PPA2

1 Problém, algoritmus, program

- **Problém**
  Věda se zabývá zkoumáním světa kladením otázek o něm. Takové otázky, jejichž zodpovězení vyžaduje nalezení *řešení* nebo zjištění existence řešení, se nazývají problémy.

  Problém je nežádaný stav. Nalezením řešení problému se z tohoto stavu dostaneme do žádaného stavu.

- **Algoritmus**
  Ověřený postup, který vede k nalezení řešení určitého problému. Princip řešení problému, nikoliv implementace v programovacím jazyce (*program*). Jde o postup obecný, nikoliv konkrétní (např. výpočet odmocniny libovolného čísla, ne pouze jednoho konkrétního).

  Konečná posloupnost příkazů. Příkaz je množina přesně definovaných mechanicky vykonatelných operací, při jejichž provádění není potřeba dalšího přemýšlení. Příkazy jsou například rozhodnutí o tom, kterou operací se bude pokračovat (podmínky, opakování, výběr z několika možností), přiřazení, aritmetické operace apod.

  Algoritmus může mít žádný, jeden i více vstupů. Pro libovolnou kombinaci vstupních hodnot však musí najít řešení v konečném počtu kroků. Počet kroků ku počtu vstupů pak určuje algoritmickou složitost.

- **Výpočetní metoda**
  Postup řešení bez podmínky konečnosti se nazývá výpočetní metoda. Algoritmy jsou konečnými výpočetními metodami. Výpočetní metody mohou například určovat princip fungování automatů běžících v nekonečné smyčce, které opakovaně reagují se svým okolím (automat na kávu, snímač JIS, ...).

- **Program**
  Zápis výpočetní metody tak, aby ji byl po případném převedení na strojový kód schopen vykonávat počítač. Nejdříve byly rozkládány přímo do instrukcí počítače, později byly pro lepší srozumitelnost vyvinuty vyšší programovací jazyky. Program je výpočetní metoda zapsaná v souladu s pravidly programovacího jazyka. Součástí programu může být také definice datových struktur používaných v implementaci výpočetní metody. Programem může být také elektrické zapojení nebo mechanický stroj.

2 Vykonání programu

- **Kompilace**
  Program zapsaný v programovacím jazyce je celý najednou převeden na posloupnost strojových instrukcí a v této podobě uložen do paměti. Při spuštění se vykonávají přímo uložené instrukce.

Výhody:	optimalizace při kompilaci
	rychlost
	nalezení více druhů chyb
	chyby odhalovány před spuštěním
Nevýhody:	nutnost nové kompilace pro spuštění na jiné platformě
Jazyky:	C, C++, Pascal, ADA

- **Interpretace**
  Program je buď čten po jednotlivých příkazech, ty jsou samostatně převáděny na strojový kód a ihned vykonávány, nebo je celý překládán do strojového kódu v okamžiku spuštění.

Výhody:	stejný program bez úprav na více platformách
Nevýhody:	pomalejší
	chyby nalezeny až během vykonávání
Jazyky:	PHP, Perl, Python, Ruby

Některé jazyky používají kombinaci obou přístupů. Zdrojový kód v programovacím jazyce je přeložen do univerzálního kódu, který je pak při spuštění převáděn do strojového kódu počítače. Spojují se tak výhody kompilace s možností spouštět program na více platformách, nevýhodou je nižší rychlost provádění programu. Příkladem jazyků jsou Java nebo C#.

3 Objekt, třída

Pro zvládnutí rozsáhlých problémů je použití procedurálního přístupu velmi složité a v některých případech i nemožné. Rozložením na menší části můžeme docílit lepších výsledků.

- **Třída**
  Abstrakce předmětů reálného světa patřících do jedné skupiny. Data popisující jejich společné vlastnosti + operace nad nimi. Proměnné instance (členské proměnné) vs. proměnné třídy, metody instance vs. metody třídy. Veřejné a neveřejné proměnné a metody. Zapouzdření, nezávislost na implementaci. Dědičnost. Poprvné v jazyce Simula67.
- **Objekt**
  Instance třídy, naplnění třídy konkrétními údaji, jeden konkrétní předmět. Inicializace konstruktorem při vytváření. Samostatný program s vlastními daty a akcemi, které jsou definované deklarací třídy.

