Problém, algoritmus, program

Problém
- jsou otázky, které vyžadují nalezení řešení nebo rozhodnutí, např. jestli mají řešení
- jedním ze způsobů řešení těchto otázek je použití algoritmu

Algoritmus
- je konečná množina příkazů, které vedou k nalezení řešení určitého problému

Vlastnosti algoritmu
- *konečnost* – konečný počet kroků
- *jednoznačnost* – každý krok musí být jednoznačně určen
- *vstup* – žádný nebo více
- *výstup* – jeden nebo více

Program
- je zápis, který vyhovuje pravidlům programovacího jazyka
- obecně jde o výpočetní metodu (např. program s nekonečnou smyčkou)
- splňuje-li podmínku konečnosti jde o algoritmus

Vykonání programu

Pro vykonávání programu je třeba počítače. Hlavní části, které se podílí na výpočtu, je procesor a hlavní paměť.

Počítač
- v hlavní paměti je uložen program, ve strojovém kódu jako posloupnost instrukcí a data (*von Neumannova koncepce*)
- v procesoru se vykonávají instrukce, nemusí být vykonávány postupně, viz. Skokové instrukce nebo volání podprogramu
- procesor s hlavní pamětí jsou propojeny sběrnicí

Kompilace
- kompilace je transformování programu z vyššího programovacího jazyka na posloupnost instrukcí
- kompilaci zajišťuje kompilátor a operační systém
- vyžaduje pro každou hardwarovou platformu svůj kompilátor

Interpretace
- nezávislá na HW platformě
- definuje se mezijazyk (v Javě byte-code), který je vykonáván interpretem
- interpret = virtuální stroj (v Javě JVM), nepracuje s registry, ale se zásobníkem

Objekt, třída

Třída

- obsahuje kolekce datových položek (proměnných obsahujících data) a kolekce metod (vykonávají akce nad těmito položkami)
- datové položky a metody se nazývají členy třídy
- třída může být veřejná a nebo privátní, podle toho je uvozena *public* nebo *private*

Objekt

- je samostatný program mající vlastní lokální data a akce definované deklarací třídy
- z pohledu datových typů je objekt datový prvek, který je vytvořen podle vzoru třídy
- podle jednoho vzoru třídy lze vytvořit libovolné množství objektů
- třída je šablona a objekt je vyplněná šablona
- objekt třídy se dynamicky vytváří operátorem *new*, vytvoří se tak reference na objekt a ta se uloží v referenční proměnné, např. *Trida ref_prom = new Trida();*
- inicializace datových členů je umožněna v Javě konstruktory

Spojové datové struktury

Často potřebujeme uchovávat součásti reálného světa, které jsou reprezentovány údaji různých typů. Tato abstrakce má obecný název záznam a říkáme, že jednotlivá data uchováváme v položkách záznamu.

Často potřebujeme sdružovat záznamy v nějaké datové struktuře, např. pole.

Spojové datové struktury se využívají v případě, nemůžeme-li dopředu odhadnout počet ukládaných datových jednotek, protože dynamicky vznikají a zanikají. Uchovávaná datová jednotka obsahuje navíc ukazatel, který ukazuje na další datovou jednotku.

V Javě reprezentuje záznam objekt, který je přístupní pomocí reference (odkazu), která je uložena v referenční proměnné.

Pomocí ukazatelů lze vytvořit libovolné datové struktury.

Správnost programů

Testováním programu, tj. ověřením, že pro daný vstup získáme správný výsledek, *nemůžeme* obecně prokázat správnost programu. Množství vstupů je většinou tak veliké, že testování není možné ani za použití velice rychlých počítačů. Na druhé straně testování *může* alespoň prokázat chybu v programu.

Protože empirickým testováním nemůžeme zaručit správnost programu, musíme ho nahradit analytickým přístupem pro porozumění správnosti programu.

Sekvence příkazů (přiřazení, alternativy a cyklu) umožňuje dokázání správnosti programu tak, že můžeme napsat tvrzení o hodnotách proměnných před vykonáním příkazu, které nazýváme *předpoklad* a tvrzení po jeho vykonání, které nazýváme *důsledek*.

Příkazy sekvenčně

Například pokud máme dva příkazy sekvenčně za sebou, tak vyjdeme z požadovaného důsledku druhého příkazu. Na základě důsledku stanovíme požadovaný předpoklad pro druhý příkaz, což je také požadovaný důsledek pro první příkaz atd. až získáme hodnoty možných vstupů.

Příkazy alternativy

Příkazy alternativy zachovávají sekvenční vykonávání příkazu.

Příkazy cyklu

Pro pochopení příkazů opakování zavedeme pojem invariant cyklu.
Pro invariant cyklu musíme obecně ukázat tři vlastnosti:

- *Inicializace* - je splněn před vykonáním prvního cyklu.
- *Udržování* - je-li splněn pře vykonáním cyklu, zůstává splněn i po něm.
- *Skončení* - když příkaz cyklu skončí, invariant nám ukáže správnost algoritmu.

Přiklad na řazeni pole vkládáním:

Inicializace: Po začátečním přiřazení i = 1 je pole s prvky s indexy 0, ..., i-1 osahující jeden prvek triviálně seřazeno.

Udržování: Cyklus for posouvá již seřazené prvky a[i-1], a[i-2], ... o jednu pozici doprava dokud se nenajde správné místo pro a[i]. Výsledkem je, že prvky pole a[0] ... a[i] jsou seřazeny. Invariant je tedy splněn i pro následující opakování, kdy i je v hlavičce cyklu inkrementováno.

Skončení: Cyklus for skončí když i=n, kde n je počet prvků pole. Dosazením do textu invariantu cyklu dostáváme, že prvky s indexy 0 ... n-1 obsahují seřazené původní prvky pole, co je to co jsme požadovali.

Analýza programů

Analýza algoritmu určuje požadované prostředky na jeho vykonání. Prostředky mohou být čas potřebný na výpočet, paměť pro uložení dat, šířka komunikačního pásma pro přenos dat ap. Při analýze nás především zajímá čas výpočtu, ten je ale ovlivněn mnoha faktory, např. CPU, kompilátorem, chováním ostatních programů, velikostí vstupu atd. Proto je nutné vyjádřit čas výpočtu algoritmu, tak aby nebyl závislý na ovlivňujících faktorech.

