

IDT, Přednáška 5

Libor Váša

Katedra informatiky a výpočetní techniky, Západočeská univerzita v Plzni

4. 3. 2024

Výpočetní složitost

Definice

$\mathcal{O}(f(n))$ označíme **množinu všech funkcí** $g(n)$ takových, pro které platí, že $g(n) < c * f(n)$ pro všechna $n > n_0 > 0$ a nějaké $c > 0$.

Definice

$\mathcal{O}(f(n))$ označíme **množinu všech funkcí** $g(n)$ takových, pro které platí, že $g(n) < c * f(n)$ pro všechna $n > n_0 > 0$ a nějaké $c > 0$.

Co to znamená?

- je-li $g(n) \in \mathcal{O}(f(n))$, pak $g(n)$ „se vejde pod“ $f(n)$

Definice

$\mathcal{O}(f(n))$ označíme **množinu všech funkcí** $g(n)$ takových, pro které platí, že $g(n) < c * f(n)$ pro všechna $n > n_0 > 0$ a nějaké $c > 0$.

Co to znamená?

- je-li $g(n) \in \mathcal{O}(f(n))$, pak $g(n)$ „se vejde pod“ $f(n)$
- je-li $g(n) \in \mathcal{O}(f(n))$, pak $g(n)$ „neroste rychleji než“ $f(n)$

Určení \mathcal{O} -notace

Prosté: najdeme n_0 a c a ukážeme, co se děje:

Určení \mathcal{O} -notace

Prosté: najdeme n_0 a c a ukážeme, co se děje:

$$g(n) = 20n + 100$$

Určení \mathcal{O} -notace

Prosté: najdeme n_0 a c a ukážeme, co se děje:

$$g(n) = 20n + 100$$

$$f(n) = n$$

Určení \mathcal{O} -notace

Prosté: najdeme n_0 a c a ukážeme, co se děje:

$$g(n) = 20n + 100$$

$$f(n) = n$$

$$g(n) < c * f(n)$$

Určení \mathcal{O} -notace

Prosté: najdeme n_0 a c a ukážeme, co se děje:

$$g(n) = 20n + 100$$

$$f(n) = n$$

$$g(n) < c * f(n)$$

$$20n + 100 < c * n$$

Určení \mathcal{O} -notace

Prosté: najdeme n_0 a c a ukážeme, co se děje:

$$g(n) = 20n + 100$$

$$f(n) = n$$

$$g(n) < c * f(n)$$

$$20n + 100 < c * n$$

$$20 + 100/n < c$$

Určení \mathcal{O} -notace

Prosté: najdeme n_0 a c a ukážeme, co se děje:

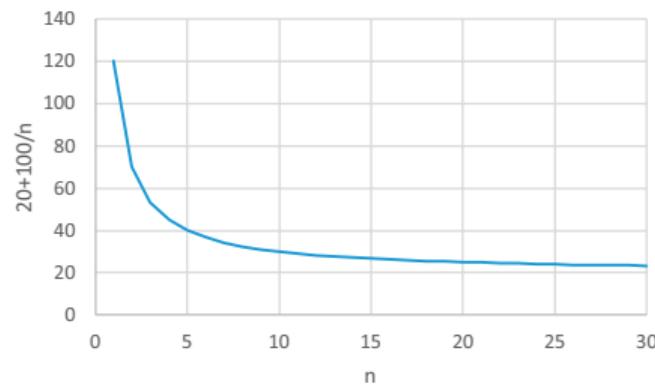
$$g(n) = 20n + 100$$

$$f(n) = n$$

$$g(n) < c * f(n)$$

$$20n + 100 < c * n$$

$$20 + 100/n < c$$



Určení \mathcal{O} -notace

Prosté: najdeme n_0 a c a ukážeme, co se děje:

$$g(n) = 20n + 100$$

$$f(n) = n$$

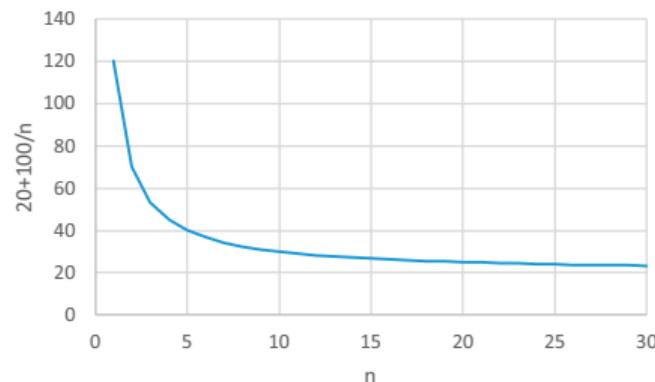
$$g(n) < c * f(n)$$

$$20n + 100 < c * n$$

$$20 + 100/n < c$$

pro $n \geq 1$ platí $100/n < 101$,

zvolíme tedy třeba $n_0 = 1$, $c = 121$



Určení \mathcal{O} -notace

Prosté: najdeme n_0 a c a ukážeme, co se děje:

$$g(n) = 20n + 100$$

$$f(n) = n$$

$$g(n) < c * f(n)$$

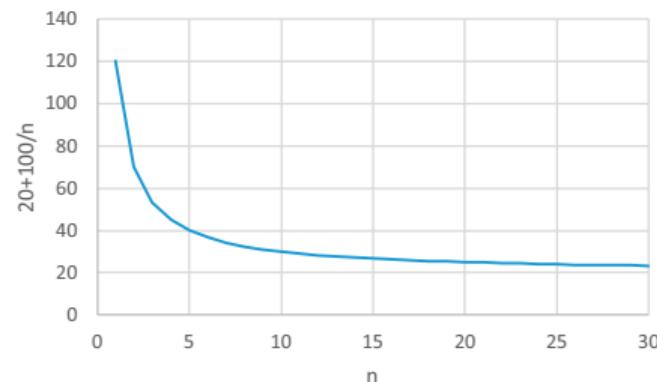
$$20n + 100 < c * n$$

$$20 + 100/n < c$$

pro $n \geq 1$ platí $100/n < 101$,

zvolíme tedy třeba $n_0 = 1$, $c = 121$

(nebo třeba $n_0 = 5$, $c = 41$)



Určení \mathcal{O} -notace

Je ale třeba se nad tím zamyslet:

Určení \mathcal{O} -notace

Je ale třeba se nad tím zamyslet:

$$g(n) = 20n - 10$$

Určení \mathcal{O} -notace

Je ale třeba se nad tím zamyslet:

$$g(n) = 20n - 10$$

$$f(n) = n$$

Určení \mathcal{O} -notace

Je ale třeba se nad tím zamyslet:

$$g(n) = 20n - 10$$

$$f(n) = n$$

$$g(n) < c * f(n)$$

Určení \mathcal{O} -notace

Je ale třeba se nad tím zamyslet:

$$g(n) = 20n - 10$$

$$f(n) = n$$

$$g(n) < c * f(n)$$

$$20n - 10 < c * n$$

Určení \mathcal{O} -notace

Je ale třeba se nad tím zamyslet:

$$g(n) = 20n - 10$$

$$f(n) = n$$

$$g(n) < c * f(n)$$

$$20n - 10 < c * n$$

$$20 - 10/n < c$$

Určení \mathcal{O} -notace

Je ale třeba se nad tím zamyslet:

