Algoritmizace

Rozděl a panuj

Rozděl a panuj / Divide et impera

Dekompozice: Rozděl problém na podproblémy

Rekurzivně najdi řešení podproblémů

Syntéza: Z řešení podproblémů sestroj řešení původního problému

☀ **Příklady**

- Hanojská věž
- vyhodnocení aritmet. výrazu (v infixové notaci)
- třídění sléváním MergeSort
- třídění rozdělováním QuickSort

Třídění sléváním – MergeSort

Báze: Pole délky $n \leq 1$ je již setříděno

Dekompozice: Pole délky $n \geq 2$ rozděl na poloviny

Rekurzivně setřiď obě části

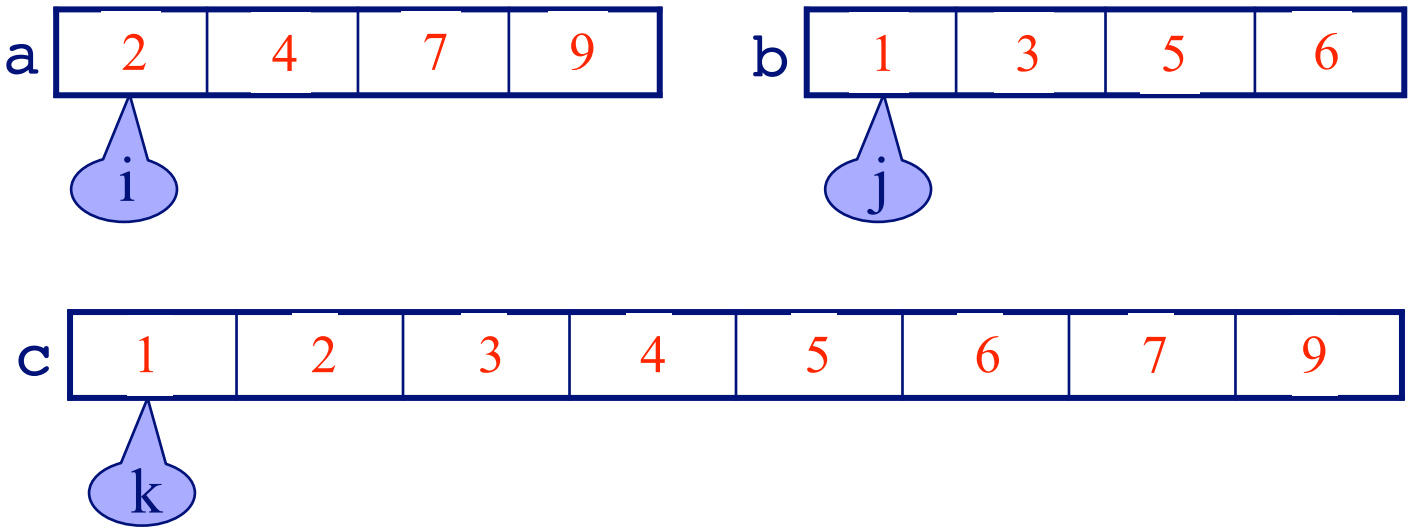
Syntéza: Obě setříděné části slej (merge) do jednoho setříděného pole

Jak sloučíme dvě setříděná pole?

Pozorování

- dvě setříděná pole lze sloučit do jednoho výsledného setříděného pole v lineárním čase
- = operace slévání (merge)

Příklad



MergeSort – dekompozice & rekurze

```
def mergeSort(a):
    if len(a) > 1:
        stred = len(a)//2 # stred pole
        # kopie první a druhé poloviny
        levy, pravy = a[:stred], a[stred:]
        # obě poloviny setřídíme
        mergeSort(levy)
        mergeSort(pravy)
        # indexy polí levy, pravy, a
        i = j = k = 0
```

