — jedno z těchto čísel je počet znaků v prohledávaném textu a druhé počet znaků v hledaném vzorku. Důvod je ten, že doba provádění některých částí algoritmů pro hledání zadaného vzorku v textu i množství paměti, kterou algoritmus pro toto hledání potřebuje, závisí jen na velikosti hledaného vzorku, ale nikoliv na velikosti prohledávaného textu (který se jen sekvenčně načítá znak po znaku bez toho, že by bylo třeba ukládat již jednou zpracované znaky).

  Někdy může mít smysl uvažovat kromě dvou výše uvedených parametrů udávajících velikost vstupu ještě třetí parametr, a to velikost abecedy, ze které jsou znaky, které se mohou v zadaných řetězcích vyskytnout.

- Pokud je vstupem nějaký libovolný počet řetězců, je možné brát jako velikost vstupu celkový počet znaků v těchto řetězcích.

  Podle situace může být vhodné uvažovat jako další parametr udávající velikost vstupu kromě celkového počtu znaků i počet těchto řetězců, například abychom mohli od sebe odlišit případ, kdy je vstupem velké množství krátkých řetězců, a případ, kdy je vstupem malé množství dlouhých řetězců.

- U grafových problémů, kde vstupem je graf (a případně nějaké další informace), je většinou velikost vstupu reprezentována jako dvojice čísel n a m, kde n je počet vrcholů grafu a m počet hran tohoto grafu.

- Vezměme si nyní problémy, kde vstupem je jedno číslo, dvojice čísel apod. Příkladem takových problémů jsou třeba problém, zda dané číslo je prvočíslo, kde vstupem je jediné přirozené číslo, nebo problém nalezení největšího společného dělitele dvou čísel, kde vstupem je dvojice přirozených čísel.

  V takovém případě samozřejmě nemá smysl definovat velikost vstupu jako počet čísel na vstupu, protože by velikost byla vždy 1 nebo 2. Doba běhu algoritmů, které řeší takové problémy, navíc závisí na hodnotách čísel na vstupu. V takovém případě se nabízí možnost považovat za velikost vstupu přímo hodnoty těchto čísel (a tím pádem nedělat rozdíl mezi vstupem a jeho velikostí).

  Většinou se však u takových problémů a algoritmů bere jako velikost vstupu ***počet bitů***, které jsou potřeba pro zápis těchto čísel ve dvojkové soustavě (binárně). Toto má smysl zejména tehdy, pokud velikost čísel na vstupu není omezena (např. na 32-bitová čísla), takže čísla na vstupu mohou být libovolně velká.

- Někdy je vstupem (libovolně dlouhá) posloupnost čísel, kde ale hodnoty těchto čísel ovlivňují dobu výpočtu, například pro větší čísla trvá výpočet déle. V takovém případě může být přirozené považovat za velikost vstupu celkový počet bitů ve všech číslech na vstupu. Zejména to platí v případech, kdy tato čísla mohou být libovolně velká.

Jak vyplývá z předchozích příkladů, co konkrétně považovat za velikost vstupu, si můžeme zvolit pro daný problém nebo algoritmus podle potřeby, zejména podle toho, o jaký typ problému se jedná a podle toho, jak podrobnou analýzu složitosti chceme provádět.

Běžně zaužívanou konvencí je, že pokud je velikost vstupu dána jediným číslem, je toto číslo označováno písmenem n nebo N.

Pokud je velikost vstupu reprezentována dvojicí čísel, používají se většinou pro jejich označení písmena n a m (nebo N a M). V takovém případě by mělo být vždy jasně řečeno, co je n a co je m — například, že n je počet vrcholů grafu a m počet jeho hran, nebo třeba, že n je celkový

počet řetězců na vstupu a m součet délek všech těchto řetězců (tj. celkový počet znaků v těchto řetězcích), apod.

4.2 Doba výpočtu algoritmu

Zaměřme se nyní na časovou složitost algoritmů, tj. na zkoumání doby jejich běhu. Řekněme, že máme nějaký algoritmus *Alg* a tento algoritmus řeší nějaký problém P. Například si můžeme představit třeba algoritmus Quicksort, který řeší problém třídění. Symbolem *In* označme množinu všech instancí problému P, tj. množinu všech přípustných vstupů pro algoritmus *Alg*. V případě algoritmus Quicksort a problému třídění to bude množina všech možných polí prvků nějakého typu.

Předpokládejme, že algoritmus *Alg* bude vykonáván strojem $\mathcal{M}$. Jako tento stroj si můžeme představit třeba skutečný počítač, který bude vykonávat program ve strojovém kódu, který vznikl překladem programu napsaného v jazyce C, nebo třeba virtuální stroj vykonávající program napsaný v jazyce Python, nebo třeba nějaký idealizovaný matematický model počítače, apod.

