# 3. Problém třídění

## Algoritmizace

Jiří Balun

# Obsah

# Rekapitulace a motivace

## Na minulé přednášce

- datová struktura pole
- časová složitost algoritmu: v nejhorším případě a v průměrném případě
- srovnání funkcí, které se často vyskytují jako složitosti algoritmů
- asymptotická složitost: $\mathcal{O}$-notace, $\Omega$-notace, $\Theta$-notace a ostré meze

## Na této přednášce

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

# Problém třídění

# Problém třídění

**Motivace:** třídění (sorting) patří mezi základní a nejčastěji studované algoritmické úlohy. Objevuje se v mnoha oblastech informatiky: databáze, vyhledávání a zpracování dat atd.

Problém – třídění čísel

**Vstup:** posloupnost (nebo také $n$-tice) přirozených čísel $\langle a_0, \ldots, a_{n-1} \rangle$.
**Výstup:** permutace vstupní posloupnosti $\langle a'_0, \ldots, a'_{n-1} \rangle$ taková, že $a'_0 \leq a'_1 \leq \ldots \leq a'_{n-1}$.

- hledáme efektivný způsoby, jak z neuspořádané posloupnosti vytvořit seřazenou
- pro jednoduchost se omezujeme na celá čísla, ale třídit lze i jiné datové typy

**Proč je třídění důležité?**

- je jedním z nejčastějších kroků při zpracování dat
- usnadňuje další operace (například vyhledávání, odstraňování duplicit atd.)
- umožňuje efektivní organizaci informací a rychlejší přístup k datům
- představuje typický příklad úlohy, kde lze porovnávat efektivitu algoritmů

# Naivní třídicí algoritmus

**Intuice:** nejjednodušší způsob třídění spočívá v tom, že zkoušíme všechny možné přeuspořádání (permutace) vstupní posloupnosti dokud nenajdeme tu, která je seřazená.

**Popis algoritmu:**

- z vstupní posloupnosti $\langle a_0, a_1, \ldots, a_{n-1} \rangle$ postupně generuj všechny její permutace
- každou permutaci otestuj, zda je seřazená (tj. zda platí $a'_0 \leq a'_1 \leq \ldots \leq a'_{n-1}$)
- pokud nalezneš seřazenou permutaci, vrať ji jako výsledek

**Časová složitost:**

- počet všech permutací délky $n$ je $n!$ (faktoriál $n$)
- kontrola jedné permutace zabere čas úměrný $n$
- celkový počet kroků v nejhorším případě je tedy $\Theta(n!)$
- **tento postup je extrémně pomalý** – složitost roste mnohem rychleji než jakýkoli prakticky použitelný algoritmus

# Základní vlastnosti třídicích algoritmů

**Motivace:** při porovnávání třídicích algoritmů nás často zajímají kromě časové složitosti také jejich další vlastnosti, které ovlivňují vhodnost použití v praxi.

## Definice

**Třídění porovnáváním** rozhoduje o pořadí prvků pouze pomocí operací „$<$“ nebo „$>$“:

- pro setřídění čísel používá jen informaci získanou porovnáváním (nepouívá například informaci o poslední cifře čísla).

**Třídění na místě** nevyžaduje další paměť úměrnou velikosti vstupu:

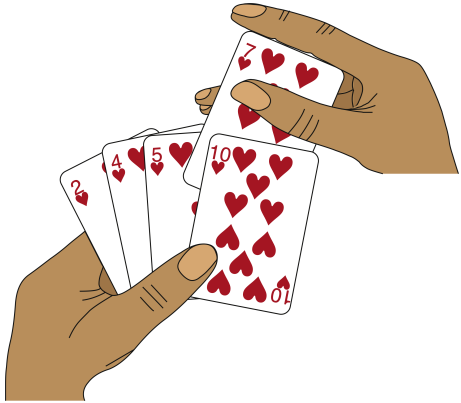
- algoritmus mění pořadí prvků přímo v poli,
- potřebná pomocná paměť je jen konstantní nebo velmi malá.

