REKURZE

Sebeopakování

Rekurentní výpočty

-Na základě znalosti prvního prvku a pravidla pro výpočet následujícího prvku spočítá druhý prvek

- **zobecnění – z každého prvku je možné pomocí téhož pravidla získat prvek následující**
- např. v matematice – aritmetické, geometrické posloupnosti, řady
- **riziko** – u geometrické posloupnosti, která není konvergentní – nabývá rychle velkých hodnot – může snadno přesáhnout rozsah povolený datovým typem – dojde k **„přetečení“**

Rekurze v programování

- **Opakované vnořované volání stejné funkce (podprogramu)**

1. Na začátku **musí být** podmínka určující, kdy se má vnořování zastavit = **ukončovací podmínka !!!!**

   chybná formulace ukončující podmínky vede k zacyklení do nekonečného cyklu a ke zhroucení programu (přetečení zásobníku)
2. Programovací jazyk musí umožnit volat podprogram ještě před ukončením předchozího volání
3. Po každém kroku volání sebe sama musí dojít ke zjednodušení problému, aby rekurze spěla ke konci

Co se děje při rekurzi?

Pohádka o slepičce a kohoutkovi

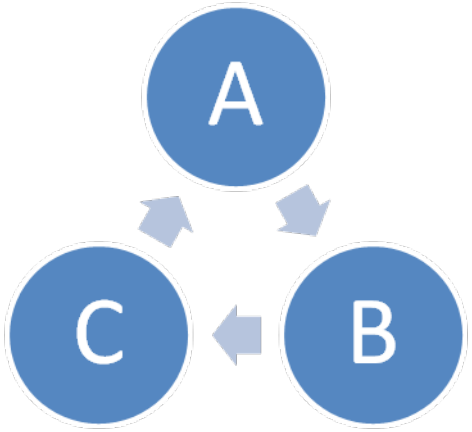
Volání funkce **DONES** (studánka, voda)
Volání funkce **DONES** (švadlenka, šátek)
Volání funkce **DONES** (švec, střevíčky)
Volání funkce **DONES** (louka, travička pro kravičku)
Volání funkce **DONES** (nebe, rosička)
Nebe se slitovalo – už k žádnému volání funkce nedojde a končí cyklus

Funkce DONES vždy zavolá úplně stejnou funkci DONES (parametr1, parametr2), pokaždé s jinými parametry

Zpětný chod rekurzní funkce –odspodu se zavírají všechna volání a posbírají se výsledky – dosazují se do nedořešených výrazů – (kohoutek dostane vodu)

Základní rozdělení, využití

- Přímá rekurze
  - Pokud podprogram volá přímo sebe sama
- Nepřímá rekurze
  - Vzájemné volání podprogramů vytvoří kruh
  - (v příkazové část funkce A je volána funkce B, ve funkci B voláme funkci C , která volá funkci A



EFEKTIVITA REKURZE

Využití

- Řešení obecných úloh rozkladem na dílčí úlohy, které se řeší stejným způsobem

VÝHODY: Jednoduchost a přehlednost algoritmu

NEVÝHODY:

- Při velkém počtu vnoření obsadí poměrně velké množství paměti
  - Při každém vyvolání podprogramu se uloží do zásobníku paměti:
    - lokální proměnné
    - předávané parametry
    - návratové adresy
  - Po ukončení rekurze: díky uloženým návratovým adresám procházíme zpět stejnou cestou a uložené informace si znovu vyzvedáváme
- Z toho plyne: při větším počtu vnoření je to velmi pomalá varianta řešení problému