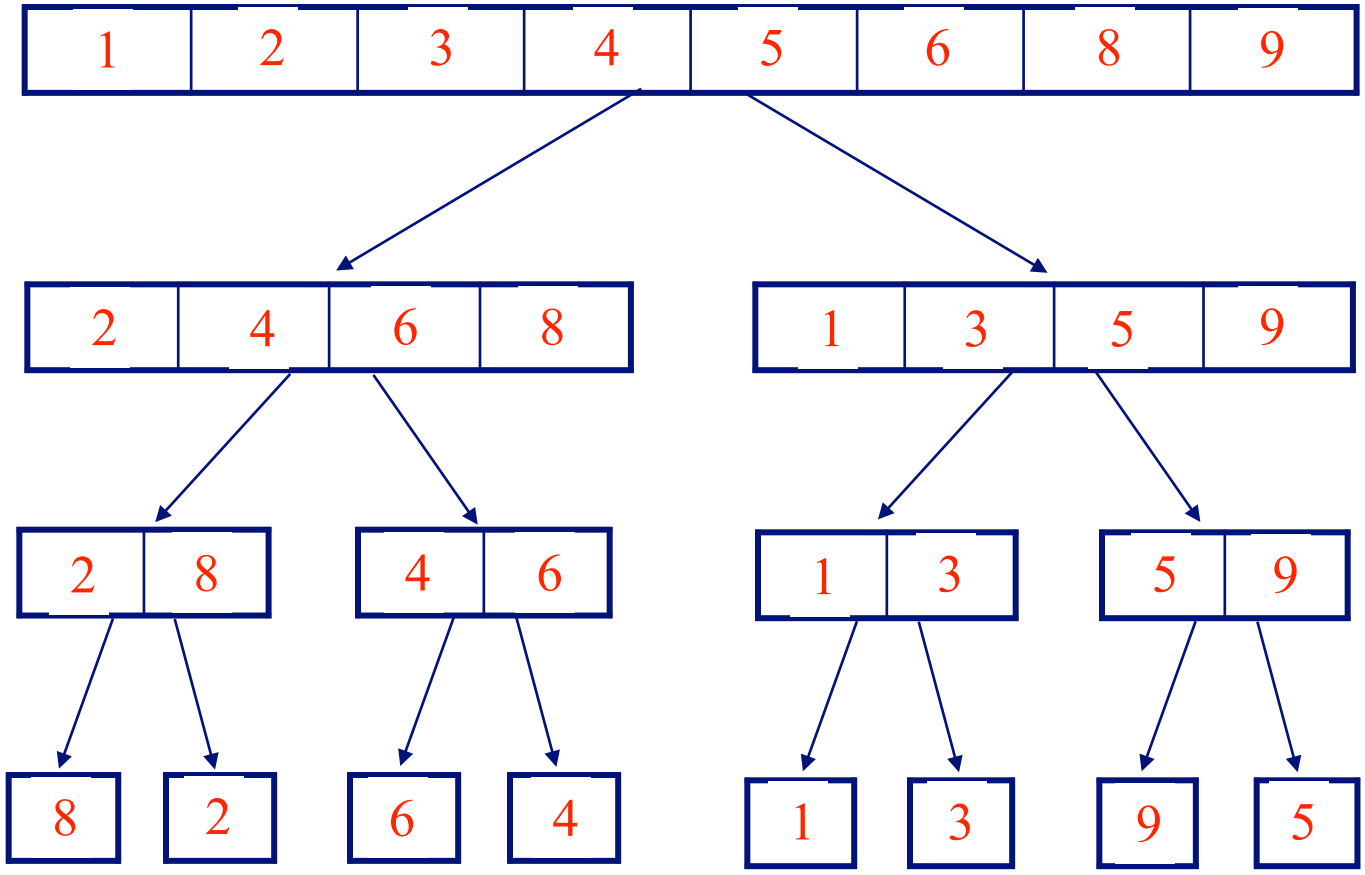
MergeSort – slévání

```
# slejeme setříděné části
# levy[] a pravy[] do pole a[]
while i < len(levy) and j < len(pravy):
    if levy[i] < pravy[j]:
        a[k] = levy[i]
        i += 1
    else:
        a[k] = pravy[j]
        j += 1
    k += 1
```

