Algoritmus

Algoritmus je konečná, uspořádaná množina úplně definovaných pravidel pro vyřešení nějakého problému.

Data

jsou opakovaně interpretovatelná formalizovaná podoba informace vhodná pro komunikaci, vyhodnocování nebo zpracování.

Informace

je poznatek (týkající se jakýchkoliv objektů, např. fakt, událostí, věcí, procesů nebo myšlenek, včetně pojmů), který má v daném kontextu specifický význam

Program

Program je předpis, podle kterého je počítač (nebo jiný výkonný prvek - procesor) schopen provádět výpočty nějakého algoritmu.

Procesor

Procesor je prvek, kterému je svěřeno vykonávání algoritmu.

Strategie řešení problémů

Dekompozice

Dekomponovat znamená najít v složitém problému takové hierarchické uspořádání, které umožní zapsat složité akce pomocí akcí jednodušších. Ty mohou být stejným způsobem redukovány na akce ještě jednodušší. *(metoda shora dolů)*

Abstrakce

Abstrakce je koncepční zjednodušení složitého problému ignorováním detailů.
Pojmenováním akcí a dočasným ignorováním detailů je možné celý složitý problém řešit po částech. Abstrakce tedy umožňuje oddělit jádro řešeného problému od detailů. *(metoda zdola nahoru)*

Datová abstrakce

S daty lze manipulovat bez znalosti jejich vnitřní implementace.

Konstrukci programů lze logicky rozdělit na dvě části:

1. **Datové struktury** - specifikace dat

2. **Řídící struktury** - algoritmy

Vlastnosti algoritmů

Rezultativnost = konečnost

Úloha musí být vyřešena po konečném počtu kroků.

Každý algoritmus musí někdy skončit.

Hromadnost = obecnost

Jedním algoritmem lze řešit celou třídu úloh stejného druhu.

Algoritmus musí vyřešit zadanou úlohu pro libovolně zadané vstupní hodnoty.

Determinovanost

Algoritmus musí být zadaný ve formě konečného počtu jednoznačných pravidel. *(Program nesmí mít žádnou slepou cestu.)*

Efektivnost

Na správný průběh programu nemá žádný vliv, zajišťuje pouze to, aby program trval co nejkratší dobu

Vyjádření algoritmů

- Slovní popis
- Vývojové diagramy
- Rozhodovací tabulky *(seznam podmínek, seznam činností, kombinace)*

Strukturované programování

Zaznamenání algoritmu *(Dijkstra)*

1. **Sekvence** = posloupnost
2. **Selekce** = větvení *(if, if-else, switch)*
3. **Iterace** = cyklus *(while, do-while, for)*

Programovací jazyky

Syntaxe

soubor pravidel udávajícípřípustnékonstrukce programů

Sémantika

určuje logický význam jednotlivých výrazů jazyka

Syntaktické diagramy

Terminální symbol

Dále se nerozgenerovává Znak, klíčové slovo, ...

Zapisuje se v kroužku nebo oválku.

Nonterminální symbol

Rozgenerovává se - Dále se rozvádí podle diagramu.

Zapisuje se v obdélníku.

BNF = Backus-Naurova Forma

Textový zápis syntaxe, stejná vyjadřovací schopnost jako syntaktický diagram.

<> - Úhlové závorky značí nonterminály

= - Definiční znak

| - Odděluje jednotlivé varianty --> seznam variant

...

EBNF = Rozšířené BNF

Zjednodušuje zápis, ale nezvyšuje vyjadřovací schopnost.

{} - opakování 0 až *n*

[] - nepovinný řetězec symbolů

() - seskupování konstrukcí

Sémantika

viz. IAL - Sémantika ADT

Kompilátor – překladač, který přeloží celý zdrojový kód programu do jazyku stroje najednou

Interpret – překladač, který neprovádí přímý překlad. Překládá se za běhu programu

- program se může modifikovat za běhu X malá rychlost, náročnější na paměť, nutnost mít překladač

Paradigmata

Naivní – prostě naprogramovat jakýmkoliv způsobem (BASIC)

Procedurální = imperativní – sekvence po sobě jdoucích příkazů, důležitou rolu zde hraje přiřazovací příkaz (C,Pascal)

