

Algoritmy - IAL

Prof. Ing. Jan M Honzík, CSc.

UIFS FIT VUT v Brně

2015/16

Základní informace o předmětu

- v IS je zapsaných přes 400 studentů ve dvou přednáškových skupinách. Je to hodně. Prosíme o spolupráci i shovívavost.
- 2 domácí úlohy (po 10 b)
- projekt s miniobhajobou pro kolektiv o stanoveném počtu studentů (obhajoba losem) (15 bodů)
- půlsestrální zkouška (14 bodů)
- semestrální zkouška (51 bodů)
- prémiové body

Obsah a zdroje 1. přednášky

1. Přednáška

- Úvod: Cíle předmětu, formy výuky, účast na výuce, pravidla hodnocení, přednáška, domácí cvičení a zadání, půlsementrální a závěrečná zkouška, prémiové úkoly.
- Seznámení s učiteli vedoucími projekt a domácí úlohy
- Seznámení s audiovizuálním systémem přednášek
- Vlastnosti pseudojazyka pascalovského typu.
- Zdroje:
karta předmětu – soubory – (dokumentový sklad FIT) (

Studijní zdroje

- Studijní zdroje:
 - Studijní opora IAL – text cca 260 stran (+ příklady) elektronicky přístupný. Obsahuje veškeré organizační i obsahové informace. Opora je základní studijní zdroj. Obsah opory je současně obsahem předmětu:
 - Předmluva
 - Úvod do datových struktur a algoritmů
 - Abstraktní datové typy
 - Vyhledávání
 - Řazení
 - Závěr

Slajdy z přednášek jsou oporou přednášejícího! Nelze je považovat za postačující studijní zdroj! Za chyby a překlepy v nich vyučující neručí.

Ikony (piktogramy) opory

Ikona



Navrhovaný význam

Počítačové cvičení, příklad

Ikona



Navrhovaný význam

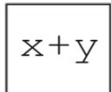
Správné řešení



Otázka, příklad k řešení



Obtížná část



Příklad



Důležitá část



Slovo tutora, komentář



Cíl



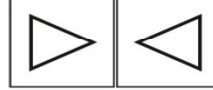
Potřebný čas pro studium,
doplněno číslicí přes hodiny



Definice



Reference



Zajímavé místo (levá, pravá
varianta)



Souhrn



Rozšiřující látka, informace,
znalosti. Nejsou předmětem
zkoušky.

Informace o organizaci domácích úloh

- Ing. Ivana burgetová, PhD, Dr. Bohuslav Křena,
- Poznámka:
 - Nové pojetí předmětu tohoto typu v Bc. studijním programu (dříve jako předmět magisterského studijního programu ADS - „Algoritmy a datové struktury“ (před tím „Programovací techniky“))
 - Výhody a nevýhody

Přibližný odhad časové náročnosti předmětu

- 1 kredit = 25-30 hodin práce (hodina 50min?)
- 5 kreditům odpovídá min 125-150 hod. studijní práce, z toho:
- přednášky 39 hod
- 2 domácí úlohy 26 hod
- práce na projektu 35 hod
- průběžné studium 20 hod
- příprava na půlsem. a záv. zk. 30 hod

Historie předmětu IAL

- Předmět vznikl již v 70. letech minulého století, především na základech světových učebnic jako je „The Art of Computer Programming“ jednoho z nejvýznamnějších informatiků Donalda Knutha „Algoritms + Data Structures = Programs.
-