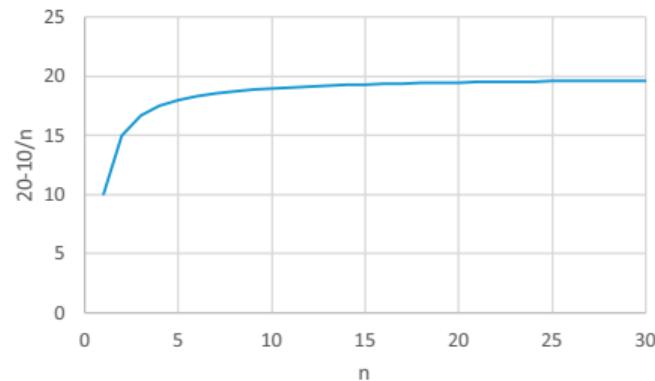
$$g(n) = 20n - 10$$

$$f(n) = n$$

$$g(n) < c * f(n)$$

$$20n - 10 < c * n$$

$$20 - 10/n < c$$



Určení \mathcal{O} -notace

Je ale třeba se nad tím zamyslet:

$$g(n) = 20n - 10$$

$$f(n) = n$$

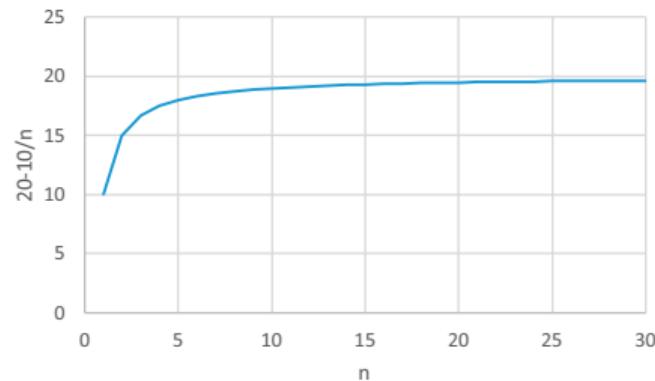
$$g(n) < c * f(n)$$

$$20n - 10 < c * n$$

$$20 - 10/n < c$$

s růstoucím n konverguje $20 - 10/n$ k 20,

pro libovolné n_0 musíme volit $c \geq 20$



Příklad

Patří $g(n) = n * \sqrt{n}$ do $\mathcal{O}(n)$?

Příklad

Patří $g(n) = n * \sqrt{n}$ do $\mathcal{O}(n)$?

$$n * \sqrt{n} < c * n$$

Příklad

Patří $g(n) = n * \sqrt{n}$ do $\mathcal{O}(n)$?

$$n * \sqrt{n} < c * n$$

$$\sqrt{n} < c$$

Příklad

Patří $g(n) = n * \sqrt{n}$ do $\mathcal{O}(n)$?

$$n * \sqrt{n} < c * n$$

$$\sqrt{n} < c$$

Potřebovali bychom konstantu c která bude větší než \sqrt{n} pro všechna $n > n_0$, taková ale neexistuje

Příklad

Patří $g(n) = n * \sqrt{n}$ do $\mathcal{O}(n)$?

$$n * \sqrt{n} < c * n$$

$$\sqrt{n} < c$$

Potřebovali bychom konstantu c která bude větší než \sqrt{n} pro všechna $n > n_0$, taková ale neexistuje

- pro libovolné c vždycky existuje n takové, že $\sqrt{n} > c$ (např. $n = c^2 + 1$)
- podmínka tedy nemůže být splněna

Příklad

Patří $g(n) = n * \sqrt{n}$ do $\mathcal{O}(n)$?

$$n * \sqrt{n} < c * n$$

$$\sqrt{n} < c$$

Potřebovali bychom konstantu c která bude větší než \sqrt{n} pro všechna $n > n_0$, taková ale neexistuje

- pro libovolné c vždycky existuje n takové, že $\sqrt{n} > c$ (např. $n = c^2 + 1$)
- podmínka tedy nemůže být splněna

$g(n) = n * \sqrt{n}$ do $\mathcal{O}(n)$ nepatří.

Interpretace příslušnosti do \mathcal{O} množiny

Výrok $g(n) \in \mathcal{O}(f(n))$ lze interpretovat jako tvrzení o **uspořádání**, tedy $g(n) \preceq f(n)$

- symbol \preceq reprezentuje uspořádání z hlediska rychlosti růstu
- $g(n)$ roste pomaleji nebo stejně rychle jako $f(n)$

Poznámky k \mathcal{O} -notaci

- ve volbě n_0 a c máme velkou svobodu

Poznámky k \mathcal{O} -notaci

- ve volbě n_0 a c máme velkou svobodu
- \mathcal{O} -notace omezuje funkci jen shora, tudíž např. $g(n) = 1 \in \mathcal{O}(n^5)$

Poznámky k \mathcal{O} -notaci

- ve volbě n_0 a c máme velkou svobodu
- \mathcal{O} -notace omezuje funkci jen shora, tudíž např. $g(n) = 1 \in \mathcal{O}(n^5)$
- bývá **zvykem** uvádět „nejtěsnější“ mez, ale někdy to není možné

Poznámky k \mathcal{O} -notaci

- \mathcal{O} -notace neklade žádné podmínky na $f(n)$
 - můžeme tedy vyšetřovat třeba zda $g(n) \in \mathcal{O}(156.12 + 3.14n^2)$
 - nemá to ale moc smysl
 - snažíme se volit $f(n)$ jednoduché

Poznámky k \mathcal{O} -notaci

- \mathcal{O} -notace neklade žádné podmínky na $f(n)$
 - můžeme tedy vyšetřovat třeba zda $g(n) \in \mathcal{O}(156.12 + 3.14n^2)$
 - nemá to ale moc smysl
 - snažíme se volit $f(n)$ jednoduché
- konkrétně, pokud $g(n) \in \mathcal{O}(k * f(n))$, pak $g(n) \in \mathcal{O}(f(n))$, tj. $\mathcal{O}(f(n)) = \mathcal{O}(k * f(n))$

Poznámky k \mathcal{O} -notaci

- \mathcal{O} -notace neklade žádné podmínky na $f(n)$
 - můžeme tedy vyšetřovat třeba zda $g(n) \in \mathcal{O}(156.12 + 3.14n^2)$
 - nemá to ale moc smysl
 - snažíme se volit $f(n)$ jednoduché
- konkrétně, pokud $g(n) \in \mathcal{O}(k * f(n))$, pak $g(n) \in \mathcal{O}(f(n))$, tj. $\mathcal{O}(f(n)) = \mathcal{O}(k * f(n))$
- podobně, pokud $g(n) \in \mathcal{O}(k + f(n))$, pak $g(n) \in \mathcal{O}(f(n))$, tj. $\mathcal{O}(f(n)) = \mathcal{O}(k + f(n))$

Poznámky k \mathcal{O} -notaci

- \mathcal{O} -notace neklade žádné podmínky na $f(n)$
 - můžeme tedy vyšetřovat třeba zda $g(n) \in \mathcal{O}(156.12 + 3.14n^2)$
 - nemá to ale moc smysl
 - snažíme se volit $f(n)$ jednoduché
- konkrétně, pokud $g(n) \in \mathcal{O}(k * f(n))$, pak $g(n) \in \mathcal{O}(f(n))$, tj. $\mathcal{O}(f(n)) = \mathcal{O}(k * f(n))$
- podobně, pokud $g(n) \in \mathcal{O}(k + f(n))$, pak $g(n) \in \mathcal{O}(f(n))$, tj. $\mathcal{O}(f(n)) = \mathcal{O}(k + f(n))$
- rovněž $\mathcal{O}(\log_a(n)) = \mathcal{O}(\log_b(n)/\log_b(a)) = \mathcal{O}(\log_b(n))$ (nezáleží na základu logaritmu)

