# Algoritmy 1

## část 2

Radim BĚLOHLÁVEK
Katedra informatiky
Univerzita Palackého v Olomouci

# Základní datové struktury
## úvod

... více probereme později

# Co je datová struktura?

Volně řečeno, způsob, způsob uložení dat v počítači a způsob, jakým můžeme k datům přistupovat.

Základní datové struktury:

- pole
- seznam (někdy také spojový seznam; jednosměrný nebo dvousměrný)
- zásobník
- fronta
- strom (jedna z nejdůležitějších, existuje mnoho variant, uvidíme)
- graf
- další ...

# Pole

Anglicky "array".

Jednoduchá datová struktura: posloupnost datových položek (stejného typu).

Datovými položkami mohou být čísla (to bude náš nejčastější případ), textové znaky, ale i další položky (jiné datové struktury).

Pole $A$ čísel (celých čísel) velikosti 8 obsahující po řadě čísla 4, -2, 0, 2, 15, 2, 7, 1:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | -2 | 0 | 2 | 15 | 2 | 7 | 1 |

– Píšeme např.: $A = \langle 4, -2, 0, 2, 15, 2, 7, 1 \rangle$.

– Přístup k prvkům pole velikosti $n$:
$A[i]$ . . . $i$-tý prvek pole ($1 \leq i \leq n$), $i$ . . . index,
tj. $A = \langle A[1], A[2], \ldots, A[n] \rangle$,
tedy $A[1] = 4$, $A[2] = -2$, $A[3] = 0$, . . . , $A[7] = 7$, $A[8] = 1$.

– Indexovali jsme “od jedničky” Někdy je výhodné indexovat “od nuly”, tj. pak
$A = \langle A[0], A[1], \dots, A[n-1] \rangle$,
tedy $A[0] = 4$, $A[1] = -2$, $A[2] = 0$, ..., $A[6] = 7$, $A[7] = 1$.
Při zápisu algoritmu musí být jasné, zda indexujeme “od nuly” nebo “od jedničky”. Většinou budeme indexovat “od nuly”.

– Při indexování “od jedničky”:
$A[i] \leftarrow 3$ ... na $i$-té místo pole vloží číslo 3,
$t \leftarrow A[4]$ ... do proměnné $t$ vloží hodnotu čtvrtého prvku pole.
Při indexování “od nuly””:
$A[i-1] \leftarrow 3$ ... na $i$-té místo pole vloží číslo 3,
$t \leftarrow A[3]$ ... do proměnné $t$ vloží hodnotu čtvrtého prvku pole.

– $A[i \dots j]$ ... označuje část pole (“podpole”, samo pole) od $i$-tého do $j$-tého prvku,
tedy $A[i \dots j] = \langle A[i], A[i+1], \dots, A[j] \rangle$.

– Tedy $A$ je $A[0 \dots n-1]$.

Při zápisu algoritmů budeme předpokládat, že velikost pole $A$ je známa a budeme ji označovat $n$ apod., popř se na ni budeme odkazovat $length(A)$, $l(A)$ apod.

Algoritmus, který "vynuluje" všechny prvky pole $A$.

Set-To-Zero($A$)

**for** $i \leftarrow 0$ **to** $n - 1$
  **do** $A[i] \leftarrow 0$

nebo

Set-To-Zero($A$)

**for** $i \leftarrow 0$ **to** $length(A) - 1$
  **do** $A[i] \leftarrow 0$

nebo (při indexování "od jedničky")

Set-To-Zero($A$)

**for** $i \leftarrow 1$ **to** $n$
  **do** $A[i] \leftarrow 0$

# Třídění

# Problém třídění

**problém (třídění):**
**vstup:** $\langle a_1, \dots, a_n \rangle$ (posloupnost $n$ čísel)
**výstup:** permutace $\langle b_1, \dots, b_n \rangle$ vstupní posloupnosti taková, že
$b_1 \leq b_2 \leq \cdots \leq b_n$

Tj. výstupní polsoupnost vznikne přerovnáním prvků vstupní posloupnosti tak, aby byla "setříděna".

Vstupní posloupnost je obvykle reprezentována polem
$A[0 \dots n-1] = \langle a_1, \dots, a_n \rangle$, které po skončení výpočtu podle algoritmu obsahuje setříděnou posloupnost, tj. $A[0 \dots n-1] = \langle b_1, \dots, b_n \rangle$.

vstup $\langle 4, -2, 0, 2, 15, 2, 7, 1 \rangle$
odpovídající výstup $\langle -2, 0, 1, 2, 2, 4, 7, 15 \rangle$

vstup $\langle -2, 4, 5, 8, 10, 15, 37, 91 \rangle$
odpovídající výstup $\langle -2, 4, 5, 8, 10, 15, 37, 91 \rangle$

vstup $\langle 16, 8, 4, 2, 1 \rangle$
odpovídající výstup $\langle 1, 2, 4, 8, 16 \rangle$

**Proč je problém třídění důležitý:**

– Vyskytuje se jako úloha při řešení mnoha úloh zpracování dat.
  – Setřídit pole naměřených hodnot (např. abychom v něm mohli lépe vyhledávat).
  – Setřídit zaměstnance podle věku, popř. podle příjmu.
  – Při přípravě bankovního výpisu z účtu setřídit transakce podle data.
  
  Atd.

– Algoritmy pro řešení složitějších problémů využívají algoritmy pro třídění.

– Často potřebujeme setřídit pole složitějších datových položek než jsou čísla. Např. při třídění zaměstnanců podle příjmu obsahuje pole strukturované záznamy obsahující kromě údaje o příjmu údaj o jménu, zaměstnaneckém čísle, apod. V poli se pak přeuspořádávají celé záznamy, nejen čísla. Pak je třeba zabezpečit, aby se zbytečně nepřemísťovaly velké objemy dat (velké záznamy). To lze vyřešit (přemísťujeme indexu záznamů, nikoli samotné záznamy).
  V principu se ale nic nemění, třídění probíhá podle jisté položky záznamu, která je číslem. Této položce se říká klíč.

– Metodický a historický význam. Algoritmy třídění používají řadu užitečných technik pro návrh algoritmů.
– Problém třídění je zajímavý z hlediska informatiky (známe zajímavé věci o složitosti tohoto problému, např. dolní odhad složitosti). Uvidíme později.

Dva pojmy: Algoritmus třídění

– patří mezi algoritmy třídění porovnáváním (provádí třídění porovnáváním), pokud pro setřídění čísel používá jen informaci získanou porovnáváním čísel (nepoužívá např. informaci o poslední cifře čísla). Takové algoritmy lze používat i pro třídění polí obsahujících jiné, vždy porovnatelné, položky (znaky abecedy apod.).
– pracuje "na místě" (in place), pokud až na konstantní (na velikosti pole nezávislý) počet prvků pole je během činnosti algoritmu uložen mimo pole (např. v pomocné proměnné *temp* pro výměnu prvků pole).

# První, “naivní” algoritmus

Přímo z definice problému třídění lze uvažovat tento algoritmus:

Procházej všechny možné permutace pole $A$, pro každou z nich ověř, zda je pole $A$ setříděné. Pokud ano, skonči. Pokud ne, přejdi k další permutaci.

Je velmi neefektivní. V nejhorším případě musí projít všechny permutace $n$-prvkového pole, a těch je $n!$ ($n$ faktoriál). Víme, že časová složitost takového algoritmu je neúnosná a algoritmus můžeme zavrhnout, aniž bychom ho implementovali a exprimentálně testovali.

**Cvičení**
1. Navrhněte algoritmus, který generuje všechny permutace $n$-prvkového pole (popř. se k tomuto problému vraťte později).
2. Pomocí tohoto algoritmu implementujte výše popsaný algoritmus třídění.

# Insertion Sort

Třídění vkládáním.

Idea tohoto algoritmu je podobná způsobu, jak třídíme $n$ rozdaných karet: $n$ karet leží na začátku na stole. Pravou rukou je bereme a vkládáme do levé ruky tak, že v levé ruce vzniká setříděná posloupnost karet (zleva od nejmenší po největší). Drží-li levá ruka $k$ karet, pak další, $(k+1)$-ní, kartu zatřídíme tak, že ji zprava porovnáváme se setříděnými kartami a vložíme ji na správné místo.

```
Insertion-Sort(A[0..n − 1])
  for j ← 1 to n − 1
    do t ← A[j]
       i ← j − 1
       while i ≥ 0 and A[i] > t
           do A[i + 1] ← A[i]
              i ← i − 1
       A[i + 1] ← t
```

$A[0 \ldots j-1]$ obsahuje setříděnou posloupnost (karty v levé ruce).

Na začátku (při 1. vstupu do cyklu 1–7) je touto setříděnou posloupností $A[0..0]$, tj. $A[0]$.
Při vstupu do cyklu 1–7 s hodnotou $j$ je je touto setříděnou posloupností $A[0..j-1]$

Do $t$ se vloží hodnota, kterou je třeba zatřídit (postupně 2., 3., ... $n$-tý prvek), tj. hodnota $A[j]$.

V cyklu na ř. 4–6 se najde místo pro zařazení prvku $t$ procházením části pole $A[0..j-1]$. Prvky části pole se přitom posouvají vpravo (tím se uvolňuje místo pro $t$).

Na ř. 7 se $t$ vloží na nalezené místo.

Je-li $j < n-1$, jdeme na ř. 1 a postupujeme stejně pro zařazení další hodnoty pole $A$.

**Příklad** (třídění algoritmem Insertion-Sort)
zobrazujeme stav pole $A$ na ř. 1 a na ř. 7 postupně pro $j = 0, 1, \ldots, n-1$.
Máme $n = 6$.
Modře je setříděná část pole. Podtržený je zařazovaný prvek $t$.

vstup:

| | | | | | |
|---|---|---|---|---|---|
| 7 | 1 | 5 | 9 | 7 | 0 |

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $j = 1$, ř. 1 | **7** | $\underline{1}$ | 5 | 9 | 7 | 0 | ř. 7 | $\mathbf{\underline{1}}$ | **7** | 5 | 9 | 7 | 0 |
| $j = 2$, ř. 1 | **1** | **7** | $\underline{5}$ | 9 | 7 | 0 | ř. 7 | **1** | $\mathbf{\underline{5}}$ | **7** | 9 | 7 | 0 |
| $j = 3$, ř. 1 | **1** | **5** | **7** | $\underline{9}$ | 7 | 0 | ř. 7 | **1** | **5** | **7** | $\mathbf{\underline{9}}$ | 7 | 0 |
| $j = 4$, ř. 1 | **1** | **5** | **7** | **9** | $\underline{7}$ | 0 | ř. 7 | **1** | **5** | **7** | $\mathbf{\underline{7}}$ | **9** | 0 |
| $j = 5$, ř. 1 | **1** | **5** | **7** | **7** | **9** | $\underline{0}$ | ř. 7 | **0** | **1** | **5** | **7** | **7** | **9** |

# Správnost algoritmu Insertion Sort

Správnost plyne z následující vlastnosti V:

Na začátku každého cyklu 1–7 obsahuje část $A[0..j-1]$ prvky, které se před spuštěním algoritmu nacházely v této části a které jsou vzestupně uspořádané (setříděné).

To je snadno vidět.
Pro $j = 1$ je to zřejmé (tou částí je $A[0]$).

Pokud platí V pro vstup do cyklu s hodnotou $j = k$, pak V platí i pro vstup s hodnotou $j = k + 1$, protože během cyklu s hodnotou $j = k$ proběhne zařazení prvku $A[k]$ do části $A[0..k-1]$. Při dalším vstupu do cyklu, tj. s hodnotou $j = k + 1$ je tedy část $A[0..j-1]$, tj. část $A[0..k]$, setříděna.

Po skončení je $j = n$ (cyklus se přestane vykonávat, když po zvýšení $j$ o 1 není splněna podmínka $j \leq n-1$), tedy část $A[0..n-1]$ je setříděná, ale část $A[0..n-1]$ je celé pole.

Důkaz správnosti je hotov.

# Složitost algoritmu Insertion Sort

Jaká je časová složitost $T(n)$ v nejhorším případě?

Velikostí vstupu budeme chápat velikost vstupního pole, tj. $n = length(A)$.

Určíme tedy počet $t(A)$ instrukcí vykonaných v nejhorším případě pro setřídění pole $A$ velikosti $n$.

Snadno se vidí, že nejhorším případem (vstupem, pro který se provede nejvíce instrukcí) je sestupně setříděné pole.

Přesně řečeno, je-li $A$ sestupně setříděné pole, pak pro časovou složitost $T(n)$ v nejhorším případě platí

$$T(n) = \max\{t(B) \mid B \text{ je pole velikosti } n\} = t(A).$$

Pak:

```
Insertion-Sort(A[0..n − 1])
1   for j ← 1 to n − 1
2     do t ← A[j]
3         i ← j − 1
4         while i ≥ 0 and A[i] > t
5             do A[i + 1] ← A[i]
6                 i ← i − 1
7         A[i + 1] ← t
```

- Vnější cyklus 1–7 se provede $(n-1)$-krát.
- Při provedení cyklu 1–7 pro $j$ se provede
  - 4 instrukce přiřazení (ř. 1, 2, 3, 7)
  - $j$krát počet instrukcí z cyklu 4–6, tj. $4j$ instrukcí (dvě na ř. 4, po jedné na ř. 5 a 6)
  - plus 2 instrukce ř. 4, které způsobí, že cyklus 4–6 skončí,
- nakonec 1 instrukce přiřazení, po které je v $j = n$ a cyklus 1–7 skončí.

Celkem se tedy provede
$T(n) = \sum_{j=1}^{n-1}(4+4j+2)+1 = \sum_{j=1}^{n-1} 6 + \sum_{j=1}^{n-1} 4j + 1 = 6(n-1) + 4\sum_{j=1}^{n-1} j + 1 = 6(n-1) + 4\frac{n(n-1)}{2} + 1 = 6n - 6 + 2n^2 - 2n + 1 = 2n^2 + 4n - 5.$

Tedy $T(n) = 2n^2 + 4n - 5$, kvadratická složitost.

Viděli jsme (a uvidíme), že nejdůležitější informace o složitosti je dána nejrychleji rostoucím členem, což je $2n^2$.

Dále, že konstantu 2 můžeme zanedbat. Tak detailní analýza nás nezajímá; konstanta také závisí na tom, jak vypadají elementární instrukce, viz jedna instrukce swap$(x, y)$ vs. 3 instrukce $t \leftarrow x$, $x \leftarrow y$, $y \leftarrow t$.

Tedy podstatné je, že funkce obsahuje jako nejrychleji rostoucí člen $n^2$.

Že pro nás je $2n^2 + 4n - 5$ „to samé" co $n^2$ budeme později zapisovat jako $2n^2 + 4n - 5 \in \Theta(n^2)$ (složitost je zhruba $n^2$).

Je možné se k informaci, že složitost je zhruba $n^2$ dostat rychleji?

Ano: Při analýze složitosti počítáme jen počet vykonání „nejdůležitější instrukce“, tj. té, která se vykoná nejvícekrát.

Tou je (např.) instrukce porovnání $A[i] > t$ na ř. 4. Ta se v nehorším případě vykoná

$$T(n) = \sum_{j=1}^{n-1}(j+1) = \sum_{j=1}^{n-1} j + \sum_{j=1}^{n-1} 1 = \frac{n(n-1)}{2} + n - 1 = \frac{n^2+n-2}{2}$$

krát, což je opět „to samé“ co $n^2$.

Brát v potaz jen „nejdůležitější instrukce“ usnadní analýzu složitosti.

Přitom neztratíme podstatnou informaci (jen zanedbáme konstanty a pomaleji rostoucí členy).