MergeSort – slévání – závěr

```
# na konec připojíme zbylé prvky
# z levé či pravé části
while i < len(levy):
   a[k] = levy[i]
   i,k = i+1,k+1
while j < len(pravy):
   a[k] = pravy[j]
   j,k = j+1,k+1
```

MergeSort – průběh výpočtu



- výpočet – průchod stromem rekurze do hloubky
- zásobník dosud nezpracovaných podúloh

Strom rekurze

Vrcholy

- podúlohy (rekurzivně) volané během (rekurzivního) algoritmu

Kořen

- původní úloha

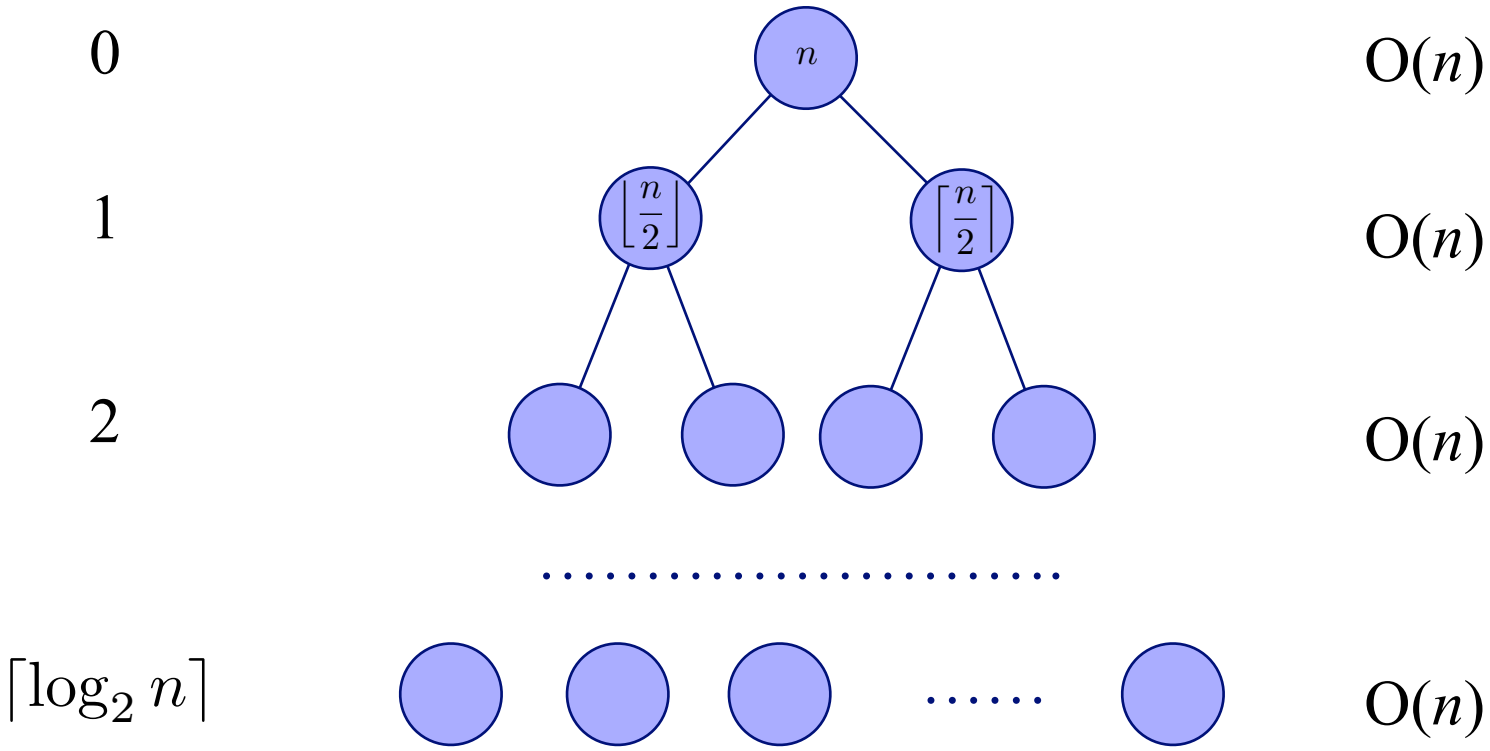
Děti každého rodiče

- představují podúlohy, (rekurzivně) volané v rodičovské úloze

Strom rekurze

hladina

čas



Časová složitost

- $T(n) = \lceil \log_2 n \rceil \cdot O(n) = O(n \log n)$

MergeSort – prostorová složitost

Prostor

- pracovní paměť: pomocné pole pro slévání
- zásobník pro obsluhu rekurze

Odvození velikosti pracovní paměti pomocí stromu rekurze

- vrchol stromu → úsek délky k → prostor $\Theta(k)$
- v paměti je vždy právě zpracovávaný vrchol
- + všichni jeho předchůdci
