Počítače I

Literatura:
Prof. Ing. Jaroslav Jandoš, CSc.: Technické prostředky informačních systémů, VŠE v Praze, 2001.
Vladimír Mařík, Olga Štěpánková, Jiří Lažanský a kolektiv: Umělá inteligence (3), Academia Praha, 2001.
Pavel Satrapa: Pascal pro zelenáče, Neokortex Praha, 2000.
J. Glenn Brookshear: Computer Science: An Overview (8th Edition), Addison Wesley, 2004, ISBN: 0-321-26971-3.

1 Počítačová reprezentace dat

Základní druhy informací dle [TPIS, str. 23]

- INFORMACE
  - INSTRUKCE (řídící činnost určená funkcí a operandy, se kterými pracuje)
  - DATA (jsou zpracovávána)
    - NUMERICKÁ (čísla)
      - S PEVNOU ŘÁDOVOU ČÁRKOU
      - S POHYBLIVOU ŘÁDOVOU ČÁRKOU
    - NENUMERICKÁ
      - ZNAKOVÉ ŘETĚZCE
      - OSTATNÍ (například bitová mapa)

1.1 Úvod

Všechny informace v počítači jsou zakódovány v binární soustavě. To znamená, že jsou převedeny na posloupnost číslic, které mohou nabývat pouze dvou hodnot 0 nebo 1. Počítače reprezentují informace ve dvojkové (binární) soustavě čísel, protože jsou založeny na dvou odlišitelných stavech. Například nízké napětí reprezentuje 0 a vyšší napětí reprezentuje 1. Aby byla data přenosná mezi různými typy počítačů s různými operačními systémy, musí existovat pro vyjádření informací v počítačích standardy. Standard je to, co je dostatečně široce uznávané a používané.

1.2 Numerická data s pevnou řádovou čárkou, což jsou v podstatě celá čísla

Numerickými daty s pevnou řádovou čárkou se rozumí čísla s konstantním počtem desetinných míst. Je-li počet desetinných míst konstantní, může se informace o něm považovat za implicitní (skrytou, nevyslovenou, zde předem dohodnutou a proto dále neuváděnou), což znamená, že se nemusí uchovávat u každého čísla zvlášť. Problematika reprezentace těchto čísel a operací s nimi se v zásadě neliší pro různé počty desetinných míst. Proto si ji ukážeme na příkladu celých čísel. Celá čísla mohou být v počítači vyjádřena různě, ale pokud s nimi mají být prováděny výpočty, jsou některé reprezentace lepší než jiné. Nejzákladnější operací, kterou počítač provádí na číslech, je vedle jejich porovnávání sčítání. Popíšeme reprezentaci, která umožňuje uplatnit pro sčítání celých čísel ten nejjednodušší algoritmus. Z důvodu jednoduchosti algoritmu je tato reprezentace také v počítačích uplatňována nejčastěji. Celá čísla budeme v našem příkladu uchovávat v 8 bitech. Větší čísla se analogickým způsobem v počítačích uchovávají také v 16, 32 nebo 64 bitech. Pro výpočty v příkladech může posloužit převodní tabulka 1.

Tabulka 1: Dekadická, binární a hexadecimální soustava

10	2	16
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Úloha: 11 + 24 = 35
Převod čísel do dvojkové soustavy:

11	=	2	·	5	+	**1**	24	=	2	·	12	+	**0**
5	=	2	·	2	+	**1**	12	=	2	·	6	+	**0**
2	=	2	·	1	+	**0**	6	=	2	·	3	+	**0**
1	=	2	·	0	+	**1**	3	=	2	·	1	+	**1**
							1	=	2	·	0	+	**1**

$11_{10} = 0000\ 1011_2 = 0B_{16}$ $24_{10} = 0001\ 1000_2 = 18_{16}$

Čísla v šestnáctkové (hexadecimální) soustavě jsou vidět ve výpisech dat a používají se také pro zápis konstant do některých zdrojových textů počítačových programů. Jejich účelem je redukce vzniku chyb při zápisu lidmi.

11_{10} $0000\ 1011_2$ $0B_{16}$
\+ 24_{10} $0001\ 1000_2$ 18_{16}
= 35_{10} $0010\ 0011_2$ 23_{16} $35 = 0 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 32 + 2 + 1$
Kladná čísla jsou v počítači reprezentována a sčítána ve dvojkové soustavě.

Úloha: 24 – 11 = 13
Číslo, které má být odečteno, se převede na reprezentaci záporného čísla se stejnou absolutní hodnotou. Reprezentace záporných čísel se tvoří z reprezentace kladných čísel podle následujícího algoritmu: V čísle se změní hodnota všech bitů a k výsledku se přičte jednička.

11_{10} = $0000\ 1011_2$ $0B_{16}$
-11_{10} = $1111\ 0101_2$ $F5_{16}$

Tato reprezentace záporného čísla se nazývá vyjádření dvojkovým doplňkem (2's complement of a positive number).

Binární reprezentace čísel se sečtou:

24_{10} 0001 1000_2 18_{16}
– 11_{10} 1111 0101_2 $F5_{16}$
= 13_{10} 1 0000 1101_2 $0D_{16}$

Při tomto součtu se zaplnil 9. bit. Tento bit se ve výsledku ignoruje. Proč?
Dvojkový doplněk čísla $11 = 2^8 - 11 =$

$2^8{}_{10}$ 1 0000 0000_2
– 11_{10} 0000 1011_2
= reprezentace -11_{10} 1111 0101_2

Odčítání se provádí podle vzorce $24 - 2^8 + 2^8 - 11$. Mezivýsledek 1 0000 1101_2 je roven $24 + 2^8 - 11$. Zbývá tedy od něj odečíst 2^8, což je to samé, jako když se škrtne 9. bit.

Úloha: $11 - 24 = -13$

11_{10} 0000 1011_2 18_{16}
+ -24_{10} 1110 1000_2 $E8_{16}$
= -13_{10} 1111 0011_2 $F3_{16}$

Tentokrát se nám nezaplnil 9. bit. S výsledkem už nic neděláme. Proč?
Odčítání se provádí podle vzorce $11 - 2^8 + 2^8 - 24$. Mezivýsledek 1111 0011_2 je roven $11 + 2^8 - 24$. Mělo by se tedy ještě od něj odečíst 2^8. Není to třeba, protože výsledkem je záporné číslo, které musí být uloženo jako dvojkový doplněk, neboli
$11 + 2^8 - 24 = 2^8 + 11 - 24 = 2^8 - (24 - 11) = 2^8 - 13$ = dvojkový doplněk čísla 13, který vytváří reprezentaci čísla –13.
Znaménko výsledku lze detekovat podle 8. bitu a tudíž místo pro 9. bit v paměti lze ušetřit. Je-li 8. bit rovný 0, je číslo kladné, je-li rovný 1, je číslo záporné.

Úloha: $-11 - 24 = -35$

-11_{10} 1111 0101_2 $F5_{16}$
+ -24_{10} 1110 1000_2 $E8_{16}$
= -35_{10} 1 1101 1101_2 DD_{16}

Z výsledku odstraníme 9. bit. Proč?
Výpočet se provádí podle vzorce $2^8 - 11 + 2^8 - 24 - 2^8 - 2^8$. Mezivýsledek 1 1101 1101_2 je roven $2^8 - 11 + 2^8 - 24$. Mělo by se tedy ještě od něj odečíst dvakrát 2^8. Správně se musí odečíst 2^8 jen jednou, protože to druhé je nutné pro reprezentaci záporného výsledku.
$2^8 + 2^8 - (11 + 24) - 2^8 = 2^8 - 35$ = dvojkový doplněk čísla 35 = reprezentace čísla (–35).

Rozsah čísel se znaménky, které se dají uchovat v 8 bitech pomocí dvojkového doplňku:

Maximální číslo	127_{10} =	$2^7 - 1$	= 0111 1111_2 =	$7F_{16}$
1	1_{10} =	2^0	= 0000 0001_2 =	01_{16}
Nula	0_{10} =	$2^0 - 1$	= 0000 0000_2 =	00_{16}
–1	-1_{10} =	$2^8 - 1$	= 1111 1111_2 =	FF_{16}
Minimální číslo	-128_{10} =	2^7	= 1000 0000_2 =	80_{16}

Pokud výsledek přesáhne mezní hodnoty uchovatelné v 8 bitech, dojde k takzvanému přetečení (overflow) nebo podtečení (underflow).
K přetečení může dojít, sčítáme-li dvě kladná čísla. Detekuje se tak, že 8. bit výsledku je roven 1, tj. značí záporné číslo.
K podtečení může dojít, sčítáme-li dvě záporná čísla. Detekuje se tak, že 8. bit výsledku je roven 0, tj. značí kladné číslo.

1.2.1 Poučení

Přetečení nebo podtečení lze někdy zabránit vhodnou algoritmizací. Například, jsou-li čísla *a, b, c* kladná a provádíme výpočet $a + b - c$, může (ale nemusí) přetečení zabránit změna pořadí operandů $a - c + b$ nebo závorky $a + (b - c)$.

1.3 Numerická data s pohyblivou řádovou čárkou neboli reálná čísla

Reprezentace čísel s pohyblivou řádovou čárkou, neboli takzvaných reálných čísel, je v mnoha počítačových systémech včetně počítačů typu PC s procesorem Intel, Macintosh a většiny počítačů s operačním systémem Unix určena IEEE standardem 754. (IEEE je zkratka pro "The Institute of Electrical and Electronic Engineers, Inc.") Tento standard vychází ze semilogaritmického tvaru čísla neboli vědecké notace čísla. Vědecká notace se skládá z mantisy a exponentu. K těmto dvěma údajům je třeba znát ještě číslo, které se má exponovat. Toto číslo však může být implicitně dohodnuté v rámci standardu a potom se nemusí uvádět u každého čísla zvlášť. IEEE standard stanovil, že základem pro exponent je číslo 2, což je ideální pro binární vyjádření čísel. Poslední věc, která musí být k jednoznačnému vyjádření čísla pomocí mantisy a exponentu ustálená, je počet míst v mantise před desetinnou čárkou. Sjednocení tohoto počtu pro různá čísla se nazývá normalizace mantisy. IEEE standard stanovil, že před čárkou musí být jedno místo.
Například:
$23{,}125_{10} = 2{,}3125_{10} \cdot 10^1{}_{10} = 10111{,}001_2 = 1{,}0111001_2 \cdot 2^4{}_{10} = 1{,}0111001_2 \cdot 10000_2$
$2{,}3125_{10} \cdot 10^1{}_{10}$ je normalizované vyjádření čísla $23{,}125_{10}$ se základem exponentu 10.
$1{,}0111001_2 \cdot 2^4{}_{10}$ je normalizované binární vyjádření čísla $23{,}125_{10}$ se základem exponentu 2. Mantisa = $1{,}0111001_2$ a exponent = 4.
$10111{,}001_2 \cdot 2^0{}_{10}$ je nenormalizované binární vyjádření čísla $23{,}125_{10}$.

IEEE standard určil pro ukládání reálných čísel do počítače 3 formáty popsané v tabulce 2.