**Stabilní třídicí algoritmus** zachovává pořadí prvků se stejným klíčem:

- dva prvky se stejnou hodnotou zůstávají na výstupu ve stejném pořadí jako na vstupu,
- stabilita je důležitá, pokud se třídí podle více klíčů (např. nejprve podle roku, poté podle měsíce, dnu atd.).

# Insertion Sort

# Motivace: třídění karet v ruce




Zdroj obrázku: Introduction to Algorithms (Cormen et al.)

- začneme s jednou kartou v ruce – ta je už „setříděná“
- každou další kartu vložíme na správné místo mezi již setříděné karty
- postupně tak vzniká stále delší setříděná část, až jsou všechny karty na správném místě

# Insertion Sort

**Intuice:** vnější cyklus postupně prochází čísla $a_1$ až $a_{n-1}$ (to aktuální je uloženo v $key$). Vnitřní cyklus posune všechny čísla v setříděné časti, které jsou větší než $key$, o jeden index dál, čímž získáme pozici kam patří $key$.

**Algorithm 1:** Insertion-Sort

**Input:** array $A = \langle a_0, a_1, \ldots, a_{n-1} \rangle$
**Output:** sorted array $A$

```
1 for j ← 1 to n − 1 do
2     key ← A[j]
3     i ← j − 1
4     while i ≥ 0 and A[i] > key do
5         A[i + 1] ← A[i]
6         i ← i − 1
7     A[i + 1] ← key
```

- jde o stabilní algoritmus, který třídí na místě pomocí porovnání

# Příklad: průběh algoritmu Insertion sort

## Příklad (část 1/3)

**Vstup:** pole osmi čísel

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

**Postup:**

- proměnná $j$ označuje index prvku, který se právě vkládá do setříděné části,
- proměnná $key$ obsahuje hodnotu tohoto prvku,
- prvky $A[0\,..\,j-1]$ jsou vždy setříděné a prvek $A[j]$ musíme zatřídit na správné místo, čímž postupně rozšiřujeme setříděnou část pole zleva doprava.

**Krok 1:** $j = 1$, $key = 2$

[5] 2 4 6 1 3 8 7 (jen první prvek je setříděn)

$\Rightarrow$ [5 5] 4 6 1 3 8 7 (porovnání $2 < 5 \rightarrow$ posun 5)

$\Rightarrow$ [2 5] 4 6 1 3 8 7 ($key = 2$ vložen na začátek pole)

## Příklad (část 2/3)

**Krok 2:** $j = 2$, $key = 4$

[2 5] 4 6 1 3 8 7 (první dva prvky už jsou setříděny)

$\Rightarrow$ [2 5 5] 6 1 3 8 7 (porovnání $4 < 5 \rightarrow$ posun 5)

$\Rightarrow$ [2 4 5] 6 1 3 8 7 ($4 > 2 \rightarrow key = 4$ vložen mezi 2 a 5)

**Krok 3:** $j = 3$, $key = 6$

[2 4 5] 6 1 3 8 7 ($6 > 5 \rightarrow$ bez posunu)

$\Rightarrow$ [2 4 5 6] 1 3 8 7 ($key = 6$ je vložen na svůj původní index)

**Krok 4:** $j = 4$, $key = 1$

[2 4 5 6] 1 3 8 7

$\Rightarrow$ [2 4 5 6 6] 3 8 7 ($1 < 6 \rightarrow$ posun 6)

$\Rightarrow$ [2 4 5 5 6] 3 8 7 ($1 < 5 \rightarrow$ posun 5)

$\Rightarrow$ [2 4 4 5 6] 3 8 7 ($1 < 4 \rightarrow$ posun 4)

$\Rightarrow$ [2 2 4 5 6] 3 8 7 ($1 < 2 \rightarrow$ posun 2)

$\Rightarrow$ [1 2 4 5 6] 3 8 7 ($key = 1$ vložen na začátek pole)

## Příklad (část 3/3)

**Krok 5:** $j = 5$, $key = 3$

[1 2 4 5 6] 3 8 7

$\Rightarrow$ [1 2 4 5 6 6] 8 7 ($3 < 6 \rightarrow$ posun 6)

$\Rightarrow$ [1 2 4 5 5 6] 8 7 ($3 < 6 \rightarrow$ posun 5)

