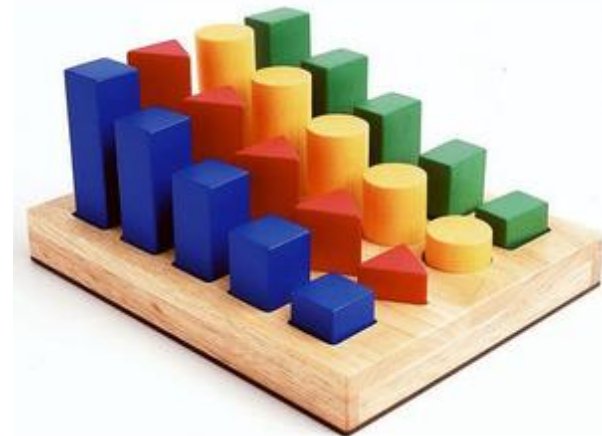
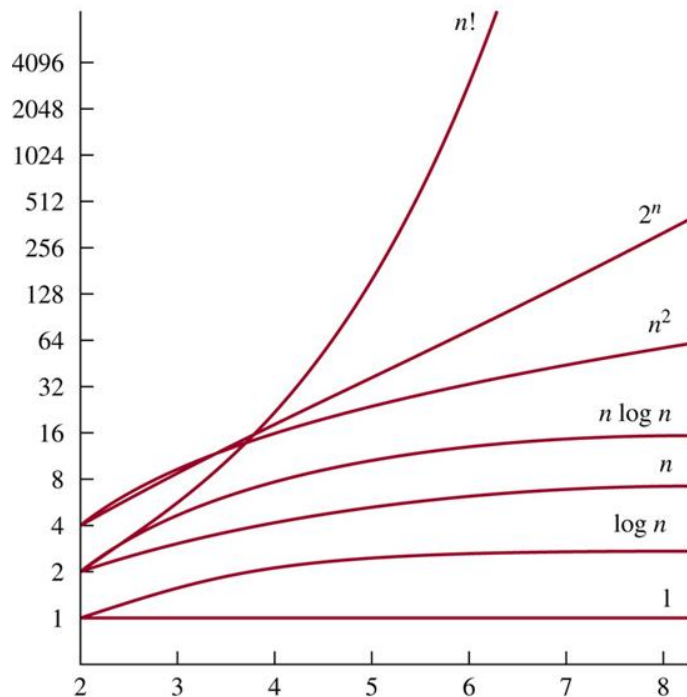




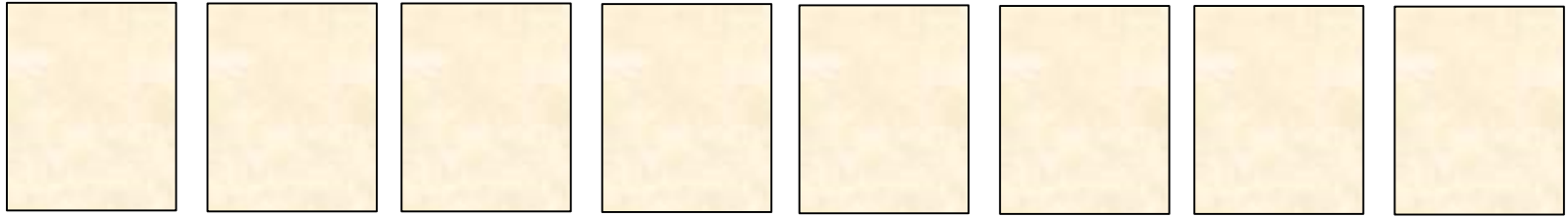
2. prednáška (16.2.2026)

O (d triedenia) k zložitosti





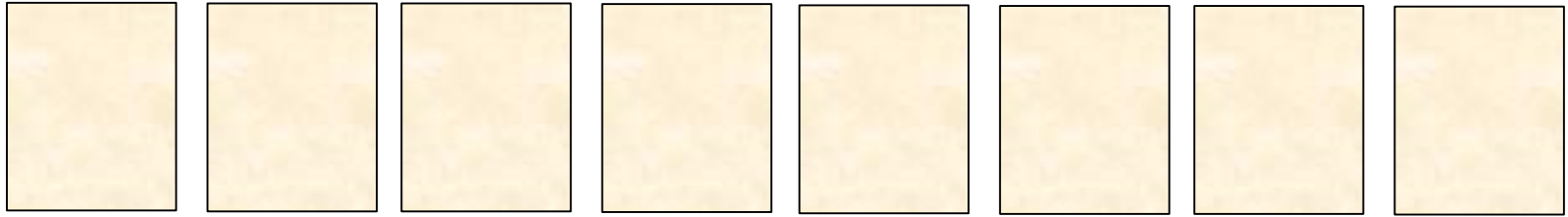
Hľadanie ihly v kope sena



- 8 kariet, pod každou je jedno číslo...
- *Úloha: Ako na najmenej otočení kariet zistiť, či je na niektorej z kariet zadané číslo?*
 - **Najlepší prípad:** Ak máme šťastie, tak ho nájdeme na prvý pokus.
 - **Najhorší prípad:** Ak tam číslo nie je, tak musíme nazrieť pod každú kartu.
 - pri n kartách po nanajvyš n krokoch vieme dať odpoveď