- nejhorší případ: pracovní paměť pro všechny podúlohy (vrcholy) na cestě z kořene do listu

MergeSort – prostorová složitost

Nechť $n = 2^k$. Pak pracovní paměť

$$S(n) = O(n) + O(n/2) + O(n/4) + \cdots + O(1) = O(n)$$

Nechť $2^k < n < 2^{k+1}$.

- položme $n^* = 2^{k+1}$
- pak $n^* < 2 \cdot n$

$$\begin{aligned} S(n) &\leq O(n^*) + O(n^*/2) + O(n^*/4) + \cdots + O(1) \\ &\leq c \cdot n^* < c \cdot 2 \cdot n \ = O(n) \end{aligned}$$

Složitost – alternativní odvození

Časovou i prostorovou složitost vyjádříme rekurentním vztahem

$T(1) = O(1), T(n) = 2 \cdot T(n/2) + O(n)$

$S(1) = O(1), S(n) = S(n/2) + O(n)$

Řešení

- postupné dosazování
- hypotéza o přesném řešení
- důkaz matematickou indukcí

MergeSort bez rekurze

☀ Idea

- běh – souvislý setříděný úsek pole
- začneme s 1prvkovými běhy vstupního pole
- poté budeme slévat vždy dva sousední běhy do jediného běhu o dvojnásobné délce
- v poslední iteraci bude pole obsahovat jediný běh

