1 Rekurze

Iterace v programovacích jazycích je opakování procesu uvnitř programu. Iterace se provádí pomocí řídících struktur (for, while, …).

Rekurze je algoritmus, který v průběhu svého běhu volá sám sebe.

Rekurzivní funkce ve svém těle volá sama sebe. Rekurze je vedle iterace druhým přístupem k provádění opakování v programu.

Výhody rekurze:

- program je přehlednější a kratší
- některé algoritmy nelze bez rekurze vyřešit (nebo jde, ale jsou časově i paměťově náročnější)

Nevýhody rekurze:

- může vzniknout velká časová náročnost způsobená opakováním výpočtu
- každé nové rekurzivní zavolání funkce má vlastní lokání proměnné (=> větší paměťová náročnost)
- vytvořit rekurzi je někdy dost náročné

Většinou lze převést iterační algoritmus na rekurzivní a obráceně.

1.1 Dělení rekurzí

- Přímá – funkce volá sama sebe
- Nepřímá – funkce A volá funkci B, ta volá funkci C a ta opět volá funkci A

- Lineární – v těle funkce dochází pouze k jednomu rekurzivnímu volání
- Stromová – v těle funkce dochází k více volání (např. v těle funkce je cyklus, ve kterém se v každém jeho průběhu zavolá tato funkce znova)

1.2 Příklady (využití) rekurze

1.2.1 Faktoriál

Faktoriál čísla ***x*** je součin všech kladných celých čísel, které jsou menší nebo rovny číslu **x**.

Př: 5! = 5 * 4 * 3 * 2 * 1 = 120

Program by šel napsat pomocí iterace (cyklus for), ale lze také zapsat pomocí rekurze:

```
def faktorial(x):
    if(x == 0):
        return 1
    return faktorial(x-1) * x
```

1.2.2 Fibonacciho posloupnost

Pomocí rekurze lze spočítat n-tý člen Fibonacciho posloupnosti

```
def fibonacci(n):
    if(n < 2):
        if(n < 1):
            return 0
        return 1
    return fibonacci(n-1) + fibonacci(n-2)
```

1.2.3 Výpis souborového systému

Rekurze se používá například při výpisu obsahu adresářů na disku.

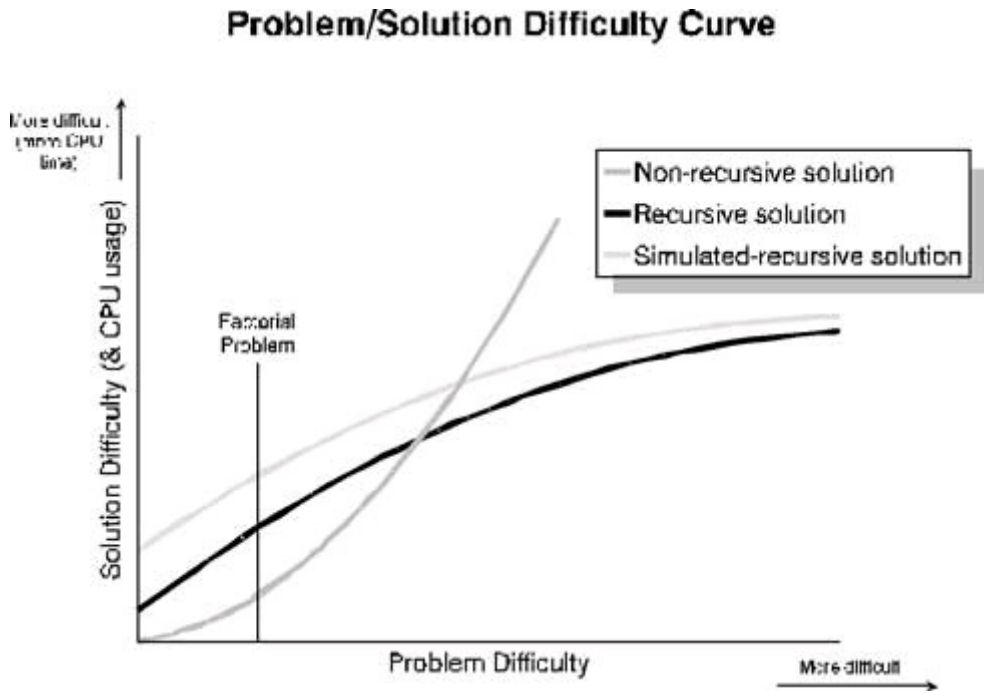
Zadání: Chceme na obrazovku vypsat všechny soubory a adresáře z aktuálního umístění a chceme data vypsat „stromově“.