Hľadanie ihly v kope sena

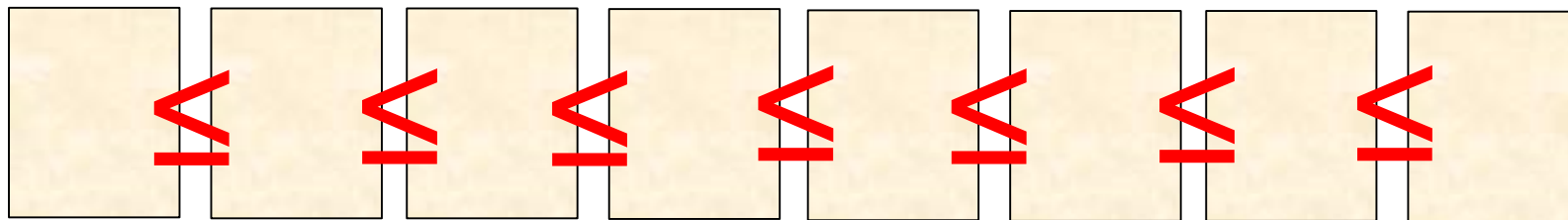


nanajvýš n opakovaní

```
for (int i=0; i<pole.length; i++)  
    if (pole[i] == hladaneCislo)  
        return true;  
  
return false;
```



Hľadanie ihly v kope sena



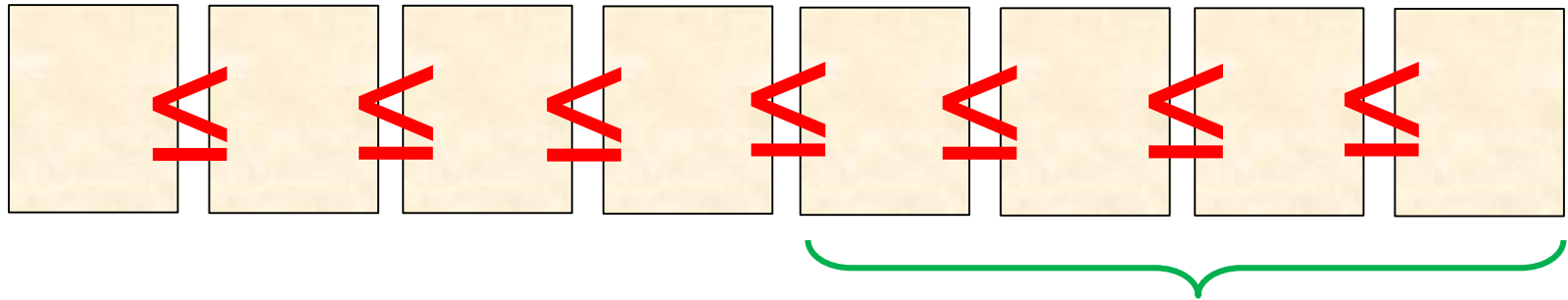
- 8 kariet, pod každou je jedno číslo...
- *Úloha: Ako na najmenej otočení kariet zistiť, či je na niektorej z kariet zadané číslo?*
- **Bonus:** čísla pod kartami tvoria neklesajúcu postupnosť...

Pomôže to nejako?



Hľadanie ihly v kope sena

Hľadané číslo **45**



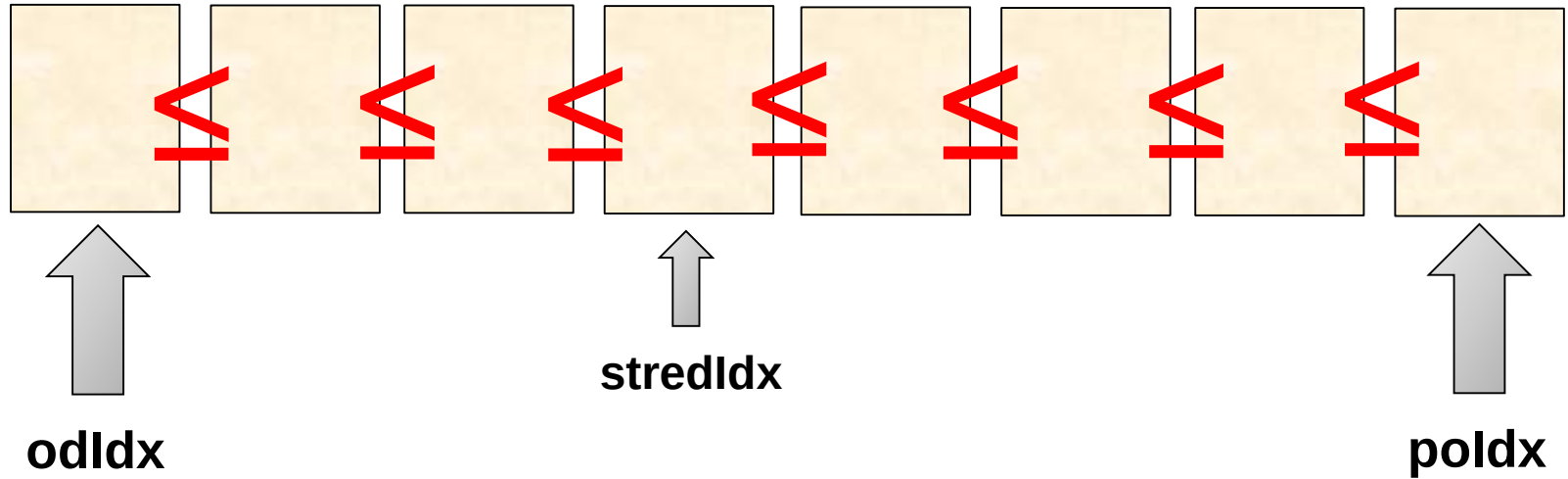
Ak je číslo 45 pod niektorou kartou, tak iba pod niektorou z týchto...

Zmenšenie problému:
z 8 kariet na 4 karty

Ak vyberáme kartu približne **v strede**, tak problém veľkosti **n** redukujeme na problém veľkosti **$n/2$**



Ako charakterizovať problém?