$\Rightarrow$ [1 2 4 4 5 6] 8 7 ($3 < 6 \rightarrow$ posun 4)

$\Rightarrow$ [1 2 3 4 5 6] 8 7 ($3 > 2 \rightarrow key = 3$ vložen mezi 2 a 4)

**Krok 6:** $j = 6$, $key = 8$

[1 2 3 4 5 6] 8 7 ($8 > 6 \rightarrow$ bez posunu)

$\Rightarrow$ [1 2 3 4 5 6 8] 7 ($key = 8$ je vložen na svůj původní index)

**Krok 7:** $j = 7$, $key = 7$

[1 2 3 4 5 6 8] 7

$\Rightarrow$ [1 2 3 4 5 6 8 8] ($7 < 8 \rightarrow$ posun 8)

$\Rightarrow$ [1 2 3 4 5 6 7 8] ($7 > 6 \rightarrow key = 7$ vložen mezi 6 a 8, konec)

# Korektnost

**Intuice:** ve zdůvodnění korektnosti použijeme tzv. **invariant** – podmínka, která vždy platí při vykonávání nějaké části algoritmu.

**Invariant cyklu na řádku 1:**

- na začátku každé iterace platí, že prvky $A[0\,..\,j-1]$ jsou setříděné
- tj. algoritmus postupně „sestavuje“ uspořádanou posloupnost zleva doprava

**Zdůvodnění korektnosti**

- **Inicializace:** Na začátku (před první iterací) obsahuje setříděnou část pouze jeden prvek, který je triviálně setříděný.
- **Průběžný krok:** Vkládáním prvku $A[j]$ na správné místo mezi $A[0\,..\,j-1]$ se zachová vlastnost, že levá část zůstává setříděná.
- **Ukončení:** Po dokončení poslední iterace cyklu na řádku 1 (tj. $j = n$) je levá část $A[0\,..\,n-1]$ setříděná – tedy celé pole.
- **Závěr:** Invariant cyklu platí ve všech krocích, a proto je algoritmus **korektní**.

# Časová složitost

**Nejhorší případ:**

- nastává tehdy, když je vstupní pole setříděno **v opačném pořadí**
- každý nově vkládaný prvek musí být posunut až na začátek setříděné části
- počet přesunů (porovnání a přiřazení) pro všechny prvky pole:

$$\sum_{i=1}^{n-1} i = 1 + 2 + 3 + \cdots + (n-1) = \frac{n(n-1)}{2}$$

- časová složitost v nejhorším případě je tedy $\Theta(n^2)$

**Průměrný případ:**

- v průměru bývá prvek vložen zhruba do poloviny seřazené části
- počet porovnání a přesunů je tedy také úměrný $n^2$, i když s menším koeficientem než v nejhorším případě
- proto i v průměrném případě je časová složitost $\Theta(n^2)$

# Časová složitost v nejlepším případě

**Motivace:** v praxi se běžně vyskytují případy, kdy vstupní pole je z velké části, nebo dokonce celé, setříděno. Zajímá nás jak Insertion Sort obstojí v takovém případě.

**Nejlepší případ:**

- nastává tehdy, když je vstupní pole již **setříděná vzestupně**, například

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

- každý nový prvek je porovnán pouze s předchozím, protože není potřeba žádný přesun
- v každé iteraci se tedy provede právě jedno porovnání a žádný posun
- celkový počet operací roste lineárně s počtem prvků $n$, tj. Insertion sort je v tomto případě **velmi efektivní** a jeho složitost je $\Theta(n)$

**Závěr:**

- Insertion Sort je efektivní pro malé vstupy nebo téměř seřazená pole, ale pro velké náhodné vstupy je neefektivní

# Quick Sort

# Motivace: princip „rozděl a panuj“

**Intuice:** namísto řešení složitého celého problému najednou jej rozdělíme na menší (jednodušší) části, které vyřešíme samostatně a jejich výsledky spojíme.