Pokud si nyní vezmeme nějaký libovolný vstup w z množiny *In* (v případě problému třídění třeba pole $[6, 13, 1, 8, 4, 5, 8]$) a necháme stroj $\mathcal{M}$ provádět algoritmus *Alg* pro tento vstup w, bude tento výpočet nějakou dobu trvat. Označme dobu trvání tohoto výpočtu $t(w)$.

Zde si ovšem můžeme položit otázku, v jakých jednotkách vůbec dobu výpočtu vyjadřovat. Tato otázka vypadá možná na první pohled hloupě, ale při troše zamyšlení je vidět, že se ve skutečnosti nabízí několik rozdílných možností, v čem dobu výpočtu vyjadřovat. Ani jedna z těchto možností nevypadá nějak výrazně lepší než ostatní. Spíše se dá říci, že se každý hodí k něčemu jinému.

Zde je několik přirozených možností, v čem počítat dobu běhu algoritmu:

- ***v sekundách*** — Tato možnost asi napadne člověka jako první (máme zde na mysli i jakékoliv jiné časové jednotky jako třeba milisekundy, minuty, hodiny, apod.). Skutečná doba běhu v sekundách je tím, co nás z praktického hlediska nakonec zajímá, a proč vlastně analýzu časové složitosti děláme. Skutečná doba běhu se dá pro konkrétní implementaci algoritmu a konkrétní vstupní data také velice snadno změřit.

  Na druhou stranu má vyjadřování doby běhu algoritmu v sekundách celou řadu nevýhod. Hlavní problém spočívá v tom, že tato doba nezávisí jen na daném algoritmu, ale je dána nespočtem dalších faktorů, které tuto dobu ovlivňují, především použitým hardware a jeho parametry, ale i překladačem a jeho nastavením, operačním systémem, apod. Někdy i drobné rozdíly v tom, jak je algoritmus konkrétně naprogramován, mohou způsobit měřitelné rozdíly v době běhu algoritmu. Dokonce i když spustíme tentýž program se stejnými vstupními daty na tomtéž počítači několikrát za sebou, tak se většinou naměřené hodnoty budou drobně lišit.

  Porovnávat dobu běhu jednoho algoritmu (resp. jedné jeho konkrétní implementace) na jednom počítači s dobou běhu jiného algoritmu na jiném počítači z hlediska délky výpočtu v sekundách je tak většinou nesmyslné, neboť v tomto případě jde o porovnávání jablek s hruškami.

- ***v počtu provedených kroků*** — Zde je ovšem potřeba specifikovat, co je považováno za jeden krok.

  Pokud budeme uvažovat programy běžící na skutečném počítači (a ne třeba nějaký idealizovaný matematický model počítače), dají se počítat kroky na mnoha různých úrovních abstrakce. Za kroky algoritmu můžeme považovat třeba:

  - Příkazy vyššího programovacího jazyka — tj. provedení jednoho příkazu (jako třeba provedení přiřazení $a[i] := a[j + 1] * 2$) je považováno za jeden krok.
  - Instrukce strojového kódu nebo bytekódu — jedna instrukce vyššího programovacího jazyka může vyžadovat provedení celé řady instrukcí na nižší úrovni.

- Takty procesoru — provedení jedné instrukce strojového kódu procesorem je rozloženo na více jednodušších operací, kde doba trvání každé této jednodušších operace je dána jedním tikem hodin (tj. taktovací frekvencí procesoru). Provedení různých instrukcí strojového kódu může trvat různý počet taktů.

Jak se ukáže dále, při běžně prováděné analýze časové složitosti algoritmů, kdy se celá řada věcí stejně zanedbává, není mezi těmito různými možnostmi v praxi příliš velký rozdíl. Pokud bychom ale prováděli hodně podrobnou analýzu doby výpočtu a hodnoty bychom chtěli vyjadřovat v číslech, bylo by třeba určit v jakých jednotkách dobu výpočtu počítáme.

* * *

Řekněme, že pro daný algoritmus *Alg* prováděný strojem $\mathcal{M}$ jsme specifikovali jednotky, ve kterých budeme dobu výpočtu počítat, a pro každou instrukci, která se při výpočtech tohoto algoritmu bude provádět, jsme specifikovali, jak dlouho trvá provedení této instrukce na stroji $\mathcal{M}$.

Pro jednoduchost předpokládejme, že dobu provedení každé jednotlivé instrukce můžeme vyjádřit jako (kladné) reálné číslo a že při opakovaném provádění této instrukce během výpočtu je doba provedení této instrukce pokaždé stejná. (Doba provedení různých instrukcí může být různá.) Navíc pro jednoduchost předpokládejme, že instrukce se provádějí sekvenčně jedna po druhé, tj. v jednom okamžiku se vždy provádí jediná instrukce a další instrukce se začne provádět až po jejím skončení.

