Anotace

- Typové proměnné,
- možnosti inicializace proměnných,
- množiny,
- poznámky k předání parametru hodnotou a referencí,
- náhodná a pseudonáhodná čísla,
- algoritmy Monte Carlo a Las Vegas,
- Dijkstrův algoritmus.

Poznámka

- Open array parameters nám umožňovaly předávat pole předem neznámé délky,
- syntax: `function f(a:array of integer);...`
- ve druhé paralelce byly pod aliasem *konformní schémata polí.*

Typové proměnné

- Syntakticky děláme, jako bychom chtěli definovat konstantu, ale určíme jí typ:
- `const cislo:integer=7;`
- nejedná se o konstantu, obsah můžeme měnit:
- `cislo:=5;`
- K čemu lze použít "nekonstantní konstantu"?
- Například jako proměnnou, kterou rovnou máme zinicializovánu.

Další možnosti inicializace proměnných

- Na uvedené použití typových proměnných GPC ohlásí:
- `warning: assignment: typed const misused as initialized variable`
  `warning: (Better use ISO 10206 Extended Pascal initialized`
  `warning: types and variables: ‘var foo: integer value 7’.)`
  `warning: assignment: typed const misused as initialized variable`
- čili podle ISO 10206 můžete proměnné inicializovat pomocí klíčového slova `value`:
- `x:string value 'xxx';`
- Ovšem jen u překladačů, které ISO 10206 implementují.

Množiny

- Chceme-li organizovat více údajů, můžeme použít:
- pole (`array`),
  na mnoho dat téhož typu
  přes prvky pole je možno cyklit,
- strukturu (`record`),
  na stanovené množství souvisejících dat
  přes prvky struktury není možno udělat cyklus
- anebo množiny (`set`).
- Příklad: `var samohlasky:set of char;`
- `samohlasky:=['a','e','i','o','u'];`
- Množiny lze sjednocovat ($+$), pronikat ($*$), odečítat ($-$).
- Prvek v množině buďto je, nebo není. (Ale není vícekrát!)

Příklad

```
var samohlasky, souhlasky, znaky, cisla:set of char;
     c:char; sam,sou,cis:integer;
begin sam:=0;sou:=0;cis:=0;
      samohlasky:=['a','e','i','o','u','y'];
      souhlasky:=['a'..'z']-samohlasky;
      cisla:=['0'..'9'];
      znaky:=[];{prazdna mnozina}
      while not eoln do begin
            read(c);
            znaky:=znaky+[c];
            if c in samohlasky then inc(sam);
      end;
      writeln(sam,' samohlasek.');
end.
```

Poznámky k předání parametru

- Předání parametru hodnotou: `procedure f(i:char);` předávaná hodnota je okopírována.
- Předávání parametru referencí: `procedure f(var i:char);` okopírujeme pointer.
- Změníme-li hodnotu parametru při předání hodnotou, nic se po konci funkce nestane.
- Změníme-li hodnotu při předání referencí, je obsah předávané proměnné změněn (přepsán).
- Hodnotou lze předat jakýkoliv výraz,
- referencí lze předat jen proměnnou!
- Jak bere parametr například funkce `sin` a proč?
- Ano, hodnotou, protože jinak by nešlo napsat: $sin(2 * x + 15)$, ale $2 * x + 15$ bychom museli přiřadit do nějaké proměnné!

Náhodná a pseudonáhodná čísla

- Náhodná čísla se získávají fyzikálním způsobem (čas mezi stisky dvou kláves), ani to však není příliš náhodné,
- používá se pseudonáhoda, tedy výsledky algoritmu, ze kterého lezou čísla na první pohled náhodně vyhlížející (na druhý pohled už je to slabší).
- V Pascalu je (podle dostupných informací) lineární kongruenční generátor, ovšem podle testů kolegů poměrně dobře vyladěný.
- Pseudonáhodný generátor je deterministický algoritmus pracující s několika proměnnými, které určují, jaká "náhodná" čísla budou vygenerována.
- Všechny hodnoty řídící generování lze nastavit pomocí longintové proměnné `RandSeed`, tak je možno vynutit si předvidatelnost vygenerovaných pseudonáhodných hodnot.

- Jednotlivé náhodné hodnoty získáme voláním funkce `Random`.
- Funkce `Random` bez parametru generuje hodnoty typu `real` z intervalu $< 0, 1)$ a pokouší se tvářit náhodně nezávisle uniformně.
- Funkci `Random` můžeme předat jeden parametr typu `word`. V tom případě se generuje přirozené číslo (tedy včetně nuly) menší, než je zadaný parametr.
- Chceme-li zinicializovat generátor náhodně, zavoláme funkci `Randomize` (bez parametrů), která `RandSeed` zinicializuje hodnotou časovače, což je jednen z relativně náhodně se chovajících prvků.