4 Spojové datové struktury

Použití v problémech, kde neznáme dopředu počet datových jednotek, nebo se jejich počet během vykonávání programu mění. Datovou jednotkou je *záznam* obsahující data o dané jednotce a rozšířený o ukazatel(e) na sousední jednotku (jednotky) ve struktuře. Struktura bývá obalena třídou, která obsahuje ukazatel na první položku a operace, které mohou být nad strukturou prováděny (vložení, výběr, hledání).

- lineární spojový seznam (jedno/obousměrný, bez hlavy/s hlavou)
- cyklický spojový seznam (-"-)
- skiplist
- strom

5 Správnost programů

Program je složen z příkazů (přiřazení, podmínka, cyklus). Vytvoříme tvrzení o hodnotě proměnné před provedením příkazu – *předpoklad* a po provedení příkazu – *důsledek*. Při analýze programu vyjdeme z požadovaného důsledku posledního příkazu a určíme předpoklady, které k tomuto důsledku povedou. Ty jsou pak důsledkem předchozího příkazu a takto pokračujeme, dokud nezjistíme předpoklady pro všechny vstupy programu.

- **přiřazení**
  Přiřazením $b = a$ získáme tvrzení o proměnné a jako tvrzení o proměnné b
- **podmínka**
  Potřebujeme vyjádřit účinek příkazu na základě účinků příkazů uvnitř podmínky
- **cyklus**
  Formulujeme *invariant cyklu* (důsledek i-té iterace) a dokazujeme, že platí před vykonáním cyklu (*inicializace*) a že platí-li před vykonáním obrátky, platí i po něm (*udržování*). Pokud invariant platí i po skončení cyklu, je správnost dokázána.

6 Analýza programů

Určení prostředků (času, paměti, přenosového pásma, ...) potřebných pro vykonání v programu použitého algoritmu. Neměříme čas vykonání programu na počítači, protože je ovlivněn rychlostí počítače, během ostatních programů, konkrétními vstupy a také implementací algoritmu. Místo toho definujeme množinu standardních operací (porovnání, sčítání, násobení, ...) a dobu trvání každé z nich jako násobek elementárního kroku. Algoritmus pak rozložíme na tyto operace a doba vykonání algoritmu je pak součtem dob vykonání všech operací v něm obsažených.

Čas výpočtu obvykle roste s velikostí vstupu. Velikostí vstupu může být počet zpracovávaných položek, velikost vstupní hodnoty apod. Dále závisí čas výpočtu na konkrétních hodnotách vstupu. K jeho ohraničení se používá nejhorší případ, tedy nejdelší doba, kterou může vykonání algoritmu nad vstupy dané velikosti trvat. Někdy je výhodné použití průměrného případu. Míra růstu výpočetního času v závislosti na velikosti vstupu je pak časovou složitostí algoritmu. Limitní případ pro velikost vstupu $\leftarrow \infty$ je *asymptotická složitost*. Asymptotická složitost v nejhorším a průměrném případě jsou obvykle shodné.

- Θ **zápis** ... omezuje funkci shora i zdola kladným násobkem jiné funkce, která je pak jejím *asymptoticky těsným omezením*
- **O zápis** ... omezuje funkci shora kladným násobkem jiné funkce (horní asymptotické omezení)
- Ω **zápis** ...-"- zdola -"- (dolní asym. omezení)

7 Rekurze

K získání výsledku operace potřebujeme znát výsledek téže operace (většinou s jiným vstupem), pro jistý vstup. Obvykle metoda volající sama sebe. Všechna rekurzivní volání metod jsou aktivní naráz a každá má své kopie proměnných ⇒ *paměťově náročné*. Pomocí rekurze lze definovat množiny (spojové seznamy, stromy), fraktální útvary (Hilbertova křivka, Kochova vločka nebo algoritmy (faktoriál, Hanojské věže, průchod stromem). Lze eliminovat použitím zásobníku, někdy také převedením na iterační algoritmus.

