4. Merge Sort

Algoritmizace

Jiří Balun

Obsah

1 Datová struktura strom

2 Merge Sort
- Procedura Merge
- Popis algoritmu
- Složitost algoritmu

3 Dolní odhad složitosti třídění porovnáním

4 Radix Sort
- Popis algoritmu
- Složitost algoritmu

Rekapitulace a motivace

Na minulé přednášce

- problém třídění a základní vlastnosti třídících algoritmů
- Insertion Sort – třídění vkládáním
- Quick Sort – rychlé třídění

Na této přednášce

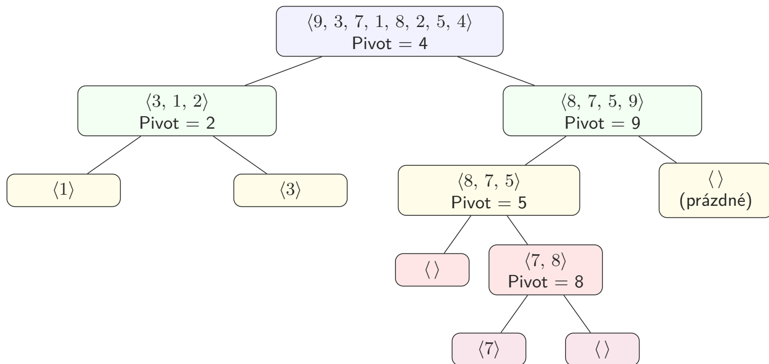
- datová struktura: strom
- Merge Sort – třídění slučováním
- dolní odhad složitosti algoritmů třídění porovnáním
- Radix Sort – číslicové třídění

Datová struktura strom

Motivace: datová struktura *strom*

- při analýze rekurzivních algoritmů (např. Quick Sort, Merge Sort atd.) se přirozeně objevuje **hierarchická struktura** volání
- jedno volání funkce vytváří dvě podvolání – vzniká rozvětvená struktura
- takovou hierarchii neumíme snadno zobrazit pomocí pole nebo seznamu

Řešení: Použijeme datovou strukturu **strom**, kterou umíme zachytit vztah „část–podčást“.



Strom – intuitivní představa

Strom představuje hierarchické uspořádání objektů:

- prvky stromu se nazývají **uzly** (nodes),
- uzel, který je v této hierarchii nejvýše položený, se nazývá **kořen** (root),
- uzly mohou mít **potomky** (children),
- uzly bez potomků jsou **listy** (leaves).

Vztahy a vlastnosti:

- každý uzel (kromě kořene) má právě jednoho **rodiče** (parent)
- mezi uzly existuje přirozený **směr od kořene k listům**
- strom je **binární**, pokud každý uzel má nejvýše dva potomky
- **velikost stromu:** počet uzlů
- **výška (hloubka):** délka nejdelší cesty od kořene k listu
- **podstrom:** část stromu tvořená uzlem a všemi jeho potomky

Merge Sort

Sloučení dvou setříděných polí

Motivace

Problém: Máme dvě již **setříděné posloupnosti** čísel a chceme je sloučit do jedné.

Pozorování:

- protože jsou obě části **setříděné**, víme, že nejmenší prvek celé dvojice se nachází vždy na začátku jedné z nich,
- stačí tedy **porovnávat pouze první prvky obou posloupností** a menší z nich zapsat do výsledku,
- tento postup opakujeme, dokud nevyčerpáme jeden z obou vstupů,
- zbytek druhého vstupu je již setříděný, proto jej stačí přepsat do výsledku.

Důsledek:

- celé sloučení lze provést **v jediném průchodu oběma poli**, tj. v čase úměrném jejich celkové délce.

Procedura Merge (část 1/2)

Intuice: na vstupu máme pole A a indexy, které vymezují dvě setříděné podpole $A[p\,..\,q]$ a $A[q+1\,..\,r]$. Výsledkem Merge je setříděné podpole $A[p\,..\,r]$.