THE CLASSIC WORK
NEWLY UPDATED AND REVISED

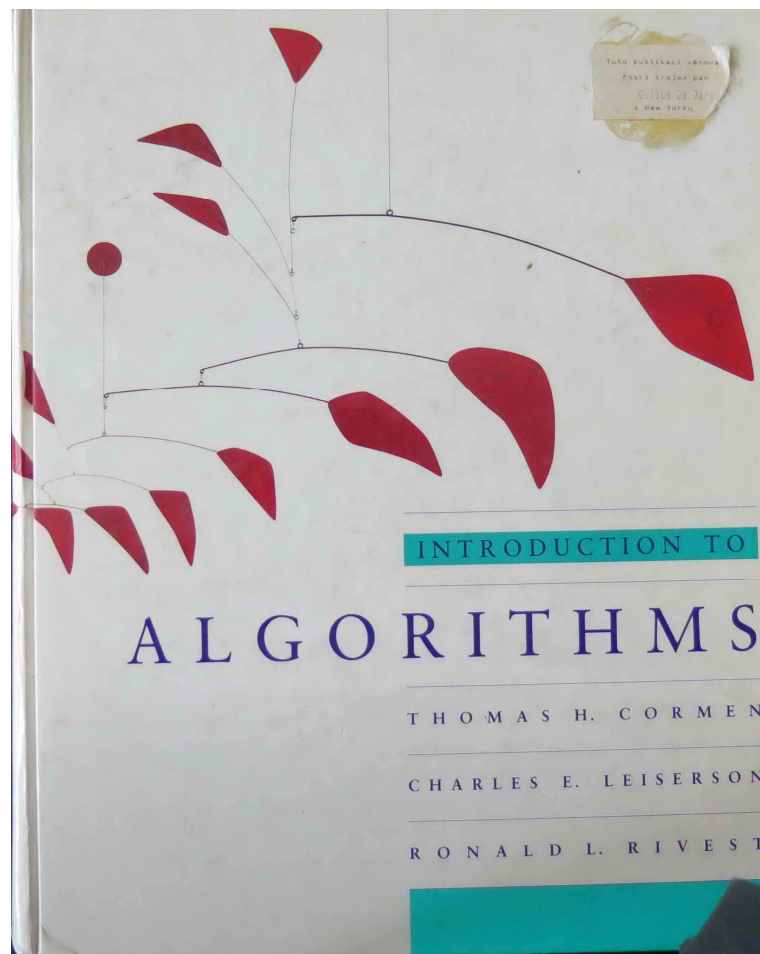
The Art of Computer Programming

VOLUME 3

Sorting and Searching
Second Edition

DONALD E. KNUTH

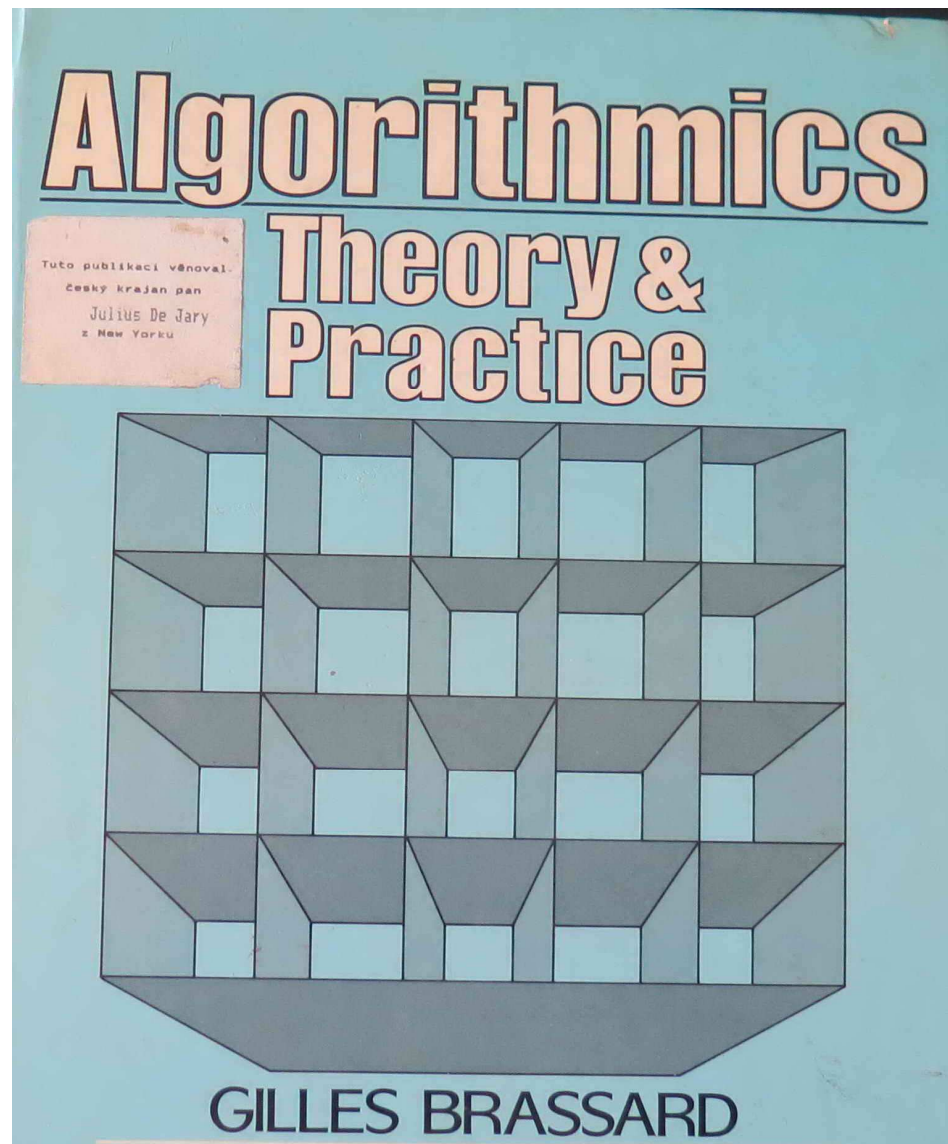
- Mezi nejvýznamnější učebnice na významných světových i evropských univerzitách patří:

