**INFORMATIKA A PROGRAMOVÁNÍ**

**1. Programování – neimperativní, imperativní, praktické programování.**

* **programování**: proces algoritmizace dané úlohy, tj. vytváření postupu vedoucí k vyřešení dané úlohy; proces tvorby programu:
* ***pc program*** – tj. zdrojový kód sepsaný v určitém programovacím jazyce
* ***programovací jazyky***– pro zápis algoritmů, slouží k tvorbě pc programů, 2 zákl. typy – imperativní a deklarativní
* ***programování imperativní***: krok po kroku, co se má dělat (určuje algoritmu přesný postup), program má tvar posloupnosti příkazů (využívají proměnné – jejichž hodnota se může v průběhu výpočtu měnit)
* ***deklarativní (neimperativní) programování***: popíše se pc, jaký výsledek chceme získat (co se má dělat, ne jak se to má udělat), SQL, Prolog (logické programování), funkcionální (vycházející z teorie fcí lambda kalkul= formální aparát výpočetní silou ekvivalentní Turingovu stroji)

**2. Algoritmy a datové struktury (reprezentace objektů, serializace, techniky algoritmizace) – pole, seznamy, objekty, algoritmy typu rozděl a panuj, dynamické programování...**

* ***serializace*** – tj. uchování stavu objektu (objekt se převede na sekvenci bytů a poté se uloží do paměti pc/databáze/souboru)
* ***datové struktury*** – tj. způsob uložení dat v paměti pc a způsob+reprezentují, jaké operace lze s těmito daty provádět; různé způsoby uložení (tj. datové typy); datové typy – specifikuje množinu povolených hodnot a množinu povolených operací; (a) jednoduchý dat. typ (v 1 okmažiku uchovávají jedinou hodnotu), jsou ordinální a neordinální,; (b) strukturovaný dat. typ (v 1 okamžiku uchovávají více hodnot), jsou homogenní (komponenty stejného typu, položky), heterogenní (složky)
* ***typy datových struktur***: **(1) základní**: vyskytují se téměř ve všech programovacích jazycích (za běhu nemění svůj rozsah): proměnná, pole, struktury, objekt; **(2) odvozené** (ADT=abstraktní datové struktury/kontejnery); za běhu mohou měnit svůj rozsah: seznam, strom, zásobník, fronta, tabulka
* ***členění datových struktur***: **(1) dle počtu a uspořádání komponent** – mění se (dynamické), nemění se (statické), **(2) dle komponent uvnitř struktury** – stejného typu (homogenní; komponenty=položky), různého typu (heterogenní; složky); **(3) existuje bezprostřední následník** – ano (lineární)/ne (nelineární)
* ***pole*** – tj. homogenní, statická dat. struktura (tj. posloupnost pevného počtu prvků stejného typu); k prvkům se přistupuje pod jedním identifikátorem (názvem pole); prvky pole jsou indexovány (od 0, tj. k jednotl. prvkům přistupujeme pomocí indexů n-1); indexy – většinou typu integer; 1rozměrné pole (též vektor; k prvku vektoru přistupujeme pomocí 1 indexu p[0]), 2rozměrné (též matice; pomocí 2 indexů – p[0,1] –> 0. řádek, 1. sloupec)
* ***seznam*** – tj. lineární, homogenní, dynamická dat. struktura; prvky lez přidávat na libovolné místo v seznamu; vhodný pro procházení po po řadě (sekvenční průchod); prvkem seznamu může být libovolný dat. typ (seznam seznamů, prázdný); ***spojový seznam*** (lineární seznam) – ukládání předem neznámé délky; zákl. stavební jednotka: uzel (kt. obsahuje ukládanou hodnotu+ukazatel/odkaz na následující prvek)
* ***objekt*** – tj. ADT, objekt je chápán jako datový typ, kt. je tvořen daty, ale také odkazy na podprogramy (operace nad datovými složkami)
* ***fronta* (FIFO)** – pracovat lze pouze s prvkem, kt. je v čele fronty ("kdo dřív přijde, ten dřív mele"); prioritní fronta – hraje roli ohodnocení prvků (ohodnocovací fce) –
* ***zásobník (LIFO)*** – pracuje se s daty, kt. byla uložena jako poslední
* ***dynamické programování*** – definuje se spíše jako přístup k řešení problémů (algoritmus), kdy se složitější úloha rozloží na jednotl. podúlohy, kt. jsou závislé (překrývají se, tj. řeší se úlohy stejného typu); vytvoření DP pro řešení optimalizační úlohy se skládá ze 4 kroků: (1)
* ***algoritmus rozděl a panuj***: užívá se pro řešení komplikované úlohy, kt. se rozdělí na nezávislé podúlohy: rozděl úlohu na nezávislé podúlohy, rekurzivně vyřeš podúlohy, zkombinuj řešení jednotl. podúloh do celkového výsledku úlohy
* vychází z faktu, že se problém rozdělí na podproblémy stejného typu (nejsou na sobě závislé), např. hanojské věže; výpočet rychlé FTT (Fourierovy transformace)
* **https://is.mendelu.cz/eknihovna/opory/zobraz\_cast.pl?cast=370**

**http://kam.mff.cuni.cz/~kuba/ka/**

**http://voho.eu/wiki/turinguv-stroj/**

**3. Grafy a stromy, grafové a stromové algoritmy.**

* ***graf* –** nelineární, dynamická dat. struktura; jednoduchý graf – uspořádaná dvojice G (V, E); sousední (závislé) vrcholy – 2 různé vrcholy spojené hranou x nesousední; podgraf grafu – obsahuje pouze vybrané vrcholy a hrany z grafu G; izomorfní grafy – liší pouze pojmenováním, nebo pořadím svých částí (stejná struktura, podstatné vztahy platí rovnako); stupeň vrcholu – tj. počet hran, kt. z něj vystupují; souvislé komponenty (nejv. souvislé grafy), kostra grafu – tj. podgraf, kt. obsahuje všechny vrcholy původního grafu (je to strom; musí se odstranit všechny cykly)
* ***zadání grafů*** – (1) algebraicky (určením množiny vrcholů a hran), G = (V, E), V= {v1, v2 atd.}; (2) graficky (diagram; vrcholy zakreslíme jako body v rovině a hrany jako křivky, kt. spojují příslušné body); (3) popisem vlastností (je to graf na 5 vrcholech...)
* ***typy grafů*** – triviální graf (1 vrchol, žádná hrana), kružnice (stejný počet vrcholů a hran; stupeň všech vrcholů – 2), cesta (kružnice s 1 chybějící hranou; počáteční a koncový vrchol), úplný graf (všechny vrcholy spojeny hranou), souvislý (mezi každými 2 vrcholy cesta), strom (acyklický, souvislý), orientovaný graf (šipky na hranách, tj. hrany jsou orientovány), ohodnocený graf (hrany jsou ohodnoceny, např. vzdáleností mezi vrcholy), multigraf (povoluje tzv. smyčky – hrana může začínat a končit ve stejném vrcholu)
* ***strom*** – acyklický, souvislý; uzly: kořen strom (1 vyznačený vrchol, kreslíme většinou nahoře), listy stromu (vrcholy stupně 1, kt. nejsou kořenem, kreslíme většinou dole), stromy jsou kostry grafu (podgraf, kt. obsahuje všechny vrcholy původního grafu)
* **grafové algoritmy – DFS (prohledávání do hloubky)**: prohledává se vždy nejlevější neexpandovaný uzel; procedurální (imperativní) program. jazyky uzly uloží do zásobníku; deklarativní (Prolog) – užívá rekurzi, vlastnosti: není úplný (nekonečná větev, cykly), čas. složitost: exponenciální, prostor. složitost: lineární; nejv. problém – nekonečná větev (program nikdy neskončí),; ***BFS*** (prohledávání do šířky): prohledává se vždy nejlevější neexpandovaný uzel s nejmenší hloubkou, imperativní program. jazyky uzly uloží do fronty (FIFO), Prolog – ukládá seznam cest,

**4. Reprezentace čísel a znaků v počítači.**

**5. Databáze, jazyk SQL.**

* ***databázové systémy* –** tj. speciální programy pro ukládání a správu datových souborů (soubory lze třídit, organizovat, měnit, mazat, vyhledávat v nich atd.); uložená data=báze dat (tj. množina informací, o nichž chceme mít přehled); databáze = systém řízení báze dat (program, kt. řídí veškeré přístupy k datům v bázi dat) + báze dat; databázové schéma – popisuje objekty a vztahy mezi nimi; charakteristické rysy: nezávislost dat a programu (změna v datech nevyvolává změnu v programu a naopak); databázový model – určuje vztahy mezi datovými strukturami;
* ***typy databází***: (1) hierarchická (databáze založena na hierarchickém modelu, logické uspořádání dat má stromovou strukturu); (2) síťová databáze (založena na síťovém modelu, ve kt. jsou data logicky i fyzicky uspořádány jako uzly rovinného grafu, každý záznam může být spojený s libovolným počtem dalších záznamů); (3) objektová databáze (databáze založena na objektech, jejich zapouzdření a dědičnosti) (4) relační databáze (založena na relačním modelu, v nichž jsou data logicky uspořádána do relací, tj. výsledků kartouzského součinu nad doménami neboli množinami údajů; relační databáze je založena na tabulkách; nejpoužívanějším typem databází)
* tabulka v relační databázi – sloupce, řádky, každá tabulka má primární klíč (jednoznačně identifikuje záznam v tabulce, žádné 2 řádky nemají stejnou hodnotu primárního klíče), např. ičo
* ***SQL*** – neprocedurální programovací jazyk využívající se zejm. pro dotazování na data (pro komunikaci s relačními databázemi)

**6. Značkovací jazyky – (X)HTML, XML, DTD, ...**

* ***značkovací jazyk*** (Markup Language) – pomocí speciální značek vysvětluje významu (sémantiku) různých částí textu a určuje vzhled (formát) jednotl. částí textu**; dodatečné informace** přidávají do textu v podobě: (a) tagů/značek, (b) příkazů, (c) direktiv; užívají otevřený textový formát (není třeba speciálního vybavení k editaci)
* ***členění značkovacích jazyků***: **(1) jazyky popisné** (tzv. deskriptivní, slouží k popisování druhu informací v dokumentu –> popisují, co je nadpis/odstavec apod.; XML, HTML; **(2) jazyky výkonné** (tzv. procedurální; obsahují navíc instrukce na úrovni programovacích jazyků a umožňují podrobně popsat výstupní vzhled; TeX, PostScript; **(3) jazyky prezentační** (jsou jen o způsobu, jak bude označkovaný text vypadat ve finále – v zobrazení a tisku), RTF
* ***SGML*** – (a) metajazyk, kt. umožňuje definovat, kt. značky (tzv. elementy) lze v textu použít a jak spolu souvisí (např. že uvnitř elementu vyznačující kapitolu se mohou vyskytovat elementy vyznačující odstavce); (b) otevřený standard nezávislý na platformách, soubory SGML jsou ukládány jako text ASCII (což zajišťuje jejich použitelnost prakticky na libovolné pc platformě)
* ***DTD*** (Document Type Definition) – definuje přípustné elementy a vztahy mezi nimi
* ***HTML*** (hypertextový značkovací jazyk) – tj. používá se zejm. na tvorbu a zobrazování hypertextových dokumentů ve www
* XML

**7. Algoritmická složitost – lineární, polynomiální a exponenciální algoritmy, složitost algoritmu a složitost problému, třídy složitosti.**

Turingův stroj je jednoduché abstraktní zařízení, kt. se používá ke studiu vypočítatelnosti (tj. které problémy lze s jeho užitím vyřešit a které ne), vypočítatelnost souvisí s existencí konečného sekvence elementárních kroků, která vede k vyřešení problému (tato sekvence – algoritmus)

