SZZ – otázka I2

Algoritmy řazení a vyhledávání. Algoritmy prohledávání grafů a prohledávání stavového prostoru.

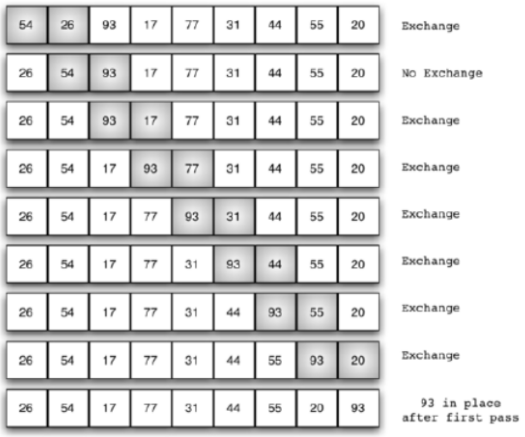
Reference: Algoritmy a programování (https://cw.fel.cvut.cz/b181/courses/b3b33alp/prednasky/start - 4, 5, 11, 13)

Úvodní pojmy

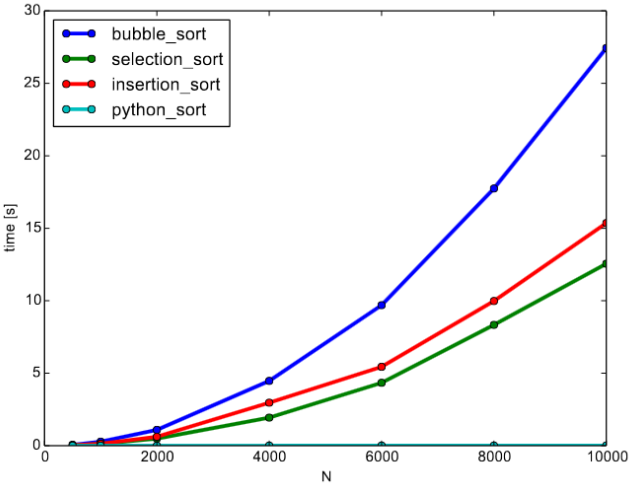
- algoritmus – přesný, detailní a úplný postup
- program – zápis algoritmu v nějakém programovacím jazyce

Řazení (třídění)

- máme vstupní posloupnost *a* a nějakou porovnávací relaci (≤)
- požadovaný výstup: permutace *b* vstupní posloupnosti *a*, ve které platí $b_i \leq b_{i+1}$
- třídění lze provádět
  - na místě (ve stejném poli) – Python: `a.sort()`
  - vytvořením nového pole se seřazenými hodnotami – v Pythonu: `sorted(a)`
  - vytvořením pole indexů, které uvádějí pořadí hodnot v původním poli
- stabilní třídění – třídění, které má tu vlastnost, že zachová pořadí ekvivalentních prvků
- bubble sort
  1. porovná dva sousední prvky a prohodí je, pokud jsou ve špatném pořadí
  2. posune se na další dvojici
  3. opakuje se, dokud nedojdeme na konec pole
  - během jednoho cyklu se na konec dostane největší číslo
  - je třeba tento cyklus opakovat, aby se na správné místo dostalo i 2., 3. (atd.) největší číslo
  - vnější cyklus proběhne *(N-1)*krát, vnitřní *i*krát, *i*<*N*
  - složitost: $O(N^2)$

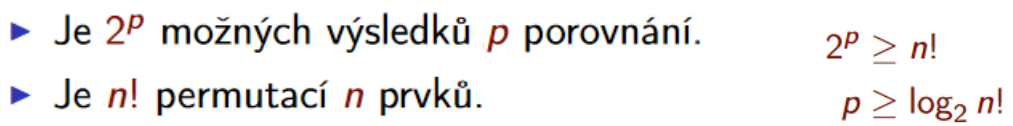


- insertion sort
  - vezmeme první dva prvky posloupnosti a seřadíme je
  - vezmeme třetí prvek a zařadíme ho na správné místo v předchozí posloupnosti
  - vezmeme další prvek a zařadíme ho na správné místo…
  - opakuje se, dokud nezařadíme poslední prvek na správné místo
  - složitost: $O(N^2)$
- selection sort
  - vyberu největší prvek a zařadím ho na konec
  - vyberu 2. největší prvek a zařadím ho na 2. místo od konce
  - vyberu další největší prvek a zařadím ho na další místo od konce…
  - opakuje se až k nejmenšímu prvku
  - složitost: $O(N^2)$
- obrázek: porovnání časové náročnosti □