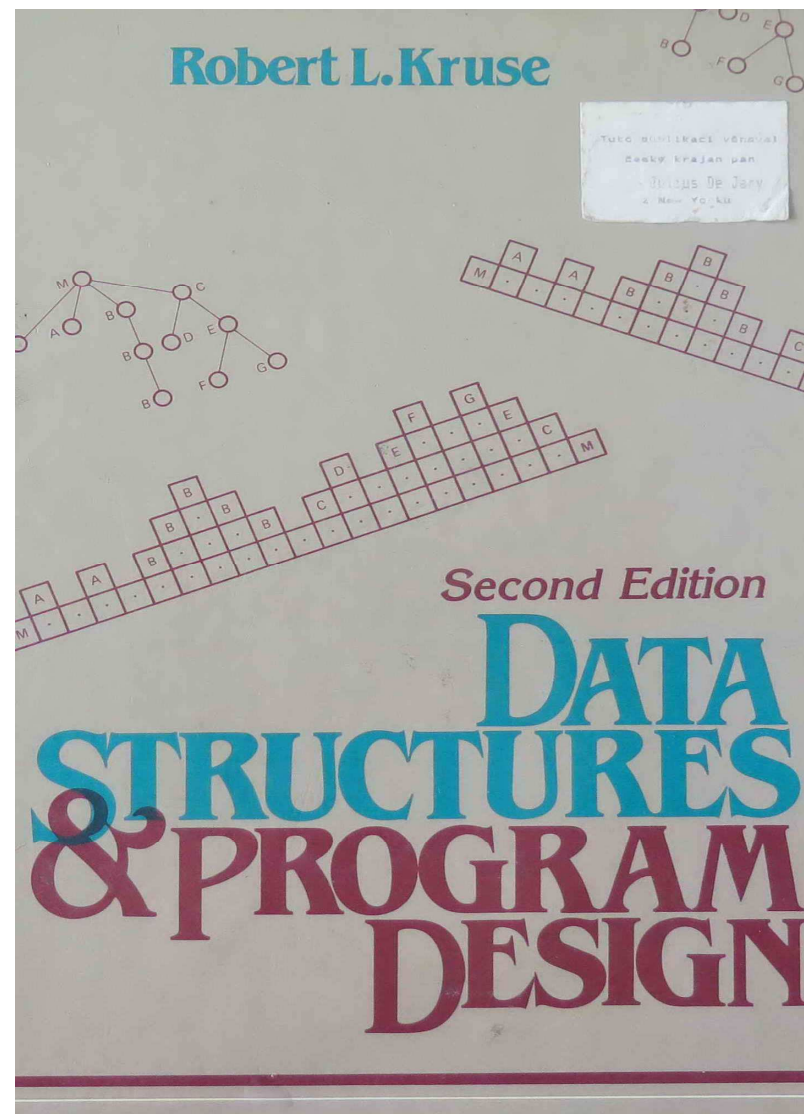


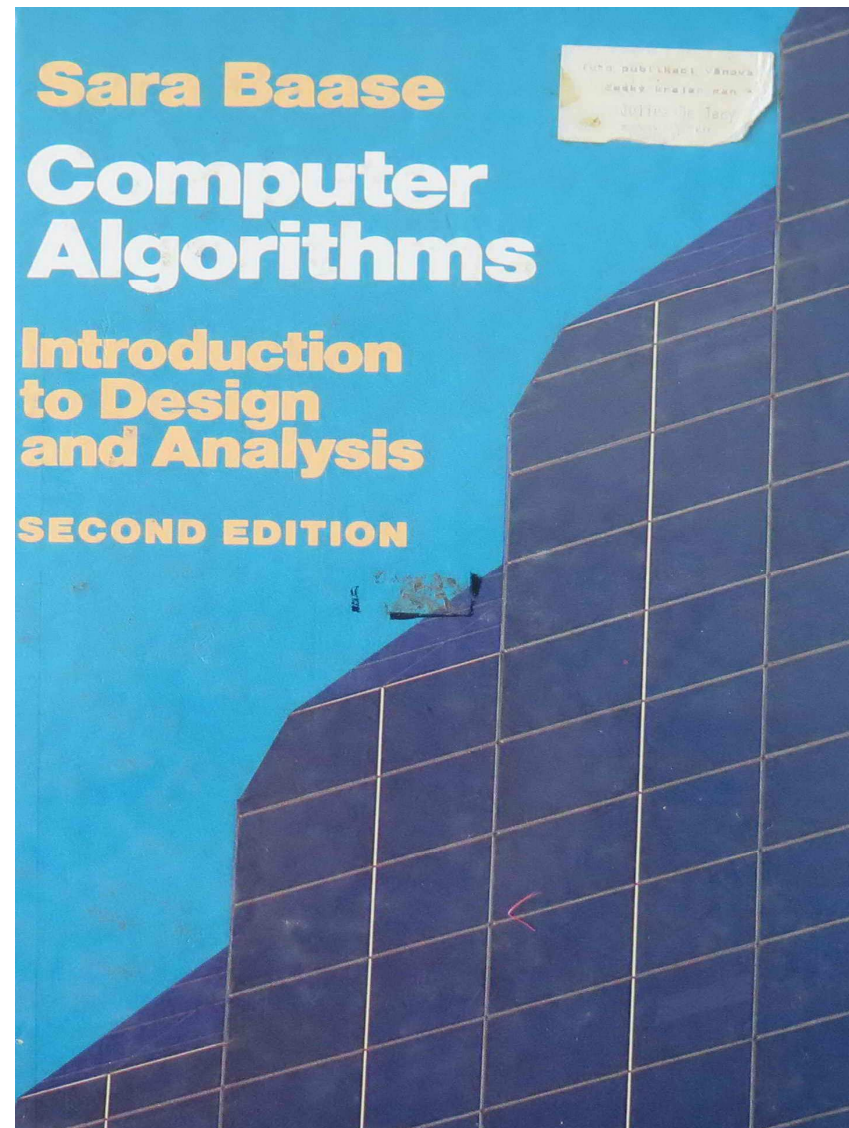
1.9.2015

1. přednáška

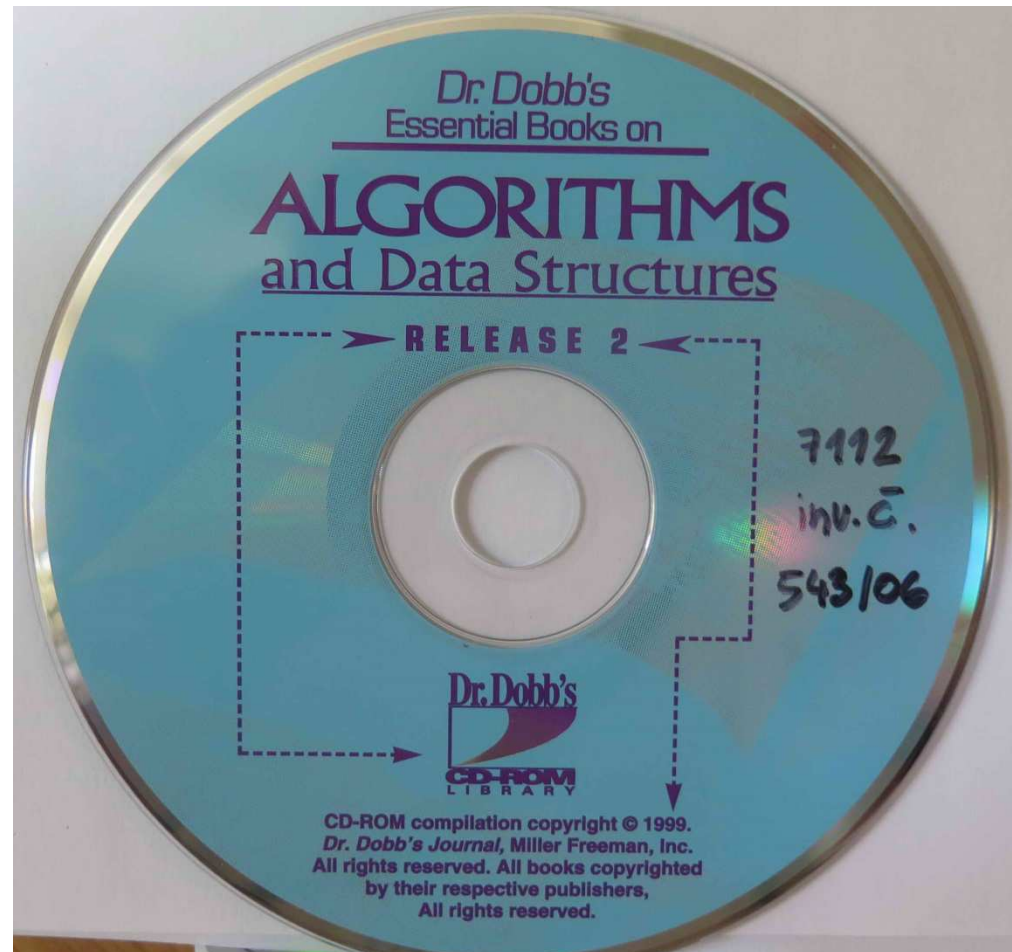
11







CD ROM – Dr.Dobbs Algorithms and Data Structures



- Žádná z těchto učebnic nepoužívá pro zápis algoritmů konkrétní programovací jazyk. Většina použitých pseudojazyků vychází ze zvyklostí jazyka Pascal.
- V IAL se proto pro zápis algoritmů bude také používat pseudojazyk vycházející z jazyka Pascal.

Metodické informace

- Jazyk zápisu algoritmů (jazyk realizace programů, jazyk při zkoušce)
- odborná a anglická terminologie
- žargon a jeho vymezení
- srozumitelnost zápisu (komentáře, dokumentace, navigace, identifikátory a názvy, grafická úprava zápisu, indentace, pravidla českého jazyka - pravopis)

Učební cíle a kompetence - specifické

- Seznámit se základními principy složitosti algoritmů.
- Seznámit se s principy dynamického přidělování paměti.
- Seznámit se se základními abstraktními datovými typy a strukturami, naučit se je implementovat a používat.
- Seznámit se s vyhledávacími metodami, naučit se je implementovat a používat.
- Seznámit se s řadicími metodami , naučit se je implementovat a používat.
- Naučit se rekurzivní a nerekurzivní zápisy základních algoritmů.
- Naučit se porozumět principům a analyzovat algoritmy vyhledávání a řazení.
- Osvojit si zásady dobrého programovacího stylu.

Učební cíle a kompetence - generické

- Seznámit se pasivně s algoritmickým jazykem pascalovského typu.
- Naučit se základům týmové práce při tvorbě malého projektu.
- Naučit se základům tvorby dokumentace projektu.
- Naučit se základům prezentace projektu a obhajobě dosažených výsledků.
- Seznámit se se základy anglické terminologie v oblasti algoritmizace.

Anotace

- Přehled základních datových struktur a jejich použití.
- Principy dynamického přidělování paměti.
- Specifikace abstraktních datových typů (ADT).
Specifikace a implementace ADT: seznamy, zásobník, fronta, pole, vyhledávací tabulka, graf, binární strom.
- Algoritmy nad binárním stromem.
- Vyhledávání: sekvenční, v neseřazeném a seřazeném poli, vyhledávání se zarážkou, binární vyhledávání, binární vyhledávací strom, vyvážený strom (AVL).
- Vyhledávání v tabulkách s rozptýlenými položkami.