**Tři kroky principu „rozděl a panuj“:**

1. **rozděl** – rozdělíme problém na menší podproblémy stejného typu
2. **panuj** – vyřeš podproblémy (rekurzivně)
3. **spoj** – sloučíme výsledky dílčích řešení do celkového výsledku

**V kontextu algoritmu Quick Sort:**

- funkce partition **rozdělí** pole podle pivota, aby menší prvky byly vlevo a větší vpravo
- obě části **setřídíme rekurzivně** funkcí quick-sort
- výsledné pořadí je již **setříděné celé pole**

# Quick Sort

**Intuice:** jedná se o **rekurzivní algoritmus** – během svého běhu volá sám sebe na menší podproblémy, dokud nedosáhne triviálního případu (například pole délky 1).

**Algorithm 2:** quick-sort

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

```
1 if p < r then
2     q ← partition(A, p, r)
3     quick-sort(A, p, q − 1)
4     quick-sort(A, q + 1, r)
```

- indexy $p$ a $r$ říkají, kterou část pole dané rekurzovní volání zrovna třídí
- celé pole setřídíme pomocí quick-sort$(0, n-1)$
- Quick Sort třídí na místě pomocí porovnání, ale není stabilní

# Procedura Partition

**Algorithm 3:** partition

**Input:** array $A = \langle a_0, a_1, \ldots, a_{n-1} \rangle$, indexes $p$ and $r$
**Output:** index of pivot $x$ in array $A$

```
1 x ← A[r]
2 i ← p − 1
3 for j ← p to r − 1 do
4     if A[j] ≤ x then
5         i ← i + 1
6         swap(A[i], A[j])
7 swap(A[i + 1], A[r])
8 return i + 1
```

- prvku $x$ se říká **pivot**, přičemž funkce partition na konci vrací index pivotu
- všechny prvky vlevo od pivota jsou menší než pivot a všechny prvky vpravo jsou větší
- volba pivota může probíhat různě: obvykle je to první, prostřední nebo poslední prvek dané části pole

# Příklad: průběh funkce partition

**Příklad**

**Vstup:** $A = \langle 9, 3, 7, 1, 8, 2, 5, 4 \rangle$, $p = 0$, $r = 7$

**Inicializace:** $x = A[r] = 4$ (pivot), $i = p - 1 = -1$

**Průběh cyklu:**

$j = 0$: [9 3 7 1 8 2 5 4] $A[j] = 9 > 4$ $\Rightarrow$ nic se neděje

$j = 1$: [3 9 7 1 8 2 5 4] $A[j] = 3 \leq 4$ $\Rightarrow i = 0$, prohodíme $A[0] \leftrightarrow A[1]$

$j = 2$: [3 9 7 1 8 2 5 4] $A[j] = 7 > 4$ $\Rightarrow$ nic se neděje

$j = 3$: [3 1 7 9 8 2 5 4] $A[j] = 1 \leq 4$ $\Rightarrow i = 1$, swap $A[1] \leftrightarrow A[3]$

$j = 4$: [3 1 7 9 8 2 5 4] $A[j] = 8 > 4$ $\Rightarrow$ nic se neděje

$j = 5$: [3 1 2 9 8 7 5 4] $A[j] = 2 \leq 4$ $\Rightarrow i = 2$, swap $A[2] \leftrightarrow A[5]$

$j = 6$: [3 1 2 9 8 7 5 4] $A[j] = 5 > 4$ $\Rightarrow$ nic se neděje

**Po smyčce:** $i = 2$, proto prohodíme $A[i+1] \leftrightarrow A[r]$ $\Rightarrow$ $A = \langle 3, 1, 2, 4, 8, 7, 5, 9 \rangle$

**Výsledek:** funkce vrací $i + 1 = 3$, tj. index pivotu.

# Příklad: průběh algoritmu Quick Sort

## Příklad (část 1/3)

**Vstup:**

$$A = \langle 9,\ 3,\ 7,\ 1,\ 8,\ 2,\ 5,\ 4 \rangle$$

**Hloubka rekurze: 1**

- zvolíme pivot $x = 4$ (poslední prvek),
- po zavolání partition$(A, 0, 7)$ získáme

$$A = \langle 3,\ 1,\ 2,\ 4,\ 8,\ 7,\ 5,\ 9 \rangle, \quad q = 3$$

  pivot je na správném místě ($A[3] = 4$).
- rekurzivně třídíme obě části pole (nalevo je $\langle 3, 1, 2 \rangle$ a napravo je $\langle 8, 7, 5, 9 \rangle$)