- **Problém** = množinu kariet, ktorú prehľadávame, môžeme popísať 2 číslami:
 - **index prvej** karty (kde úsek začína)
 - **index poslednej** karty (kde úsek končí)



Binárne vyhľadávanie

```

public boolean jeVPoli(int[] pole, int odIdx, int poIdx, int cislo) {
    if (odIdx > poIdx)
        return false;

    int stredIdx = (odIdx + poIdx) / 2;
    if (pole[stredIdx] == cislo)
        return true;

    if (cislo < pole[stredIdx])
        return jeVPoli(pole, odIdx, stredIdx-1, cislo);
    else
        return jeVPoli(pole, stredIdx+1, poIdx, cislo);
}

```

Ak máme 0 „kariet“

Vypočítame stred

Overíme, či v strede
nie je to, čo hľadáme

Rozhodneme sa, či v
hľadaní pokračujeme vľavo
alebo vpravo

*Nerekurzívna verzia
na cvičeniach*

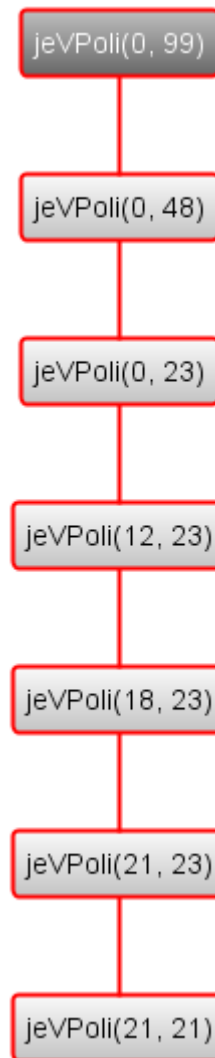


Binárne vyhľadávanie

```
public boolean jeVPoli(int[] pole, int odIdx, int poIdx, int cislo) {
    if (odIdx > poIdx)
        return false;

    int stredIdx = (odIdx + poIdx) / 2;
    if (pole[stredIdx] == cislo)
        return true;

    if (cislo < pole[stredIdx])
        return jeVPoli(pole, odIdx, stredIdx-1, cislo);
    else
        return jeVPoli(pole, stredIdx+1, poIdx, cislo);
}
```





Je binárne vyhľadávanie lepšie?

● Najhorší prípad:

- 1 krok: n kariet
- 2 krok: $n/2$ kariet
- 3 krok: polovica z $n/2$ kariet = $n/4 = n/2^2 = n/2^{3-1}$
- 4 krok: polovica z $n/4$ kariet = $n/8 = n/2^3 = n/2^{4-1}$
- ...
- k-ty krok: $n/2^{k-1}$ kariet
- ... **kedy skončíme?**
- posledný krok: **ostala 1 karta**

v informatike: $\log = \log_2$

$$\left\{ \begin{array}{l} n/2^{k-1} \leq 1 \\ n \leq 2^{k-1} \\ \log n \leq k-1 \\ 1 + \log n \leq k \end{array} \right.$$

Po nanajvýš $1 + \log n$ krochov končíme!



Je binárne vyhľadávanie lepšie?

n	Lineárne vyhľadávanie	Binárne vyhľadávanie
10	10	4
100	100	7
1000	1000	10
1000000	1000000	20
1000000000	1000000000	30
10000000000000	10000000000000	40

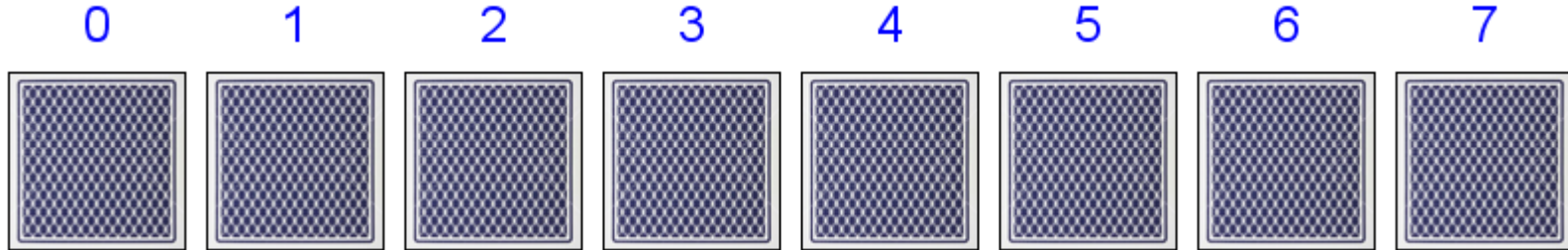


*Oplatí sa mať veci usporiadané
a vďaka tomu môcť použiť
binárne vyhľadávanie.*



Ako dať veci na správne miesto?

Kartičkový experiment...





Bublínkové triedenie

- Kým nie sú všetky susedné prvky v poli v správnom poradí, opakuj:
 - nájdi dva susedné prvky „v zlom poradí“ a navzájom ich vymeň

```
public static void bubbleSort(int[] p) {
    boolean bolaVymena;
    do {
        bolaVymena = false;

        for (int i=0; i<p.length-1; i++)
            if (p[i] > p[i+1]) {
                vymen(p, i, i+1);
                bolaVymena = true;
            }

    } while (bolaVymena);
}
```

Hľadáme susedné prvky,
ktoré sú v zlom poradí

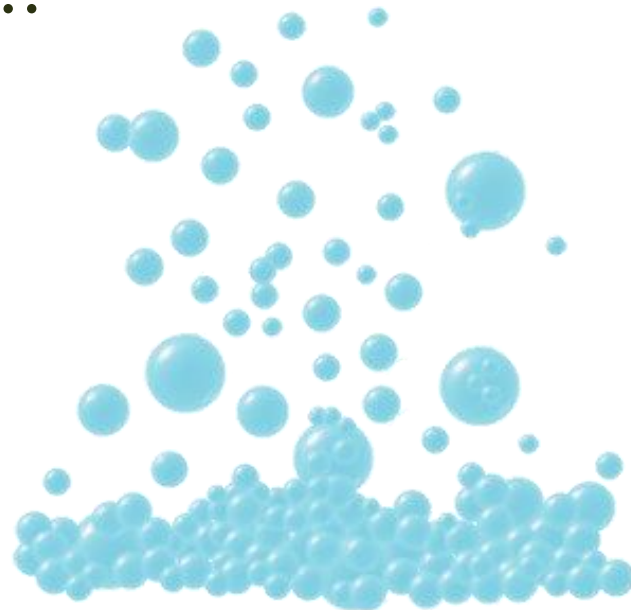
```
int pom = p[i];
p[i] = p[i+1];
p[i+1] = pom;
```