- Řazení, principy, řazení bez přesunu položek, řazení podle více klíčů.
- Nejznámější metody řazení:
 - Select-sort,
 - Bubble-sort,
 - Heap-sort,
 - Insert-sort a jeho varianty,
 - Shell-sort,
 - Quick sort v rekurzivní a nerekurzivní notaci,
 - Merge-sort, List-merge-sort,
 - Radix-sort.

- Sekvenční metody řazení.
- Rekurze a algoritmy s návratem.
- Vyhledávání podřetězců v textu.
- Dokazování programů, tvorba dokázaných programů.

Přibližný rozpis přednášek

1. Algoritmický jazyk. Přehled datových struktur. Asymptotická časová složitost.
2. Abstraktní datový typ a jeho specifikace. Specifikace, implementace a použití ADT seznam.
3. Specifikace, implementace a použití ADT zásobník, fronta. Vyčíslení výrazů s použitím zásobníku.
4. ADT pole, množina, graf, binární strom.
5. Algoritmy nad binárním stromem.
6. Vyhledávání, sekvenční, v poli, binární vyhledávání.
7. Binární vyhledávací stromy, AVL strom.
8. Vyhledávání v tabulkách s rozptýlenými položkami.
9. Řazení, principy, bez přesunu, s vícenásobným klíčem.
10. Známé metody řazení polí
11. Známé metody řazení polí, řazení souborů.
12. Rekurze, algoritmy s návratem.
13. Dokazování správnosti programů, tvorba dokázaných programů.

Další studijní zdroje

- Studijní opora (cca 160-180 stran)
- Skripta Honzík, Hruška, Máčel: „Vybrané kapitoly z programovacích technik“ v pdf na webu
- Sedgewick, R.: Algoritmy v C. (Základy. Datové struktury. Třídění. Vyhledávání.) Addison Wesley 1998. Softpress 2003.

Doporučená literatura

- Rozsáhlá webová stránka s řadou příspěvků, článků, výtahů, námětů a odkazů v Pascalu, v C i v jiných jazycích.
- **Dr.Dobbs CD-ROM: Algorithms and Data Structures.** (CD ROM s 10 knižními publikacemi z oblasti algoritmů. Možno zapůjčit v knihovně FIT VUT v Brně).
- Kučera Luděk: Algovize aneb procházka Krajinou algoritmů, Univerzita Karlova 2009, ISBN 978-80-902938-5-4; www.algovision.org

Chyby ve studijní opoře

- Text studijní opory vznikl v krátkém časovém období s využitím dříve vytvořených elektronických zdrojových textů. To může být častou příčinou chybných konstrukcí a překlepů. Za elektronickou zprávu o závažnějších chybách může být udělen prémiový bod. Posílejte je na adresu: honzik@fit.vutbr.cz
- Za zprávy o případných chybách a překlepech ve slajdech se body neudělují!

Některé zvyky

- Konzultace Honzík: pátek 12.00-14.00 Po předchozí domluvě (možno emailem)
 - Prosím o uvážlivost v kontaktech i elektronických (je vás skoro 400...)
 - Pozor na „Honzík řekl, že na písemce bude...”

Úvod do datových struktur a algoritmů

- Algoritmický jazyk
- Pojmy:
 - syntax
 - sémantika
 - pseudopříkaz
 - příkaz
- Komentář

Vybrané identifikátory-klíčová slova

- `begin, end` levá a pravá příkazová závorka
- `type, var` specif. typů a deklarace proměnných
- `integer, real, boolean, char` standardní jednod. typy
- `array, record, (file, set)` standardní strukt. typy
- `for, to, downto, while, repeat, until`
vybraná slova pro tvorbu cyklů
- `if, then, else, case, of` konstrukce altern. příkazů
- `procedure, function` dekl. procedury resp. funkce
- `div, mod, not, and, or` aritmetické a logické operátory
- `false, true` literály booleovských hodnot

Datové typy a struktury

- Datové typy se dělí na **jednoduché**, **strukturované** a **typ ukazatel**. Nad všemi datovými typy je definována operace přiřazení "**:=**". což je varianta symbolu "**←**". (Rozdíl od jazyka C, kde je "**=**").
- Podmínkou přiřazení je **kompatibilita vzhledem k přiřazení**
- Pro usnadnění zápisu bude algoritmický jazyk používat operace výměny hodnot (*swap*) ve tvaru "**B:=A**", která nahrazuje trojici příkazů "**P:=A; A:=B; B:=P**"

- Pro jednoduché datové typy vystačíme ve většině příkladů s ordinálními typy **integer**, **výčtovým typem** a **typy char** a **boolean**.
- Definice: Ordinální typ je jednoduchý typ, pro jehož každou hodnotu (s výjimkou největší) existuje právě jedna následující a (s výjimkou nejmenší) právě jedna předcházející hodnota.
- Definice: Výraz je předpis na získání hodnoty daného typu a je současně nositelem této hodnoty.
- Příklad: Zápis "5+7" je předpisem i nositelem hodnoty
- Typ integer: +, -, *, div, mod, (/)
- Výčtový typ(nahrazuje číselník – převodní tabulku)
TStav=(svobodny,zenaty,rozvedeny,vdovec)

Existují mechanismy pro změnu typu – funkce `trunc(r)`, `real(i)` aj. Většinou je nebudeme potřebovat

Typ znak - "char" je ordinální typ. Deklarace typu má tvar:
var

ch1, ch2:char;

Pro převod celočíselné hodnoty proměnné `i` z rozsahu 0..9 na znak (číslici) lze zapsat výraz:

`chr(ord('0') + i)`

Nebudeme používat „magických čísel“ v podobě číselných kódů.

Typ boolean je ordinální typ, který nabývá dvou hodnot, jejichž zápis je false a true. Na základě ordinality typu platí, že $\text{true} = \text{succ}(\text{false})$.

Deklarace proměnných typu boolean má tvar:

var

B1, B2, B3, B4: boolean;

Nad typem boolean jsou definovány operace:

negace

"not"

logický součin (konjunkce)

"and"

logický součet (disjunkce)

"or,,

Zkratové vyhodnocení Booleovských výrazů:

B1 and B2 and B3 and ..and BN, pro $(B_i = \text{false}) \rightarrow$ celý výraz false

B1 or B2 or B3 or or BN, pro $(B_i = \text{true}) \rightarrow$ celý výraz true

- **Definice:**
- Skalární typ je jednoduchý typ, pro jehož každé dva prvky lze stanovit, zda jsou si rovny nebo zda je jeden z nich menší nebo větší než druhý.
- Pro operace relace budeme používat tyto dyadické *) operátory
- ekvivalence (rovno) =
- nerovno \neq nebo v kódu programu častěji $<>$
- menší než $<$
- menší roven \leq nebo v kódu programu častěji $<=$
- větší než $>$
- větší roven \geq nebo v kódu programu častěji $>=$

*) dyadický (z řečtiny) totéž co binární (z latiny)

Strukturované datové typy.