Tabulka 2: Popis formátů pro reálná čísla určených IEEE standardem 754

Název formátu	Počet bitů Celkem	Znaménko	Exponent	Mantisa	Exponent			Rozsah hodnot
					Posunutí (bias)	Minimální	Maximální	
Jednoduchá přesnost (single precision)	32	1	8	23	127	-126	127	$\pm 10^{38,53}$
Dvojnásobná přesnost (double precision)	64	1	11	52	1023	-1022	1023	$\pm 10^{308,25}$
Rozšířená přesnost (extended precision)	80	Využívá se pouze vnitřně v procesoru.						

Na rozdíl od čísel s pevnou řádovou čárkou nejsou čísla s pohyblivou řádovou čárkou rovnoměrně rozprostřena mezi nejvyšší a nejnižší reprezentovatelnou možnou hodnotou.

http://www.dspguide.com/ch28/4.htm

Mezera mezi dvěma sousedními reprezentovatelnými čísly pro jednoduchou přesnost je přibližně desetmilionkrát menší než je hodnota těchto čísel, takže jsou velké mezery mezi velkými čísly a malé mezery v okolí nuly. Poměr „desetmilionkrát“ je roven podílu rozdílu mezi sousedními čísly a jednoho z těchto čísel bez ohledu na exponent, což je rozdíl čísel, která se liší o poslední bit v mantise, ve které jich je 23.

$$\frac{\left(2^{23}+2^{0}\right)\cdot 2^{E}-2^{23}\cdot 2^{E}}{2^{23}\cdot 2^{E}}=\frac{2^{0}}{2^{23}}=2^{-23}=10^{-23\log_{10}2}=10^{-6,9}$$

Absolutní hodnota exponentu čísla 10 ve výsledku výše uvedeného výpočtu je rovna počtu desetinných míst, která v desítkové soustavě můžeme brát jako platná s tím, že správnou hodnotu jich může mít i méně v závislosti na charakteru výpočtu, jehož je číslo výsledkem.

Ukážeme si převod čísla do paměti počítače a zpět pro jednoduchou přesnost. Dvojnásobná přesnost je tomu analogická. Výsledek můžeme ověřit na stránce http://babbage.cs.qc.cuny.edu/IEEE-754/.

Bity jednoho reálného čísla jsou v paměti počítače seřazeny tímto způsobem:

Z	Exponent								Mantisa																							
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32

Převeďte číslo –3,8125 do formátu dle IEEE standardu 754.

Číslo $3,8125_{10}$ převedeme do binární soustavy. Na jeho znaménku nezáleží.

Převedeme zvlášť část čísla před desetinnou čárkou a za ní.

Převod celé části čísla:

3 = 2 · 1 + **1**

1 = 2 · 0 + **1**

$3_{10} = 11_2$

Převod zlomkové části čísla:

0,8125 · 2 = **1**,625

0,625 · 2 = **1**,25

0,25 · 2 = **0**,5

0,5 · 2 = **1**,0

Vycházíme zde z rovnice čísla c, které je zlomkovou částí čísla v libovolné soustavě, pro jeho vyjádření v binární soustavě:

$c = k_{-1} \cdot 2^{-1} + k_{-2} \cdot 2^{-2} + k_{-3} \cdot 2^{-3} + k_{-4} \cdot 2^{-4} + \ldots$

$k_{-1}, k_{-2}, k_{-3}, k_{-4}, \ldots$ jsou bity zlomkové části binárního čísla, které mají hodnotu 0 nebo 1.

Násobíme-li tuto rovnici číslem 2, zjistíme hodnotu bitu k_{-1} z toho, jestli je před desetinnou čárkou výsledku 0 nebo 1, a potom jej můžeme z rovnice vyřadit a znovu rovnici násobit 2 a tak dále.

$2c = k_{-1} + k_{-2} \cdot 2^{-1} + k_{-3} \cdot 2^{-2} + k_{-4} \cdot 2^{-3} + \ldots$

$3,8125_{10} = 11,1101_2 = 1,11101_2 \cdot 2^{1}{}_{10}$

Číslo –3,8125 je záporné. Záporná čísla se zapisují s bitem pro znaménko rovným **1**.

Exponent o základu 2 = 1.

8 bitů pro exponent může reprezentovat jak kladné tak záporné celé číslo. Jeho reprezentace je jiná než u dat s pevnou řádovou čárkou. Ke skutečnému exponentu reprezentovaného čísla je připočteno takzvané posunutí a výsledek je potom převeden do binárního kódu. Z tabulky 2 lze vypočíst, že přípustné hodnoty exponentu s připočteným posunutím jsou 1 (= minimální exponent + posunutí) až 254 (= maximální exponent + posunutí). Do 8 bitů je však možno zapsat hodnoty 0 až 255. Hodnoty exponentu + posunutí rovné 0 nebo 255 se využívají pro reprezentaci speciálních hodnot nula, nekonečno a chybových stavů. Posunutí je pro jednoduchou přesnost rovno 127.

$1 + 127 = 128_{10} = 2^{7}{}_{10} = \mathbf{10000000}_2$

Normalizovaná mantisa je rovna 1,**11101**.

Všechna normalizovaná binární čísla kromě nuly mají bit před čárkou roven 1. Toto pravidlo se využilo k tomu, že bit před čárkou se v počítači neukládá a tím se ušetří jeden bit.

Teď již můžeme dát vše dohromady:

Z	Exponent								Mantisa																							
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
1	1	0	0	0	0	0	0	0	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C				0					7			4				0				0				0				0				

Obrácená úloha: Ve výpisu paměti je hexadecimální číslo C0D4 0000. Jaké reálné číslo reprezentuje?
Výpis převedeme na bity.

Z	Exponent								Mantisa																							
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
1	1	0	0	0	0	0	0	1	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C				0					D			4				0				0				0				0			

1. bit = 1, což znamená, že číslo je záporné.
Exponent + posunutí = $10000001_2 = 2^7 + 2^0 = 128 + 1 = 129_{10}$.
Exponent = 129 – 127 = 2.
Mantisa = $1{,}10101_2$. Je k ní přidán implicitní bit před čárkou.
Číslo v binární soustavě = $1{,}10101_2 \cdot 2^2_{10} = 110{,}101_2$.
Číslo v desítkové soustavě = $2^2 + 2^1 + 2^{-1} + 2^{-3} = 4 + 2 + 0{,}5 + 0{,}125 = 6{,}625$.
C0D4 0000 reprezentuje číslo $-6{,}625_{10}$.

Speciální hodnoty reálných čísel mají společné to, že pole pro exponent je celé zaplněno pouze jedničkami nebo pouze nulami.

Z	Exponent								Mantisa																							
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0				0					0			0				0				0				0				0			

= 0

Z	Exponent								Mantisa																							
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7				F					8			0				0				0				0				0			

= +∞

Z	Exponent								Mantisa																							
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
F				F					8			0				0				0				0				0			

= –∞

Tím nejsou všechny speciální hodnoty vyčerpány. Zbytek možných stavů speciálních hodnot náleží chybovým hodnotám.

1.3.1 Poučení

Výpočty s reálnými čísly zatěžují procesor podstatně více než výpočty s celými čísly. To znamená, že výpočet trvá déle a je zabráno i více operační paměti. Proto všude tam, kde je to možné, bychom měli používat pro proměnné v počítačových programech celočíselné datové typy. Porovnávání reálných čísel však složité není.
Všechna reálná čísla nemohou být v počítači reprezentována. Některá z toho důvodu, že jsou příliš velkého nebo příliš malého řádu tak, že pro jeho reprezentaci nestačí pole bitů pro exponent. Typičtější situací ale je, že číslo, například π, může být pouze aproximováno konečným počtem desetinných míst. Přesně, tj. pomocí konečného počtu desetinných míst, lze v binární soustavě reprezentovat pouze čísla vyjádřitelná ve formě $p/2^n$, kde p a n jsou celá čísla. Mnoho čísel v desítkové soustavě s konečným počtem desetinných míst při převodu do binární soustavy se stane číslem s nekonečným počtem míst za čárkou, jejichž vzor se periodicky opakuje.
Například:
0,1 · 2 = 0,2
0,2 · 2 = 0,4
0,4 · 2 = 0,8
0,8 · 2 = 1,6
0,6 · 2 = 1,2
0,2 · 2 = 0,4
$0{,}1_{10} = 0{,}00011001100110011...$
Při výpočtech, které v desítkové soustavě dají přesný výsledek, se můžeme u výstupu z počítače setkat se zaokrouhlovací chybou. Z toho plyne také to, že bychom neměli využít všechny číslice výsledku z počítače. Jednoduchá přesnost má výstup o 6 platných číslicích, dvojnásobná přesnost má výstup o 15 platných číslicích. Při používání dvojnásobné přesnosti se doporučuje spoléhat se z 15 číslic jen na 9 až 10 prvních v závislosti na kvalitě použitého algoritmu. V žádném případě nelze čekat správnou hodnotu 15. číslice.
V programech bychom nikdy neměli testovat rovnost dvou reálných čísel způsobem "IF $a = b$ THEN...".
Doporučenou technikou je stanovit si mez přesnosti a hodnoty, které se od sebe liší méně, než je stanovená mez, považovat za shodné, například "IF abs($a – b$) < 0.00001 THEN..." [Pascal, str. 49]
Operace typu násobení/dělení mohou způsobit menší zaokrouhlovací chyby než operce typu sčítání/odčítání.
Při násobení se vynásobí mantisy a sečtou exponenty: $m_1 \cdot 2^{e_1} \cdot m_2 \cdot 2^{e_2} = m_1 \cdot m_2 \cdot 2^{e_1 + e_2}$.
Při sčítání se musí čísla převést do tvaru se stejnými exponenty: $m_1 \cdot 2^{e_1} + m_2 \cdot 2^{e_2} = m_1 \cdot 2^{e_1 - e_2} \cdot 2^{e_2} + m_2 \cdot 2^{e_2}$.
Při sčítání se číslo s nižším řádem musí denormalizovat neboli převést do tvaru $m_1 \cdot 2^{e_1 - e_2} \cdot 2^{e_2}$.

Při tom se bity v mantise m_1 posunou o $e_2 - e_1$ míst dozadu a tím $e_2 - e_1$ bitů z mantisy m_1 vypadne.
Čísla, která se liší o 6 řádů v desítkové soustavě, již nemá smysl v jednoduché přesnosti sčítat.
Zajímavou zkušenost mají s tímto jevem matematičtí lingvisté. Při sčítání frekvencí slov (= četnost slova / počet všech slov) přišli na to, že frekvence musí nejdříve srovnat podle velikosti a potom sčítat od těch nejmenších. Naprostá většina slov má minimální frekvenci, ale existují slova (předložky, spojky) s velkou frekvencí. Sčítají-li se hodnoty od těch nejmenších, jejich kumulace může vytvořit mezisoučet, který je již řádově srovnatelný s nejvyššími hodnotami.
Peněžní částky jsou typicky ve formě „celá část + 2 desetinná místa“. Pro jejich přesnou reprezentaci jsou v některých programovacích jazycích k dispozici datové typy „Currency“ a „Decimal“. Datový typ Currency je celočíselný a číslo se dvěma číslicemi za desetinou čárkou je v něm reprezentováno tak, že je vynásobeno 100, takže vznikne celé číslo. Datový typ Decimal má pohyblivou řádovou čárku a je vnitřně pomocí speciálních algoritmů reprezentován pomocí základu 10, takže všechna čísla v desítkové soustavě, která mají konečný počet desetinných míst (v rámci limitu pro daný typ Decimal), jsou reprezentovány bez ztráty přesnosti. Operace s datovým typem Decimal jsou však výrazně pomalejší než operace s ostatními datovými typy pro čísla s pohyblivou řádovou čárkou. Je možné se obejít i bez těchto speciálních datových typů tak, že peněžní částky se budou už od začátku uchovávat v dostatečně velkém celočíselném datovém typu vynásobeny 100 podobně, jako je tomu u datového typu Currency.
V počítači je možné reprezentovat i čísla s libovolným konečným počtem číslic, pokud se vejdou do paměti. Aplikacemi jsou výpočty patřící do teoretické matematiky nebo asymetrické šifrování využívané při zabezpečené komunikaci na Internetu, při kterém se počítá s celými čísly o stovkách číslic.