Čas výpočtu algoritmu

- instrukce jsou různě dlouhé
- trvání instrukce můžeme vyjádřit počtem elementárních kroků konstantní délky
- čas výpočtu algoritmu, pak bude vyjadřovat počet elementárních kroků $c_{operace}$
- pokud je operace vykonána n-násobně pak $n \cdot c_{operace}$
- čas výpočtu roste s velikostí vstupu

Velikost vstupu

- počet zpracovávaných položek
- charakterizován může být vstup více hodnotami nebo jen jednou, např. zjištění je-li zadané číslo prvočíslo a pak počet zpracovávaných položek závisí na velikosti prvočísla
- mírou velikosti tedy není počet položek na vstupu, ale musíme vstup nějak kvantifikovat, v případě prvočísla např. počtem bitů v jeho binární reprezentaci, tj. $\log_2 n$, kde n je zadané číslo.

Vzhledem k tomu, že čas výpočtu algoritmu pro nejhorší případ je horním ohraničením času jeho výpočtu pro jakýkoliv vstup, je tento čas vhodným kritériem pro porovnávání algoritmů a je tak mírou časové složitosti algoritmů.

Asymptotická složitost

- vyjadřuje limitní růst času výpočtu, když velikost problému neomezeně roste
- až na vstupy malé velikosti, algoritmus, který je asymptoticky efektivnější je pak většinou lepší volbou

Tři druhy zápisu:

- omezuje funkci shora i zdola $g(n) = q(f(n)) : \exists\ 0 \le c_1 f(n) \le g(n) \le c_2 f(n)$
- omezení funkce shora $g(n) = O(f(n)) : \exists\ 0 \le g(n) \le cf(n)$
- omezení funkce zdola $g(n) = \Omega(f(n)) : \exists\ 0 \le cf(n) \le g(n)$

Bude-li čas výpočtu algoritmu omezen shora pro nejhorší případ a zdola pro nejlepší případ stejnou funkcí, bude tato funkce jeho asymptoticky těsným omezení.

Čas výpočtu algoritmu je *Θ(f(n))* právě když jeho čas výpočtu pro nejhorší případ je *O(f(n))* a pro nejlepší případ je *Ω(f(n))*.

Srovnávání složitosti algoritmů

Polynomiální algoritmy

- čas výpočtu je asymptoticky omezen polynomiální funkcí
- reálně použitelné algoritmy

Exponenciální algoritmy

- čas výpočtu je asymptoticky omezen exponenciální funkcí
- použitelné jen pro malé vstupy

Rekurze

- o rekurzi mluvíme, je-li něco částečně složeno nebo definováno ze sebe samého
- základem rekurzivního výpočtu je volání programu sebou samým s vhodnými parametry
- ne každý programovací jazyk umožňuje rekurzivní volání
- základní význam rekurze je, že umožňuje definovat nekonečnou množinu objektů konečným příkazem

Rozlišujeme:

Rekurze v datových strukturách

- rekurzivní struktura je taková, že prvky datové struktury mohou být prvky téhož typu
- použití u definice struktur skládající se z dynamických objektů, např. stromové struktury

Rekurze v řídících strukturách

- rekurzivní procedura je taková, která volá sama sebe
- volání může být přímé (funkce volá sama sebe) a nepřímé (funkce volá jinou funkci, která je zpětně vyvolána)
- problém je s uchováváním starých hodnot v lokálních proměnných
- řešení je ukládání do zásobníkové struktury, což je ale časově a paměťově náročné

Rekurzivní algoritmy jsou vhodné tehdy, operují-li nad datovými strukturami definovanými rekurzivně. Není-li tomu tak, lze s jistotou rekurzivní výpočet odstranit iteračním. Lze-li použít iterační výpočet, dáme mu přednost před rekurzivním z důvodu časové a paměťové náročnosti.

Abstraktní datové typy

Na reprezentanci vlastností částí reality byly vytvořeny matematické abstrakce jako čísla, geometrické útvary, funkce atd., které jsou modely zkoumané reality.

V programovacích jazycích abstrakce datového typu definuje možné hodnoty dat a operace nad hodnotami těchto datových typů.

ADT je matematický model společně s operacemi nad tímto modelem. Používání ADT nám umožňuje abstrahovat konkrétní reprezentace dat a implementace operací nad těmito daty.

Definice

ADT je datový typ, který je přístupný jenom přes rozhraní. Program, který používá ADT nazýváme klientem a program, který je realizací ADT nazýváme implementací.

Všechny členské proměnné a ADT by měly být označeny jako privátní, aby přístup k nim byl zajištěn jen pomocí metod rozhraní. Naopak metody rozhraní musí být veřejné, aby k nim mohl přistupovat jakýkoliv klient, který chce ADT využít.

Rozhraní

- vykonání operace rozhraní implementované metodou, znamená její volání s argumenty správných typů
- na její používání tedy potřebujeme znát typ návratové hodnoty, jméno metody a typy jejích parametrů (signatura metody)
- v Javě můžeme signaturu metody získat z její deklarace vynecháním jmen formálních parametrů v hlavičce, jakož i jejího těla, např. *void zmenaSpotreby (float)*

ADT se staly významným nástrojem pro modulární programování, které slouží na organizování velkých moderních softwarových systémů.

Zásobník, fronta, seznam

Zásobník a fronta jsou dynamické množiny, pro které je specificky definována operace vybrání prvku z množiny.

Zásobník

- dynamická množina, pro kterou je specificky definována operace vybrání prvku z množiny
- operace *vyber ze zásobníku*, vyjme prvek, který byl operací *vlož do zásobníku* vložen jako poslední (LIFO)
- je to důležitá dynamická množina využívána mnoha jinými aplikacemi
- ADT zásobník obsahuje metody na vytvoření prázdného zásobníku, testování prázdnosti a metody na vložení a výběr prvku
- zásobník lze implementovat prostřednictvím pole a spojového seznamu
- operace *pop* nad prázdným zásobníkem způsobí podtečení zásobníku
- operace *push* nad plným zásobníkem způsobí přetečení zásobníku
- přetečení a podtečení lze ošetřit výjimkami nebo dynamickým rozšiřováním pole nebo implementací zásobníku spojovým seznamem, který je však náročnější na čas a paměť

Fronta

- dynamická množina, pro kterou je specificky definována operace vybrání prvku z množiny
- operace *vyber z fronty*, vyjme prvek, který je ve frontě vložený nejdéle, tj. první (FIFO)
- použití například u tiskového serveru
- ADT fronta obsahuje metody na vytvoření prázdné fronty, testování prázdnosti a metody na vložení a výběr prvku
- fronta lze implementovat prostřednictvím pole a spojového seznamu
- u implementace pomocí pole se udržují dvě hodnoty indexů (začátek - odkud vybíráme a konec – kam ukládáme)
-