Funkcionální – postupné aplikování funkcí, aplikování funkcí na výsledky jiných funkcí. (Lisp)

Objektové orientované – základním prvkem jsou objekty komunikující prostřednictvím zpráv. Každý objekt má data a metody. (C++)

Logické – počítači je předložen program ve formě klauzulí (pravidla,tvrzení) . (ProLog)

Paralelní – rozdělení úlohy na podlohy, které se mohou vykonávat současně různými procesory. (SR)

2. Principy vyšších programovacích jazyků

Datové struktury

Pro stejné operace nad daty vedou některé struktury k více či méně efektivním algoritmům než jiné. Volba algoritmu a volba **datové struktury** jsou navzájem propojeny a správným výběrem se snažíme najít cesty k úspoře času a místa.

Jsou-li všechny komponenty dané struktury téhož typu, označujeme strukturu **homogenní**, v opačném případě **heterogenní**.

Statická datová struktura

Nemůže v průběhu výpočtu měnit počet svých komponent ani způsob jejich uspořádání.

Dynamická datová struktura

Může v průběhu výpočtu měnit počet svých komponent a způsob jejich uspořádání.

Vlastnosti datových struktur je třeba posuzovat na určité **úrovni abstrakce**. Na úrovni stroje jsou data vždy homogenní a statická.

Datové typy

Datový typ

Datový typ je definován množinou hodnot, které může nabývat a množinou operací nad ním definovaných.

Typ proměnné/konstanty je určen při její deklaraci/definici. Typ výrazu je dán operátory a operandy, které jsou ve výrazu použity.

Příslušnost k typu je **syntaktickou vlastností objektu**.

Kardinalita datového typu

Počet různých hodnot, které může datový typ nabývat.

Primitivní typ

Hodnoty primitivního typu lze definovat výčtem *(enumerací)*.

Standardní typy

Standardní typy jsou předdefinovány programovacím jazykem.

Strukturované datové typy

- pole
- struktura *(záznam)*
- množina
- soubor

Typový systém

Typovým systémem programovacího jazyka rozumíme soubor pravidel, která přiřazují výrazům typ.

Typový systém nazveme **silným**, pokud akceptuje pouze typově bezpečné výrazy. Typový systém, který není silný, se nazývá **slabý**.

Lexikální jednotky jazyka C

- Rezervované slovo
- Identifikátor
- Konstanta
- Řetězcový literál
- Oddělovač

Pojmenované konstanty

- Pomocí preprocesoru - `#define`
- Pomocí výčtu - klíčové slovo `enum`
- Pomocí konstantní proměnné - klíčové slovo `const`

Proměnné a deklarace

Datový objekt*(nesouvisí s OOP)* je obecné označení jakéhokoliv údaje uloženého v paměti.

Paměťová místa, ve kterých uchováváme data, označujeme **proměnné**.

Deklarace proměnné

Deklarace proměnné je konstrukce, která přidělí proměnné jméno a typ, ale nevytvoří ji.

Definice proměnné

Definice proměnné kromě jména a datového typu přidělí proměnné i paměťový prostor.

Inicializace proměnné

Přiřazení hodnoty při definování.

Operátory a výrazy

Výraz

Výraz je konstrukce jazyka, která má hodnotu. *(nějakého datového typu)*

Část výrazu, na kterou je aplikován jeden z operátorů, se nazývá **operand**. Operandem může být opět výraz, někdy se operandům říká **podvýrazy**.

Operátor určuje, jakým způsobem se z operandů získá hodnota. Pořadí vyhodnocování podvýrazů je dáno **prioritami operátorů** nebo závorkami.

Příkaz

Výraz se stává příkazem ukončením středníkem. Úkolem příkazu je vykonat nějaký kód.

Operátor přiřazení

Operátor přiřazení (=) kopíruje hodnotu na své pravé straně do *L-hodnoty* na své levé straně.

L-hodnota je objekt v paměti, kterému lze přiřadit hodnotu.

R-hodnota je výraz, který má hodnotu a vystupuje na pravé straně přiřazovacího příkazu.

Přiřazení je v jazyce C definováno jako **operátor**, výsledkem je tedy opět R-hodnota.