Časová složitost $O(n \log n)$

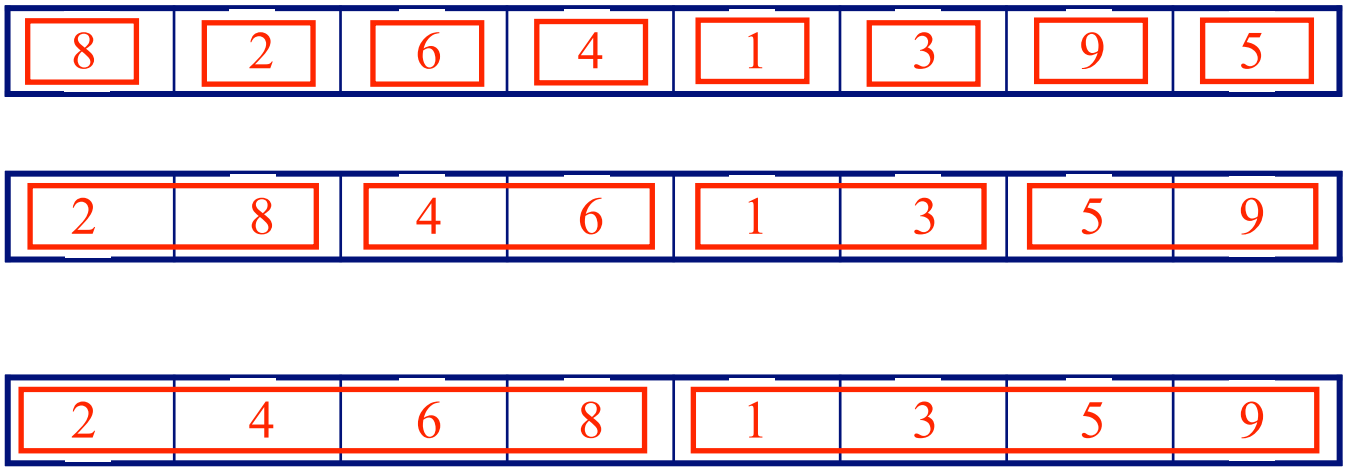
- jako rekurzivní MergeSort

Prostorová složitost $O(n)$

- potřebujeme pomocné pole, do něhož sléváme cílové běhy ze zdrojových běhů v poli původním

Nerekurzivní MergeSort – čas

délka běhu | čas



.....

2^k O(n)

$$2^k < n \implies k < \log_2 n \leq \lceil \log_2 n \rceil \implies k + 1 \leq \lceil \log_2 n \rceil$$

Časová složitost

- $T(n) = \lceil \log_2 n \rceil \cdot O(n) = O(n \log n)$

MergeSort – slévání

```
def merge(a,temp,zacatek,stred,konec):
 i,j,k = zacatek,stred+1,zacatek
 while i <= stred and j <= konec:
    if a[i] < a[j]:
        temp[k] = a[i]
        i += 1
    else:
        temp[k] = a[j]
        j += 1
    k += 1
```

MergeSort – slévání – závěr

```
# na konec připojí zbylé prvky
# z levého / pravého běhu
while i <= stred:
   temp[k] = a[i]
   i,k = i+1,k+1
while j <= konec:
   temp[k] = a[j]
   j,k = j+1,k+1
# výsledek zkopíruje do pole a
a[zacatek:konec+1]=
                    temp[zacatek:konec+1]
```

MergeSort – třídění

```
def mergeSort(a):
 n = len(a)  # délka vstupního pole
 temp = [None] * n # alokuje pomocné

 # slévá sousední běhy délek 1,2,4,...
 beh = 1
 while beh < n:
  for zacatek in range(0,n-beh,2*beh):
      stred = zacatek + beh - 1
      konec = min(stred + beh, n-1)
      merge(a,temp,zacatek,stred,konec)
  beh *= 2
```

Problém 

① Vylepšete implementaci algoritmu MergeSort tak, abyste se na konci funkce Merge vyhnuli kopírování z pomocného pole `temp` zpět do vstupního pole `a`

```
a[zacatek:konec+1] = temp[zacatek:konec+1]
```

② Rozmyslete si, jak by šel algoritmus MergeSort využít pro setřídění spojových seznamů.

Půjde to snáze rekurzí nebo iterací?

Aplikace: Vnější třídění

Problém

- setřídit data, která se nevejdou do RAM

☝ Nové kritérium

- minimalizovat I/0 operace

Idea

- slévání běhů
- běh uložen v souboru
- fáze rozdělování, fáze slévání

Přímé slučování

Na začátku

- tříděná data v jednou souboru (triviální běhy délky 1)

Fáze rozdělování

- setříděné běhy ze vstupního souboru rozdělujeme
- střídavě do dvou pomocných souborů S1 a S2

Fáze slévání

- nejprve sloučíme první běhy z S1 a S2 do běhu
- o dvojnásobné délce, zapíšeme do výstupního souboru
- stejně pro druhé běhy
- takto sloučením S1 a S2 vytvoříme jeden soubor
- tvořený běhy o dvojnásobné délce