# Ke složitosti Insertion Sort

- nejlepší případ = vstupní pole setříděné, provede se řádově lineární počet kroků
  tj. složitost v nejlepším případě je $\Theta(n)$
- složitost v průměrném případě je (jako v nejhorším případě) kvadratická
- přes kvadratickou složitost je algoritmus velmi rychlý pro malá pole, rychlejší než pokročilejší algoritmy (např. Quick Sort, viz dále)
- používá se v kombinaci s Quick Sort: třídíme pomocí Quick Sort, ale když dosáhne velikost pole např. 10 nebo menší, použije se Insertion Sort
- existuje významné vylepšení, Shell Sort (D. L. Shell 1959):
  - porovnává vzdálené prvky, vzdálenost postupně zmenšuje
  - např. ob 10, pak ob 5, pak ob 3, pak ob 1
  - při průchodu ob $k$ provede $k$-krát Insertion Sort na $k$ prolínajících se posloupností ve vstupním poli
  - poslední průchod = normální Insertion Sort

## Cvičení

1. Upravte algoritmus Insertion Sort tak, aby výsledná posloupnost byla setříděná sestupně.

2. Upravte algoritmus Insertion Sort tak, aby se zařazované prvky (prvky vkládané do $t$) vkádaly do setříděné části zleva.

# Selection Sort

Třídění výběrem.

Idea: Najdi v $A[0..n-1]$ nejmenší prvek a vyměň ho s $A[0]$.
Najdi v $A[1..n-1]$ nejmenší prvek a vyměň ho s $A[1]$.
...
Najdi v $A[n-2..n-1]$ nejmenší prvek a vyměň ho s $A[n-2]$.

Selection-Sort($A[0..n-1]$)

**for** $j \leftarrow 0$ **to** $n-2$
  **do** $iMin \leftarrow j$
    **for** $i \leftarrow j+1$ **to** $n-1$
      **do** if $A[i] < A[iMin]$ **then** $iMin \leftarrow i$
    $t \leftarrow A[j]$; $A[j] \leftarrow A[iMin]$; $A[iMin] \leftarrow t$

**Příklad** (třídění algoritmem Selection-Sort)
zobrazujeme stav pole $A$ po skončení cyklu 3–4 (před provedením ř. 5) a po provedení ř. 5 postupně pro $j = 0, 1, \ldots, n-2$. Máme $n = 6$.
Modře je setříděná část pole. Podtržený je prvek na pozici odpovídající indexu *iMin*.

| vstup: | | | | | | |
|---|---|---|---|---|---|---|
| | 7 | 1 | 5 | 9 | 7 | 0 |

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $j = 0$, před ř. 5 | 7 | 1 | 5 | 9 | 7 | $\underline{0}$ | po ř. 5 | **0** | 1 | 5 | 9 | 7 | $\underline{7}$ |
| $j = 1$, před ř. 5 | **0** | $\underline{1}$ | 5 | 9 | 7 | 7 | po ř. 5 | **0** | $\underline{\mathbf{1}}$ | 5 | 9 | 7 | 7 |
| $j = 2$, před ř. 5 | **0** | **1** | $\underline{5}$ | 9 | 7 | 7 | po ř. 5 | **0** | **1** | $\underline{\mathbf{5}}$ | 9 | 7 | 7 |
| $j = 3$, před ř. 5 | **0** | **1** | **5** | 9 | $\underline{7}$ | 7 | po ř. 5 | **0** | **1** | **5** | **7** | $\underline{9}$ | 7 |
| $j = 4$, před ř. 5 | **0** | **1** | **5** | **7** | 9 | $\underline{7}$ | po ř. 5 | **0** | **1** | **5** | **7** | **7** | $\underline{\mathbf{9}}$ |

V posledním poli jsou všechny prvky setříděné, protože zbývající posloupnost $A[5..5]$ obsahuje jediný prvek (9), a tedy $A[5..5]$ je setříděné.

# Správnost algoritmu Selection Sort

Správnost plyne z následující vlastnosti V:

Po provedení každého cyklu 1–5 obsahuje část $A[0..j]$ prvních $j$ prvků celé setříděné posloupnosti.

To je snadno vidět.
Pro $j = 0$ je to zřejmé (tou částí je $A[0]$).

Pokud platí V pro $j = k$, pak V platí i pro $j = k + 1$, protože v cyklu pro $j = k + 1$ zůstanou prvky z $A[0..k]$ na místě a do $A[k + 1]$ se přesune nejmenší prvek z $A[k + 1..n - 1]$.

Důkaz správnosti je hotov.

# Složitost algoritmu Selection Sort

Jaká je časová složitost $T(n)$ v nejhorším případě?

... $T(n)$ je polynom stupně 2, tj. nejrychleji rostoucí člen je $c \cdot n^2$.

# Selection Sort vs Insertion Sort

- Selection Sort (S): po $k$ průchodech je na prvních $k$ pozicích v poli prvních $k$ nejmenších prvků pole
  Insertion Sort (I): po $k-1$ průchodech jsou na prvních $k$ pozicích v poli setříděné prvky, které byly na začátku v této části
- S: při zpracování prvku $k$ prochází prvky napravo od něj
  I: při zpracování prvku $k$ prochází prvky nalevo od něj
- nevýhoda S: vždy musí projít od prvku $k$ na konec pole
  I ne, při $A[k] > A[k-1]$ zpracování prvku $k$ hned skončí (např. když vstupní pole je částečně setříděno)
- v nejhorším případě je počet porovnání I stejný jako S
  v nejhorším případě je I horší: provádí více zápisů do paměti (řádově $n^2$, S jen řádově $n$)
  S je tedy lepší, když zápis je náročná operace (flash paměti)

# $O$-notace a růst funkcí

**. . . odbočka od algoritmů třídění**

Viděli jsme, že při analýze složitosti algoritmů docházíme k funkcím jako je např. $2n^2 + 4n - 5$ (viz časová složitost Insertion Sort v nejhorším případě). Řekli jsme si, že taková informace o složitosti může být „zbytečně“ přesná, tj. že může postačovat hrubší informace. V tomto případě např. může stačit vědět, že složitostí v nejhorším případě je polynom druhého stupně. Zakryjeme nepodstatné nebo méně podstatné (např. konstantu 4, členy $4n$ a $-5$) a zdůrazníme podstatné ($n^2$).

To je výhodné zejména při porovnávání algoritmů podle jejich složitosti.

Ukážeme si nyní prostředky pro takovou hrubší analýzu. Tyto prostředky patří mezi základní pojmy používané v analýze složitosti (a jsou užitečné i v jiných oblatech).

Příslušná oblast se nazývá $O$-notace („big Oh notation“).

# Proč hrubší analýzu?

Proč tedy místo $2n^2 + 4n - 5$ uvažovat jen $n^2$?

Jde totiž o to, jak rychle složitost jako funkce $T(n)$ „roste“, když se $n$ (neomezeně) zvyšuje (jak se $T$ chová pro velké velikosti vstupu), tj. jde o tzv. **asymptotický růst funkcí**.

Z tohoto pohledu je člen $-5$ nevýznamný (konstanta, nepřispívá k růstu), člen $2n$ má větší význam, ale přispívá k růstu mnohem méně než člen $2n^2$. Je tedy přirozené členy $-5$ a $4n$ zanedbat.

Proč ale zanedbat konstantu 4? Dva důvody.

1. Představuje konstantní faktor růstu, to podstatné je $n^2$.
2. Velikost konstanty závisí na (pseudo)kódu.

Příklad (algoritmus, který vymění hodnoty polí $A$ a $B$):

```
Swap-Arrays(A[0..n − 1],B[0..n − 1])
1   for i ← 0 to n − 1
2     do temp ← A[i]
3        A[i] ← B[i]
4        B[i] ← temp
```

Nyní v pseudokódu, ve kterém exsituje instrukce *swap* (ta vymění dvě hodnoty).

```
Swap-Arrays(A[0..n − 1],B[0..n − 1])
1   for i ← 0 to n − 1
2     do swap(A[i], B[i])
```

První algoritmus má časovou složitost $4n + 1$ (v cyklu proběhne 1 přiřazení hodnoty proměnné $i$ a 3 přiřazení na 2, 3, 4, nakonec zvýšení $i$ na hodnotu $n$ a ukončení cyklu).
Druhý algoritmus má časovou složitost $2n + 1$.

Je tedy rozumné konstantu zanedbat a setřít tím závislosti na konkrétním pseudokódu.

Co zanedbáním konstanty u „hlavního“ členu získáme?

- jednoduchost a přehlednost,
- vyzdvihneme podstatné, potlačíme nepodstatné.

Co ztratíme

- přesnost analýzy.

Vysoká konstanta (např. v $68n^2$) signalizuje velký počet instrukcí vykonávaných uvnitř cyklu. Rozdíl mezi $68n^2$ a $2n^2$ zpravidla není jen v jiném zápisu jednoho algoritmu (např. použitím různých pseudokódů). Tedy, algoritmus se složitostí $68n^2$ má skutečně větší složitost než ten s $2n^2$.
Zanedbáme-li konstanty, zůstane v obou případech jen $n^2$ a informace o výše popsaném rozdílu se ztratí.

# Základní pojmy pro porovnávání růstu funkcí

$f$, $g$ apod. označují funkce z množiny $\mathbb{N}$ přirozených čísel (popř. z množiny $\mathbb{R}$ reálných čísel) do množiny $\mathbb{N}$ (popř. $\mathbb{R}$).

V následujících pojmech se vyskytuje tento pohled:
$f$ je „shora ozemena“ funkcí (neroste asymptoticky rychleji než funkce) $g$, jestliže

- nezáleží na tom, v jakém vztahu jsou hodnoty $f(n)$ a $g(n)$ pro malé hodnoty $n$ (zajímá nás chování pro velké hodnoty $n$);
- počínaje jistou hodnotou $n_0$ je $f(n)$ menší nebo rovna jistému $c$-násobku hodnoty $g(n)$ (zanedbáváme konstantní faktory růstu).

Pro danou funkci $g$ (píšeme také $g(n)$) budeme definovat množiny funkcí tvaru
$M(g(n)) = \{f \mid f \text{ je funkce s vlastností } V\}$. Např.
$M(n^3) = \{f \mid \text{ pro každé } n \in \mathbb{N} \text{ je } |f(n) - n^3| \leq 2\}$
je množina všech funkcí, jejichž hodnota se od $n^3$ neliší o víc než o 2. Tedy např. $n^3 - 1 \in M(n^3)$.

# $O(g)$ . . . asymptotická horní mez (odhad)

**Definice** Pro funkci $g(n)$ je

$$O(g(n)) = \{f(n) \mid \text{existuje } c > 0 \text{ a } n_0 \in \mathbb{N} \text{ tak,} \\ \text{že pro každé } n \geq n_0 \text{ je } 0 \leq f(n) \leq cg(n)\}$$

$h \in O(g(n))$ znamená, že $h$ je jednou z funkcí z množiny $O(g(n))$.
Často se také píše $h = O(g(n))$ (zavádějící, ale má výhody).
Ilustrace:

**Příklad** Dokažme $2n^2 - 4 = O(n^2)$.

Je tedy $f(n) = 2n^2 - 4$ a $g(n) = n^2$.

Zvolme $c = 2$, $n_0 = 2$.

Pro každé $n \geq n_0$ platí $0 \leq f(n) \leq cg(n)$.

Tato podmínka totiž znamená, že pro každé $n \geq 2$ je $0 \leq 2n^2 - 4 \leq 2n^2$, což zřejmě platí. $\square$

**Příklad** Dokažme $3n^2 + 10 = O(n^2)$.

Je tedy $f(n) = 3n^2 + 10$ a $g(n) = n^2$.

Zvolme $c = 4$.

Požadovaná nerovnost pak je $3n^2 + 10 \leq 4n^2$, což je ekvivalentní $10 \leq n^2$, což platí pro $\sqrt{10} \leq n$.

Zvolíme-li tedy $n_0 = 4$, našli jsme $c$ a $n_0$ vyhovující podmínkám definice.

Lze ale zvolit i $c = 5$.

Požadovaná nerovnost pak je $3n^2 + 10 \leq 5n^2$, což je ekvivalentní $5 \leq n^2$, což platí pro $\sqrt{5} \leq n$.

Zvolíme-li tedy $n_0 = 3$, našli jsme jiná $c$ a $n_0$, prokazující $3n^2 + 10 = O(n^2)$. $\square$

**Příklad** Dokažme $2n^2 + 4n - 2 = O(n^2)$.

Je tedy $f(n) = 2n^2 + 4n - 2$ a $g(n) = n^2$.

Zvolme $c = 3$.

Požadovaná nerovnost pak je $2n^2 + 4n - 2 \leq 3n^2$, což je ekvivalentní $n^2 - 4n + 2 \geq 0$. Pro $n > 3$ tato nerovnost platí.

$c = 3$ a $n_0 = 3$ (nebo $n_0 = 4, 5, \dots$) jsou tedy hodnoty, pro které nerovnost platí, což ukazuje $2n^2 + 4n - 2 = O(n^2)$. $\square$

**Příklad** Dokažme $n^2 = O(2n^2 - 14n)$.

Je tedy $f(n) = n^2$ a $g(n) = 2n^2 - 14n$.

Zvolme $c = 1$.

Požadovaná nerovnost pak je $n^2 \leq 2n^2 - 14n$, což je ekvivalentní $n^2 - 14n \geq 0$, což platí pro každé $n > 14$.

Zvolíme-li tedy $n_0 = 14$, našli jsme $c$ a $n_0$ vyhovující podmínkám definice. $\square$

**Příklad** Dokažme $3n^2 + 10 = O(n^3)$.

Je tedy $f(n) = 3n^2 + 10$ a $g(n) = n^3$.

Zvolme $c = 1$.

Požadovaná nerovnost pak je $3n^2 + 10 \leq n^3$, což je ekvivalentní $0 \leq n^3 - 3n^2 - 10$, což platí např. pro každé $n \geq 4$. Zvolíme-li tedy $n_0 = 4$, našli jsme $c$ a $n_0$ vyhovující podmínkám definice. $\square$

**Příklad** Dokažme, že neplatí $0.5n^3 = O(20n^2)$.

Je tedy $f(n) = 0.5n^3$ a $g(n) = 20n^2$.

Zvolme libovolné $c > 0$.

Požadovaná nerovnost pak je $0.5n^3 \leq 20cn^2$, což je ekvivalentní $0.5n \leq 20c$, tj. $n \leq 40c$.

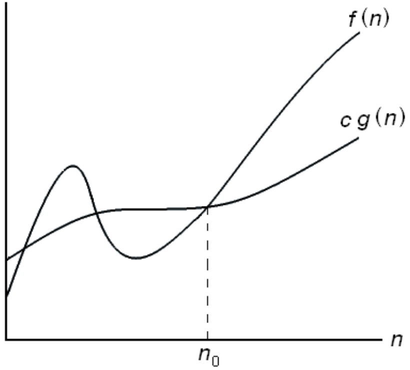
Požadovaná nerovnost tedy neplatí pro žádné $n \geq 40c$. Neexistuje tedy $n_0$ tak, aby požadovaná nerovnost platila pro každé $n \geq n_0$.

Čísla $c$ a $n_0$ požadovaná definicí tedy neexistují, proto $0.5n^3 = O(20n^2)$ neplatí. □

# $\Omega(g)$ . . . asymptotická dolní mez (odhad)

**Definice** Pro funkci $g(n)$ je

$$\Omega(g(n)) \;=\; \{f(n) \mid \text{existuje } c > 0 \text{ a } n_0 \in \mathbb{N} \text{ tak,} \\ \text{že pro každé } n \geq n_0 \text{ je } 0 \leq cg(n) \leq f(n)\}$$



**Příklad** Dokažme $n^2 = \Omega(2n^2 - 4)$.
Je tedy $f(n) = n^2$ a $g(n) = 2n^2 - 4$. Zvolme $c = 1/2$ a $n_0 = 1$. Pak pro $n \geq n_0$ platí $cg(n) = n^2 - 2 \leq n^2$. Důkaz je hotov.

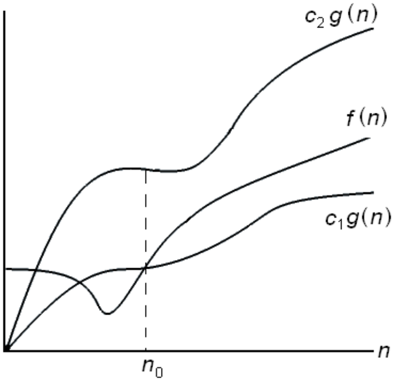
**Příklad** Dokažme $2n^2 - 4 = \Omega(n^2)$.
Je tedy $f(n) = 2n^2 - 4$ a $g(n) = n^2$. Zvolme $c = 1$ a $n_0 = 3$. Pak pro $n \geq n_0$ platí $cg(n) = n^2 \leq 2n^2 - 4$. Důkaz je hotov.