Operátor čárka

Tento operátor zřetězuje dva výrazy a zajišťuje přesné pořadí jejich vyhodnocení zleva doprava. Nabývá hodnoty nejpravějšího podvýrazu.

Aritmetické operátory

Unární - + a -

Binární - + - * / %

Speciální unární - ++ --

3. Řízení chodu programu

Prototyp funkce

Prototyp funkce deklaruje funkci před jejím použitím a před tím než je definována.

4. Typ ukazatel, strukturované datové typy

Ukazatel

Ukazatel je proměnná která obsahuje paměťovou adresu. Jeho hodnota říká, kde je nějaký objekt uložen.

Součástí deklarace ukazatele je informace o **typu dat**, která jsou na uložené adrese očekávána.
Pojmem **dereference ukazatele** rozumíme zpřístupnění objektu, na který ukazatel ukazuje.

Konstantní ukazatel - nelze měnit, odkazovanou paměť ano (int * const p_x)

Ukazatel na konstantu - lze měnit, odkazovanou paměť ne (const int * p_x)

Konstantní ukazatel na konstantu - const int * const p_x

Obecný ukazatel - lze přetypovat na libovolný typ, neumožňuje typovou kontrolu (void *)

Konverze ukazatele - přetypování

Ukazatelová aritmetika - p_x+1 == p_x + sizeof(int)

Vícenásobná dereference - i=12; int *p_i=&i; int **p_p_i=&p_i; int j=**p_p_i;

Paměťový prostor

- Kódová oblast - kód programu
- Datová oblast - globální proměnné + konstanty
- Hromada - dynamické proměnné (malloc)
- Zásobník - lokální proměnné + parametry funkcí

Operátor `sizeof()`

Únik paměti ***(memory leak)***

Pole

Pole je kolekce proměnných stejného typu. Jednotlivá proměnná v poli se nazývá **prvek pole**. K prvkům pole se přistupuje prostřednictvím identifikátoru pole a **indexu**.

Rozměr globálního pole je nutné zadat prostřednictvím literálu.

Vícerozměrné pole

Vícerozměrné pole řádu *n* je v jazyce C definováno jako jednorozměrné pole řádu *n-1*.

Použití ukazatelů pro práci s poli

Použití samotného identifikátoru pole má stejný význam jako použití konstantního ukazatele na první prvek pole.

Hodnotu ukazatele tvořeného jménem pole nelze měnit.

5. Výčtový typ, strukturované datové typy

Specifikátor typedef

Specifikátor `typedef` slouží k vytvoření nového označení datového typu. Používá se pro zvýšení přehlednosti programu. Pozor: `typedef` není operátor, ale **specifikátor**.

Výčtový typ

V jazyce C lze definovat sadu pojmenovaných celočíselných konstant, nazývanou **výčet**. Výčtový typ je kompatibilní s celými čísly, tyto konstanty lze tedy používat jako celočíselné konstanty.

Datový typ struktura

Struktura je odvozený datový typ, který může obsahovat datové složky různého typu. Říkáme, že jde o **heterogenní** datový typ.

Kompatibilita struktur

Vytvoříme-li dvě struktury se stejnými složkami, budou to dva různé (nekompatibilní) typy.

Operátor přiřazení

Nad datovým typem struktura je definován operátor přiřazení. Obsah jedné struktury lze tímto operátorem přiřadit jiné, pokud jsou obě stejného typu. V tomto případě se provádí tzv. **mělká kopie**.

Struktura odkazující sama na sebe

Pokud musí struktura odkazovat sama na sebe, nelze k tomu použít identifikátor vytvořený pomocí `typedef`.

Předávání struktury

Návratovou hodnotou funkce může být struktura. Struktura může být také předána funkci jako parametr. Předávání (větší) struktury hodnotou je však neefektivní, proto se dává přednost **ukazateli na konstantní strukturu**.

Práce se soubory

Standardní vstup/výstup

Během inicializace programu v jazyce C se automaticky otevírají tři soubory: `stdin`, `stdout`, `stderr`. Standardní vstup/výstup se používá zejména u programů, které zpracovávají data **proudovým způsobem**. Takovým programům se někdy říká **filtry**.