8 Abstraktní datové typy

Njenižší abstrakce – bity, potom základní datové typy prog.jazyka, uživatelsky vytvořené typy popisující realitu. Vyšší úroveň abstrakce – ADT. ADT je matematický model společně s operacemi nad tímto modelem. Model ADT vyžaduje přístup k datům pouze přes *rozhraní* ⇒ nezávislost na implementaci. Program využívající ADT = *klient*. Při návrhu programu pomocí ADT nepřemýšlíme nad implementací, pouze definujeme rozhraní.

Dynamická množina – souhrn proměnného počtu prvků identifikovaných pomocí *klíče*, základ většiny ADT. Má operaci vložení a výběru prvku, případně nalezení prvku s daným klíčem.

9 Zásobník, fronta, seznam

- **Zásobník**
  Operace výběru vyjme prvek vložený naposledy, *LIFO* (last in, first out). Vložení = *push*, výběr = *pop*. Implementace statickým polem a spojovým seznamem O(1), dynamickým polem složitější.
- **Fronta**
  Výběr vyjme prvek vložený nejdříve, FIFO (first in, first out). Implementace -"-.
- **Seznam**
  Vkládat a vybírat lze na libovolné pozici. Pozice je buď parametrem metody, nebo uložena okamžitá pozice v seznamu. Implementace pomocí pole O(n), spojovým seznamem s uloženou pozicí O(1) a s předávanou pozicí O(n). Oddělení dat seznamu a operací nad nimi – *iterátor*. Možnost více iterátorů nad jedním seznamem (každý uchovává vlastní pozici v seznamu). Kruhový a obousměrný seznam. Implementace seznamu polem (uložen index další položky místo ukazatele na ni).

10 Strom, průchody stromem, BVS

Příklady stromů: rodokmen, členění knih, systém souborů. Prvky stromu = *vrcholy*, spojení vrcholů = *hrany*, vrcholy bez následovníků = *listy*, ostatní = *vnitřní vrcholy*, nejvyšší vrchol = *kořen*. Rekurzivní definice: list je stromem a je jeho kořenem; spojení vrcholu s kořeny několika stromů je také stromem; připojené stromy jsou pak *podstromy*. *Cesta* je posloupnost vrcholů spojených hranou, počet vrcholů v posloupnosti (bez výchozího) je její délka. Z kořene vede právě jedna cesta do každého vrcholu ⇒ strom neobsahuje kružnice. *Hloubka vrcholu* je délka cesty od kořene k tomuto vrcholu. *Výška stromu* je největší hloubka ve stromu. Unární (seznam), binární, ternární, . . . , n-ární stromy.

Binární strom je takový, jehož každý vrchol má nejvýše dva následovníky. Lze implementovat pomocí pole, kdy kořen je uložen pod indexem 1 a je-li vrchol uložen pod indexem i, pak jeho levý a pravý potomek jsou uloženi na pozicích $2i$ resp. $2i + 1$. Výhodnější je implementace pomocí ukazatelů. Projít všechny vrcholy binárního stromu lze třemi způsoby, které lze implementovat rekurzí nebo pomocí zásobníku. Vždy začínáme od kořene stromu.

- **Preorder** (přímý průchod)
  Nejprve provedeme požadovanou operaci s vrcholem, pak pokračujeme v jeho levém a nakonec v pravém podstromu.
- **Inorder** (vnitřní průchod)
  Začneme levým podstromem, poté vykonáme operaci na vrcholu a končíme pravým podstromem.
- **Postorder** (zpětný průchod)
  Projdeme nejdříve levý a pravý podstrom a nakonec vykonáme operaci s vrcholem.

Nahrazením zásobníku u preorderu frontou docílíme průchodu po úrovních zleva doprava.

Speciální typ binárního stromu je *binární vyhledávací strom*. Levý potomek má hodnotu klíče nižší, než je hodnota klíče vrcholu, a pravý naopak vyšší. Inorder průchodem dostaneme seřazenou posloupnost. Vkládání a

hledání. Při odebírání vnitřních vrcholů nahrazujeme symetrickým následovníkem. Vkládáním seřazené posloupnosti dostaneme lineární seznam – nejhorší případ, složitost operací O(n). V průměrném případě je složitost O($log_2 n$). Mřížková reprezentace – řádek odpovídá hodnotě klíče, sloupec poloze ve stromu.