Strukturovaný datový typ sestává z komponent jiného (dříve definovaného) typu, kterému říkáme **kompoziční typ**. Pokud jsou všechny komponenty strukturovaného typu stejného kompozičního typu, říkáme, že strukturovaný typ je **homogenní**. Komponentám homogenního typu se někdy říká **položky** (*item*), zatím co komponentám heterogenního typu se říká **složky** (*component*). Strukturovaný typ má **strukturovanou hodnotu** (*structured value*). Tato hodnota je definovaná tehdy, když jsou definované hodnoty všech jejích komponent.

Komponentou strukturovaného typu může být jiný, dříve definovaný strukturovaný typ. To vede k možnosti vytvářet hierarchické a rekurzivní struktury.

Pole - array

Pole (array) je homogenní ortogonální (pravoúhlý) datový typ. Jednorozměrné pole je **vektor**, dvojrozměrné je **matice**.

Typ pole je specifikován **rozsahem svých dimenzí** (rozměrů) a komponentním typem. Pole, jehož jméno a velikost dimenzí jsou pevně dána deklarací a nemohou se měnit v průběhu programu, se nazývá **statické**.

Pokus o přístup k prvku pole, jehož index je mimo **rozsah**, způsobí chybu.

type

TVektor = array[1..100] of integer; (* vektor sta celých čísel *)

TMaticeV = array[1..10] of TVektor; (* vektor deseti vektorů o 100 celých číslech, tedy matice 10*100 *)

TMaticeP = array[1..10,1..100] of integer; (* matice o deseti řádcích a 100 sloupcích celých čísel *)

Jazyky blízké stroji, používají většinou jen horní hodnotu rozsahu dimenze (s implicitní spodní hranicí rovnou nule). V jazycích vyššího typu se používá i dolní hranice, nejčastěji s hodnotou 1. Uvědomme si, že dva vektory:

TVekt0 = array[0..9] of integer;

TVekt1 = array[1..10] of integer;

mají stejný počet prvků.

Algoritmu, v němž postupně projdeme (a postupně stejným způsobem zpracujeme) všechny prvky homogenní datové struktury, říkáme průchod. Pro práci s maticí jsou nejtypičtější průchody **"po řádcích"** a **"po sloupcích"**. Při průchodu po řádcích se pro každý řádek postupně mění hodnoty sloupcového (druhého, pravého) indexu. Až se vyčerpají všechny jeho hodnoty, zvýší se hodnota řádkového (prvního, levého) indexu. Obecně říkáme, že **při průchodu vícerozměrným polem "po řádcích", se rychlost změny indexu snižuje zprava doleva.**

Řetězec (*string*) je strukturovaný homogenní datový typ. Položkami řetězce je typ **znak** a řetězec má vlastnosti podobné jednorozměrnému poli znaků. Nad typem řetězec je definován bohatý repertoár operací.

Konstruktor je operace, která dovoluje ustavit strukturovanou hodnotu strukturovaného typu výčtem všech jeho komponent.

Nad řetězcem je definován konstruktor.

Příklad: `s:='CANON'` je totéž jako:

`s[1]:='C'; s[2]:='A'; s[3]:='N'; s[4]:='O'; s[5]:='N';`

Záznam (*record*) je obecně strukturovaný **statický** **heterogenní** datový typ. Jeho komponenty mohou být libovolného, dříve definovaného typu a často jim říkáme **složky** (*component*).

Jména a počet jeho komponent jsou dány při specifikaci typu a nemohou se měnit v průběhu programu. Protože komponentou záznamu může být i jiný strukturovaný typ, lze vytvářet záznamy s obecně neomezenou hierarchickou strukturou.

type

TDatum=record (* Záznam typu data v roce *)
rok,den,mesic:integer;
end;

TOsoba=record (* osoba se jménem a datem narození *)
jmeno:string;
datnar: TDatum
end;

var

Os1, Os2:TOsoba;
Dat:TDatum;
roknar:integer;

...

Dat.rok:=1990; (* ustavení roku data narození *)

Dat.mes:=2; (* ustavení měsíce data narození *)

Dat.den:=29; (* ustavení dne data narození *)

Os1.jmeno:='Novak'; (* ustavení jména osoby *)

Os1.datnar:=Dat; (* ustavení data narození osoby *)

Os1.datnar.rok:=1950; (* oprava data narození osoby *)

rok nar:=Os2.datnar.rok; (* uložení roku narození osoby Os2
do proměnné integer rok nar*)

...

Záznam je klíčová datová struktura pro tvorbu dynamických datových struktur. Umožňuje v jedné struktuře existenci pracovních položek i ukazatelů.

Typ ukazatel je spjat s dynamickými typy. Prvek dynamického typu vzniká v průběhu výpočtu jako výsledek operace (procedury) "**new (Uk)**", která v části paměti určené po dynamické struktuře vyhradí místo pro prvek (podle jeho velikosti) a ukazateli Uk přiřadí hodnotu, která umožní přístup k tomuto místu. Vytvořený prvek se zruší operací dispose. **Hodnota ukazatele použitého v dispose není definovaná!**

Dynamický prvek nemá jméno a jeho hodnota se tedy nezpřístupňuje identifikátorem, ale prostřednictvím ukazatele.

Pro specifikaci typu ukazatel a pro referenci (zpřístupnění) dynamického prvku budeme používat zápisu s pascalovským symbolem "**^**" (tzv. "šipka-notace").

Typ ukazatel má zvláštní hodnotu označovanou jako "**nil**", která má sémantiku "**ukazatel, který neukazuje na žádný prvek**" nebo "**prázdný ukazatel**". Hodnota "**nil**" je kompatibilní se všemi hodnotami typu ukazatel.

Příkl.

type

```
TUkUz=^TUzel; (* TUzel je záznam uzlu bin. stromu *)
TUzel=record                                     (* Specifikace typu Uzel *)
    DataUzlu:string; (* data uzlu *)
    LUK,PUK:TUkUz  (* levý a pravý uk. na uzly synů *)
end;
```

...

var

```
UkKoren:TUkUz; (* statická proměnná typu ukazatel *)
```

....

...

```
new(UkKoren);
```

```
(* vytvoření dynamického prvku, zpřístupňovaného  
statickým ukazatelem UkKoren *)
```

```
UkKoren^.DataUzlu:='KAREL';
```

```
(* reference dynamického prvku s přiřazením hodnoty typu  
string složce DataUzlu *)
```

...

```
dispose(UkKoren); (* zrušení prvku zpřístupňovaného  
ukazatelem UkKoren *)
```

Příkazy

Příkazy lze rozdělit na **jednoduché, strukturované a příkazy procedury**.

Jednoduché příkazy. Nejdůležitějším jednoduchým příkazem je **přiřazovací příkaz**. Přiřazovací příkaz přiřazuje proměnné na levé straně hodnotu na pravé straně přiřazovacího operátoru ":=". Proměnná i výraz musí být vzájemně **kompatibilní vzhledem k přiřazení**. Pro účely algoritmického jazyka postačí zjednodušená definice.