$$\text{quick-sort}(A, 0, 2) \quad \text{a} \quad \text{quick-sort}(A, 4, 7)$$

## Příklad (část 2/3)

**Hloubka rekurze: 2**

**Levý podproblém** quick-sort$(A, 0, 2)$ **pro podpole** $\langle 3, 1, 2 \rangle$

- pivot $x = 2$ (poslední prvek),
- po zavolání partition$(A, 0, 2)$:

$$\langle 1,\ 2,\ 3,\ 4,\ 8,\ 7,\ 5,\ 9 \rangle, \quad q = 1$$

- levá část $\langle 1 \rangle$ a pravá část $\langle 3 \rangle$ mají délku $1 \Rightarrow$ netřídí se dál.

**Pravý podproblém** quick-sort$(A, 4, 7)$ **pro podpole** $\langle 8, 7, 5, 9 \rangle$

- pivot $x = 9$ (poslední prvek),
- po zavolání partition$(A, 4, 7)$:

$$\langle 1,\ 2,\ 3,\ 4,\ 8,\ 7,\ 5,\ 9 \rangle, \quad q = 7$$

- pivot $9$ je na správném místě, levé podpole je $\langle 8, 7, 5 \rangle$ a pravé je prázdné,
- rekurzivně třídíme levé podpole quick-sort$(A, 4, 6)$.

## Příklad (část 3/3)

**Hloubka rekurze: 3**

- nyní třídíme $\langle 8, 7, 5 \rangle$, tj. quick-sort$(A, 4, 6)$ levá část podproblému quick-sort$(A, 4, 7)$
- pivot $x = 5$ (poslední prvek),
- po partition$(A, 4, 6)$:

$$\langle 1,\ 2,\ 3,\ 4,\ 5,\ 7,\ 8,\ 9 \rangle, \quad q = 4$$

- levá část je prázdná, pravou část $\langle 7, 8 \rangle$ třídíme rekurzivně quick-sort$(A, 5, 6)$.

**Hloubka rekurze: 4**

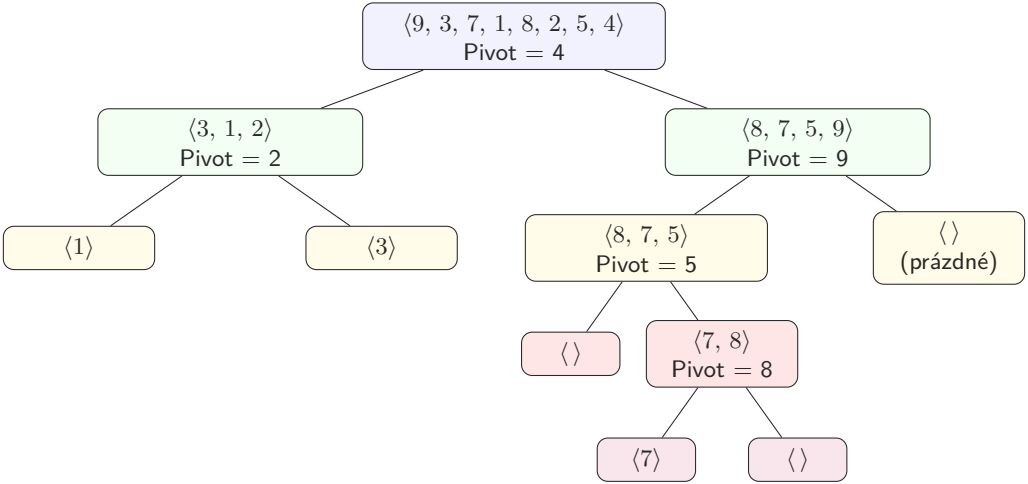
- nyní třídíme $\langle 7, 8 \rangle$, tj. quick-sort$(A, 5, 6)$ pravá část podproblému quick-sort$(A, 4, 6)$
- pivot $x = 8$ (poslední prvek),

$$\langle 1,\ 2,\ 3,\ 4,\ 5,\ 7,\ 8,\ 9 \rangle$$

přičemž levá část $\langle 7 \rangle$ má délku 1 a pravá část je prázdná, proto se dál netřídí.