11 Grafy a jejich implementace

Graf je množina vrcholů (*uzlů*) V a množina spojení mezi nimi (*hran*) H, $G = (H, V)$. Hrana je definována dvojicí uzlů, záleží-li na pořadí – *orientovaný graf*. *Stupeň vrcholu* je počet hran, kterých je vrchol součástí. Je-li $|H| << |V|^2$, graf je *řídký*, naopak pro $|H| \approx |V|^2$ je graf *hustý*.

Grafy lze implementovat dvěma způsoby:

- **Seznam sousednosti**
  Vrchol má položku obsahující spojový seznam s odkazy na všechny vrcholy, do nichž z tohoto vede hrana. Pro neor.graf jsou všechny hrany zaznamenány dvakrát. Vrcholy jsou obvykle uloženy v poli, odkazy jsou pak indexy. Lze použít i pro ohodnocené grafy – ohodnocení se přidá k položkám seznamů. Vhodnější pro řídké grafy (paměťově efektivní, ale pomalejší).

- **Matice sousednosti**
  Hrany jsou zaznamenány ve čtvercové matici rozměru počtu vrcholů. Je-li hrana z vrcholu u do vrcholu v, je v matici 1 na indexu u, v, jinak 0. Pro ohodnocené grafy lze ukládat ohodnocení. Vhodnější pro husté grafy (rychlé, ale počítá i s neex.hranami).

12 Prohledávání grafu

- **Do šířky** (breadth-first search, BFS)
  Vybereme počáteční vrchol a projdeme všechny jeho sousedy. Následně vybereme každého z jeho sousedů a opět projdeme jeho sousedy. Takto postupujeme, dokud neprojdeme všechny sousedy všech vrcholů v grafu. Na začátku vrchol bílý, při prohledání šedý, po prohledání všech sousedů černý. Prohledávání končí nalezením hledaného vrcholu nebo obarvením všech vrcholů na černo. Lze také k vrcholům ukládat předchůdce a počet hran od počátečního vrcholu, čímž získáme cestu a její délku. Prohledáním celého grafu získáme *BFS strom*. Implementace pomocí fronty, složitost O($|H| + |V|$).

- **Do hloubky** (depth-first search, DFS)
  Z počátečního vrcholu jdeme do jeho prvního souseda, z něj do jeho prvního souseda atd., dokud nenajdeme hledaný vrchol nebo nedojdeme do již prohledaného. Pokud najdeme již prohledaný vrchol nebo dokončíme prohledávání všech sousedů vrcholu, vrátíme se o jeden zpět po cestě a zkusíme jeho dalšího souseda. Pokračujeme do nalezení hledaného nebo prohledání všech. Obarvování shodné jako BFS. Prohledáním celého grafu získáme *DFS strom*. Implementace rekurzí nebo zásobníkem, složitost O($|H| + |V|$). Stromové hrany (patřící do DFS stromu), zpětné hrany (opačné ke stromovým), dopředné (nemusí ležet ve stromě, ale zachovávají jeho hierarchii) a křižující (opačné k dopředným). Zpětné hrany indikují cykly v grafu.

Je-li graf rozdělen do několika izolovaných částí, zbudou po prvním prohledávání bílé vrcholy. Můžeme vybrat jeden jako další počáteční a znovu z něho spustit vyhledávání, případně postup opakovat, pokud zůstanou opět bílé vrcholy. Získáme tím les stromů.

13 Topologické řazení

Pro množinu prvků máme definované dvojice, u kterých je určeno pořadí, v jakém se mají vyskytovat. Na základě toho vytvoříme *acyklický orientovaný graf*, který prohledáváme do hloubky. Při dokončení (obarvení na černo) každého vrcholu ho přidáme na začátek seznamu. Výsledný seznam je obrácenou posloupností dokončování vrcholů a udává pořadí, v jakém se mají prvky vyskytovat. Použití například u složitých výrobních procesů, kde se některé prvky musí montovat postupně a jiné nezávisle.

14 Tabulka s přímým adresováním

Máme skupinu záznamů, jejichž klíče lze bez duplicit převést na celé číslo. Je-li neobsazených jen málo hodnot klíče mezi minimální a maximální hodnotou ve skupině, je výhodné ukládat záznamy v poli a klíč použít jako index. Složitost vyhledávání je pak O(1) a pole se nazývá tabulka s přímým adresováním. Pro případy, kdy by nevyužitých hodnot bylo příliš velké procento, je lepší použít rozptylovou tabulku. Příklady: matematické tabulky, seznam závodníků.