Pro konkrétnost si vezměme třeba Algoritmus 1 popsaný grafem řídícího toku na Obrázku 2.4. Každé hraně tohoto grafu přiřaďme kladné reálné číslo udávající, jak dlouho trvá provedení této instrukce (není teď podstatné jestli tyto hodnoty udávají tuto dobu v nanosekundách, taktech procesoru, počtu instrukcí strojového kódu nebo v něčem jiném). Označme dobu provedení instrukce I_0 symbolem c_0, dobu provedení instrukce I_1 označme c_1, atd. Doby trvání instrukcí $I_0, I_1, \ldots, I_8$ jsou tedy $c_0, c_1, \ldots c_8$, přičemž $c_0, c_1, \ldots c_8$ jsou kladná reálná čísla.

V dalším textu budeme množinu všech kladných reálných čísel označovat $\mathbb{R}^+$. Hodnoty c_0, c_1, $\ldots$, c_8 jsou tedy prvky množiny $\mathbb{R}^+$.

Pokud si nyní vezmeme nějaký konkrétní vstup $w \in \mathit{In}$ pro algoritmus *Alg* (pro Algoritmus 1 například vstupní instanci $([3, 8, 4, 5, 2], 5)$), provede tento algoritmus pro vstup w nějaký konkrétní výpočet, tj. nějakou konkrétní posloupnost provedených instrukcí. Celková doba provedení tohoto výpočtu $t(w)$ je dána jako součet dob provedení všech jednotlivých instrukcí, provedených během tohoto výpočtu.

Hodnota $t(w)$ je tedy kladné reálné číslo. Pokud bychom například pro Algoritmus 1 stanovili, že $c_0 = 4$, $c_1 = 4$, $c_2 = 10$, $c_3 = 12$, $c_4 = 14$, $c_5 = 12$, $c_6 = 5$, $c_7 = 6$ a $c_8 = 5$, mohli bychom třeba odsimulovat výpočet tohoto algoritmu pro vstup $w = ([3, 8, 4, 5, 2], 5)$ a určit konkrétní hodnotu $t(w)$. My to zde ale teď tímto pracným způsobem dělat nebudeme. (Později tuto hodnotu spočítáme jednodušeji.)

* * *

Když budeme v dalším popisu mluvit o hodnotách $c_0, c_1, \ldots, c_8$ a ne o konkrétních číslech, můžeme tímto způsobem alespoň částečně abstrahovat od konkrétních vlastností stroje $\mathcal{M}$. Pro různé stroje budou tyto hodnoty různé, ale my se zde nemusíme teď zajímat o to, jak konkrétně určit tyto hodnoty pro daný typ stroje, ani se nemusíme zabývat detailním popisem konkrétního stroje.

Výše popsaný případ, kdy jsou jednotlivým instrukcím přiřazeny konstantní hodnoty, je v některých ohledech poněkud zjednodušený oproti tomu, jak to může „ve skutečnosti“ být, protože obecně doba provedení jedné instrukce nemusí být konstantní (záleží samozřejmě na typu stroje a na tom, o jakou instrukci se jedná).

Například některé instrukce mohou trvat různou dobu v závislosti na tom, jak velké jsou hodnoty operandů, se kterými pracují. Jistě si lze představit stroj, kde například sečtení dvou velkých čísel bude trvat delší dobu než sečtení dvou malých čísel.

V moderních počítačích jsou velké rozdíly mezi rychlostí, s jakou pracuje procesor, a s jakou pracuje paměť RAM. Aby přístupy do paměti nebrzdily výpočet, bývá mezi procesorem a pamětí RAM celá hierarchie pamětí cache, které mají výrazně menší kapacitu než paměť RAM, ale jsou podstatně rychlejší. V praxi je tak velký rozdíl v době přístupu k hodnotě, která je v paměti cache, a k hodnotě, která v paměti cache není. Pokud se například často pracuje s nějakou proměnnou, první přečtení hodnoty této proměnné, než se její hodnota dostane do paměti cache, může trvat mnohonásobně déle, než další přístupy k této proměnné.

Ve skutečných počítačích také není úplně pravda, že se instrukce provádějí sekvenčně v tom smyslu, že se další instrukce začne provádět až po dokončení předchozí instrukce. V moderních mikroprocesorech se používá celá řada různých technik pro urychlení práce procesoru. Procesor se skládá z mnoha bloků, které pracují do značné míry paralelně. Pokud nějaká takový blok procesoru dokončí svou část zpracování instrukce, kterou právě provádí, nečeká na dokončení této instrukce ve zbytku procesoru, ale pokračuje prováděním další instrukce. Provádění po sobě jdoucích instrukcí se tak různě prolíná a určení přesné doby provedení určité sekvence instrukcí nemusí být vůbec snadné. Tato doba může být o dost menší než prostý součet dob zpracování každé jednotlivé instrukce.