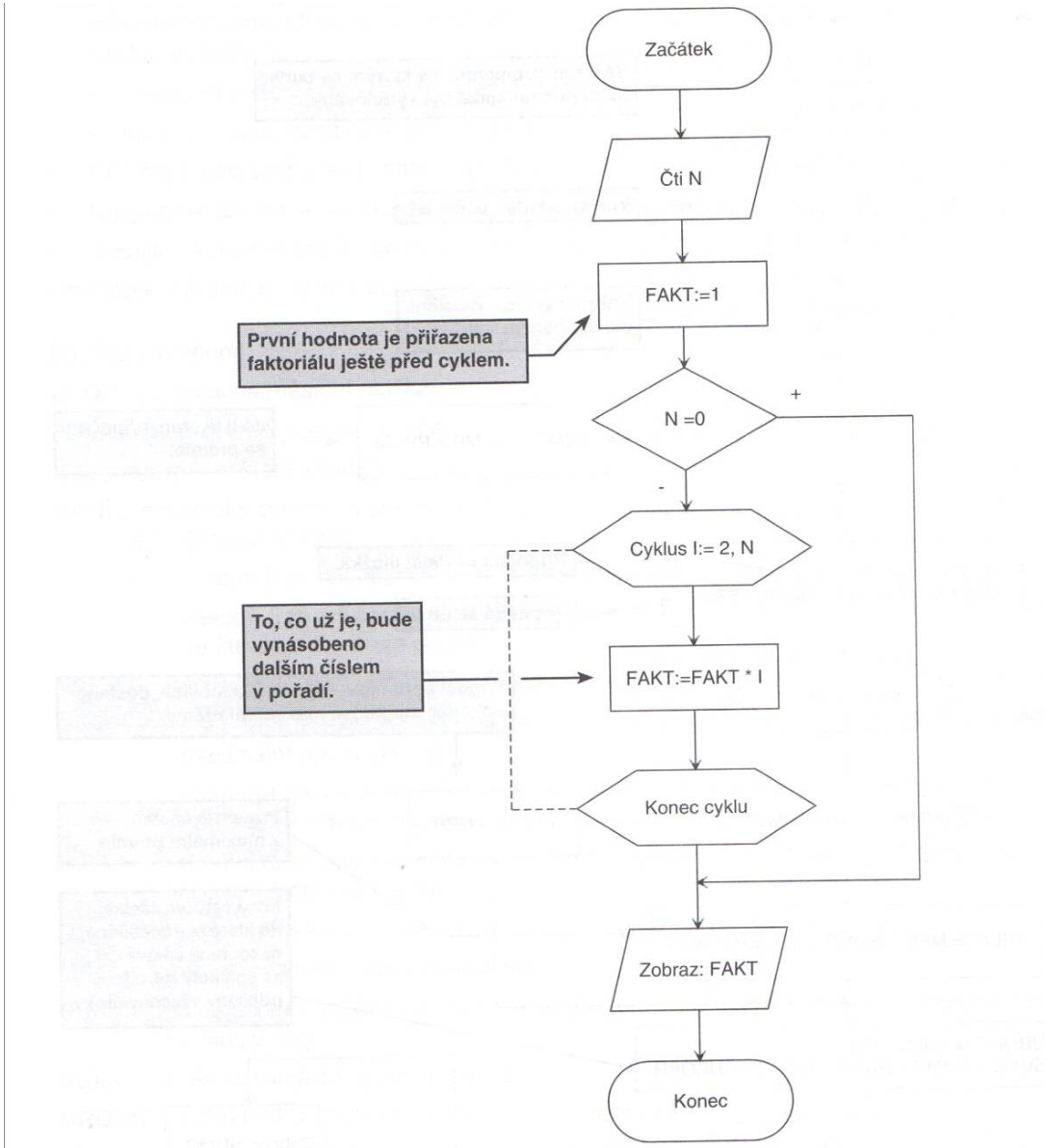
Příklady rekurze

1. Výpočet faktoriálu čísla n
2. Aritmetická, geometrická posloupnost
3. Fibonacciho posloupnost
4. Třídící algoritmus Quicksort

1. Faktorial

- Pro celá kladná: $n! = n \cdot (n-1) \cdot (n-2) \cdot (n-3) \ldots 1$
- $n! = n \cdot (n-1)!$
- Pro $n=0$ je $0!=1$
- Při každém průchodu funkcí snižujeme n o 1 a testujeme, zda již $n = 0$ ➔konec