Vstup/výstup znaků

```
int getchar (void);
int putchar (int);
```

Formátovaný výstup - funkce `printf()`

Formátovací specifikace začínají znakem `%`, za ním následují nepovinné parametry (šířka, přesnost, ...) a povinný parametr - **konverze** (`%d`, `%f`,`%s`, ...).

Formátovaný vstup - funkce `scanf()`

Stejné formátovací specifikace jako u `printf()`, všechny parametry se předávají odkazem.

Datový typ soubor

Se soubory se pracuje pomocí datového typu `FILE*`. Jazyk C rozlišuje dva druhy souborů - **textový** a **binární**. V textovém souboru lze pracovat s jednotlivými řádky textu nezávisle na typu OS.

Otevření a uzavření souboru

```
FILE *fopen (const char *name, const char *mode);
int fclose (FILE *file);
```

Mód otevření souboru

`"r"`	čtení
`"w"`	zápis nebo přepsání
`"a"`	připojení na konec
`"r+"`	čtení a zápis
`"w+"`	čtení, zápis nebo přepsání
`"a+"`	čtení a zápis na konec

Pro práci s binárním souborem se k řetězci, který udává mód, přidá prefix `"b"`

Souborový vstup/výstup po znacích

```
int getc (FILE*);
int putc (int, FILE*);
```

Formátovaný vstup/výstup nad soubory

```
int fscanf (FILE *, const char *, ...);
int fprintf (FILE *, const char *, ...);
```

Souborový vstup/výstup po řádcích

```
char *fgets (char *dest, int max, FILE*);
int fputs (char *src, FILE* file);
```

Testování chyb

```
// Vrací 0, pokud nedošlo k chybě
int ferror (FILE *);
```

Konkrétní kód chyby lze přečíst z globální proměnné `errno` typu `int` deklarované v `errno.h`.

6. Řešení rekurentních problémů

Pokud problém vede na iteraci, kde každý krok iterace závisí na výsledku z předchozí iterace, říkáme problému **rekurentní**.

Rekurentní vztah

Následující hodnota je funkcí hodnot předchozích. Musí existovat konečný počet iterací, po jejichž provedení výpočet skončí.

Posloupnosti

`B(y)` - predikát, který říká, že výsledná hodnota splňuje podmínky (např. přesnost).

```
y = y0;

while (!B(y))

    y = F(y);
```

Algoritmické schéma

Algoritmickou konstrukci, ve které jsou uvedeny pouze symboly proměnných, funkcí a predikátů, nazýváme algoritmické schéma.

Řady

Aproximace funkcí

Přibližný výsledek funkce se aproximuje částečným součtem řady. Ukončení algoritmu je založeno na požadavku na **přesnost aproximace**. Ta je vyjádřena zpravidla přírůstkem, tj. posledním členem řady v dané iteraci.

Heuristika

Vstupní data pro aproximaci funkce je potřeba předzpracovat tak, aby vlastní výpočet probíhal v tom nejvýhodnějším intervalu. Opatřením, které vedou ke snížení náročnosti výpočtu a ke zvýšení efektivity, se říká **heuristika**.

7. Rekurzivní metody v programování

Rekurze

Rekurze je způsob specifikace části počítačového programu odkazem na sebe.

IAL - rekurze

Každou rekurzi lze nahradit **iterací** a naopak. Iterace bývá často efektivnější, rekurzi lze zase někdy zapsat průzračnějším algoritmem.

Musí mít ukončovací podmínku. (stejně jako iterace)

Rekurzivní funkce

- **Přímá** – A volá A
- **Nepřímá** – A volá B, B volá C, C volá A

Rekurze a zásobník

Při volání funkce se na vrcholu zásobníku vymezí oblast potřebná pro:

- Lokální proměnné
- Parametry, se kterými byla funkce volána
- Výsledek funkce (pokud funkce vrací hodnotu)
- Informace pro správný návrat z funkce

Tato oblast se nazývá **Aktivační záznam funkce**.

Hanojské věže

1. Přeneseme *N-1* horních disků z A na B s použitím C jako odkládací jehly.
2. Přeneseme největší disk zbylý na A na jehlu C.
3. Přeneseme věž s *N-1* disky z B na C s použitím A jako odkládací jehly.

8. Složitost algoritmů

IAL - Složitost algoritmů

Dolní odhad složitosti

Dolní odhad složitosti určuje ideální složitost algoritmu, tj. jeho nejrychlejší možné provedení.