Například provedení každé jednotlivé instrukce samo o sobě může trvat jednotky až desítky taktů. Díky současnému zpracovávání více instrukcí najednou však není neobvyklé, že při provádění dlouhé sekvence instrukcí vychází v průměru doba zpracování jedné instrukce na 1 až 2 takty (když podělíme celkový počet taktů, který trvalo zpracování celé sekvence, počtem instrukcí).

Těmito detaily, které souvisí spíše s architekturou skutečných počítačů, se zde nebudeme podrobněji zabývat. Pro naše účely bude postačovat výše uvedená jednoduchá představa. Měli bychom si však být těchto věcí vědomi a v případech, kdy bychom prováděli tak podrobnou analýzu, kde by hrály tyto faktory nějakou roli, bychom je museli vzít v úvahu.

4.3 Časová složitost algoritmu

Nyní se postupně dostáváme k tomu, co to přesně je časová složitost algoritmu.

Řekněme, že máme pro daný algoritmus *Alg* a daný stroj $\mathcal{M}$ určeno, jaká je doba výpočtu $t(w)$ algoritmu *Alg* nad vstupem w, pro každý vstup w z množiny všech možných vstupů *In*.

Dále předpokládejme, že jsme zvolili, co budeme považovat za velikost vstupu. Pro jednoduchost se zaměřme na případ, kdy je velikost vstupu vyjádřena jediným číslem. Předpokládáme tedy, že každému vstupu w z množiny *In* je přiřazeno přirozené číslo, udávající velikost vstupu w. Toto číslo budeme označovat zápisem $\mathit{size}(w)$. (Z formálního hlediska je $\mathit{size} : \mathit{In} \to \mathbb{N}$ funkce, která vstupům přiřazuje jejich velikost.)

Ve skutečnosti existuje více různých druhů časové složitosti algoritmu. Nejpoužívanějším druhem časové složitosti je ***časová složitost v nejhorším případě***. Pokud se někde mluví o časové složitosti algoritmu a není tam explicitně zdůrazněno, o který konkrétní typ složitosti se jedná, prakticky vždy je tím myšlena časová složitost v nejhorším případě.

Kromě složitosti v nejhorším případě se někdy analyzuje také ***časová složitost v průměrném případě***. Analýza tohoto typu složitosti je však často o dost komplikovanější než analýza složitosti v nejhorším případě, a proto se zde zaměříme především na složitost v nejhorším případě.

Definice 4.1 ***Časová složitost algoritmu*** *Alg* ***v nejhorším případě*** je funkce $T : \mathbb{N} \to \mathbb{R}^+$, která každému přirozenému číslu n přiřazuje maximální dobu výpočtu algoritmu *Alg* nad vstupem velikosti n.

Pro každé $n \in \mathbb{N}$ tedy platí:

- Pro každý vstup $w \in \mathit{In}$ takový, že $\mathit{size}(w) = n$, je $t(w) \leq T(n)$.
- Existuje vstup $w \in \mathit{In}$ takový, že $\mathit{size}(w) = n$ a $t(w) = T(n)$.

Z výše uvedené definice vidíme, že časová složitost algoritmu je funkce, jejíž přesné hodnoty závistí nejen na daném algoritmu *Alg*, ale také na následujících věcech:

- definici stroje $\mathcal{M}$, na kterém algoritmus *Alg* běží,
- definici doby výpočtu $t(w)$ algoritmu *Alg* na stroji $\mathcal{M}$ pro vstup $w \in In$,
- definici velikosti vstupu (tj. definici funkce *size*).

Neformálně si určení hodnot funkce T můžeme představit tak, že hodnotu $T(n)$ určíme tím způsobem, že vezmeme všechny vstupy velikosti n, a z nich vybereme ten, pro který trvá výpočet nejdéle. Dobu trvání tohoto nejdelšího výpočtu přiřadíme $T(n)$. Toto provedeme postupně pro všechna n, tj. pro $n = 0, 1, 2, 3, \dots$ (Toto je samozřejmě jen myšlenkový postup, který není možné ve skutečnosti provést, protože bychom museli projít nekonečně mnoho hodnot.)

* * *

V praxi může být přesné určení časové složitosti daného algoritmu (tj. přesných hodnot výše popsané funkce T) velmi obtížné a pracné a to i pro poměrně jednoduché algoritmy a za přijetí různých zjednodušujících předpokladů. Proto se většinou provádějí jen přibližné odhady toho, jak funkce T vypadá. Především se zkoumá, jak zhruba rostou hodnoty této funkce s tím, jak se zvětšuje velikost vstupu n, tj. jak zhruba závisí hodnota $T(n)$ na hodnotě n. Pro vyjádření této závislosti se používá tzv. ***asymptotická notace***, kterou se budeme zabývat později.

Použití asymptotické notace analýzu časové složitosti výrazně zjednodušuje. Zároveň však při vyjádření odhadů funkce T pomocí asymptotické notace nelze prakticky nic říci o konkrétních hodnotách funkce T, protože asymptotická notace se týká rychlosti růstu funkcí, nikoli jejich konkrétních hodnot.