Přímé slučování

Složitost

- chceme minimalizovat počet operací se soubory
- # read = # write
- $2n \lceil \log_2 n \rceil$

Vylepšení

- sloučit rozdělování + slévání
- slévat více běhů
- využít přirozeně setříděných částí vstupu
- předtřídit počáteční běhy v RAM

Jednofázové slučování

- Idea: sloučit rozdělování + slévání
  - rozdělování do 2 souborů jen na začátku
  - dále: 2 fáze → 1 fáze

Nové běhy hned rozdělujeme

  - střídavě do dvou souborů

Počet operací

  - # read $\approx n \log_2 n$

Zvýšení stupně slučování

☝ Idea: k-cestné slučování

- rozdělujeme k běhů do k souborů

Počet kroků p

- $k^{p-1} < n \leq k^p$
- $p = \lceil \log_k n \rceil$
- zrychlení $\log_2 n \,/\, \log_k n = \log_2 k$

✔ Jak vybrat minimum z k běhů?

- halda: čas O(log k)

✖ Počet otevřených souborů

- 2fázové: k + 1 soubor
- 1fázové: 2k souborů (k vstupních a k výstupních)

Přirozené slučování

- Idea: využít již setříděné úseky na vstupu
  - některé úseky vstupu mohou být již setříděny
  - proč toho nevyužít?

Počáteční běhy

- již setříděné úseky vstupu
- délka běhu ≥ 1

Počet kroků

- počet počátečních běhů b
- počet kroků $\log_2 n \rightarrow \log_2 b$

Časová složitost

- $O(n \log b)$

Předtřídit počáteční běhy

- Idea: předtřídit počáteční běhy v RAM
  - lze spojit s rozdělovací fází 1.kroku
  - # read / write se nezvýší

Časová složitost

- m = délka přetříděného běhu
- počet běhů $\lceil n/m \rceil$
- celkový čas $O(n \log(n/m))$

MergeSort – další variace

TimSort

- Tim Peters (2002), využívá Python od verze 2.3
- stabilní hybridní třídící algoritmus
- metoda: Peter McIlroy, SODA 1993
- délka běhu $b \in \langle 32,64 \rangle$ tak, aby n/b byla mocnina 2
- běhy jsou setříděny vkládáním (InsertionSort)
- slévání běhů (MergeSort)

Třídění rozdělováním – QuickSort

☀ Idea

- pole délky ≤ 1 je již setříděno
- jinak ve vstupním poli zvolíme prvek zvaný pivot
- prvky v poli přeskupíme tak, aby výsledné pole tvořily dva navazující úseky
  - » levý s prvky $\leq$ pivot
  - » pravý s prvky $\geq$ pivot
- oba úseky setřídíme rekurzivně

QuickSort – Partition

```
def quickSort(a,start,stop):
  """ setřídí a[start..stop] """
  levy, pravy = start, stop #ukazatele
  pivot = a[(levy+pravy)//2]
  while levy <= pravy: #nepřeskočily se
   while a[levy]<pivot: #patří vlevo
     levy += 1  #posouvat levý ukazatel
   while pivot<a[pravy]:#patří vpravo
     pravy -= 1#posouvat pravý ukazatel
  #pokračuje –>
```

QuickSort – Partition

```
if levy < pravy:  # nesetkaly se
  # vyměň hodnoty
  a[levy], a[pravy] = a[pravy], a[levy]
if levy <= pravy: # nepřeskočily se
  # pokračuj
  levy, pravy = levy + 1, pravy - 1
# konec while-cyklu
```

QuickSort – rekurze

```
# netriviální úseky setřiď rekurzivně
 if start < pravy:
   quickSort(a, start, pravy)
 if levy < stop:
   quickSort(a, levy, stop)
```