- algoritmy zmíněné výše jsou pomalé – kvadratická složitost
- rychlejší algoritmy většinou využívají toho, že si problém rozdělí na menší sekvence, které postupně vyřeší
- větší rychlost je dosažena obvykle na úkor větší paměťové náročnosti (čekající podposloupnosti je potřeba uchovávat v paměti)
- rekurze – funkce volá sama sebe (obvykle takto opakovaně dělí problém na menší části, které je snazší vyřešit)

- merge sort
    1. rozděl pole na dvě poloviny (podobně velké části)
    2. každou polovinu setřiď rekurzivně pomocí merge sort
    3. spoj setříděné poloviny
    - jak běží rekurze: pokud má část, která se má setřídit, více než 1 prvek, volá se na ni opět merge sort rozdělí se na další dvě části, které je opět třeba setřídit
    - dělení na části probíhá, dokud každá část nemá délku 1 – seřazení je triviální
    - části o délce 1 se pak postupně skládají do větších celků ve správném pořadí
    - potřebuje pomocná pole = neprobíhá na místě
    - je stabilní
    - složitost: $O(n \times \log n)$
    - lze implementovat pomocí fronty
- quick sort
    1. vybere z pole prvek p (pivot)
    2. setřídíme podposloupnosti na obou stranách od pivota
    3. rekurzivně opakujeme quick sort na podposloupnosti
    - probíhá na místě
    - není stabilní
    - průměrná složitost: $O(n \times \log n)$
    - složitost v nejhorším případě: $O(n^2)$
    - typicky bývá nejrychlejší
    - lze implementovat pomocí zásobníku
- nelze dosáhnout lepší asymptotické složitosti než $n \times \log n$



- radix sort
    - funguje pouze pro třídění n přirozených čísel (nebo řetězců) pevné délky k
    - časová složitost $O(nk)$
    - může být implementován jako stabilní
    - potřebuje pomocnou paměť
    - máme přihrádku pro každou možnou číslici (nebo znak řetězce)
        1. každý prvek přidáme do přihrádky dle číslice daného řádu
        2. obsah přihrádek vypíšeme v pořadí dle hodnoty číslic
- heap sort
    1. vytvoříme z pole haldu
    2. opakovaně odebíráme nejmenší prvek získáváme setříděnou posloupnost
    - složitost v nejhorším i průměrném případě: $O(n \times \log n)$
    - nepotřebuje pomocnou paměť

Vyhledávání

- vyhledáváme hodnotu (dále „q“ jako querry) v poli nebo jiné struktuře (dále „a“ jako array)
- výstup: informace, zda se v dané struktuře hodnota nachází (0/1), případně kde (index)
- pokud hledání opakujeme mnohokrát, může být výhodné si vstupní strukturu setřídit
- nejjednodušší implementace v Pythonu – operátor in:

```
hledaná hodnota in vstupní struktura  True/False

2 in [0, 2, 4, 6, 8, 10]  True

'ha' in 'Praha'  True

'something good' in 'Brno'  False
```

- lineární (sekvenční) vyhledávání
  - procházíme *a* dokud nenajdeme *q*
  - složitost *O(N)*, kde N je délka pole *a*
  - v nejhorším případě musíme projít všechny prvky
- binární vyhledávání (metoda půlení intervalu)
  - vyhledáváme v setříděné posloupnosti ($a_i < a_{i+1}$)
  - hledaný prvek *q* porovnáme s prostřední hodnotou posloupnosti, pro další krok si vybereme jen odpovídající podposloupnost a opakujeme
  - konec: pokud najdeme *q* nebo pokud už nemáme žádnou hodnotu k porovnání
  - složitost *O(*log *n)*
  - pokud tedy vyhledáváme v *a* více hodnot, vyplatí se nejdříve si ho setřídit

Prohledávání grafů

- binární vyhledávací strom
  - struktura pro rychlé vyhledání porovnatelných dat
  - podporované operace: vložení prvku, odstranění prvku, dotaz na přítomnost prvku
  - rychlé operace (složitost *O(*log *n)* nebo lepší)
  - vlastnosti
    - klíč v uzlu není menší než všechny uzly v levém podstromu
    - klíč v uzlu není větší než všechny uzly v pravém podstromu
  - nejhorší případ – degenerovaný strom – hloubka *n-1* □ skoro všechny prvky jsou v jednom podstromu, složitost podporovaných operací je pak *O(n)*
  - pro zabránění nejhoršímu případu lze stromy průběžně vyvažovat
- graf – uzly + hrany (orientované/neorientované)
  - reprezentace je možná pomocí matice sousednosti
  - hledání všech nejkratších cest v grafu – Floyd-Warshallův algoritmus
    - hledá všechny nejkratší cesty
    - dynamické programování – postupně aktualizujeme nejkratší cesty mezi vrcholy
    - nalezne matici délek nejkratších cest mezi všemi dvojicemi uzlů na základě matice sousednosti s nezápornými cykly
  - nejkratší cesty z daného uzlu – Dijkstrův algoritmus
    - na vstupu je orientovaný graf s nezápornými cenami hran a počáteční uzel
    - hledá nejkratší cestu do všech ostatních uzlů
    - prioritní prohledávání do šířky s aktualizací při nalezení kratší cesty

