Správnosť algoritmu

Algoritmus je vzhľadom na vstupné a výstupné podmienky správny, ak:
1. pre všetky vstupné údaje spĺňajúce vstupnú podmienku sa proces predpísaný algoritmom zastaví a
2. výstupné údaje spĺňajú výstupnú podmienku.

Algoritmus je **čiastočne správny**, ak pre vstupné údaje, ak skončí, dáva správne výsledky.
Poznámka: Čiastočne správny je aj algoritmus, ktorého výpočet neskončí pre žiadne vstupné údaje.

Algoritmus je **správny**, ak je
1. čiastočne správny a
2. konečný, t.j. jeho výpočet pre všetky vstupné údaje skončí.

Poznámky:
Dokázať konečnosť algoritmu znamená dokázať najmä konečnosť v ňom použitých cyklov. Dokázať konečnosť cyklu znamená dokázať, že hodnoty premennej riadiacej počet prechodov cyklom tvoria klesajúcu, zdola ohraničenú postupnosť alebo rastúcu, zhora ohraničenú postupnosť.
Často je jednoduchšie dokázať, že algoritmus nie je správny. K dôkazu „nesprávnosti" algoritmu stačí nájsť aspoň jeden konkrétny príklad vstupných údajov, pre ktoré algoritmus zlyhá.

Výpočtová zložitosť algoritmu a programu

Rozsah problému je prirodzené číslo N, ktoré vyjadruje veľkosť vstupných údajov.
Napr. počet čísel v poli, počet cifier v čísle a pod.

Časová výpočtová zložitosť $\mathcal{T}$(N) je funkcia, ktorá popisuje závislosť medzi rozsahom problému a potrebným počtom krokov na jeho vyriešenie.
Kroky napr.: počet operácií abstraktného procesora, počet „významných" operácií (aritmetické, logické, porovnania, presuny,...).

Pamäťová výpočtová zložitosť $\mathcal{S}$(N) je funkcia, ktorá popisuje závislosť medzi rozsahom problému a veľkosťou pamäte potrebnej na jeho vyriešenie.
Napr. počet bitov potrebných na uloženie všetkých údajov.

Optimálny algoritmus je taký, ktorý
- rieši daný problém a
- lepšie (efektívnejšie) sa už vyriešiť nedá.

Najhorší prípad pri danom N je konkrétna skupina údajov, ktorá vedie podľa voľby vstupných údajov k najhoršiemu prípadu (výpočet trvá najdlhšie).
Napr. pri bubblesorte je to prípad, keď sú vstupné údaje utriedené zostupne.

Priemerný prípad (štatistický pojem) je očakávaná zložitosť pri náhodne zvolenej skupine vstupných údajov rozsahu N.

Asymptotická časová výpočtová zložitosť $O(f)$, zjednodušene povedané, vyjadruje, ako rýchlo rastie časová výpočtová zložitosť s rastúcou hodnotou N (teoreticky s $N \to \infty$). $O(f)$ je asymptotickým vyjadrením funkcie $\mathcal{T}$ (N). V asymptotickej časovej zložitosti vynechávame multiplikačné konštanty a členy, ktoré sú menšie ako člen s najväčšou hodnotou.

Napr. pre bubblesort s cyklami `for cp in range(1,n): for i in range(0,n-cp):...` (kde n = len(pole))

$\mathcal{T}(N) = (n - 1) * \frac{n}{2} = \frac{1}{2} n^2 - \frac{1}{2} n$, čo zapisujeme $O(n^2)$.

Asymptotická časová výpočtová zložitosť môže byť:

- **polynomiálna** $O(n^{exp})$, napr.:
  - **konštantná** $O(1)$, napr. výpočet podľa vzorca, nájsť maximum v utriedenom poli
  - **lineárna** $O(n)$, napr. hľadanie prvku s požadovanou vlastnosťou v neutriedenom poli (potrebných rádovo n porovnaní, kde n je počet prvkov poľa)
  - **kvadratická** $O(n^2)$, napr. triedenia bubblesort, insertsort, selectsort,...
  - **kubická** $O(n^3)$, napr. súčin dvoch matíc (tri vnorené cykly)
- **logaritmická**
  - $O(\log_2 n)$, napr. hľadanie prvku s požadovanou vlastnosťou v utriedenom poli (potrebných rádovo $\log_2 n$ porovnaní, kde n je počet prvkov poľa)