Výpočet faktoriálu bez rekurze



Faktoriál

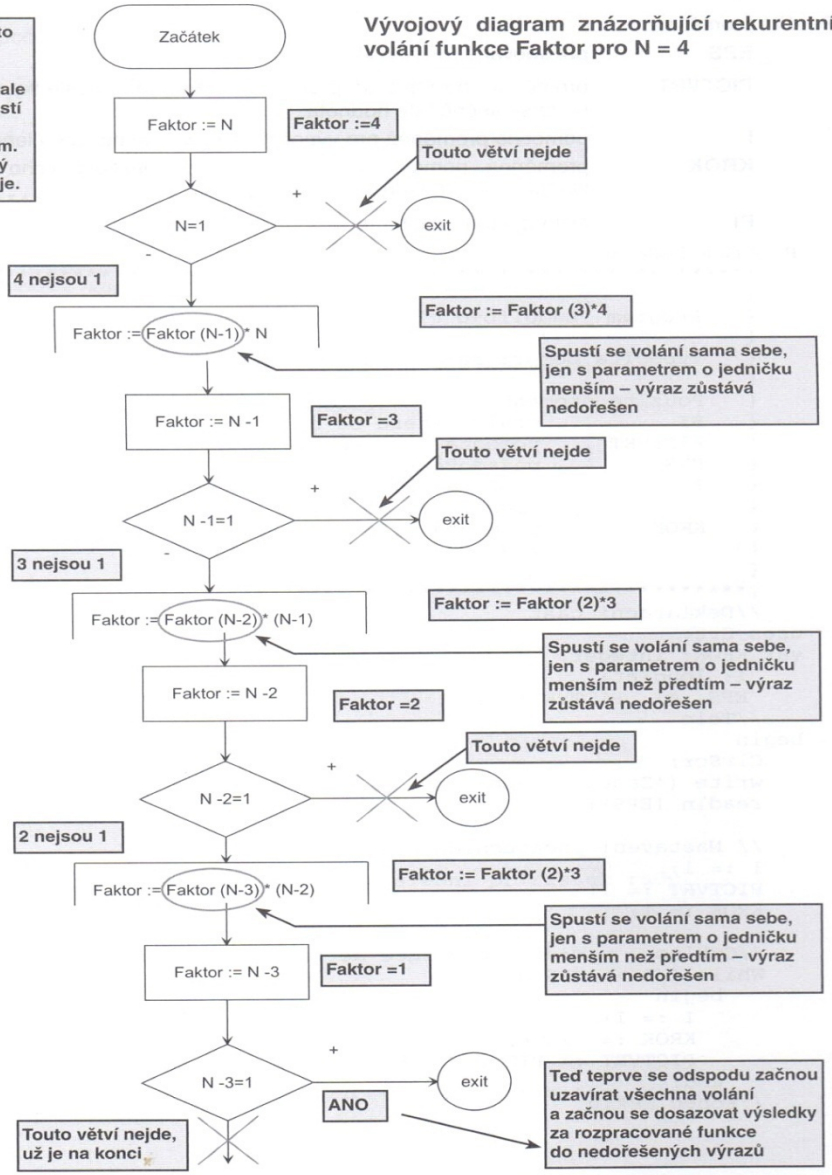
Nyní bude stejný postup aplikován na funkci pro výpočet faktoriálu:

```
function Faktor (N: integer): longint;
    var I: integer;
    begin
       Faktor := N;
       if N=1 then exit;       //Vyskoci z podprogramu do hlavniho programu
       N := N-1;
       Faktor := Faktor(N-1) * N;       //Rekurze (vola samu sebe)
    end;
```

Až bude N=1, na další volání funkce už nedojde a vše se může po posledním volání začít uzavírat.

Pokud nebude N=1, tak se tento příkaz v programu nemůže vyřešit hned, jak na něho přijde řada, ale zůstane „rozpracovaný" – spustí nové volání téže funkce s parametrem o jedničku menším. To však má ve svém těle stejný příkaz, takže se situace opakuje.

Vývojový diagram znázorňující rekurentní volání funkce Faktor pro N = 4



Zdrojový kód - faktorial

```
function faktorial(n: integer):
    integer;
begin
  if n = 0 then
    faktorial:= 1
  else
    faktorial:= n*faktorial(n-1);
end;
```

```
var
  n: integer;
  c: char;
  faktorial: longint;
begin
  repeat until (c = 'y') or (c = 'Y')
    ;
    Writeln('Napis cislo.');
    Readln(n);
    if n<0  then
    begin
       write ('Cislo je zaporné. ');
      exit;
    End;
    Writeln(n, '! = ', faktorial(n));
    Writeln('Skoncit?(Y/N)');
    Readln(c);
  end.
```

2. Fibonacciho posloupnost

Posloupnost f(0)=0, f(1)=1,

f(2)=f(0)+f(1)=1

f(3)=f(1)+f(2)=2 ....

f(n)=f(n-2)+f(n-1)

Příklad: 0,1,1,2,3,5,8,13,21,34,55,....

- Rekurze – exponenciální časová složitost, počítáme s prvky, které jsme již počítali
- Bez rekurze – prvky posloupnosti počítat od začátku a ukládat do nového pole

Zdrojový kód Fibonacciho posloupnost

```
function fib(n: integer):
    integer;
begin
  if n <= 2 then
    fib:= 1
  else
    fib:= fib(n-1) + fib(n-2);
end;
```

```
var
  n: integer;
  c: char;
begin
  repeat
    Writeln('Napis cislo.');
    Readln(n);
    Writeln(n, '-te fibonacciho
    cislo je ', fib(n));

    Writeln('Skoncit?(Y/N)');
    Readln(c);
  until (c = 'y') or (c = 'Y') ;
end.
```

3. Největší společný dělitel

Vytvořte rekurzívní funkci pro výpočet největšího společného dělitele

```
Function NSD(a,b:integer):integer;
Begin
  if a=b then
    NSD:=a
  else
    if a>b then
      NSD:=NSD(a-b,b)
    else
      NSD:=NSD(a,b-a);
End;
```