**Příklad** Dokažme, že pro každou funkci $f(n)$ a libovolnou $k > 0$ je $f(n) = \Omega(kf(n))$.
Zvolme $c = 1/k$ a $n_0 = 1$. Podmínky definice pak zřejmě platí.

# $\Theta(g)$ ... asymptotická oboustranná (těsná) mez (odhad)

**Definice** Pro funkci $g(n)$ je

$$\Theta(g(n)) = \{f(n) \mid \text{existují } c_1 > 0,\ c_2 > 0 \text{ a } n_0 \in \mathbb{N} \text{ tak, že pro každé } n \geq n_0 \text{ je } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$



Důležitý vztah:
**Věta** $f(n) = \Theta(g(n))$ právě když $f(n) = O(g(n))$ a $f(n) = \Omega(g(n))$.

**Důkaz**
1. $f(n) = \Theta(g(n))$ implikuje $f(n) = O(g(n))$ a $f(n) = \Omega(g(n))$:
Jednoduše přímo z definice.

2. $f(n) = O(g(n))$ a $f(n) = \Omega(g(n))$ implikuje $f(n) = \Theta(g(n))$:

$f(n) = O(g(n))$ znamená, že existuje $c_2 > 0$ a $n_{2,0}$ tak, že pro $n \geq n_{2,0}$ je $f(n) \leq c_2 g(n)$.

$f(n) = \Omega(g(n))$, tedy existuje $c_1 > 0$ a $n_{1,0}$ tak, že pro $n \geq n_{1,0}$ je $0 \leq c_1 g(n) \leq f(n)$.

Položíme-li tedy $n_0 = \max(n_{1,0}, n_{2,0})$, pak tedy existují $c_1, c_2 > 0$ a $n_0$ tak, že pro každé $n \geq n_0$ je $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$, což znamená $f(n) = \Theta(g(n))$.

Příklad použití Věty:
Víme, že $2n^2 - 4 = O(n^2)$ a $2n^2 - 4 = \Omega(n^2)$. Z Věty tedy plyne $2n^2 - 4 = \Theta(n^2)$.

# $o(g)$ ... asymptotická ostrá (netěsná) horní mez (odhad)

As. horní odhad nemusí být těsný. Např. $3n = O(n^2)$ není těsný, zatímco $3n^2 = O(n^2)$ je těsný (protože je i $n^2 = O(3n^2)$). Netěsné horní odhady se značí $o(g)$ a $f = o(g)$ pak znamená, že $f$ je asymptoticky ostře menší než $g$.

**Definice** Pro funkci $g(n)$ je

$$\begin{aligned} o(g(n)) \ = \ \{f(n) \mid \ & \text{pro každou } c > 0 \text{ existuje } n_0 > 0 \text{ tak,} \\ & \text{že pro každé } n \geq n_0 \text{ je } 0 \leq f(n) < cg(n)\} \end{aligned}$$

Např. $3n = o(n^2)$, ale $3n^2 \neq o(n^2)$. Proč?

$3n = o(n^2)$: Nechť $c > 0$ je libovolná. Pak pro každé $n > 3/c + 1$ je zřejmě $3n < cn^2$. Tedy stačí zvolit $n_0 = \lceil 3/c + 1 \rceil$.

$3n^2 \neq o(n^2)$: Je $f(n) = 3n^2$, $g(n) = n^2$. Pro $c = 3$ zřejmě neexistuje $n_0$ tak, že pro každé $n \geq n_0$ platí $f(n) < cg(n)$.

# $\omega(g)$ ... asymptotická ostrá dolní mez (odhad)

**Definice** Pro funkci $g(n)$ je

$$\omega(g(n)) = \{f(n) \mid \text{ pro každou } c > 0 \text{ existuje } n_0 > 0 \text{ tak,} \\ \text{že pro každé } n \geq n_0 \text{ je } 0 \leq cg(n) < f(n)\}$$

Např. $3n^2 = \omega(n)$, ale $3n^2 \neq \omega(n^2)$. Proč?

Zdůvodnění je podobné jako v příkladě uvedeném u $o(g(n))$.

# Asymptotické značení v rovnostech a nerovnostech

Asympt. značení se s výhodou používá v aritmetických výrazech.

Víme, že $3n^2 + 5 = \Theta(n^2)$ chápeme jako $3n^2 + 5 \in \Theta(n^2)$.

Výraz jako $3n^2 + \Theta(n)$ chápeme tak, že jde o některý z výrazů $3n^2 + f(n)$, kde $f(n)$ je některou funkcí z množiny $\Theta(n)$.

Např. $3n^2 + 4n - 3 = 3n^2 + \Theta(n)$ znamená, že existuje funkce $f(n) \in \Theta(n)$, pro kterou $3n^2 + 4n - 3 = 3n^2 + f(n)$. Tato rovnost platí (pro $f(n) = 4n - 3$).

Podobně chápeme nerovnosti se symboly asymptotické notace na pravé straně.

Výrazy se symboly asymptotické notace na obou stranách rovnosti, jako např. $n^2 + \Theta(n) = \Theta(n^2)$ chápeme takto:

Pro každou funkci $f(n) \in \Theta(n)$ existuje funkce $g(n) \in \Theta(n^2)$, pro kterou je $n^2 + f(n) = g(n)$.

Platí $n^2 + \Theta(n) = \Theta(n^2)$?

Pro procvičení uveďte rovnosti a nerovnosti s výskyty symbolů asymptotické notace, které jsou pravdivé, i ty, které jsou nepravdivé.

# Základní pravidla

**Věta** (tranzitivita odhadů)
Pokud $f = O(g)$ a $g = O(h)$, pak $f = O(h)$.
Pokud $f = \Omega(g)$ a $g = \Omega(h)$, pak $f = \Omega(h)$.
Pokud $f = \Theta(g)$ a $g = \Theta(h)$, pak $f = \Theta(h)$.
Pokud $f = o(g)$ a $g = o(h)$, pak $f = o(h)$.
Pokud $f = \omega(g)$ a $g = \omega(h)$, pak $f = \omega(h)$.

**Důkaz**
Dokažme první tvrzení.
$f(n) = O(g(n))$ znamená, že existuje $c_1 > 0$ a $n_{1,0}$ tak, že pro $n \geq n_{1,0}$ je $f(n) \leq c_1 g(n)$.
$g(n) = O(h(n))$ znamená, že existuje $c_2 > 0$ a $n_{2,0}$ tak, že pro $n \geq n_{2,0}$ je $g(n) \leq c_2 h(n)$.
Položme $c = c_1 c_2$ a $n_0 = \max(n_{1,0}, n_{2,0})$. Pak pro $n \geq n_0$ platí $f(n) \leq c_1 g(n) \leq c_1 c_2 h(n) = ch(n)$, tedy $f(n) = O(h(n))$.

**Věta** (reflexivita odhadů)
$f = O(f)$.
$f = \Omega(f)$.
$f = \Theta(f)$.

**Důkaz**
$f = O(f)$: Stačí položit $c = 1$ a $n_0 = 1$. Stejně pro $f = \Omega(f)$ a $f = \Theta(f)$.

**Věta** (symetrie odhadů)
$f = \Theta(g)$ právě když $g = \Theta(f)$.

**Důkaz**
Nechť $f = \Theta(g)$. Pak existují $c_1, c_2 > 0$ a $n_0$ tak, že pro $n \geq n_0$ je
$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$. Položme $c_1' = 1/c_1$ a $c_2' = 1/c_2$.
Pak máme $c_1', c_2' > 0$ a $n_0$ taková, že pro $n \geq n_0$ platí
$0 \leq c_2' f(n) \leq g(n) \leq c_1' f(n)$, tedy $g = \Theta(f)$.

**Věta** (symetrie horních a dolních odhadů)
$f = O(g)$ právě když $g = \Omega(f)$.
$f = o(g)$ právě když $g = \omega(f)$.

**Důkaz**
Dokažme první tvrzení.

$f(n) = O(g(n))$ znamená, že existuje $c > 0$ a $n_0$ tak, že pro $n \geq n_0$ je $0 \leq f(n) \leq cg(n)$.
Položme $c' = 1/c$. Pak $c' > 0$ a pro $n \geq n_0$ je $0 \leq c'f(n) \leq g(n)$, tedy $g = \Omega(f)$.

Podobně se dokáže, že z $g = \Omega(f)$ plyne $f(n) = O(g(n))$.

## Příklady použití výše uvedených základních vztahů

... na cvičení

**Další příklady:**

Ukažte, že pro funkci $h(n) = \max(f(n), g(n))$ platí $h(n) = \Theta(f(n) + g(n))$.

Ukažte, že $o(f(n)) \cap \omega(f(n)) = \emptyset$.

# Asymptotické značení v popisu složitosti

Má tedy smysl říct, že čas. složitost v nejhoším případě algoritmu A je $O(f(n))$.
Význam: pro časovou složitost $T(n)$ algoritmu A je $T(n) = O(f(n))$.

Víme tedy např., že časová složitost v nejhorším případě algoritmů Insertion-Sort, Selection-Sort, Bubble-Sort (poslední uvidíme za chvíli) je $O(n^2)$.

Víme ale dokonce to, že časová složitost v nejhorším případě algoritmů Insertion-Sort, Selection-Sort, Bubble-Sort je $\Theta(n^2)$. Totiž, např. pro Insertion-Sort jsme odvodili, že $T(n) = 4n^2 + 2n - 5$ a víme, že $4n^2 + 2n - 5 = \Theta(n^2)$.

Jak jsme viděli u algoritmu Insertion-Sort, zjistit, že to je $T(n) = O(n^2)$ je snadnější než určit $T(n)$ přesně.

**Příklad** Určete nějakou $f(n)$ tak, že časová složitost (v nejhorším případě) následujícího algoritmu, který sečte všechny možné násobky čísel z pole $A$ s čísly z pole $B$, je $T(n) = O(f(n))$.

Sum-Of-Products($A[0..n-1]$,$B[0..n-1]$)

$sum \leftarrow 0$<br>
**for** $i \leftarrow 0$ **to** $n-1$<br>
    **do for** $j \leftarrow 0$ **to** $n-1$<br>
        **do** $sum \leftarrow sum + A[i] * B[j]$<br>
**return** $sum$

Dva vnořené cykly, každý se provede $n$-krát. Je tedy hned vidět, že $T(n) = O(n^2)$.
Je také ovšem hned vidět, že $T(n) = \Theta(n^2)$, což je přesnější odhad.

Podobně má smysl říct, že časová složitost v nejlepším případě algoritmu A je $\Omega(f(n))$. Znamená to, že pro časovou složitost (v nejlepším případě) $T(n)$ algoritmu A platí $T(n) = \Omega(f(n))$.

Vysvětlete, proč časová složitost algoritmu Insertion-Sort v nejlepším případě je $\Omega(n)$.
Je časová složitost algoritmu Insertion-Sort v nejlepším případě $\Theta(n)$?

Proč neplatí, že časová složitost algoritmu Insertion-Sort v nejlepším případě je $\Omega(n^2)$?

Vysvětlete, proč časová složitost algoritmu Selection-Sort v nejlepším případě je $\Omega(n^2)$.

# Asymptotické značení a polynomy

Nechť $f(n) = \sum_{i=0}^{d} a_i n^i$ a $a_d > 0$ (polynom stupně $d$).
Pak platí:

- pro $k \geq d$ je $p(n) = O(n^k)$,
- pro $k \leq d$ je $p(n) = \Omega(n^k)$,
- pro $k = d$ je $p(n) = \Theta(n^k)$,
- pro $k > d$ je $p(n) = o(n^k)$,
- pro $k < d$ je $p(n) = \omega(n^k)$.

# Bubble Sort

Bublinkové třídění.

Nejmenší prvek „probublá“ vlevo, pak „probublá“ druhý nejmenší, atd.

Bubble-Sort($A[0..n-1]$)

**for** $j \leftarrow 0$ **to** $n-2$
  **do for** $i \leftarrow n-1$ **downto** $j+1$
    **do if** $A[i] < A[i-1]$
      **then** $temp \leftarrow A[i]$; $A[i] \leftarrow A[i-1]$; $A[i-1] \leftarrow temp$

**Příklad** (třídění algoritmem Bubble-Sort)

vstup:

| | | | | | |
|---|---|---|---|---|---|
| 7 | 1 | 5 | 9 | 7 | 0 |

Zobrazujeme stav pole $A$ po provedení řádku 4 pro jednotlivé hodnoty $j$ a $i = j + 1$ (tj. po provedení vnitřního cyklu); pro $j = 0$ také pro jednotlivé hodnoty $i$. Je $n = 6$. Modře je setříděná část pole.

| | | | | | | |
|---|---|---|---|---|---|---|
| $j = 0$, $i = 5$: | 7 | 1 | 5 | 9 | 0 | 7 |
| $j = 0$, $i = 4$: | 7 | 1 | 5 | 0 | 9 | 7 |
| $j = 0$, $i = 3$: | 7 | 1 | 0 | 5 | 9 | 7 |
| $j = 0$, $i = 2$: | 7 | 0 | 1 | 5 | 9 | 7 |
| $j = 0$, $i = 1$: | **0** | 7 | 1 | 5 | 9 | 7 |
| $j = 1$, $i = 2$: | **0** | **1** | 7 | 5 | 7 | 9 |
| $j = 2$, $i = 3$: | **0** | **1** | **5** | 7 | 7 | 9 |
| $j = 3$, $i = 4$: | **0** | **1** | **5** | **7** | 7 | 9 |
| $j = 4$, $i = 5$: | **0** | **1** | **5** | **7** | **7** | **9** |

# Správnost algoritmu Bubble Sort

Proč je Bubble-Sort správný?

Zdůvodněte sami.
Návod: Po provedení každého cyklu pro hodnotu $j$ je část $A[0..j]$ setříděná a obsahuje prvky $\leq$ prvky z $A[j+1..n-1]$.

# Složitost algoritmu Bubble Sort

Časová složitost $T(n)$ v nejhoršímm případě je $\Theta(n^2)$.

Zdůvodněte. Nejdřív bez určení $T(n)$.

Potom určete $T(n)$.

# Varianty algoritmu Bubble Sort

**Optimalizace vynecháním některých průchodů**

Pozorování: Platí nejen to, že po každém průchodu cyklu pro $j$ (tj. po provedení vnitřního cyklu pro $j$) je setříděná část $A[0..j]$, ale dokonce to, že část pole $A$ až po místo, kde došlo k poslední výměně na ř. 4, je setříděná a tvoří počáteční část setříděného pole $A$, které vznikne po skončení výpočtu.

Přesněji, došlo-li k poslední výměně ve vnitřním cyklu pro hodnotu $i$, je setříděná část $A[0..i-1]$ a tato část tvoří počáteční část setříděného pole $A$, které vznikne po skončení výpočtu ($A[0..i]$ je také setříděná, ale nemusí tvořit počáteční část setříděného pole $A$).

V dalším průchodu můžeme tedy přeskočit provádění vnějšího cyklu pro další hodnoty $j$ a můžeme pokračovat pro $j = i$.
Nedošlo-li k žádné výměně, je $A[0..n-1]$ celé setříděné, a lze tedy výpočet ukončit (přejít na $j = n-1$).
(V původní verzi se nic nepřeskakuje a pokračuje se pro $j+1$.)

Tím dojde k urychlení. Modifikovaný algoritmus lze snadno popsat a implementovat (proveďte).

Porovnejte chování původního Bubble Sort a modifikovaného algoritmu na vstupu

| | | | | | |
|---|---|---|---|---|---|
| 7 | 1 | 5 | 9 | 7 | 0 |

.

Modifikovaný algoritmus je efektivnější než původní verze. Např. posloupnost, která je již setříděná, zpracuje v jednom průchodu (po průchodu pro $j = 0$ se "přeskočí" na $j = n - 1$, a dojde tedy k ukončení).

Přesněji: Oba algoritmy mají v nejhorším případě složitost $\Theta(n^2)$. V modifikované verzi se totiž v nejhorším případě (vstupní pole je setříděné sestupně), nic nepřeskakuje.
V nejlepším případě (vstupní pole je setříděné vzestupně) má původní algoritmus složitost $\Theta(n^2)$, ale modifikovaný $\Theta(n)$.