**http://voho.eu/wiki/turinguv-stroj/**

**8. Formální jazyky a automaty – regulární jazyk, bezkontextový jazyk, konečný automat, formální gramatika, typy gramatik podle Chomského.**

* ***Formální jazyk*** – (1) tj. množina řetězců (vět) terminálních symbolů (slov) generovaná formální gramatikou; (2) jazyk logiky (výroková logika, predikátová logika atd.), (3) programovací jazyky; v rámci teorie formálních jazyků se užívají 2 typy symbolů: terminály (symboly, kt. mohou být přepisovány); neterminály (symboly, kt. se používají v přepisovacích pravidlech pro popis struktury jazyka)
* ***Formální gramatika*** – je zadaná jako množina přepisovacích pravidel, S (větný symbol; kořenový symbol gramatiky) –> NP VP (neterminály); Zájmeno –> já | ty (terminály); jedním z cílů studia gramatiky je konstrukce formálních gramatik, kt. generují či rozpoznávají jazyky, jež jsou blízko k PJ
* ***Chomského hierarchie jazyků a gramatik*** – Chomsky rozdělil gramatiky do 4 typů na základě různých omezení na tvar pravidel; platí, že každá gramatika regulární je také bezkontextovou, kontextovou a typem 0 (naopak to neplatí, tj. platí zde ostrá inkluze)
* typ 0 (gramatiky bez omezení/frázové gramatiky): na tvar pravidel nejsou kladena žádná omezení (jsou to jazyky tzv. rekurzivně spočetné)
* typ 1 (kontextová): generuje kontextový jazyk, kt. je rozpoznatelný lineárně omezeným automatem, na rozdíl od bezkontextových je možné přehazovat symboly v průběhu derivace (v rámci vnitřní složitosti slov zahrnují všechny PJ, čímž představují horní hranici strukturní složitosti PJ)
* typ 2 (bezkontextová): typicky se pomocí bezkontext. gramatiky modeluje syntakt. struktura PJ (neboť dtivá většina syntakt. struktur je bezkontextová, avšak složitější syntakt. struktury pomocí kontextových)
* typ 3 (regulární):
* ***konečný automat*** – tj. matematický model s konečnou pamětí schopný rozpoznávat určitý jazyk (dají se jimi rozpoznávat regulární jazyky)
* ***regulární výrazy*** – tj. vedle konečného automatu další způsob formální reprezentace regulárních jazyků; umožňují popsat libovolný regulární jazyk výsledek kompozice několika jednoduchých operací nad jazyky

**9. Regulární výrazy, CQL.**

* ***dotazovací jazyk* -** slouží v informatice k dotazování v databázových systémech
* ***CQL*** – tj. korpusový dotazovací jazyk (užívaný v korpusových manažerech)
* ***regulární výrazy*** – jejich fce spočívá v definování v definování vzorů (vyhledávacích masek), kt. se program následně snaží rozpoznat a vyhledat v daném textu
* obecně se CQL a regulární výrazy využívají ke korpusovému vytěžování dat

**https://wiki.korpus.cz/doku.php/pojmy:dotazovaci\_jazyk**

**10. Umělá inteligence (jaké problémy řeší a jaké algoritmy používá).**

* inteligentní agenti (chovají se rozumně; využívají se pro automatické řešení úloh): tj. systém, kt. jedná za nějakým účelem, jedná samostatně, jedná na základě vstupů ze svého prostředí, adaptuje se na změny
* strojové učení
* robotika

***Chomski: Syntaktické struktury:***

***Obecně:***

* syntax studuje principy a procesy kontruování vět v jednotl. jazycích
* cílem syntakt. zkoumání daného jazyka je sestavení gramatiky, kt. by sloužila jako prostředek k vytváření vět daného jazyka
* základním pojmem v lingvist. teorii – lingvist. rovina (např. fonémika, morfologie a frázová struktura), což je v podstatě soubor popisných prostředků použitelný pro konstruování gramatik

***Nezávislost gramatiky***

* jazyk chápán jako množina (konečná či nekonečná) vět, z nichž je každá věta konečná svou délkou a je zkonstruována z konečného počtu prkvů (všechny jazyky ve své formě mluvené/psané jsou jazyky v tomto smyslu), neboť každý přirozený jazyk má konečný počet fonémů (písmen v abc) a každou větu lze reprezentovat jako konečnou posloupnost těchto fonémů (nebo písmen), i když existuje nekonečně mnoho vět
* zákl. úkolem lingvist. analýzy jazyka je L je oddělit gramat. posloupnosti, jež jsou větami L od negramatických, jež větami L nejsou a prostudovat strukturu gramat. posloupností
* gramatika jazyka L bude tedy prostředek, kt. generuje všechny gramat. polsoupnosti L a žádné negramatické

Každé pravidlo je tvaru A → aB nebo A → a, výjimku tvoří S → ε, pokud se S nevyskytuje na pravé straně žádného pravidla. Tzn. přepisujeme **neterminál → terminál[neterminál]** (S → aS, S → b). Jsou ekvivalentní síle konečných automatů, neumí anbn.Jediný možný neterminální symbol na pravé straně pravidla stojí zcela vpravo.

*pozn.*

*Nejčastěji se pracuje s regulárními a bezkontextovými gramatikami a jazyky. Tyto typy gramatik jsou efektivně zpracovatelné na počítačích a jako základ se používají i při zpracování textů v přirozeném jazyce. Přirozený jazyk byl dlouho pokládán za bezkontextový, nyní je prokázáno, že obsahuje kontextové prvky.*

Automaty

* abstraktní model charakterizující jazyky
* stavové systémy, které čtou po znacích slovo na vstupu, s přečtením každého znaku změní stav a na konci rozhodnou, zda slovo je akceptováno či nikoliv
* nejjednodušší je *konečný automat*

Automat A nad abecedou = {a, b} je znázorěn

stavovým diagramem vlevo. Jeho součásti jsou:

**Q** = {q0 , q1 , q2 } . . . množina stavů

**Σ** = {a, b} . . . (vstupní abeceda); symbol ε značí prázdné slovo

**δ** : Q × Σ → Q . . . přechodová funkce, kde

(viz stavový diagram vpravo): δ(q0 , a) = q0 , δ(q0 , b) = q1 ,

δ(q1 , a) = q2 , δ(q1 , b) = q1 , δ(q2 , a) = δ(q2 , b) = q1

**q0** . . . počáteční stav automatu

**F** = {q1 } . . . množina přijímacích stavů automatu (může jich být více)

příklad:

w1 = ababab

|  |
| --- |
| (q0 , ababab) ⊢  zkráceně (v řeči stavů): |

w2 = abba

|  |
| --- |
| (q0 , abba) ⊢ |

*pozn.*

*Je-li A konečný automat nad abecedou Σ , pak jazykem přijímaným automatem A míníme množinu L(A) všech slov prijímaných automatem A, tj. L(A) = {w* ∈ *Σ; A přijímá slovo w}.*

Analýza činnosti konečného automatu

Je dán konečný deterministický automat *A* nad abecedou Σ s množinou stavů Q,

přechodovou funkcí δ : Q x Σ → Q a množinou přijímacích stavů F ⊆ Q.

Jazyk přijímaný automatem *A* je L (*A*).

Iterace přechodové funkce (označovaná δ\*) působí na množině všech konfigurací jako (stav, slovo) ⊢ \* (stav', slovo'). Takový převod nazýváme **dílčím výpočtem automatu *A***.

Speciálně pro libovolné slovo w ∈ Σ\* má výpočet automatu *A* na slově w tvar (qo, w) ⊢ \* (qi, ε ); řekneme, že výpočet skončil ve stavu qi.

Zřejmě platí: w ∈ L(*A*) právě když výpočet automatu *A* na slově w skončí v přijímacím stavu.

Příklad: *Ukázkový automat A nad abecedou Σ = {a, b} je znázorněn stavovým diagramem vlevo.*

Ukažme si nejdříve dílčí výpočet našeho automatu na slově

w = aababba odstartovaný ze stavu q1 mající 4 kroky:

|  |
| --- |
| (q1, aababba) ⊢  Zkrácená forma: (q1, aababba) ⊢ \* |

Přehled malých automatů nad abecedou Σ = {a, b}

Existují právě dva jednostavové automaty nad Σ.

První z nich přijímá všechna slova jazyka Σ\*

a druhý z nich žádné.

Dvojstavových automatů nad Σ se stavy qo,qi je celá řada, ovšem všechny ty, pro něž je F = {q0, qi} přijímají všechna slova jazyka Σ\*, zatímco ty, pro něž je F = Ø nepřijímají žádné slovo.

Dále zvláštní skupinu tvoří ty automaty, jejichž přechodová funkce dává δ(q0, a) = δ(q0, b) = q0.

V nich je totiž stav qi *nedosažitelný*, takže se *redukuji* na jednostavové (viz úvahy výše).

Tabulka níže zachycuje všechny "podstatné" dvoustavové automaty. Ostatní se z nich získají výměnou rolí znaků *a* a *b* či změnou přijímacího stavu z q1 na q0:

**1. Programování – neimperativní, imperativní, praktické programování**

Programovací jazyky slouží k tvorbě počítačových programů (programování). Programování je proces algoritmizace dané úlohy, tj. vytváření postupu vedoucí k vyřešení dané úlohy

XML, HTML, WML apod. nejsou programovací jazyky, ale značkovací jazyky.

http://info.spsnome.cz/Programovani/Zaklady-Pojmy

http://www.interdact.cz/interdact-online/?page\_id=756

**2. Algoritmy a datové struktury (reprezentace objektů, serializace, techniky algoritmizace) – pole, seznamy, objekty, algoritmy typu rozděl a panuj, dynamické programování...**

* **ALGORITMY A DAT. STRUKTURY (POLE, SPOJOVÝ SEZNAM, FRONTA, ZÁSOBNÍK)**

***ALGORITMUS – OBECNĚ:***

* tj. posloupnost konečného počtu element. kroků vedoucí k vyřešení úlohy (jinými slovy přesný postup, jakými lze vyřešit daný problém) –> kvalita algoritmu je dána nejmenším počtem kroků, kt. je k dosažení cíle potřeba
* schematický postup (přesný návod) pro řešení určitého druhu problémů, který je prováděn pomocí konečného množství přesně definovaných kroků. Ačkoliv se dnes tento pojem používá především v informatice a přírodních vědách obecně, tak je jeho působnost daleko širší (kuchyňské recepty, návody a postupy...).
* Je tvořen posloupností pokynů (příkazů, instrukcí), které popisují určitou činnost nebo-li akci
* problém algoritmu: algoritmus je takový postup, kt. se dá realizovat Turingovým strojem (Turingův stroj podle Alana Turinga dokáže vyřešit každou algoritmicky řešitelnou úlohu – tj. kteroukoli úlohu, pro jejíž řešení je možné najít algoritmus –> nemůžeme však ztotožniz pojem TM (Turing Machine) s algoritmem –> neboť TM se nemusí nutně zastavit (např. úloha 1/3 –> stále by připisoval nějaká čísla) –> problémy zastavení a nezstavení TM jsou nerozhodnutelné, ale existují takové Turingovy stroje, o kt. nelze přesně říct, zda svůj výpočet dokončí/nedokončí; další problém: souvisí s výpočtovou náročností algoritmu – reálný čas
* pojem algoritmus – je vágní
* *Akce* je činnost, které má konečné trvání a přesně určený účinek.
* *Příkaz* je popis akce (popis toho, co se má provést).
* *Proces* je postupné vykonávání (realizace) vlastní činnosti (akce) a postupné provádění příkazů návodu procesorem.
* *Procesor* je to, co uskutečňuje daný proces. Podle jednoho návodu (algoritmu) může proběhnout několik různých procesů. Toto zajišťují podmíněné příkazy
* *Proměnná*- je to objekt, který má pevně stanovené označení; má určitou hodnotu, která se v průběhu procesu měnil
* *program* – Algoritmus zapsaný v programovacím jazyce, který řeší nějaký konkrétní úkol. Jedná se o posloupnost instrukcí.