Poznámky k \mathcal{O} -notaci

- \mathcal{O} -notace neklade žádné podmínky na $f(n)$
 - můžeme tedy vyšetřovat třeba zda $g(n) \in \mathcal{O}(156.12 + 3.14n^2)$
 - nemá to ale moc smysl
 - snažíme se volit $f(n)$ jednoduché
- konkrétně, pokud $g(n) \in \mathcal{O}(k * f(n))$, pak $g(n) \in \mathcal{O}(f(n))$, tj. $\mathcal{O}(f(n)) = \mathcal{O}(k * f(n))$
- podobně, pokud $g(n) \in \mathcal{O}(k + f(n))$, pak $g(n) \in \mathcal{O}(f(n))$, tj. $\mathcal{O}(f(n)) = \mathcal{O}(k + f(n))$
- rovněž $\mathcal{O}(\log_a(n)) = \mathcal{O}(\log_b(n)/\log_b(a)) = \mathcal{O}(\log_b(n))$ (nezáleží na základu logaritmu)

Mezi některými množinami $\mathcal{O}(f(n))$ platí následující vztahy:

$$\mathcal{O}(1) \subset \mathcal{O}(\log(n))$$

Poznámky k \mathcal{O} -notaci

- \mathcal{O} -notace neklade žádné podmínky na $f(n)$
 - můžeme tedy vyšetřovat třeba zda $g(n) \in \mathcal{O}(156.12 + 3.14n^2)$
 - nemá to ale moc smysl
 - snažíme se volit $f(n)$ jednoduché
- konkrétně, pokud $g(n) \in \mathcal{O}(k * f(n))$, pak $g(n) \in \mathcal{O}(f(n))$, tj. $\mathcal{O}(f(n)) = \mathcal{O}(k * f(n))$
- podobně, pokud $g(n) \in \mathcal{O}(k + f(n))$, pak $g(n) \in \mathcal{O}(f(n))$, tj. $\mathcal{O}(f(n)) = \mathcal{O}(k + f(n))$
- rovněž $\mathcal{O}(\log_a(n)) = \mathcal{O}(\log_b(n)/\log_b(a)) = \mathcal{O}(\log_b(n))$ (nezáleží na základu logaritmu)

Mezi některými množinami $\mathcal{O}(f(n))$ platí následující vztahy:

$$\mathcal{O}(1) \subset \mathcal{O}(\log(n)) \subset \mathcal{O}(\sqrt(n))$$

Poznámky k \mathcal{O} -notaci

- \mathcal{O} -notace neklade žádné podmínky na $f(n)$
 - můžeme tedy vyšetřovat třeba zda $g(n) \in \mathcal{O}(156.12 + 3.14n^2)$
 - nemá to ale moc smysl
 - snažíme se volit $f(n)$ jednoduché
- konkrétně, pokud $g(n) \in \mathcal{O}(k * f(n))$, pak $g(n) \in \mathcal{O}(f(n))$, tj. $\mathcal{O}(f(n)) = \mathcal{O}(k * f(n))$
- podobně, pokud $g(n) \in \mathcal{O}(k + f(n))$, pak $g(n) \in \mathcal{O}(f(n))$, tj. $\mathcal{O}(f(n)) = \mathcal{O}(k + f(n))$
- rovněž $\mathcal{O}(\log_a(n)) = \mathcal{O}(\log_b(n)/\log_b(a)) = \mathcal{O}(\log_b(n))$ (nezáleží na základu logaritmu)

Mezi některými množinami $\mathcal{O}(f(n))$ platí následující vztahy:

$$\mathcal{O}(1) \subset \mathcal{O}(\log(n)) \subset \mathcal{O}(\sqrt{n}) \subset \mathcal{O}(n)$$

Poznámky k \mathcal{O} -notaci

- \mathcal{O} -notace neklade žádné podmínky na $f(n)$
 - můžeme tedy vyšetřovat třeba zda $g(n) \in \mathcal{O}(156.12 + 3.14n^2)$
 - nemá to ale moc smysl
 - snažíme se volit $f(n)$ jednoduché
- konkrétně, pokud $g(n) \in \mathcal{O}(k * f(n))$, pak $g(n) \in \mathcal{O}(f(n))$, tj. $\mathcal{O}(f(n)) = \mathcal{O}(k * f(n))$
- podobně, pokud $g(n) \in \mathcal{O}(k + f(n))$, pak $g(n) \in \mathcal{O}(f(n))$, tj. $\mathcal{O}(f(n)) = \mathcal{O}(k + f(n))$
- rovněž $\mathcal{O}(\log_a(n)) = \mathcal{O}(\log_b(n)/\log_b(a)) = \mathcal{O}(\log_b(n))$ (nezáleží na základu logaritmu)

Mezi některými množinami $\mathcal{O}(f(n))$ platí následující vztahy:

$$\mathcal{O}(1) \subset \mathcal{O}(\log(n)) \subset \mathcal{O}(\sqrt{n}) \subset \mathcal{O}(n) \subset \mathcal{O}(n \log(n))$$

Poznámky k \mathcal{O} -notaci

- \mathcal{O} -notace neklade žádné podmínky na $f(n)$
 - můžeme tedy vyšetřovat třeba zda $g(n) \in \mathcal{O}(156.12 + 3.14n^2)$
 - nemá to ale moc smysl
 - snažíme se volit $f(n)$ jednoduché
- konkrétně, pokud $g(n) \in \mathcal{O}(k * f(n))$, pak $g(n) \in \mathcal{O}(f(n))$, tj. $\mathcal{O}(f(n)) = \mathcal{O}(k * f(n))$
- podobně, pokud $g(n) \in \mathcal{O}(k + f(n))$, pak $g(n) \in \mathcal{O}(f(n))$, tj. $\mathcal{O}(f(n)) = \mathcal{O}(k + f(n))$
- rovněž $\mathcal{O}(\log_a(n)) = \mathcal{O}(\log_b(n)/\log_b(a)) = \mathcal{O}(\log_b(n))$ (nezáleží na základu logaritmu)

Mezi některými množinami $\mathcal{O}(f(n))$ platí následující vztahy:

$$\mathcal{O}(1) \subset \mathcal{O}(\log(n)) \subset \mathcal{O}(\sqrt{n}) \subset \mathcal{O}(n) \subset \mathcal{O}(n \log(n)) \subset \mathcal{O}(n^2)$$

Poznámky k \mathcal{O} -notaci

- \mathcal{O} -notace neklade žádné podmínky na $f(n)$
 - můžeme tedy vyšetřovat třeba zda $g(n) \in \mathcal{O}(156.12 + 3.14n^2)$
 - nemá to ale moc smysl
 - snažíme se volit $f(n)$ jednoduché
- konkrétně, pokud $g(n) \in \mathcal{O}(k * f(n))$, pak $g(n) \in \mathcal{O}(f(n))$, tj. $\mathcal{O}(f(n)) = \mathcal{O}(k * f(n))$
- podobně, pokud $g(n) \in \mathcal{O}(k + f(n))$, pak $g(n) \in \mathcal{O}(f(n))$, tj. $\mathcal{O}(f(n)) = \mathcal{O}(k + f(n))$
- rovněž $\mathcal{O}(\log_a(n)) = \mathcal{O}(\log_b(n)/\log_b(a)) = \mathcal{O}(\log_b(n))$ (nezáleží na základu logaritmu)

Mezi některými množinami $\mathcal{O}(f(n))$ platí následující vztahy:

$$\mathcal{O}(1) \subset \mathcal{O}(\log(n)) \subset \mathcal{O}(\sqrt{n}) \subset \mathcal{O}(n) \subset \mathcal{O}(n \log(n)) \subset \mathcal{O}(n^2) \subset \mathcal{O}(n^3)$$