Algorithm 1: merge (part 1/2)

Input: array $A = \langle a_0, a_1, \ldots, a_{n-1} \rangle$, indexes p, q and r
Output: sorted subarray $A[p\,..\,r]$

1 $n_1 \leftarrow q - p + 1$
2 $n_2 \leftarrow r - q$
3 initialize two new arrays $L[0\,..\,n_1]$ and $R[0\,..\,n_2]$
4 **for** $i \leftarrow 0$ **to** $n_1 - 1$ **do**
5 $\quad L[i] \leftarrow A[p + i]$
6 **for** $j \leftarrow 0$ **to** $n_2 - 1$ **do**
7 $\quad R[j] \leftarrow A[q + j + 1]$

- Merge vyžaduje další pamět pro pomocné pole L a R
- řádky 4-5 kopírují podpole $A[p\,..\,q]$ do pole L (analogicky 6-7 kopírují $A[q{+}1\,..\,r]$ do R)

Procedura Merge (část 2/2)

Algorithm 1: merge (part 2/2)

```
8   L[n_1] ← ∞
9   R[n_2] ← ∞
10  i ← 0
11  j ← 0
12  for k ← p to r do
13      if L[i] ≤ R[j] then
14          A[k] ← L[i]
15          i ← i + 1
16      else
17          A[k] ← R[j]
18          j ← j + 1
```

- na řádcích 8-9 používáme metodu zarážky: pokud například vyčerpáme všechny prvky z pole L dřív než z R, pak každý zbývající prvek v R bude menší než $L[n_1] = \infty$

Příklad: průběh procedury Merge

Příklad (část 1/3)

Cíl: sloučit dvě setříděné části pole do jedné setříděné posloupnosti.

Vstupy:

$$A = \langle 1, 4, 5, 7, 2, 3, 6, 8 \rangle$$

$$p = 0, \quad q = 3, \quad r = 7$$

Postup:

- vypočítáme hodnoty $n_1 = q - p + 1 = 4$ a $n_2 = r - q = 4$,
- vytvoříme pomocné pole $L[0\,..\,4]$ a $R[0\,..\,4]$ (do každého můžeme uložit 5 prvků),
- po inicializaci (na řádku 12) je $i = 0, j = 0$ a pomocné pole mají tento obsah:

$$L = \langle 1, 4, 5, 7, \infty \rangle, \quad R = \langle 2, 3, 6, 8, \infty \rangle$$

- chceme vytvořit z L a R jednu setříděnou posloupnost $\langle 1, 2, 3, 4, 5, 6, 7, 8 \rangle$ postupným vybíráním menšího prvku z čel obou polí.

Příklad (část 2/3)

Z předchozího kroku máme pole $L = \langle 1, 4, 5, 7, \infty \rangle$ a $R = \langle 2, 3, 6, 8, \infty \rangle$.

Krok	**Porovnání**	**Zápis do $A[k]$**	**Stav ukazatelů**
1	$L[0] = 1 < R[0] = 2$	$A[0] \leftarrow 1$	$i = 1,\ j = 0$
2	$L[1] = 4 > R[0] = 2$	$A[1] \leftarrow 2$	$i = 1,\ j = 1$
3	$L[1] = 4 > R[1] = 3$	$A[2] \leftarrow 3$	$i = 1,\ j = 2$
4	$L[1] = 4 < R[2] = 6$	$A[3] \leftarrow 4$	$i = 2,\ j = 2$

Aktuální obsah pole:

$$A = \langle 1, 2, 3, 4, 1, 3, 6, 8 \rangle$$

- Na každém kroku se vybírá menší prvek z čela L a R.
- Ukazatel v daném poli se posune o 1.

Příklad (část 3/3)

Z pvního kroku máme pole $L = \langle 1, 4, 5, 7, \infty \rangle$ a $R = \langle 2, 3, 6, 8, \infty \rangle$.

Krok	**Porovnání**	**Zápis do** $A[k]$	**Stav ukazatelů**
5	$L[2] = 5 < R[2] = 6$	$A[4] \leftarrow 5$	$i = 3,\ j = 2$
6	$L[3] = 7 > R[2] = 6$	$A[5] \leftarrow 6$	$i = 3,\ j = 3$
7	$L[3] = 7 < R[3] = 8$	$A[6] \leftarrow 7$	$i = 4,\ j = 3$
8	$L[4] = \infty > R[3] = 8$	$A[7] \leftarrow 8$	$i = 4,\ j = 4$

Výsledek:

$$A = \langle 1, 2, 3, 4, 5, 6, 7, 8 \rangle$$

Shrnutí:

- po vyčerpání jednoho z polí (L nebo R) se zbývající prvky druhého jednoduše dopíší,
- procedura merge má časovou složitost $\Theta(n)$, kde $n = n_1 + n_2$.