Průměrná složitost ***(očekávaná složitost)***

Průměrná složitost se spočítá jako střední hodnota náhodné složitosti při určitém rozložení vstupních dat.

Časová složitost

Potřebné množství času

Prostorová složitost

Potřebné množství paměti pro datové struktury

- O(1) – konstantní
- O(log N) – logaritmická – když se N zdvojnásobí, délka běhu se zvýši o konstantu
- O(N) – lineární – 2N -> délka běhu se také zdvojnásobí
- O(N*log N) – linearitmická – 2N -> delka běhu se více než zdvojnásobí¨
- $O(N^2)$ – kvadratická – 2N -> délka běhu 4x
- $O(N^3)$ – kubická – 2N -> délka běhu 8x
- $O(2^N)$ – exponenciální – 2N ->délka běhu vzroste s druhou mocninou

9. Algoritmy pro práci s vektory a s maticemi

Skalární součin

$$\vec{a} \cdot \vec{b} = c$$

$$c = \sum_{i=1}^{n} a_i b_i$$

Vektorový součin

$$\vec{w} = \vec{a} \times \vec{b}$$

$$\vec{w} = \left(\begin{vmatrix} a_2, & a_3 \\ b_2, & b_3 \end{vmatrix}, \begin{vmatrix} a_3, & a_1 \\ b_3, & b_1 \end{vmatrix}, \begin{vmatrix} a_1, & a_2 \\ b_1, & b_2 \end{vmatrix} \right)$$

Eratosthenovo síto

Eratosthenovo síto je efektivní algoritmus pro hledání všech prvočísel od *2* do *N*.

1. Vytvoříme bitové pole, pro každé přirozené číslo v rozsahu od *2* do *N* vyhradíme 1 bit. Index pole udává číslo, položka nabývá hodnoty 1, je-li číslo prvočíslo.

2. Pole je inicializováno samými jedničkami.
3. Procházíme prvky pole a hledáme první nenulový bit.
4. Nalezené číslo *(P)* je prvočíslo, necháme na jeho místě jedničku. Potom na místa všech násobků *P* zapíšeme nuly.
5. Cyklus opakujeme tak dlouho, dokud není P větší než odmocnina z *N*.

Matice

Soubor čísel uspořádaných do řádků a sloupců nazýváme maticí.

Matice je **nulová**, když jsou všechny její prvky rovny nule. Matice je **jednotková**, když jsou prvky hlavní diagonály rovny jedné a všechny ostatní prvky jsou rovny nule.
Transponovaná matice k matici `A` vznikne záměnou řádků matice `A` za sloupce, značíme `AT`.
Matice je **symetrická**, je-li čtvercová a platí `A == AT`.

Násobení matic

$$\mathbf{AB} = \mathbf{C} = (c_{ij}) = \left(\sum_{k=1}^{p} a_{ik} b_{kj} \right)$$

10. Algoritmy pro vyhledávání a řazení

IAL - Vyhledávání

Sekvenční vyhledávání

Vyhledávaná struktura se prochází sekvenčně.

Sekvenční průchod všemi položkami až do nalezení hledané položky nebo dosažení konce.

Index-sekvenční vyhledávání

(ISAM = Index Sequential Access Method) Přímým přístupem se nalezne blok a v něm se sekvenčně dohledá požadovaná položka.

Nesekvenční vyhledávání

Index prvku se nalezne rychleji než sekvenčním průchodem

Vyhledávání v uspořádaných stromech

Speciálně organizované struktury pro uložení prohledávaných dat.

Vlastnosti řadících algoritmů

Stabilita

Stabilita řazení je vlastnost algoritmu, který zachová relativní pořadí položek se stejnou hodnotou klíče.

Přirozenost

Pro přirozený řadící algoritmus platí, že doba řazení již uspořádané posloupnosti je menší, než doba řazení náhodně uspořádané posloupnosti a ta je menší, než doba řazení opačně seřazené posloupnosti.

T(seřazená) < T(náhodně uspořádaná) < T (opačně uspořádaná)

Bubble Sort

Bubble Sort je řazení na principu **výběru**.
Víme...