Definice (zjednodušená) :

Dva prvky jsou kompatibilní vzhledem k přiřazení, když jsou oba identického (ekvivalentního) typu.

Prázdný příkaz má postavení příkazu a neodpovídá mu žádná činnost ve fázi výpočtu. Na základě definice středníku, jako oddělovače dvou příkazu, lze zápis:

... ; ; ...

interpretovat, jako prázdný příkaz oddělený dvěma středníky a zápis:

... ; end

jako prázdný příkaz mezi středníkem a pravou příkazovou závorkou end.

Strukturované příkazy

Mezi strukturované příkazy algoritmického jazyka jsou zařazeny **složený příkaz, alternativní příkaz a příkaz cyklu**.

Složený příkaz je definován jako sekvence příkazů oddělených středníky, uzavřená mezi příkazové závorky `begin` a `end`.

Alternativní příkazy jsou: **podmíněný příkaz, alternativní příkaz (úplný podmíněný příkaz) a vícečetný alternativní příkaz case**.

Podmíněný příkaz je příkaz, který se provede, když má jeho booleovský řídící výraz hodnotu true.

Příkl.:

```
if B  
then S
```

Příkaz S se provede, když má booleovský výraz B hodnotu true; v jiném případě má příkaz význam prázdného příkazu. Příkazem S může být libovolný (jeden) příkaz (např. složený příkaz).

Alternativní příkaz (úplný podmíněný příkaz) provede jednu ze dvou alternativ v závislosti na hodnotě řídicího booleovského výrazu B

Příkl:

```
if B
then S1
else S2
```

V případě, že B má hodnotu true, provede se příkaz S1, jinak se provede příkaz B2.

Pozn. Beprostředně před "else" se nesmí vyskytnout středník. Středník odděluje dva příkazy a mezi "then" a "else" smí být jen jeden příkaz (byť strukturovaný).

begin

S (* příkaz *)

end else begin

S (* příkaz *)

Konvence zápisu ve tvaru:

if B

then end (* if B *)

tomu zabraňuje.

Vícečetný alternativní příkaz case je řízen **výrazem** ordinálního typu. Použití řídí. Více ve studijní opoře (SO).

Příkazy cyklu

Příkazy cyklu dělíme na **počítané** (explicitní), **nepočítané** (implicitní) a **nekonečné**.

Počítaný cyklus (for) se použije v případě, kdy můžeme explicitně stanovit počet jeho opakování. Řídicí proměnná (počítadlo) je ordinárního typu a pascalovské pojetí předpokládá jeho inkrementaci (zvýšení na hodnotu následníka) nebo dekrementaci snížení na hodnotu předchůdce). Vlastní průchod cyklem (jedno provedení těla cyklu, smyčka) má podobu jednoho příkazu uvedeného za vybraným slovem do. Po skončení příkazu cyklu provedením jeho posledního průchodu se předpokládá **nedefinovaná hodnota řídicí proměnné**.

Příklad:

```
for i:=1 to 100 do begin
```

```
    Sum:=Sum + Pole[i]  
end; (* for *)
```

...

```
for j:= 50 down to 30 do begin  
    if Pole[i] >= 0  
    then Sum:=Sum + Pole[i]  
    else Sum:= Sum - Pole[i]  
end;
```

Nepočítané cykly se dělí podle toho, kdy se provádí test na konec cyklu.

Cyklus while (označovaný také jako cyklus "nula nebo n krát"), provádí test před začátkem průchodu cyklem, na základě hodnoty booleovského výrazu B. Cyklus se provede, pokud má výraz B hodnotu "true". Má následující syntaktické schéma:

```
while B do S
```

Ve většině případů předchází cyklu typu while nastavení počátečních podmínek cyklu.

Příklad:

```
i:=1; (* nastavení počítadla *)  
F:=1; (* nastavení multiplikační proměnné *)  
while F < 100 do begin  
  (* Výpočet nejmenší hodnoty faktoriálu větší než 100 *)  
  i:=i+1  
  F:=F*i;  
end (* while *)  
(* F = i! *)
```

Cyklus repeat - until (označovaný také jako „cyklus jedna nebo n krát“), provádí test na konci průchodu a jeho tělo se musí provést alespoň jednou. Cyklus se ukončí, je-li hodnota řídicí proměnné B rovna "true". V některých případech, kdy je z povahy algoritmu zřejmé, že se cyklus provede nejméně jednou, může vést použití cyklu "repeat - until" ke kratšímu a jednoduššímu zápisu. Cyklus má následující syntaktické schéma:

```
repeat  
  S  
until B
```


Příklad:

i:=1; (* nastavení počítadla *)

F:=1; (* nastavení multiplikační proměnné *)

repeat

(* Výpočet nejmenší hodnoty fakt. větší nebo rovné 100 *)

 i:=i+1;

 F:=F*i

until F>=100

(* F=i! *)

Příkaz with se tváří jako cyklus, má ale spíše vlastnost složeného příkazu ohraničujícího sekvenci příkazů, na něž se vztahuje zjednodušení přístupu ke složkám jednoho záznamu.

Příkaz má podobu:

```
with I do  
begin S  
end
```

kde I je identifikátor proměnné typu záznam. Pak ve všech příkazech v rámci těla tohoto "cyklu" se nemusí složky záznamu I zapisovat s referenční předponou "I".

Budeme ho používat k zjednodušení zápisu.

Příklad: Místo zápisu

begin

Osoba.jmeno:='NOVAK';

Osoba.RokNar=:1980;

Osoba.vyska:=180;

Osoba.vaha:=78

end;

Ize zapsat jednodušší zápis:

with Osoba do

begin

jmeno:='NOVAK';

RokNar=:1980;

vyska:=180;

vaha:=78

end; a vyhnout se tak opakovanému zápisu stejného identifikátoru (Osoba) téhož záznamu.

Grafická úprava zápisu - odsazování (indentation)

```
for i:= 1 to 10 do begin (* cyklus pro
    aktivitu A *)
    (* tělo cyklu *)
end (* for i:= *);
while B do begin      (* cyklus pro aktivitu B *)
    (* tělo cyklu *)
end (* while B *);
repeat      (*cyklus pro aktivitu C *)
    (* tělo cyklu *)
until Konec;
```

Procedury a funkce

Procedura (funkce) je nástroj zvyšování abstrakce příkazů (výrazů). Má formu "uzavřeného podprogramu"; to znamená, že na místě každého použití (volání) procedury (funkce) se překladem generuje skok do podprogramu, který je obecně umístěn na jiném místě paměti, než hlavní program. Po provedení podprogramu se řízení vrací za skok do podprogramu.

Pro použití procedury se v sekci deklarace uvede "předpis" na provedení procedury, který sestává z "hlavičky" a z "těla" procedury (funkce). V hlavním programu se procedura "vyvolá" **příkazem procedury**. Funkce se vyvolá **zápisem funkce** na místě výrazu.

Hlavička procedury (funkce) sestává z vybraného slova "**procedure**" ("**function**"), identifikátoru a seznamu formálních parametrů. Formální parametry jsou odděleny (po skupinách parametrů téhož typu) středníkem. Formální parametry předávané odkazem jsou uvedeny vybraným slovem "**var**". Ostatní parametry jsou předávané hodnotou.