**Coctail Sort** (také Bidirectional Buble Sort, Shaker Sort)
Pozorování: Velmi malé prvky umístěné (před tříděním) v poli vpravo se dostanou na správné místo rychle (nejmenší prvek se tam dostane při prvním průchodu pro $j = 0$). (Těmto "rychlým" prvkům se říká zajíci.)

Ale: Velké prvky umístěné v poli vlevo se dostávají na správné místo pomalu. Musí totiž být "předběhnuty" menšími prvky zleva. Pokud je např. nějvětší prvek (před tříděním) v $A[0]$, dostane se na správné místo až při posledním provedení ř. 4 (tj. pro $j = n - 2$ a $i = n - 1$). (Těmto "pomalým" prvkům se říká želvy.)

Coctail Sort pracuje tak, že opakovaně provádí dvojice operací:
1. V nesetříděné části $A[p..q]$ pole nech "probublat nejmenší" prvek zprava doleva na pozici $A[p]$.
2. V nesetříděné části $A[p+1..q]$ pole nech "probublat největší" prvek zleva doprava na pozici $A[q]$.
Pak pokračuj s 1. pro $A[p+1..q-1]$, atd. než je pole celé setříděné (tj. poslední provedení 1. nebo 2. se provede pro část tvaru $A[k..k+1]$).

Modifikovaný algoritmus lze snadno popsat a implementovat (proveďte).

Porovnejte chování původního Bubble Sort a Coctail Sort na vstupu

| | | | | | |
|---|---|---|---|---|---|
| 7 | 1 | 2 | 4 | 5 | 6 |

.

Coctail Sort lze vylepšit optimalizací vynecháním některých průchodů, jako jsme to provedli v případě původního Bubble Sort.

Vylepšený algoritmus lze snadno popsat a implementovat (proveďte).

# Quick Sort

Česky také „Quick Sort“.

Charles Anthony Richard Hoare, 1961. Algorithm 64: Quicksort. Communications of ACM. 4 (7): 321

Algoritmus typu „rozděl a panuj“ (divide and conquer).

Přirozeně se nabízí rekurzívní formulace a implementace algoritmu.

Pro setřídění části pole $A[p..r]$ algoritmus Quick-Sort postupuje následovně.

Fáze "rozděl": Zvolí $q$ (prvek $A[q]$ se nazývá pivot) a přemístí prvky pole tak, že všechny prvky v části $A[p..q-1]$ jsou menší nebo rovny $A[q]$ a všechny prvky v části $A[q+1..r]$ jsou větší než $A[q]$.

Volbu pivota a přemístění provede funkce Partition.

Fáze "panuj": Setřídí části $A[p..q-1]$ a $A[q+1..r]$. Setřídí je opět algoritmem Quick-Sort (nejdřív jednu, pak druhou).

Jedná se o tzv. rekurzívní algoritmus. Totiž, při provádění algoritmu Quick-Sort (ve fázi "panuj") se "volá" sám Quick-Sort. To však nevede k zacyklení (k nekonečné smyčce), protože algoritmus je "'volán" pro stále kratší části pole $A$, a při volání pro část tvaru $A[p..p]$ se provádění okamžitě ukončí ($A[p..p]$ je totiž setříděná). To uvidíme.

```
Quick-Sort(A, p, r)
  if p < r
     then q ← Partition(A, p, r)
          Quick-Sort(A, p, q − 1)
          Quick-Sort(A, q + 1, r)
```

Pro $p < r$ provede Quick-Sort($A$, $p$, $r$) setřídění části $A[p..r]$.

Tedy Quick-Sort($A$, 0, $n-1$) setřídí celé pole $A[0..n-1]$.

Přitom Partition($A, p, r$) provede výše popsané přemístění prvků pole $A$ a vrátí index $q$ pivota.

Partition($A$,$p$,$r$)

1 $x \leftarrow A[r]$
2 $i \leftarrow p-1$
3 **for** $j \leftarrow p$ **to** $r-1$
4     **do if** $A[j] \leq x$
5         **then** $i \leftarrow i+1$
6             swap($A[i], A[j]$)
7 swap($A[i+1]$,$A[r]$)
8 **return** $i+1$

ř. 1: pivotem je hodnota $A[r]$, ta se uloží do $x$

Na pole se lze dívat následovně. Při vstupu do cyklu 3–6 platí:

(a) Pro $p \leq k \leq i$ je $A[k] \leq x$.
($A[p..i]$ je konstruovanou částí s prvky $\leq x$.)

(b) Pro $i+1 \leq k \leq j-1$ je $A[k] > x$.
($A[i+1..j-1]$ je konstruovanou částí s prvky $> x$.)

(c) Část $A[j..r-1]$ je dosud nezpracovanou částí.

(d) $A[r] = x$.

**Příklad** (Partition(A,0,7))

Modře je $A[p..i]$, červeně je $A[i+1..j-1]$, **3** je pivot, $p=0$, $r=7$.

vstup:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 9 | 8 | 0 | 2 | 6 | 7 | 3 |

průběh (stavy pole po provedení ř. uvedeného za polem při uvedených $i,j$)

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 9 | 8 | 0 | 2 | 6 | 7 | **3** | ř. 3, $p=0$, $r=7$, $i=-1$, $j=0$ |
| **1** | 9 | 8 | 0 | 2 | 6 | 7 | **3** | ř. 6, $i=0$, $j=0$ (ř. 5, 6 se provedly) |
| **1** | **9** | 8 | 0 | 2 | 6 | 7 | **3** | ř. 6, $i=0$, $j=1$ (ř. 5, 6 se neprovedly) |
| **1** | **9** | **8** | 0 | 2 | 6 | 7 | **3** | ř. 6, $i=0$, $j=2$ (ř. 5, 6 se neprovedly) |
| **1** | **0** | **8** | **9** | 2 | 6 | 7 | **3** | ř. 6, $i=1$, $j=3$ (ř. 5, 6 se provedly) |
| **1** | **0** | **2** | **9** | **8** | 6 | 7 | **3** | ř. 6, $i=2$, $j=4$ (ř. 5, 6 se provedly) |
| **1** | **0** | **2** | **9** | **8** | **6** | 7 | **3** | ř. 6, $i=2$, $j=5$ (ř. 5, 6 se neprovedly) |
| **1** | **0** | **2** | **9** | **8** | **6** | **7** | **3** | ř. 6, $i=2$, $j=6$ (ř. 5, 6 se neprovedly) |
| **1** | **0** | **2** | **3** | **8** | **6** | **7** | **9** | ř. 7, $i=2$, výměna $A[3]$ a $A[7]$ |

# Správnost algoritmu Quick Sort

Z rekurzívního popisu algoritmu pseudokódem je zřejmé, že algoritmus je správný, pokud je správná procedura Partition.

Pro zdůvodnění její správnosti použijeme výše uvedené vlastnosti (a)–(d).

(a)–(d) platí při prvním vstupu do cyklu 3–6.

Platí-li (a)–(d) při vstupu do průchodu cyklu 3–6 pro hodnotu $j$, platí i po provedení tohoto průchodu.

Po skončení cyklu 3–6 je tedy $A[p..r-1]$ srovnaná takto:
$A[p], \ldots, A[i] \leq A[r] < A[i+1], \ldots, A[r-1] > A[r]$, pivot je $A[r]$.

Nakonec se provede výměna na ř. 7 ($A[r]$ a $A[i+1]$) a pole je správně srovnané.

# Složitost algoritmu Quick Sort

Časová složitost:

v nejhorším případě je $\Theta(n^2)$,

v průměrném případě je $\Theta(n \log n)$.
Proto je Quick Sort efektivním algoritmem.

(Zdůvodníme a rozvedeme později.)

# Neformální úvahy o složitosti Quick Sort

Trvání výpočtu Quick Sort pro dané vstupní pole závisí na tom, jak vyvážená jsou pole vytvářená funkcí Partition, tj. zda levý a pravý úsek mají (přibližně) stejnou velikost.

Viděli jsme, že pro vstupní pole

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 9 | 8 | 0 | 2 | 6 | 7 | 3 |

vznikne po provedení Partition pole

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **1** | **0** | **2** | **3** | **8** | **6** | **7** | **9** |

, které je vyvážené.

Pro vstupní pole

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | 9 | 8 | 0 | 2 | 6 | 7 | 1 |

ale vznikne po provedení Partition pole

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **0** | **1** | **8** | **3** | **2** | **6** | **7** | **9** |

, které není vyvážené.

Nejhorším případem je např.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | 9 | 8 | 1 | 2 | 6 | 7 | 0 |

.

Po provedení Partition totiž vznikne pole

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **0** | **9** | **8** | **1** | **2** | **6** | **7** | **3** |

, které je maximálně nevyvážené (prázdná levá část, podobně by mohla vzniknout prázdná pravá část).

**Nejhorší případ**

Pokud vznikne maximálně nevyvážené pole při každém volání Partition (např. když je vstupní pole vzestupně nebo setupně setříděné), lze trvání výpočtu $t(A)$ algoritmem Quick Sort pro pole $A$ velikosti $n$ odvodit následovně. Předpokládejme, že vstupní pole $A[p..r]$ vypadá takto: $A = \langle 2, 3, 4, 5, 6, 1 \rangle$ (vzestupně, ale poslední prvek je nejmenší; říkejme, že $A$ je typu B).

Pro $p \geq r$: Při vstupu do Quick-Sort je proveden test na ř. 1. Protože $p \geq r$, provádění (aktuálního volání) Quick-Sort se ukončí. Byla provedena 1 instrukce (ř. 1). Protože $1 = \Theta(1)$ (podobně $c = \Theta(1)$ pro každou konstantu $c$), lze říct, že bylo provedeno $\Theta(1)$ kroků.

Pro $p < r$: Provede se test na ř. 1. Pak se provede Partition$(A, p, r)$. Snadno se vidí, že toto provedení trvá $\Theta(r - p + 1)$ kroků ($r - p + 1$ je počet prvků pole). Po provedení Partition je levý úsek prázdný, $A[p]$ obsahuje pivot (nejmenší prvek $A[p..r]$) a pravá část $A[p + 1..r]$ je typu B. Na ř. 3 se provede Quick-Sort$(A, p, p - 1)$, trvání je dle případu "Pro $p \geq r$" $\Theta(1)$. Na ř. 4 se provede Quick-Sort$(A, p + 1, r)$, trvání je $t(A[p + 1..r])$.

Trvání $t(A[p..r])$ algoritmu Quick-Sort pro $A[p..r]$ je tedy

$$t(A[p..r]) = 1 + \Theta(r - p + 1) + \Theta(1) + t(A[p + 1..r]).$$

Pro $A[0..n-1]$ ($p = 0$, $r = n - 1$) tedy dostaneme

$$t(A[0..n-1]) = 1 + \Theta(n) + \Theta(1) + t(A[1..n-1]) = \Theta(n) + t(A[1..n-1]) + \Theta(1),$$

protože $1 + \Theta(1) = \Theta(1)$. Podobně, $\Theta(1) + \Theta(1) = \Theta(1)$ (ověřte; uvědomte si, že to říká: součet dvou konstantních funkcí je konstantní funkce). Tedy

$$\begin{aligned}
&t(A[0..n-1]) = \Theta(n) + t(A[1..n-1]) + \Theta(1) = \\
&\Theta(n) + \Theta(n-1) + t(A[2..n-1]) + \Theta(1) = \\
&\Theta(n) + \Theta(n-1) + \Theta(n-2) + t(A[3..n-1]) + \Theta(1) = \cdots = \\
&\Theta(n) + \Theta(n-1) + \Theta(n-2) + \cdots + \Theta(2) + \Theta(1) + \Theta(1) = \\
&\Theta(n) + \Theta(n-1) + \Theta(n-2) + \cdots + \Theta(2) + \Theta(1).
\end{aligned}$$

Platí

$$\Theta(n) + \Theta(n-1) + \Theta(n-2) + \cdots + \Theta(2) + \Theta(1) = \Theta(n^2).$$

Ověřte to. Využijte při tom $n + (n-1) + \cdots + 2 + 1 = \frac{(n+1)\cdot n}{2} = \Theta(n^2)$, což víme. (Návod: dokažte např. matematickou indukcí přes $n$.)

Je tedy

$$t(A[0..n-1]) = \Theta(n^2).$$

Protože pole $A$ typu B je nejhorším případem, platí pro časovou složitost $T(n)$ Quick-Sort v nejhorším případě

$$T(n) = \Theta(n^2).$$

Připomeňme si, že u Insertion-Sort pro sestupně setříděnou posloupnost je $t(A[0..n-1]) = \Theta(n)$ (nejlepší případ pro Insertion-Sort).

## Nejlepší případ

Když vznikne při každém volání Partition maximálně vyvážené pole. Tj. když při každém volání Partition na pole s $k$ prvky vznikne pole, jehož jedna část má $\lfloor \frac{k}{2} \rfloor$ a druhá $\lceil \frac{k}{2} \rceil - 1$ prvků.

Je-li $A[0..n-1]$ takové pole, je časová složitost Quick-Sort v nejlepším případě $T(n) = t(A[0..n-1])$.

Platí tedy

$$T(n) \leq 2T(n/2) + \Theta(n)$$

(porovnání na ř. 1: $\Theta(1)$, Partition na ř. 2: $\Theta(n)$, dvě volání Quick-Sort na ř. 3, 4: jedno má trvání $T(\lfloor \frac{n}{2} \rfloor) \leq T(\frac{n}{2})$, druhé $T(\lceil \frac{n}{2} \rceil - 1) \leq T(\frac{n}{2})$).

Lze ukázat (pomocí tzv. Master Theorem, později), že pak

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

## Intuice pro průměrný případ

Nejhorší případ: $T(n) = \Theta(n^2)$ ... max. nevyvážené pole.
Nejlepší případ: $T(n) = \Theta(n \lg n)$ ... max. vyvážené pole.

Pro složitost v průměrném případě je zásadní fakt, že trvání Quick-Sort je v případě, kdy Partition produkuje pole, která nejsou ani max. vyvážená, ani max. nevyvážená, asymptoticky stejné jako v případě, kdy Partition produkuje max. vyvážená pole!

Příklad. Předpokládejme, že Partition produkuje pole rozdělené v poměru $9:1$ (např. 11prvkové pole, levá část má 9 prvků, pravá 1). Podobně jako v případě max. vyváženého pole získáme pro

$$t(n) \leq t(\frac{9}{10}n) + t(\frac{1}{10}n) + \Theta(n).$$

Zde $t(n)$ označuje trvání Quick-Sort za uvedených podmínek (tj. dělení $9:1$). Lze ukázat (Master Theorem), že řešením je opět funkce $t(n) = \Theta(n \lg n)$.

# Merge Sort

Třídění sléváním (slučováním, angl. merge). J. von Neumann, 1945.

Algoritmus typu "rozděl a panuj": Sertřiď levou polovinu pole, setřiď pravou polovinu pole, "slij" obě poloviny pole.

```
Merge-Sort(A, p, r)
if p < r
   then q ← ⌊(p + r)/2⌋
        Merge-Sort(A, p, q)
        Merge-Sort(A, q + 1, r)
        Merge(A, p, q, r)
```

Merge-Sort($A$, $p$, $q$) setřídí $A[p..q]$,
Merge-Sort($A$, $q+1$, $r$) setřídí $A[q+1..r]$,
Merge($A$, $p$, $q$, $r$) vytvoří "slitím" ze setříděných částí $A[p..q]$ a $A[q+1..r]$ setříděné pole $A[p..r]$.

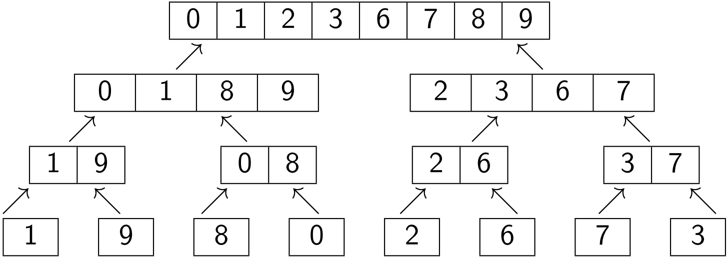
Merge-Sort($A$, 0, $n-1$) setřídí $A[0.n-1]$.