Příklad

Nagenerování pseudonáhodných čísel

```
var a:array[1..1000] of real;
    b:array[1..1000] of integer; begin
      Randomize;        for i:=1 to 1000 do
      begin
            a[i]:=random;{Napln a nahodnymi cisly
<0,1)}
            b[i]:=random(2);{Pole b obsahuje na
pohled nahodna cisla 0 a 1}
      end;
end.
```

Pravděpodobnostní algoritmy

- Pokud je problém komplikovaný, můžeme si zkusit pomoci náhodou.
- Příklad: Quicksort – jako pivot nevybereme první prvek, ale prvek náhodný.
- Existují dva typické způsoby použití náhody zvané *Monte Carlo* a *Las Vegas*.

Algoritmy Monte Carlo

- Algoritmy této rodiny pomocí náhody přibližným způsobem zkoušejí zjistit řešení problému – zpravidla hodnotu nějaké veličiny a dost často geometrickým způsobem.
- Nějaký výsledek je tedy k dispozici téměř hned, aby měl ale nějaký smysl, je potřeba počkat, až pro nás začne hrát teorie pravděpodobnosti
- (zejména zákon velkých čísel, centrální limitní věta, všechny možné nerovnosti...).
- Monte Carlo algoritmy typicky nekončí, typicky se jejich delším během zvyšuje přesnost.

Monte Carlo

Příklad – výpočet plochy kruhu

```
const MAX=1000;
var x,y:real; uvnitr,i:integer;
begin uvnitr:=0;
      randomize;
      for i:=0 to MAX do
      begin
            x:=random;
            y:=random;
            if(x*x+y*y)<1 then
                  inc(uvnitr);
      end;
      writeln('Plocha je: ',4*uvnitr/MAX;
end.
```

Las Vegas

- Výpočet algoritmu by běžel i bez náhody,
- náhoda jen zkouší eliminovat špatná rozhodnutí.
- Algoritmus vždy odpoví správně.
- Příklad: Quicksort, kde za pivot bereme náhodně vybraný prvek.

Las Vegas příklad

faktorizace

- Načti číslo n.
- `prvoc:=true;`
- `for i:=n-2 downto 1 do begin`
  - `x:=random(i);`
  - do j prirad x-te dosud netestovane cislo (z $\{1..n\text{-}1\}$);
  - `if n=(j*(n div j)) then begin`
    - `writeln(n,'=',j,'*',n div j);`
    - `prvoc:=false;`
    - `break;{Ukonci cyklus}`
- `if prvoc then writeln(n 'je prvocislo');`

Klíčová slova `break` a `continue`

- `break` ukončí provádění nejbližšího cyklu (a pokračuje se až za ním),
- `continue` ukončí současnou iteraci cyklu (a pokračuje se znovu vyhodnocením cyklící podmínky).
- Tato dvě klíčová slova nepoužívejte bez vážného důvodu, narušují strukturované programování, je potřeba na ně dávat obzvláštní pozor, ve zdrojovém kódu snadno zaniknou a člověk se diví, proč konkrétní iterace cyklu nedojela až do konce, ale zadrhla se někde uprostřed.

Dijkstrův algoritmus

Hledá nejkratší cestu z daného vrcholu (do všech ostatních)

Vstup: Graf s `nezáporně` ohodnocenými hranami.

- Udržujeme "frontu" vrcholů setříděnou podle dosud nejkratší cesty do nich.
- Na začátku zinicializujeme vzdálenosti do všech vrcholů kromě startovního nekonečnem (tedy dost vysokou hodnotou) a vzdálenost do startu nulou.
- Startovní vrchol přidáme do "fronty" dosažitelných vrcholů.
- Vrchol, do kterého se dostaneme nejkratší cestou z fronty odstraníme a pokusíme se cestu jdoucí z něj rozšířit do jeho sousedů.
- Toto opakuj, dokud je "fronta" neprázdná.

Rozšíření cesty

Rozšíření cesty vypadá tak, že pro vrchol v ve vzdálenosti $d(v)$ zkusíme pro každou hranu $\{v, w\}$, zda

$$d(w) > d(v) + \mathit{delka}(\{v, w\}).$$

Pokud ano, $d(w) := d(v) + \mathit{delka}(\{v, w\})$ a oprav pozici vrcholu w ve "frontě".

Analýza

- Algoritmus je popsán a dokázán v mnoha knihách (a skriptech), kupř. Kapitoly z diskrétní matematiky nebo Algebraické algoritmy (Kučera, Nešetřil)...
- Konečnost: V každé iteraci odstraníme z "fronty" jeden vrchol, který se v ní už neobjeví
  (protože mezi vrcholy ve frontě měl nejmenší vzdálenost od startu a vzdálenosti jsou nezáporné).
- Parciální správnosti pomůže invariant:
  V každém kroku evidujeme nejkratší cesty ze startu používající pouze vrcholy již odstraněné z "fronty".
- Z invariantu plyne korektnost.
- Jde jen o modifikovaný algoritmus vlny, tedy hledání do šířky!
- Složitost významně závisí na reprezentaci grafu a na reprezentaci "fronty"!