Seznam

- matematický model ADT seznamu je posloupnost
- ADT seznam je dynamická množina, z které lze prvky vybírat a vkládat na libovolné pozici posloupnosti, např. řádek znaků na obrazovce
- důležitou vlastností seznamu je uspořádání jeho prvků, které je definováno relací následování v množině prvků seznamu, tj. e^i následuje e^{i-1}

Druhy:

Lineární spojový seznam

- přístup k prvkům od okamžité pozice do konce
- poslední prvek má hodnotu *null*

Kruhový spojový seznam

- poslední prvek má odkaz na první prvek seznamu

Obousměrný spojový seznam

- prvky obsahují odkaz nejen na následující prvek, ale i na prvek předchozí

Obousměrný kruhový seznam

- kombinace předchozích dvou

Strom, průchody stromem, binární vyhledávací stromy

Strom

- nelineární ADT, využíván v mnoha algoritmech, např. rodokmen, sportovní výsledky, systém struktury dat a adresářů..
- vyjadřují nějakou hierarchii vztahů
- strom obsahuje vrcholy (prvky), které jsou spojeny hrany
- vrcholy mají své *předchůdce* a *následovníky*
- vrchol, který nemá předchůdce je *kořen*
- vrchol, který nemá následovníky je *list*
- posloupnost vrcholů spojené hranou je *cesta*
- *délka cesty* je počet hran cesty
- *hloubka vrcholu* ve stromě je délka cesty
- maximální hloubka je *výška stromu*
- strom lze implementovat pomocí pole a spojových struktur

Průchody stromem

Při průchodu stromem začínáme od kořene a dále máme tři možnosti:

Přímý průchod (preorder)

- navštívíme vrchol a potom pravý a levý podstrom

Vnitřní průchod (inorder)

- navštívíme levý podstrom, vrchol a pravý podstrom

Zpětný průchod (postorder)

- navštívíme levý a pravý podstrom a potom vrchol

Průchody stromem lze vyjádřit rekurzí. Rekurzi lze odstranit použitím zásobníku.

Binární vyhledávací strom BVS

Pro BVS je operace vkládání do uspořádané množiny prvků nebo výběr prvku z neuspořádané množiny efektivní na rozdíl od pole nebo seznamu.

Prvky jsou ve vrcholech BVS uspořádány tak, že splňují následující BVS vlastnost:

- nechť x je vrchol stromu
- je-li y vrchol v levém podstromu, potom $y.klíč < x.klíč$
- je-li y vrchol v pravém podstromu, potom $y.klíč > x.klíč$

Vlastnosti BVS

- pokud projdeme BVS inordrem dostaneme seřazenou posloupnost.
- snadno nalezneme minimální a maximální prvek, stačí sledovat levý nebo pravý podstrom
- vkládání a výběr prvku je stejně efektivní jako hledání
- časová složitost v nejhorším případě je $O(N)$, kde N je počet různých prvků (strom degraduje na seznam)
- časová složitost v nejlepším případě je $O(log_2 N)$ (AVL stromy)
- obecně složitost BVS je $O(log_2 N)$

Grafy a jejich implementace

Grafy jsou velice obecným prostředkem pro modelování vztahů mezi prvky. Prvky jsou vrcholy grafu a vztahy mezi nimi vyjadřují hrany mezi vrcholy.

Formálně je graf dvojice $G = (V, H)$ kde:

- V je množina vrcholů (uzlů)
- H je množina hran
- hrana je dvojice (u,v) $u,v \hat{I} V$
- počet vrcholů grafu G je $|V|$
- počet hran geafu G je $|H|$
- množina vrcholů grafu G je $V(G)$
- množina hran grafu G je $H(G)$.

Orientované grafy

Platí-li (u,v) ≠ (v,u) je hrana orientovaná, tj. má začáteční a koncový vrchol. Takový graf nazýváme orientovaný a jsou v nich možné hrany z vrcholu do téhož vrcholu. Stupeň vrcholu orientovaného grafu je dán součtem počtu hran do něj vstupujících a počtu hran z něj vystupujících.

Neorientované grafy

Kromě orientovaného grafu máme ještě graf neorientovaný (u,v) = (v,u) , přičemž budeme požadovat u ≠ v. V neorientovaném grafu je stupeň vrcholu dán počtem incidentních hran, tj. hran, kterých je vrchol součástí.

Reprezentace grafu

Dva standardní příklady v informatice reprezentace grafů, použitelné jak pro orientované tak pro neorientované grafy jsou:

Seznam sousednosti

- pro každý vrchol grafu $G = (V, H)$ je vytvořen seznam sousedů
- seznam sousedů obsahuje všechny sousedící vrcholy spojené s vrcholem hranou
- sousedící vrcholy jsou obecně uloženy v seznamech v libovolném pořadí
- potenciální nevýhodou je zjištění existence hrany, kdy se musí procházet spojové seznamy
- paměťová náročnost je $\Theta(|V| + |H|)$

Matice sousednosti

- je vytvořena matice sousednosti S $|V|$ $x|V|$
- matice $S = (s_{ij})$, $i,j = 1, \ldots, |V|$
- je-li $(i,j) \in H$, $s_{ij} = 1$ jinak $s_{ij} = 0$
- implementace dvourozměrným polem
- paměťová náročnost je $\Theta(|V|^2)$

Prohledávání grafů

V případě grafů můžeme, na rozdíl od stromů, začít prohledávání z kteréhokoliv vrcholu.

Prohledávání do šířky BFS (Breath First SEarch)

- algoritmus funkční jak pro orientované tak pro neorientované grafy
- použití finty s obarvováním vrcholu
- čas prohledávání do šířky je $O(|V| + |H|)$

Algoritmus

- vybereme první vrchol, obarvíme ho šedě a vložíme do fronty
- vyjmeme vrchol z fronty a najdeme všechny jeho sousedy a obarvíme ho černě
- sousedi, kteří nejsou ještě ve frontě, přidáme
- pokračujeme, dokud nevyprázdníme frondu a nebudou všechny vrcholy černé

Prohledávání do šířky umožňuje nalezení vzdálenosti vrcholů grafu od začátečního vrcholu, jakož i cesty ze začátečního vrcholu k vrcholům grafu. Pro nalezení cesty stačí uložit předchůdce, pokud je vrchol poprvé nalezen.

BFS je základem např. pro algoritmus minimální kostry nebo minimální cesty.