***Způsob zápisu algoritmu:***

* algoritmus lze znázorňovat mnoha způsoby – nejčastěji:

1. *lineárním (textovým) vyjádřením:* v současné době se používá nejčastěji; pomocí formalizovnaého jazyka; využíván pseudokód nebo PDL (Program Description Language)
2. *grafickým zápisem* (v podobě vývojového diagramu; v podobě strukturogramu)

*Vlastnosti algoritmů*

* **Konečnost (finitnost)** – algoritmus má konečné množství kroků (resp. má skončit po vykonání konečného počtu elementárních kroků) – tato vlastnost zaručuje, aby výpočet vždy skončil po vykonání určitého počtu kroků
* **Obecnost (univerzálnost/ hromadnost)** – algoritmus je použitelný na všechny úlohy daného typu (ne však všechny algoritmy jsou univerzální, některé jsou dělané pouze na konkrétní problém a není možné u nich vstupní parametr měnit)
* **Elementárnost** – tj. postup (algoritmus) složený z jednoduchých (elementárních kroků), kt. jsou pro vykonávání (PC, člověk) srozumitelné –> při jeho navrhování je dbát na srozumitelnost, jednoduchost a jednoznačnost
* **Určitost (determinovanost)** – všechny kroky algoritmu jsou přesně definovány, v každé situaci musí být naprosto zřejmé, co a jak se má provést (přesně víme, jak bude algoritmus dále pokračovat); determinovanost – se týká jednotl. kroků výpočtu
* **Korektnost** – algoritmus skončí pro libovolná (korektní) data správným výsledkem v konečném množství kroků.
* **Výstup (resultativnost)** – výpočet dává po konečném počtu kroků výsledek (správný výsledek při řešení libovolné ze skupiny úloh, pro kt. byl vytvořen) – algoritmus musí řešit zadaný problém

### *DĚLENÍ ALGORITMŮ:*

**Rekurzivní a iterativní algoritmy**

***Iterativní algoritmus*** je takový, který spočívá v opakování určité své části (bloku).

***Rekurzivní algoritmus*** naproti tomu opakuje kód prostřednictvím volání sebe sama (obvykle na podproblémech menší velikosti). Každý rekurzivní algoritmus lze převést do iterativní podoby. Samotný převod často řeší automaticky kompilátor nebo virtuální stroj daného programovacího jazyka.

Výhoda rekurzivních algoritmů je v jejich snadno čitelném a kompaktním zápisu. Nevýhodou je spotřeba dodatečných systémových prostředků pro udržení jednotlivých rekurzivních volání.

* ***Iterativní*** – *bubble sort*, *insertion sort*
* ***Rekurzivní*** – *merge sort*, *quicksort*

**Deterministické a nedeterministické algoritmy**

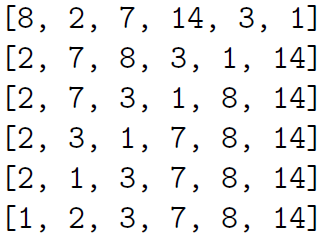
***Deterministický*** je takový algoritmus, který má v každém svém kroku právě jednu možnost, jak pokračovat. ***Nedeterministický*** jich má více. Příkladem může být *deterministický a nedeterministický automat*.

**Sériové, paralelní a distribuované algoritmy**

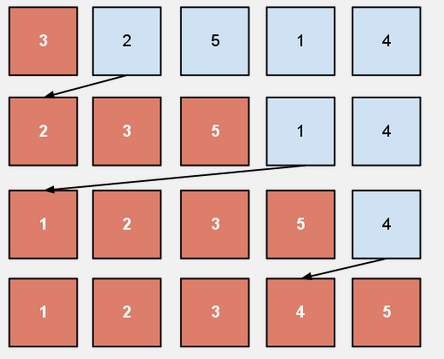
***Sériový*** ***algoritmus*** vykonává všechny kroky v sérii (jeden po druhém), ***paralelní algoritmus*** tyto kroky vykonává zároveň (ve více vláknech) a ***distribuovaný algoritmus*** kroky vykovává zároveň na více strojích.

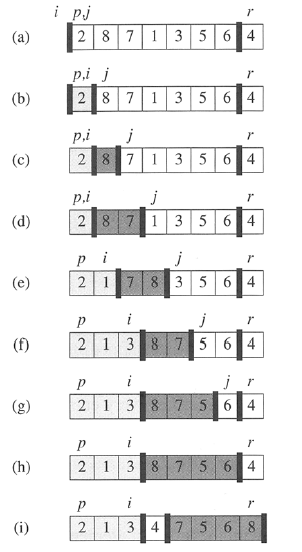
**VYHLEDÁVÁNÍ, ŘAZENÍ;**

u řadicích (třídicích) algoritmů (zápisy z Pythonu); např. seznam (9,8,3,5) se má seřadit podle velikosti

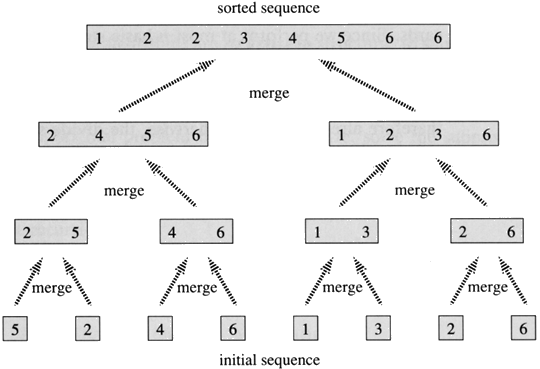
**Bublinkové algoritmy (bubble sort)** – „probublávání vyšších hodnot nahoru“; srovnávání a prohazování sousedů, opakovaně prochází program (nesprávné prohazuje, jinak je nechá být); po i iteracích je nejvyšších i členů na svém místě; řazení posledních částí seznamu je rychlejší, třídí se jen část seznamu; kvadratická složitost

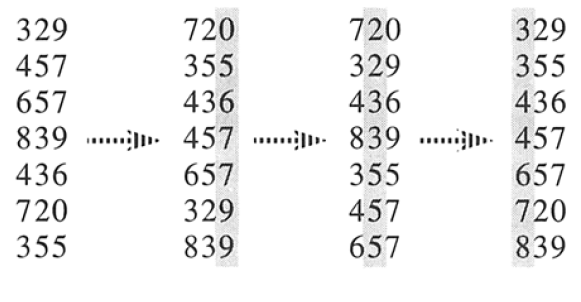
**Řazení výběrem (select sort)** – projdeme dosud nezařazenou část pole a vybereme nejmenší prvek a snažíme se jej dát na správnou pozici; nejmenší prvek zařadíme na aktuální pozici (výměnou); porovnávání s celým programem a zařazuju; kvadratická složitost

**Řazení vkládáním (insert sort)** – prefix pole udržujeme seřazený, každá další hodnota zařazena tam, kam patří (porovnávám ho se sousedy a probublávám); neužitečné moc dlouhé; kvadratická složitost, závisí na vstupu – rychlé, pokud už máme část seřazenou



**Quicksort** – rekurzivní algoritmus, vybereme „pivota“ (ústřední prvek) a pole rozdělíme na dvě části (tj. větší než pivot a menší než pivot); obě části se pak seřazují nezávisle; poté vybíráme dalšího pivota; pokud vybereme špatného pivota, je postup pomalý jako minulé přístupy (obr. č. 4 vpravo) ->>

**Řazení slučováním (merge sort)** – rekurzivní algoritmus, rozdělíme pole na poloviny a ty seřadíme (merge sort), po seřazení je zařadíme do jednoho pole – „zipování“; vždy efektivní



**Radix sort** – předtím se porovnávaly dvě hodnoty, u RS aplikujeme doplňující předpoklady (jde jen o číslo) -> aplikovatelné na krátká čísla

**DATOVÉ STRUKTURY (pole, spojový seznam, fronta, zásobník)**

***Obecně:***

* data používaná programy v paměti – uchovávána v dat. strukturách (data lze reprezentovat různými dat. typy)
* dat. struktury reprezentují vždy nějaký druh dat a umí s ním provádě určité operace (uvnitř dat. struktury pak volíme konkrétní uložení dat a algoritmy pro provádění jednotl. operací –> z toho pak plyne prostor. a čas. složitost operací)
* strukturovaný dat. typ se skládá z kompomenent jiného typu (již dříve definovaného) – tzv. kompoziční typ; pokud jsou všechny komponenty stejného kompozičního typu –> říkáme, že strukturovaný typ je homogenní
* komponentám homogenního typu se někdy říká položky (item), heterog. typu –> složky (component)
* dat. struktura má strukturovanou hodnotu –> ta je definována tehdy, jsou-li definovány hodnoty všech jejích komponent

***Dělení dat. struktur:***

* ***základní***: vyskytují se téměř ve všech program. jazycích (za běhu programu nemění svůj rozsah)
  + ***proměnná*** – slouží k uchování dat a dalších pomocných info v pragramech po dobu výpočtu (každá proměnná má nějaké označení, pomocí něhož se na ni příkazy programu odvolávají); jméno proměnné, kt. má podobu tzv. identifikátoru (jména proměnných nalezneme za klíč. slovem var); typ proměnné určuje, jaké hodnoty se do ní mohou ukládat –> přidělování pamětu do proměnné (tzv. alokace paměti) – 3 základ. metody: statická alokace, alokace na zásobníku a dynamická alokace na haldě
  + pole, struktura, objekt
* ***odvozené***: nazývány jako abstrktní dat. struktury; často implementovány jako objekty; za běhu programu mohou měnit svj rozsah (ADT – abstraktní dat. struktury –> všechny ADT lze označit za kontejnery)
  + seznam, strom, zásobník, fronta, tabulka, fronta
  + členící kritéria:
    - počet a vzájemné uspořádání složek – se mění (dynamické), se nemění (statické)
    - typ objektů uvnitř – všechny stejného typu (homogenní), různého typu (nehomogenní/heterogenní)
    - existuje bezprostřední následník – existuje (lineární; pole, seznam), neexistuje (nelineární; strom, tabulka)

Ukazatel

* tj. zákl. stav. kámen v dynamických dat. strukturách

Od nich:

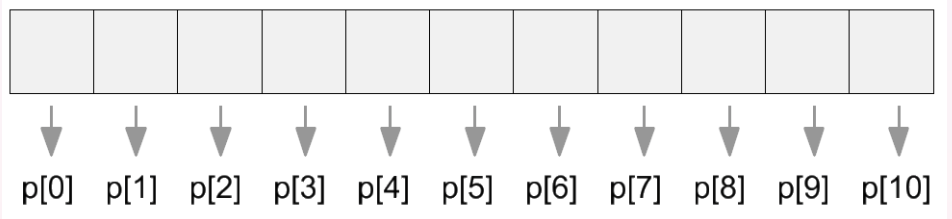
* Datová struktura je konkrétní způsob uložení dat v paměti počítače. Kvůli různým technickým omezením není možné vytvořit jednu univerzální a za všech okolností efektivní datovou strukturu. Proto bylo navrženo mnoho rozličných datových struktur, které jsou používány tam, kde se zdají být nejvhodnější.
* Požadavky na datové struktury jsou často protichůdné. Někdy je prioritou rychlost přístupu k datům, jindy je přednější maximální úspora paměti. Volba té či oné datové struktury by měla být vždy učiněna s přihlédnutím k účelu, k němuž má být struktura nejčastěji použita.
* Operace na datových strukturách – přístup k *n-*tému prvku, vložení prvku, odstranění *n*-tého prvku.