Poznačíme si, že sme
našli v poli niečo „zlé“



Bublinkové triedenie

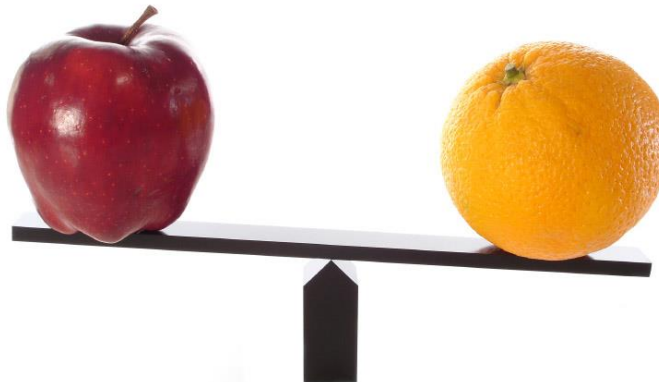
- **Pozorovanie:** od konca poľa sa postupne vytvára (vybubláva) **usporiadaná podpostupnosť**
 - **vylepšenie:** pri hľadaní „zlých“ susedov nemusíme ísť do konca poľa ($n, n-1, n-2, n-3, \dots$)
 - vylepšená verzia na cvičeniach...
- Koľko krát sa zopakuje **do-while** cyklus?
 - v najlepšom prípade (usporiadané pole): **1** krát
 - v najhoršom prípade: **n** krát





Bublinky a počet porovnaní

- Aký je **počet porovnaní**, ktoré vykoná algoritmus?
 - v najlepšom prípade: $n-1$
 - 1 iterácia **do-while** cyklu
 - v najhoršom prípade: $n(n-1) = n^2 - n$
 - $n-1$ porovnaní v jednej iterácii **do-while** cyklu
 - nanajvýš n iterácií **do-while** cyklu





Triedenie výberom

- SelectionSort, MinSort
- Stratégia pre pole veľkosti n :
 - ak je pole **veľkosti 1**, tak už je usporiadané a môžeme skončiť
 - **inak** ($n \geq 2$)
 - **nájd** **najmenšie** číslo
 - **vymeň** najmenšie číslo s číslom na začiatku poľa
 - prvé číslo v poli „ignoruj“ a usporiadať **zvyšok** poľa (zvyšných $n-1$ čísel) **rovnakým postupom**



Triedenie výberom

```
public static int indexNajmensieho(int[] p, int odIdx, int poIdx) {
    int najIdx = odIdx;
    for (int i = odIdx + 1; i <= poIdx; i++)
        if (p[i] < p[najIdx])
            najIdx = i;

    return najIdx;
}
```

nájd najmenšie číslo

vymeň najmenšie číslo s číslom na začiatku poľa

- prvé číslo v poli „ignoruj“ a usporiadať **zvyšok** poľa (zvyšných N-1 čísel) **rovnakým postupom**

```
public static void vymen(int[] p, int idx1, int idx2) {
    int pom = p[idx1];
    p[idx1] = p[idx2];
    p[idx2] = pom;
}
```




Triedenie výberom

```
public static void selectionSort(int[] p,
                                int odIdx, int poIdx) {
    if (odIdx == poIdx)
        return;

    vymen(p, odIdx, indexNajmensieho(p, odIdx, poIdx));
    selectionSort(p, odIdx + 1, poIdx);
}
```

*Rekurzívna myšlienka neznamená
rekurzívny program!*

Zmenšenie
problému o 1

- **nájdí najmenšie** číslo
- **vymeň** najmenšie číslo s číslom na začiatku poľa
- prvé číslo v poli „ignoruj“ a usporiadaj **zvyšok** poľa (zvyšných N-1 čísel) **rovnakým postupom**



Triedenie výberom

```
public static void selectionSort(int[] p) {  
    for (int i = 0; i < p.length - 1; i++) {  
  
        int minIdx = i;  
        for (int j = i + 1; j < p.length; j++)  
            if (p[j] < p[minIdx])  
                minIdx = j;  
  
        int pom = p[i];  
        p[i] = p[minIdx];  
        p[minIdx] = pom;  
    }  
}
```



Nájdi index najmenšieho prvku v podpoli začínajúcom na indexe i .

Vymeň prvok na indexe i a najmenší prvok v podpoli od indexu i .



Triedenie výberom a porovnania

```
public static void selectionSort(int[] p) {
    for (int i = 0; i < p.length - 1; i++) {

        int minIdx = i;
        for (int j = i + 1; j < p.length; j++)
            if (p[j] < p[minIdx])
                minIdx = j;

        vymen(p, i, minIdx);
    }
}
```

`p.length = 8`

i=0	7
i=1	6
i=2	5
i=3	4
i=4	3
i=5	2
i=6	1

Počet porovnaní pri poli veľkosti n :

$$(n-1) + (n-2) + \dots + 1 = \sum_{k=1}^{n-1} k = \frac{(n-1)n}{2}$$

Počet porovnaní



Bublinky vs. výber

Počet porovnaní	V najlepšom prípade	V najhoršom prípade
BubbleSort	$n-1$	$n(n-1)$
Vylepšený BubbleSort	$n-1$	$n(n-1)/2$
SelectionSort	$n(n-1)/2$	$n(n-1)/2$

- A čo ďalšie operácie v algoritmoch (výmeny, priradenia, ++, ...)?
- Ktorý z algoritmov je rýchlejší?