**Výsledek:** $A = \langle 1,\ 2,\ 3,\ 4,\ 5,\ 7,\ 8,\ 9 \rangle$

# Ilustrace rekurzivních volání Quick Sort




Obrázek: Každé volání quick-sort rozdělí pole podle pivota a rekurzivně třídí levou a pravou část.

# Korektnost algoritmu Quick Sort

**Intuice:** QuickSort je založen na principu rozděl a panuj. Pokud je korektní rozdělení (funkce partition) a rekurzivní třídění menších částí, je korektní i celý algoritmus.

**Invariant cyklu (pro partition):**

- po dokončení partition platí:
  - prvky $A[p..q-1]$ jsou $\leq A[q]$,
  - prvky $A[q+1..r]$ jsou $\geq A[q]$,
  - pivot $A[q]$ je na své správné (konečné) pozici.

**Zdůvodnění korektnosti (rekurzí)**

- **Základ:** Pokud $p \geq r$, pak pole má nejvýše jeden prvek – triviálně setříděno.
- **Indukční krok:** Po partition jsou obě části (nalevo a napravo od pivota) menší než původní pole a rekurzivní volání quick-sort je obě správně setřídí.
- Po návratu z rekurzí je celé pole $A[p..r]$ seřazené.

**Závěr:** Kombinací korektnosti partition a rekurzivního třídění je celý algoritmus **korektní**.

# Časová složitost Quick Sortu v nejhorším případě

**Nejhorší případ:**

- nastává tehdy, když volba pivotu **rozdělí pole nerovnoměrně**
- pivot je vždy **nejmenší nebo největší prvek** v dané části pole
- v takovém případě má jedno podpole délku $n-1$ a druhé délku 0
- počet instrukcí roste kvadraticky s velikostí pole, což vede na časovou složitost $\Theta(n^2)$

**Počty porovnání v partition (nejčastěji vykonávaná instrukce):**

| | | |
|---|---|---|
| 1. volání: | $n-1$ | porovnání (všechny prvky s pivotem) |
| 2. volání: | $n-2$ | porovnání |
| 3. volání: | $n-3$ | porovnání |
| | $\vdots$ | |
| (poslední) | $1$ | porovnání |

**Celkem:**

$$1+2+3+\cdots+(n-1)=\frac{n(n-1)}{2}=\Theta(n^2)$$

# Časová složitost v nejlepším a průměrném případě

**Intuice:** Quick Sort je velmi efektivní, pokud pivoty nejsou voleny extrémně nevhodně.

**Nejlepší případ:**

- nastává, když pivot **rozdělí pole na dvě přibližně stejně velké části**
- každé rekurzivní volání tedy pracuje s polovinou prvků
- funkce partition provede $\mathcal{O}(n)$ operací pro každou úroveň rekurze
- hloubka rekurze je $\log_2 n$, proto dostáváme výslednou složitost $\Theta(n \log n)$

**Průměrný případ:**

- pivoty obvykle nedělí pole zcela rovnoměrně, ale přesto rozdělí pole „dostatečně dobře“
- strom rekurzivních volání má podobnou hloubku jako v nejlepším případě
- proto i složitost v průměrném je lineárně–ogaritmická $\Theta(n \log n)$

# Závěrem

**Třídící algoritmy**

- Insertion Sort – třídění vkládáním
  - složitost v nejhorším případě je $\mathcal{O}(n^2)$
  - složitost v průměrném případě je $\mathcal{O}(n^2)$
  - složitost v nejlepším případě je $\mathcal{O}(n)$
- Quick Sort – rychlé třídění
  - složitost v nejhorším případě je $\mathcal{O}(n^2)$
  - složitost v průměrném případě je $\mathcal{O}(n \log n)$
  - složitost v nejlepším případě je $\mathcal{O}(n \log n)$

**Samostudium**

- Selection Sort – na 2. slidech od prof. Bělohlávka najdete (od slajdu 22)
- Bubble Sort – na 2. slidech od prof. Bělohlávka najdete (od slajdu 56)