1.4 Nenumerická data – znaky a jejich řetězce [TPIS, str. 27]

Znak je písmeno, číslice, nebo jiný symbol. Každý znak má svůj jedinečný kód, pod kterým je v počítači uložen. Kód může být 8-bitový (v něm lze vyjádřit $2^8 = 256$ znaků), 16-bitový, nebo i 32-bitový. Posloupnost znaků se nazývá znakový řetězec (string) a v počítači je reprezentován posloupností kódů jednotlivých znaků v něm. Existuje několik různých standardů, které znakům přiřazují kódy.
V roce 1968 vznikl kód ASCII (American Standard Code for Information Interchange). Tento kód je 7-bitový a vycházejí z něj pozdější standardy, které mají s ním shodnou hodnotu kódů pro číslice a anglickou abecedu. Během 80. let 20. století vznikla v Evropě množina 8-bitových kódů, které do ASCII přidávají znaky národních abeced. Pro češtinu je nejlepší standard ISO 8859-2 označovaný Latin 2 obsahující znaky češtiny, slovenštiny a němčiny.
Firma Microsoft vytvořila pro operační systémy počítačů typu PC kódové tabulky označované jako kódové stránky. Pro češtinu se užívá v operačním systému MS-DOS kódová stránka 852 (PC Latin 2) a v operačním systému MS-Windows 8-bitová kódová stránka 1250 vycházející z ISO 8859-2.
V roce 1991 vyšel první standard Unicode, který používá variabilní počet (1 až 4) bytů pro kódování znaků a je zpětně kompatibilní s ASCII. Zatímco předtím různé země používaly různé kódové tabulky, Unicode je univerzální (vejde se do něj 1 114 112 znaků, které mohou patřit do mnoha národních abeced) a proto se používá v moderních nástrojích pro vývoj softwaru, který je nutné lokalizovat do různých národních prostředí.
Složitá problematika kódování znaků je vysvětlena například ve zdroji Pavel Herout: Přehled kódování, 2004.
Text je v počítači uložen jako posloupnost znaků neboli znakový řetězec. U znakového řetězce je třeba kromě vlastních znaků uložit i informaci o délce řetězce. Délka řetězce je uložena s daným řetězcem buďto jako celé číslo, jehož maximální hodnota je i maximální délkou řetězce, nebo je řetězec ukončen speciálním znakem zpravidla se všemi bity rovnými nule, a potom je délka řetězce omezena velikostí dostupné paměti.

1.4.1 Poučení

Chceme-li, aby se bez problému mohl používat soubor, adresář, adresa webové stránky, nebo třeba heslo na všech operačních systémech, musíme jej zapsat pomocí znaků, které byly součástí nejstaršího standardu ASCII z roku 1968. Názvy souborů mají pro konkrétní operační systém ještě některé znaky z této množiny zakázány, protože v daném operačním systému mají nějakou funkci. Platí to typicky pro soubory, které chceme dát ke stažení na web, soubory, které jsou přílohami e-mailu, nebo pro soubory, které jsou součástí takzvaného projektu, což je adresářová struktura s automaticky generovanými soubory, kterou vytvářejí například vývojová prostředí pro tvorbu softwaru. Kromě znaků s diakritikou v takových případech způsobují problémy i mezery, takže víceslovné názvy je vhodné spojovat například podtržítkem (ASCII znak s kódem 95).

2 Algoritmy

http://courses.cs.vt.edu/~cs1104/Algorithms/TOC.html

2.1 Tvorba algoritmů

http://courses.cs.vt.edu/~cs1104/Introduction/Chapter1.020.htm
Věda o počítačích neboli informatika (computer science) je vědou o algoritmech.
http://courses.cs.vt.edu/~cs1104/Introduction/lees.law.html
http://courses.cs.vt.edu/~cs1104/Introduction/Chapter1.030.htm
Alternativní definice informatiky je to, že to je věda o informacích, jejich reprezentaci, organizaci, manipulaci a transformaci za určitým cílem, a tato manipulace a transformace je prováděna ve výpočtech, které jsou určeny algoritmy.
Počítače se používají k tomu, aby nám pomohly řešit problémy. **Existuje několik metod řešení problémů**:
http://courses.cs.vt.edu/~cs1104/ProblemSolving/PS.010.html
http://courses.cs.vt.edu/~cs1104/ProblemSolving/TOC.html

Výčet všech přípustných řešení nebo všech možností, jak to může dopadnout – je užitečné, když jich není mnoho. Pokud jich je mnoho, nazývá se tento přístup také řešení problému hrubou silou. Takový algoritmus spočívá většinou ve výčtu všech možností posloupnosti kroků a není považován za inteligentní. Mnoho logických problémů se řeší výčtem všech možností, například:
Krétský král Mínós měl 3 mladé vězně, kteří čekali na to, až budou dáni k sežrání dravému Mínótaurovi, slavnému napůl muži napůl býkovi. Král se rozhodl, že jim dá šanci se zachránit. Předvolal si vězně a řekl jim: „Dám každému z vás na hlavu červenou nebo modrou korunu, kterou nebudete vidět, a jejíž barvu určím hozením mincí. Potom vás rovnoměrně rozestavím do kruhu tak, abyste

viděli koruny svých dvou spoluvězňů. U každého z vás bude stát stráž, která vám utne hlavu, kdybyste si dávali nějaké signály. Vaším úkolem bude uhádnout barvu koruny, kterou máte na své hlavě. Máte také možnost hádání se vzdát. Když se hádání vzdáte všichni, jdete k Minótaurovi. Když někdo z vás uhodně špatně, jdete k Minótaurovi. Když někdo z vás pošle signál, ti, co zbudou naživu, jdou k Minótaurovi. Teď si asi myslíte, že máte poloviční šanci zůstat naživu, když určíte jednoho z vás, kdo bude hádat. Ale, jestli jste chytří, můžete vymyslet strategii, která vám zvýší šance na 3 ku 4. Strategií je pravidlo pro vyřčení vašeho výroku (červená, modrá, vzdávám), které každý z vás musí dodržet.“

http://courses.cs.vt.edu/~cs1104/ProblemSolving/Minotaur/Minotaur.html
http://courses.cs.vt.edu/~cs1104/ProblemSolving/PS.020.html

Redukce na jednodušší problém – například při třídění je základní úlohou výměna a při hledání je základní úlohou porovnání a uložení do paměti.

http://courses.cs.vt.edu/~cs1104/ProblemSolving/PS.030.html

Nalezení příbuzného problému z jiné oblasti nebo více možných algoritmů pro řešení problému – Následující anekdota ze života Johna von Neumanna naznačuje, že řešení některých úloh, které člověka napadne jako první, nemusí být to nejlepší: *John von Neumann jednoho večera večeřel s přáteli. Jeden z nich mu předložil problém o mouše a vlacích. Úloha zní takto: Představte si dva vlaky jedoucí rychlostí 20 mil za hodinu po jedné koleji naproti sobě. Když jsou vlaky 20 mil od sebe, z jednoho vlaku vylétne moucha a letí rychlostí 60 mil za hodinu naproti druhému vlaku. Tam se okamžitě obrátí a letí zpátky k prvnímu vlaku. To se opakuje, dokud se vlaky nesrazí a mouchu nerozmáčknou. Po jak dlouhé dráze moucha letěla? John von Neumann odpověděl okamžitě. „Jak jsi to udělal?“, zeptal se přítel. „Jednoduše“, odpověděl von Neumann, „sečetl jsem nekonečnou posloupnost“, a pokračoval ve večeři.*

http://courses.cs.vt.edu/~cs1104/ProblemSolving/PS.040.html

Rozděl a panuj – rozděl problém na snáze řešitelné podproblémy. Většina problémů se skládá z podproblémů, které již někdy byly vyřešeny. Problémy lze řešit v týmech, jejichž členové řeší podproblémy nezávisle na sobě.

http://courses.cs.vt.edu/~cs1104/ProblemSolving/PS.050.html
http://courses.cs.vt.edu/~cs1104/ProblemSolving/PS.057.html

Nalezení přibližného řešení – Pro některé problémy neexistuje algoritmus, který by zaručeně vedl k nejlepšímu možnému řešení, protože možných postupů řešení je tolik, že jejich výčet je v rozumném čase nemožný. Potom se musí možná řešení hledat vyhledávacími heuristickými metodami ve stromu variant posloupností kroků, jak lze při řešení postupovat. Tento typ problémů se nazývá „těžký“ (hard), v praxi se musí řešit velice často (rozvrhy, jízdní řády) a nalézání inteligentních algoritmů, které jej umí řešit alespoň ve většině případů, je umění informatiky. **Heuristika** je soubor pravidel, podle kterých se dá k cíli postupovat rychleji, ovšem bez záruky, že to bude fungovat vždy.

http://courses.cs.vt.edu/~cs1104/ProblemSolving/PS.060.html
http://courses.cs.vt.edu/~cs1104/ProblemSolving/PS.069.html

Problémy se řeší pomocí algoritmů postupem daným body 1 až 6.

http://courses.cs.vt.edu/~cs1104/Algorithms/algorithm_1.html

1. **Přesný popis a pochopení problému a žádoucího výstupu.**
2. **Určení metody řešení problému** – většinou technikou „Rozděl a panuj“.
3. **Jednoznačný zápis algoritmu** v nějakém programovacím jazyce.
4. **Verifikace** – důkaz správnosti. U mnoha problémů je dost obtížné ho pořídit. Je třeba testovat, jestli algoritmus správně zpracovává různá vstupní data.
5. **Implementace** [UI3, str. 33] je způsob, jak zajistit, aby daný softwarový program řídil reálný průběh příslušného výpočetního procesu v daném typu hardware. Implementace tak určuje vztah mezi SW a HW. Algoritmy samy o sobě jsou nezávislé na HW. Jeden algoritmus může mít více implementací.
6. **Testování** – ověření v praxi, které může vést k návratu do některé z předchozích fází řešení.

http://courses.cs.vt.edu/~cs1104/Algorithms/algorithm_2.html

Při návrhu algoritmů se většinou nepostupuje lineárně v daném sledu kroků 1 až 6 ale iterativně v kruzích, kdy je řešení postupně zlepšováno.

http://courses.cs.vt.edu/~cs1104/Algorithms/algorithm_3.html
http://courses.cs.vt.edu/~cs1104/Algorithms/algorithm_6.html

V algoritmech účinkují proměnné a identifikátory.