Řešení: Vytvoříme funkci, která bude mít jeden parametr (cesta/název adresáře). Tato funkce projde všechny „objekty“ v tomto adresáři, pokud se jedná o soubor, tak ho vypíše na obrazovku, pokud se jedná o adresář, zavolá se tato funkce znova... a znova... a znova :)

1.3 Závěr

Příklady na faktoriál a Fibonacciho posloupnost jsou při použití rekurze časově i paměťově *náročnější* než kdyby se použila iterace (cyklus). Proč tedy vlastně používat rekurzi?

Př: Budeme mít dvě posloupnosti A a B a musíme najít jejich společnou část (třeba do tisícího členu budou čísla obou posloupností stejná a tisícíprvé číslo se pak bude lišit :-)). Řešení pomocí cyklů by bylo náročnější než pomocí rekurzí, viz graf:



Důvod použití rekurze může být také ten, že je důležitější čas než paměťová náročnost.

2 Třídící algoritmy

Účel: Seřazení dat podle požadovaného kritéria (podle velikosti čísel, podle abecedy)

Třídící algoritmy se dělí podle různých kritérií:

- vnitřní - všechna řazená data jsou v operační paměti
- vnější - všechna data se do operační paměti nevejdou a část z nich musí být ve vnější paměti (pevný disk)

Algoritmy se můžou dělit podle toho, jak se chovají k již částečně seřazeným datům:

- přirozené
- nepřirozené

Přirozené třídící algoritmy vykonají práci rychleji (nepřirozený algoritmus se sice snaží data setřídit, ale ve skutečnosti částečně setřízená data „omylem" rozhází a pak je opět seřazuje).

Dělení podle principu řazení:

- princip *výběru* - přesouvají minimum a maximum do výstupní posloupnosti
- princip *vkládání* - vkládají postupně prvky do již seřazené výstupní posloupnosti
- princip *rozdělování* - rozdělují postupně prvky na dvě podmnožiny tak, že prvky v první podmnožině jsou menší než prvky v druhé podmnožině
- princip *slučování* – sloučení více již seřazených podmnožin do jedné jedné velké množiny

Efektivita řadící metody (*časová náročnost*) – Zjišťuje se z počtu porovnání klíčů a z počtu přesunů prvků nutných pro seřazení **n** prvků.

Stabilita řazení je vlastnost algoritmu, který zachová relativní pořadí položek se stejnou hodnotou klíče. Metody můžeme tedy dělit na:

- stabilní
- nestabilní

(Pokud budu mít v datech tři stejná čísla a tyto data pak setřídím pomocí *stabilní* řadící metody, na výstupu budou tyto čísla ve stejném pořadí jako na vstupu).

2.1 Select sort

Jádrem algoritmu je cyklus pro vyhledání pozice extrémního prvku (min. nebo max.) v zadaném segmentu pole. Nalezený prvek se vždy vymění s prvkem pole na odpovídajícím indexu.