Prohledávání do hloubky DFS (Depot First Search)

- algoritmus funkční jak pro orientované tak pro neorientované grafy
- použití finty s obarvováním vrcholu
- čas prohledávání do šířky je $O(|V| + |H|)$
- DFS se hodí pro vyhledávání cyklů v grafu

Algoritmus

- vybereme první vrchol, obarvíme ho šedě
- pokud projdeme všechny sousedy, obarvíme ho černě
- vrátíme se k vrcholu, z kterého byl objeven
- pokračujeme druhým bodem

Kromě položek *barva* a *predchudce* zavedeme pro každý vrchol dvě časové značky, pro které je jednotka „času“ přechod na další vrchol a to:

- *objeven* - čas nalezení vrcholu, kdy je obarven šedě
- *dokoncen* - čas konce procházení seznamu sousedů vrcholu, kdy je vrchol obarven černě

Jestliže po vykonání DFS zůstaly neobjevené vrcholy, můžeme jeden z nich vybrat a opět z něho začít prohledávání. Takto můžeme pokračovat, dokud je nějaký vrchol bílý a vznikne tak tzv. les stromů prohledáváním do hloubky. Podobně můžeme postupovat i při prohledávání grafu do šířky. To se hodí například pro topologické řazení.

Topologické řazení

Mějme množinu prvků, ve které je definované uspořádaní jen pro některé dvojice, např. prvky jsou činnosti v čase. Zjednodušeným příkladem je oblékaní, kdy „oblečení“ hodinek je nezávislé na všech ostatních úkonech, avšak před tím než si oblečeme sako, jistě si musíme obléct košili.

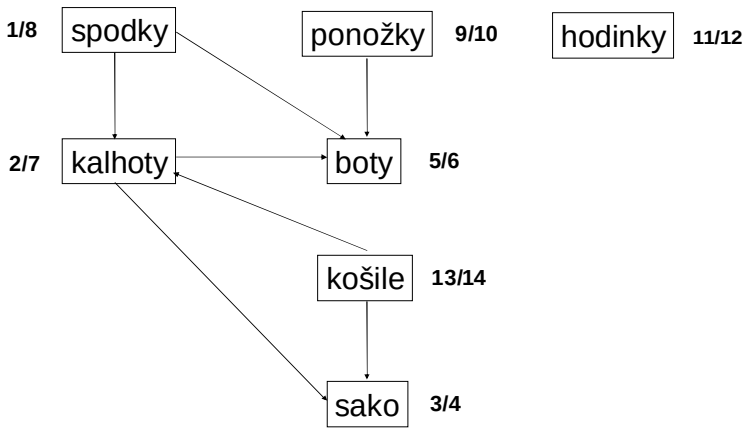
Uspořádáním dvojice prvků *(u,v)* potom tvoří hranu orientovaného acyklického grafu.

Úkolem algoritmu topologického řazení je seřadit vrcholy tak, aby vzájemné pořadí dvojic zůstalo zachováno.

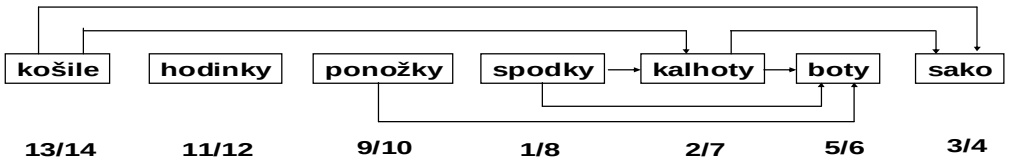
Algoritmus

- zavolá se DFS a vytvoří se les DFS stromů
- když je vrchol ukončen přidej ho na začátek seznamu
- vrať seznam vrcholů

Čas výpočtu topologického řazení je $O(|V| + |H|)$.



Protože vrcholy grafu byly vkládány do seznamu při jejich dokončení, časy dokončení jsou sestupně seřazeny.



Tabulka s přímým adresováním

Odpovídající abstrakce je organizace dat, kde podle jednoznačné hodnoty klíče prvku množiny zpřístupňujeme další hodnoty uložené v prvku množiny. Tímto způsobem práce s daty jsme se setkali již u ADT BVS.

V reálném světě může představovat tabulku například přiřazení závodního čísla závodníkovi. Klíčem pro vyhledávání pak může být přidělené číslo a z tabulky můžeme zjistit např. jméno, věk, pohlaví…. Závodníka.

Tabulky s přímým adresováním jsou vhodné pro menší množiny klíčů. Přístup k informaci je *O(1)*, tj. velmi rychlé operace.

Klíče všech prvků dynamické množiny jsou různé a prvky jsou reprezentovány objekty třídy prvek. Rozhraní ADT tabulky s přímým zobrazováním bude tvořeno následujícími operacemi:

- vytvoření tabulky
- hledání prvku
- výběr prvku
- vložení prvku

Tabulka bude reprezentována polem o velikosti K, jehož prvky budou obsahovat reference na prvky tabulky, respektive budou mít hodnotu null nejsou-li využity.

Alternativní implementace by mohla namísto referencí na prvky dynamické množiny v poli implementujícím tabulku obsahovat přímo samotné prvky. Navíc, jsou-li prvky přímo v tabulce, známe jejich klíč na základě indexu a nemusíme klíč uchovávat v prvku samotném. Musíme ovšem být schopni poznat, že prvek pole je prázdný.

Rozptylové tabulky (hash) s vnějším zřetězením

Rozptylové tabulky se užívají v případech, kdy máme velikou množinu všech klíčů a využijeme jen mnohem menší část z této množiny. Pokud bychom v takovém případě užili tabulku s přímým adresováním, byl by to vzhledem k paměti nešetrný přístup.

Oproti tomu rozptylové tabulky umožňují na jedné pozici (indexu) v tabulce mapovat více klíčů. Nepotřebujeme pak užívat velké tabulky.

K převodu klíčů na indexy se využívají rozptylové (hash) funkce. V ideálním případě přiřadí funkce jednomu klíči jeden index v tabulce.

Při postupném vkládání prvků dynamické množiny do tabulky, v případě, že prvek má být umístněn na pozici v tabulce, která je již obsazena (kolize), v zásadě existují dvě možnosti. První je *vnější zřetězení* prvků, kterých klíče rozptylová funkce zobrazí na stejnou hodnotu. Druhou je *otevřené adresování*, kdy v případě kolize je prvek množiny systematicky umístněn a následně hledán v tabulce.