Poznámky k \mathcal{O} -notaci

- \mathcal{O} -notace neklade žádné podmínky na $f(n)$
 - můžeme tedy vyšetřovat třeba zda $g(n) \in \mathcal{O}(156.12 + 3.14n^2)$
 - nemá to ale moc smysl
 - snažíme se volit $f(n)$ jednoduché
- konkrétně, pokud $g(n) \in \mathcal{O}(k * f(n))$, pak $g(n) \in \mathcal{O}(f(n))$, tj. $\mathcal{O}(f(n)) = \mathcal{O}(k * f(n))$
- podobně, pokud $g(n) \in \mathcal{O}(k + f(n))$, pak $g(n) \in \mathcal{O}(f(n))$, tj. $\mathcal{O}(f(n)) = \mathcal{O}(k + f(n))$
- rovněž $\mathcal{O}(\log_a(n)) = \mathcal{O}(\log_b(n)/\log_b(a)) = \mathcal{O}(\log_b(n))$ (nezáleží na základu logaritmu)

Mezi některými množinami $\mathcal{O}(f(n))$ platí následující vztahy:

$$\mathcal{O}(1) \subset \mathcal{O}(\log(n)) \subset \mathcal{O}(\sqrt{n}) \subset \mathcal{O}(n) \subset \mathcal{O}(n \log(n)) \subset \mathcal{O}(n^2) \subset \mathcal{O}(n^3) \subset \mathcal{O}(2^n)$$

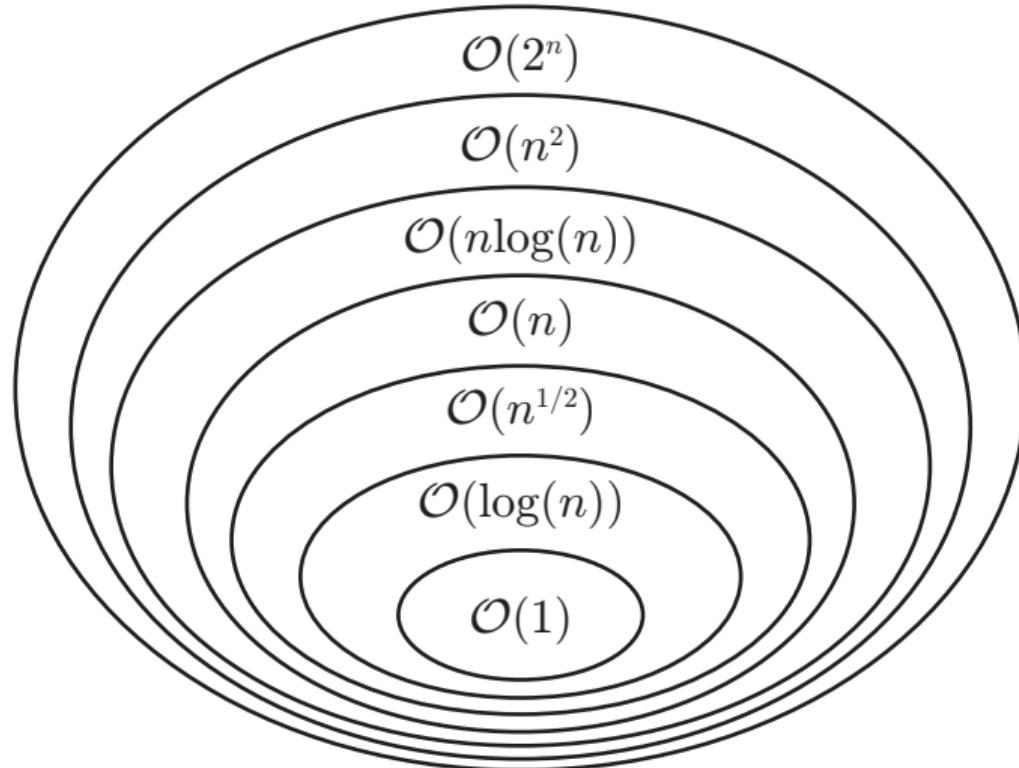
Poznámky k \mathcal{O} -notaci

- \mathcal{O} -notace neklade žádné podmínky na $f(n)$
 - můžeme tedy vyšetřovat třeba zda $g(n) \in \mathcal{O}(156.12 + 3.14n^2)$
 - nemá to ale moc smysl
 - snažíme se volit $f(n)$ jednoduché
- konkrétně, pokud $g(n) \in \mathcal{O}(k * f(n))$, pak $g(n) \in \mathcal{O}(f(n))$, tj. $\mathcal{O}(f(n)) = \mathcal{O}(k * f(n))$
- podobně, pokud $g(n) \in \mathcal{O}(k + f(n))$, pak $g(n) \in \mathcal{O}(f(n))$, tj. $\mathcal{O}(f(n)) = \mathcal{O}(k + f(n))$
- rovněž $\mathcal{O}(\log_a(n)) = \mathcal{O}(\log_b(n)/\log_b(a)) = \mathcal{O}(\log_b(n))$ (nezáleží na základu logaritmu)

Mezi některými množinami $\mathcal{O}(f(n))$ platí následující vztahy:

$$\mathcal{O}(1) \subset \mathcal{O}(\log(n)) \subset \mathcal{O}(\sqrt{n}) \subset \mathcal{O}(n) \subset \mathcal{O}(n \log(n)) \subset \mathcal{O}(n^2) \subset \mathcal{O}(n^3) \subset \mathcal{O}(2^n) \subset \mathcal{O}(e^n)$$

Inkluze \mathcal{O} množin



Definice

$\Omega(f(n))$ označíme množinu všech funkcí $g(n)$ takových, pro které platí, že $g(n) > c * f(n)$ pro všechna $n > n_0 > 0$ a pro nějaké $c > 0$.

Omega notace

Definice

$\Omega(f(n))$ označíme množinu všech funkcí $g(n)$ takových, pro které platí, že $g(n) > c * f(n)$ pro všechna $n > n_0 > 0$ a pro nějaké $c > 0$.

Co to znamená?

- Omega notace je opakem \mathcal{O} -notace
- funkce $g(n)$ patří do $\Omega(f(n))$ když $g(n)$ „roste stejně rychle nebo rychleji než $f(n)$ “

Omega notace

Definice

$\Omega(f(n))$ označíme množinu všech funkcí $g(n)$ takových, pro které platí, že $g(n) > c * f(n)$ pro všechna $n > n_0 > 0$ a pro nějaké $c > 0$.

Co to znamená?

- Omega notace je opakem \mathcal{O} -notace
- funkce $g(n)$ patří do $\Omega(f(n))$ když $g(n)$ „roste stejně rychle nebo rychleji než $f(n)$ “

Jak ukázat, že $g(n)$ patří do $\Omega(f(n))$

- stejně jako s \mathcal{O} -notací: najít n_0 a c

Interpretace příslušnosti do Ω množiny

Výrok $g(n) \in \Omega(f(n))$ také lze interpretovat jako tvrzení o **uspořádání**, tentokrát $g(n) \succeq f(n)$

- symbol \succeq opět reprezentuje uspořádání z hlediska rychlosti růstu
- $g(n)$ roste rychleji nebo stejně rychle jako $f(n)$

Omega notace

Mezi Ω množinami existují obrácené vztahy:

$$\Omega(e^n) \subset \Omega(2^n) \subset \Omega(n^3) \subset \Omega(n^2) \subset \Omega(n \log(n)) \subset \Omega(n) \subset \Omega(\sqrt{n}) \subset \Omega(\log(n)) \subset \Omega(1)$$

Theta notace

Definice

Pokud $g(n) \in \mathcal{O}(f(n))$ a zároveň $g(n) \in \Omega(f(n))$, pak $g(n) \in \Theta(f(n))$

Definice

Pokud $g(n) \in \mathcal{O}(f(n))$ a zároveň $g(n) \in \Omega(f(n))$, pak $g(n) \in \Theta(f(n))$