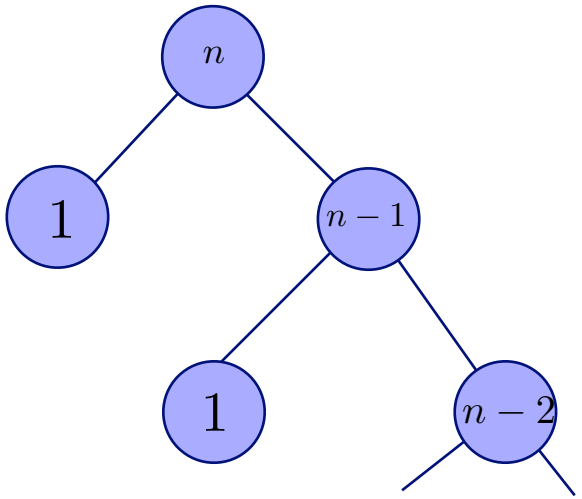
QuickSort – časová složitost

Časová složitost $T(n)$ v nejhorším případě

- $T(n) = O(n) + \max_{1 \le i \le n-1}(T(i)+T(n-i))$

Horní odhad

- pomocí stromu rekurze
- na každé hladině: čas $O(n)$
- počet hladin $\le n$
- $T(n) = O(n^2)$



Dolní odhad

- $T(n) \ge \Omega(n) + T(n-1) = \Omega(n^2)$

✎ pro jaké vstupy nastane?

QuickSort – časová složitost

Časová složitost v nejlepším případě

- co když se podaří v každém kroku zvolit jako pivotový prvek medián ?
- $T(n) = O(n) + 2\,T(n/2)$
- výška stromu rekurze logaritmická
- čas $O(n \cdot \log n)$

Časová složitost v průměrném případě $O(n \cdot \log n)$

- důkaz : ADS I

QuickSort – zrychlení

Volba pivotového prvku

- určení mediánu v čase $O(n) \Rightarrow$ teoretické zrychlení na čas $O(n \cdot \log n)$ i v nejhorším případě

Volba pivota v praxi

- náhodně
- medián z `a[1]`, `a[n//2]`, `a[-1]`
- medián ze 3 náhodně zvolených

Odstranění konce rekurze

- malé $n \Rightarrow$ setřídit přímou metodou (InsertionSort)

QuickSort bez rekurze

Rekuzivní volání ⇒

- uložení hranic úseku k setřídění do zásobníku
- simulace průchodu stromem rekurze

✔ Může být úspornější

- časově (obsluha rekurze)
- i prostorově (stačí zásobník logaritmické velikosti)

✖ Složitější kód

QuickSort – prostorová složitost

Netřídí na místě !

K obsluze rekurze je třeba zásobník

- prostorová složitost $O(n)$
- lze zlepšit na $O(\log n)$

✎ Jaké úseky budeme ukládat na zásobník, aby nám stačil prostor $O(\log n)$?

✎ Problémy: QuickSort

③ Prvek posloupnosti délky n nazveme skoromediánem, pokud leží v uspořádané posloupnosti na k-tém místě, kde

$$n/4 \leq k \leq 3n/4 .$$

Kdyby se nám v algoritmu QuickSort podařilo v každém kroku vybrat v čase $O(1)$ jako pivota skoromedián, změnilo by to nějak časovou složitost v nejhorším případě?

Ve vaší analýze složitosti můžete předpokládat, že prvky na vstupu jsou po dvou různé.

✎ Problémy: QuickSort

④ Upravte QuickSort tak, aby pro

- zadané pole `a`
- a číslo k

vrátil k-tý nejmenší prvek `a` bez toho, že by pole třídil. Mělo by k tomu stačit jediné rekurzivní volání.

Časová složitost výsledného algoritmu je v nejhorším případě stále kvadratická, dá se ale ukázat, že v průměrném případě pracuje v lineárním čase.