Pro ilustraci si nyní ukažme analýzu časové složitosti Algoritmu 1 ***bez*** použití asymptotické notace. Tento algoritmus je natolik jednoduchý, že je pro něj zvládnutelné i přesné určení funkce T, která vyjadřuje jeho časovou složitost v nejhorším případě. Zdůrazněme však, že tímto způsobem se analýza časové složitosti v praxi v naprosté většině případů nedělá. Zde tento příklad slouží jen jako ukázka toho, jak moc použití asymptotické notace analýzu zjednodušuje a co vše je při použití asymptotické notace zanedbáno, od čeho můžeme abstrahovat, atd.

Předpokládejme tedy, že máme Algoritmus 1 popsaný grafem řídícího toku na Obrázku 2.4 a že jednotlivým instrukcím $I_0, I_1, \dots, I_8$ jsou přiřazeny doby jejich trvání $c_0, c_1, \dots, c_8$ vyjádřené jako kladná reálná čísla.

Pro vstup (A, n), kde A je pole hodnot (pro jednoduchost budeme uvažovat pole celých čísel) a n počet prvků tohoto pole, budeme jako velikost tohoto vstupu brát číslo n. Například velikost vstupu $([-3, 1, 9, 13, -1, -1], 6)$ je tedy 6.

Uvažujme tedy nějaký libovolný vstup (A, n) velikosti n, kde $n \geq 1$, a zkusme určit co nejpřesněji dobu běhu Algoritmu 1 pro tento vstup.

Pokud uvažujeme o výpočtu Algoritmu 1 nad vstupem (A, n) jako o posloupnosti instrukcí, kde navíc předpokládáme, že doba trvání každé jednotlivé instrukce je dána nějakou konstantou (a která se tedy v průběhu výpočtu nemění), asi nás napadne, že stačí pro každou jednotlivou instrukci určit, kolikrát bude během tohoto výpočtu provedena. Pokud instrukce I_0 bude provedena m_0 krát, instrukce I_1 bude provedena m_1 krát, a tak dále, až instrukce I_8 bude provedena m_8 krát, tak celkovou dobu výpočtu můžeme určit jako

$$c_0 m_0 + c_1 m_1 + \cdots + c_8 m_8 \,. \qquad (*)$$

Prozkoumáním algoritmu (a s využitím toho, co už o něm víme na základě analýzy jeho korektnosti) snadno zjistíme následující:

- Instrukce I_0, I_1, I_3 a I_8 se všechny provedou během výpočtu jedenkrát bez ohledu na to, jak vypadá vstup.

- Cyklus tvořený instrukcemi I_2, I_4, I_5, I_6 a I_7 se provede celkem $n-1$ krát, protože při průchodech tímto cyklem proměnná i nabývá postupně hodnot $1, 2, \ldots, n-1$ (hodnota proměnné i se s každým průchodem cyklem zvětšuje o 1 a z cyklu se vyskočí, jakmile dosáhne hodnoty n).
- Instrukce I_2 a I_7 se při každém průchodu cyklem provedou vždy jedenkrát. Celkem se tedy každá z těchto instrukcí provede $n-1$ krát.
- Při každém průchodu cyklem se buď provede instrukce I_4 (a neprovedou instrukce I_5 a I_6) nebo instrukce I_5 a I_6 (v tom případě se neprovede I_4).
  To, jestli se v daném průchodu cyklem provede instrukce I_4 nebo se provedou instrukce I_5 a I_6, závisí na tom, jak dopadne v tomto průchodu test podmínky v řídícím stavu 3. Pokud platí $A[i] \leq A[k]$ provede se instrukce I_4, jinak (tj. pokud $A[i] > A[k]$) se provedou instrukce I_5 a I_6.
  Označme ℓ počet těch průchodů cyklem, kdy platí $A[i] > A[k]$. Zjevně je $0 \leq \ell < n$. Instrukce I_5 a I_6 se provedou celkem ℓ krát a instrukce I_4 celkem $n-1-\ell$ krát.

Při použití výše zavedeného značení jsou tedy počty provedení instrukcí $I_0, I_1, \ldots, I_8$ pro vstup (A, n) následující:

$$
\begin{array}{lll}
m_0 = 1 & m_3 = 1 & m_6 = \ell \\
m_1 = 1 & m_4 = n-1-\ell & m_7 = n-1 \\
m_2 = n-1 & m_5 = \ell & m_8 = 1
\end{array}
$$

Po dosazení těchto hodnot do výrazu $(*)$ můžeme tedy dobu výpočtu pro vstup (A, n) vyjádřit jako

$$d_1 \ + \ d_2 \cdot (n-1) \ + \ d_3 \cdot (n-1-\ell) \ + \ d_4 \cdot \ell,$$

kde

$$
\begin{array}{ll}
d_1 = c_0 + c_1 + c_3 + c_8 & d_3 = c_4 \\
d_2 = c_2 + c_7 & d_4 = c_5 + c_6
\end{array}
$$

Tento výraz můžeme upravit na

$$(d_2 + d_3) \cdot n \ + \ (d_4 - d_3) \cdot \ell \ + \ (d_1 - d_2 - d_3) \,. \qquad (**)$$

Pomocí vzorce $(**)$ můžeme snadno určit hodnotu $t(w)$, pro každý konkrétní vstup w.