Vnější zřetězení

Kolize jsou při této metodě řešeny tak, že se vytvoří seznam prvků, jejichž klíče jsou zobrazeny na stejnou pozici v tabulce.

Prvky dynamické množiny jsou reprezentovány objekty třídy *Prvek*, která kromě klíče a dat obsahuje ukazatel na další a předcházející objekt této třídy. Ukazatel ***predch*** není obecně nutný, ale umožní nám efektivní implementaci operace vybrání prvku z tabulky. Opereta nad tímto seznamem jsou pak reprezentovány metodou *hledej*, *vloz* a *vyber*.

Vkládáme-li prvek na začátek seznamu s referencí na sebe, je časová náročnost vložení *O(1). Zásobníkový* charakter takového seznamu je v případě některých aplikací výhodou.

Operace *vyber*, je díky implementaci obousměrného seznamu také *O(1).*

Operace *hledej*, zřejmě závisí na tom, kolik prvků je před hledaným prvkem v odpovídajícím seznamu vloženo v důsledku kolizí.

Některé rozptylové funkce

- multiplikativní metoda
- modulární metoda

Pro rozptylovou tabulku s vnějším zřetězením, můžeme využít tabulku, která je menší než počet používaných klíčů, což například u otevřeného adresování nelze.

Prioritní fronta

Klasickým příkladem, kde se využívají prioritní fronty, jsou výpočetní systémy. Požadavky na přidělení procesoru jsou typicky obsluhovány podle svých priorit. Když se procesor stane volným, přidělí se úloze s nejvyšší prioritou. Během jejího výpočtu mohou přijít další úlohy, a když její výpočet skončí, vybere se opět úloha s aktuálně nejvyšší prioritou, atd.

Libovolnou prioritní frontu můžeme použít na seřazení jejich prvků. Opakováním výběru největšího prvku, získáme jejich sestupné seřazení.

Implementace

Pro ADT prioritní fronta je obdobně jako pro zásobník a frontu možnost implementace prostřednictvím pole nebo spojového seznamu.

Rozhraní ADT obsahuje metody pro operace:

- vytvoření prázdné fronty
- test, je-li fronta prázdná
- vložení prvku
- výběr největšího prvku

Implementace polem

Prvky prioritní fronty můžeme uchovávat v uspořádaném poli, kdy operace výběru prvku, přímo odebere největší prvek z konce pole, ale vložení prvku musí zachovat uspořádaní. Vkládaní, je založeno na algoritmu řazení vkládáním, kde se postupně každý prvek vložil na správné místo do posloupnosti uspořádaných prvků před ním. V nejhorším případě nutno projít všechny prvky v prioritní frontě.

Alternativně můžeme prioritní frontu implementovat neuspořádaným polem, kdy nevětší prvek budeme hledat až při jeho vybíraní. V tomto případě operace vložení prvku má konstantní čas, kdežto operace výběru největšího prvku v nejhorším případě potřebuje projít všechny prvky pole.

Implementace spojovým seznamem

Při implementaci neuspořádaným obousměrným spojovým seznamem, prvek vkládáme na jeho začátek a největší prvek po jeho nalezení přímo odebereme.

Implementace pomocí uspořádaného spojového seznamu by odebírala prvek ze začátku seznamu a pro jeho vložení musí nalézt místo pro vložení. Časová náročnost operací vložení a výběru největšího prvku je v nejhorším případě pro implementaci spojovým seznamem stejná jako pro implementaci pomocí pole.

Uchovávaní prvků prioritní fronty jako neuspořádané posloupnosti je příkladem tzv. *trpělivého (lazy) přístupu* k řešení problému, kdy to co v rámci dané operace nemusíme udělat, odložíme na později. Na druhé straně, uchovávaní prvků prioritní fronty jako uspořádané posloupnosti je příkladem tzv. *netrpělivého (eager) přístupu* k řešení problému, kdy v rámci operace vykonáme co nejvíce práce potřebné pro efektivní implementaci jiných operací.

Dalším efektivní implementací ADT prioritní fronty je prostřednictvím BVS.

Halda

Halda vychází z implementace prioritní fronty pomocí BVS. Pokud potřebujeme, aby prvky ukládané ve stromě byly zohledněny vůči své velikosti klíčů, tak aby byl efektivně vybrán prvek s největším klíčem, pak použijeme právě ADT halda.

Strom má vlastnost haldy, když klíč v každém vrcholu je větší nebo roven klíčům v jeho následnících, pokud je má. Ekvivalentně, klíč v každém vrcholu je menší nebo roven klíči v jeho předchůdci, pokud ho má. Z uvedené vlastnosti plyne, že žádný vrchol ve stromě s vlastností haldy nemá klíč větší než kořen.

Halda je úplný binární strom s vlastností haldy reprezentován pomocí pole.

Problém s výběrem a vložením prvku

Výběr (kořene) prvku vyžaduje jeho náhradu. Abychom zachovali požadavek úplného stromu, nahradíme ho posledním prvkem nejnižší úrovně, což může vést k porušení vlastnosti hlady, pokud by tento prvek měl menší klíč, než některý z jeho následníků.

Na druhé straně, vložíme-li prvek, bude tento prvek dalším listem, a pokud má větší klíč než jeho předchůdce, opět došlo k porušení vlastnosti haldy.

Obnovení vlastnosti haldy

Při porušení vlastnosti haldy se musí halda obnovit.

Obnovení po výběru prvku

- vyměníme klíč s jeho vyšším následníkem
- pokud je porušena vlastnost haldy o úroveň níž, postup opakujeme
- skončíme, pokud není porušena vlastnost haldy nebo se prvek stane listem

Obnovení po vložení prvku

- vyměníme klíč s jeho předchůdcem (má menší hodnotu)
- pokud je porušena vlastnost haldy o úroveň víš, postup opakujeme
- skončíme, pokud není porušena vlastnost haldy nebo se prvek stane kořenem

Řazení haldou je v nejhorším případě $n \log_2 n$.

Algoritmy řazení *O(N logN)*

Problém řazení je považován za nejdůležitější problém při studiu algoritmů. Úkolem je uspořádat množinu prvků obsahujících klíč podle definovaného kriteria.

Řazení haldou

Viz. Halda

Shellovo řazení

Algoritmus Shell Sort vychází z algoritmu Insert Sort, je ale mnohem ekonomičtější. Pan Sell přišel s myšlenkou jak podstatně zkrátit cestu vkládaného prvku na správnou pozici.