Analýza algoritmov

- **Korektnosť:**
 - *Robí algoritmus **vždy** to, čo má?*
- **Časová zložitosť:**
 - *Aký rýchly je algoritmus?*
- **Pamäťová zložitosť:**
 - *Koľko pamäte potrebuje algoritmus pri svojom behu?*
- ...
 - *komunikačná zložitosť, náročnosť na prostriedky a zdroje, ...*



Ktorý algoritmus je rýchlejší?

- Naprogramujem a **pomerám čas**:
 - na rôznych počítačoch rôzne operácie trvajú rôzne dlho (CISC vs. RISC, cachovanie, ... viac na princípoch počítačov, ...)
 - algoritmus **neoverím na všetkých vstupoch**
 - čo ak som algoritmus netestoval práve na vstupe, na ktorom algoritmus „počíta“ najdlhšie
 - čo ak algoritmus potrebujem pre také veľké vstupy, že len otestovanie akéhokoľvek z nich trvá niekoľko dní?

Riešenie: teoretická analýza



Analýza časovej zložitosti

- **Elementárna operácia:** 1 krok
 - čo je elementárna operácia závisí od uvažovaného výpočtového modelu
 - náš pohľad: každá jednoduchá operácia v Jave ako porovnanie, priradenie, ...
 - **pozor:** jeden príkaz nie je vždy jeden krok, napr. volanie metódy môže skrývať množstvo krokov
 - počet elementárnych operácií je **priamoúmerný** času
- Analýza časovej zložitosti algoritmu
 - = počítanie počtu krokov algoritmu bez toho, aby sme ho spustili



Analýza časovej zložitosti

- Koľko ste sa dozvedeli?
 - Algoritmus na poli $\{5, 1, 5, 2\}$ vykonal 131 krokov a na poli $\{6, 7, 1, 3, 1, 9\}$ vykonal 192 krokov.
 - Algoritmus A na poli veľkosti n vykoná nanajvýš $3*n^2-n+192$ krokov a algoritmus B vykoná presne $2*n^2+332*n-2$ krokov.
 - ktorý je rýchlejší?
- Čo nás naozaj zaujíma?





Analýza časovej zložitosti

- **Presný počet** krokov algoritmu závisí od konkrétneho vstupu
- Ak je vstup väčší, počet krokov algoritmu by mal byť väčší
 - časová zložitosť by mala byť funkciou od veľkosti vstupu
 - ak n je veľkosť vstupu, potom časová zložitosť je vyjadrená ako $T(n)$ - napr. maximálny počet krokov, ktoré potrebuje algoritmus na vstupe veľkosti n .
- Zvyčajne nás zaujíma **horné ohraničenie počtu** krokov (poskytuje „garancie“ trvania)

Na čom záleží?

Čo ak by sme každému z nich zvýšili mesačný príjem o ...

+ 5%

+1 €



+ 5%

+1 €



+ 5%

+1 €



Žiadna z týchto operácií zásadne nezmení „zaradenie človeka do nejakej príjmovej kategórie“.



Na čom záleží?

Aký najväčší vstup dokáže algoritmus s daným počtom operácií spracovať za určitý čas?

čas/zložitosť	N	N^2	N^3	2^N	$N!$
milisekunda	1 000 000	1 000	100	20	10
sekunda	1 000 000 000	30 000	1 000	30	12
minúta	∞	250 000	4 000	35	14
hodina	∞	2 000 000	15 000	41	15
deň	∞	9 000 000	44 000	46	16
mesiac	∞	51 000 000	130 000	51	17
rok	∞	170 000 000	310 000	54	18
tisícročie	∞	∞	3 100 000	64	21



Na čom záleží?

- Pri počítači, ktorý spraví 10^9 operácií za sekundu na 1, 2, 3, 1000, ... operáciách navyše **nezáleží...**
- 2-násobne viac operácií „vyrieši“ 2-krát rýchlejší počítač (napr. n^2 vs. $2n^2$)
- Z predchádzajúcej tabuľky:
 - rozdiel medzi n^2 a n^3 , či n^3 a 2^n žiadne konštantné zrýchlenie počítača nevyrieši pre všetky vstupy
- **Výzva:**

Vieme algoritmy na základe časovej zložitosti rozdeliť do nejakých „kategórií“ tak, aby v každej z nich boli „podobne“ časovo náročné algoritmy?



Asymptotická zložitosť

- Zaujíma nás, ako rastie zložitosť algoritmu, ak **veľkosť vstupu rastie** do nekonečna ($n \rightarrow \infty$)

- Algoritmus A: $T_A(n) = 3n^2 - 4n + 100$
- Algoritmus B: $T_B(n) = 300n^2 + 10$
- Algoritmus C: $T_C(n) = n^3 + n^2$

Zložitosť algoritmov A a B “rastie rovnako”.

B vs. A:
$$\lim_{n \rightarrow \infty} \frac{T_B(n)}{T_A(n)} = \lim_{n \rightarrow \infty} \frac{300n^2 + 10}{3n^2 - 4n + 100} = 100$$

C vs. B:
$$\lim_{n \rightarrow \infty} \frac{T_C(n)}{T_B(n)} = \lim_{n \rightarrow \infty} \frac{n^3 + n^2}{300n^2 + 10} = \infty$$

Zložitosť C “rastie neporovnateľne rýchlejšie” ako B



Theta Θ ako kategorizátor

- $T_A(n) = 3n^2 - 4n + 100$

- $T_B(n) = 300n^2 + 10$

$T_A(n)$ a $T_B(n)$ sú z hľadiska rastu porovnateľné.