- Řazení na principu výběru
- Metoda je nestabilní, přirozená a má kvadratickou časovou složitost (proto se hodí pro řazení menšího počtu prvků)

```
def selection(pole):
    for i in range(len(pole)):
        #Najit nejmensi prvek z casti neserazeneho pole
        imin = i
        for j in range(i+1, len(pole)):
            if(pole[j] < pole[imin]):
                imin = j

        #A tento prvek zamenit za prvni prvek neserazene casti pole
        if(imin != i):
            pomocna = pole[imin]
            pole[imin] = pole[i]
            pole[i] = pomocna
    return pole
```

2.2 Buble sort

Algoritmus opakovaně projíždí pole a porovnává dva sousední prvky – pokud nejsou ve správném pořadí, tak je prohodí. Tak tedy nejmenší a největší čísla „probublají“ na okraje.

Ripple-sort - přeskakuje dvojice, u nichž je jasné, že se nebudou vyměňovat (v příkladu)

Shaker-sort - střídá směr probublávání (houpačková metoda, zleva doprava a pak zprava doleva)

- Řazení na principu výběru
- Metoda stabilní, přirozená a má kvadratickou časovou náročnost - opět se nehodí pro větší počet prvků; ale v případě, že máme částečně seřazenou posloupnost je tato metoda jedna z nejrychlejších! (jedna z nejvíc přirozených metod)

```
def bubble(pole):
    delkaPole = len(pole)
    for i in range(delkaPole):
        vymena = False
        for j in range(delkaPole - i):
            if(j != delkaPole - 1 and pole[j] > pole[j+1]):
                pomocna = pole[j]
                pole[j] = pole[j+1]
                pole[j+1] = pomocna
                vymena = True

        if(not vymena):
            break
    return pole
```

2.3 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.

- Řazení na principu vkládání
- Metoda stabilní, časová náročnost je kvadratická

```
def insertsort(pole):
    for i in range(1, len(pole)):
        hodnota_i = pole[i]
        j = i
        while j > 0 and pole[j - 1] > hodnota_i:
            pole[j] = pole[j - 1]
            j = j - 1
        pole[j] = hodnota_i
    return pole
```

2.4 Quick sort

1. Zvolíme tzv. *pivot*, což je jeden z řazených prvků
2. Přerovnáme prvky tak, že na začátku řazených dat budou všechny prvky, které jsou menší nebo stejné jako pivot, a v pravé části jsou všechny prvky, které jsou větší nebo stejné jako pivot. Prvky mající stejnou velikost jako pivot tedy mohou ležet v levé i v pravé části řazených dat. Po této operaci je již pivot na správném místě a dál se s ním

nehýbe.

3. Rekurzivně stejnou technikou seřadíme levou a pravou část řazených dat

Implementace:

- levý index se nastaví na začátek zpracovaného úseku
- pravý index se nastaví na konec zpracovaného úseku
- zvolí se pivot (nejlépe střed: (levý index + pravý index)/2)
- cyklus (rozdělení na "malé a velké")
    - levý index se posunuje doprava a zastaví se na prvku větším nebo rovném pivotovi
    - pravý index se posunuje doleva a zastaví se na prvku menším nebo rovném pivotovi
    - pokud je levý index ještě před pravým (iL < iR), příslušné prvky se prohodí a oba indexy se posunou o 1 ve svém směru (iL++, iR++)
    - celý cyklus se opakuje dokud se indexy nepřekříží (iL <= iR), tj. pravý se dostane před levého
- Následuje rekurzivní volání (zpracování "malých" a "velký" zvlášť)
- na úseku od začátku do pravého(!) indexu včetně a na úsek od levého(!) indexu včetně až do konce, má-li přislušný úsek délku větší než 1

Vlastnosti: Nestabilní, nepřirozený, časová náročnost: linearitmická ($N * \log_2 N$)

```
def quick(pole):
    if(len(pole) <= 1):
        return pole
    levaCast = []
    pravaCast = []

    pivot = 0
    for j in pole:
        pivot += j
    pivot /= len(pole)

    for i in pole:
        if(i <= pivot):
            levaCast.append(i)
        else:
            pravaCast.append(i)
    return quick(levaCast)+quick(pravaCast)
```

Pozn: U Quick Sortu je největší problém určit pivota, v ideálním případě by to měl být medián, ale spočítat medián je docela problém (časová náročnost). Pivot se tedy může určit úplně

náhodně a nebo to může být polovina pole.