Princip je takový, že budeme řadit pouze prvky vzdálené od sebe *h*. Celá posloupnost bude organizována *h* podposloupnostmi začínající prvky *0,1,…,h-1*a každá obsahuje prvky vzdálené *h*. Na tyto prvky tedy aplikujeme Insert Sort.

Volbou dostatečně velkého *h* vzhledem k počtu řazených prvků, tak můžeme posunout prvky posloupnosti na velkou vzdálenost. Dále postup opakujeme pro klesající posloupnost hodnot *h*. Je-li poslední hodnota *1* získáme seřazenou posloupnost.

Příklad:

4	2	7	6	***3***	9	8	5	***1***	0
4	***2***	7	6	3	***9***	8	5	1	***0***
4	2	***7***	6	3	9	***8***	5	1	0
4	2	7	***6***	3	9	8	***5***	1	0
3	2	7	6	***4***	9	8	5	***1***	0
1	2	7	6	***3***	9	8	5	***4***	0
1	***2***	7	6	3	***9***	8	5	4	***0***
1	***0***	7	6	3	***2***	8	5	4	***9***
1	0	***7***	6	3	2	***8***	5	4	9
1	0	7	***5***	3	2	8	***6***	4	9

Problém je trochu s určením jak má být velké *h*.

Shellovo řešení

- hodnotu *h* určit pro *n* prvků jako výchozí *n/2* a v každém kroku ji půlit až dosáhneme jedné
- porovnávány prvky na sudých pozicích s prvky na lichých pozicích budou až v posledním kroku

Knuthovo řešení

- navrhnul zvolit vzdálenost přibližně třetinovou a pak ji dělit třemi

Čas výpočtu algoritmu Shellovým řazením závisí na výběru posloupnosti vzdáleností. Obecné řešení není dosud známo.

Shellovo řazení je nejjednodušší efektivní řazení (i když ne nejefektivnější) a může být proto první volbou v praktických aplikacích, zejména pro rozsah řazených posloupností v řádu do desítek tisíc prvků. Až když Shellovo řazení není postačující, sáhneme k efektivnějším, ale složitějším, algoritmům.

Řazení dělením (Quicksort)

Algoritmus řazení dělením je pravděpodobně nejpoužívanější algoritmus řazení. Algoritmus je založen na rekurzivním řazení pole prvků děleného na dvě části.

Základem uvedeného algoritmu je metoda dělení, která přeuspořádá řazené pole tak, aby byly splněny následující podmínky:

- prvek *a[i]* je na své konečné správné pozici
- prvky *a[l], ..., a[i-1]* jsou menší nebo rovny *a[i]*
- prvky *a[i+1], ..., a[p]* jsou větší nebo rovny *a[i]*

Princip metody

- zvolíme dělící prvek (pivota), který zařadíme na konečnou pozici (většinou se volí právě poslední prvek)
- procházíme prvky zleva a porovnáváme je s pivotem, pokud je prvek větší zastavíme se na něm
- procházíme prvky zprava a opět je porovnáváme s pivotem, pokud je prvek menší zastavíme se na něm
- prvky prohodíme a pokračujeme v procházení stejným postupem
- pokud se indexy zkříží, vyměníme pivota s prvkem, který je v posloupnosti s většími prvky co nejvíce vlevo
- pivot je umístěn na své konečné pozici a nalevo od něj jsou všechny prvky menší nebo rovny a napravo větší
- vznikli tím tak dvě posloupnosti, v každé vybereme pivota a postup dělení znova aplikujeme
- pokračuje až do seřazení celé posloupnosti

Při nalezení prvku stejného jako pivot, můžeme zastavit procházení a použít ho na výměnu. Navíc tato strategie vede k vyváženému dělení pole.

Efektivnost řazení dělením závisí na tom, jak vyvážené je rozdělení řazené posloupnosti. Nejhorší případ nastane, když při každém dělení vznikne jedna část prázdná a druhá se zbývajícími prvky, kromě dělícího prvku, čímž vznikne nejvíc nevyvážené dělení. Složitost je pak $O(n^2)$. V průměrném případě je však složitost $O(n \log_2 n)$.

Rekurzi i v tomto případě lze potlačit použitím zásobníku.

Řazení slučováním

Algoritmus řazení slučováním je komplementární k řazení dělení. Namísto dělení posloupnosti je algoritmus řazení slučováním založen na slučování seřazených podposloupností.

Princip metody

- posloupnost rekurzivně dělíme až na posloupnost o dvou prvcích
- následným slučováním vytvoříme seřazenou posloupnost
- slučování provádíme, dokud není seřazena celá posloupnost

Vidíme, že obdobně jako pro řazení haldou, algoritmus řazení slučováním je *N log2N* pro libovolnou posloupnost *N* prvků.

Redix Sort

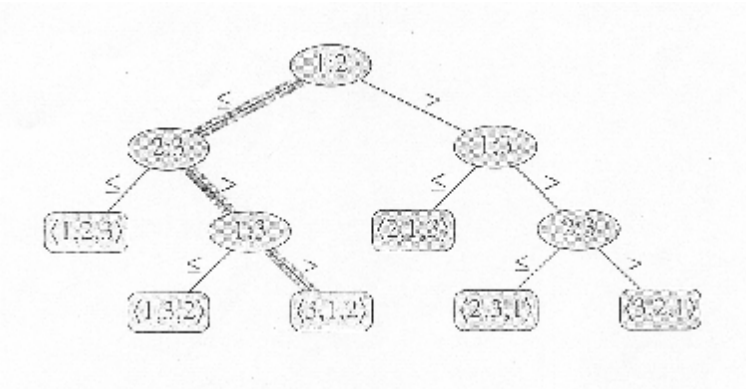
Metoda založena na řazení podle jednotlivých bitů. Dva ukazatelé na koncích a koukáme vpravo na jedničky a vlevo na nuly a prohazujeme.

Dolní omezení pro porovnávací řazení

Dolní omezení počtu porovnání *T(n)* pro porovnávací algoritmy řazení znamená, že počet porovnání pro seřazení *n* prvků musí být větší než nějaká hodnota *g(n)*.

Předpokládejme, že všechny prvky posloupnosti $a_1, a_2, ..., a_n$ jsou různé. Porovnávací řazení můžeme znázornit rozhodovacím stromem. Ve vnitřních vrcholech jsou prvky, které algoritmus porovná a v listech je permutace všech prvků původní posloupnosti, která je seřazena.