- $\Theta(g(n))$ – množina všetkých funkcií, ktoré sú z hľadiska rastu **porovnateľné** s funkciou $g(n)$

$$\Theta(g(n)) = \left\{ f(n) \mid \exists c_1, c_2 \in R^+, \exists n_0 \in N, \right. \\ \left. \forall n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n) \right\}$$

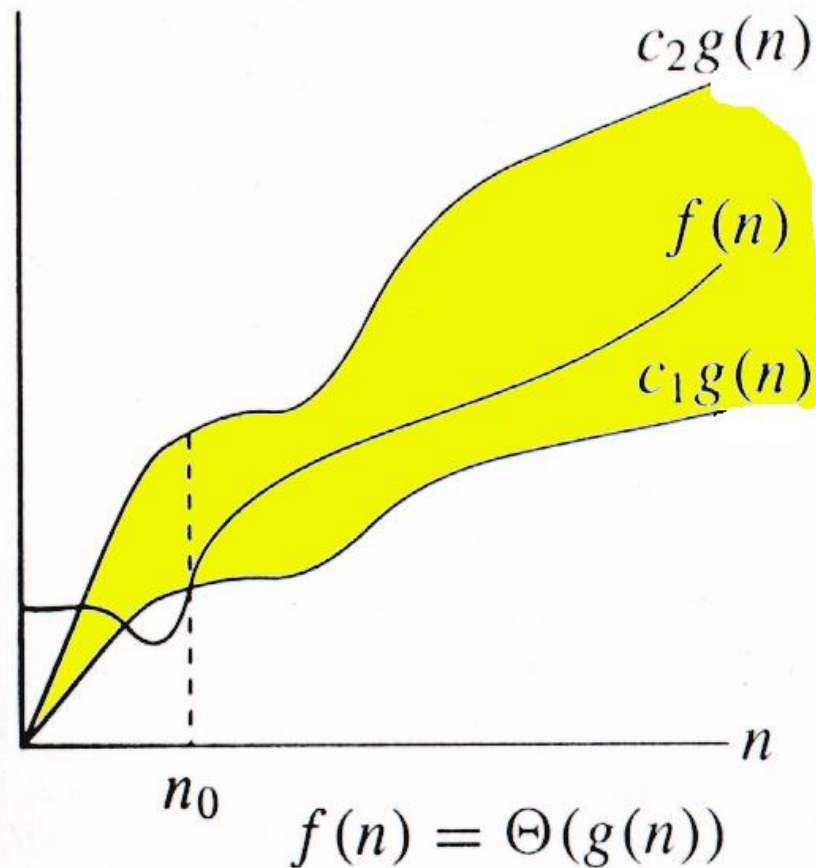
- Do $\Theta(n^2)$ patria $300n^2 + 10$, $2n^2 + n + \log n$, $0.5n^2$, ...

Pozorovanie: na multiplikatívnej a aditívnej konštantne nezáleží.



Theta Θ

$$\Theta(g(n)) = \left\{ f(n) \mid \exists c_1, c_2 \in R^+, \exists n_0 \in N, \right. \\ \left. \forall n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n) \right\}$$





Príklad s Theta Θ

Dokážte: $3n^2 - 4n + 100 \in \Theta(n^2)$

Dôkaz:

$$\Theta(g(n)) = \left\{ f(n) \mid \exists c_1, c_2 \in R^+, \exists n_0 \in N, \right. \\ \left. \forall n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n) \right\}$$

$$3n^2 - 4n + 100 \in \Theta(n^2) \Leftrightarrow$$

$$\exists c_1, c_2 > 0, \exists n_0 \in N, \forall n \geq n_0, c_1 n^2 \leq 3n^2 - 4n + 100 \leq c_2 n^2$$

Zvoľme $c_1=2$, $c_2=3$, $n_0=25$. Potrebujeme ukázať, že platí:

$$2n^2 \leq 3n^2 - 4n + 100 \leq 3n^2 \quad \text{pre } \forall n \geq 25$$

$$\begin{aligned} 2n^2 &\leq 3n^2 - 4n + 100 \\ -100 &\leq n^2 - 4n \\ -100 &\leq n(n-4) \end{aligned} \quad \uparrow$$

$$\begin{aligned} 3n^2 - 4n + 100 &\leq 3n^2 \\ 100 &\leq 4n \\ 25 &\leq n \end{aligned} \quad \uparrow$$



Príklad: Theta Θ

Dokážte: $n^3 \notin \Theta(n^2)$

Dôkaz (sporom):

$$\Theta(g(n)) = \left\{ f(n) \mid \exists c_1, c_2 \in R^+, \exists n_0 \in N, \right. \\ \left. \forall n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n) \right\}$$

$$n^3 \in \Theta(n^2) \Rightarrow \exists c_1, c_2 > 0, \exists n_0 \in N, \forall n \geq n_0, c_1 n^2 \leq n^3 \leq c_2 n^2$$

$$\Leftrightarrow \exists c_1, c_2 > 0, \exists n_0 \in N, \forall n \geq n_0, c_1 \leq n \leq c_2$$

Dôsledok:

$$n^3 \in \Theta(n^3) \wedge n^3 \notin \Theta(n^2) \Rightarrow \\ \Theta(n^3) \neq \Theta(n^2)$$

Spor, neplatí pre $n > \max(c_2, n_0)$

Dá sa ukázať, že sú navyše aj disjunktné.



Časová zložitosť *BubbleSort-u*

```
public static void bubbleSort(int[] p) {
    boolean bolaVymena;
    do {
        bolaVymena = false;

        for (int i=0; i<p.length-1; i++)
            if (p[i] > p[i+1]) {
                vymen(p, i, i+1);
                bolaVymena = true;
            }

    } while (bolaVymena);
}
```

Počet všetkých krokov algoritmu možno zhora aj zdola ohraničiť konštantným násobkom počtu porovnaní.

$$c_1.(\# \text{ porovnaní}) \leq \# \text{ krokov} \leq c_2.(\# \text{ porovnaní})$$



Θ a triedenia

Počet porovnaní	V najlepšom prípade	V najhoršom prípade
BubbleSort	$n-1$	$n(n-1)$
Vylepšený BubbleSort	$n-1$	$n(n-1)/2$
SelectionSort	$n(n-1)/2$	$n(n-1)/2$

- $c_1n(n-1)$ aj $c_2n(n-1)/2$ patria do $\Theta(n^2)$
 - BubbleSort aj SelectionSort majú **rovnakú asymptotickú časovú zložitost'** v najhoršom prípade!