Příklad:

```
procedure NejdNeklesPosl (var Pole:TPole; pocet:integer; var  
zacatek, delka:integer);
```

Určení délky nejdelší neklesající posloupnosti (NNP) hodnot po sobě jdoucích prvků typu integer v jednorozměrném poli **Pole**. Pole má "**pocet**" prvků ($pocet > 0$). Výsledná nejdelší neklesající posloupnost začíná na indexu "**zacatek**" a má "**delka**" prvků. V případě více shodných NNP je výsledkem první NNP v pořadí od začátku pole.

Tělo procedury má tvar bloku, tzn., že sestává z nepovinné úvodní sekce dovolující specifikaci konstant, typů a deklaraci proměnných a procedur (funkcí) následovanou složeným příkazem.

Příklad (tělo procedury NejdNeklesPosl):

```
var    (* deklarace lokálních proměnných *)
    PomDel, PomZac: integer; (* pom. délka a začátek *)
    i, h: integer;          (* počítadlo a pracovní hodnota*)
begin  (* vlastní tělo procedury *)
    (* inicializace proměnných *)
    delka:=1;
    zac:=1;
    PomDel:=1
    . . . . . (* Viz Studijní Opora *)
end;
```

·* Vyvolání procedury má postavení příkazu a sestává ze **jména** procedury a **seznamu skutečných parametrů**.

Skutečné parametry musí souhlasit s formálními co do **počtu, pořadí i typu**. Skutečné parametry, odpovídající formálním parametrům předávané odkazem, jsou identifikátory proměnných odpovídajícího typu. Skutečné parametry předávané hodnotou, jsou výrazy odpovídajícího typu. Tato vlastnost je typická pro jazyky se **silnou kontrolou typů**.

Funkce má vlastnosti procedury, ale má postavení **výrazu** s hodnotou, jejímž nositelem je **vyvolání (zápis) funkce**.

Hlavička funkce je doplněna identifikátorem typu hodnoty funkce. V algoritmickém jazyce může funkce nabývat hodnot typů: "**integer**", "**real**", "**char**", "**boolean**", "výčtového typu" a typu "**string**". V těle funkce musí být alespoň jeden příkaz, v němž se jménu funkce přiřazuje hodnota. Hodnota, která je přiřazena při provedení těla funkce jménu funkce, je **vracená výsledná hodnota funkce**. Jméno funkce se smí v těle funkce použít jen na místě proměnné (na levé straně přiřazovacího výrazu nebo jako výstupní parametr procedury). Na jiném místě se interpretuje jako rekurzivní volání této funkce.

Volba, zda daný algoritmus nebo část programu má mít formu procedury nebo funkce je dána zkušeností a programovacím stylem programátora.

Funkci volíme nejčastěji tehdy, když výsledkem úseku je jedna hodnota jednoduchého typu (jakým může být funkce).

Velmi častým použitím je funkce typu boolean.

Vícenásobné volání téže funkce se stejnými parametry je neefektivní.

Rekurzivní volání procedury (funkce) je konstrukce, při níž procedura (funkce) je volána v těle sebe samé. Rekurse je účinným nástrojem zápisu řady algoritmů.

Rekurzivní volání funkce má vždy tvar výrazu, který je součástí (nejčastěji přiřazovacího) příkazu.

Definice: Vyvolání procedury (funkce) má stejný účinek, jako kdyby na tomto místě bylo tělo procedury, v němž jsou všechny formální parametry předávané odkazem nahrazeny odpovídajícími skutečnými parametry a všem formálním parametrům předávaným hodnotou jsou přiřazeny hodnoty jim odpovídajících skutečných parametrů před provedením prvního příkazu těla procedury.

Parametry jsou významným prostředkem k předávání informace mezi procedurou (funkcí) a jejím okolím.

Procedury "**bez parametrů**" jsou výjimkou a je účelné se jim vyhýbat!

Seznam parametrů vytváří rozhraní (*interface*) a je jako "konektor", kterým je "černá skříňka" procedury připojena k programu.

Existují případy, kdy jsou rozumné důvody pro použití globálních proměnných. Je na to ale vždy nutno upozornit v komentáři procedury! Komunikace mimo parametry má často podobu "**vedlejšího jevu**".

Vedlejší jev (*side effect*) je pojem, kterým se označuje změna hodnoty globální proměnné uvnitř těla procedury. Častým případem je vstupní parametr předávaný odkazem.

Vedlejší jev je většinou nežádoucí a nebezpečný. Více si přečtete v SO. Budeme se mu vyhýbat

Základní pojmy složitosti algoritmů

Časová a prostorová složitost algoritmu.

(**Nezaměňujeme pojem časová složitost s dobou běhu programu!**)

Asymptotická časová složitost je nejčastějším hodnoticím kritériem pro algoritmy. Vyjadřuje se porovnáním algoritmu s jistou funkcí pro N blížícímu se nekonečnu. Porovnání má podobu tří různých složitostí:

- O Omikron (velké O , \mathcal{O} , *big O*) - horní hranici chování
- Ω Omega - dolní hranici chování
- Θ Theta - vyjadřuje třídu chování

Pozn. V následujícím textu budeme pro kvantifikátory používat následující identifikátory:

$\exists \rightarrow Exist$

$\forall \rightarrow ForAll$

Složitost Omikron

Složitost vyjádřenou zápisem **Omikron** (nebo také Big O, 'O) vyjadřuje **horní hranici** časového chování algoritmu.

Omikron ($g(n)$) označuje **množinu funkcí $f(n)$** , pro které platí:

$$\{f(n) : \textbf{Exist } (c>0, n_0>0) \text{ takové, že } \textbf{ForAll } n \geq n_0 \text{ platí } [0 \leq f(n) \leq c \cdot g(n)]\}$$

kde **c** a **n_0** jsou určité vhodné kladné konstanty.

Pak zápis **$f(n) = \text{Omikron}(g(n))$** , nebo **$O(g(n))$**

označuje, že funkce $f(n)$ je rostoucí **maximálně tak rychle** jako funkce $g(n)$. Funkce $g(n)$ je horní hranicí množiny takových funkcí, určené zápisem **$\text{Omikron}(g(n))$** nebo **$O(g(n))$** .

Složitost Omega

$\Omega(g(n))$, nebo také $\Omega(g(n))$ označuje množinu všech funkcí $f(n)$ pro které platí:

$\{f(n) : \textbf{Exist } (c>0, n_0>0) \text{ takové, že } \textbf{ForAll } n \geq n_0$
platí $[0 \leq c \cdot g(n) \leq f(n)]\}$

kde c a n_0 jsou určité vhodné kladné konstanty.

Pak zápis $f(n) = \Omega(g(n))$ nebo $\Omega(g(n))$ označuje, že funkce $f(n)$ je rostoucí minimálně tak rychle, jako funkce $g(n)$. Funkce $g(n)$ je dolní hranicí množiny všech funkcí, určených zápisem $\Omega(g(n))$ nebo $\Omega(g(n))$.