15 Rozptylové tabulky s vnějším řetězením

Při velkém množství možných hodnot klíče oproti množství ukládaných prvků (typicky řetězec klíčem) není vhodné či možné použít tabulku s přímým adresováním. Použijeme tedy *rozptylovou funkci*, která množinu klíčů transformuje na množinu celých čísel z menšího rozsahu, aby bylo využití pole dostatečně efektivní. Tato čísla jsou pak indexy v poli. Více klíčů se tím ale zobrazí na stejný index. Pokud pak potřebujeme uložit dva prvky se stejným indexem (tedy dojde ke kolizi), můžeme použít *vnitřní* nebo *vnější* řetězení. Při vnitřním řetězení se kolidující prvek ukládá na jiné volné místo v tomtéž poli a ke stávajícímu je uložen index tohoto místa. Docílíme tím větší paměťové efektivity, ale komplikuje se ukládání dalších prvků, jejichž místo jsme takto obsadili. Druhým způsobem je vnější řetězení, kdy je vyhrazeno další místo pro kolidující prvky a opět jsou ukládány odkazy. Případně lze k položkám tabulky připojit spojové seznamy a do nich ukládat všechny položky pro daný index. Ve všech případech se zvyšuje složitost, protože prohledávání seznamů má složitost O(n). Důležitá je přitom především rozptylová funkce, která by měla ideálně rozdělovat klíče na indexy rovnoměrně (obvykle dělení modulo nebo přepočet na menší interval a zaokrouhlení). Dále je důležitý poměr velikosti tabulky a počtu vkládaných prvků, nejlepší volbou je velikost rovná počtu prvků, kdy ani při přidání dalších prvků se příliš nezhorší doba vyhledávání.

16 Prioritní fronta

Speciální případ ADT podobně jako zásobník a fronta, výběr odebere prvek s nejvyšší či nejnižší prioritou (maximová resp. minimová PF), vkládat lze prvky s libovolnou prioritou. Implementace uspořádaným polem / spoj.seznamem (vložení O(n), výběr O(1)) – *eager přístup* (provádíme vše hned), neuspořádaným polem / spoj.sez. (vložení O(1), výběr O(n)) – *lazy přístup* (odkládáme na později). Implementace pomocí BVS (vložení i výběr O($log_2 n$)), nejvýhodnější implementace = halda.

17 Halda

Strom, jehož každý vrchol má větší nebo stejnou hodnotu klíče jako jeho potomci, má *vlastnost haldy*. Jde-li o binární strom, který je zaplňován po úrovních (je *úplný*), pak je to halda (max-halda, obrácením nerovnosti dostaneme min-haldu). Haldou bývá implementována prioritní fronta. Při výběru největšího prvku (kořene) je tento nahrazen posledním prvkem poslední úrovně (zachovává se úplnost). Nový prvek se přidává za poslední prvek poslední úrovně. V obou případech může dojít k porušení vlastnosti haldy. V prvním se obnovuje zaměňováním nahrazeného prvku s větším z obou jeho potomků, dokud není halda obnovena. Ve druhém je přidaný prvek zaměňován s rodičem, dokud -"-. Oba postupy mají složitost O($log_2 n$). Implementace často pomocí pole. Používá se také k řazení – *heapsort*.

18 Algoritmy řazení O($n\, log_2 n$)

- **Řazení haldou** (heapsort)
  Na začátku je neseřazené pole. Postupně obnovujeme vlastnost haldy pro první dva, tři, čtyři atd. prvky, až máme na poli vytvořenu haldu. Následně zaměníme nejvyšší prvek (vyjmeme ho) s posledním a obnovíme vlastnost haldy na $n-1$ prvcích. To opakujeme až zbude jednoprvková halda a celé pole je seřazené. Nestabilní.