Příklad 4.1 Pokud bychom například pro Algoritmus 1 stanovili, že $c_0 = 4$, $c_1 = 4$, $c_2 = 10$, $c_3 = 12$, $c_4 = 14$, $c_5 = 12$, $c_6 = 5$, $c_7 = 6$ a $c_8 = 5$, bude $d_1 = 25$, $d_2 = 16$, $d_3 = 14$ a $d_4 = 17$, takže doba výpočtu $t(w)$ pro vstup w bude $30n+3\ell-5$ (přičemž hodnoty n a ℓ závisí na vstupu w).

Konkrétně například pro vstup $w = ([3, 8, 4, 5, 2], 5)$ je $n = 5$ a $\ell = 1$, takže doba výpočtu pro tento vstup je $30 \cdot 5 + 3 \cdot 1 - 5 = 148$.

Není těžké si rozmyslet, že pro každé $n \geq 1$ a každé $\ell \in \mathbb{N}$, kde $0 \leq \ell < n$, existuje vstup velikosti n, takový, že instrukce I_5 a I_6 se provedou při výpočtu nad tímto vstupem právě ℓ krát. Například stačí vzít vstup, kde

$$A = [0, 1, 2, \ldots, \ell-1, \ell, 0, 0, \ldots, 0] \,.$$

Například pro $n = 10$ můžeme pro $\ell = 6$ vzít vstup, kde $A = [0, 1, 2, 3, 4, 5, 6, 0, 0, 0]$, pro $\ell = 9$ vstup, kde $A = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$, a pro $\ell = 0$ vstup, kde $A = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$.

Pokud nyní chceme určit časovou složitost Algoritmu 1 v nejhorším případě (tj. popsat ji jako funkci $T(n)$), musíme si promyslet, pro který vstup velikosti n bude výpočet trvat nejdéle, tj. pro který vstup velikosti n bude mít výraz $(**)$ největší hodnotu.

Pro jakýkoliv vstup velikosti n budou mít výrazy $(d_2 + d_3) \cdot n$ a $(d_1 - d_2 - d_3)$ stále stejnou hodnotu. Jediné, co se u vstupů velikosti n může lišit, je hodnota výrazu $(d_4 - d_3) \cdot \ell$.

Nyní nastávají dvě možnosti, které odpovídají tomu, zda trvá déle provedení instrukce I_4 nebo provedení dvojice instrukcí I_5 a I_6 (jsou to dvě příkazu **if** a následující dva případy se liší podle toho, která z těchto dvou větví se provádí déle).

V prvém případě je $d_3 \geq d_4$ a tedy $d_4 - d_3 \leq 0$. Vyšší hodnota ℓ tedy znamená kratší dobu běhu. Nejhorší případ (tj. nejdelší doba běhu) nastává pro $\ell = 0$, tj. například pro vstupy tvaru $[0, 0, \ldots, 0]$ nebo třeba $[n, n-1, n-2, \ldots, 2, 1]$, apod.

Ve druhém případě je $d_3 \leq d_4$ a tedy $d_4 - d_3 \geq 0$. Teď naopak vyšší hodnota ℓ znamená delší dobu běhu algoritmu. Nejhorší případ tak nastává pro $\ell = n - 1$, tj. například pro vstupy tvaru $[0, 1, \ldots, n-1]$.

(V případě, kdy obě větve trvají stejně dlouho, tj. když $d_3 = d_4$, je $d_4 - d_3 = 0$. V tomto případě, nemá hodnota ℓ na dobu běhu vliv a výpočet bude trvat stejně dlouho pro všechny vstupy velikosti n.)

Z předchozího vyplývá, že v případě, kdy $d_3 \geq d_4$, má Algoritmus 1 časovou složitost

$$T(n) = (d_2 + d_3) \cdot n + (d_1 - d_2 - d_3)$$

a v případě, kdy $d_3 \leq d_4$, má časovou složitost

$$\begin{aligned} T(n) &= (d_2 + d_3) \cdot n \;+\; (d_4 - d_3) \cdot (n-1) \;+\; (d_1 - d_2 - d_3) \\ &= (d_2 + d_4) \cdot n \;+\; (d_1 - d_2 - d_4) \,. \end{aligned}$$

V obou případech je tedy časová složitost Algoritmu 1 funkce tvaru $T(n) = an + b$, kde a a b jsou konstanty, jejichž přesné hodnoty závisí na délce trvání jednotlivých instrukcí. (Konkretně můžeme tyto konstanty vyjádřit jako $a = d_2 + \max\{d_3, d_4\}$ a $b = d_1 - d_2 - \max\{d_3, d_4\}$).