Prohledávání stavového prostoru

- stavový prostor
  - množina stavů *S*
  - počáteční stav s_0
  - množina cílových stavů *T*
  - seznam akcí *A*
  - přechodová funkce *f:* $S \times A$ □ *S*
- hledáme sekvenci akcí a stavů, která převede počáteční stav do žádaného cílového stavu
- příklady: plánování cesty, procházení bludištěm, řešení sudoku, …
- prohledávání do hloubky (depth first search)
  - *„V každém stavu zkusím něco udělat. Když už to nejdem tak se vrátím a zkusím něco jiného.“*
  - pokud jsem v cíli, hotovo
  - existuje-li neprozkoumaný následník, rekurzivně ho prozkoumám
  - pokud neexistuje, vrátím se o jednu akci zpět
  - prozkoumané uzly označuji, abych se do nich nemusel vracet
  - pamatuji si cestu do aktuálního uzlu
  - nerekurzivní řešení: otevřené uzly ukládáme do zásobníku
  - vlastnosti
    - používá zásobník
    - malá paměťová náročnost *O(D)*, *D* je max. hloubka stavového prostoru
    - velká časová náročnost $O(b^D)$, *b* je faktor větvení

        - vhodné pro fyzické prohledávání nebo pokud je změna stavu výpočetně náročná
        - je vhodné omezení maximální hloubky, v případě nenalezení řešení lze omezení zvýšit
        - nenajde vždy nejkratší řešení
        - při špatném rozhodnutí může náprava trvat dlouho
- prohledávání do šířky (breadth first search)
    - hledáme v pořadí délky sekvence akcí
    - nejprve prozkoumáme všechny sousedy, pak všechny jejich sousedy atd.
    - lze si představit jako šíření vlny z daného bodu
    - velká paměťová náročnost $O(b^d)$, d je délka řešení
    - velká časová náročnost$O(b^d)$
    - vždy najde řešení (bez omezení hloubky)
    - vždy najde nejkratší řešení
    - zásobník nahradíme frontou
- informované prohledávání (informed search)
    - víme, kam jdeme začneme nejslibnějšími akcemi
    - urychlení řešení
    - může, ale nemusí zaručit nalezení řešení
    - prioritní (hladové) vyhledávání (greedy search)
        - akce mají ceny
        - pro každý stav odhadneme cenu dosažení minima
        - akce čekající na zpracování ukládáme do prioritní fronty a bereme je v pořadí podle odhadnuté ceny
        - nalezení řešení je zaručeno
        - není zaručeno nalezení minima
        - časová a prostorová složitost jako u prohledávání do šířky
    - algoritmus A^*
        - prioritní hledání ignoruje cenu aktuální cesty, zavedeme tedy cenu cesty z kořene do aktuálního uzlu
        - pořadí vyhledávání je určeno celkovou cenou aktuální + zbývající (odhad) cesty
        - výhoda: neprodlužuje příliš dlouhé cesty dál
        - není zaručeno optimální řešení
        - časová a prostorová složitost jako u prohledávání do šířky
- problém splnitelnosti
    - speciální případ prohledávání stavového prostoru
    - stav se skládá z proměnných
    - cílové stavy musí splňovat podmínky
    - cesta dosažení cíle není důležitá
    - příklady: logické hádanky (8 dam na šachovnici, sudoku, aritmogram, …)
    - popis problému
        - množina proměnných, každá má svůj obor
        - množina podmínek
        - hledáme takové hodnoty proměnných, aby byly splněny všechny podmínky
    - algoritmy
        - hrubou silou
        - postupné přiřazování
            - stav je často velký – snažíme se nekopírovat
            - včasná eliminace špatných větví („pruning“)
        - propagace omezujících podmínek
        - lokální hledání
            - postupná optimalizace (postupně zlepšujeme předchozí odhad)
            - využití předchozích řešení
            - minimalizujeme počet konfliktů
        - strukturou problému
            - graf závislostí
            - rozklad na podúlohy (stromová struktura)