- **Shellovo řazení** (shellsort)
  Posloupnost rozdělíme na podposloupnosti s krokem h (každý h-tý počínaje prvním, počínaje druhým až počínaje $h-1$-ním), které nezávisle seřadíme vkládáním (insertsortem). Následně snižujeme hodnotu h a provádíme totéž až do seřazení s $h = 1$. Prvky daleko od své výsledné pozice se k ní dostanou v menším počtu kroků, než u obyčejného insertsortu. Shell – počáteční h = polovina délky pole, snížení vydělením dvěma; Knuth – posloupnost $3i + 1$, počáteční h = její prvek nejbližší třetině délky pole. Jednoduchý a efektivní algoritmus, složitá analýza, nestabilní.

- **Řazení dělením** (splitsort, quicksort)
  Určíme prvek, jehož hodnota bude rozdělovat pole (*pivot*). Procházíme z obou stran pole a menší hodnoty vpravo prohazujeme s většími vlevo, až se oba průchody potkají. Na tuto pozici přesuneme pivot – vlevo jsou hodnoty menší, vpravo větší. Totéž provedeme s částí pole vlevo od pivotu i vpravo od něj. Rekurze končí, je-li předán pouze jeden prvek k seřazení. Nejčastěji se pivot určuje jako krajní prvek pole (nezasahuje pak do procesu dělení). Lze použít zásobník místo rekurze. Snadno implementovatelné, nestabilní.

- **Řazení slučováním** (mergesort)
  Pole je rekurzivně rozdělováno na poloviny. Dojdeme až na úroveň dvojic prvků, které seřadíme (*sloučíme*).

O úroveň výše opět po dvojicích sloučíme (kopírujeme vždy menší prvek z obou polí a posuneme se za něj) a postupujeme až sloučíme obě poloviny původního pole. Je to nejčastěji používaná metoda a je stabilní.

19 Dolní omezení pro porovnávací řazení

Porovnávací řazení lze znázornit úplným rozhodovacím stromem, kde ve vnitřních vrcholech dochází vždy k porovnání dvou prvků a jejich případnému prohození (např. levý potomek = původní posloupnost, pravý = posloupnost s prohozenou dvojicí). Listy tvoří různé permutace prvků, úplný rozhodovací strom obsahuje každou permutaci alespoň jednou. Strom má tedy nejméně $n!$ listů, zároveň se počet listů dá omezit 2^h, kde h je výška stromu. V nejhorším případě bude nutné projít celou výšku stromu, $T(n) = h$. Potom $n! \leq 2^h$, $log_2 n! \leq h$, $n! \approx \left(\frac{n}{e}\right)^n$ a $n\,log_2 n - n\,log_2 e \leq h = T(n)$. Získali jsme omezení pro nejhorší případ $\Omega(n\,log_2 n)$. Řazení haldou a slučováním mají v nejhorším případě složitost $\mathrm{O}(n\,log_2 n)$, jsou tedy asymptoticky optimální.

20 Generičnost

Máme-li skupinu reálných předmětů se společnými vlastnostmi popsánu třídou, lze přidáním dalších vlastností omezit tuto skupinu jen na ty předměty, které se podobají i v nově přidaných vlastnostech. Stejně lze ale vlastnosti i ubírat, čímž zahrneme pod takovou třídu více předmětů. Odebereme-li z definice třídy všechny vlastnosti, zahrneme tím pod ni všechny ostatní třídy, vytvoříme třídu obecnou, obvykle pojmenovanou *Object*. Toho lze využít při implementaci ADT, které jsou pak univerzální – *generické*.

21 Dědičnost

Vytvoření specializovanější třídy přidáním dodatečných vlastností nebo operací k jiné třídě se nazývá *dědičnost*. Nová třída dědí vlastnosti a metody této třídy a přidává k nim další, je *podtřídou* (potomkem) a třída, ze které byla odvozena je *nadtřídou* (rodičovskou třídou). Vzniká hierarchie tříd, kdy nadtřída třídy je i nadtřídou jejích potomků a obráceně. Instanci podtřídy pak můžeme reprezentovat i jako instanci její nadtřídy, ovšem pak lze přistupovat pouze k vlastnostem a metodám definovaným v této nadtřídě. Nejvyšší nadtřídou bývá třída *Object*. ADT s položkami typu nadtřídy lze používat pro položky libovolné její podtřídy, lze vytvořit generické ADT.