    Počet delení (porovnaní): n, $\frac{n}{2}$, $\frac{n}{4}$,..., v najhoršom prípade $\frac{n}{2^p} = 1 \Rightarrow p = \log_2 n$
  - $O(\log n)$, napr. pri operáciách s ciframi v desiatkovom čísle
- **exponenciálna** $O(a^n)$ – napr. rekurzívny výpočet Fibonacciho čísla, prehľadávanie stromov do hĺbky - prakticky nerealizovateľná na počítači pre veľké n
- $O(n!)$, napr. zo všetkých možností usporiadania n prvkov vybrať tie, pri ktorých sú prvky usporiadané vzostupne
- $O(n * \log_2 n)$, napr. triediaci algoritmus Quicksort
- $O(\sqrt{n})$,…

Efektívnosť algoritmu a programu

Z dvoch algoritmov riešiacich tú istú algoritmickú úlohu je **časovo** efektívnejší ten, ktorého realizácia vyžaduje menej krokov (trvá kratšie). Efektívnejší je ten, ktorý je rýchlejší pri väčšom rozsahu problému (pri väčšom N). **Pamäťovo** efektívnejší je ten, ktorý potrebuje menej pamäťového miesta (menej premenných).
Úpravu algoritmu na efektívnejší tvar nazývame **optimalizáciou** algoritmu.

Zásady optimalizácie:

- výber najefektívnejšej metódy riešenia pre najpravdepodobnejší rozsah problému
- nerobiť v cykle opakovane to, čo sa dá spraviť jedenkrát pred ním
- optimalizovať predovšetkým telá cyklov
- využívať predchádzajúce výsledky (volania procedúr)
- zjednodušovať výrazy vybraním pred zátvorku

Napríklad

- výpočet hodnoty polynómu n-tého stupňa (sledujeme počet násobení):
  - umocňovaním $a_n x^n + a_{n-1}x^{n-1} +...+a_1 x + a_0$: $\mathcal{T}(n) = n + (n-1) + ... + 1 = \frac{n}{2}(n+1) \approx n^2$ – kvadratická zložitosť
  - Hornerovou schémou $(...((a_n x + a_{n-1})*x +a_{n-2})*x +...)*x + a_0$: $\mathcal{T}(n) = n$ – lineárna zložitosť
- výpočet NSD (A, B)
  - odčítaním napr. NSD (5 000, 5) =NSD(4 995, 5)=NSD(4 990, 5)=...spolu 999 odčítaní ...= 5
  - funkciou mod napr. NSD (5 000, 5) = NSD (0, 5) = 5 (jedno delenie)
- použitie statického poľa je časovo efektívnejšie ako dynamického
- použitie parametrov nahradzovaných odkazom je pre štruktúrované údajové typy pamäťovo efektívnejšie, ako použitie parametrov nahradzovaných hodnotou

Časová výpočtová zložitosť lineárneho a binárneho spôsobu vyhľadávania

Asymptotická časová výpočtová zložitosť funkcií lineárneho vyhľadávania uvedených v študijnom texte Lineárne vyhľadávanie je lineárna, t.j. O(n), kde n je počet prvkov poľa (pri priemernom prípade potrebujeme spracovať - „pozrieť“ ½n prvkov, pričom ½ je multiplikačná konštanta, ktorá sa pri asymptotickej časovej zložitosti zanedbáva; v najhoršom prípade potrebujeme pozrieť všetkých n prvkov, čo znova vedie k O(n)).
Asymptotickú časovú výpočtovú zložitosť binárneho vyhľadávania môžeme odhadnúť nasledujúcou úvahou:
Ak by sme sa zahrali hru na uhádnutie mysleného celého čísla napr. od 1 po 1000 a protihráč bude na náš tip oznamovať: „Veľa, uber!“, „Málo, pridaj!“, „Uhádol si!“, musíme myslené číslo uhádnuť, v najhoršom prípade, na

