Řešení

Binární vyhledávání

Řešení, které najde první prvek i bez vnořeného cyklu:

```
function BINARY_SEARCH(A,x)
    l ← 0                                          ▷ Levý okraj prohledávaného pole
    r ← length(A) − 1                              ▷ Pravý okraj prohledávaného pole
    while l ≤ r do                                 ▷ Dokud prohledáváme pole s alespoň jedním prvkem
        m ← ⌊(l + r)/2⌋                            ▷ Najdeme střed pole
        if A[m] equals x then
            i ← m                                  ▷ Uložíme si pozici hledaného prvku (nemusí být finální)
            r = m − 1                              ▷ Posuneme pravý okraj těsně před střed
        else if A[m] > x then
            r = m − 1                              ▷ Posuneme pravý okraj těsně před střed
        else
            l = m + 1                              ▷ Posuneme levý okraj těsně za střed
        end if
    end while
    return i                                       ▷ Vrátíme nalezený index prvku
end function
```

Další možností je najít libovolný výskyt prvku a následně pomocí vnořeného cyklu posunout index doleva na správnou pozicí. To ovšem v nejhorším případě znamená $\mathcal{O}(n)$ kroků (např. pokud pole obsahuje pouze prvky se stejnou hodnotou), **nejedná se tedy už striktně vzato o binární vyhledávání**.

Téměř seřazené pole

Na seřazení pole můžeme použít standardní variantu algoritmu Insertion sort.

Časovou složitost $\mathcal{O}(nk)$ můžeme dokázat indukcí podle pozice prvku:

- V kroku $i = 1$ máme posloupnost o délce 1, která je setříděna triviálně.
- V kroku $i = 2, \ldots, n$ se snažíme zatřídit prvek na pozici $i - 1$ do setříděné posloupnosti v levé části pole. Při zatřiďování prvku neprovedeme více než k kroků. Proč?

  Ze zadání víme, že prvek je nejvýše k pozic od své pozice seřazeném poli. Pokud bychom prvek posunuli o $k+x$ kroků doleva (kde $x \geq 1$), znamenalo by to, že před něj musíme v dalších krocích algoritmu zařadit x prvků, které ho posunou na správnou pozici. Každý z těchto prvků bychom museli posunout o více než k pozic (protože je řadíme před prvek, který už jsme posunuli o více než k pozic). Tedy alespoň v posledním kroku algoritmu by se našel jeden prvek, který bychom posunuli o více než k pozic a prvek již na této pozici zůstal, což je v rozporu s podmínkou v zadání.

Celkem tedy máme n iterací (jednou pro každý prvek, který zatřiďujeme) a k kroků v každé iteraci, což nám dává časovou složitost $\mathcal{O}(nk)$.

Další možností je upravit algoritmus Selection sort, aby jeho vnitřní cyklus procházel jen k pozic od

aktuální pozice. Časová složitost vyplývá přímo ze zápisu algoritmu (vnější cyklus provádí n kroků, vnitřní cyklus provádí k kroků), měli bychom ale dokázat jeho správnost.

Zařazení prvku na správnou pozici si můžeme rozdělit na dva případy:

1. Pokud je prvek *napravo* (o maximálně k pozic) od i-té pozice v seřazeném poli, algoritmus ho v i-tém kroku najde a protože je v tomto kroku nejmenším prvkem v intervalu $(i, i + k)$, tak ho na i-tou pozici dosadí.
2. Pokud je prvek *nalevo* (o maximálně k pozic) od i-té pozice v seřazeném poli, dostane se v krocích $i - k, \ldots, i$ na svou pozici nebo napravo od ní, protože bude vyměněn za menší prvky, čímž se případ převede na případ v bodu 1).

Řazení spojového seznamu

Na řazení můžeme použít algoritmus Merge sort přizpůsobený pro práci se spojovým seznamem.

Stejně jako v běžné verzi algoritmu Merge sort postupně sléváme seřazené běhy o délce $1, 2, 4, \ldots, n/2$. Nemůžeme ale používat pomocné pole. Proto si při slévání dvou běhů L, R (operace MERGE) držíme několik ukazatelů:

- x_h: *hlava*
- x_l: *následující prvek v běhu* L
- x_r: *následující prvek v běhu* R

Hlava x_h je prvek, na který v každém kroku napojíme buď x_l, nebo x_r. V prvním kroku je hlava prvek těsně před začátkem běhu L.

- Pokud hodnota $x_l < x_r$, tak na hlavu napojíme x_l. Následně hlavu a x_l posuneme o prvek dál, tedy $x_h \leftarrow x_l$, $x_l \leftarrow x_{l+1}$.
- Pokud hodnota $x_l > x_r$, tak na hlavu napojíme x_r. Následně hlavu a x_r posuneme o prvek dál, tedy $x_h \leftarrow x_r$, $x_r \leftarrow x_{r+1}$.
- Opakujeme, dokud jedním ukazatelem nedojdeme na konec běhu. Pak již pouze napojíme zbytek druhého běhu.

Pro každou délku běhu nám stačí jeden průchod seznamem, protože po každé operaci MERGE pouze přejdeme pomocí dopředných ukazatelů na začátky následujících běhů. Velikost běhu se v každém kroku zdvojnásobí, stejně jako v běžné verzi Merge sortu tedy algoritmus pracuje v čase $\mathcal{O}(n \log N)$.

Mince bez vah

a) Protože víme, že pouze falešná mince je lehčí, odhalí ji libovolné porovnání s jinou mincí. Stačí tedy zajistit, aby se každá mince účastnila alespoň jednoho porovnání. Navíc můžeme jednu minci vynechat: pokud všechna porovnání dopadnou rovností, zbývající mince je falešná. Žádost by tedy mohla vypadat např. takto:

$$m_1 \leftrightarrow m_2$$
$$m_3 \leftrightarrow m_4$$
$$m_5 \leftrightarrow m_6$$
$$m_7 \leftrightarrow m_8$$

b) V tomto případě musíme zvážit všechny dvojice mincí. Pokud bychom nějakou dvojici nezvážili, mohlo by se stát, že tyto dvě mince budou vyhodnoceny jako lehčí v porovnání se všemi ostatními. Bez porovnání této dvojice ale nebudeme vědět, která z mincí je nejlehčí. Celkem tedy potřebujeme 8 vážení pro první minci (s mincemi $m_2 \ldots m_9$), 7 vážení pro druhou minci (s mincemi $m_3 \ldots m_9$, protože s m_1 už jsme minci porovnali), atd., celkem:

$$\sum_{i=0}^{n-1} i = 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + 0 = 36 \text{ vážení}$$