**Příklad** (Merge-Sort(A,0,7))

vstup:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 9 | 8 | 0 | 2 | 6 | 7 | 3 |

Třídění probíhá takto:




Na posledním ř. jsou výsledky Merge-Sort$(A, 0, 0)$, . . . , Merge-Sort$(A, 7, 7)$,

Na předchozím ř. jsou výsledky Merge-Sort$(A, 0, 1)$, . . . , Merge-Sort$(A, 6, 7)$,

Na prvním ř. je výsledek Merge-Sort$(A, 0, 7)$, tedy setříděné pole.

Slévání prováděné funkcí Merge je vyznačeno dvojicemi šipek, které vedou od slévaných částí pole ke slitému úseku. Např. slitím $\boxed{2 \mid 6}$ a $\boxed{3 \mid 7}$ vznikne $\boxed{2 \mid 3 \mid 6 \mid 7}$.

Merge($A$, $p$, $q$, $r$)

```
n1 ← q − p + 1
n2 ← r − q
vytvoř nová pole L[0..n1] a R[0..n2]
for i ← 0 to n1 − 1
    do L[i] ← A[p + i]
for j ← 0 to n2 − 1
    do R[j] ← A[q + 1 + j]
L[n1] ← ∞
R[n2] ← ∞
i ← 0
j ← 0
for k ← p to r
    do if L[i] ≤ R[j]
          then A[k] ← L[i]
               i ← i + 1
          else A[k] ← R[j]
               j ← j + 1
```

**Poznámky k** Merge($A$,$p$, $q$, $r$)

Merge využívá kromě pole $A[p..r]$ další paměť, totiž pomocná pole $L$ a $R$ o celkové velikosti $r - p + 3$ (velikost $A + 2$). Merge, a tedy i Merge-Sort tedy netřídí "na místě" (in place). To je rozdíl oproti všem předchozím algoritmům i oproti Heap Sort (příště).

Merge je přímočarý algoritmus. Slévání probíhá tak, že se nejdříve slévané části zkopírují do $L$ a $R$ a pak se v 12–17 vždy vezme menší z dosud nezpracovaných prvků $L[i]$ a $R[j]$ polí $L$ a $R$ a vloží se na další pozici pole $A$, tj. na $A[k]$.

Používá techniku "zarážky". Zarážkou je v tomto případě hodnota $\infty$, která je větší než každá jiná hodnota.

# Správnost algoritmu Merge Sort

Z pseudokódu je jasné, že Merge Sort je správný, pokud má Merge následující vlastnost:

Jsou-li části $A[p..q]$ a $A[q+1..r]$ před provedením funkce Merge vzestupně uspořádané, je po ukončení funkce Merge vzestupně uspořádané celé pole $A[p..r]$.
Snadno se vidí, že tato vlastnost platí.

# Složitost algoritmu Merge Sort

Časová složitost $T(n)$ v nejhorším případě:

Vstup $A[0..n-1]$.
Pro jednoduchost předpokládejme, že počet prvků pole je mocinou dvojky, tj. že $n = 2^k$ pro celé číslo $k$. Později uvidíme, že tento předpoklad neovlivní výsledek naší analýzy složitosti.

Uvědomme si nejprve, že složitost Merge (vstupem je $A, p, q, r$, velikostí je $n = r - p + 1$) je $\Theta(n)$. Merge totiž provede dva cykly 4–5 a 6–7, jejichž délka je v součtu $n$, cyklus 12–17 délky $n$, ř. 3 s trváním max. $n + 2$ a několik dalších instrukcí (každá s trváním nezávislým na $n$).

Z pseudokódu plyne, že platí

$$T(n) = \Theta(1) \text{ pro } n = 1, \quad T(n) = 2T(n/2) + \Theta(n) \text{ pro } n > 1. \quad (*)$$

Pro $n = 1$ je to zřejmé (konstantní čas, protože dojde jen k porovnání na ř. 1). Pro $n > 1$: Dojde k dvěma voláním Merge-Sort, každé pro vstup velikosti $n/2$, proto člen $2T(n/2)$, poté Merge s trváním $\Theta(n)$.

Z rovnice (*) plyne, že $T(n) = \Theta(n \lg n)$.

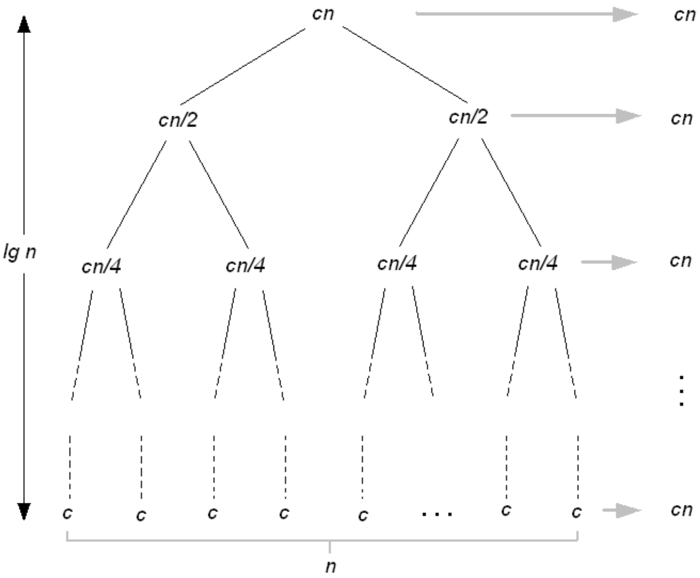
To lze odvodit z tzv. Master Theorem, se kterým se seznámíme později.

Lze to však také odvodit přímo, což uděláme. Přepíšeme (*) tak, že výrazy $\Theta(1)$ a $\Theta(n)$ nahradíme $c_1$ a $c_2 n$ a pro zjednodušení vezmeme $c = \max\{c_1, c_2\}$ (i bez zjednodušení dojdeme k $T(n) = \Theta(n \lg n)$, promyslete):

$$T(n) = c \text{ pro } n = 1, \quad T(n) = 2T(n/2) + cn \text{ pro } n > 1.$$

Hodnotu $T(n)$ pak můžeme znázornit pomocí binárního stromu.

[OBRÁZEK NA PŘEDNÁŠCE A DALŠÍ SLAJD]


cn
cn
cn/2
cn/2
cn
lg n
cn/4
cn/4
cn/4
cn/4
cn
c
c
c
c
c
. . .
c
c
cn
n

Hodnota $T(n)$ je rovna součtu hodnot ve všech uzlech stromu. V první úrovni je jeden uzel s hodnotou $cn$. Součet hodnot uzlů v první úrovni je tedy $cn$. Ve druhé úrovni jsou dva uzly s hodnotami $cn/2$. Součet hodnot uzlů v druhé úrovni je tedy $cn$. Ve třetí úrovni jsou čtyři uzly s hodnotami $cn/4$. Součet hodnot uzlů v druhé úrovni je tedy $cn$. Atd. V poslední úrovni je $n$ uzlů s hodnotami $c$. Součet hodnot uzlů v poslední úrovni je tedy $cn$.

Součet hodnot uzlů v každé úrovni je tedy $cn$. Protože strom má $\lg n + 1$ úrovní, je celkový součet hodnot všech uzlů roven
$(\lg n + 1)cn = cn \lg n + cn$. Tedy

$$T(n) = cn \lg n + cn = \Theta(n \lg n).$$

**Cvičení** Zapište algoritmus, který kombinuje dobré vlastnosti Merge-Sort a Insertion-Sort, totiž efektivitu pro třídění velkých polí a efektivitu pro třídění malých polí. Algoritmus třídí jako Merge-Sort velká pole až do velikosti $k$. Je-li pole velikosti menší nebo rovno $k$, probíhá třídění metodou Insertion-Sort. Algoritmus implementujte a proveďte pokusy s nastavením hodnoty $k$. Začněte s $k = 10$.

# Heap Sort

Třídění haldou (hromadou).

Třídění, které využívá datovou strukturu zvanou halda.

(Binární) halda je pole, ve kterém uložení prvků simuluje jejich přirozené uložení v binárním stromu, které respektuje jistá přirozená pravidla. Uložení prvků v haldě má výhody (rychlý přístup k prvkům, haldu lze rychle vytvořit).

Příklad haldy:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 12 | 9 | 8 | 9 | 7 | 6 | 7 | 5 | 3 | 4 |

Její reprezentace binárním stromem: [OBRAZEK]

Halda ještě jednou, s vyznačenými indexy:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 12 | 9 | 8 | 9 | 7 | 6 | 7 | 5 | 3 | 4 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Prvek $A[0]$ je rodičem prvků $A[1]$ a $A[2]$ (12 je rodičem 9 a 8, $A[1]$ a $A[2]$ jsou levý a pravý potomek $A[0]$),
$A[1]$ je rodičem $A[3]$ a $A[4]$ (9 je rodičem 9 a 7),
$A[2]$ je rodičem $A[5]$ a $A[6]$ (8 je rodičem 6 a 7),
$A[3]$ je rodičem $A[7]$ a $A[8]$ (9 je rodičem 5 a 3),
$A[4]$ je rodičem $A[9]$, další prvky halda neobsahuje.

Z indexu $i$ prvku lze snadno určit (spočítat) index Parent($i$) jeho rodiče, index Left($i$) jeho levého potomka i index Right($i$) jeho pravého potomka:

Parent($i$)
1 **return** $\lfloor (i-1)/2 \rfloor$

Left($i$)
1 **return** $2i+1$

Right($i$)
1 **return** $2i+2$

Vyzkoušejte na předchozím příkladě (např. Parent(7) $=3$, Left(0) $=1$, Right(3) $=8$). Dokažte, že to platí obecně.

**Definice** Pole $A[0..n-1]$ se nazývá **max-halda**, pokud pro každý $i = 1, \dots, n-1$ platí, že $A[i] \leq A[\text{Parent}(i)]$ (této nerovnosti říkáme vlastnost max-haldy).
$A[0..n-1]$ se nazývá **min-halda**, pokud pro každý $i = 1, \dots, n-1$ platí, že $A[i] \geq A[\text{Parent}(i)]$.

Výše uvedené pole je max-halda. "Halda" bude znamenat "max-halda".

Tedy, největší prvek (max-)haldy je $A[0]$.

Pro pole $A$ označujeme $length(A)$ velikost pole. Tedy, je-li $A[0..n-1]$ pole, je $length(A) = n$.

V úvahách níže může haldu tvořit jen část pole $A$. Velikost této části budeme značit $heap\text{-}size(A)$. Tedy část $A[0..heap\text{-}size(A)]$ tvoří haldu. Je-li $heap\text{-}size(A) = length(A)$, pak celé pole tvoří haldu.

Uvažujeme-li pro haldu $A[0..n-1]$ odpovídající binární strom $T_A$, můžeme zavést tyto pojmy:
-výška prvku $A[i]$ je výška uzlu reprezentujícího tento prvek v $T_A$, tj. počet hran nejdelší cesty, která vede z tohoto uzlu k některému z listů stromu $T_A$ (list je uzel, který nemá potomka). Např. výška prvku $A[1]$ je 2, výška prvku $A[2]$ je 1.
-výška haldy je výška $A[0]$, tj. výška kořene stromu $T_A$. Např. výška uvedené haldy je 3.

Výška haldy $A[0..n-1]$ je $\Theta(\lg n)$ (Ověřte na příkladě. Pak dokažte obecně, uvědomte si, že stačí dokázat, že výška je $\lfloor \lg n \rfloor$).

To je hlavní důvod efektivity třídění haldou. Základní operace totiž pracují v čase (tj. mají časovou složitost v nejhorším případě) asymptoticky shora omezeném výškou haldy, tj. mají časovou složitost v nejhorším případě $O(\lg n)$.

Uvedeme tři funkce: Max-Heapify, Build-Max-Heap, Heap-Sort (vlastní metoda třídění haldou).

# Max-Heapify(A,i)

Předpokládá se, že části pole $A$, které odpovídají stromu s kořenem $A[Left(i)]$ (tj. stromu, jehož kořen je prvek s indexem $Left(i)$) i stromu s kořenem $A[Right(i)]$ tvoří haldy. Cílem je zařadit správně prvek $A[i]$ tak, aby část pole odpovídající stromu s kořenem $A[i]$ tvořila haldu.

```
Max-Heapify(A, i)
  l ← Left(i)
  r ← Right(i)
  if l ≤ heap-size(A) and A[l] > A[i]
      then largest ← l
      else largest ← i
  if r ≤ heap-size(A) and A[r] > A[largest]
      then largest ← r
  if largest ≠ i
      then swap(A[i],A[largest])
           Max-Heapify(A, largest)
```

## Příklad Max-Heapify

Vstup $A$:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 12 | 2 | 8 | 7 | 9 | 6 | 7 | 5 | 3 | 4 |

Předpokládejme, že *heap-size(A)=10*.

Volání Max-Heapify($A$, 1), modře je zařazovaný prvek.

Na začátku tedy:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 12 | **2** | 8 | 7 | 9 | 6 | 7 | 5 | 3 | 4 |

Stavy pole $A$:

volání Max-Heapify($A$, 1), před provedením ř. 10:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 12 | 9 | 8 | 7 | **2** | 6 | 7 | 5 | 3 | 4 |

($largest = 4$),

volání Max-Heapify($A$, 4), před provedením ř. 10:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 12 | 9 | 8 | 7 | 4 | 6 | 7 | 5 | 3 | **2** |

($largest = 9$),

volání Max-Heapify($A$, 9), před provedením ř. 10:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 12 | 9 | 8 | 7 | 4 | 6 | 7 | 5 | 3 | **2** |

($i = largest = 9$, další volání Max-Heapify($A$, *largest*) se tedy neprovede).

Znázorněte průběh zařazování prvku $A[1]$ v binárním stromu.

## Časová složitost Max-Heapify

V nejhorším případě zařazovaný prvek "propluje" až do listu stromu. Má-li tedy zařazovaný prvek výšku $h$, je složitost v nejhorším případě $O(h)$ ($h$ je třeba chápat jako funkci, jejíž hodnota závisí na $n$, pro kořen je $h = \lfloor \lg n \rfloor$, pro prvek v úrovni pod kořenem je $h = \lfloor \lg n/2 \rfloor$, v další úrovni je $h = \lfloor \lg n/4 \rfloor$). Protože výška celého stromu s $n$ prvky je $\lfloor \lg n \rfloor$, je $h \leq \lfloor \lg n \rfloor$. Tedy složitost Max-Heapify v nejhorším případě je $O(\lfloor \lg n \rfloor)$ (je-li totiž $f \in O(g)$ a $g \leq g'$, je $f \in O(g')$). Protože $O(\lfloor \lg n \rfloor) = O(\lg n)$, je složitost Max-Heapify v nejhorším případě $O(\lg n)$.

To lze odvodit také z Master Theorem (později).

## Správnost Max-Heapify

Je snadno vidět (zdůvodněte).

# Build-Max-Heap(A)

Používá funkci Max-Heapify k vytvoření max-haldy z vstupního pole $A[0..n-1]$.
Idea: Prvky vstupního pole, které tvoří v odpovídajícím stromu listy, tvoří jednoprvkové max-haldy (nemají totiž potomky). To jsou právě prvky $A[\lfloor n/2 \rfloor], \dots, A[n-1]$ (ověřte, nejprve na příkladě, pak dokažte).
Build-Max-Heap prochází ostatní prvky od $A[\lfloor n/2 \rfloor - 1]$ až po $A[0]$ a každý zařadí funkcí Max-Heapify.

Build-Max-Heap($A[0..n-1]$)

*heap-size(A)* $\leftarrow n$
**for** $i \leftarrow \lfloor n/2 \rfloor - 1$ **downto** 0
  **do** Max-Heapify($A, i$)

## Příklad Build-Max-Heap

Vstup $A$:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 5 | 3 | 7 | 2 | 4 | 8 | 9 | 6 | 4 |

Volání Build-Max-Heap($A$), zobrazujeme stavy pole po provedení ř. 3, modře je zařazený prvek, podtženy jsou prvky na pozicích, přes které se zařazený prvek dostal na své místo.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 5 | 3 | 7 | $\underline{4}$ | 4 | 8 | 9 | 6 | **2** | $i = 4$, po provedení ř. 3 |
| 1 | 5 | 3 | $\underline{9}$ | 4 | 4 | 8 | **7** | 6 | 2 | $i = 3$, po provedení ř. 3 |
| 1 | 5 | $\underline{8}$ | 9 | 4 | 4 | **3** | 7 | 6 | 2 | $i = 2$, po provedení ř. 3 |
| 1 | $\underline{9}$ | 8 | $\underline{7}$ | 4 | 4 | 3 | **5** | 6 | 2 | $i = 1$, po provedení ř. 3 |
| $\underline{9}$ | $\underline{7}$ | 8 | $\underline{6}$ | 4 | 4 | 3 | 5 | **1** | 2 | $i = 0$, po provedení ř. 3 |

Znázorněte průběh v binárním stromu.