Jak je vidět z této analýzy, to, který vstup velikosti n je nejhorší, tj. pro který trvá výpočet nejdéle, nezávisí jen na daném problému a algoritmu, ale může to být ovlivněno i různými detaily týkajícími se stroje, na kterém algoritmus běží, detaily implementace tohoto algoritmu a délkou trvání jednotlivých instrukcí.

* * *

Pozornému čtenáři možná neuniklo, že v předchozí analýze není hodnota $T(n)$ řádně definována pro $n = 0$, protože podle definice problému MAX-ELEMENT a toho, jak jsme definovali velikost vstupu pro tento problém, neexistují žádné korektní vstupy velikosti 0. Tato situace není nijak neobvyklá a nepředstavuje žádný zvláštní problém.

U mnoha problémů velikost vstupů definovaná nějakým přirozeným způsobem může vést k tomu, že pro daný problém neexistují žádné korektní vstupy určitých velikostí. Zejména se to týká velmi malých vstupů, kdy například nemusí existovat vstupy velikosti 0 nebo 1, apod. Například některé grafové problémy mohou dávat smysl až pro grafy od určité minimální velikosti.

Pokud tedy pro daný problém neexistují pro některá $n \in \mathbb{N}$ žádné vstupy velikosti n, hodnota funkce T vyjadřující časovou složitost daného algoritmu prostě pro tyto hodnoty n není definována a příslušné hodnoty můžeme ignorovat.

Může však existovat ještě jiný důvod, proč nemusí být hodnota funkce $T(n)$ pro některá n korektně definovaná a to i přesto, že existují vstupy této velikosti n. Důvod může být ten, že pro dané n existuje nekonečná posloupnost vstupů, které sice všechny mají velikost n, ale kde každý další vstup má delší dobu výpočtu než všechny předchozí, tj. že existuje posloupnost vstupů

$$w_0, w_1, w_2, w_3, \ldots,$$

kde pro každé $i \in \mathbb{N}$ platí $\mathit{size}(w_i) = n$ a $t(w_i) < t(w_{i+1})$, neboli

$$t(w_0) < t(w_1) < t(w_2) < t(w_3) < \cdots.$$

V tomto případě tedy neexistuje žádný „nejhorší“ vstup velikosti n, na kterém by trval výpočet nejdéle, protože ke každému takovému vstupu existuje vstup velikosti n, na kterém trvá výpočet ještě déle.

Toto znamená, že byla zvolena nevhodná definice velikosti vstupu, tj. definice funkce *size*, protože tato zvolená definice velikosti vstupu plně nezachycuje, jak jsou jednotlivé vstupy „velké“ z hlediska doby výpočtu.

Jako příklad, kdy toto může nastat, uveďme třeba problém, kde by vstupem mohla být libovolně dlouhá (konečná) posloupnost čísel $x_1, x_2, \ldots, x_n$ a kde bychom jako velikost vstupu zvolili počet těchto čísel (tj. n). Přitom by doba výpočtu daného algoritmu nezávisela jen na počtu těchto čísel, ale rostla by s hodnotami těchto čísel a hodnoty těchto čísel by mohly být libovolně velké. Příkladem takového problému by mohl být třeba problém najít největší společný dělitel čísel $x_1, x_2, \ldots, x_n$. Už pro dvě čísla (tj. pro posloupnost x_1, x_2 délky 2) asi není možné omezit dobu výpočtu jejich největšího společného dělitele nějakou konstantou, pokud mohou být tato čísla libovolně velká.

Apendix A

Symboly

V matematice se pracuje s objekty různých typů, například s čísly, funkcemi, množinami, relacemi, posloupnostmi, s různými objekty specifickými pro danou oblast, například v geometrii s body, přímkami, kružnicemi, v teorii grafů s grafy, jejich vrcholy a hranami, apod. Objekty, se kterými se pracuje, je třeba nějak pojmenovat a k tomu se používají různé ***symboly***. V textu se pak mluví například o prvku x, o funkci f a podobně. Pojmenování objektů umožňuje se na daný objekt snadněji odkázat.