Metoda je **stabilní**, **přirozená** a má **kvadratickou** časovou složitost. Nejrychlejší metoda v případě již seřazené posloupnosti!

Ripple-sort

Ripple-sort si pamatuje polohu první výměny a díky tomu přeskočí dvojice, u nichž je jasné, že se nebudou vyměňovat.

Shaker-sort

Shaker-sort střídá směr probublávání (houpačková metoda).

Insert sort

Pole je rozděleno na dvě části - levou seřazenou a pravou neseřazenou. Na začátku tvoří levou část první prvek. V cyklu se berou prvky z neseřazené části a vkládají se do seřazené části na správné místo. Zbytek seřazené části se musí vždy o 1 posunout (vkládání do pole, víme...).

11. Dynamické datové struktury

Základní výhoda – proměnná velikost

Skládají se z prvků svázaných ukazateli

Lineární seznamy

Cyklické seznamy

Zásobník (LIFO – last in, first out)

Fronta (FIFO – first in, first out)

Stromy, obecný graf, hashovací tabulka

Zásobník

První prvek – vrchol zásobníku

Vhodné pro zpracování dat v opačném pořadí, než jsme je načetli

Fronta

Vložení prvku na konec fronty, výběr prvku ze začátku fronty

Vhodné pro zpracování dat, kde potřebujeme zachovat pořadí příchozích prvků (buffer)

Strom

Obsahuje kořen (začátek), každá položka obsahuje ukazatel na pravý a levý podstrom

Použití u vyhledávání, analyzátory zdrojových kódů.

12. Verifikace algoritmů

O algoritmu lze tvrdit, že je správný, když lze dokázat, že je správný vzhledem k zadání.

IAL - Dokazování správnosti algoritmu

Částečná správnost

Skončí-li algoritmus, pak je výsledek vždy správný. *To znamená, že algoritmus nemusí skončit pro všechny korektní vstupy v reálném čase.*

Úplná správnost

Je-li algoritmus částečně správný a skončí-li pro libovolné vstupní hodnoty, pak je úplně správný.

Průběžné dokazování ***(As-You-Go)***

Průběžné dokazování je založené na dekompozici problému na malé (snadno dokazatelné) celky a postupné syntéze algoritmů spojené s vytvářením prostředků pro vedení důkazu.

Automatizovaná verifikace

Automatizovaná verifikace se používá hlavně u paralelních programů, kde je nutné řešit synchronizační problémy. Existují dva základní přístupy k automatizované verifikaci - **theorem proving** a **model checking**.

Theorem proving

Theorem proving je obdoba dokazování programu člověkem. Nástroje si vedou přesnou evidenci použitých zákonů logiky, apod. Důkaz však musí vést člověk.

Model checking

Model checking je založený na systematickém generování a zkoumání všech programem dosažitelných stavů.

13. Algoritmy pro numerické výpočty

Hornerovo schéma

Hornerovo schéma představuje efektivní algoritmus pro výpočet hodnoty polynomu v bodě.

$3x^4 + 2x^3 - x^2 + x + 4 = (((3x+2)x - 1) x + 1) + 4 = P(2) = 66$

Metoda půlení intervalu

Funkce musí být spojitá

Funkční hodnoty v okrajových bodech zvoleného intervalu musí mít opačná znaménka

Výpočet určitého integrálu

Obdélníková metoda

Lichoběžníková metoda

Simpsonova metoda

Používá se konstantní šířka a sudý počet aproximačních intervalů. Tři sousední body křivky se vždy aproximují vhodnou parabolou. Metoda vykazuje velmi dobré výsledky (přesnost) již při velmi nízkém počtu intervalů.

14. Modulární výstavba programů

Modularita

Modularita představuje obecně vyšší stupeň strukturovanosti programu. Modularita usnadňuje dekompozici problému na jednodušší části - moduly.

Modul

Modul tvoří samostatnou jednotku, kterou lze samostatně kompilovat a opakovaně využívat. S moduly se pracuje pomocí jejich **rozhraní**, samotná implementace jednotlivých operací je skryta.

Rozhraní modulu

Rozhraní modulu definuje vývoz *(export)* a dovoz *(import)* modulu, víme...

```
#ifndef __MODUL__
#define __MODUL__
// obsah *.H souboru
#endif
```