## Časová složitost Build-Max-Heap

Snadno lze odvodit, že časová složitost Build-Max-Heap v nejhorším případě je $O(n \lg n)$: V Build-Max-Heap dojde k $\lfloor n/2 \rfloor$ voláním funkce Max-Heapify, jejíž časová složitost v nejhorším případě je $O(\lg n)$ (viz výše). Tedy časová složitost Build-Max-Heap v nejhorším případě je $O(\lfloor n/2 \rfloor \lg n) = O(n \lg n)$. Tento odhad ale není těsný, existuje lepší horní odhad! Lze totiž ukázat [odvození nebudu u zkoušky požadovat], že časová složitost Build-Max-Heap v nejhorším případě je $O(n)$ (to je přesnější horní odhad, protože $n = o(n \lg n)$).

## Správnost Build-Max-Heap

Plyne z následující vlastnosti: Při každém vstupu do cyklu 2–3 tvoří stromy s kořeny odpovídajícími prvkům s indexy $i+1$, $i+2$, $\ldots n$ max-haldy.

To je pravda při prvním vstupu do cyklu 2–3, protože tyto stromy jsou jednoprvkové (ony kořeny jsou zároveň listy).

Pokud vlastnost platí při vstupu pro hodnotu $i = k$, pak platí i při dalším vstupu (tj. pro $i = k - 1$), protože při průchodu pro $i = k$ se na ř. 3 prvek $A[k]$ zařadí tak, že strom s kořenem odpovídajícím $A[k]$ tvoří max. haldu.

Při posledním vstupu na ř. 2 je $i = -1$ (pak dojde k ukončení), a tedy speciálně strom s kořenem odpovídajícím prvku $A[i+1]$, což je $A[0]$, tvoří max-haldu. To znamená, že celé pole $A[0..n-1]$ tvoří max-haldu.

# Heap-Sort(A)

Idea: Vytvoříme max-haldu z vstupního pole $A[0..n-1]$. $A[0]$ tedy obsahuje největší prvek. Vyměníme ho s $A[n-1]$ a dále pracujeme jen s $A[0..n-2]$ (*heap-size*$(A)$ snížíme o 1). Nový prvek $A[0]$ zařadíme pomocí Max-Heapify$(A,0)$. Tak vznikne halda $A[0..n-2]$ s největším prvkem $A[0]$. $A[0]$ vyměníme s $A[n-2]$ a tak dále.

Heap-Sort($A[0..n-1]$)
Build-Max-Heap($A$)
**for** $i \leftarrow n-1$ **downto** 1
    **do** swap($A[0]$,$A[i]$)
        *heapsize*$(A) \leftarrow$ *heapsize*$(A) - 1$
        Max-Heapify$(A, 0)$

## Příklad Heap-Sort

Vstup $A$:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 5 | 3 | 7 | 2 | 4 | 8 | 9 | 6 | 4 |

Po provedení ř. 1 je (viz příklad k Build-Max-Heap):

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 9 | 7 | 8 | 6 | 4 | 4 | 3 | 5 | 1 | 2 |

Znázorníme stavy pole $A$ při každém vstupu do cyklu (po provedení ř. 2) a (pokud se v cykllu pokračuje) po provedení ř. 3. Modře je setříděná část pole, podtrženy jsou prvky vyměněné na ř. 3.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 7 | 8 | 6 | 4 | 4 | 3 | 5 | 1 | 2 | $i = 9$; po ř. 2 |
| $\underline{2}$ | 7 | 8 | 6 | 4 | 4 | 3 | 5 | 1 | $\underline{\mathbf{9}}$ | $i = 9$; po ř. 3 |
| 8 | 7 | 4 | 6 | 4 | 2 | 3 | 5 | 1 | 9 | $i = 8$; po ř. 2 |
| $\underline{1}$ | 7 | 4 | 6 | 4 | 2 | 3 | 5 | $\underline{\mathbf{8}}$ | **9** | $i = 8$; po ř. 3 |
| 7 | 6 | 4 | 5 | 4 | 2 | 3 | 1 | 8 | 9 | $i = 7$; po ř. 2 |
| $\underline{1}$ | 6 | 4 | 5 | 4 | 2 | 3 | $\underline{\mathbf{7}}$ | **8** | **9** | $i = 7$; po ř. 3 |
| 6 | 5 | 4 | 1 | 4 | 2 | 3 | 7 | 8 | 9 | $i = 6$; po ř. 2 |
| $\underline{3}$ | 5 | 4 | 1 | 4 | 2 | $\underline{\mathbf{6}}$ | **7** | **8** | **9** | $i = 6$; po ř. 3 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 4 | 4 | 1 | 3 | 2 | 6 | 7 | 8 | 9 | $i = 5$; po ř. 2 |
| 2 | 4 | 4 | 1 | 3 | **5** | **6** | **7** | **8** | **9** | $i = 5$; po ř. 3 |
| 4 | 3 | 4 | 1 | 2 | 5 | 6 | 7 | 8 | 9 | $i = 4$; po ř. 2 |
| 2 | 3 | 4 | 1 | **4** | **5** | **6** | **7** | **8** | **9** | $i = 4$; po ř. 3 |
| 4 | 3 | 2 | 1 | 4 | 5 | 6 | 7 | 8 | 9 | $i = 3$; po ř. 2 |
| 1 | 3 | 2 | **4** | **4** | **5** | **6** | **7** | **8** | **9** | $i = 3$; po ř. 3 |
| 3 | 1 | 2 | 4 | 4 | 5 | 6 | 7 | 8 | 9 | $i = 2$; po ř. 2 |
| 2 | 1 | **3** | **4** | **4** | **5** | **6** | **7** | **8** | **9** | $i = 2$; po ř. 3 |
| 2 | 1 | 3 | 4 | 4 | 5 | 6 | 7 | 8 | 9 | $i = 1$; po ř. 2 |
| **1** | **2** | **3** | **4** | **4** | **5** | **6** | **7** | **8** | **9** | $i = 1$; po ř. 3 |

Znázorněte průběh v binárním stromu.

## Časová složitost Heap-Sort

Časová složitost Build-Max-Heap v nejhorším případě je $O(n)$.
Pak je proveden $(n-1)$-krát cyklus, ve kterém se na ř. 3 a 4 provedou 2 instrukce a na ř. 5 se provede Max-Heapify s časovou složitostí v nejhorším případě $O(\lg n)$. Celkem je tedy časová složitost Heap-Sort v nejhorším případě $O(n + (n-1)(2 + \lg n))$, což je $O(n \lg n)$.

Všimněte si, že k $O(n \lg n)$ dojdeme i s odhadem $O(n \lg n)$ pro Build-Max-Heap.

## Správnost Heap-Sort

Plyne z následující vlastnosti: Po provedení ř. 3 obsahuje část $A[i..n-1]$ $n-i$ největších prvků pole $A$, které jsou vzestupně setříděné.

Tato vlastnost platí při prvním průchodu cyklem 2–5, zachovává se každým průchodem cyklem a její platnost při posledním průchodu znamená, že $A[0..n-1]$ je vzestupně setříděné.

# Dolní odhad složitosti algoritmů třídění porovnáváním

Dosud probrané algoritmy mají společný rys: nepoužívají jinou informaci o prvcích pole než tu, kterou lze získat jejich porovnáváním (tj. $A[i] \leq A[j]$, $A[i] = A[j]$, $A[i] < Aj$ apod.).

Takovým algoritmům se říká **algoritmy třídění porovnáváním**.

Nepoužívají např. informaci o hodnotě čísel, které se třídí (viz např. Counting Sort uvedený později).

Důležitý výsledek:

**Věta** Časová složitost v nejhorším případě libovolného algoritmu třídění porovnáváním je $\Omega(n \lg n)$.

Důkaz na dalším slajdu.

Tj. složitost je asymptoticky zdola omezena funkcí $n \lg n$. Protože Merge Sort i Heap Sort mají složitost v nejhorším případě $O(n \lg n)$ (shora omezena funkcí $n \lg n$), oba jsou tzv. **asymptoticky optimální algoritmy**.

## Důkaz předchozí věty

Je založen na vhodné abstrakci pojmu algoritmus třídění porovnáním.
Algoritmus $\approx$ binární strom. Přesně:
činnost algoritmu pro pole s $n$ prvky $\approx$ binární strom s uzly a hranami, které jsou označené následovně.

Uzly obsahují výrazy $i : j$ $(1 \leq i, j \leq n)$, např. $1 : 2$, $1 : 3$, ....

Z uzlu vychází dvě hrany, jedna je označena $\leq$, druhá $>$.

Každý list stromu je označen permutací čísel $1, \ldots, n$, která označuje, jak je třeba vstupní posloupnost přerovnat tak, aby vznikla uspořádaná posloupnost. Např. $\langle 1, 3, 2 \rangle$ říká, že je třeba 3. prvek vstupní posloupnosti přesunout na 2. místo (ve výsledné posloupnosti) a 2. prvek vstupní posloupnosti na 3. místo.

Činnost každého algoritmu třídění porovnáváním pro vstupy velikosti $n$ lze reprezentovat takovým stromem.

Strom algoritmu Insert Sort pro $n = 3$ je na dalším slajdu.

Převzato z knihy Cormen et al.: Introduction to algorithms. MIT Press.



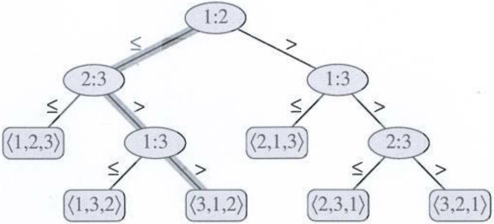



**Figure 8.1** The decision tree for insertion sort operating on three elements. An internal node annotated by $i{:}j$ indicates a comparison between $a_i$ and $a_j$. A leaf annotated by the permutation $\langle \pi(1), \pi(2), \ldots, \pi(n) \rangle$ indicates the ordering $a_{\pi(1)} \le a_{\pi(2)} \le \cdots \le a_{\pi(n)}$. The shaded path indicates the decisions made when sorting the input sequence $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$; the permutation $\langle 3, 1, 2 \rangle$ at the leaf indicates that the sorted ordering is $a_3 = 5 \le a_1 = 6 \le a_2 = 8$. There are $3! = 6$ possible permutations of the input elements, so the decision tree must have at least 6 leaves.

Vysvětlení na přednášce.

Argument: Uvažujme strom reprezentující činnost daného algoritmu pro vstupní posloupnost velikosti $n$.

1. Označme $h$ hloubku tohoto stromu, označme $l$ počet listů tohoto stromu.

2. Strom musí v listech obsahovat aspoň $n!$ permutací (pro každou permutaci $\pi$ existuje vstupní posloupnost taková, že k jejímu uspořádání je třeba provést $\pi$). Tedy $n! \leq l$.

3. Binární strom hloubky $h$ má nejvýše $2^h$ listů, tedy $l \leq 2^h$.

Celkem tedy $n! \leq l \leq 2^h$.
Tedy (logaritmováním nerovnosti) $\lg n! \leq h$ a protože platí $\lg n! = \Omega(n \lg n)$ (to nebudeme dokazovat), je zřejmé, že $h = \Omega(n \lg n)$.

To znamená, že v nejhorším případě (výpočet odpovídající nejdelší cestě od kořene k listu, délka této cesty je $h$) musí algoritmus provést aspoň $\Omega(n \lg n)$ porovnání. Důkaz je hotov.

Pozn.: $h$ je třeba chápat jako funkci s argumentem $n$.

# Counting Sort

Algoritmus třídění, který využívá jinou informaci než porovnávání. Lze použít pro třídění celých čísel $0 \ldots k$.

Základní myšlenka: Vstupní pole $A[0..n-1]$, výstupní (setříděné) pole je $B[0..n-1]$.

Pro jednoduchost předpokládajme, že všechny prvky $A$ jsou navzájem různé. Kam má přijít prvek $A[n-1]$ v poli $B$?
Označmě $C[i]$ počet prvků v $A$ menších než nebo rovných $i$.

Pak $A[n-1]$ přijde na pozici s indexem $C[A[n-1]]-1$,
$A[n-2]$ přijde na pozici s indexem $C[A[n-2]]-1$,
...
$A[0]$ přijde na pozici s indexem $C[A[0]]-1$.

Tj. provedeme $B[C[A[n-1]]-1] \leftarrow A[n-1]$,
$B[C[A[n-2]]-1] \leftarrow A[n-2], \ldots, B[C[A[0]]-1] \leftarrow A[0]$.

**Příklad** $k = 7$, $n = 5$.

$A = \boxed{7}\boxed{0}\boxed{5}\boxed{2}\boxed{1}$

Pak $C = \boxed{1}\boxed{2}\boxed{3}\boxed{3}\boxed{3}\boxed{4}\boxed{4}\boxed{5}$.

Tedy provedeme

$B[C[A[n-1]]-1] \leftarrow A[n-1]$, tj. $B[C[A[4]]-1] \leftarrow A[4]$, tj. $B[1] \leftarrow 1$,

$B[C[A[n-2]]-1] \leftarrow A[n-2]$, tj. $B[C[A[3]]-1] \leftarrow A[3]$, tj. $B[2] \leftarrow 2$,

$B[C[A[2]]-1] \leftarrow A[2]$, tj. $B[3] \leftarrow 5$,

$B[C[A[1]]-1] \leftarrow A[1]$, tj. $B[0] \leftarrow 0$,

$B[C[A[0]]-1] \leftarrow A[0]$, tj. $B[4] \leftarrow 7$,

tedy $B = \boxed{0}\boxed{1}\boxed{2}\boxed{5}\boxed{7}$.

```
Counting-Sort(A, B, k)
for i ← 0 to k
    do C[i] ← 0
for j ← 0 to n − 1
    do C[A[j]] ← C[A[j]] + 1
        // C[i] obsahuje počet prvků v A rovných i
for i ← 1 to k
    do C[i] ← C[i] + C[i − 1]
        // C[i] obsahuje počet prvků v A ≤ i
for j ← n − 1 downto 0
    do B[C[A[j]] − 1] ← A[j]
       C[A[j]] ← C[A[j]] − 1
```

**Příklad** $k = 7$, $n = 5$. Obměna předcházejícího, pole obsahje stejné prvky.
$A = \boxed{7 \mid 1 \mid 5 \mid 1 \mid 1}$ Pak $C = \boxed{0 \mid 3 \mid 3 \mid 3 \mid 3 \mid 4 \mid 4 \mid 5}$.

Tedy provedeme

$B[C[A[n-1]]-1] \leftarrow A[n-1]$, tj. $B[C[A[4]]-1] \leftarrow A[4]$, tj. $B[2] \leftarrow 1$,
a po provedení ř. 9 je $C = \boxed{0 \mid 2 \mid 3 \mid 3 \mid 3 \mid 4 \mid 4 \mid 5}$

$B[C[A[n-2]]-1] \leftarrow A[n-2]$, tj. $B[C[A[3]]-1] \leftarrow A[3]$, tj. $B[1] \leftarrow 1$,
a po provedení ř. 9 je $C = \boxed{0 \mid 1 \mid 3 \mid 3 \mid 3 \mid 4 \mid 4 \mid 5}$

$B[C[A[2]]-1] \leftarrow A[2]$, tj. $B[3] \leftarrow 5$,
a po provedení ř. 9 je $C = \boxed{0 \mid 1 \mid 3 \mid 3 \mid 3 \mid 3 \mid 4 \mid 5}$

$B[C[A[1]]-1] \leftarrow A[1]$, tj. $B[0] \leftarrow 1$,
a po provedení ř. 9 je $C = \boxed{0 \mid 0 \mid 3 \mid 3 \mid 3 \mid 3 \mid 4 \mid 5}$

$B[C[A[0]]-1] \leftarrow A[0]$, tj. $B[4] \leftarrow 7$,
a po provedení ř. 9 je $C = \boxed{0 \mid 0 \mid 3 \mid 3 \mid 3 \mid 3 \mid 4 \mid 4}$

tedy $B = \boxed{1 \mid 1 \mid 1 \mid 5 \mid 7}$.