Složitost procedury Merge

Intuice: složitost Merge je **lineární** v čase i paměti vzhledem k velikosti vstupu.

Časová složitost:

- každý prvek z levého i pravého pole je zpracován právě jednou,
- na každém kroku provádíme jedno porovnání a jedno přiřazení,
- celkový počet kroků je tedy úměrný součtu délek obou polí:

$$T(n_1, n_2) = n_1 + n_2$$

- pro $n = n_1 + n_2$ tedy platí:

$$T(n) = \Theta(n)$$

Paměťová složitost:

- vytváříme dvě pomocná pole L a R o velikostech n_1 a n_2.
- navíc potřebujeme jen několik proměnných pro indexy a počty
- celková paměťová náročnost: $M(n) = n_1 + n_2 = \Theta(n)$

Merge Sort (třídění slučováním)

Intuice: vstupní pole půlíme na čím dál menší části, dokud nezískáme jednoprvková pole (už jsou triviálně setříděné) a ty pak postupně slučujeme pomocí Merge.

Algorithm 2: merge-sort

Input: array $A = \langle a_0, a_1, \ldots, a_{n-1} \rangle$, indexes p and r
Output: sorted subarray $A[p \mathinner{..} r]$

```
1 if p < r then
2 |   q ← ⌊(p + r)/2⌋
3 |   merge-sort(A, p, q)
4 |   merge-sort(A, q + 1, r)
5 |   merge(A, p, q, r)
```

- Merge Sort je rekurzivní algoritmus založený na principu „rozděj a panuj“
- jde o stabilní algoritmus, který třídí pomocí porovnání, ale netřídí na místě
- korektnost algoritmu plyne z korektnosti procedury Merge (evidentní)

Příklad: Merge Sort

Příklad

Vstupy:

$$A = \langle 12, 3, 7, 9, 14, 6, 11, 2 \rangle, \quad p = 0, \quad r = 7$$

Postup:

- vypočítáme $q = \lfloor (p + r)/2 \rfloor = \lfloor (0 + 7)/2 \rfloor = 3$
- rekurzivní volání merge-sort$(A, 0, 3)$ setřídí první 4 prvky:

$$A = \langle 3, 7, 9, 12, 14, 6, 11, 2 \rangle$$

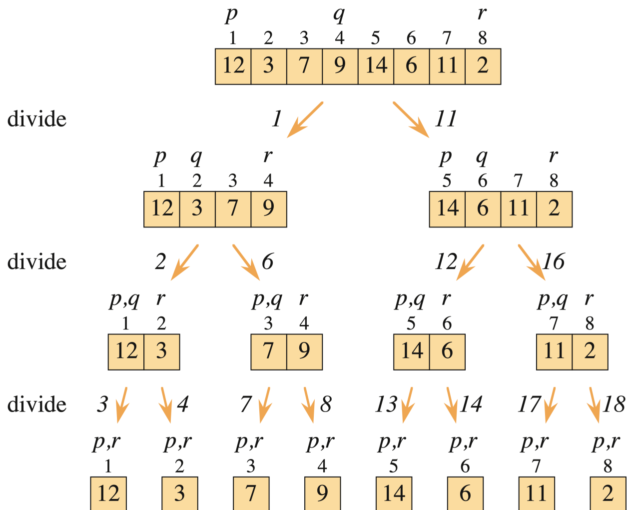
- rekurzivní volání merge-sort$(A, 4, 7)$ setřídí druhou polovinu pole:

$$A = \langle 3, 7, 9, 12, 2, 6, 11, 14 \rangle$$

- zavolání merge$(A, 0, 3, 7)$ sloučí obě podpole $A[0 \mathinner{..} 3]$ a $A[4 \mathinner{..} 7]$:

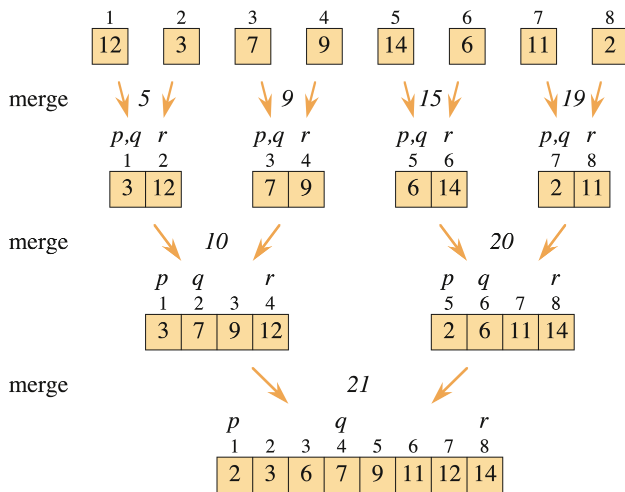
$$A = \langle 2, 3, 6, 7, 9, 11, 12, 14 \rangle$$

Příklad: Merge Sort – fáze *rozdělování*



Obrázek: pole jsou zde indexovány od 1 do n.
Zdroj: Introduction to Algorithms (Cormen et al.)

Příklad: Merge Sort – fáze *slučování*



Obrázek: pole jsou zde indexovány od 1 do n.
Zdroj: Introduction to Algorithms (Cormen et al.)

Paměťová složitost algoritmu Merge sort

Pozorování:

- algoritmus Merge Sort používá pomocnou proceduru Merge, která při každém slučování vytváří **dvě dočasná pole** L a R
- dohromady mohou mít až n prvků, kde n je délka vstupního pole
- proto pro jedno volání Merge potřebujeme dodatečně $\Theta(n)$ paměti

Celkový dopad:

- rekurzivní rozklad pole sice vytváří mnoho volání Merge, ale pomocná pole L a R existují pouze po dobu vykonávání této procedury
- v jeden okamžik tedy potřebujeme pomocná pole jen pro (jedno) aktuální volání Merge
- celková dodatečná paměť (mimo vstupního pole) Merge Sortu je tedy

$$M(n) = \Theta(n)$$

Rekurence pro časovou složitost Merge sortu

Co je rekurence:

- rekurence popisuje vztah mezi **časem potřebným pro vstup velikosti** n a časem potřebným pro menší podproblémy
- jinými slovy: definuje $T(n)$ **rekurzivně pomocí** T**, ale s menší velikostí vstupu**

Pro algoritmus Merge Sort:

$$T(n) = \begin{cases} c_1 & \text{pokud } n = 1, \\ 2T(n/2) + c_2 n & \text{pokud } n > 1. \end{cases}$$

Význam jednotlivých členů:

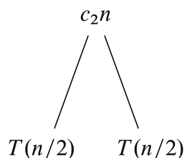
- $T(n)$ – celkový čas pro vstup o velikosti n
- $2T(n/2)$ – čas potřebný pro setřídění dvou polovin pole
- $c_2 n$ – čas na **sloučení** (merge) obou setříděných částí
- c_1 – konstanta představující čas pro řešení triviálního případu $(n = 1)$

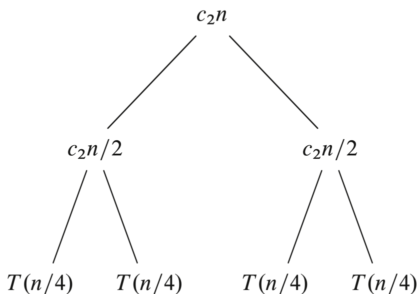
Merge Sort – Rekurzní strom

$$T(n) = \begin{cases} c_1 & \text{pokud } n = 1 \\ 2T(n/2) + c_2 n & \text{pokud } n > 1 \end{cases}$$

- **Úroveň 0 (kořen):** problém velikosti n, cena $c_2 n$.
- **Úroveň 1:** dva problémy velikosti $n/2$, každý s cenou $c_2 n/2$, celkem $2 \times c_2 n/2 = c_2 n$.
- **Úroveň 2:** čtyři problémy velikosti $n/4$, každý s cenou $c_2 n/4$, celkem $4 \times c_2 n/4 = c_2 n$.