22 Rozhraní

Rozhraní se podobá třídě, ale obsahuje pouze *hlavičky metod*, nikoliv jejich implementaci. Třída může rozhraní použít (*implementovat*, zdědit) a potom v ní musí být obsažena implementace všech metod rozhraní. Jedna třída může implementovat několik rozhraní, čímž lze obejít nebezpečnou *vícenásobnou dědičnost*. Instanci třídy implementující rozhraní lze reprezentovat jako instanci tohoto rozhraní s přístupem omezeným pouze na metody rozhraní. Podobnou kategorií jsou *abstraktní třídy*, které mohou obsahovat jak pouze hlavičky metod (označovaných jako abstraktní), tak i metody implementované.

23 Algoritmická řešitelnost problémů

Problém definujeme jako binární relaci mezi množinou instancí problému I (tj. množinou všech možností vstupu) a množinou řešení S. Dvě instance mohou mít stejné řešení, stejně tak jedna může mít více řešení. Algoritmická řešitelnost zkoumá, zda pro všechny formulovatelné problémy lze nalézt algoritmus řešení.

Ve 30.l. 20.st. objevil Alan Turing formální opis algoritmu – *Turingův stroj*, ve spojení s Alonzem Churchem vytvořili *Churchovu-Turingovu tezi* a sice, že každý algoritmus (ne problém!) lze vykonat Turingovým strojem. Tezi lze vyvrátit nalezením algoritmu nevykonatelného TS. Moderní programovací jazyky pak byly navrženy tak, aby libovolný program šlo převést na TS a naopak. Problém, který nelze vyřešit pomocí TS, tedy ani pomocí programu, je pak *algoritmicky neřešitelný*.

Vytvoříme-li takový rozhodovací problém (řešení je ano/ne), který pro každý rozhodovací algoritmus a alespoň jeden jeho vstup dává pro tento vstup opačnou odpověď, než daný algoritmus se stejným vstupem, takový problém není řešitelný žádným z těchto algoritmů a je tedy *nerozhodnutelný*. První takový problém – *problém zastavení* – našel Turing: program má o všech vytvořitelných programech rozhodnout, zda pro každý ze vstupů daný program zastaví (tehdy vrací jeho výstup v podobě přirozeného čísla + 1) nebo nezastaví (pak vrací 0). Takový program ale nedokáže rozhodnout sám o sobě (pokud by byl vytvořitelný, patřil by mezi zkoumané programy také a musel by ve svém výstupu vrátit svůj výstup + 1).

24 Klasifikace problémů

Problémy můžeme rozdělit na algoritmicky řešitelné a neřešitelné, v případě rozhodovacích rozhodnutelné a nerozhodnutelné.

Řešitelné problémy můžeme dále dělit na základě toho, jak závisí doba výpočtu nejefektivnějšího algoritmu na velikosti vstupních dat. Potom rozlišujeme problémy, jejichž řešení je možné získat v polynomiálním čase (lehké, složitost $O(n^k)$) a v nepolynomiálním čase (těžké). *Třída složitosti P* pak obsahuje všechny problémy řešitelné s $O(n^k)$. Další skupinou problémů jsou takové, které sice nelze řešit v polynomiálním čase, ale lze ověřit správnost poskytnutého řešení v polynomiálním čase. To je *třída složitosti NP* (nedeterministicky polynomiální). Platí $P \subseteq NP$, ale předpokládá se, že $P \neq NP$.

Najdeme-li pro dva rozhodovací problémy A, B funkci, která vstupy A převede na vstupy B tak, aby B dávalo stejné výsledky jako A, je problém A *převeditelný* na problém B a není tedy těžší než B. Pokud lze převodní funkci najít v polynomiálním čase, je A *polynomiálně převeditelný* na B ($A \leq_P B$). Pak $B \in P \ \wedge \ A \leq_P B \ \Rightarrow A \in P$ a složitost algoritmu A je součtem složitosti B a složitosti výpočtu převodní fce.

Nejtěžšími problémy jsou NP-úplné problémy. Na ně lze polynomiálně převést každý problém z třídy NP, tedy T je NP-úplný, jestliže patří do NP a je NP-těžký ($T \in NP \ \wedge \ S \leq_P T, \forall S \in NP$). Existuje-li polynomiální algoritmus pro libovolný NP-úplný problém, pak $P = NP$, naopak neexistuje-li pro žádný, pak $P \neq NP$.