Indexování:

* klíč – jednoznačný identiifikátor záznamu
* pro rychlejší vyhledávání nad klíčem vybudovat index (index = seřazená posloupnost hodnot pro jeden klíč, s ukazateli na záznam)
* indexování indexu – rychlé vyhledávání
* příkl. korpus

*Pole (array)*

* tj. skupina (posloupnost) hodnot (proměnných) stejného typu chápaných jako 1 celek (homogenní typ) – tyto hodnoty jsou společně přístupné pod jedním identifikátorem (názvem pole) – prvky jsou uspořádány souvisle za sebou (–> ***pevný počet položek stejného typu***) x pole lze definovat také dynamicky (nemusí se uvádět počet prvků) – buď pomocí ukazatelů (C, Pascal) nebo jej některé jazyky podporují přímo (Python, PHP)
* každé paměť. místo reprezentováno prvkem pole (k prvku pole přistupujeme pomocí číselného indexu – index pole nesmí být překročen –> tzn. pokud o přístup k prvku, jehož index je mimo specifikovaný rozsah, způsobí chybu) – (číselný index – vyjadřuje pořadí prvku od začátku pole) –> index pole je většinu typu integer, ale může se jednat i ordinální typ
* velikost pole – omezena paměť. nároky
* pole je indexováno vždy od 0 (posl. prvek je proto n-1)
* 1rozměrné pole – též vektor, 2rozměrné pole – též matice (prvke vektoru je zpřístupněn 1 indexem, prvek matice 2 inndexy)
* díky homogenitě nazýváme prvky pole – položky
* typ pole je specifikován rozsahem svých dimenzí (rozměrů) a komponentním typem: rozměr pole je dán počtem prvků, kt. se do pole mohou vložit
* pole může být alokováno staticky/dynamicky –> statická alokace se provádí deklarací (tj. zápisem dat. typu následovaný identifikátorem s hranat. závorkami s uvedením počtu prvků pole); dynamická alokace
* alokace – tj. přidělení paměti
  + ***statické pole*** – tj. pole, jehož jméno a velikost dimenzí je pevně dané deklarací a v průběhu programu se nemění (vs. dynamické)



***Jednodimenzionální pole***

* prvky pole tvoří proměnné (nikoli pole nějakého pole)
* představuje ho n prvků s indexy 0,..., n-1

***Vícerozměrné pole***

* tj. pole označované jako n-rozměrné
* prvky n-rozměrného pole tvoří prvky (n-1) rozměrného pole
* nejčastěji: dvojrozměrné pole, n=2 (prvky tohoto pole tvoří prvky jednorozměrn. pole)



* tvořeno m řádky a n sloupci, představuje matici
* řádk. index: 0,...m-1
* sloupc. index: 0,...n-1
* nejrychleji se mění posl. index (tj. index nejvíce vpravo)
* využití 2rozměrných polí: reprezentace tabulkou, rastrů
* pro práci s poli používány cykly (opakování čínností) – v některých jazycích je pole deklarováno jako dynamické proměnné
* operace s 1D neuspořádaným polem – přístup, hledání – v lineárním čase
* s 2D neuspořádaným polem – v kvadratickém čase

Od nich:

* uložení *n* hodnot v **souvislé paměťové oblasti**, souvislá lineární posloupnost homogenních prvků v paměti, jejichž pravidelné uspořádání zaručuje rychlý přístup ke každému z nich
* **indexování číslem** (→ adresou v paměti) - každý prvek je jednoznačně určen adresou prvního prvku pole a dalším přirozeným číslem (tzv. *indexem*) označujícím vzdálenost požadovaného prvku od začátku pole. Absolutní adresa prvku je pak vypočtena jako součet těchto dvou čísel.
* **prostorová složitost**: *O*(*c · n*), kde *n* je počet záznamů, *c* velikost jednoho záznamu (konstanta)
* **přístup k *n*-tému prvku:** *O*(1) (adresování v paměti) – probíhá v konstantním čase
* **vložení prvku:** *O*(*n*) (realokace)
* **odstranění prvku:** *O*(*n*) (realokace)
* rychlejší přístup, pomalejší modifikace



*Obecně – seznam:*

* tj. zobecnění fronty a zásobníku
* data lze přidávat a odebírat na libovolné místo seznamu
* vhodný pro procházení prvků popořadě –> sekvenční uspořádání dat (nevhodný pro náhodný přístup k prvkům)
  + přístup k 1. prvku – přímo, k ostatním prvkům sekvenční ve směru průchodu
* prvkem seznamu může být libovolný dat. typ – např. seznam seznamů; může být i prázdný

*Spojový seznam (lineární seznam)*

* tj. kontejner určený k ukládání dat předem neznámé délky
* jedná se o nejjednodušší dynamickou strukturu (tj. soustava několika dynamick. proměnných, mezi nimiž jsou vybudovány vzájemné odkazy pomocí ukazatelů); je to nejčastější forma implementace seznamu
* zákl. stavební jednotkou: uzel, kt. obsahuje ukládanou hodnotu a ukazatel na následující prvek (uzly téhož typu)
* (lineárnost – tzn. že každý prvek má právě jednoho předchdce a jednoho následníka – výjimkou je 1. a posl. prvek a navíc nesmí existovat cykly ve vzájemných odkazech – prvky lineární
* každý prvek lineárního spojového seznamu obsahuje záznam obsahující jednak vlastní uložená data libovolného typu, jednak 1 ukazatel na další prvek seznamu
* konec (tail) a začátek (head) – označují zarážky seznamu

***Typy:***

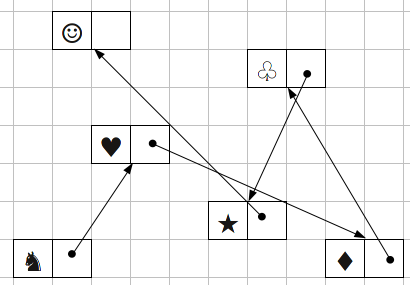
* ***jednosměrný zřetězený seznam***
  + každý prvek seznamu obsahuje mimo dat také ukazatel na další uzel (následující prvek seznamu); mimo posl. prvek (ukazuje na null) –> traverzování 1 směrem (tj. tento seznam lze procházet pouze 1 směrem)
  + samotná struktura seznamu obsahuje pouze ukazatel na 1. prvek
  + data jsou přidávána na začátek seznamu
* obousměrně zřetězený seznam
  + každý prvek seznamu obsahuje odkaz na předchozí a následující prvek seznamu;
* kruhový spojový seznam
  + obsahuje cyklus, vytvořený navázáním konce a začátku seznamu

***Operace lineárního seznamu:***

* vytvoření seznamu
* vložení prvku: probíhá s asymptotickou sloužitostí O(1) – neboť pouze proběhne vložení prvku na začátek seznamu
* odebrání a čtení prvku: proběhne v čase O(i), neboť spojový seznam neumožňuje náhodný vstup
* test prázdnosti
* vyhledávání prvku: O(n)
* počet prvků, vymazání celého, posun vpřed/vzad

Od nich:

* uložení *n* hodnot v **nesouvislé paměťové oblasti**
* **pojme libovolné množství prvků (omezené množstvím dostupné paměti)**
* s každou hodnotou je asociovaný **ukazatel na následníka**
* **prostorová složitost**: *O*(*c · n*), kde *n* je počet záznamů, *c* velikost jednoho záznamu (konstanta) – ale: záznam zde musí obsahovat i ukazatel na následníka → vyšší paměťové nároky než pole
* **časová náročnost** některých operací je vyšší než u pole (vyhledávání, vkládání, mazání), neznáme přesná umístění všech prvků (každý může být na libovolném místě, není řazen lineárně), seznam je tak často nutné procházet
* **přístup k *n*-tému prvku**: *O*(*n*) (lineární průchod seznamem)
* **vložení prvku:** *O*(1) (přepojení ukazatelů)
* **odstranění prvku:** *O*(1) (přepojení ukazatelů)
* pomalejší přístup, rychlejší modifikace
* seznam v Pythonu: seznam prvků oddělených čárkami a uzavřený do hranatých závorek; k prkvům přistupujem přes např. indexy, seznamy lze modifikovat (na rozdíl od řetězců)



*Pole vs. seznam:*

* Pole: rychlejší přístup, pomalejší modifikace
* Seznam: pomalejší přístup, rychlejší modifikace
* Vhodnost použití závisí na datech

*Hashování*

* v předchozích příkladě bylo indexem vždy číslo
* můžeme ovšem chtít indexovat např. řetězcem („ke každému slovu přiřaď četnost“)
* nutná transformace nečíselné hodnoty na číslo – vytvoření tzv. *hashe* pomocí *hashovací funkce*

***Hashovací funkce***

*h* : *D → N*, kde *D* je doména dat; požadavky:

* rychlá
* deterministická (pro stejná data vždy stejný výsledek)
* uniformní (výstupní hodnoty mají přibližně stejnou pravděpodobnost, všechny mají stejnou velikost)
* výstup volitelné délky
* obtížná reverzibilita (použití: hesla)
* minimální změna vstupu vyvolá velkou změnu výstupu

Hashovací funkci nazýváme perfektní, je-li její zúžení na konkrétních vstupních datech injektivní.

* obecně je hashovací funkce vždy neinjektivní a vznikají kolize
* důlěžitý požadavek: neměnnost vstupních dat po vytvoření hashe (*immutability*)

**Hashovací tabulka**

* je datová struktura, která slouží k ukládání dvojic klíč-hodnota
* hashovací tabulka kombinuje výhody vyhledávání pomocí indexu (složitost O(1)) a procházení seznamu (nízké nároky na paměť).

*Fronta*

* reprezentuje model **FIFO** (First In First Out); queue
* opakem zásobníku; pracivat lze pouze s prvkem, kt. je v čele fronty
* prvky se z fronty odebírají v tom pořadí, v jakém se do fronty vkládají (kdo dřív přijde..)
* konec fronty – tj. prvek na posl. pozici ve frontě; do fronty přidán jako posl.
* čelo fronty – tj. prvek na první pozici ve frontě; do fronty přidán jako 1.
* ***operace***:
  + vytvoření prázdné fronty (create)
  + vložení prvku na konec fronty (push)
  + odebrání prvku ze začátku fronty (pop)
  + test prázdnosti (is\_empty)
* ***prioritní fronta*** (neboli fronta s předbíháním):
  + varianta fronty, u kt. hraje roli ohodnocení prvku (díky ohodnocovací fci)
  + princip: prvky přidány do fronty v pořadí, v jakém jsou na vstupu; prvky odebírány z fronty na základě ohodnocení (priorita prvku) –> prvky s nejvyšší prioritou odebrány jako 1.