- $g(n)$ neroste ani rychleji ani pomaleji než $f(n)$, roste stejně rychle

Theta notace

Definice

Pokud $g(n) \in \mathcal{O}(f(n))$ a zároveň $g(n) \in \Omega(f(n))$, pak $g(n) \in \Theta(f(n))$

- $g(n)$ neroste ani rychleji ani pomaleji než $f(n)$, roste stejně rychle
- graf $g(n)$ je od jistého n_0 možné **uzavřít mezi** grafy $c_1 * f(n)$ a $c_2 * f(n)$

Definice

Pokud $g(n) \in \mathcal{O}(f(n))$ a zároveň $g(n) \in \Omega(f(n))$, pak $g(n) \in \Theta(f(n))$

- $g(n)$ neroste ani rychleji ani pomaleji než $f(n)$, roste stejně rychle
- graf $g(n)$ je od jistého n_0 možné **uzavřít mezi** grafy $c_1 * f(n)$ a $c_2 * f(n)$
- patrně $c_1 < c_2$

Definice

Pokud $g(n) \in \mathcal{O}(f(n))$ a zároveň $g(n) \in \Omega(f(n))$, pak $g(n) \in \Theta(f(n))$

- $g(n)$ neroste ani rychleji ani pomaleji než $f(n)$, roste stejně rychle
- graf $g(n)$ je od jistého n_0 možné **uzavřít mezi** grafy $c_1 * f(n)$ a $c_2 * f(n)$
- patrně $c_1 < c_2$
- množiny $\Theta(e^n)$, $\Theta(2^n)$, $\Theta(n^3)$, $\Theta(n^2)$, $\Theta(n \log(n))$, $\Theta(n)$, $\Theta(\sqrt{n})$, $\Theta(\log(n))$, $\Theta(1)$ jsou **disjunktní** - není žádná $g(x)$, která by patřila do dvou z nich současně

Důležitá poznámka

- $\mathcal{O}(f(n))$, $\Omega(f(n))$ a $\Theta(f(n))$ jsou množiny funkcí, neříkali jsme jaký mají funkce **význam!**

Důležitá poznámka

- $\mathcal{O}(f(n))$, $\Omega(f(n))$ a $\Theta(f(n))$ jsou množiny funkcí, neříkali jsme jaký mají funkce **význam!**
- Může to být
 - čas výpočtu pro jakýkoli vstup velikosti n
 - čas výpočtu pro nejlepší možný vstup velikosti n
 - čas výpočtu pro nejhorší možný vstup velikosti n
 - průměrný čas výpočtu pro vstup velikosti n (průměr přes všechny možné vstupy velikosti n)
 - počet instrukcí pro ...
 - množství paměti nutné pro zpracování ...
 - ...
- je třeba **nesrovnávat hrušky s pomeranči**

Postup při analýze složitosti

Algoritmus/program

```
bool Contains(int[] data, int x) {  
    for (int i = 0;i<data.Length;i++) {  
        if (data[i] == x)  
            return true;  
        if (data[i]>x)  
            return false;  
    }  
    return false;  
}
```

Postup při analýze složitosti

Algoritmus/program

```
bool Contains(int[] data, int x) {  
    for (int i = 0; i < data.Length; i++) {  
        if (data[i] == x)  
            return true;  
        if (data[i] > x)  
            return false;  
    }  
    return false;  
}
```



Funkce

$$g(n) = \frac{n(n-1)}{2}$$

$$g(n) = \frac{n(n^2 + 1)}{4}$$

Postup při analýze složitosti

Algoritmus/program

```
bool Contains(int[] data, int x) {  
    for (int i = 0; i < data.Length; i++) {  
        if (data[i] == x)  
            return true;  
        if (data[i] > x)  
            return false;  
    }  
    return false;  
}
```

Funkce

$$g(n) = \frac{n(n-1)}{2}$$

$$g(n) = \frac{n(n^2 + 1)}{4}$$



Odvození funkce

Možnosti:

- nejlepší/nejlepší/průměrný případ, ...
- počet operací, počet vytvořených instancí, ...

Postup při analýze složitosti

Algoritmus/program

```
bool Contains(int[] data, int x) {  
    for (int i = 0; i < data.Length; i++) {  
        if (data[i] == x)  
            return true;  
        if (data[i] > x)  
            return false;  
    }  
    return false;  
}
```

Funkce

$$g(n) = \frac{n(n-1)}{2}$$

$$g(n) = \frac{n(n^2+1)}{4}$$

Výrok o funkci

$$g(n) \in \Omega(f(n))$$

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

Odrození funkce

Možnosti:

- nejlepší/nejlepší/průměrný případ, ...
- počet operací, počet vytvořených instancí, ...

Postup při analýze složitosti

```
bool Contains(int[] data, int x) {  
    for (int i = 0; i < data.Length; i++) {  
        if (data[i] == x)  
            return true;  
        if (data[i] > x)  
            return false;  
    }  
    return false;  
}
```

Algoritmus/program

Funkce

Výrok o funkci

$$g(n) = \frac{n(n-1)}{2}$$

$$g(n) = \frac{n(n^2+1)}{4}$$

$$g(n) \in \Omega(f(n))$$

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

Odvození funkce

Možnosti:

- nejlepší/nejlepší/průměrný případ, ...
- počet operací, počet vytvořených instancí, ...

Důkaz příslušnosti

Možnosti:

- O, Ω, Θ
- Parametr množiny

Význam výroku o funkci

V computer science funkce obvykle popisuje nějaké **náklady** (čas, paměť, ...) v závislosti na velikosti vstupu.

Význam výroku o funkci

V computer science funkce obvykle popisuje nějaké **náklady** (čas, paměť, ...) v závislosti na velikosti vstupu.

- **efektivní** algoritmy jsou popsány **pomalu** rostoucími funkcemi

Význam výroku o funkci

V computer science funkce obvykle popisuje nějaké **náklady** (čas, paměť, ...) v závislosti na velikosti vstupu.

- **efektivní** algoritmy jsou popsány **pomalu** rostoucími funkcemi
- **neefektivní** algoritmy jsou popsány **rychle** rostoucími funkcemi

Význam výroku o funkci

V computer science funkce obvykle popisuje nějaké **náklady** (čas, paměť, ...) v závislosti na velikosti vstupu.

- **efektivní** algoritmy jsou popsány **pomalu** rostoucími funkcemi
- **neefektivní** algoritmy jsou popsány **rychle** rostoucími funkcemi

Výrok o funkci popisuje, zda roste pomalu, nebo rychle

Význam výroku o funkci

V computer science funkce obvykle popisuje nějaké **náklady** (čas, paměť, ...) v závislosti na velikosti vstupu.

- **efektivní** algoritmy jsou popsány **pomalu** rostoucími funkcemi
- **neefektivní** algoritmy jsou popsány **rychle** rostoucími funkcemi

Výrok o funkci popisuje, zda roste pomalu, nebo rychle

Je třeba zvolit výrok, který má příslušný význam

\mathcal{O} notace

- příslušnost do $\mathcal{O}(f(n))$ indikuje efektivitu, pokud $f(n)$ roste dost pomalu

\mathcal{O} notace

- příslušnost do $\mathcal{O}(f(n))$ **indikuje efektivitu, pokud** $f(n)$ roste dost pomalu
- příslušnost do $\mathcal{O}(f(n))$, kde $f(n)$ roste rychle, **neindikuje neefektivitu**: je možné, že daná funkce patří zároveň do nějaké jiné $\mathcal{O}(f(n))$

\mathcal{O} notace

- příslušnost do $\mathcal{O}(f(n))$ **indikuje efektivitu, pokud** $f(n)$ roste dost pomalu
- příslušnost do $\mathcal{O}(f(n))$, kde $f(n)$ roste rychle, **neindikuje neefektivitu**: je možné, že daná funkce patří zároveň do nějaké jiné $\mathcal{O}(f(n))$
- \mathcal{O} -notace je záruka (nebude to horší než ...)