Proměnné jsou objekty, kterým mohou být přiřazeny různé hodnoty. V informatice reprezentují data. V matematice narozdíl od algoritmů je obsah proměnné v uvažovaném matematickém vzorci neměnný.

Identifikátory jsou jména objektů. Označují proměnné, datové typy, funkce a procedury.

Algoritmy jsou složeny z posloupnosti příkazů neboli kroků. Každý algoritmus má mezi svými příkazy **přiřazení**, **vstup** a **výstup**.

Posloupnost příkazů je určována **řídícími strukturami**, kterými jsou podmíněné příkazy a cykly.

Podmíněné příkazy:

IF podmínka s hodnotou „true“ nebo „false“
THEN blok příkazů, které se mají vykonat, pokud podmínka platí (má hodnotu „true“),
ELSE blok příkazů, které se mají vykonat, pokud podmínka neplatí (má hodnotu „false“).

Cykly mohou mít dvě varianty „repeat“ a „while“. Teoreticky by stačil na zápis všech algoritmů libovolný jeden z nich. Pro některé algoritmy je jednodušší použít „repeat“ cyklus a pro jiné zase „while“ cyklus. Pro některé algoritmy je nejjednodušší použít cyklus s předem daným počtem průchodů zvaný „for-cyklus“.

REPEAT blok příkazů UNTIL podmínka platí. Česky: „Opakuj blok příkazů, dokud nezačne platit podmínka.“

WHILE platí podmínka DO blok příkazů. Česky: „Dokud platí podmínka, opakuj blok příkazů.“

Pro řídící struktury je nezbytné sdružování příkazů do skupin neboli **bloků**, aby bylo jasné, které příkazy se mají provést za určité podmínky nebo které příkazy se mají opakovat v cyklu. Algoritmy lze zapisovat různými způsoby. Čím méně při tom používáme druhů slov, tím jednoznačnější je výsledek. Slova, která se dostala do existujících programovacích jazyků se nazývají **klíčová slova** (**key words**). Jsou to například IF, THEN, ELSE, REPEAT, UNTIL, WHILE, DO. Programovací jazyk Pascal obsahuje přibližně 40 klíčových slov. [Pascal, str. 23]

http://courses.cs.vt.edu/~cs1104/Algorithms/algorithm_13.htm

2.2 Efektivita algoritmů

Důležitými parametry algoritmů z hlediska jejich užití v počítačích je čas spotřebovaný na výpočet a velikost potřebné operační paměti. To poukazuje na další rozdíl mezi algoritmy v matematice a informatice. V matematice stačí, když lze dokázat, že nějaký algoritmus vede k řešení. V informatice jsou použitelné pouze takové algoritmy, které v přijatelném čase najdou řešení za podmínek daných stavem technologie určující rychlost procesorů a velikost pamětí.
http://courses.cs.vt.edu/~cs1104/Efficiency/Chapter3.010.htm
Informatici musí často porovnávat algoritmy z hlediska, kolik potřebují paměti a jak jsou rychlé. Praktický způsob takového porovnání se nazývá **benchmarking**. Algoritmy se pustí na stejná data na stejném počítači a změří se čas na jejich práci a paměťové nároky. Naměřené hodnoty budou záviset na určitém programu napsaném v určitém jazyce zkomplikovaném určitým překladačem na určitém počítači a na určitých vstupních datech. Jinými slovy, výsledek bude příliš specifický. Teoretická informatika potřebuje vědět, jak bude určitý algoritmus pracovat na libovolném hardwaru. Obtížnější než určení nezbytné velikosti operační paměti pro výpočet je určení času, který bude výpočet trvat. Tento čas je určen buďto počtem všech příkazů, které se musí provést, nebo počtem časově náročnějších bloků příkazů, které se v algoritmu mohou opakovat. Čas nebudeme počítat v časových jednotkách, ale v počtu příkazů v závislosti na počtu vstupů (velikosti vstupních dat). Jiným názvem pro tento abstraktní čas je **efektivita algoritmu**.
http://courses.cs.vt.edu/~cs1104/Efficiency/Chapter3.040.htm
Nejčastěji užívanou mírou efektivity algoritmu je takzvaný nejhorší případ. Když lze očekávat, že vstupní data budou rozumná, budou mít například normální statistické rozdělení, může se použít průměrná časová náročnost.
http://courses.cs.vt.edu/~cs1104/Efficiency/Chapter3.060.htm

2.2.1 Výpočet počtu kroků algoritmu pro nalezení nevětší položky:

http://courses.cs.vt.edu/~cs1104/Efficiency/Chapter3.070.htm

1. Čti počet položek n
2. Čti položky a_1 až a_n
3. *Největší* = a_1
4. *IndexNejvětšího* = 1
5. i = 2
6. WHILE i <= n DO
7. BEGIN IF a_i > *Největší* THEN DO
8. BEGIN *Největší* = a_i
9. *IndexNejvětšího* = i
   END
10. i = i + 1
    END
11. Vypiš *Největší*, *IndexNejvětšího*

Nejhorší případ nastane, když jsou položky srovnány od nejmenší do největší.
Příkazy č. 1, 2, 3, 4, 5 a 11 se v programu provedou jen jednou. Tedy 6 příkazů tvoří konstantní část programu.
Příkazy č. 6, 7, 8, 9 a 10, tj. 5 příkazů, se v programu provede $(n - 1)$-krát.
Nejhorší případ = $6 + 5 \cdot (n - 1) = \mathbf{5n + 1}$ příkaz.
Nejlepší případ nastane, když jsou položky srovnány od největší do nejmenší.
Příkazy č. 6, 7 a 10, tj. 3 příkazy, se v programu provede $(n - 1)$-krát.
Příkazy č. 8 a 9, tj. 2 příkazy, se v programu neprovedu ani jednou.
http://courses.cs.vt.edu/~cs1104/Efficiency/Chapter3.080.htm
Nejlepší případ = $6 + 3 \cdot (n - 1) = \mathbf{3n + 3}$ příkazy.
Průměrný případ nastane, když polovina porovnání v příkazu 7 povede k vykonání příkazů 8 a 9.
Průměrný případ = $6 + 3 \cdot (n - 1) + 2 \cdot (n - 1) / 2 = 6 + 3n - 3 + n - 1 = \mathbf{4n + 2}$ příkazy.

2.2.2 Asymptotická analýza

Příkaz 2 by se dal rozepsat na n samostatných příkazů. Na tom však příliš nezáleží, když uvážíme, že v algoritmu se provádí různé příkazy, které trvají různou dobu. Pro posouzení efektivity algoritmu záleží pouze na těch příkazech, jejichž počet vykonání závisí na počtu vstupních dat. Asymptotická analýza předpokládá, že počet vstupních dat se blíží nekonečnu. Potom začne být čas na příkazy, které se v algoritmu vykonají pro libovolný počet dat v konstantním množství, vzhledem k celkovému času zanedbatelný. Z toho důvodu můžeme v rovnicích získaných analýzou algoritmů vynechat konstanty. Počet příkazů v mnoha algoritmech se dá vyjádřit nějakým mnohočlenem (polynomem), například $7n^4 + 3n^3 - n^2 + 5n - 1$. Když se n blíží nekonečnu, záleží nejvíc na největším členu v polynomu $7n^4$ a záleží víc na exponentu n než na koeficientu n. Výsledek asymptotické analýzy pro ukázkový polynom by byl ten, že algoritmus je úměrný (nebo že má řádovou hodnotu) n^4. Formálně se to zapisuje $7n^4 + 3n^3 - n^2 + 5n - 1 = O(n^4)$ nebo $\Theta(n^4)$. Znamená to, že když se velikost vstupních dat zdvojnásobí, časová složitost algoritmu vzroste 2^4-krát. Exaktněji je to vysvětleno v [UI3, str. 262 – 265].
http://courses.cs.vt.edu/~cs1104/Efficiency/Chapter3.110.htm
http://courses.cs.vt.edu/~cs1104/Efficiency/Chapter3.120.htm

2.2.3 Porovnání efektivity 2 různých algoritmů pro vyhledávání

Máme seznam n jmen seřazených vzestupně podle abecedy a máme v něm najít jedno jméno. Efektivita algoritmu je dána počtem porovnání jmen, která budou muset být při tom vykonána.

2.2.3.1 Sekvenční vyhledávání (sequential search)

Seznam se prohledává od začátku položka po položce, dokud se jméno nenalezne nebo dokud se nedojde na konec seznamu.
http://courses.cs.vt.edu/~cs1104/Efficiency/Chapter3.090.htm
Nejhorší případ = ***n*** porovnání, tj. jméno je až na konci seznamu, nebo v něm není.
Nejlepší případ = **1** porovnání, tj. jméno je na začátku seznamu.

Průměrný případ = ***n*/2**.

2.2.3.2 Binární vyhledávání (binary search)

Předpokladem jeho uplatnění je to, že prohledávaný seznam je setříděn. Algoritmus porovná se jménem položku uprostřed seznamu, pokud to není ta hledaná, rozhodne, jestli má hledat před ní nebo za ní, a v příslušné půlce seznamu se chová stejně, jako předtím v celém seznamu.

http://courses.cs.vt.edu/~cs1104/Efficiency/Chapter3.180.htm
http://courses.cs.vt.edu/~cs1104/Efficiency/Chapter3.210.htm

Nejhorší případ = $\log_2 n$ porovnání, protože seznam můžeme dělit tolikrát, dokud v něm zbude alespoň jedna položka. Tedy $n/2^x = 1$ a odtud $n = 2^x$ a odtud $x = \log_2 n$.

Nejlepší případ = **1** porovnání, tj. jméno je přesně uprostřed seznamu.

Průměrný případ závisí na datech. Bylo by možné jej zjistit statistickým zkoumáním.