Pro posloupnost tří prvků pak vypadá rozhodovací strom takto:



V prvním kroku porovnáme prvky s indexy 1 a 2. Ve druhém kroku jsou-li ve správném pořadí porovnáme prvky s indexy 2 a 3. Nebyly-li ve správné pořadí jsou algoritmem vyměněny v prvním kroku a ve druhém kroku, v tomto případě porovnáváme prvky s indexy 1 a 3. Celý postup pokračuje až v listu stromu jsou indexy v pořadí, které vyjadřuje permutaci, ve které jsou prvky seřazeny.

Jakýkoliv správný algoritmus musí v rozhodovacím stromu vytvořit každou z n! permutací prvků původní posloupnosti, aby dokázal seřadit jakoukoliv vstupní posloupnost. Každá z těchto permutací musí být alespoň v jednom listě.

Nejhorší případ představuje v rozhodovacím stromě nejdelší cesta od kořene stromu k listu $T(n) = h$, což je výška stromu.

Nyní musíme najít dolní omezení všech rozhodovacích stromů. Nechť rozhodovací strom o výšce h má m listů, potom $m \leq 2h$. Současně listů musí být alespoň tolik kolik je permutací, tedy $n! \leq m$. Potom $n! \leq m \leq 2^h$ a logaritmováním dostaneme $\log_2 (n!) \leq h = T(n)$

Použitím aproximace $\left(\frac{n}{e}\right)^n$ pro $n!$ je $\log_2(n!) = n*\log_2 n - n*\log_2 e$, čím dostáváme dolní omezení pro nejhorší případ $\Omega(n*\log_2 n)$.

Protože pro řazení haldou a slučováním je horní omezení času výpočtu $O(n*\log_2 n)$ jsou tato řazení asymptoticky optimální.

Generičnost

Synonymem pojmu generičnost, generický je obecný, univerzální a antonymum je specifický. Zajímavé je použití pojmu generický v metafyzice, epistemologii a logice, kde jde o obecnou kategorii, jako vlastnost nebo vztah, považovanou za rozdílnou od věcí, které jsou její instancí anebo příkladem.

Vrátíme-li se k problematice datových typů a algoritmů, zatím jsme například zásobník museli implementovat znovu, potřebovali-li jsme ho pro ukládání prvků jiného typu. Podobně algoritmy řazení jsme implementovali pro specifický typ klíče, int.

V programovacích jazycích nemusíme pro nový typ prvku psát novou konkrétní implementaci. Máme prostředky pro generickou implementaci datových struktur a algoritmů. Jinými slovy, umíme implementovat generické datové struktury a algoritmy pro obecné prvky a z nich odvodit implementace pro konkrétní typ prvků.

Například z obecného pojmu lampa, od které řekněme požadujeme, aby vydávala světlo, lze odvodit specifické lampy přidáním dalších vlastností. Například stropní musí mít možnost upevnění na strop, stojací musí umět stát, přitom původní vlastnost si zachová, říkáme, že ji zdědí. Podobně můžeme mít od obecného pojmu auto odvozené speciálnější, například osobní auto a nákladní auto. Navíc takováto specializace může pokračovat: auto - nákladní auto - kamión.

Stejná koncepce je v různých formách podporována v objektově orientovaných programovacích jazycích a to dokonce již od jazyka SIMULA 67. Z jistého nadhledu můžeme říct, že třída je opisem vlastností a vztahů pro její instance - objekty. Speciálnější objekty pak musíme popsat novou třídou, odvozenou od původní třídy tak, že zdědí vlastnosti a vztahy původní třídy. Nová třída bude mít další vlastnosti, ale její instance budou tvořit podmnožinu všech instancí původní třídy a budeme je říkat podtřída.

Bezpochyby z důvodů efektivnosti tvorby programů, tj. náročnosti na práci na jejich napsání a odladění, jsou genericky implementované datové typy a algoritmy velice užitečné.

Dědičnost

Obecný způsob vytvoření více specializované třídy je vytvoření podtřídy nějaké třídy, kterou nazveme nadtřídou. Terminologie není ustálená a jako synonyma se používají také základní a odvozená třída nebo rodičovská třída a třída potomek. Tomuto procesu se říká dědičnost. Podtřída, odvozená třída, potomek dědí vlastnosti nadtřídy, základní třídy, rodiče.

V Javě deklarujeme podtřídu B třídy A pomocí klíčového slova *extends*.

class B extends A {...}

Jako podtřídy nějaké třídy může být deklarováno více tříd, například třída C je také podtřídou třídy A

class C extends A {...}

Vztah rodič potomek, odvozování dalších tříd, může dále pokračovat například vytvořením podtřídy D třídy C.

class D extends C {...}

Vzniká tak hierarchie tříd. Třída D je také podtřídou třídy A, protože má všechny její vlastnosti, které získala tím, že je odvozena od třídy C a tato je odvozena od třídy A.

Podtřída, odvozená třída

- získává všechny metody a proměnné nadtřídy
- může přidat nové metody a proměnné třídy
- může předefinovat metody a proměnné rodiče

Pokud má rodičovská třída konstruktor s parametry, konstruktor potomka volá s odpovídajícími argumenty konstruktor rodiče pomocí ***super( )***. Pomocí ***super*** tak můžeme zpřístupnit v potomkovi proměnné i metody rodiče, podobně jako pomocí ***this*** zpřístupňujeme proměnné a metody vlastního objektu.

Referenční proměnná nadtřídy může obsahovat i odkazy na objekty podtřídy. Pokud je objektů podtřídy více, není jasné, na jaký objekt podtřídy referenční proměnná ukazuje. No zjištění reference se používá operátor ***instanceof***. Použití operátoru ***instanceof*** vrací hodnotu ***true*** nebo ***false*** podle toho, je-li objekt dané třídy nebo ne.

Obecně má v Javě každá třída jedinou rodičovskou třídu. Jediná třída, která nemá rodičovskou třídu a současně je tak nadtřídou všech tříd je třída ***Object***. To nám umožní vytvořit například generický zásobník obsahující jakékoliv prvky. Můžeme pak vkládat do zásobníku objekty libovolných tříd a při jejich výběru je přetypovat. Chceme-li generický zásobník použít pro základní datové typy, musíme použít obalující třídy, které jsou k dispozici pro každý základní typ, např. pro ***int*** třída ***Integer***.

Uvedený způsob použití generického zásobníku odporuje definici ADT pomocí definovaného rozhraní. Řešením je použít generickou třídu pro implementaci speciální třídy, kterou nazýváme ***adaptér*** a implementuje požadované rozhraní ADT.

Adaptér je návrhový vzor implementující rozhraní známé klientovi a umožňuje přístup k třídě, která není známá klientovi.