Výroky o funkcích v souvislosti s efektivitou

\mathcal{O} notace

- příslušnost do $\mathcal{O}(f(n))$ indikuje efektivitu, pokud $f(n)$ roste dost pomalu
- příslušnost do $\mathcal{O}(f(n))$, kde $f(n)$ roste rychle, neindikuje neefektivitu: je možné, že daná funkce patří zároveň do nějaké jiné $\mathcal{O}(f(n))$
- \mathcal{O} -notace je záruka (nebude to horší než ...)

Ω notace

- příslušnost do $\Omega(f(n))$ indikuje neefektivitu, pokud $f(n)$ roste rychle

\mathcal{O} notace

- příslušnost do $\mathcal{O}(f(n))$ **indikuje efektivitu, pokud $f(n)$ roste dost pomalu**
- příslušnost do $\mathcal{O}(f(n))$, kde $f(n)$ roste rychle, **neindikuje neefektivitu**: je možné, že daná funkce patří zároveň do nějaké jiné $\mathcal{O}(f(n))$
- \mathcal{O} -notace je záruka (nebude to horší než ...)

Ω notace

- příslušnost do $\Omega(f(n))$ **indikuje neefektivitu, pokud $f(n)$ roste rychle**
- příslušnost do $\Omega(f(n))$, kde $f(n)$ roste pomalu, **neindikuje efektivitu**: je možné, že daná funkce patří zároveň do nějaké jiné $\Omega(f(n))$

\mathcal{O} notace

- příslušnost do $\mathcal{O}(f(n))$ **indikuje efektivitu, pokud $f(n)$ roste dost pomalu**
- příslušnost do $\mathcal{O}(f(n))$, kde $f(n)$ roste rychle, **neindikuje neefektivitu**: je možné, že daná funkce patří zároveň do nějaké jiné $\mathcal{O}(f(n))$
- \mathcal{O} -notace je záruka (nebude to horší než ...)

Ω notace

- příslušnost do $\Omega(f(n))$ **indikuje neefektivitu, pokud $f(n)$ roste rychle**
- příslušnost do $\Omega(f(n))$, kde $f(n)$ roste pomalu, **neindikuje efektivitu**: je možné, že daná funkce patří zároveň do nějaké jiné $\Omega(f(n))$
- Ω -notace říká "nebude to lepší než ..."- může to ale být ještě horší!

Častá chyba

Výrok

Algoritmus je pomalý, protože jeho složitost je $\mathcal{O}(n^3)$.

Častá chyba

Výrok

Algoritmus je pomalý, protože jeho složitost je $\mathcal{O}(n^3)$.

Ekvivalentní výrok

Algoritmus je pomalý, protože jeho funkce nákladů patří do $\mathcal{O}(n^3)$.

Častá chyba

Výrok

Algoritmus je pomalý, protože jeho složitost je $\mathcal{O}(n^3)$.

Ekvivalentní výrok

Algoritmus je pomalý, protože jeho funkce nákladů patří do $\mathcal{O}(n^3)$.

Ekvivalentní výrok

Funkce nákladů algoritmu roste rychle, protože jeho funkce nákladů patří do $\mathcal{O}(n^3)$.

Častá chyba

Výrok

Algoritmus je pomalý, protože jeho složitost je $\mathcal{O}(n^3)$.

Ekvivalentní výrok

Algoritmus je pomalý, protože jeho funkce nákladů patří do $\mathcal{O}(n^3)$.

Ekvivalentní výrok

Funkce nákladů algoritmu roste rychle, protože jeho funkce nákladů patří do $\mathcal{O}(n^3)$.

Ekvivalentní výrok

Funkce nákladů algoritmu roste rychle, protože roste **pomaleji** než n^3 (nebo stejně).

Poznámky k Θ notaci

- příslušnost $g(n)$ do $\Theta(f(n))$ docela dobře popisuje chování $g(n)$

Poznámky k Θ notaci

- příslušnost $g(n)$ do $\Theta(f(n))$ docela dobře popisuje chování $g(n)$
- efektivita se dá seřadit podle pořadí odpovídajících $\mathcal{O}(f(n))$ množin

Poznámky k Θ notaci

- příslušnost $g(n)$ do $\Theta(f(n))$ docela dobře popisuje chování $g(n)$
- efektivita se dá seřadit podle pořadí odpovídajících $\mathcal{O}(f(n))$ množin
- např. pokud $g_1(n)$ a $g_2(n)$ popisují čas výpočtu dvou algoritmů, a $g_1(n) \in \Theta(n^2)$ a $g_2(n) \in \Theta(n)$, pak se dá předpokládat, že algoritmus s časem výpočtu $g_2(n)$ bude efektivnější

Poznámky k Θ notaci

- příslušnost $g(n)$ do $\Theta(f(n))$ docela dobře popisuje chování $g(n)$
- efektivita se dá seřadit podle pořadí odpovídajících $\mathcal{O}(f(n))$ množin
- např. pokud $g_1(n)$ a $g_2(n)$ popisují čas výpočtu dvou algoritmů, a $g_1(n) \in \Theta(n^2)$ a $g_2(n) \in \Theta(n)$, pak se dá předpokládat, že algoritmus s časem výpočtu $g_2(n)$ bude efektivnější
- ALE: pokud $g_1(n) = 1000n$, $g_2(n) = n\log_2(n)$, pak pro všechna realistická n platí $g_2(n) < g_1(n)$, přestože $g_2(n) \in \Theta(n\log(n))$ a $g_1(n) \in \Theta(n)$

Praktické určení výpočetní složitosti

je třeba určit funkci $g(n)$

```
int Example(int n) {  
    int result;  
    for (int i = 0; i < n; i++)  
        for (int j = 0; j < 3*n; j++)  
            result += Math.Sin(i*j);  
    return result;  
}
```

Praktické určení výpočetní složitosti

je třeba určit funkci $g(n)$

```
int Example(int n) {  
    int result;  
    for (int i = 0; i < n; i++)  
        for (int j = 0; j < 3*n; j++)  
            result += Math.Sin(i*j);  
    return result;  
}
```

- čas výpočtu $\sin(x)$ je t_1 , ostatní můžeme (tentokrát!) zanedbat

Praktické určení výpočetní složitosti

je třeba určit funkci $g(n)$

```
int Example(int n) {  
    int result;  
    for (int i = 0; i < n; i++)  
        for (int j = 0; j < 3*n; j++)  
            result += Math.Sin(i*j);  
    return result;  
}
```

- čas výpočtu $\sin(x)$ je t_1 , ostatní můžeme (tentokrát!) zanedbat
- $g(n) = 3 * n^2 * t_1$

Praktické určení výpočetní složitosti

je třeba určit funkci $g(n)$

```
int Example(int n) {  
    int result;  
    for (int i = 0; i < n; i++)  
        for (int j = 0; j < 3*n; j++)  
            result += Math.Sin(i*j);  
    return result;  
}
```

- čas výpočtu $\sin(x)$ je t_1 , ostatní můžeme (tentokrát!) zanedbat
- $g(n) = 3 * n^2 * t_1$
- nejlepší, nejhorší i očekávaný čas je stejný

Praktické určení výpočetní složitosti

je třeba určit funkci $g(n)$

```
int Example(int n) {  
    int result;  
    for (int i = 0; i < n; i++)  
        for (int j = 0; j < 3*n; j++)  
            result += Math.Sin(i*j);  
    return result;  
}
```