2.2.4 Porovnání efektivity třídících algoritmů

Máme seznam *n* jmen nebo čísel v náhodném pořadí. Máme změnit pořadí položek seznamu tak, aby byla jména seřazena abecedně nebo čísla podle hodnot. Třídění (řazení) je jeden z nejdůležitějších problémů informatiky a existuje pro něj mnoho algoritmů.

http://courses.cs.vt.edu/~cs1104/Efficiency/Chapter3.230.htm

2.2.4.1 Řazení přímým výběrem (selection sort)

Nalezneme největší položku seznamu a přesuneme ji na jeho konec (obvykle výměnou za položku z konce). Seznam zkrátíme o poslední položku seznamu a v jeho zbytku opět nalezneme největší položku a přesuneme ji na konec. To děláme tak dlouho, dokud se seznam nezkrátí na jedinou položku.

http://courses.cs.vt.edu/~cs1104/Efficiency/Chapter3.240.htm

2.2.4.2 Bublinkové řazení (bubble sort)

Tento algoritmus opakovaně prochází seznam od konce k začátku a kdykoliv narazí na dvojici položek, z nichž první je větší než druhá, vymění je navzájem. Díky tomu po prvním průchodu zcela jistě nejmenší prvek „probublá“ až na první pozici seznamu, po druhém průchodu bude na svém místě druhý atd. Bublinkové řazení vyměňuje bezprostřední sousedy, přenáší tudíž prvky jen na krátké vzdálenosti. Důsledkem je velký počet přesunů položek a efektivita o něco nižší než u předchozího algoritmu. [Pascal, str. 196] Bublinkové řazení bývá obecně považováno za nejhorší. Existují však situace, ve kterých je nejlepší, například třídění extrémně malých seznamů (2 nebo 3 prvky) nebo třídění dat na přepisovatelné pásce, kdy nemáme další pásku, na kterou bychom přepsali výsledná setříděná data, a do operační paměti se nám vejdou jen dva záznamy, abychom je mohli na pásce přehodit, a potřebujeme minimalizovat výskyt operace převíjení pásky.

http://courses.cs.vt.edu/~cs1104/Algorithms/Bubble.explan.html
http://stackoverflow.com/questions/276113/what-is-a-bubble-sort-good-for
https://news.ycombinator.com/item?id=4646509

2.2.4.3 Porovnání efektivity řazení přímým výběrem a bublinkovým řazením

Nejčastější operací v třídících algoritmech je porovnání dvou položek, druhou nejčastější operací je přehození dvou položek. Operace porovnání se musí provést vždycky, operace přehození jen v případě, že je nutná. Počet přehození během třídění závisí na počátečním uspořádání hodnot tříděného seznamu. Při odhadu efektivity algoritmu se tedy nejvíce přihlíží k počtu operací porovnání.

Při řazení přímým výběrem při 1. průchodu seznamem provedeme $(n - 1)$ porovnání, ve 2. průchodu seznamem provedeme $(n - 2)$ porovnání, v posledním $(n - 1)$. průchodu seznamem provedeme 1 porovnání. Celkem se tedy provede $(n - 1) + (n - 2) + ... + 1 = ((n - 1) + 1) \cdot (n - 1) / 2 = n \cdot (n - 1) / 2 = (n^2 - n) / 2 = 0{,}5n^2 - 0{,}5n$ porovnání.

Při bublinkovém řazení provedeme stejný počet porovnání jako u řazení přímým výběrem, ale počet přehození dvou položek nebude nikdy nižší. O to je tedy bublinkové řazení pomalejší.

Při řazení hledáme v seznamu nebo jeho části vždy tu největší (nebo nejmenší) položku a musíme ho tedy projít celý. Z toho důvodu se v počtu porovnání neliší nejhorší, nejlepší a průměrný případ. Všechny jsou úměrné n^2.

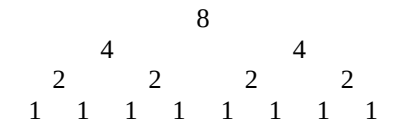
2.2.4.4 Řazení rozdělováním (quick sort)

Algoritmus vybere ze seznamu jeden prvek zvaný mezník a následně přeorganizuje jeho položky tak, aby na začátku seznamu (v jeho levé části) byly jen hodnoty menší nebo rovny mezníku a na konci seznamu (v jeho pravé části) jen hodnoty větší nebo rovny. Potom provede to samé odděleně s levou a pravou částí seznamu. Takto drobí zpracovávaný seznam, až dospěje ke kouskům velikosti jedna, které samozřejmě jsou seřazeny. [Pascal, str. 197] Quicksort byl vynalezen britským počítačovým vědcem C. A. R. Hoarem v roce 1961. Tento vědec se zabýval strojovým překladem jazyků a algoritmus quicksort vynalezl pro rychlé vyhledávání ve slovnících.

http://courses.cs.vt.edu/~cs1104/Efficiency/Chapter3.270.htm

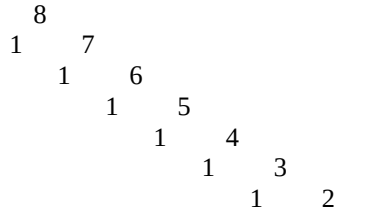
2.2.4.4.1 Nejlepší případ

Ve všech částech tříděného seznamu se stane, že pro rozdělování je vždy vybrán takový prvek, že právě polovina prvků v seznamu je menší a polovina větší než on. V takovém případě se při prvním průchodu seznamem provede $(n - 1)$ porovnání, tj. každý prvek kromě mezníku se porovná s mezníkem. Při druhém průchodu seznamem se pracuje odděleně ve dvou částech. V každé z nich se provede $(n/2 - 1)$ porovnání atd. Počet průchodů seznamem je roven $\log_2 n$, viz kapitola 2.2.3. „Porovnání efektivity 2 různých algoritmů pro vyhledávání“. Celkem se tedy provede $(n - 1) + 2 \cdot (n/2 - 1) + 4 \cdot (n/4 - 1) + ... + n/2 \cdot (2 - 1)$ porovnání. Z toho lze odvodit, že když se *n* blíží nekonečnu, je počet porovnání v algoritmu úměrný $n \cdot \log_2 n$. Neexistuje třídící algoritmus pro jediný procesor, který by dosahoval lepší efektivity.



2.2.4.4.2 Nejhorší případ

Ve všech částech tříděného seznamu se stane, že za mezník se zvolí vždy ten největší nebo nejmenší prvek. Při 1. průchodu seznamem se provede $(n - 1)$ porovnání, při 2. průchodu se provede $(n - 2)$ porovnání, při posledním $(n - 1)$. průchodu se provede $(n - (n - 1))$ porovnání. Součet je roven $0{,}5n^2 - 0{,}5n$ porovnání, což je stejný výsledek, jako u řazení přímým výběrem. V praxi k tomu může dojít, když třídíme seznam již setříděný a za mezník volíme vždy první nebo poslední položku.



2.2.4.4.3 Průměrný případ

Stejně jako u výše zmíněného algoritmu pro binární vyhledávání závisí na uspořádání dat. Podle

statistických sledování se průměrné chování algoritmu řazení rozdělováním od optima příliš neliší.

2.2.4.5 Kdy který třídící algoritmus použít?

Efektivita řádu $n \cdot \log_2 n$ je podstatně lepší než n^2. Třídící algoritmy s efektivitou řádu n^2 však nejsou k nepotřebě. Jejich použití je optimální pro malá data. To ukazuje prostý výpočet pro malá n. Dále je třeba vzít v úvahu i to, že vzhledem k tomu, že jsou jednodušší, potřebují méně příkazů, které obalují porovnávání a přehazování. Z toho vyplývá i možnost dalšího vylepšení třídění rozdělováním. Když má tříděný úsek seznamu méně než asi 10 položek, doporučuje se na něj uplatnit řazení přímým výběrem.

n	$0{,}5(n^2 - n)$	$n \cdot \log_2 n$
1	0	0
2	1	2
3	3	5
4	6	8
5	10	12
6	15	16
7	21	20
8	28	24
9	36	29
10	45	33

2.2.4.6 Nejhorší třídící algoritmus

1. WHILE NOT seznam již je setříděn DO
2. Náhodně seznam zamíchej.

Náročnost tohoto algoritmu je $n!$.

http://courses.cs.vt.edu/~cs1104/Efficiency/Chapter3.275.htm

Algoritmy, které testují všechny možnosti vygenerované například permutováním, se nazývají **algoritmy hrubé síly** (**brute force algorithms**). Počet položek, které v rozumném čase dovedou zpracovat, nikdy nepřesáhne 20, i kdyby se podstatně zlepšila úroveň hardwaru. Tyto algoritmy patří mezi algoritmy s exponenciální časovou náročností, protože funkce $n!$ roste pro velká n víc než c^n, kde $c > 1$.

http://courses.cs.vt.edu/~cs1104/Efficiency/Chapter3.290.htm

2.2.5 Základní dělení algoritmů dle časové náročnosti

Z hlediska časové náročnosti odlišujeme algoritmy s **polynomiální** složitostí od ostatních složitějších zvaných **nepolynomiální**. Proč se to odlišuje, když například polynomiální algoritmus řádu n^{100} je pro $n < 997$ časově složitější než exponenciální algoritmus se složitostí 2^n? U polynomiálních algoritmů extrémních stupňů se musí při asymptotické analýze k jejich specifikům přihlédnout. Asymptotická analýza totiž předpokládá, že analyzované algoritmy budou pro úlohy takzvaně „ze života“, a pro ty bývá stupeň polynomu maximálně 4, tj. 4 vnořené cykly, a ani potřebná konstanta nebývá velká. [UI3, str. 269] Mnoho úloh ze života však není řešitelné jinak než výčtem všech možností, mají tedy složitost $n!$ nebo c^n. Proto bývá mezi úlohami polynomiálními a nepolynomiálními podstatný rozdíl. Polynomiální algoritmy jsou označovány za „lehké“ (tractable, easy), kdežto ty, které nemají polynomiální složitost, jsou „nezvládnutelné“ (intractable, hard).

http://courses.cs.vt.edu/~cs1104/Efficiency/Chapter3.300.htm

2.2.5.1 Složitost typických úloh

Součet n čísel ..... $\Theta(n)$
Zpracování n čísel v m vnořených cyklech ..... $\Theta(n^m)$
Binární vyhledání v n číslech ..... $\Theta(\lg n)$
Nejlepší případ seřazení n čísel ..... $\Theta(n \lg n)$
Nejhorší případ seřazení n čísel ..... $\Theta(n^2)$
Nejhorší případ hledání podřetězce s m znaky v řetězci s n znaky ..... $\Theta(nm)$
Výčet všech možností, jak lze vybrat podmnožinu z n čísel ..... $\Theta(2^n)$
Výčet všech možností, jak lze seřadit n čísel ..... $\Theta(n!)$

2.2.5.2 Využití údajů o složitosti algoritmu

Známe-li závislost počtu vykonaných instrukcí na množství dat, můžeme na libovolném hardwaru odhadnout délku zpracování úlohy podle toho, jak dlouho trvalo zpracování určitého množství dat.

Příklad č. 1:

Program, který dostane jako vstup n hodnot, vykoná při jejich zpracování počet instrukcí přímo úměrný druhé mocnině n. Pro n = 1000 hodnot trvalo zpracování na počítači přibližně 5 minut. Kolik minut bude na stejném počítači za stejných podmínek trvat zpracování 2000 hodnot?

$5 / 1000^2 = x / (2 \cdot 1000)^2$
$x = 5 \cdot 2^2 = 20$ minut

Příklad č. 2:

Program, který dostane jako vstup n hodnot, vykoná při jejich zpracování počet instrukcí přímo úměrný $n!$. Pro n = 5 hodnot trvalo zpracování na počítači přibližně 3 minuty. Kolik minut bude na stejném počítači za stejných podmínek trvat zpracování 7 hodnot?

$3 / 5! = x / 7!$
$x = 3 \cdot 7! / 5! = 3 \cdot 7 \cdot 6 \cdot 5! / 5! = 126$ minut

Příklad č. 3:

Program, který dostane jako vstup n hodnot, vykoná při jejich zpracování počet instrukcí přímo úměrný 2^n. Pro n = 5 hodnot trvalo zpracování na počítači přibližně 80 minut. Kolik minut bude na stejném počítači za stejných podmínek trvat zpracování 3 hodnot?