Složitost Theta

$\Theta(g(n))$ nebo $(\Theta(g(n)))$ označuje množinu všech funkcí $f(n)$, pro něž platí :

$\{f(n) : \textbf{Exist } (c_1 > 0, c_2 > 0, n_0 > 0) \text{ takové, že}$

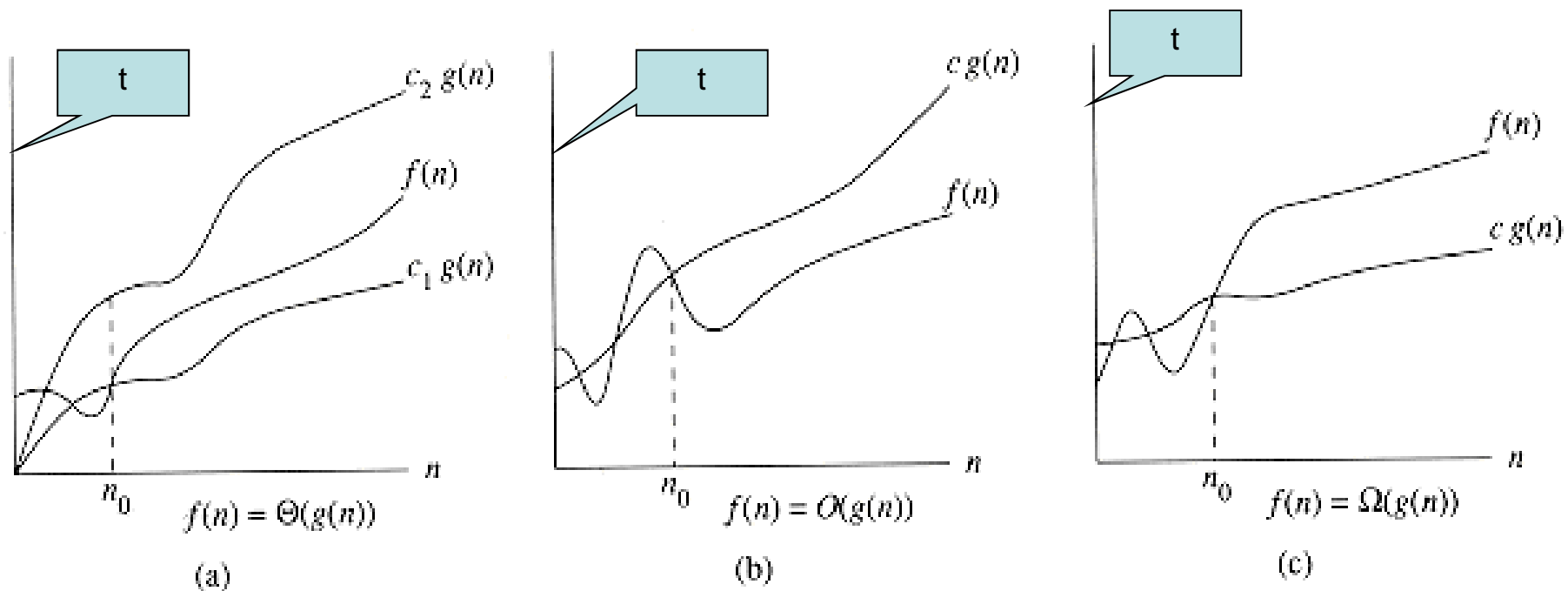
ForAll $(n \geq n_0)$ platí $[0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n)]\}$

kde c_1, c_2 and n_0 jsou vhodné kladné konstanty.

Složitost vyjádřená zápisem Theta (Θ) označuje časové chování shodné jako daná funkce.

Pak zápis $f(n) = \Theta(g(n))$ nebo $\Theta(g(n))$ označuje, že funkce $f(n)$ roste tak rychle jako funkce $g(n)$. Funkce $g(n)$ vyjadřuje horní a současně dolní hranici množiny funkcí, označených zápisem $\Theta(g(n))$ nebo $\Theta(g(n))$.

Grafické vyjádření složitosti



Časová složitost – vliv řádu algoritmu a kardinality úlohy

Algoritmus N	$33n$	$46 n \log n$	$13 n^2$	$3.4 n^3$	2^n
10	0.00033 s	0.015 s	0.0013 s	0.0034 s	0.001 s
100	0.0033 s	0.03 s	0.13 s	3.4 s	$4 \cdot 10^{14}$ století
1000	0.033 s	0.45 s	13 s	94 hod	
10 000	0.33 s	6.1 s	22 min	39 dní	
100 000	3.3 s	1.3 min	1.5 dní	108 roků	
Maximální velikost n pro čas					
1 s	30 000	2 000	280	67	20
1 min	1 800 000	82 000	2 200	260	26

Řád a konstanta

	CRAY – 1 Fortran	TRS-80 Basic
n	$3n^3$	19 500 000 n
10	$3 \times 10^{-6} \text{ s}$	$200 \times 10^{-3} \text{ s}$
100	$3 \times 10^{-3} \text{ s}$	2 s
1 000	3 s	20 s
2 500	50 s	50 s
10 000	49 min	3.2 min
1 000 000	95 let	5.4 hod

Klasifikace algoritmů podle časové složitosti

1. $\Theta(1)$ je označení algoritmů s konstantní časovou složitostí.
2. $\Theta(\lg(n))$ je označení algoritmů s logaritmickou časovou složitostí. Základ logaritmu není podstatný, protože různé základy se liší pouze konstantou vzájemného převodu. Touto složitostí se vyznačují např. rychlé vyhledávací algoritmy.
3. $\Theta(n)$ je označení algoritmů s lineární časovou složitostí. Tuto složitost mají běžné vyhledávací algoritmy a řada algoritmů sekvenčně zpracovávajících datové struktury.
4. $\Theta(n \cdot \lg(n))$ je označení algoritmů nazvané také **“linearitmické”**. Tuto časovou složitost mají např. rychlé řadicí algoritmy, založené na modelu porovnávání vyhledávání.

5. **Theta(n^2)** je označení algoritmů s kvadratickou časovou složitostí. Takovou časovou složitostí se vyznačují algoritmy sestavené z dvojnásobného počítaného cyklu do n . Patří mezi ně řada klasických řadicích algoritmů (např. bublinové řazení).
6. **Theta(n^3)** je označení algoritmů s kubickou časovou složitostí. Algoritmy s touto časovou složitostí jsou prakticky použitelné především pro málo rozsáhlé problémy. Kdykoli se n zdvojnásobí, čas zpracování je osminásobný.
7. **Theta(k^n , $k \exp n$)**, kde k je přirozené číslo, je označení algoritmů s exponenciální (pro $k=2$ binomickou) časovou složitostí. Existuje několik prakticky použitelných algoritmů s touto složitostí. Velmi často se označují jako algoritmy pracující s “hrubou silou” (brute-force algorithms). Kdykoli se n zdvojnásobí je čas řešení kvadrátem.