Od nich:

* slouží k ukládání a výběru dat takovým způsobem, aby prvek, který byl uložen jako první, byl také jako první vybrán, uchovává prvky ve stejném pořadí, v jakém byly vloženy
* implementace pomocí pole nebo spojového seznamu
* Python: list.append(), list.pop(0), ale pomalé (collections.deque)!
* př. použití: prohledávání stromu do šířky



*Zásobník*

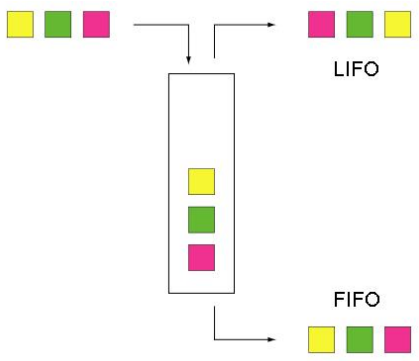
* LIFO – last-in-First-out
* homogenní, lineární, dynamický
* využití – odložení info, výběr v opačném pořadí
* zásobník lze implementovat pomocí pole/spojového seznamu
* operace:
  + vytvoření prázdného zásobníku (create)
  + vložení prvku na vrchol zásobníku (push)
  + odebrání prvku z vrcholu zásobníku (pop)
  + testování prázdnosti zásobníku (top, is\_empty)

Od nich:

* datová struktura typu LIFO (Last In First Out)
* poslední vložený prvek jde na výstup jako první, předposlední jako druhý a tak dále, uchovává prvky v opačném pořadí, než v jakém byly do zásobníku vloženy
* implementace pomocí pole nebo spojového seznamu
* využívá se především pro dočasné ukládání dat v průběhu výpočtu
* Python: list.append(), list.pop()
* př. použití: prohledávání stromu do hloubky



**LIFO vs. FIFO**



**3. Grafy a stromy, grafové a stromové algoritmy**

**GRAFY, STROMY, VLASTNOSTI GRAFŮ, ZÁKL. GRAF. ALGORITMY, PROHLEDÁVÁNÍ STROMU DO HLOUBKY, DO ŠÍŘKY**

***Obecně:***

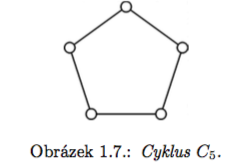
* Nelineární dynamické struktury
  + nejpoužívanější nelineární strukturou – strom

***GRAF***

* jednoduchý graf G je uspořádaná dvojice (V, E), kde V je množina vrcholů, E je nějaká dvouprvkových podmnožin množiny V (tj. množina hran) – prvkům E říkáme hrany
  + množina vrcholů grafu G – označení V(G)
  + množina hran E(G)
* sousední (neboli závislé) vrcholy – tj. 2 různé vrcholy u, v spojené nějakou hranou (tedy existuje hrana uv)
  + v opačném případě – nesousední (nezávislé)
* stupeň vrcholu – počet hran, kt. z daného vrcholu vychází
  + vrchol stupně 0 – tj. izolovaný vrchol
  + graf, ve kt. jsou všechny vrcholy stejného stupně – pravidelný/regulární

***Zadání grafu***

* ***algebraicky*** (tj. určením množiny vrcholů a množiny hran)
  + G= (V, E), kde V={v1, v2, v3, v4, v5}, E={v1v2,v2v3,v3v4, v4v5,v1v5}
* ***graficky*** (tj. zakreslením diagramu – znazorníme vrcholy jako body v rovině a hrany jako křivky, kt. spojují vždy oba body odpovídající vrcholům, jež hranu určují)

******

* ***popisem vlastností (event. ve speciálních případech jménem grafu)***:
  + je to graf na pěti vrcholech, ve kt. je každý vrchol incidentní s právě 2 hranami

***Podgraf grafu G***

* obsahuje pouze vybrané vrcholy a hrany z grafu G
* hrany musí být pouze mezi vybranými vrcholy (výsledek musí tvořit opět graf)

***Isomorfismus mezi grafy G’a G***

* bijekce f : V(G) –> V(G’) taková, že {u, v} je hrana právě tehdy, když {f(u), f(v)} je hrana v G
* grafy jsou isomorfní (shodné), pokud mezi nimi existuje isomorfismus

***Typy grafů***

* obecně:
  + triviální graf – obsahuje jediný vrchol a žádnou hranu
* ***kružnice*** (délky n > 2)
  + V= {1, 2, 3,..,n}, E = {{1, 2}, {2, 3},...,{n-1,n}, {n, 1}}
  + stejný počet vrcholů a hran
  + všechny vrcholy stupně 2
  + nákres grafu tvoří kružnice
* ***cesta*** (na n vrcholech)
  + V = {1, 2, 3,...,n}, E = {{1, 2},{2, 3},...{n-1,n}}
  + kružnice s jednou chybějící hranou
  + počáteční a koncový vrchol
* ***úplný graf*** (na n vrcholech)
  + V={1, 2, 3...,n}, E= {{u, v}| u, v ∈V}
  + každé 2 vrcholy spojeny hranou
* ***zajímavé podgrafy***:
  + ***cyklus*** (kružnice) ***v grafu***: podgraf, kt. je isomorfní s nějakou kružnicí
  + ***cesta v grafu***: podgraf, kt. je isomorfní s nějakou cestou
  + ***klika v grafu***: podgraf, kt. je isomorfní s nějakým úplným grafem
* ***acyklický, resp. les***
  + neobsahuje kružnici (cyklus) jako podgraf
* ***souvislý***
  + mezi každými 2 vrcholy existuje cesta
* ***strom***
  + acyklický souvislý graf

***Další rozšíření pojmu graf***

* ***orientovaný graf***
  + hrany jsou orientovány
  + –> zdrojový a cílový vrchol
  + –> množina hran je množina uspořádaných dvojic
* ***ohodnocený graf***
  + hrany jsou ohodnoceny (např. vzdáleností mezi vrcholy)
  + formálně fce e : G(E) –> **R**
* ***multigraf***
  + povoluje více hran mezi dvěma stejnými vrcholy
  + povoluje hrany začínající a končící ve stejném vrcholu (“smyčky”)

***Analogie se známými pojmy***

* ***graf lze popsat jako relaci na množině vrcholů***
  + množina hran chápána jako relace
  + orientovaný graf – nereflexivní relace
  + neorientovaný graf – nereflexivní symetrická relace
* ***přechodový graf konečného automatu***
  + orientovaný ohodnocená multigraf
  + ohodnocení symboly abc (nikol čísly)
  + (navíc máme vrcholy 2 typů)

***Souvislé komponenty***

* tj. nejv. souvislé grafy
* –> mezi každými 2 vrcholy existuje cesta

***Silné souvislé komponenty***

* v případě orientovaných grafů
* mezi každými 2 vrcholy existuje cesta tami zpět

***Vzdálenost v grafu***

* délka cesty
  + neohodnocený graf: počet hran v cestě
  + ohodnocený graf: součet ohodnocení jednotl. hran v cestě
* vzdálenost mezi 2 vrcholy X, Y
  + je délka nejkratší cesty z X do Y

***Uzly***

* ***kořen stromu:***
  + 1 vyznačený vrchol
  + kreslíme většinou nahoře
* ***listy stromu:***
  + vrcholy stupně 1, kt. nejsou kořenem
  + kreslíme většinou dole

***Kostra grafu***

* ***podgraf, kt.:***
  + obsahuje všechny vrcholy původního grafu
  + je strom
  + –> musíme odstranit všechny cykly
* ***minimální kostra grafu:***
  + pro ohodnocený graf
  + kostra s nejmenším součtem ohodnocených hran
  + analogicky max. kostra

***PROCHÁZENÍ GRAFU***

* např. hledáme určitý vrchol, chceme projít všechny
* DFS a BFS – tj. algoritmy procházení grafu

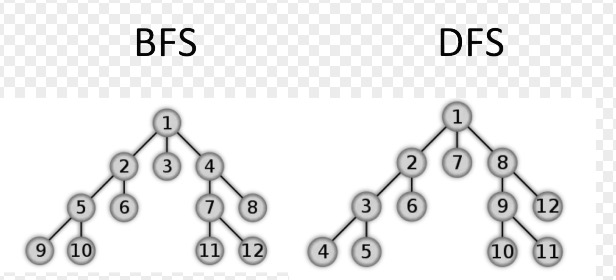
***Procházení do hloubky (depth-first search)***

* začínáme z nějaké vrcholu, ten označíme
* označíme libovolný sousední neoznačený vrchol a pokračujeme z něj
* pokud to dál nejde (všechny sousední vrcholy jsou již označené), vrátíme se k nejbližšímu vrcholu, ze kt. to ještě jde

***Např. z vrcholu u:***

DFS (G, u)

* označ u
* for všechny hrany (u,v) vycházející z vrcholu u:
  + if: není označen:
    - DFS(G, v)



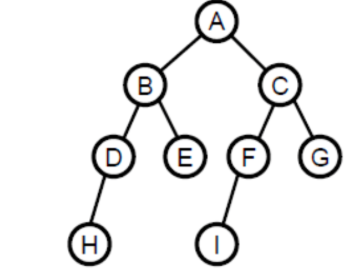
***Procházení do hloubky (breadth-first search)***

* začínáme z nějakého vrcholu, ten označíme
* vybereme všechny sousední neoznačené vrcholy a přidáme je do seznamu
* postupně ze začátku seznamu odebíráme a provádíme předchozí kroky
* končíme, když je seznam prázdný

***Např. Procházení do šířky z vrcholu v***

BFS(G, v)

* Q=[u]
* while Q je neprázdný:
  + odstraň první prvek z Q a přiřaď jej do t
  + označ t
  + přidej všechny neoznačené sousedy t na konec Q



***STROM***

* jedná se o 1 z nejjednoduších a zákl. typu grafů (tj. souvislý graf bez cyklů/kružnic)
* spolu s ostatními grafy jsou předmětem matemat. disiplíny zvané teorie grafů
* tj. takové uspořádání dat, kdy každý prvek má nejvýše jednoho předchodce a může mít více než 1 následníka
* strom tvořen vrycholy (uzly)
* vzdálenost od kořene k nejvzdálenějšímu listu = tj. hloubka stromu
* pro označení vzájemné polohy uzlů se někdy užívají slova představující příbuzenské vztahy (o následnících libovolného uzlu – říkáme že jsou jeho synové a o předchodcích – že jsou jeho otec) – synům téhož otce = bratři
* stromová dat. struktura – rekurzívní: strom je tvořen kořenem, z něhož vede několi (v krajním případě žádný, v případě binárního stromu nanejvýš 2) odkazů na postromy, z nichž každý je opět stromem
* strom je buď prázdný, nebo obsahuje další dat. struktury – podstromy (podstrom definujeme stejně jako strom)
* mezi každými 2 vrcholy (uzly) existuje právě jedna cesta (musí existovat, neboť strom je souvislý graf a více také ne – neboť netvoří kružnice)
* platí pro ně vztah: |E| = |V| - 1 (tzn. počet hran je vždy o 1 menší než je počet uzlů)

***Vlastnosti:***

* stupeň (arita) uzlu – počet přímých následovníků vnitřního uzlu
* stupeň stromu – max. stupeň mezi všemi uzly

***Typy uzlů***:

* listy – uzly bez následníka; není k nim připojen žádný podstrom
* kořen – výchozí bod stromu; uzel bez předchůdce; existuje právě 1; kořen má několik následníků, ty mají za své následníky atd. (max. počet následníků libovolného uzlu není obecně nijak vymezen)
* vnitřní uzly – uzly, kt. nejsou listem, ani kořenem

***KRUSKALŮV ALGORITMUS***

***Obecně:***

* slouží k vyhledání minimální kostry ohodnoceného grafu
* postupně vybíráme nejkratší hrany, aniž by došlo k vytvoření kružnice
* celková váha (součet délek) hran grafu je minimální

***Vstup***

* neorientovaný graf G ohodnocení hran w

*Výstup*

* minimální kostra grafu G

***Algoritmus je tzv. hladový***

* v každém kroku vybírá lokálně optimální možnost

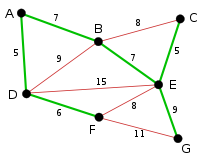
***Idea***

* + setřídit hrany podle ohodnocení
  + v každém kroku přidat do kostry tu nejmenší, která nevytvoří cyklus
  + udržujeme si seznam souvislých komponent kostry

***Kruskalův algoritmus (G, w)***

* K <– []; comp <– {}
* for u in G(V):
  + comp[u] <– set(u)
* setřiď G(E) podle w:
* for (u,v) in G(E)
  + if comp[u] ̸= comp[v] :
    - K.append((u,v))
    - newset = union(comp[u], comp[v ])
    - for x in newset : comp[x] ← newset
* K je minimální kostra grafu