$80 / 2^5 = x / 2^3$
$x = 80 \cdot 2^3 / 2^5 = 80 \cdot 2^{3-5} = 80 / 2^2 = 20$ minut

Příklad č. 4:

Program, který dostane jako vstup n hodnot, vykoná při jejich zpracování počet instrukcí přímo úměrný $\log_2 n$. Pro $n = 2^8$ hodnot trvalo zpracování na počítači přibližně 80 minut. Kolik minut bude na stejném počítači za stejných podmínek trvat zpracování 2^{10} hodnot?

$80 / \log_2 2^8 = x / \log_2 2^{10}$
$x = 80 \cdot 10 / 8 = 100$ minut

2.2.5.3 Stručná klasifikace úloh dle teorie složitosti (complexity theory)

http://www.wikipedia.org/wiki/Complexity_theory_in_computation

Polynomiální (Polynomial) – úlohy, které mohou být vyřešeny v polynomiálním čase.
NP (Non-deterministic Polynomial) – úlohy, pro které zatím nikdo nenašel algoritmus s polynomiální složitostí, protože takový algoritmus pravděpodobně ani neexistuje. Existuje-li či ne, je nejdůležitější dosud nezodpovězená otázka v teoretické informatice. Pro úlohu v této třídě je alespoň možné v polynomiálním čase ověřit správnost určitého řešení, pokud existuje. Příkladem NP úlohy je varianta takzvaného „problému obchodního cestujícího" (Traveling Salesman Problem – TSP). Zde je zadána poloha všech měst, která má obchodní cestující právě jednou navštívit, a máme najít, ve kterém pořadí má města navštívit tak, aby mu to zabralo trasu kratší, než je určitá hodnota. Pro konkrétní posloupnost měst jsme schopni v polynomiálním čase spočítat délku trasy na jejich objetí a rozhodnout tak, zdali to je nebo není správné řešení.
NP-úplné (NP-complete) – nejtěžší NP úlohy, u kterých je na první pohled jasné, že pro ně polynomiální algoritmus neexistuje. Je to například varianta TSP, ve které máme rozhodnout, zda existuje trasa kratší, než je určitá hodnota. Odpověď „ano" nebo „ne" nelze rychle ověřit. Patří sem i nalezení určitého řešení úlohy typu 15-puzzle.
NP-těžké (NP-hard) – optimalizační varianty NP-úplných úloh, jako například určit pořadí měst zaručující nejkratší trasu obchodního cestujícího ze všech možných, nebo naleznout nejkratší sekvenci tahů pro vyřešení úlohy typu 15-puzzle. Kromě optimalizačních sem patří i některé rozhodovací úlohy, jako například rozhodnout, zda existuje přesně *L* různých způsobů, jak lze obarvit politickou mapu zeměkoule 4 barvami. Odpověď „ano" nebo „ne" opět nelze rychle ověřit. I kdyby někdo vypsal *L* obarvení států 4 barvami, neměli bychom jistotu, že jiná obarvení neexistují.

3 Implementace jako vztah mezi softwarem a hardwarem

http://courses.cs.vt.edu/~cs1104/VirtualMachines/Chapter6.010.htm

3.1 Operační systém

Hardware počítače je pro většinu lidí těžko pochopitelný. Je třeba zařídit, aby jej přesto většina lidí dovedla používat. K tomu jsou zapotřebí programy (software), které na daném hardwaru dokáží vytvořit pro běžné lidi pochopitelné (user-friendly) virtuální prostředí. Takovéto prostředí nám umožňuje při práci s počítačem přemýšlet v termínech celá čísla, reálná čísla, znaky, příkazy vyšších programovacích jazyků, aritmetické výrazy, proměnné, soubory a adresáře/složky namísto termínů bity, byty, strojové instrukce, paměťové adresy, stopy a sektory. Programy, které mají toto za úkol, se nazývají **systémový software**. Jiná definice systémového softwaru je „sada počítačových programů, které spravují zdroje počítače a zprostředkovávají k nim přístup".
Zdroje spravované systémovým softwarem jsou procesor a operační a disková paměť.
Systémový software se skládá z operačního systému a utilit.
Operační systém (OS) je software, který ovládá periferní hardware, plánuje úkoly (určuje preference v jejich vykonávání), přiděluje paměť uživatelům a programům a poskytuje uživatelské rozhraní (interface), když zrovna není puštěn nějaký aplikační software.
Aplikační SW je tvořen programy, kvůli kterým je počítač využíván. Příkladem jsou Microsoft Office, překladače, hry.
Utility rozšiřují možnosti operačního systému. Příkladem je aplikace Poznámkový blok nebo Windows Media Player. Smyslem oddělování utilit od OS je možnost přizpůsobení OS. Rozdíl mezi OS, utilitami a aplikačním SW je často vágní.
Uživatelské rozhraní má dva základní druhy: příkazový řádek (Command Line Interface, např. MS DOS) a grafické rozhraní (Graphical User Interface – GUI, např. MS Windows).

3.2 Úrovně programovacích jazyků

Počítačový software je psán v programovacích jazycích, které mají několik úrovní:
1. **Strojový kód** (**machine language/code**) je programovací jazyk nejnižší úrovně, kterému přímo rozumí hardware, protože je složen výhradně z binárních čísel. Skládá se z nejjednodušších možných instrukcí. Lidé jej pro psaní programů nepoužívají.
2. **Asembler** (**assembly language**) je programovací jazyk, který používají výrobci počítačů pro tvorbu systémového softwaru nejnižší úrovně a tvůrci překladačů, viz kapitola 3.3. Asembler se skládá z těch samých instrukcí jako strojový kód, ale instrukce a proměnné mají jména místo čísel, aby v něm mohli psát programy lidé. Asembler je specifický pro každý typ počítače, protože i strojový kód, který je jím vyjádřen, lze vykonávat pouze na určitém typu hardware. Program psaný v jazyce asembler je textový soubor zvaný **zdrojový text** nebo **zdrojový kód**, který je překládán programem zvaným také **asembler** (**assembler**) do strojového kódu, čímž vznikne spustitelný program neboli program, kterému přímo rozumí hardware.
http://courses.cs.vt.edu/~cs1104/HLL/TOC.html
http://courses.cs.vt.edu/~cs1104/HLL/HLL_1_0.html
3. **Vyšší programovací jazyky** (**high-level languages**) jsou jazyky, které používají uživatelé počítačů pro tvorbu vlastních programů. Na rozdíl od asembleru jsou jejich instrukce nezávislé na hardwaru. Vyšší programovací jazyky totiž **abstrahují** od detailních instrukcí konkrétního hardwaru k obecnějším stavebním prvkům algoritmů a datových struktur. Jednotlivé stavební prvky mohou zahrnovat celý sled instrukcí, které by se v asembleru psaly jednotlivě. To usnadňuje jejich využití pro vyjádření algoritmů. Stejně jako v případě asemblerů jsou algoritmy ve vyšších programovacích jazycích psány ve formě textových souborů zvaných **zdrojový text** nebo **zdrojový kód** a program zvaný **překladač** je překládá do strojového kódu. Programy psané ve vyšších programovacích jazycích mají větší portabilitu než asembler. **Portabilita** je přenositelnost programu na jinou platformu. **Platforma** je určitá kombinace hardwaru, operačního systému a překladače. Pro každou platformu a určitý vyšší programovací jazyk lze totiž vytvořit překladač, který zařídí, aby šlo na dané platformě algoritmus zapsaný v tomto jazyce vykonat. Nejznámějšími druhy vyšších programovacích jazyků jsou Basic, C a Pascal. Patří mezi ně všechny jazyky zmíněné v kapitole 3.5.

3.3 Překladače

Existují dva základní typy překladačů: kompilátor a interpret.
Kompilátor (**compiler**) přeloží celý zdrojový text programu (source code) najednou do strojového kódu, který se v této souvislosti nazývá object code. Výsledkem je, že z textového souboru s algoritmem zapsaným ve vyšším programovacím jazyce vznikne spustitelný soubor, který má většinou příponu „exe". Příklady takových jazyků jsou C a Pascal.
Interpret (**interpreter**) překládá program po jednotlivých řádcích a hned je vykonává. Vykonání programu pomocí interpretu je pomalejší než pomocí kompilátoru, protože se při něm překládá do strojového kódu za běhu. Příkladem takového jazyka je VBA (Visual Basic for Applications) a Perl.

Implementace Javy [Computer Science, str. 262], http://en.wikipedia.org/wiki/Java_programming_language
V animované webové stránce je software řídící animaci přenášen přes Internet spolu se stránkou. Kdyby byl tento software dodáván ve formě zdrojového textu, prohlížení stránky by bylo zdržováno kvůli nutnosti přeložit tento software do strojového kódu předtím, než může být takto vzniklý program spuštěn. Avšak dodávání softwaru přímo ve formě strojového kódu by znamenalo, že pro klientské počítače používající různé strojové kódy by se musely poskytovat různé verze softwaru.
Společnost Sun Microsystems vyřešila tento problém vytvořením univerzálního „strojového kódu“ zvaným bytecode, do kterého mohou být přeloženy zdrojové programy napsané v jazyce Java. Ačkoliv bytecode není skutečný strojový kód, může být rychle vykonáván jakýmkoli počítačem, který má vhodný interpret. Takové interprety jsou standardní součástí současných internetových prohlížečů. Takže, když je software pro ovládání webové stránky napsán v jazyce Java a přeložen do bytecode, potom je tento bytecode možné přenášet do prohlížečů na různých typech počítačů, kde poskytují efektivní animaci.
Java je příkladem multiplatformního software (cross-platform software).

3.4 Instrukce v programovacích jazycích

Čím vyšší je programovací jazyk, tím více byla při jeho tvorbě použita **abstrakce** neboli zobecnění neboli skrytí nepodstatných detailů instrukcí a dat za identifikátory.
Identifikátory označují proměnné, datové typy a podprogramy.
Datový typ určuje pro každý identifikátor přípustné hodnoty, které může mít, a přípustné operace, které se ním mohou dělat.
Podprogram (**subroutine**) může být buďto funkce nebo procedura. Rozlišování funkcí a procedur je závislé na druhu programovacího jazyka, viz [Computer Science, str. 235]. Níže uvedená definice funkce a procedury je obvyklá v jazyce Pascal.
Funkce je podprogram, který ze vstupních dat vypočítá jedinou funkční hodnotu a vstupní data nezmění.
Procedura je určitá část algoritmu, která může měnit (přepočítat) data, se kterými pracuje.
Při psaní programů musíme dodržovat syntaxi a sémantiku. Jinak vzniknou dva druhy chyb:
Syntaktické chyby porušují pravidla platící v daném jazyce. Překladač je odhalí a dokud je programátor všechny neopraví, nelze program přeložit do spustitelného souboru. To, že je program syntakticky správný, neznamená, že je správný úplně, protože v něm mohou být ještě sémantické chyby.
Sémantické chyby mohou způsobit běhovou chybu při vykonávání programu počítačem (například přetečení proměnné) nebo chybné zpracování vstupu. Programátor je musí v programu najít sám. Sémantika programu určuje, co má program dělat.