2.5 Shell sort

Shell sort (řazení se snižujícím se přírustkem) je metoda založená na principu insert sortu. Rozdíl je ten, že se určuje mezera mezi prvky, které se mají spolu porovnávat. Tato mezera se postupně snižuje až dojde k nule a v posledním kroku se seřadí všechna čísla.

Teoretické analýzy nenašly nejvhodnější řadu snižujících se kroků. Kerninghan a Ritchie ve své implementaci používali krok N/2, který pak postupně dělili dvěma.

Shell sort je nestabilní řadící metoda. Její experimentálně naměřená složitost odpovídá linearitmickým řádícím metodám.

```
def shellSort(pole):
    mezera = len(pole) / 2
    while mezera > 0:
        # Zde se provadi insert sort
        for i in range(mezera, len(pole)):
            hodnota_i = pole[i]
            j = i
            while j >= mezera and pole[j - mezera] > hodnota_i:
                pole[j] = pole[j - mezera]
                j -= mezera
            pole[j] = hodnota_i
        mezera /= 2
    return pole
```

3 Efektivita algoritmu

Obecně požadujeme, aby algoritmus byl efektivní, v tom smyslu, že požadujeme, aby každá operace požadovaná algoritmem, byla dostatečně jednoduchá na to, aby mohla být alespoň v principu provedena v konečném čase (tj. byla elementární).

V praxi jsou proto předmětem zájmu hlavně takové algoritmy, které jsou v nějakém smyslu kvalitní. Takové algoritmy splňují různá kritéria, měřená např. počtem kroků potřebných pro běh algoritmu, nebo jednoduchost či elegance algoritmu. Problematikou efektivity algoritmů, tzn. metodami, jak z několika známých algoritmů řešících konkrétní problém vybrat ten nejlepší, se zabývají odvětví informatiky nazývané algoritmická analýza a teorie složitosti. Výpočetní proces je posloupnost akcí nad daty uloženými v paměti počítače.

Pojmy, které se používají s termínem efektivita algoritmu:

- složitost
- nejlepší, průměrný a nejhorší případ
- konkrétní odhady složitosti jednoduchých algoritmů
- složitost problému

U řadících algoritmů se efektivitou algoritmu rozumí časová a paměťová náročnost.

4 Datový typ ukazatel

Statické přidělování paměti

Staticky se paměť přiděluje deklarovaným datovým strukturám v době překladu. Paměťové úseky jsou pak přístupné pomocí jména proměnné, uvedeného v deklaraci.

Dynamické přidělování paměti

Dynamicky přiděluje paměť systém na základě požadavku vzniklého v době řešení programu. K paměťovému úseku se pak přistupuje nepřímo, prostřednictvím ukazatele.

Ukazatel reprezentuje adresu objektu. Nese sebou současně informaci o datovém typu, který se na této adrese nachází.

Rozlišuje se:

- práce se hodnotou adresy ukazatele
- práce s hodnotou datového prvku ukazatele

Jinak řečeno: Buď ukazatel ukazuje na buňku v paměti, ve které je adresa další buňky, a nebo je tam rovnou hodnota, se kterou budeme pracovat. (TÍM SI NEJSEM TAM JISTÝ?)

U některých programovacích jazyků je datový typ ukazatel nahrazen referencí na objekt (Python, Java, PHP).

5 Binární strom

Definice: Strom můžeme definovat buď jako prázdný strom, nebo jako vrchol, k němuž jsou připojeny dvě další stromové struktury - podstromy.

Jinak řečeno: každý strom je tvořen dalšími stromy, pro které platí stejná definice. Je to taková rekurze :-)

Nejvyšší vrchol stromu se nazývá **kořen** stromu, prvky, které leží v nejnižší části stromu a nemají žádné následovníky, se nazývají **listy** stromu. Ostatní prvky jsou souhrnně nazývány **vnitřní vrcholy**.