Spomenuté triediace algoritmy patria do „rovnej kategórie“



A čo vieme povedať o **každom** behu?

Počet porovnaní	V najlepšom prípade	V najhoršom prípade
BubbleSort	$n-1$	$n(n-1)$
Vylepšený BubbleSort	$n-1$	$n(n-1)/2$
SelectionSort	$n(n-1)/2$	$n(n-1)/2$

Časová zložitost' **každého behu** BubbleSort-u je „medzi približne $n-1$ a $n(n-1)$ “

Každý beh SelectionSort-u spraví $\Theta(n^2)$ krokov.

Čo vieme povedať o časovej zložitosti **každého behu** (t.j. nielen v najhoršom prípade) algoritmu BubbleSort?



Horné ohraničenie ako garancia

- V praxi:
 - sú dôležité **garancie** typu „nebude to trvať dlhšie ako“...
 - pri analýze algoritmov je niekedy (často?) problém zaradiť časovú zložitosť algoritmu do Θ -množiny nejakej „peknej“ funkcie (n , n^2 , $\log n$, 2^n , $n \log n$...)

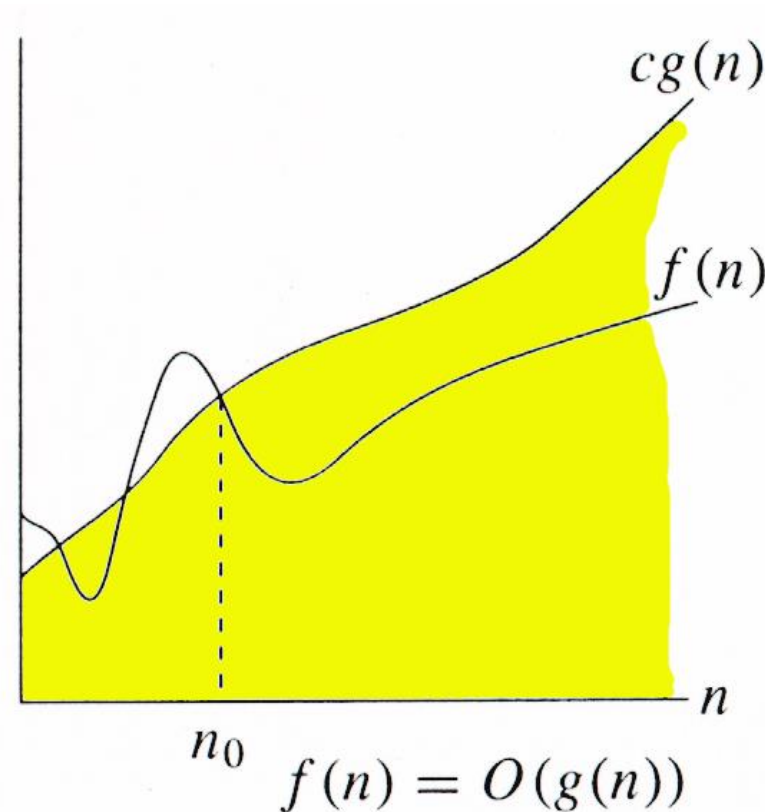
O



O-notácia

$$O(g(n)) = \{f(n) \mid \exists c \in R^+, \exists n_0 \in N, \forall n \geq n_0, f(n) \leq cg(n)\}$$

- $O(g(n))$ všetky funkcie, ktoré sú **asymptoticky zhora ohraničené** funkciou $g(n)$

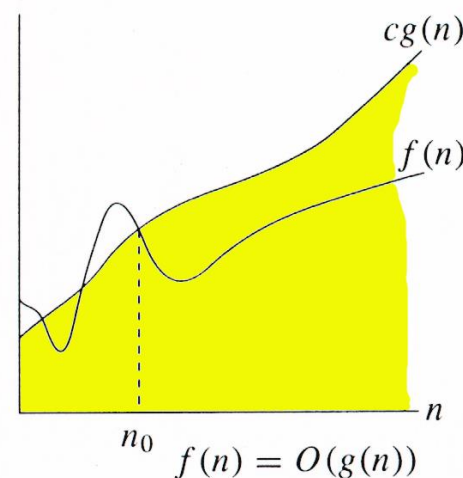




O-notácia príklady

$$O(g(n)) = \{f(n) \mid \exists c \in R^+, \exists n_0 \in N, \forall n \geq n_0, f(n) \leq cg(n)\}$$

- $n^2 \in O(n^2)$ $c=1, n_0=1$
- $2n^2 \in O(n^2)$ $c=2, n_0=1$
- $n \in O(n^2)$ $c=1, n_0=1$
- $2n^2 + 4n \in O(n^2)$ $c=3, n_0=4$
- $2022 \in O(n^2)$ $c=2022, n_0=1$



$$f(n) \in \Theta(g(n)) \Rightarrow f(n) \in O(g(n))$$

$$\Theta(g(n)) \subseteq O(g(n))$$



A čo vieme povedať o *každom* behu?

Počet porovnaní	V najlepšom prípade	V najhoršom prípade
BubbleSort	$n-1$	$n(n-1)$
Vylepšený BubbleSort	$n-1$	$n(n-1)/2$
SelectionSort	$n(n-1)/2$	$n(n-1)/2$

Cieľom analýzy asymptotickej časovej zložitosti algoritmu je nájsť čo najpomalšie rastúcu funkciu $f(n)$ takú, že časová zložitosť algoritmu je $O(f(n))$

Časová zložitosť každého behu algoritmu BubbleSort je:

$$O(n^2)$$



Divná konvencia?