Tedy během provádění algortimu (po ř. 9) je $C[i]$ rovno počtu prvků v $A$ s hodnotou $< i$ plus počtu prvků v $A$ s hodnotou $= i$, které ještě nebyly zařazeny do $B$.

# Složitost Counting Sort

ř. 1–2: $\Theta(k)$ instrukcí.
ř. 3–4: $\Theta(n)$ instrukcí.
ř. 5–6: $\Theta(k)$ instrukcí.
ř. 7–9: $\Theta(n)$ instrukcí.

Celkem tedy $\Theta(k+n)$ instrukcí, složitost Counting Sort v nejhorším případě je tedy $\Theta(k+n)$.

Je-li $k = O(n)$ (např. $k \leq n$ nebo obecně $k \leq cn$ pro $c > 0$), je tedy složitost v nejhorším případě $\Theta(n)$.

# Radix Sort

Základní myšlenku tohoto algoritmu používali operátoři mechanických třídiček děrných štítků. První odkaz 1929, algoritmus radix sort 1954 (Seward).

Základní myšlenka: Třídíme $d$-místná čísla (na každém z $d$ míst jsou číslice $0 \dots 9$). Třídění proběhne v $d$ průchodech. V 1. průchodu se čísla setřídí podle jejich poslední číslice, v 2. průchodu pak podle jejich předposlední číslice, . . . , v průchodu $i$ pak podle číslice na pozici $i$ (zprava), . . . , v posledním průchodu nakonec podle první číslice.

Přitom třídění při každém průchodu musí být stabilní. Tj. je-li při vstupu do průchodu $i$ číslo $a$ ve vstupním poli před číslem $b$ a mají-li $a$ a $b$ shodné číslice na pozici $i$ (podle které třídění probíhá), je číslo $a$ před číslem $b$ i ve výstupním poli průchodu $i$.

**Příklad** $d = 3$.

$$
\begin{array}{ccccccc}
644 & & 50\mathbf{1} & & 5\mathbf{0}1 & & \mathbf{0}99 \\
728 & & 64\mathbf{4} & & 1\mathbf{1}8 & & \mathbf{1}18 \\
501 & & 72\mathbf{8} & & 7\mathbf{2}8 & & \mathbf{1}28 \\
128 & \longrightarrow & 12\mathbf{8} & \longrightarrow & 1\mathbf{2}8 & \longrightarrow & \mathbf{5}01 \\
099 & & 11\mathbf{8} & & 6\mathbf{4}4 & & \mathbf{6}44 \\
118 & & 09\mathbf{9} & & 0\mathbf{9}9 & & \mathbf{7}28
\end{array}
$$

Proč musí být třídění v jednotlivých průchodech stabilní?

Zformulujte obecnou definici stabilního třídicího algoritmu.

Lze třídít od 1. číslice (v 1. průchodu) po poslední (v posledním průchodu)? (Co by to obnášelo?)

```
Radix-Sort(A, d)
    for i ← 1 to d
        do Stable-Sort(A, i)
```

$A$ ... vstupní pole, $d$ ... počet číslic čísel v poli $A$

Přitom Stable-Sort je libovolný stabilní třídicí algoritmus, Stable-Sort($A$, $i$) setřídí $A$ podle číslic na pozici $i$ zprava. Takovým algoritmem je např. Counting Sort (proč je stabilní?).

Implementujte Radix-Sort (cvičení).

Které z algoritmů Insertion Sort, Selection Sort, Bubble Sort, Quick Sort, Merge Sort, Heap Sort jsou stabilní?

Radix Sort lze použít nejen pro třídění polí $d$-místných čísel s číslicemi $0 \dots 9$, tj. polí s prvky tvaru

| | | | | |
|---|---|---|---|---|
| $d$.číslice | $(d-1)$.číslice | $\cdots$ | 2.číslice | 1.číslice |

, ale obecně i polí s prvky, které lze chápat jako zřetězení $d$ položek. Tj. prvky, které mají $d$ pozic. V každé z nich se vyskytují prvky, které lze navzájem porovnávat, a tedy podle nich třídit.

**Příklady**

$d$-místná čísla s číslicemi $0 \dots k$, tj. pro $d = 5$ a $k = 7$ je prvkem pole např.

| | | | | |
|---|---|---|---|---|
| 7 | 0 | 1 | 6 | 7 |

.

Řetězce znaků délky $d$, tj. pro $d = 5$ je prvkem pole např.

| | | | | |
|---|---|---|---|---|
| $K$ | $a$ | $r$ | 1 | 0 |

, tj. řetězec Kar10.

Příkladem setříděného pole se 4 prvky je

| | | | |
|---|---|---|---|
| $Ala02$ | $Jan11$ | $Jan20$ | $Kar10$ |

.

Data ve tvaru rok-měsíc-den (pak tedy $d = 3$), tj. prvkem pole je např.

| | | |
|---|---|---|
| 2011 | 2 | 16 |

, tj. 2011–2–16.

Příkladem setříděného pole se 5 prvky je

| | | | | |
|---|---|---|---|---|
| 1918-10-28 | 1939-3-15 | 1948-2-25 | 1968-8-21 | 1989-11-17 |

.

Řetězce bitů dané délky, rozdělené na $d$ podřetězců délky $r$ (důležitý příklad). Např. řetězce délky 32 lze chápat jako zřetězení 8 podřetězců ($d = 8$) délky 4 ($r = 8$). Prvkem pole pak bude např.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0011 | 1010 | 1111 | 0001 | 0000 | 1101 | 0101 | 0101 |

.

**Správnost Radix-Sort** se snadno vidí (podrobně zdůvodněte).

# Složitost Radix Sort

Předpokládejme, že prvky pole $A$ jsou $d$-místná čísla s číslicemi $0 \dots k$. Pole $A$ obsahuje $n$ prvků.

Pokud časová složitost v nejhorším případě algoritmu Stable-Sort je $\Theta(n+k)$, je časová složitost v nejhorším případě algoritmu Radix-Sort $\Theta(d(n+k))$.

Důkaz: Stable-Sort se provede $d$-krát, tedy se celkem provede $d\Theta(n+k)$ kroků a $d\Theta(n+k) = \Theta(d(n+k))$ (uvědomte si, co tato rovnost znamená).

Všimněte si, že argumentem složitosti jsou tři čísla, $n, k, d$. To je při popisu složitosti algoritmů běžná praxe, $n$ je velikost vstupu, $d, k$ jsou parametry.

Chápeme-li $d$ a $k$ jako konstanty, je tedy složitost v nejhorším případě $\Theta(n)$.

## Třídění bitových řetězců

Předpokládejme nyní, že prvky pole $A$ jsou $b$-bitové řetězce (mohou reprezentovat čísla, znaky apod.). Předpokládejme, že správné uspořádání těchto prvků je shodné s jejich uspořádáním coby řetězců z 0 a 1 (neboli coby čísel zapsaných binárně).

K setřídění pole je možné použít Radix-Sort následujícím způsobem:
Na $b$-bitový řetězec se díváme jako na posloupnost $d$ $r$-bitových řetězců. Protože obecně nemusí být $b = dr$, první z nich může mít méně než $r$ bitů (to je pouze technický problém). Platí $d = \lceil b/r \rceil$.
Každý $r$-bitový řetězec reprezentuje číslo z intervalu 0 až $2^r - 1$. Můžeme tedy použít Radix-Sort s hodnotami $d$ a $k = 2^r - 1$.

Z výše uvedeného vztahu pro složitost Radix-Sort plyne, že řasová složitost v nejhorším pžípadě je

$$\Theta(d(n+k)) = \Theta(\lceil b/r \rceil (n + 2^r - 1)) = \Theta((b/r)(n + 2^r)).$$

Jak zvolit $r$, tj. jak bitový řetězec rozdělit?

Jak zvolit při daných $n$ a $b$ hodnotu $r$ tak, aby složitost byla nejmenší, tj. aby hodnota $(b/r)(n+2^r)$ byla nejmenší? Je jasné, že musí být $r \leq b$. Rozlišme dva případy.

1. $b < \lfloor \lg n \rfloor$.
   Pak volba $r = b$ (nebo $r = b/2, b/3, \dots$) zajistí složitost $\Theta(n)$ (a ta je asymptoticky optimální).
   Proč: Pak je totiž $n + 2^r = \Theta(n)$ ($2^r$ roste pomaleji než $n$, protože $2^r < 2^{\lfloor \lg n \rfloor} \leq 2^{\lg n} = n$). Tedy $(b/r)(n+2^r) = (n+2^r) = \Theta(n)$.
2. $b \geq \lfloor \lg n \rfloor$.
   Pak volba $r = \lfloor \lg n \rfloor$ zajistí asymptoticky optimální složitost.
   Proč: Pak je složitost $\Theta(b/\lfloor \lg n \rfloor(n + 2^{\lfloor \lg n \rfloor})) = \Theta(bn/\lg n)$. Ta je optimální. Totiž, když $r > \lfloor \lg n \rfloor$, pak protože $2^x$ roste rychleji než $x$, je $(b/r)(n+2^r) \geq b/\lfloor \lg n \rfloor(n + 2^{\lfloor \lg n \rfloor}) = \Theta(bn/\lg n)$, tedy $(b/r)(n+2^r) = \Omega(bn/\lg n)$.
   A když $r < \lfloor \lg n \rfloor$, pak $b/r > b/\lfloor \lg n \rfloor$ a přitom $(n+2^r) = \Theta(n)$, z čehož plyne $(b/r)(n+2^r) = \Omega(bn/\lg n)$.

# Bucket Sort

Předpokládá, že prvky vstupního pole jsou čísla z intervalu $[0, 1)$, která vznikla náhodným procesem s rovnoměrným rozdělením (ve smyslu teorie pravděpodobnosti). Pro naše potřeby stačí říct, že to znamená: Pravděpodobnost, že prvek $A[i]$ je v intervalu $[a, b]$ $(0 \leq a \leq b < 1)$ je $b - a (= \frac{b-a}{1-0})$.

Základní myšlenka: Interval $[0, 1)$ rozdělíme na $n$ intervalů stejné velikosti, tj. na $[0, 1/n)$, $[1/n, 2/n)$, ..., $[(n-1)/n, 1)$. Pro každý z těchto intervalů vytvoříme spojový seznam (dynamické pole) $B[i]$ (interval $i$ je $[i/n, (i+1)/n)$, nazývá se také "bucket").

Projdeme prvky pole $A$ a každý z nich vložíme do příslušného intervalu, tj. do příslušného seznamu $B[i]$. Díky předpokladu obsahují seznamy málo prvků.

Každý seznam setřídíme.

Prvky setříděných seznamů $B[0], \ldots, B[n-1]$ vložíme po řadě do výstupního pole. Získáme tak setříděné pole.

**Příklad** $n = 10$.

1. sloupec = indexy
2. sloupec = vstupní pole
3. sloupec = prvky zařazené v seznamech $B[i]$

| i | A[i] | | B[i] | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0.35 | | $\emptyset$ | | | | |
| 1 | 0.95 | | $\rightarrow$ | 0.11 | | | |
| 2 | 0.11 | | $\rightarrow$ | 0.25 | 0.21 | | |
| 3 | 0.38 | | $\rightarrow$ | 0.35 | 0.38 | 0.32 | |
| 4 | 0.74 | $\longrightarrow$ | $\emptyset$ | | | | $\longrightarrow$ |
| 5 | 0.25 | | $\emptyset$ | | | | |
| 6 | 0.71 | | $\emptyset$ | | | | |
| 7 | 0.32 | | $\rightarrow$ | 0.74 | 0.71 | | |
| 8 | 0.21 | | $\emptyset$ | | | | |
| 9 | 0.91 | | $\rightarrow$ | 0.95 | 0.91 | | |

(pokračování)

1. sloupec = indexy
2. sloupec = prvky v seznamech $B[i]$ po setřídění seznamů
3. sloupec = výstupní, setříděné pole

| i | B[i] | | | | | A[i] |
|---|---|---|---|---|---|---|
| 0 | $\emptyset$ | | | | | 0.11 |
| 1 | $\rightarrow$ | 0.11 | | | | 0.21 |
| 2 | $\rightarrow$ | 0.21 | 0.25 | | | 0.25 |
| 3 | $\rightarrow$ | 0.32 | 0.35 | 0.38 | | 0.32 |
| 4 | $\emptyset$ | | | | $\longrightarrow$ | 0.35 |
| 5 | $\emptyset$ | | | | | 0.38 |
| 6 | $\emptyset$ | | | | | 0.71 |
| 7 | $\rightarrow$ | 0.71 | 0.74 | | | 0.74 |
| 8 | $\emptyset$ | | | | | 0.91 |
| 9 | $\rightarrow$ | 0.91 | 0.95 | | | 0.95 |

# Pseudokód Bucket Sort

```
Bucket-Sort(A[0..n − 1])
1  for i ← 0 to n − 1
2      do vlož A[i] do seznamu B[⌊n · A[i]⌋]
3  for i ← 0 to n − 1
4      do Sort(B[i])
5  vlož postupně prvky z B[0], . . . , B[n − 1] do pole A
```

Pro setřídění seznamu $B[i]$ na ř. 4 (Sort($B[i]$)) lze použít např. Insertion-Sort (lze snadno implementovat pro třídění seznamů).

**Správnost Bucket-Sort** se snadno vidí (podrobně zdůvodněte).

# Složitost Bucket-Sort

Na ř. 1–2 se provede $\Theta(n)$ kroků.
Na ř. 5 se provede $\Theta(n)$ kroků.

**V nejhorším případě** To je případ, kdy všechny prvky z $A$ byly umístěny do jednoho seznamu $B[i]$. Má-li algoritmus Sort složitost v nejhorším případě $\Theta(f(n))$, pak protože $f(n) = \Omega(n)$, je
$\Theta(n) + \Theta(f(n)) + \Theta(n) = \Theta(f(n))$, a tedy složitost Bucket-Sort v nejhorším případě je $\Theta(f(n))$. Zvolíme-li tedy Insertion-Sort, je to $\Theta(n^2)$.

**V průměrném případě** Lze ukázat (s použitím jednoduchého aparátu teorie pravděpodobnosti), že je $\Theta(n)$ (ukážeme později).
Klíčový je při tom fakt, že složitost setřídění seznamu $B[i]$ v průměrném případě je $O(2 - 1/n)$.

# Který algoritmus třídění je tedy nejlepší?

Dva algoritmy s řádově stejnou složitostí se mohou prakticky chovat rozdílně. To je dáno i tím, že konstanty ukryté v O-notaci jejich odhadu složitosti jsou různé. Dále je to dáno povahou vstupních dat (mohou být příznivá pro jeden, ale nepříznivá pro jiný algoritmus).
Pravidlo: Volíme algoritmus s menší čas. složitostí (např. Heap-Sort místo Insertion-Sort). U dvou algoritmů se stejnou řádovou složitostí zvolíme podle jejich praktického chování (měříme doby trvání na reálných datech, tj. na typických datech, pro která budou algoritmy používány).

Z algoritmů porovnáváním je velmi rychlý Quick-Sort (malá režie).

Použít Quick-Sort nebo Radix-Sort? Složitost v průměrném případě pro Quick-Sort je $\Theta(n \lg n)$, v nejhorším případě pro Radix-Sort, když $b \leq \lg n$, je $\Theta(n)$. Režie (konstanta ukrytá v O-notaci) u Radix-Sort je ale větší než u Quick-Sort. Každý z řádově $n$ kroků Radix-Sort může tedy trvat podstatně déle než každý z řádově $n \lg n$ kroků Quick-Sort. Který z nich zvolit nakonec záleží na charakteristice vstupních dat a na implementaci (např. Quick-Sort obvykle efektivně využívá cache paměti).