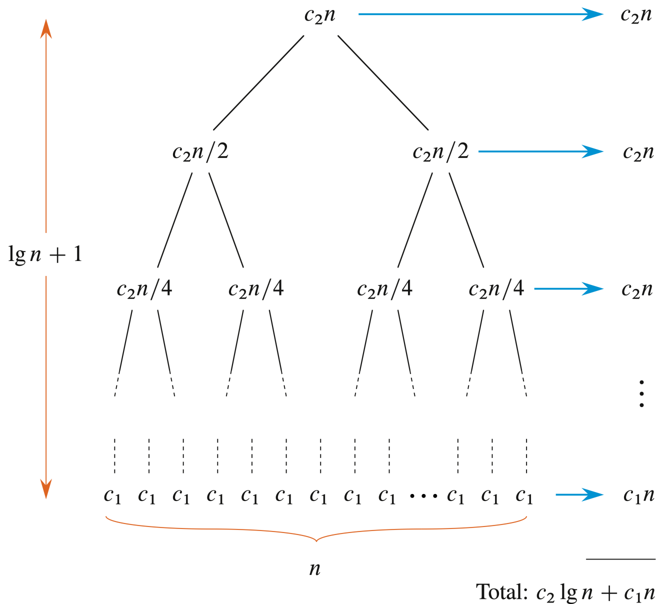
$T(n)$





(a) (b) (c)

Merge Sort – Rekurzní strom



Odvození časové složitosti Merge sortu

Z rekurzního stromu:

- každá úroveň i obsahuje 2^i podproblémů o velikosti $n/2^i$
- cena každého podproblému je $c_2 \cdot \frac{n}{2^i}$
- celková cena je stejná na každé úrovni:

$$2^i \cdot c_2 \frac{n}{2^i} = c_2 n$$

Počet úrovní:

- na každé úrovni vzniká dvojnásobný počet podproblémů o poloviční velikosti
- rekurze končí, když velikost podproblému dosáhne 1, proto vyřešíme rovnici pro k:

$$\frac{n}{2^k} = 1 \quad \Rightarrow \quad n = 2^k \quad \Rightarrow \quad k = \log_2 n$$

- hledané k je výška rekurzivního stromu, proto **počet úrovní je** $\log_2 n + 1$ (včetně listů)

Celková cena:

$$T(n) = (\text{cena na úrovni}) \times (\text{počet úrovní}) + (\text{cena listů})$$

$$T(n) = (c_2 n) \times \log_2 n + c_1 n$$

Výsledná časová složitost je tedy: $T(n) = \Theta(n \log n)$

Složitost Merge Sortu ve všech případech

Pozorování:

- algoritmus Merge Sort vždy provádí stejný počet dělení vstupu:

$$n \to \frac{n}{2} \to \frac{n}{4} \to \cdots \to 1$$

  tedy $\log_2 n$ úrovní rekurze.
- v každé úrovni se vždy provádí operace slučování (*merge*) všech prvků – bez ohledu na jejich pořadí

Důsledek:

- čas potřebný pro slučování na každé úrovni je vždy $c_2 n = \Theta(n)$
- celkový čas tedy závisí pouze na velikosti vstupu, nikoli na jeho uspořádání
- Merge Sort má stejný průběh pro každý vstup – nerozlišuje „snadné“ a „těžké“ případy

Dolní odhad složitosti třídění porovnáním

Dolní odhad složitosti třídění porovnáním

Otázka: Existuje třídicí algoritmus založený na porovnávání, který je v nejhorším případě rychlejší než $O(n \log n)$? Má smysl takový algoritmus hledat?

Pozorování:

- algoritmy třídění porovnáním rozhodují pouze podle výsledků porovnání dvojic prvků
- průběh výpočtu lze zobrazit jako **rozhodovací strom**, kde každé porovnání odpovídá jednomu rozvětvení uzlu na dva potomky, např. pro $\langle a_0, a_1, a_2 \rangle$ a porovnání $a_0 < a_1$:
  - vznikne potomek $\langle a_1, a_0, a_2 \rangle$ prohozením a_0 a a_1 pro $a_1 < a_0$
  - a v dalším potomku zůstane pole stejné $\langle a_0, a_1, a_2 \rangle$, protože $a_0 < a_1$
- každý list tohoto stromu odpovídá jedné možné permutaci vstupu (tj. jednomu výsledku setřídění vstupního pole)

Důsledek:

- aby algoritmus rozlišil všech $n!$ možných vstupních permutací délky n, musí mít rozhodovací strom alespoň $n!$ listů
- z toho plyne, že počet potřebných porovnání roste alespoň jako $\log_2(n!) = \Omega(n \log n)$

Dolní odhad složitosti třídění porovnáním

Formální odvození:

- binární strom výšky h má nejvýše 2^h listů – v našem případě musí platit $2^h \geq n!$
- logaritmováním nerovnosti dostáváme (nebudeme dokazovat $\log_2(n!) = \Omega(n \log n)$):

$$h \geq \log_2(n!) = \Omega(n \log n)$$

- výška h udává nejdelší cestu od kořene k listu a tím i počet potřebných porovnání pro vytvoření výsledné permutace ze vstupního pole
- dolní odhad pro složitost v nejhorším případě je tedy $T(n) = \Omega(n \log n)$

Důsledky:

- žádný třídicí algoritmus založený pouze na porovnávání nemůže být asymptoticky rychlejší než $n \log n$
- algoritmy jako Merge sort nebo Heap sort (uvidíme později) tento dolní odhad dosahují – jsou tedy **asymptoticky optimální**
- pro lepší složitost musíme opustit třídění porovnáváním (Counting Sort, Radix Sort...)

Radix Sort

Radix sort

Motivace: dosud jsme používali algoritmy, které třídí porovnáváním prvků. Lze třídit rychleji než $\Omega(n \log n)$, pokud využijeme strukturu prvků v A (například počet cifer)?

Algorithm 3: radix-sort

Input: array $A = \langle a_0, \ldots, a_{n-1} \rangle$ of numbers, each with d digits
Output: sorted array A

```
1 for i ← 1 to d do
2     stable-sort(A, i)
```

- třídí čísla podle jednotlivých **cifer** (bitů, číslic, složek), od nejméně významné po nejvíce významnou cifru (na řádku 2 třídíme A podle i-té cifry)
- každé třídění podle cifry musí být **stabilní**, tj. zachová pořadí prvků se stejnou cifrou
- nehodí se pro libovolné typy dat, ale dokáže třídit rychle, pokud jsou čísla krátká a jeho jednotlivé cifry (složky) jsou z malého rozsahu

Příklad: průběh Radix Sortu

Příklad

Vstup:

$$A = \langle 329, 457, 657, 839, 436, 720, 355 \rangle$$

Postup:

- jako stabilní třídící algoritmus zvolíme například Insertion Sort,
- třídíme podle jednotlivých cifer z prava doleva:

1. krok (jednotky): $\Rightarrow \langle 720, 355, 436, 457, 657, 839, 329 \rangle$

2. krok (desítky): $\Rightarrow \langle 720, 329, 839, 436, 355, 657, 457 \rangle$

3. krok (stovky): $\Rightarrow \langle 329, 355, 436, 457, 657, 720, 839 \rangle$

Dojde tedy ke třem spuštěním Insertion Sortu:

- po každém kroku je pole setříděno vzestupně podle aktuální cifry,
- díky stabilitě mezikroků se výsledné pořadí stává plně setříděným.

Složitost algoritmu Radix sort

Časová složitost:

- typicky se používá **Counting Sort**, který má čas $\Theta(n + k)$ pro k různých cifer
- každé třídění podle jedné cifry (složky) trvá tedy $\Theta(n + k)$
- třídíme d cifer, proto složitost v obecném případě máme celkem

$$T(n) = \Theta(d \cdot (n + k))$$

- pro pevný počet cifer d a konstantní k (např. cifry 0–9) dostáváme tedy

$$T(n) = \Theta(n)$$

Paměťová složitost:

- určena stabilním tříděním podle cifry (např. pro Counting Sort): $\Theta(n + k)$

Závěrem

Problém třídění:

- třídění porovnáním prvků nemůže být v nejhorším případě rychlejší než $\Omega(n \log n)$
- Merge Sort – třídění slučováním
  - časová složitost je všech případech $\Theta(n \log n)$
  - paměťová složitost je $\Theta(n)$
- Radix Sort – číslicové třídění
  - časová i paměťová složitost je závislá na použitém stabilním třídicím algoritmu
  - v ideálním případě (pro konstantní d a k) je lineární $\Theta(n)$

Samostudium

- Counting Sort – na 2. slidech od prof. Bělohlávka (108–112)