Protože je v matematických textech často potřeba pojmenovat hodně různých objektů a 26 písmen latinské abecedy by na to nemuselo stačit, používá se několik různých způsobů, jak rozšířit množství symbolů, které jsou k dispozici. Nejprve je třeba zdůraznit, že v matematických textech se prakticky vždy při pojmenovávání různých objektů rozlišují velká a malá písmena, takže například symboly M a m mohou označovat (a většinou označují) nějaké dva úplně rozdílné objekty. Při pojmenování se rozlišuje nejen to, zda je dané písmeno velké nebo malé, ale i použitý font. Například každý z následujících šesti symbolů může v rámci jednoho textu označovat něco jiného:

$$\mathsf{B} \qquad \mathrm{B} \qquad \mathbf{B} \qquad \mathbb{B} \qquad \mathcal{B} \qquad \mathfrak{B}$$

Dalším prostředkem, jak rozšířit množství dostupných symbolů, je používání dolních a horních indexů, například s_7, s^7 a s_7^7. Dále se používá značení, kdy se příslušný symbol píše s čárkou (např. Q'), se dvěma čárkami (např. Q''), atd. K písmenům je možné kromě čárek přidávat i celou řadu dalších značek. Například symboly

$$\hat{c} \qquad \bar{c} \qquad \vec{c} \qquad \mathring{c} \qquad \dot{c} \qquad \tilde{c}$$

opět mohou označovat každý něco jiného. Všechny výše popsané možnosti (fonty, indexy, značky) je samozřejmě možné kombinovat a psát tak třeba $\vec{\mathcal{H}}_3$.

Kromě písmen latinské abecedy se v matematických textech s oblibou používají také písmena řecké abecedy, například α, δ, φ, Γ, Δ, apod. Výjimečně se používají i písmena hebrejské abecedy, například $\aleph$ (alef).

Přehled všech písmen řecké abecedy je uveden v Tabulce A.1. První sloupec obsahuje velká písmena, druhý malá, třetí český název a čtvrtý anglický název daného písmene. Některá malá písmena řecké abecedy existují ve dvou variantách. U písmen, kterých se to týká, jsou uvedeny obě varianty.

Občas se hodí použít místo jednoho znaku posloupnost více písmen, jako například *succ*, Nodes nebo MARK. Na danou posloupnost písmen se pak díváme jako na jediný symbol. I když může být název tvořený více písmeny více popisný než název tvořený jen jedním znakem, obecně se v matematických textech dává přednost jednoznakovým názvům.

Kromě písmen a číslic se v matematických textech používá nepřeberné množství různých dalších symbolů. Některé z nich mají nějaký víceméně ustálený význam (například $=$, $\in$ nebo $+$), pro většinu symbolů ale platí, že jejich použití je specifické pro daný konkrétní text (a v rámci

velké	malé	česky	anglicky
A	α	alfa	alpha
B	β	beta	beta
Γ	γ	gama	gamma
Δ	δ	delta	delta
E	ϵ, ε	epsilon	epsilon
Z	ζ	zéta	zeta
H	η	éta	eta
Θ	θ, ϑ	théta	theta
I	ι	ióta	iota
K	κ	kappa	kappa
Λ	λ	lambda	lambda
M	μ	mí	mu
N	ν	ný	nu
Ξ	ξ	ksí	xi
O	o	omikron	omicron
Π	π, ϖ	pí	pi
P	ρ, ϱ	rhó	rho
Σ	σ, ς	sigma	sigma
T	τ	tau	tau
Υ	υ	ypsilon	upsilon
Φ	ϕ, φ	fí	phi
X	χ	chí	chi
Ψ	ψ	psí	psi
Ω	ω	omega	omega

Tabulka A.1: Písmena řecké abecedy

toho textu by pak mělo být popsáno, co daný symbol označuje) nebo alespoň pro danou oblast matematiky.

Pro ilustraci uveďme několik málo příkladů symbolů, které se mohou objevit v matematických textech:

$$\preceq \qquad \sqsupset \qquad \lll \qquad \sim \qquad \bowtie \qquad \odot \qquad \rhd \qquad \uplus$$

Podobně jako u písmen, i u ostatních symbolů hrají roli jakékoli rozdíly v jejich podobě. Například se v rámci jednoho textu mohou vyskytnout následující druhy šipek

$$\longrightarrow \qquad \mapsto \qquad \Rightarrow \qquad \hookrightarrow \qquad \rightsquigarrow \qquad \twoheadrightarrow \qquad \rightarrowtail$$

a každá z těchto šipek bude označovat něco jiného.

Rovněž použité závorky mohou mít různý význam a různé druhy závorek mohou označovat velmi rozdílné věci:

$$(x) \qquad [x] \qquad \{x\} \qquad \langle x\rangle \qquad |x| \qquad \|x\| \qquad \lfloor x\rfloor \qquad \lceil x\rceil \qquad [\![x]\!]$$

Přestože je k dispozici velké množství různých symbolů, není nijak neobvyklé, že se jeden symbol v rámci jednoho textu používá současně v několika významech a několik různých objektů je pojmenováno stejně, přičemž konkrétní význam tohoto symbolu závisí na kontextu, ve kterém se nachází. V takovém případě mluvíme o tom, že je daný symbol ***přetížen***. Typickým příkladem takového symbolu, který bývá často přetěžován, je symbol $+$, který může v rámci jednoho textu označovat například sčítání reálných čísel a současně třeba sčítání matic.