3.5 Paradigmata programovacích jazyků

Jazyk, který neovlivní způsob, jakým uvažujete o programování, nemá cenu se učit. – Alan Perlis
Druhy programovacích jazyků dle popularity: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
Předchozí 2 řádky byly převzaty z prezentace http://www.cs.mcgill.ca/~bpientka/comp-thinking.pdf (autor Prof. Brigitte Pientka).
http://courses.cs.vt.edu/~cs1104/TowerOfBabel/ToB_1_0.htm
Při vývoji vyšších programovacích jazyků se uplatňují různé přístupy, které v historii existují paralelně vedle sebe. Názor, jak má vypadat programovací jazyk, se nazývá paradigma. Zpravidla se rozlišuje 5 takových paradigmat.
Procedurální neboli imperativní jazyky (**procedural/imperative languages**) – pomocí nich programátor stanoví přesný sled instrukcí, které se mají vykonat. Jsou to nejběžněji používané jazyky a jejich příkladem jsou FORTRAN, COBOL, Basic, C, Pascal a Perl. Procedurální jazyky jsou více či méně univerzální. **Univerzální jazyky** (**general-purpose languages, GPL**) lze použít na řešení libovolných problémů. Příkladem takového jazyka je C. Poněkud užší možnosti má například Pascal. C na rozdíl od Pascalu má například přístup k jednotlivým bytům paměti bez ohledu na typ proměnné určující počet bytů.
Logické neboli deklarativní jazyky (**logic/declarative/relational languages**) popisují řešený problém pomocí faktů a pravidel. Výsledek je vypočten pomocí inferenčního mechanismu, což je obdoba překladače, je to tedy něco, co neurčuje programátor. Inferenční pravidla jsou pravidla pro odvozování nových faktů ze známých faktů. Nejznámějším deklarativním jazykem je Prolog. Příkladem úlohy vhodné pro logické programování v jazyce Prolog je třeba tato: *Alík je pes. Všichni psi jsou šelmy. Všechny šelmy jedí maso. Jí Alík maso?*
Funkcionální jazyky (**functional/applicative languages**) popisují řešený problém pomocí do sebe vnořených funkcí a pomocí definic těchto funkcí. Tyto funkce mají širší definici, než je ta, kterou mají v různých procedurálních jazycích a nebo v kapitole 3.4. Funkce ve funkcionálních jazycích řeší i úlohy typické pro procedury v procedurálních jazycích například třídění. Kdybychom například chtěli vypočítat nejmenší hodnotu ze seznamu, mohl by program ve funkcionálním jazyce vypadat takto: *VyberPrvni(SeradVzestupne(Seznam))*. Mezi funkcionální jazyky patří LISP, Scheme, FP a ML. Program ve funkcionálním jazyce je jednou jedinou funkcí s množstvím vnořených funkcí a místo cyklů používá rekurzi, viz. kapitola 6.
Problémově orientované jazyky (**domain-specific/problem-oriented/special-purpose languages**) jsou jazyky, které se využívají pro řešení určité kategorie problémů typických pro určité profese, například SQL pro databáze, Linda, paralelní verze jazyka FORTRAN, STRUDL, PANIC, CIRCUIT, nebo fungujících jen v rámci určitého aplikačního softwaru, například Matlab, Mathematica, MS Excel a MS Access. Jejich opakem jsou **univerzální jazyky** zmíněné výše.
Objektově orientované jazyky (**object-oriented languages**) například Simula, Smalltalk, Java, C++ a Ada vznikly z potřeby vyvíjet stále složitější programy a ujaly se v 90. letech 20. století. [Pascal, str. 213] V objektově-orientovaném programování (OOP) jsou datové jednotky přetvořeny na aktivní „objekty“, spíše než aby byly pasivními jednotkami, tak jak je vnímá tradiční imperativní paradigma. Třemi základními myšlenkami při tvorbě objektově orientovaných jazyků byly **abstrakce**, **hierarchie** a **modularita**. Pochopení činnosti složitého softwaru vyžaduje abstrakci umožňující nahrazení datových typů a podprogramů jejich identifikátory. Abstrakce má mnoho úrovní, takže programátorský projekt lze podle metody rozděl a panuj vidět jako strom, jehož kmenem je projekt sám, uzly jsou podprogramy a listy jsou jednotlivé příkazy. Náročnost kompletace takového stromu je dále snížena sdružováním jednotlivých komponent programu v modulech, které všechny mají stejný způsob použití neboli stejné rozhraní pro spolupráci s ostatními viz [Pascal, str. 203]. **Modul** je tedy nezávislá část jednoho nebo více programů sdružující podprogramy, které řeší určitou jasně vymezenou část celkového problému. Za nosné pilíře OOP bývají považovány **principy** zapouzdření, dědičnosti a polymorfismu platící pro objekty.
Objekt je datová struktura, která je instancí neboli konkrétním příkladem obecného objektu představujícího třídu objektů. Každá **třída** (**class**) je zvláštním modulem a má svou pozici v třídní hierarchii.

Zapouzdření (**encapsulation**) přiřadí objektu podprogramy zvané metody, které objekt dokáže se sebou provádět. Při použití objektu v programu se o jeho podprogramy již nemusíme starat, pouze je voláme způsobem *JménoObjektu.JménoMetody*. Podprogramy v objektech jsou psány v imperativním stylu.

Dědičnost (**inheritance**) znamená, že metody platící pro určitou třídu objektů platí i pro třídu objektů, která je v hierarchii pod ní. Potomek může mít jen jediného rodiče (ale rodič libovolné množství potomků) a zděděné metody si může pozměnit a přidat k nim další, což je myšlenka polymorfismu.

Polymorfismus (**polymorphism**) umožňuje nadefinovat metody, které jsou pro všechny třídy objektů společné, ale jejich chování se bude lišit podle druhu daného objektu. Takové metody nazýváme polymorfními.

Jako příklad OOP si můžeme představit seznam jmen. V tradičním imperativním paradigmatu je tento seznam považován jen za kolekci dat. Každý program, který jej má zpracovat, musí obsahovat algoritmy pro vykonávání požadovaných manipulací. Tento seznam je tedy pasivní v tom smyslu, že je obsluhován vnějším programem, než aby se obsluhoval sám. V OOP je však seznam konstruován jako objekt obsahující data seznamu spolu s kolekcí procedur pro manipulaci s tímto seznamem, například proceduru pro přidání nové položky do seznamu, detekci, zda je seznam prázdný, a třídění seznamu. To znamená, že vnější program pracující se seznamem nemusí obsahovat algoritmy pro vykonávání těchto úloh. Takže místo toho, aby setřídil seznam, požádá seznam, aby se setřídil sám.

Chcete-li programovat, musíte si pořídit překladač a znát programovací jazyk. Mnoho překladačů je dodáváno spolu s vývojovým prostředím, které umožňuje programy nejen psát a překládat, ale i pohodlně ladit (tj. odstraňovat syntaktické chyby). Vývojové prostředí zpravidla obsahuje i nápovědu, ze které je možné se jazyk naučit. Mezi vývojovými prostředími, které jsou zpravidla legálně dostupné jen za peníze, názvy některých z nich obsahují slovo *Visual*., například Visual Basic, Visual C++. Tato vývojová prostředí umožňují tvorbu takzvaných GUI aplikací (aplikací s grafickým uživatelským rozhraním) v prostředí MS Windows. (Existují i vývojová prostředí stejného typu, která neobsahují v názvu slovo *Visual*, například Borland Delphi, Borland C++ Builder, Borland JBuilder.) Tyto jazyky jsou objektově orientované, ale jejich uživatelé nemusejí umět tvořit objekty. Místo toho mohou sestavovat svoje aplikace z objektů, které jsou nabízeny vývojovým prostředím.

Pro GUI aplikace je typické, že jsou ovládány kliknutím na jejich součásti nebo stiskem klávesy. Takovéto programy jsou nazývány **SW systémy řízené událostmi** (**event-driven SW systems**) nebo také **interaktivní programy**. Opakem programování řízeného událostmi je historicky straší **dávkové programování**, ve kterém programátor celý proces předem naprogramoval a uživatelé proto do chodu programu nezasahují. Dávkové programování je typicky realizováno pomocí takzvaných **skriptovacích jazyků**, které se v programovacích paradigmatech dají zařadit jako samostatná kategorie vedle univerzálních jazyků.

Programovací jazyky se také někdy dělí na generace (1 až 5GL – Generation Language):
1GL – Strojový kód, viz kapitola 3.2,
2GL – Asembler, viz kapitola 3.2,
3GL – Vyšší programovací jazyky, viz kapitola 3.2,
4GL – Problémově orientované jazyky, viz kapitola 3.5,
5GL – Logické a funkcionální jazyky, viz kapitola 3.5.

4 Datové struktury

http://courses.cs.vt.edu/~cs1104/HLL/HLL_12_0.htm

U některých objektů je vhodnější sdružit data o nich do vyšších celků. Výsledkem jsou datové struktury neboli strukturované datové typy. Datové struktury se skládají z vnořených datových struktur nebo z jednoduchých typů. Základními druhy jednoduchých typů jsou
celá čísla – typ Integer,
reálná čísla – typ Real,
pravdivostní hodnoty – typ Boolean,
znaky – typ Char.

Typy Integer, Boolean a Char se nazývají **ordinální typy**, protože se vyznačují možností seřadit všechny své hodnoty podle pořadí. Typ Real mezi ně nepatří, protože neexistuje následník reálného čísla.

4.1 Pole (Array)

Polem je konečná skupina prvků stejného typu. Každý prvek má svůj index, který musí být ordinálního typu. Typickým příkladem vhodných dat je vektor nebo matice. Pro práci s polem se využívá řídící struktura příkazů zvaná for-cyklus.

4.2 Řetězec znaků (String)

Řetězec znaků je polem, jehož prvky jsou znaky. Pro strukturu „String“ jsou v programovacích jazycích vytvořeny některé standardní funkce a procedury, takže není jedno, jestli posloupnosti znaků dáme typ „Array of Char“ nebo „String“. Vhodným objektem pro takovou reprezentaci je text.

4.3 Záznam (Record)

Záznam sdružuje prvky, které mohou být různých typů. Každý prvek má svůj identifikátor. Typickým příkladem vhodných dat je řádek databázové tabulky neboli datová věta. Celá tabulka by potom mohla být reprezentována polem záznamů (Array of Record).

4.4 Množina (Set)

Množina patří mezi kuriozity Pascalu. Může se jí pochlubit jen málokterý programovací jazyk. Množina sdružuje prvky ordinálního typu. Prvky v množině můžeme určit jejich výčtem nebo bázovým typem množiny. Je-li bázovým typem množiny například typ Char, je k uložení množiny zapotřebí tolik bitů, kolik má možných hodnot typ Char. Například, když má typ Char 256 různých hodnot, tak každému znaku patří jeden bit, který svou hodnotou vyjadřuje, jestli daný znak do množiny patří nebo nepatří. Množina s bázovým typem Char tedy zabere v paměti 256 / 8 = 32 bytů. S množinami stejných bázových typů lze dělat v programech množinové operace sjednocení, průnik a rozdíl. Dále lze testovat jejich rovnost, nerovnost, a podmnožinu. Nejčastější použití množin je test, zda něco patří do množiny s použitím operátoru a klíčového slova IN, viz část programu, která uživatele nepustí dál, dokud nezadá správnou hodnotu:

```
REPEAT
        ReadLn (C)
UNTIL C IN ['a','A','n','N']
```

4.5 Asociativní pole (Associative array, Hash table)

Je prvkem jazyka Perl. Perl je vyšší programovací jazyk specializovaný na zpracování řetězců znaků. Zatímco typ pole indexuje prvky pořadovými (ordinálními) čísly, asociativní pole indexuje prvky pomocí řetězců znaků. Pomocí této datové struktury lze například jednoduše spočítat četnosti jednotlivých slov v textu. Výslednými položkami jsou četnosti slov a jejich indexy (klíči) jsou slova.