**Kruskalův algoritmus**



***DIJKSTRŮV ALGORITMUS***

***Obecně:***

* slouží k nalezení nejkratší cesty v grafu
* pro každý vrchol si pamatuje délku nejkratší cesty, kterou se k němu dá dostat
* prochází množinu nenavštívených uzlů, dokud není prázdná

***Vstup***

* graf s hranami ohodnocenými funkcí w
* ohodnocení hran musí být nezáporné
* počáteční vrchol s

***Výstup:***

* vzdálenosti z vrcholu s do všech dalších vrcholů grafu

***Idea***

* udržujeme si nejmenší známé vzdálenosti do všech vrcholů
* na začátku nekonečno
* procházíme postupně vrcholy a hodnoty upravujeme
* for u in G(V):
  + d[u] ← infinity
* d[s] ← 0
* N ← G (V )
* p ← {}
* while N ̸= [ ]:
  + u ← vrchol z N s nejmenší hodnotou d[u]
  + for všechny hrany (u,x) vycházející z vrcholu u:
    - alt ←d[u]+w((u,x))
    - if alt < d(x) : d[x] ← alt;p[x] ← u
  + odstraň u z N
* d jsou vzdálenosti vrcholů z vrcholu s  
  p obsahuje předchozí vrcholy na nejkratší cestě z s

**4. Reprezentace čísel a znaků v počítači**

* **REPREZENTACE ČÍSEL V PC**
* ***Obecně:***
* pravidla pro zápis čísel pomocí čísliv nazýváme číselnou soustavou – každá číselná soustava, kt. zapisuje čísla pomocí mnohočlenu (polynomu) se nazývá polyadická (polynomická) číselná soustava o základu z (např. desítková soustava má základ, z = 10 apod.) –> Pc mají logické obvody, kt. pracují se 2 logickými stavy (1 – zapnuto, 0 – vypnuto), proto je základem všech dnešních PC technologie založená na dvojk. soustavě – též tzv. poziční číselné soustavy (hodnota každé číslice je dána její pozicí v sekvenci symbolů)
* základem poziční soustavy – základ (z) – což je zpravidla přirozené číslo větší než 1 (určuje počet symbolů užívaných v dané soustavě)
* číslice v dané číselné soustavě jsou vlastně zbytky po dělení čísel základem dané soustavy (pokud označíme základ číselné soustavy z, pak je počet číslic roven z a nejv. číslice (resp. největší zbytek) je roven z-1
* zapisujeme-li nějaké číslo v soustavě, je třeba udat, o jakou soustavu se jedná, jinak mu nemůžeme přiřadit žádnou hodnotu
* x nepoziční (nepolyadické) číselné soustavy – např. římské číslice
* zpravidla dochází k převodu informací na číselný kód (přičemž počítač/stroj pracuje s dvojkovou číselnou soustavou –binární) –> převod na posloupnost bitů
* běžně používané číselné soustavy: desítková (decimální), dvojková, šestnáctková (hexadecimální), (dále osmičková (oktálová))
* v počítači jsou čísla uložena v binární soustavě, stejně jsou uložena i desetinná čísla – ta jsou reprezentována jako zlomky se jmenovatelem, jež je roven mocninám čísla 2
* informace se v současnosti přenáší pomocí digitálního signálu
* počítače používají tzv. **dvojkovou soustavu**, jsou to digitální zařízení
* tento způsob se technicky dobře realizuje pomocí elektrického signálu (není napětí = 0; je napětí = 1) i pomocí mechanických prostředků (0 = není prohlubeň; 1 = je prohlubeň nebo výstupek). Jedna nula nebo jednička (něco je nebo není) je také **nejmenší jednotka informace**, která říká, který ze dvou stejně pravděpodobných stavů nastal
* všechny PC mají omezené max. a min. číslo, kt. lze zpracovat; protože je počet bitů paměti pro uložení čísla omezený, je omezené také max. a min. uložitelné číslo

***Kódování dat***

* data v PC: celá čísla, čísla s řádovou čárkou (necelá), znaky různých abeced (pro písmena, cifry, symboly atd), alfanumerické znatky, speciální a řídící unaky
* binární reprezentace – tj. kódování dat do posloupnosti binárních hodnot
* kód (kódování) – tj. zobrazení čísel a znaků na binární hodnoty, pomocí kódových schémat a tabulek
* kódové slovo – binární hodnota, obecně posloupnost kódových znaků
* dekódování – tj. převod kódového slova na původní čísla/znak
* různé kódy pro uložení dat, zpracování dat, zabezpečeční (uložení, přenos) dat proti chybám atd.
* kódující a dekódující log. obvody s pamětí – tj. kodéry, dekodéry

***Bity***

* [bit]; 1 bit (binary digit - dvojková číslice) je základní a zároveň nejmenší jednotka informace
* poskytuje množství informace potřebné k rozhodnutí mezi dvěma možnostmi
* jednotka bit se označuje **b** a může nabývat pouze dvou hodnot - 0, 1
* můžeme ji považovat za informaci typu ano/ne, platí/neplatí

***Byte***

* [bajt]; tj. 8 bitů, označení B (tj. 1B = 8 b) – tj. jednotka informace o velikost 8 b

***DVOJK. SOUSTAVA***

***Obecně:***

* princip: snažíme se nějakou informaci/stav popsat skupinou 2 stavů (značených pomocí 2 symbolů –> jde o poziční zápis, tj. záleží na pozici daného symbolu vůči jiným symbolům) –> 0 a 1
* 1/0 reprezentují 2 stavy elektr. obvodu (1 – zapnuto, 0 – vypnuto)–> prochází/neprochází el. proud
* číslo zapsané v binární soustavě se nazývá binární číslo
* z praktického hlediska se užívá pro vyjádření čísla užívá 8, 16, a 32 bitů

***Kladná čísla***

* rozsah zobrazení pro n bitů <0, 2n-1-1>; pro n=8 <0, 127>
* tento interval <minimální nekladné, maximální nezáporné>
* přímý kód: začneme např. 8bitovým číslem (tím lze ve dvojkové soustavě zapsat celé číslo od 0–255, což odpovídá hodnotě 28 (to platí v případě, že zapisovaná čísla uvažujeme bez znaménka) – pokud bychom chtěli uvažovat i záporná čísla vyhradíme nejvyšší bit (úplně vlevo) –> pro zapsání samotného čísla nám zbyde jen 7 bitů, což odpovídá zobrazovanému rozsahu 128 (přičteme-li ještě nulu – je součástí každého kladného čísla), můžeme zapsat čísla v rozsahu od -128 do +127

***Binární záporná čísla***

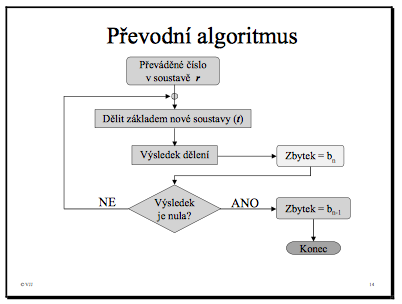
* vždy musíme vědět, kolikabitový je kód (kolik bitů máme k dispozici)
* ***přímý kód*** : rozsah zobrazení <-2n-1+1, -0>; pro n=8 <-127, -0>, <+0, +127>
  + 1. bit zleva je vyhrazen pro zápis znaménka (1 značí záporné číslo, 0 kladné číslo)
* ***inverzní kód*** (binární negace) – inverze bitů (jedničkový doplněk); kladná čísla jsou reprezentována normálním způsobem; pro zapsání záporného čísla se kladné číslo převede na negaci (inverzi) – 0 za 1 a 1 za 0; existují 2 reprezentace nuly: +0 a -0; 1. bit má význam znaménka –> logická negace bit po bitu
  + +5(10) = 0101(2)
  + -5(10) = 1010(2)
* ***dvojkový doplňkový kód*** (binární negace +1) – operace dvojkový doplněk = inverze bitů a přičtení jedničky; rozsah zobrazení <-2n-1, 2n-1-1>; pro n=8 <-128, 127>
  + kladná čísla jsou v něm stejně jako v přímém (či inverzním) kódu; záporná čísla jako v inverzním, ale ještě se přičítá 1–> 0 jen 1x:
* S celými čísly se díky metodě dvojkového doplňku (two’s complement) počítá stejně jako s přirozenými, jen využijeme **nejvyšší (první) bit jako znaménko** (nula je plus a jednička minus). Tento systém má jednu nulu a jedno tak trochu „divné“ číslo.
* Stále s 8 bity: nejmenší nezáporné číslo je 000000002 = 0, nejvyšší je 011111112 (127; máme o jeden bit méně prostoru). Záporná čísla začínají opět od „nuly“; ovšem už s nastaveným znaménkem: 100000002 a končí hodnotou 111111112. Mezi nejmenším a největším záporným číslem opět můžeme rozdělit 128 hodnot, jako jsme to udělali u kladných (0—127). Končíme −1, a proto začínáme −128.
* BITY A JEJICH POČET
* Této *nejmenší jednotce informace* (je nebo není) se říká ***1 bit*** (značka malé b). Tvůrci počítačů počítali, kolik různých znaků lze zakódovat pomocí kolika (jak dlouhého řetězce) nul a jedniček. Pokud by byla k dispozici pouze jedna nula či jednička, mohli byste zakódovat jen dva znaky. Např. písmeno A by bylo 0, B pak 1. BABA = 1010 => pro praktickou potřebu málo.
* BAJT
* Kolik nul a jedniček na jeden znak je tedy potřeba, aby bylo možné zakódovat celou abecedu, malá i velká písmena, číslice a ještě zbude rezerva? Tvůrci počítačů se shodli na 8 bitech (00000001). Počet různých kombinací z 8 nul a jedniček: 256 (28  -> obecně platí, že kombinací zakódovaných dvěma znaky je 2N, kde N je počet bitů). Této kombinací 8 nul a jedniček dali jméno **bajt** (z anglického byte) se značkou velké B. *Jeden bajt je tedy řetězec osmi bytů.*
* Protože bajtů se do počítače vejde hodně, používají se násobné jednotky:
* **1 kilobajt (KB)** je 1 024 bajtů (B)
* **1 megabajt (MB)** je 1 048 576 B, tedy 1 024 KB
* **1 gigabajt (GB)** je 1 073 741 824 B, tedy 1 048 576 KB, tedy 1 024 MB
* **1 terabajt (TB)** je 1 073 741 824 KB, tedy 1 048 576 MB, tedy 1 024GB
* Pro jednoduchost stačí většinou uvažovat zaokrouhlené hotnoty, např. KB jako tisíc bajtů, MB jako milion bajtů a GB jako miliardu bajtů nebo tisíc MB.