- Matematicky korektne:

- $g(n) \in O(f(n))$
- $g(n) \in \Theta(f(n))$

- Čo sa používa:

- $g(n) = O(f(n))$
- $g(n) = \Theta(f(n))$

$$3n-2 = O(n)$$

Funkcia

Množina funkcií





Sumarizácia o zložitosti

- Asymptotická analýza časovej zložitosti:
 - zachycuje **ako rastie čas výpočtu** (počet krokov) algoritmu v závislosti od toho, ako **rastie veľkosť vstupu** (do nekonečna)
 - abstrahuje sa od **technických detailov** - aditívne a multiplikatívne konštanty sa „ignorujú“
 - umožňuje identifikovať rôzne „porovnateľné“ **triedy časovej zložitosti**
 - Θ - asymptoticky tesné ohraničenie
 - O – asymptotické ohraničenie zhora
 - väčšinou stačí, keďže poskytuje garancie o maximálnom trvaní behu algoritmu/výpočtu



Zaujímavé triedy zložitosti

- $O(1)$ - **konštantná** zložitosť
- $O(\log n)$ - **logaritmická** zložitosť
 - binárne vyhľadávanie v n-prvkovej usporiadanej postupnosti
- $O(n)$ - **lineárna** zložitosť
 - lineárne vyhľadávanie v n-prvkovej neusporiadanej postupnosti
 - hľadanie minimálnej hodnoty (v skutočnosti $\Theta(n)$)
- $O(n^2)$ - **kvadratická** zložitosť
 - bublinkové triedenia, triedenie výberom, ...
- $O(n^3)$ - **kubická** zložitosť
- $2^{O(n)}$ - **exponenciálna** zložitosť
 - kreslenie Kochovej krivky úrovne n

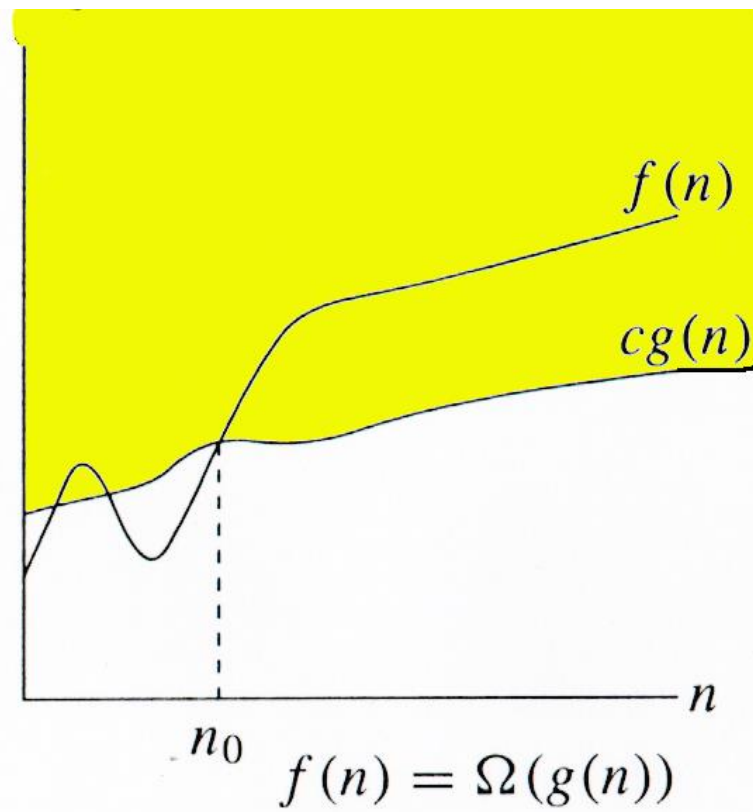
Na cvičeniach: Ktorá trieda je „lepšia“?



Ω -notácia

$$\Omega(g(n)) = \{f(n) \mid \exists c \in R^+, \exists n_0 \in N, \forall n \geq n_0, f(n) \geq cg(n)\}$$

- $\Omega(g(n))$ všetky funkcie, ktoré sú **asymptoticky zdola ohraničené** funkciou $g(n)$
- dolné ohraničenia sa používajú nie pri analýze algoritmov, ale **problémov**





Ω -notácia

- Problém hľadania minima v neusporiadanom v n -prvkom poli má časovú zložitosť $\Omega(n)$
 - žiaden algoritmus nemôže nájsť minimum v poli bez toho, aby pozrel všetkých n hodnôt, t.j. potrebuje aspoň cn krokov - jeho časová zložitosť je teda $\Omega(n)$
- „Krása“ asymptotických ohraničení zdola:
 - Umožňujú dokázať nie to, že ľudia doposiaľ nevymysleli rýchlejší algoritmus, ale že rýchlejší algoritmus nejde vymyslieť.
 - Algoritmus je **asymptoticky optimálny** pre nejaký problém, ak jeho časová zložitosť je $O(f(n))$ a dolné ohraničenie pre problém je $\Omega(f(n))$.



Príklad na záver

```
public boolean metoda(int[] p) {  
    for (int i = 0; i < p.length - 1; i++)  
        for (int j = i + 1; j < p.length; j++)  
            if (p[i] == p[j])  
                return true;  
  
    return false;  
}
```

Aká je časová zložitost' metódy?



Potrebuje to vôbec vedieť?

Dokumentácie k .Net Frameworku 4 od Microsoftu:

▲ Remarks

The `List<T>` is searched forward starting at the first element and ending at the last element.
This method determines equality using the default equality comparer `EqualityComparer<T>.Default`.
This method performs a linear search; therefore, this method is an $O(n)$ operation, where n is `C`.



Kvalitná dokumentácia obsahuje informácie o časovej zložitosti vykonávania metódy (efektívnosti implementácie).



ak nie sú otázky...

Ďakujem za pozornosť!