- čas výpočtu $\sin(x)$ je t_1 , ostatní můžeme (tentokrát!) zanedbat
- $g(n) = 3 * n^2 * t_1$
- nejlepší, nejhorší i očekávaný čas je stejný
- množina $\Theta(n^2)$

Praktické určení výpočetní složitosti

```
int Example2(int n) {  
    int result;  
    for (int i = 0; i < n; i++)  
        result += M1(i);  
  
    for (int i = 0; i < n; i++)  
        for (int j = 0; j < 3*n; j++)  
            result += M2(i*j);  
    return result;  
}
```

Praktické určení výpočetní složitosti

```
int Example2(int n) {  
    int result;  
    for (int i = 0; i < n; i++)  
        result += M1(i);  
  
    for (int i = 0; i < n; i++)  
        for (int j = 0; j < 3*n; j++)  
            result += M2(i*j);  
    return result;  
}
```

- čas běhu $M1(x)$ je t_1 , čas běhu $M2(x)$ je $t_1 * 0.000001$, ostatní opět zanedbáme

Praktické určení výpočetní složitosti

```
int Example2(int n) {  
    int result;  
    for (int i = 0; i < n; i++)  
        result += M1(i);  
  
    for (int i = 0; i < n; i++)  
        for (int j = 0; j < 3*n; j++)  
            result += M2(i*j);  
    return result;  
}
```

- čas běhu $M1(x)$ je t_1 , čas běhu $M2(x)$ je $t_1 * 0.000001$, ostatní opět zanedbáme
- $g(n) = n * t_1 + 3 * n^2 * t_1 * 0.000001$

Praktické určení výpočetní složitosti

```
int Example2(int n) {  
    int result;  
    for (int i = 0; i < n; i++)  
        result += M1(i);  
  
    for (int i = 0; i < n; i++)  
        for (int j = 0; j < 3*n; j++)  
            result += M2(i*j);  
    return result;  
}
```

- čas běhu $M1(x)$ je t_1 , čas běhu $M2(x)$ je $t_1 * 0.000001$, ostatní opět zanedbáme
- $g(n) = n * t_1 + 3 * n^2 * t_1 * 0.000001$
- výpočetní složitost je opět $\Theta(n^2)$

Praktické určení výpočetní složitosti

- časy jednotlivých instrukcí můžeme ignorovat - množina složitosti bude stejná

Praktické určení výpočetní složitosti

- časy jednotlivých instrukcí můžeme ignorovat - množina složitosti bude stejná
- můžeme tedy stejně dobře počítat jen počet instrukcí (čas instrukce = 1)

- časy jednotlivých instrukcí můžeme ignorovat - množina složitosti bude stejná
- můžeme tedy stejně dobře počítat jen počet instrukcí (čas instrukce = 1)
- ale jen tehdy, když je čas instrukce opravdu konstantní!

- časy jednotlivých instrukcí můžeme ignorovat - množina složitosti bude stejná
- můžeme tedy stejně dobře počítat jen počet instrukcí (čas instrukce = 1)
- ale jen tehdy, když je **čas instrukce opravdu konstantní!**
- výpočetní složitost stírá rozdíly mezi počítači: je jedno, zda dělení dvou čísel trvá nanosekundu nebo sto nanosekund

Banální příklad

kolikrát se vykoná metoda DoHardWork () ?

```
for(int i = 0; i < n; i++)
    for(int j = 0; j < 2*n; j++)
        DoHardWork(i, j);
```

Banální příklad

kolikrát se vykoná metoda DoHardWork () ?

```
for(int i = 0; i < n; i++)  
    for(int j = 0; j < 2*n; j++)  
        DoHardWork(i, j);
```

$$g(n) = 2n * n = 2n^2$$

Banální příklad

kolikrát se vykoná metoda DoHardWork () ?

```
for(int i = 0; i < n; i++)  
    for(int j = 0; j < 2*n; j++)  
        DoHardWork(i, j);
```

$$g(n) = 2n * n = 2n^2 \in \Theta(n^2)$$

Další banální příklad

kolikrát se vykoná metoda DoHardWork () ?

```
for(int i = 0; i<n; i++) {  
    DoHardWork(i, 0);  
    for(int j = 0; j<2*n; j++)  
        DoHardWork(i, j);  
}
```

Další banální příklad

kolikrát se vykoná metoda DoHardWork () ?

```
for(int i = 0; i<n; i++) {  
    DoHardWork(i, 0);  
    for(int j = 0; j<2*n; j++)  
        DoHardWork(i, j);  
}
```

$$g(n) = (2n + 1)n = 2n^2 + n$$

Další banální příklad

kolikrát se vykoná metoda DoHardWork () ?

```
for(int i = 0; i<n; i++) {  
    DoHardWork(i, 0);  
    for(int j = 0; j<2*n; j++)  
        DoHardWork(i, j);  
}
```

$$g(n) = (2n + 1)n = 2n^2 + n \in \Theta(n^2)$$

Trochu méně banální příklad

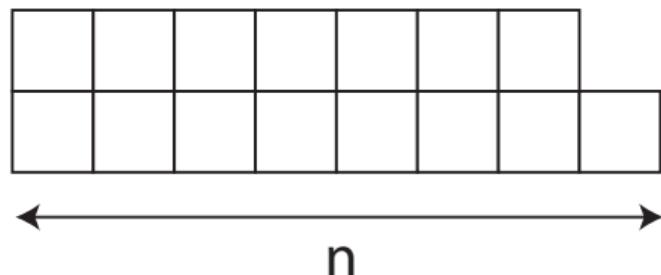
kolikrát se vykoná metoda DoHardWork () ?

```
for(int i = 0; i < n; i++) {  
    for(int j = i; j < n; j++)  
        DoHardWork(i, j);  
}
```