***Zobrazení čísla v pohyblivé řádové čárce***

* pro zobrazení velkých/desetinných čísel; používá se v moderních PC
* součástí IEEE 754 – tj. standard pro dvojkovou aritmetiku v pohyblivé řádové čárce (též nesprávně zvaná pohyblivá desetinná čárka) – jedná se o nejrozšířenější standard pro výpočty v pohybl. řád. čárce (definuje formáty pro reprezentaci čísel v pohybl. řád. čárce vč. záporné nuly atd.)
* standard IEEE ukládá čísla v plovoucí řád. čárce na 32 b; tj. jednoduchá přenost (1 bit vyhrazen pro znaménko, 8 b pro zakódování exponentu a 23 bitů pro zakódování mantisy, nebo na 64 bitech, tj. dvojitá přenost (1, 11, 52)
* čísla jsou zobrazovány ve tvaru: c = M\*zE
  + M – mantisa čísla, zobrazena v soustavě o základu z (desetinná část zlomku)
  + E – exponent,
  + z – základ, kt. se používá pro výpočet exponent. části
  + c – dané číslo

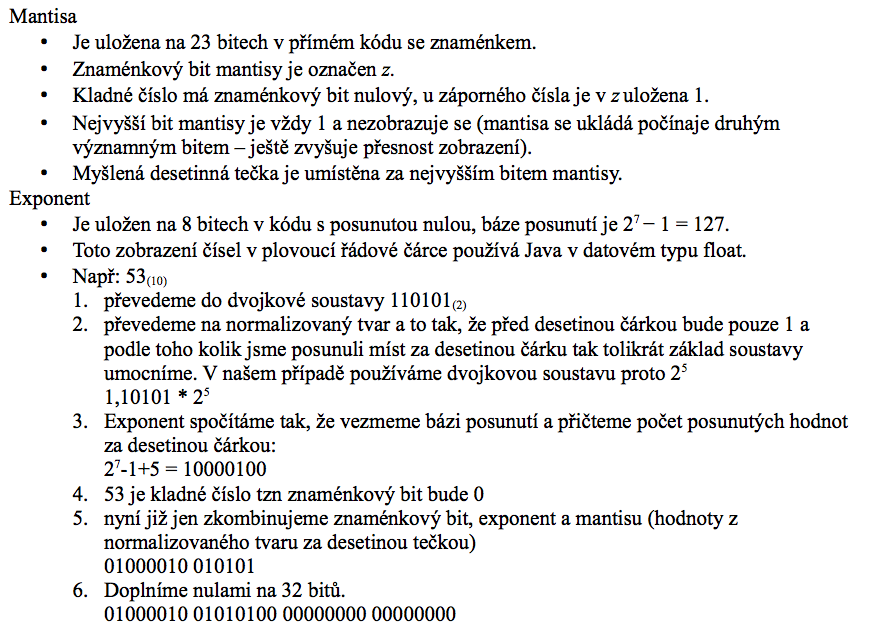
***Převod desetinného čísla do dvojk. soustavy***

* desetinné číslo/jeho část převedeme tak, že jej neustále násobíme dvěma a odečítáme a zapisujeme jedničku (pokud je výsledek >= 1, pak zapisujeme 0)
* převod končí, pokud nám zbyde 0, nebo pokud se začne generovat více číslic, než jsme schopni zobrazit

******

* ***při převodu celého čísla, dělíme původní číslo novým základem (1. zbytek – číslice s nejnižší váhou)***

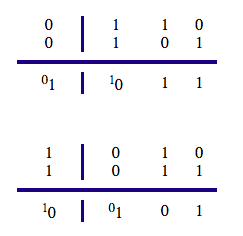
***Zobrazení čísla v jednoduché přesnosti***

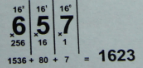
******

***Základní aritmetické operace***

* zákl. operace součet: platí stejné zásady jako při sčítání dekadických čísel s tím, že přenos jedničky do vyššího řádu je generován, nabyde-li součet hodnoty (nikoli 10), ale 2
* pozor na přetečení – číslo je příliš velké, než aby ho bylo možné vyjádřit
* součet v inverzním kódu – problém 2 nul, nutnost provádět tzv. kruhový přenos = přičtení přenosu z nejvyššího řádu k výsledku –> př.: -0+1

Příklad – sčítání v doplňkovém kódu (všechny bity se sčítají stejně, vč. znaménkového); vznikne-li přenos ze znaménk. bitu, tak se ingoruje; k přetečení dochází, pokud se přenos do znaménk. bitu nerovná přenosu ze znaménk. bitu



* odečítání: podobné jako sčítání, avšak pokud odečítáme 0-1, výsledek je 1 a 1 musíme odečíst ze sloupečku vlevo
* Při odčítání odečítané číslo (menšitel) převedeme na inverzní číslo (nuly zaměníme za jedničky a jedničky za nuly) a přičteme +1 – vytvoříme tzv. ***doplněk***. Tento doplněk pak přičteme k číslu, od kterého se mělo odečítat (k menšenci). (Menšenec – menšitel = rozdíl)
* Rozdíl je roven součtu (čísla, od kterého odečítáme a doplňku odečítaného čísla), ve kterém vynecháme
* přenos do nejvyššího řádu.
* ALU se nejlépe pracuje s čísly ve dvojkové (binární) soustavě s omezenou, pevnou délkou. Tuto posloupnost jedniček a nul nazýváme slovo (word).
* Rozsah hodnot závisí na délce slova (počtu bitů/bajtů) exponenciálně: historické 8bitové slovo pojme 28 = 256 různých hodnot (v přirozených číslech tedy rozsah 0—255), současné 64bitové unese 264 = hodně moc :-)
* Číslo 45 zapsané jako 8bitové slovo vypadá takto: 001011012, číslo 93: 010111012.
* Negace a sčítání (odčítání?)
* Člověk si opatří opak čísla 45 převrácením bitů a *přičtením jedničky*. V binárním zápise inverze (doplněk) cifer 001011012 vyjde 110100102 a po inkrementaci získáme výsledek 110100112 = −45.
* Na negaci −128 nemáme místo; při přičítání jedničky k inverzi 011111112 dochází k přetečení, kterého když si nevšimneme, získáváme opět −128.
* Součet 93 a −45 získáme klasickým sčítáním pod sebou, jako známe pro desítkovou soustavu, jen přenášíme dvojku místo desítky.
* 010111012 93
* 110100112 −45
* 001100002 48
* Čísla s plovoucí řádovou čárkou (floating-point numbers)
* Hodnoty příliš velké nebo neceločíselné se ukládají způsobem, který kopíruje tzv. vědeckou notaci. Například palec v centimetrech (2,54) by se zapsal jako 254 ⋅ 10−2. Při ukládání v počítači stačí rozdělit přidělené místo (např. 64 bitů) na dvě (nestejné) části, do první (větší) zapsat mantisu 254 a do menší části za ní exponent −2.
* DIGITALIZACE PODROBĚJI, POTŘEBNÝ POČET BITŮ
* Digitalizace = vzorkování původního (analogového) signálů a jeho záznam pomocí toku nul a jedniček.
* V počítačových zařízeních se bity používají k záznamu všeho – textů, zvuků i videa. Jak již bylo zmíněno, zápisem pomocí dvou znaků o délce N pozic můžeme zakódovat 2N možností, kde N je počet bitů. Třemi bity můžeme zapsat 23, tedy 8 možností, čtyřmi bity pak 16 možností, pěti 32 možností atd. Ve výpočetní technice se nejčastěji setkáme s těmito hodnotami:
* **8 bitů** (tedy 1 B) umožňuje záznam 28 = 256 možností. Jeden B se dlouho používal pro záznam písmen v tzv. fontech True type, dnes používaná typová písma Open type většinou využívají 3 bajty na znak.
* **16 bitů** (tedy 2B) umožňuje záznam 216 = 65 536 možností. Dnes se tato hodnota používá jen výjimečně.
* **24 bitů** (tedy 3B) umožňuje záznam 225 = 16,8 milionů možností. 24bitová barevná hloubka se často používa při **záznamu obrázků** i **snímků videa**.
* **32 bitů** (4 B) nabízí cca 4,3 miliardy možností, používá se nejčastěji pro **kódování barev**.
* DVOJKOVÁ, DESÍTKOVÁ A ŠESTNÁCTKOVÁ SOUSTAVA
* Způsob kódování čísel pomocí dvou znaků se v matematice nazývá dvojková nebo také binární soustava. Při práci se setkáme se zápisem i v šestnáctkové (hexadecimální soustavě).
* V obvyklé **desítkové soustavě** se čísla zapisují pomocí deseti číslic (0..9). Např. číslo 1 623 je součtem 1 × 1 000 + 6 × 100 + 2 × 10 + 3 × 1. Číslo se tedy skládá z mocnin (řádů) deseti násobených jejich pozičními hodnotami.
* 
* Ve **dvojkové soustavě** bude stejné číslo vyjádřeno pomocí mocnin čísla 2 násobených opět jejich pozičními hodnotami, ty však nyní mohou nabývat pouze stavů, 0 a 1. Číslo 1 623 je vyjádřeno takto:
* 
* V **šestnáctkové soustavě** se používá 16 znaků, číslic 0..9 a písmena A..10, B..11, C..12, D..13, E..14, F..15. Číslo 1 623 je vyjádřeno takto:
* 
* Neboli 110010101112 v dvojkové soustavě je 162310 v desítkové soustavě a 65716 v šestnáctkové soustavě.
* ***Převod čísla z desítkové do dvojkové soustavy*** není složitý: stále číslo dělíme dvěma a zbytek po dělení zapisujeme zprava jako vyjádření čísla ve dvojkové soustavě. Např. 1623:2=811, zbytek 1 (zapíšeme 1), 811:2 = 405, zbytek 1 (zapíšeme 1); 405:2 = 202, zbytek 1 (111), 202:2=101, zbytek 0 (0111), 101:2=50, zbytek 1 (10111); 50:2=25, zbytek 0 (010111), 25:2=12, zbytek 1 (1010111), 12:2=6, zbytek 0 (01010111), 6:2=3, zbytek 0 (001010111), 3:2=1, zbytek 1 (1001010111), 1:2=0, zbytek 1 (11001010111), **162310 = 110010101112.**
* ***Převod čísla z desítkové do šestnáctkové soustavy*** probíhá stejně jako u dvojkové, pouze samozřejmě původní číslo dělíme 16 a opět zapisujeme zbytek po dělení zprava.
* **REPREZENTACE ZNAKŮ V PC, ZNAK. SADY A JEJICH KÓDOVÁNÍ**

***Obecně:***

* text – tj. posloupnost tisknutelných znaků (znaky různých abc – pro písmena, cifry, symboly, atd.) – alfanumerické znaky + speciální (netisknutelné) a řídíc znaky
* kódování znaků – tj. vnitřní reprezentace znaků v paměti PC pomocí čísel (jedná se o zobrazení písmen – pro lidí a čísel – pro PC) –> PC rozumí pouze číslům
* ***plain text*** (čistý text) – tj. uložení textu do souboru ve formě, kt. obsahuje pouze zobrazitelné znaky (v otevřením v libovolném text. editoru) – HTML (jsou uloženy pouze jednotl. znaky bez formátovacích informací)
* ***jak se ukládá mezera/tabulátor/nový řádek –***
* ***znak. sada –*** tj. zobrazení (fce) ze znaků do čísel (tj. množina znaků v kódovací tabulce)
  + dohodnutá pravidla pro kódování a dekódování z/do binární soustavy
  + jedná se o množinu znaků, kde má každý znak přiřazený číselný kód (vymezuje nám repertoár znaků, kt. je možné v textu používat)
  + nejznámější: znak. sada ASCII, kt. definovala 128 znaků (zejm. pro aj)
  + abychom mohli text v nějaké znak. sadě uložit do souboru/přenést po sítí –> musíme jednotl. kódy znaků převést na posloupnost bajtů (bajty jsou zákl. jednotky info, do kt. se ukládají info v souborech/posílají po síti)
  + Unicode – univerzální znak. sada (umožňuje v textu užití libovolných znaků, tedy míchat texty i zcela odlišných jazyků; vznik 90. léta)
* ***kódování –*** tj. způsob převodu číselného kódu znaku do posloupnosti bajtů; způsob uložení čísla (tj. převod dat do tvaru, jemuž PC rozumí – tj. kombinace bitů)
  + ***kódování UTF-8***: nejužívanější v prostředí internetu pro kódování textu v Unicode (1 znak se může zakódovat do proměnlivého počtu bajtů – od 1 až 4; je navržen tak, aby byl zpětně kompatibilní s ASCII –> máme-li text v Unicode obsahující znaky pouze z ASCII a zapíšeme jej v kódování UTF-8, vypadá výsledný soubor stejně jakoby byl rovnou vytvořen v ASCII (tj. díky tomu, že 1. 128 znaků Unicode je převzato z ASCII a UTF-8 je přímo kóduje jako odpovídající jednobajtovou hodnotu)
  + obsahuje-li text znaky Unicode s kódy většími než 128, jsou tyto znaky reprezentovány jako několik bajtů, mezi kt. se po jednotl. bitech rozdělí hodnota číselného kódu znaku
* existují různé znak. sady a různá kódování – společným základem je většinou 7bitové kódování ASCII
* ***kód*** – tj. předpi pro jednoznačné přiřazení určité kombinací bitů příslušnému znaku (kombinace bitů zobrazující znak – tj. kódové slovo) – kódovaná slova vyjadřující alfanumerické znaky musí mít nejméně 7 b (ptže 27 = 128 b)