4.6 Dynamické datové struktury [Pascal, str. 159]

Chceme-li pracovat s polem dat, musíme dopředu znát maximální počet prvků, které bude mít. Po dobu běhu programu je v paměti počítače vyhrazeno pro pole tolik paměti, kolik by bylo potřeba, kdyby v něm byl maximální počet prvků. **Jednosměrný lineární seznam** je datová struktura, která se chová jako pole, ale v paměti zabírá jen tolik místa, kolik je nezbytné pro aktuální počet prvků. Aby bylo možné rozeznat pořadí prvků, každý prvek ukazuje na svého následníka. Poslední prvek ukazuje na speciální konstantu označenou klíčovým slovem NIL. Prvky na sebe ukazují pomocí ukazatelů. **Ukazatel** (**pointer**) je datový typ, jehož hodnotou je paměťová adresa. Jednosměrný lineární seznam lze například deklarovat následujícím způsobem:

```
TYPE   UkNum = ↑Num
       Num =  RECORD
                    Cislo : Integer;
                    Dalsi : UkNum
              END;
```

Identifikátor *UkNum* označuje typ „Ukazatel“. Identifikátor *Num* označuje typ „Záznam“, jehož položkami je celé číslo a paměťová adresa s následujícím záznamem.

Pokud je počet prvků předem znám nebo je jasně shora omezen rozumným číslem, bývá výhodnější použít pole, protože v typu „Pole“ můžeme k položkám přistupovat přímo, ale v lineárním seznamu musíme projít všemi předcházejícími prvky.

Jednosměrný lineární seznam má dokonalejší variantu zvanou **dvousměrný cyklický seznam s hlavou.** V tomto datovém typu nese každá položka kromě vlastní informace dva ukazatele – na svého následníka a předchůdce. Následníkem poslední položky seznamu je jeho začátek. Do seznamu je přidána navíc jedna fiktivní položka nazvaná „hlava“, která z hlediska nesených informací není součástí seznamu, ale slouží například k tomu, aby se hledání v seznamu zastavilo, když v něm hledaná položka neexistuje.

Pro zpracování dat popisujících například rodokmen je ideální datová struktura typu „**strom**“. V tomto datovém typu nese každá položka kromě vlastní informace dva ukazatele – na svého otce a matku. Ukazatelé listů stromu mají hodnotu NIL. Pro práci se stromem je výhodné používat rekurzívní algoritmy popsané níže.

4.7 Datové soubory

Data mohou být v počítači buďto v operační paměti nebo na pevném disku. Pokud jsou na disku, vytvářejí soubor. Soubory mohou být buďto **binární** nebo **textové.** V binárních souborech jsou data uložena pomocí těch samých bytů jako v operační paměti. Binární soubor je tvořen posloupností dat stejného typu. Mezi hodnotami nejsou oddělovače. Z programu pracujícího s binárním souborem musí být jasné, kolik bytů která hodnota zabírá. Velikost binárního souboru lze určit z použité datové struktury a počtu jejích prvků uložených do souboru. Například jsme pro data definovali typ „Záznam“ tímto způsobem:

```
RECORD
    Jméno : String [39]; {To znamená 40 bytů, protože v 1. bytu se uchovává délka řetězce}
    Plat : Integer {V dané implementaci zabírá 32 bitů}
END; {Konec seznamu položek v záznamu}
```

Do souboru jsme zapsali 100 000 datových vět tohoto typu. Kolik megabytů (přesněji mebibytů – MiB – viz následující kapitola) bude mít výsledný binární soubor?

Výsledek: 100 000 · (40 + 32 / 8) / 1024 / 1024 = 4,2 MiB.

V binárních souborech je možný přímý přístup k vybrané položce, známe-li její pořadí v souboru. Zatímco účelem binárních souborů je uchování dat, se kterými pracuje počítač, do textových souborů se obvykle zapisuje informace srozumitelná pro člověka. Textové soubory je nutné zpracovávat jako celek. Není možné v nich změnit třeba 5. slovo na 4. řádku, aniž by se nemusel přepsat celý soubor. Textové soubory jsou členěny na řádky. Na jednotlivých řádcích jsou znaky.

5 Jednotky informace

V informatice je obvyklé udávat objem dat nebo kapacitu v bytech s předponami znamenajícími mocninu 2, takže například kilobajt znamená 2^{10} bajtů, tj. 1024 B, zatímco v ostatních oborech předpona kilo znamená 1000. To je v informatice zneužíváno při prodeji výpočetní techniky. Například disk prodávaný jako 200 gigový se v počítači zobrazuje jako 186 gigový.

Příčina: 186 = 200 · 1000 · 1000 · 1000 / 1024 / 1024 / 1024.

Proto vznikl nový standard pro předpony. Tento standard byl do systému českých technických norem přejat v roce 2004 a znamená, že původní předpony K (kilo), M (mega), G (giga) a další budou vždy jen dekadické a jejich binární protějšky budou Ki (celými slovy *kibi* – *ki*lo *bi*nary), Mi (*mebi* – *me*ga *bi*nary), Gi (*gibi* – *gi*ga *bi*nary) atd. Užívání tohoto standardu se ještě úplně neujalo.

Více na http://cs.wikipedia.org/wiki/Bin%C3%A1rn%C3%AD_p%C5%99edpona a http://en.wikipedia.org/wiki/Binary_prefix.

S rostoucí mírou používání multimediálních souborů a komunikace na Internetu vzrůstá nutnost orientovat se v pojmu **přenosová rychlost** (**bit rate, bitrate**). Ta se obvykle udává v bitech za sekundu (bps, bit/s – bits per second) a předpony k, M, G a další jsou v tomto kontextu dekadické. Více viz http://en.wikipedia.org/wiki/Bit_rate a http://en.wikipedia.org/wiki/Kbps.

Máme-li soubor o určité velikosti, který se má přehrávat nebo přenášet, musíme znát vztah mezi rychlostí přehrávání nebo přenosu souboru, velikostí souboru a dobou potřebnou pro přehrání nebo přenos souboru. Máme například soubor o velikosti 1 GiB a potřebujeme vědět, kolik hodin potrvá jeho přenos při přenosové rychlosti 30 kbps.

Výsledek: 1 · 1024 · 1024 · 1024 · 8 / 1000 / 30 / 60 / 60 = 80 hodin.

6 Rekurzívní algoritmy [Pascal, str. 127]

Některé problémy lze rozložit na podproblémy, které jsou zmenšenou kopií původních problémů. Příkladem takového problému je Hanojská věž, viz

http://courses.cs.vt.edu/~cs1104/ProblemSolving/PS.030.html

Dalším příkladem jsou binární vyhledávání a algoritmus quicksort probrané výše.

Všechny tyto problémy se mohou řešit rekurzívními algoritmy. Rekurzívní algoritmus volá sám sebe. V jednom okamžiku je rozpracován více než jeden exemplář stejného podprogramu. Při tvorbě rekurzívních algoritmů je třeba dodržovat tyto zásady:
1. V algoritmu musí být definována koncová situace.
2. V každém kroku musí algoritmus problém zjednodušit.
3. Nejdříve je třeba testovat, zda nenastala koncová situace. Když ne, algoritmus zjednoduší problém a zavolá sám sebe.
Některé matematické funkce, jako je třeba determinant, se mohou definovat rekurzívně. Nejznámějším příkladem je faktoriál:
$0! = 1$
$n! = n \cdot (n - 1)!$ pro $n > 0$
Rovnice $0! = 1$ definuje koncovou situaci pro faktoriál.
Rovnice $n! = n \cdot (n - 1)!$ obsahuje definovanou funkci na obou stranách. Na pravé straně je řešený problém jednodušší, tj. počítat $(n - 1)!$ je jednodušší (vynásobí se méně čísel) než počítat $n!$.
Rekurzívně lze definovat také posloupnosti, z nichž velmi známá je v matematice Fibonacciho posloupnost definovaná takto:
$F(0) = 1$
$F(1) = 1$
$F(n) = F(n - 1) + F(n - 2)$ pro $n > 1$
Funkce počítající rekurzívně n-tý člen Fibonacciho posloupnosti zapsaná v jazyce Pascal vypadá takto:

```
FUNCTION Fibonacci (N : Integer) : Integer;
BEGIN IF N < 0 THEN Fibonacci := 0 ELSE IF N <= 1 THEN Fibonacci := 1 ELSE
        Fibonacci := Fibonacci (N – 1) + Fibonacci (N – 2);
END;
```

Fibonacciho posloupnost je dobrou demonstrací nevýhody, jakou některé rekurzívní algoritmy mohou mít. Tou nevýhodou je nízká efektivita. Programujeme-li výpočet n-tého členu Fibonacciho posloupnosti rekurzívně, nevyužijeme mezivýsledek $F(n - 2)$ k výpočtu $F(n - 1)$ ale počítáme každý člen zvlášť od začátku. Efektivita rekurzívně programované Fibonacciho posloupnosti je úměrná Fibonacciho posloupnosti. Stejnou posloupnost lze naprogramovat pomocí for-cyklu a výsledná efektivita je potom úměrná n.
Efektivita rekurzívního algoritmu pro výpočet faktoriálu je úměrná n, takže zde rekurzívnost tolik neškodí. Přesto však na počítačích pracuje rychleji algoritmus pro faktoriál využívající for-cyklus, protože volání podprogramu programem je relativně časově náročné.
Rekurzívní algoritmy zpravidla bývají jednodušší a lépe pochopitelné. Platíme však nižší efektivitou. Vždy se vyplatí zvážit, zda problém nelze řešit iterativně (cyklem). Pokud takové řešení není neúnosně komplikované, zpravidla je záhodno dát mu přednost. Teoreticky je každý rekurzívní algoritmus možné zapsat iterativně, tj. s použitím cyklu.
Algoritmy pro práci s datovou strukturou typu „strom“ je nejlepší psát v rekurzívní podobě. Jejich nerekurzívní verze by efektivnější nebyly. Tyto algoritmy začínají s daty pracovat od kmene stromu a pokračují v sousedních uzlech. Z pohledu každého uzlu stromu se strom vycházející z něj jeví stejně, má jen jinou hloubku.
Dosud diskutovaným typem rekurze byla rekurze přímá, vyznačující se tím, že podprogram volal sám sebe. Je možno použít také rekurzi nepřímou, kdy podprogram A volá podprogram B, který volá podprogram A. Zúčastněných podprogramů samozřejmě může být i více než dva.