desiaty pokus. Podmienkou však je, že musíme za tip vždy vybrať číslo v strede skúmaného intervalu. Takže naše tipy by mali byť: 500, na odpoveď Veľa, uber! 250, na odpoveď Málo, pridaj! 750 atď. (skúste si nakresliť niekoľko úrovní binárneho stromu, ktorý vznikne). **k** pokusov nám teda umožňuje, pri dôslednom delení skúmaného intervalu na polovice, vybrať správne číslo z 2^k čísel. Napríklad najviac desať pokusov nám umožňuje uhádnuť ľubovoľné celé číslo z intervalu 1 až 1024 (= 2^{10}). Ak **n** je počet čísel a **k** počet pokusov, zrejme platí $2^k = n$. Zaujíma nás počet potrebných pokusov – porovnaní, t.j. **k** v rovnici $2^k = n$. Zlogaritmovaním so základom 2 dostávame: $\log_2 2^k = \log_2 n$ a po ďalšej matematickej úprave: $k = \log_2 n$. Takže asymptotická časová zložitosť binárneho vyhľadávania je $O(\log_2 n)$ – logaritmická.
Úvaha na odvodenie asymptotickej časovej zložitosti binárneho vyhľadávania by mohla byť aj nasledujúca:
Najprv máme preskúmať celú množinu čísel, t.j. n, po prvom porovnaní už len polovicu z celej množiny (n/2), po druhom štvrtinu (n/4), po treťom porovnaní osminu čísel (n/8) atď. V najhoršom prípade po k preskúmaní – porovnaní zostáva preskúmať jeden prvok. Preto platí $n/2^k = 1$ resp. $2^k = n$, z čoho, po vyššie uvedených úpravách, dostávame logaritmickú časovú zložitosť.

Diskutujte o výpočtovej zložitosti jednotlivých funkcií na výpočet Fibonacciho postupnosti:

```
po_index = int(input("Fibonacciho čísla po index: "))

# ITERÁCIA
# najefektívnejší výpočet
def fib_iteracne_for(po_index):
    prva_hodnota, druha_hodnota = 0,1         # použité len dve premenné!
    for inx in range(po_index):
        print(prva_hodnota, end=", ")
        prva_hodnota, druha_hodnota  = druha_hodnota, prva_hodnota + druha_hodnota
    print(prva_hodnota)

print("\nNajefektívnejší výpočet pomocou iterácie:")
fib_iteracne_for(po_index)

# použitie while
def fib_iteracne_while(po_index):
    inx = 0
    prva_hodnota = 0
    druha_hodnota = 1
    while inx <= po_index:
        if inx <= 1:
            nasledujuca = inx
        else:
            nasledujuca = prva_hodnota + druha_hodnota
            prva_hodnota = druha_hodnota
            druha_hodnota = nasledujuca
        print(nasledujuca, end=", ")
        inx += 1
''' odstránenie čiarky za posledným členom pre while inx < po_index:
    if inx != 0:
        print(prva_hodnota + druha_hodnota)
    else:
        print(inx)
'''
fib_iteracne_while(po_index)

# využitie zoznamu (pre po_index >= 1)
def fib_v_zozname(po_index):
    fib = [0,1]
    for i in range(2,po_index+1):
        fib.append(fib[-1] + fib[-2])
    return fib

print("\nIteračný výpočet s ukladaním do zoznamu:")
zoznam = fib_v_zozname(po_index)
print(zoznam)
print("Ten istý zoznam vypísaný ako reťazec:")
```

```
retazec = ', '.join( str(clen) for clen in zoznam )
print(retazec)

# REKURZIA
def fib_rek_def(index):
    if index == 0:
        return 0
    elif index == 1:
        return 1
    else:
        return fib_rek_def(index-1)+fib_rek_def(index-2)

def fib_rek(index):
    if index < 2:
        return index
    else:
        return fib_rek(index-1)+fib_rek(index-2)

print("\nRekurzívny výpočet:")
for inx in range(po_index):
    print(fib_rek(inx), end=", ")
print(fib_rek(po_index))

# zefektívnenie rekurzie použitím slovníka (dict)
def fib_pom_slovnika(index, slovnik={}):
    if index in slovnik:        # najpravdepodobnejšia udalosť, hodnota už bola počítaná - je v slovníku
        return slovnik [index]
    elif index > 1:                    # vypočítanie novej hodnoty a vloženie do slovníka (rek.vetva)
        slovnik [index] = fib_pom_slovnika(index-1) + fib_pom_slovnika(index-2)
        return slovnik [index]
    return index                       # pre index == 0 alebo index == 1 (nerekurzívna vetva)

print("\nRekurzívny výpočet s využitím slovníka (dict):")
#print("Hodnota na indexe {} je {}".format(po_index, fib_pom_slovnika(po_index)))
for fib_inx in range(0,po_index):
    print(fib_pom_slovnika(fib_inx), end=", ")
print(fib_pom_slovnika(po_index))
```