***ASCII***

* American Standard Code for Information Interchange
* zákl. standard pro PC (1964)
* délka kódu – 7 b

***Vypracované***:

* Se znaky (písmeny, číslicemi, interpunkcí, mezerami, řídícími znaky atd.) pracujeme jako s čísly, každému přiřadíme systematicky jednu pozici v potřebném rozsahu. Např. americký standard ASCII si vystačil se 128 pozicemi (7 bity, osmý se používal při přenosu dat pro kontrolu), ale neobsahoval diakritiku. Jazyky s ideogramy naopak potřebují uložit tisíce znaků. Evropané se v mnoha svých standardech vešli buď do 7, nebo do 8 bitů.
* *Znakovými sadami myslíme přiřazení čísel znakům*, např. v ASCII se „e“ skrývá pod číslem 101, mezera má číslo 32, rovnítko 61 a zalomení řádku 10. V našem prostředí se používalo mnoho sad, ale stále přežívá microsoftí CP1250 (code page) a v mailech standard ISO-8859-2. Celosvětově se ale prosazuje obří **znaková sada** **Unicode** (nadmnožina ASCII), která zatím zahrnuje přes 110 000 znaků ze všech možných „abeced“.
* Kódováním dáváme číslu, které zastupuje znak, formu. 7- a 8bitové znakové sady stačí zakódovat do sekvence bajtů. Text v Unicode může sestávat z latinky spolu s rozsypaným čajem a výslovností v IPA. Existují kódování s pevným, nebo proměnlivým počtem bajtů. **Zastaralé kódování UCS-2** používá 16 bitů, zaznamenává proto pouze podmnožinu 65 536 znaků. **Populární kódování UTF-8** zapisuje každý znak 1 bajtem, pokud má znak číselnou hodnotu do 127 (a je v tomto kompatibilní s ASCII), anebo až 4 bajty, kde první má vždy hodnotu větší než 128. České znaky s háčky a čárkami se vejdou do 2 bajtů, valná většina čínštiny do 3, ve 4 jsou jen rarity.

**5. Databáze, jazyk SQL**

Databáze

***SQL*** (Structured Query Language)

* standardizovaný dotazovací jazyk používaný pro práci s daty v relačních databázích

**6. Značkovací jazyky – (X)HTML, XML, DTD, ...**

Značkovací jazyky

**7. Algoritmická složitost – lineární, polynomiální a exponenciální algoritmy, složitost algoritmu a složitost problému, třídy složitosti**

* **ZÁKL. POJMY Z ALGORITM. SLOŽITOSTI (čas., prostorová složitost algoritmu, asymptomická složitost a třídy složitostí, složitost algoritmu, složitost problému)**
* ***VÝPOČTOVÁ SLOŽITOST***
* tj. náročnost výpočtu na počet kroků výpočtu nebo na kapacitu paměti
* 2 druhy: tzv. časová složitost a tzv. paměťová složitost

***Časová složitost***

* tj. počet kroků, kt. musí algoritmus k řešení úlohy provést (počet kroků/přechodů TS provedenýc od počátku do konce)
* zajímá nás doba běhu algoritmu v nejhorším možném případě meřená v závislosti na velikosti vstupních dat (důležité je chování algoritmu pro velká vstupní data)
* nejčastěji: O(N) – lineární složitost, O(N2) – kvadratická, O(N3) – kubická, O(log N) – logaritmická, O(2N) – exponenciální a O(1) – konstantní
* nejhorší – exponenciální
* algoritmy se složitost O(Nk) – k je pevná konstantní –> tj. polynomiální a jsou chápány jako efektivní
* čas běhu algoritmu t(x) pro vstup x měříme jako sumu instrukcí, kt. program provedl při zpracování vstupu x (pokud se pro daný vstup program nezastaví: t(x) = + znak pro nekonečno
* lze ji měřit pouze zobecněným měřením, kde mírou složitosti je fce, kt. rozměru vstupu přiřazuje čas (resp. počet operací nutných pro provedení výpočtu)
  + rozměr vstupu – tj. vstupní data úlohy/rozsah dat. základny, s kt. program pracuje (případně oboje); jedná se o údaj charakterizovaný přirozeným číslem

***Paměťová složitost (zpravidla název prostorová složitost)***

* tj. počet znaků z konečné abc, kt. algoritmus potřebuje mít uložené v paměti PC (kolik paměti upotřebí) –> kolik nejvíce paměti (tzv. element. paměť. buněk) bude v daném algoritmu v každém okamžiku užito
  + v běžných program. jazycích (C, Pascal) – je možné považovat za tuto element. paměť. buňku např. prměnnou typu integer, float, byte (element. velikost rozhodně nemají např. pole či text. řetězce)
* meří paměť. náročnost – to vyjádříme pomocí fce f(N) v závislosti na velikost vstupu N
* počet buněk pásky TS požadovaný pro daný výpočet
* prostor běhu algoritmu s(x) je analogicky počet paměť. buněk použitých při výpočtu se vstupem x

***Složitost algoritmu (programu)***

* zákl. teoretický přístup vychází z faktu, že každý algoritmus je implementovaný TS –> zavedení TS nám umožňuje klasifikovat problémy (resp. fce) do 2 třídy:
  + problémy, jež nejsou algoritmicky ani částečně rozhodnutelné (resp. fce algoritmicky nevyčíslitelné)
  + problémy algoritmicky částečně rozhodnutelné
* analýzou složitosti algoritmu budeme chápat jako analýzu složitosti výpočtů příslušného TS, jejímž cílem je vyjádřit (kvantifikovat) požadované zdroje (čas, prostor) jako fci závisející na délce vstupního řetězce (zjednodušeně analýza algoritmu by nám měla ukázat, kolik času a místa potřebujeme k vyřešení problému)
* proč analyzovat: pro většinu problémů existuje více než jedno řešení (díky analýze –> výběr vhodného algoritmu)
* výhradně pracujeme s analýzou složitosti v nejhorším případě
* Pod složitostí algoritmu rozumíme dobu prováděného algoritmu => časovou složitost a rozsah použité operační paměti – paměťovou náročnost (složitost).
* rozumíme složitost konkrétní instance zadaného algoritmu (implementovaného v nějakém programovacím jazyce) Algoritmy pracující s lepší než exponenciální/faktoriálovou složitostí označujeme jako efektivní.
* složitost konkrétních algoritmů – složitost Turingových strojů
* nejhorší složitost (worst-case):
  + chování algoritmu pro nejhorší možnou množinu dat
  + odhad pesimistické složitosti je jednodušší nežli průměrné
  + je zajištěno, že nikdy nebude trvat déle
* složitost algoritmu má nejčastěji podobu fce velikosti zpracovávaných dat, udávané počtem položek N

***Složitost problému***

* tj. složitost optimálního algoritmu konkrétně řešícího zadaný problém
  + resp. vychází z teoretické představy, že máme k dispozici všechny algortimy řešící daný problém a porovnáváme jejich složitost (čas. složitost je rovna čas. složitosti nejrychlejšího z algoritmů, kt. problém řeší) – má tedy význam dolního odhadu složitosti S, kt. lze dosáhnout (v principu nám říká, že nemůže existovat algoritmus, kt. by řešil tento problém s menší složitostí je S a zároveň říká, že existuje algoritmus řešící problém se složitostí S)

***Asymptotická složitost algoritmu***

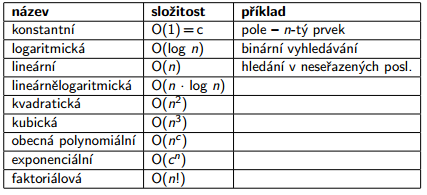
* ***definice:*** každému algoritmu lze přiřadit rostoucí fci zvanou asymptotická složitost, kt. charakterizuje početoperací algoritmu v závislosti na rostoucím rozsahu vstupních dat (čím pomaleji tato fce roste, tím je algoritmus rychlejší)
* ***zjednodušení:*** obvykle namísto asymptotické složitosti algoritmu říkáme složitost algoritmu
* charakterizuje počet provedených operací v závislosti na velikosti dat (jedná se o růst funkcí v závislosti na růstu rozměru vstupu („lepší“ algoritmus je ten, jehož fce čas. složitosti má pomalejší rychlost růstu)
* jedná se o nejčastější hodnotící kritérium algoritmů –> jedná se o rozdělení algoritmů do tříd složitosti, u kt. platí, že od určité velikosti dat, je algoritmus dané třídy vždy pomalejší než algoritmus třídy předchozí
* jedná se o určité zjednodušení – aneb asymptotická složitost (zanedbáváme operace nezávislé na datech, apod.)
* O (omikron) – složitost vyjdřující nejhorší případ (worst-case); horní hranice čas. chování algoritmu – nejčastěji (vyjadřuje vlastně dobu, do kt. algoritmus určitě skončí – pokud skončí)
* Ω (omega) – dolní hranice čas. chování algoritmu
* Θ – vyjadřuje třídu čas. chování algoritmu – viz čas. složitost: lienární...
* zkoumá chování algoritmu na velkých datech (tj. nebere v potaz multiplikativní a aditivní konstanty), pouze zařazuje algoritmy do “kategorií” podle jejich skutečné čas. složitosti

Od nich:

* snaha vyjádřit složitost jako funkci vstupu
* říct, jak roste složitost algoritmu vzhledem k rostoucímu vstupu
* složitost algoritmu udává, jak je daný algoritmus rychlý (kolik provede elementárních operací) vzhledem k množině vstupních dat
* můžeme uvažovat:
* složitost v nejlepším případě
* složitost v průměrném případě
* složitost v nejhorším případě
* amortizovanou složitost: určuje časovou složitost algoritmu v sekvenci nejhorších možných vstupních dat – nevyužívá pravděpodobnosti, a proto je zaručená

*Asymptotická složitost* algoritmu charakterizuje počet provedených operací v závislosti na velikosti dat. Například pokud procházíme pole, pak složitost bude lineární (na každý prvek připadá konstantní množství operací), pokud jej ovšem řadíme například *bubble sortem*, pak složitost bude kvadratická (na n prvků bude připadat  operací).

**ZÁKLADNÍ SLOŽITOSTI**



Třída složitosti

*Třída složitosti* stanovuje obtížnost rozhodnutelnosti daného problému na *Turingově stroji*.

**Třída *P*** – obsahuje problémy rozhodnutelné v polynomiálním čase.

* Má daný *graf* kostru o velikosti maximálně *k*?
* Existuje v daném acyklickém grafu mezi uzly *a* a *b* cesta, jejíž délka je nejvýše *k*?

**Třída *NP*** – obsahuje problémy, které jsou rozhodnutelné pomocí nedeterministického Turingova stroje v polynomiálním čase – tzn. jsme schopni ověřit jejich řešení v polynomiálním čase.

* Lze daný graf obarvit maximálně *k* barvami?
* Existuje v daném grafu hamiltonovská kružnice?
* Existuje v daném grafu klika o alespoň *k* vrcholech?

**8. Formální jazyky a automaty – regulární jazyk, bezkontextový jazyk, konečný automat, formální gramatika, typy gramatik podle Chomského**

**9. Regulární výrazy, CQL**

**10. Umělá inteligence (jaké problémy řeší a jaké algoritmy používá)**