Rozhraní

Java poskytuje na specializaci, kromě dědičnosti, další prostředek – rozhraní. Rozhraní definuje soubor metod bez jejich implementace. Jde tedy o stejnou myšlenku, jako jsme použili u ADT, kde jsme pomocí rozhraní oddělili definici operací od implementace. Třída, která implementuje rozhraní, tedy definovaný soubor metod, (tj. jakoby je zdědí), musí implementovat (tj. jakoby překrýt) všechny metody rozhraní.

Deklarace rozhraní je podobná deklaraci třídy:

interface jmeno {// hlavičky metod}

Každá třída, která implementuje toto rozhraní, musí obsahovat deklarace všech metod rozhraní.

Implementování rozhraní R třídou T zapíšeme:

T implements R {..}

Za předpokladu, že rozhraní je implementováno nějakou třídou, lze k instancím takovéto třídy přistupovat pomocí referenční proměnné typu rozhraní obdobně, jako lze přistupovat k instancím podtřídy pomocí referenčních proměnných nadtřídy.

Implementace rozhraní je výhodná

- chceme-li třídě pomocí rozhraní vnutit zcela konkrétní metody
- vidíme jednoznačné podobnosti v různých třídách, ale začlenit tyto podrobnosti pomocí dědění by vyžadovalo zkonstruovat jejich předka což je činnost, která nemusí být vůbec možná a nebo obtížná

Algoritmická řešitelnost problémů

Formalizace problému

Abychom mohli studovat algoritmickou řešitelnost problému, musíme si pojem problému napřed formalizovat.

Problém Q definujeme jako binární relaci nad množinou instancí problému I a množinou řešení S.

Pro problém řazení přirozených čísel je množina instancí množina všech jednoprvkových, dvouprvkových, ... posloupností a množina řešení je množina těch posloupností, které jsou seřazeny.

Pokud má instance problému právě jedno řešení (některá další instance může mít klidně stejné) je problém vyjádřen funkcí.

Pokud má ale instance problému více řešení, jde obecně o zobrazení.

Algoritmická řešitelnost

Nyní se můžeme ptát, jestli existuje pro problém formalizovaný uvedeným způsobem algoritmus. Přitom bude postačující, pokud se omezíme na třídu rozhodovacích problémů.

Rozhodovací problémy jsou takové, které mají množinu řešení dvouhodnotovou *ano/ne*. Pro rozhodovací problémy je zobrazení množiny instancí na množinu řešení funkcí. Rozhodovací problémy můžeme formulovat ve vztahu k obecnějším problémům. Známe-li řešení rozhodovacího problému, umíme řešit i odpovídající problém.

Zapíšeme-li řešení problému ve tvaru programu do počítače, instance problému je vstup programu a řešení problému bude výstup programu.

Příklad - Problém je-li zadané přirozené číslo liché.

Instance problému (vstup) *I:* $n \in N$, kde *N* je množina přirozených čísel i s 0.
Řešení problému (výstup) *S* budeme kódovat $ano \to 1$, $ne \to 0$.
Rozhodovací problém formalizován jako funkce $f : N \to \{0,1\}$.

I / *n* Î **N**	**0**	**1**	**2**	**3**	**4**	**5**	**...**
S / *f(n)*	**0**	**1**	**0**	**1**	**0**	**1**	**...**

Problém lze řešit v Javě nejméně dvěma algoritmy a to dělení čísla mod2 nebo odečítání od *n* hodnotu dvě ve for-cyklu dokud je *n* větší jak jedna.

Předpokládejme, že najdeme problém, pro který neumíme napsat v Javě algoritmus, jinými slovy je neřešitelný pomocí JVM. Otázka je neexistuje-li jiný formalizmus pro zápis algoritmu, který by takový problém řešil.

Odpověď dává *Churchově - Turingově teze,* která tvrdí, že každý algoritmus může být vykonán Turingovým strojem. Navíc, každý program v konvenčních programovacích jazycích může být transformován na Turingův stroj a naopak. Konvenční programovací jazyky a také Java jsou dostatečné pro vyjádření jakéhokoliv algoritmu. Existuje-li tedy rozhodovací problém, pro jehož řešení neexistuje program, například v Javě, potom je tento problém algoritmicky neřešitelný a naopak.

Klasifikace problémů

Rozdělení vzhledem k řešitelnosti

- řešitelné algoritmy
- neřešitelné algoritmy

Rozdělení vzhledem k hornímu omezení času výpočtu

- polynomiální čas výpočtu $O\left(n^k\right)$, kde *k* je konstanta
- superpolynomiál (př. exponenciální čas výpočtu), např. Hanojské věže $O\left(2^n-1\right)$
-

Problémy, které jsou řešitelné v polynomiálním čase považujeme za zvládnutelné, neboli lehké a problémy, které vyžadují superpolynomiální čas považujeme za nezvládnutelné, neboli těžké.

Optimalizační vs. rozhodovací problém

Podobně jako při zkoumání řešitelnosti problémů, omezíme se na rozhodovací problémy. Na druhé straně mnoho prakticky důležitých problémů jsou optimalizační problémy, kdy výstupem je hodnota, která nejlépe splňuje zadaná kritéria.

Příkladem může být problém, který nalezení nejkratší cesty v grafu. Obvykle můžeme optimalizační problém převést na problém rozhodovací. Známe-li řešení optimalizačního problému, umíme najít řešení rozhodovacího problému. Můžeme tedy říct, že rozhodovací problém je „lehčí“ nebo alespoň není „těžší“ než problém optimalizační. Dá se ukázat, že pokud je rozhodovací problém těžký, je i optimalizační problém těžký.

Pro řešení abstraktního problému na počítači jeho instance (vstupy) musí být kódovány do binárních řetězců. Když počítač řeší nějaký abstraktní problém, ve skutečnosti řeší jeho zakódovanou instanci. Problém v tomto tvaru nazýváme konkrétní.

Třídy složitosti

Třída složitosti P

- konkrétní problém je řešitelný v polynomiálním čase
- existuje-li algoritmus, který ho řeší v čase $O(n^k)$
- časové náročnosti řešení závisí na kódování

Třída složitosti NP

- konkrétní problém není řešitelný v polynomiálním čase
- existence polynomiální verifikace
- verifikační algoritmus ověří řešení problému na základě poskytnutých dat, nazvané certifikát
- problémy, pro které existuje polynomiální verifikační algoritmus, tvoří třídu složitosti NP - nedeterministicky polynomiální.