# Závěrem ke třídění

- Ukázali jsme si nejdůležitější třídicí algoritmy.
- Existuje mnoho jejich variant.
- Existují další třídicí algoritmy. Doporučuji projít přehled, např. na internetu (Wikipedia apod.).
- Probrané algoritmy jsou určeny pro třídění v operační paměti. Nevěnovali jsme se tzv. externímu třídění, tj. třídění velkých dat, která se nevejdou celá do paměti (viz další slajd).
- K jednotlivým algoritmům se budeme průběžně vracet.

# Stručně k vnějšímu třídění

– Jsou-li data tak velká, že se nevejdou do operační paměti počítače, je třeba k jejich setřídění použít externí paměť (např. pevný disk).

– Algoritmy třídění, které používají externí paměť, se nazývají algoritmy externího (vnějšího) třídění. Z tohoto pohledu jsou dosud probrané algoritmy algoritmy interního (vnitřního) třídění.

– Algoritmy vnějšího třídění obvykle kombinují třídění v operační paměti (části dat, která se tam vejde) a slévání setříděných částí do větších částí, které jsou ukládány na disk.

– Princip slévání je stejný jako ten, použitý v algoritmu Merge-Sort. Slévání ale probíhá z více vstupních polí (v Merge-Sort se slévají dvě vstupní pole).

– Příklad: Pro setřídění 1 GB dat (např. čísel) na počítači, kde je k dispozici 100MB operační paměti:

# Stručně k vnějšímu třídění

- – Načti 100MB dat do paměti a setřiď je algoritmem vnitřního třídění (např. Quick-Sort). Setříděnou posloupnost zapiš to nového souboru $i$ na disk. To zopakuj 10x (pro $i = 1, \dots, 10$).
  - Vytvoř v paměti 11 velkých bloků $B_1, \dots, B_{10}, C$ (např. po 9 MB). Každý blok $B_i$ naplň daty ze souboru $i$ (od začátku souboru). Vytvoř nový výstupní soubor.
  - Pomocí slévání z 10 bloků $B_i$ naplň blok $C$. Obsah bloku $C$ zapiš do výstupního souboru, blok $C$ považuj za prázdný. Pokud se při slévání některý z bloků $B_i$ vyprázdní, načti do něj dalších 9 MB dat ze souboru $i$. Opakuj, dokud nejsou všechna data ze vstupních souborů zapsána ve výstupním souboru.

# Pořádkové statistiky

# Pořádkové statistiky

**Definice** ***i*-tá pořádková statistika** množiny s $n$ prvky je $i$. nejmenší prvek dané množiny ($i$. prvek po setřídění posloupnosti).

Nejmenší prvek (**minimum**) ... 1. pořádková statistika.

Největší prvek (**maximum**) ... $n$. pořádková statistika.

**Medián** ("prostřední prvek").
$n$ liché: medián $= \frac{n+1}{2}$. pořádková statistika.
$n$ sudé:
dolní medián $= \lfloor \frac{n+1}{2} \rfloor$. pořádková statistika
(pokud neřekneme jinak, bude medián znamenat dolní medián)
horní medián $= \lceil \frac{n+1}{2} \rceil$. pořádková statistika

**problém (výběr):**
**vstup:** $A[0 \dots n-1]$ (pole $n$ čísel) a číslo $i$ ($1 \leq i \leq n$)
**výstup:** $i$. pořádková statistika prvků z $A$

**Příklady**

vstup $A = \langle 2, 1, 4, 6, 2, 8, 1 \rangle$, $i = 1$, výstup $= 1$,
vstup $A = \langle 2, 1, 4, 6, 2, 8, 1 \rangle$, $i = 2$, výstup $= 1$,
vstup $A = \langle 2, 1, 4, 6, 2, 8, 1 \rangle$, $i = 3$, výstup $= 2$,
...
vstup $A = \langle 2, 1, 4, 6, 2, 8, 1 \rangle$, $i = 7$, výstup $= 8$.

Lze tedy použít následující algoritmus.

Naive-Select($A[0..n-1]$, $i$)
1 Sort($A$)
2 **return**($A[i]$)

Časová složitost v nejhorším případě algoritmu Naive-Select je rovna časové složitosti v nejhorším případě algoritmu Sort (tj. např. $O(n \lg n)$ pro Heap-Sort).

Ukážeme algoritmus, s časovou složitostí v průměrném případě $O(n)$.

Nejprve tento **problém**: Jak najít nejmenší a největší prvek v $A[0 \ldots n-1]$?

**Možnost 1**: Odděleně najít nejmenší ($n-1$ porovnání) a největší ($n-1$ porovnání), celkem $2n-2$ porovnání.

**Možnost 2**: porovnávat po dvojicích.

Porovnáme první dva prvky, menší dáme do min, větší do max (1 porovnání). Vezmeme další dojici prvků porovnáme je mezi sebou, menší z nich pak s min, vetší s max a příslušným způsobem aktualizujeme hodnoty min a max (3 porovnání).

Zbyde-li nakonec jeden prvek (při lichém počtu prvků), porovnáme ho s min, pak příp. s max a aktualizujeme min a max (nejvýše 2 porovnání).

Celkem tedy nejvýše $1 + 3\lfloor (n-2)/2 \rfloor + 2 = 3(\lfloor (n-2)/2 \rfloor + 1) = 3\lfloor n/2 \rfloor$ porovnání.

# i. pořádková statistika v průměrném čase $\Theta(n)$

Základní myšlenka: Rekurzívní postup.

Hedáme $i$. prvek v poli, které vznikne setříděním vstupního pole $A[p \ldots r]$ (na začátku $A[0 \ldots n-1]$).

Použijeme Partition z algoritmu Quick-Sort s náhodným výběrem pivota. $q$ je index pivota, pro $k = q - p + 1$ je pivot tedy $k$. prvkem v poli $A$.

Pokud $i = k$, je hledaný prvek $A[q]$.
Pokud $i < k$, je hledaný prvek $i$. prvek v poli, které vznikne setříděním levé části, $A[p \ldots q-1]$.
Pokud $i > k$, je hledaný prvek $i - k$. prvek v poli, které vznikne setříděním levé části, $A[q+1 \ldots r]$.

Rozdíl oproti Quick-Sort: Po provedení Partition se zabýváme vždy jen jednou částí (levou nebo pravou).

```
Randomized-Select(A, p, r, i)
  if p = r
     then return A[p]
  q ← Randomized-Partition(A, p, r)
  k ← q − p + 1
  if i = k
     then return A[q]
  else if i < k
     then return Randomized-Select(A, p, q − 1, i)
  else return Randomized-Select(A, q + 1, r, i − q + p − 1)
```

Vrátí $i$. pořádkovou statistiku pole $A$.

```
Randomized-Partition(A, p, r)
  k ← Random(p, r)
  swap(A[k], A[r])
  return Partition(A, p, r)
```

Provede Partition s náhodným výběrem pivota. Funkce Random$(p, r)$ vrátí náhodné celé číslo z intervalu $[p, r]$.

**Správnost** Randomized-Select
Je zřejmá z popisu základní myšlenky algoritmu.

Je jen třeba si uvědomit, že nikdy nedojde k volání Randomized-Select($A$, $p$, $q-1$, $i$) pro $q-1<p$ (ř. 8), ani Randomized-Select($A$, $q+1$, $r$, $k-q$) pro $r<q+1$ (ř. 9).

Zdůvodněte.
Uvědomte si, že $q-1<p$ (ř. 8) znamená $p=q$ a že $r<q+1$ (ř. 9) znamená $q=r$.

## **Časová složitost** Randomized-Select

**nejhorší případ** $\Theta(n^2)$

Např. když pivot je vždy největším prvkem v poli $A[p \dots r]$ a hledáme nejmenší prvek ($i = 1$).

Pak dojde k $n-1$ voláním Randomized-Partition (pro pole s $n$, $n-1$, $\dots 2$ prvky). Volání Randomized-Partition na pole s $k$ prvky trvá $\Theta(k)$ kroků.

Celkem je tedy třeba $\Theta(n + (n-1) + \cdots + 2) = \Theta(n^2)$ kroků.

**průměrný případ** $\Theta(n)$, pokud jsou v poli po dvou různé prvky.

Lze ukázat pomocí aparátu teorie pravděpodobnosti (komentář na přednášce).

# i. pořádková statistika v nejhorším čase O(n)?

Takový algoritmus existuje (Select, viz např. Cormen et al.: Introduction to Algorithms, MIT Press, 2005, Section 9.3).

Základní myšlenka:
1. Rozděl pole na $\lceil n/5 \rceil$ skupin po 5 prvcích (poslední má $\leq 5$ prvky).
2. Najdi v každé skupině medián (setřiď pole, medián pak vyber ze setříděného).
3. Rekurzívním použitím Select nakonec najdi medián $x$ z $\lceil n/5 \rceil$ mediánů z 2.
4. Použij Partition s pivotem $x$ na vstupní pole. Nechť $k$ je index prvku $x$ v poli po provedení Partition.
5. Je-li $i = k$, vrať $A[k]$. Je-li $i < k$, použij Select pro nalezení $i$. nejmenšího prvku v levé části. Je-li $i > k$, použij Select pro nalezení $(i-k)$. nejmenšího prvku v pravé části.

# Stavba potrubí

Bylo nalezeno $n$ nalezišť ropy a vybudováno $n$ vrtů. Souřadnice vrtu $i$ jsou $\langle x_i, y_i \rangle$. Je třeba postavit ropovod, na který se vrty napojí. Ropovod má být rovnoběžný s osou $x$. Každý vrt na něj bude napojen potrubím, které vede kolmo na ropovod. Jak určit souřadnici $y$ má ropovod mít, aby součet délek všech potrubí napojených na ropovod byl co nejmenší? Ukažte, že existuje algoritmus se složitostí $\Theta(n)$ v nejhorším případě, který $y$ určí.

Návod: Nakreslete si obrázek. Použijte předchozí algoritmus na hledání pořádkové statistiky se složitostí $\Theta(n)$ v nejhorším případě.

# Analýza quicksort v průměrném případě

- pokročilejší problém
- naše první analýza časové složitosti algoritmu v průměrném případě
- ukážeme i obecný princip
- budeme uvažovat R-Quick-Sort = Quick-Sort s náhodným výběrem pivota
- chceme ukázat, že pro časovou složitost $T(n)$ v průměrném případě platí
$$T(n) = O(n \log_2 n).$$
- Pro zjednodušení předpokládáme, že prvky v poli jsou navzájem různé.

```
Quick-Sort(A, p, r)
if p < r
   then q ← Partition(A, p, r)
        Quick-Sort(A, p, q − 1)
        Quick-Sort(A, q + 1, r)
```

```
Partition(A,p,r)
x ← A[r]
i ← p − 1
for j ← p to r − 1
    do if A[j] ≤ x
          then i ← i + 1
               swap(A[i], A[j])
swap(A[i + 1],A[r])
return i + 1
```

Quicksort s náhodným výběrem pivota:

R-Quick-Sort($A$, $p$, $r$)
**if** $p < r$
  **then** $q \leftarrow$ Randomized-Partition($A, p, r$)
    R-Quick-Sort($A$, $p$, $q - 1$)
    R-Quick-Sort($A$, $q + 1$, $r$)

Randomized-Partition($A$, $p$, $r$)
$k \leftarrow$ Random($p, r$)
swap($A[k], A[r]$)
**return** Partition($A, p, r$)

První pozorování pro setřídění pole $A$ s $n$ prvky:

– celkový počet kroků (instrukcí provedených v R-Quick-Sort) je určen počtem kroků provedených v Partition
(tento počet je „dominující“, zbylé kroky jsou „režie“: další kroky mimo Partition)
– Partition je zavolána max. $n$ krát (výběr pivota max. $n$ krát)
– počet kroků provedených v jednom volání Partition je $\Theta$(počet porovnání na ř. 4)
(je zhruba konstanta krát počet porovnání na ř. 4)

Přesněji tedy: Jsou-li
$A_1, \ldots, A_k$ všechna vstupní pole délky $n$,
$t(A_\omega)$ je počet kroků pro setřídění pole $A_\omega$ a
$X(A_\omega)$ je počet porovnání na ř. 4 Partition při setřídění pole $A_\omega$, pak:

**Lemma**

$$t(A_\omega) \leq O(n) + c\dot{X}(A_\omega),$$

kde $c > 0$ je konstanta (udává, kolik instrukcí se provede s jedním porovnáním na ř. 4).

Označme

$$E(X) = \frac{X(A_1) + \cdots + X(A_k)}{k},$$

tj. $E(X)$ je průměrný počet porovnání na ř. 4 (průměr přes všechna vstupní pole).
Protože

$$T(n) = \frac{t(A_1) + \cdots + t(A_k)}{k},$$

platí vzhledem k

$$t(A_\omega) \leq O(n) + c\dot{X}(A_\omega),$$

zřejmě

$$T(n) \leq O(n) + cE(X).$$

Pro ověření $T(n) = O(n \log_2 n)$ stačí tedy ukázat, že

$$E(X) = O(n \log_2 n).$$

Označme pro vstupní pole $A_\omega$:

$$
\begin{aligned}
z_i(A_\omega) &= i\text{-tý nejmenší prvek v poli } A_\omega \;, \\
Z_{ij}(A_\omega) &= \{z_i, \ldots, z_j\} \text{ pro } i < j \\
X_{ij}(A_\omega) &= \begin{cases} 1 & \text{pokud při setřídění } A_\omega \text{ dojde k porovnání } z_i \text{ a } z_j \\ 0 & \text{v opačném případě} \end{cases}
\end{aligned}
$$

Uvědomme si, že platí

$$X(A_\omega) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}(A_\omega)$$

a dále z vlastností průměru (tzv. linearita: $E(X+Y) = E(X) + E(Y)$):

$$
\begin{aligned}
E(X) &= E(\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E(X_{ij}) = \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} P(\text{dojde k porovnání } z_i \text{ a } z_j).
\end{aligned}
$$

Jak odvodit pravděpodobnost $P(\text{dojde k porovnání } z_i \text{ a } z_j)$?

K porovnání při $A_\omega$ dojde, právě když z prvků v $Z_{ij}(A_\omega) = \{z_i, \ldots, z_j\}$

- jako první je za pivota zvolen $z_i$, nebo
- je jako první za pivota zvolen $z_j$.
- Jinak se totiž $z_i$ a $z_j$ ocitnou v různých polovinách a nedojde k jejich porovnání.

Tedy

$$P(\text{dojde k porovnání } z_i \text{ a } z_j) =$$
$$P(\text{jako první je za pivota zvolen } z_i) + P(\text{jako první je za pivota zvolen } z_j)$$

Protože $Z_{ij}(A_\omega)$ má $j-i+1$ prvků a protože výběr každého prvku za pivota má stejnou pravděpodobnost, je

$$P(\text{jako první je za pivota zvolen } z_i) = \frac{1}{j-i+1}$$

a

$$P(\text{jako první je za pivota zvolen } z_j) = \frac{1}{j-i+1},$$

Tedy

$$P(\text{dojde k porovnání } z_i \text{ a } z_j) = \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1}.$$

Z dříve odvozeného

$$E(X) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} P(\text{dojde k porovnání } z_i \text{ a } z_j)$$

tedy máme

$$\begin{aligned} E(X) &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\ &< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} = \sum_{i=1}^{n-1} O(\log_2 n) = (n-1)O(\log_2 n) = O(n \log_2 n) \end{aligned}$$

Použili jsme přitom $\sum_{k=1}^{n} \frac{2}{k} = O(\log_2 n)$ (nebudeme dokazovat).

# Průběh zkoušky

Viz www stránky předmětu.

Termíny budou vypsány ve STAGu.

Student obdrží otázky a bude mít cca 20 min na přípravu (tužka a papír). Pak půjde k ústnímu zkoušení (cca 25 min).

Je třeba znát látku v rozsahu probíraném na přednáškách. Tj.:

- Probrané pojmy (problém, algoritmus, složitost, O-notace, . . . ). Umět s pojmy pracovat.
- Probrané algoritmy, jak pracují, umět simulovat činnost algoritmů.
- Složitosti probraných algoritmů.
- Porozumět jednoduchému algoritmu zapsanému v pseudokódu.
- Zapsat v pseudokódu jednoduchý algoritmus.