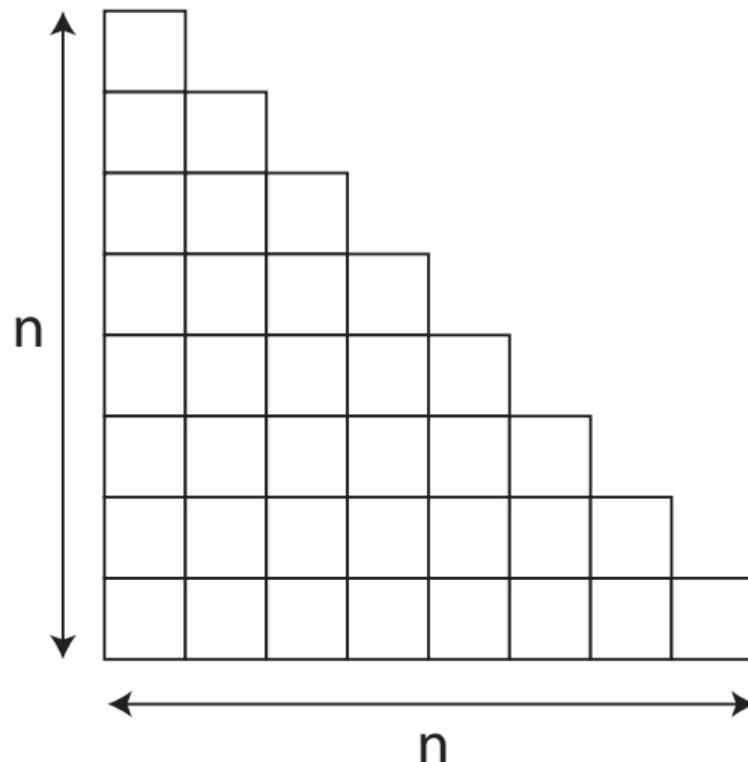
Geometrická intuice



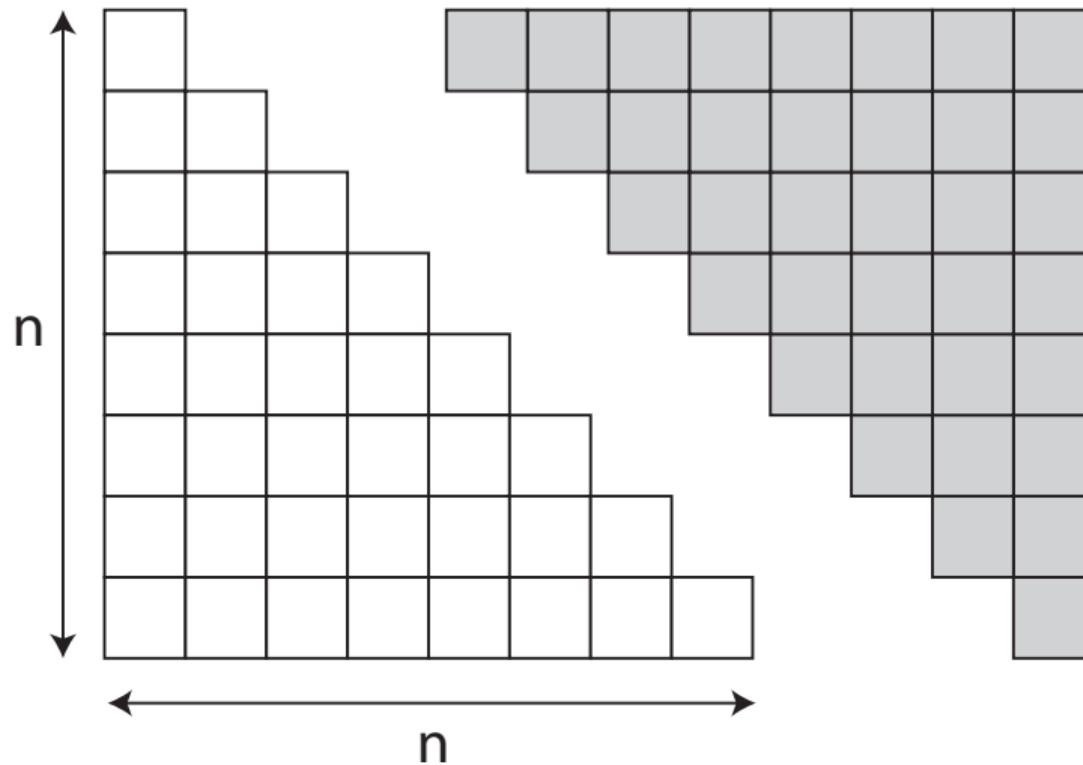
Geometrická intuice



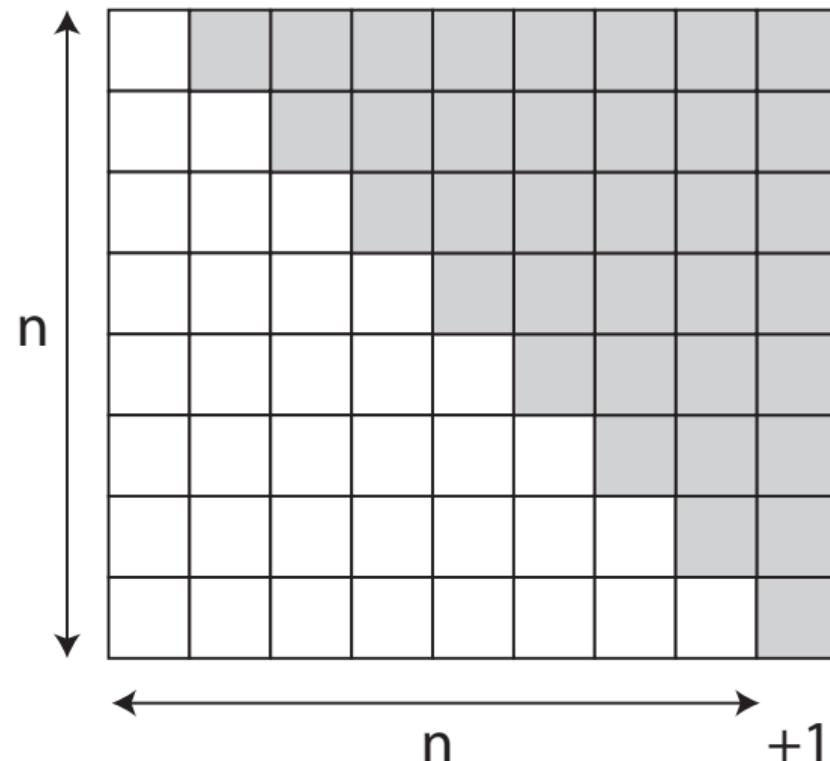
Geometrická intuice



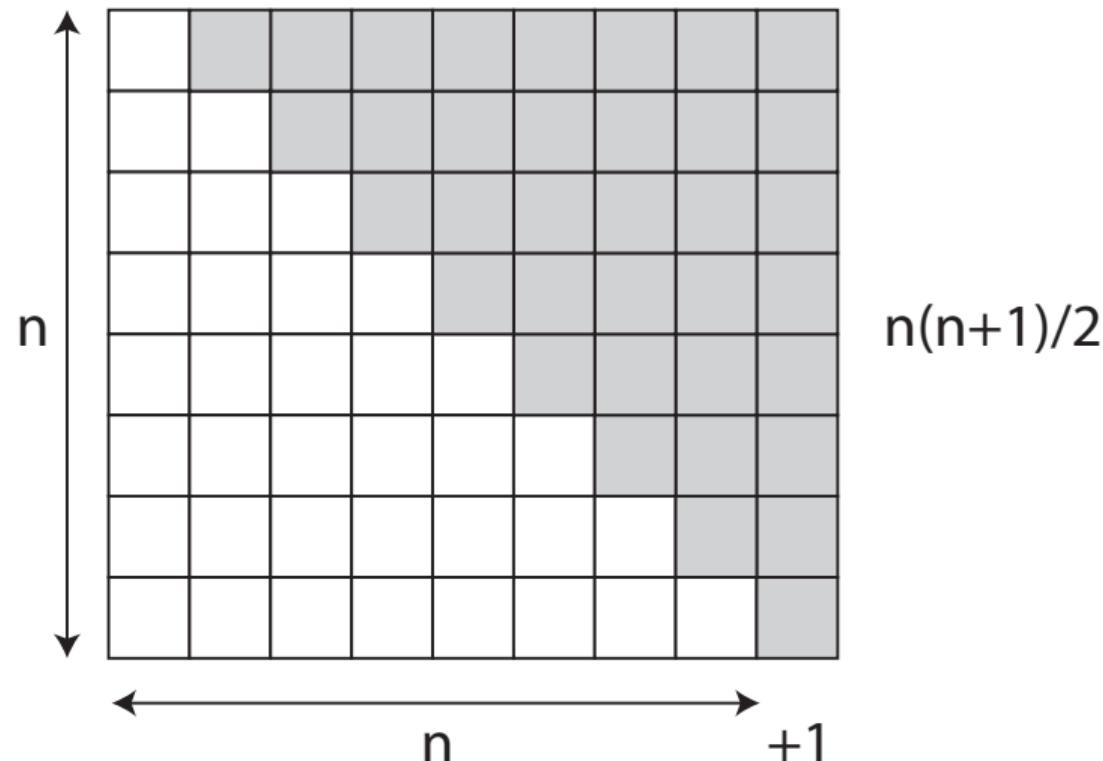
Geometrická intuice



Geometrická intuice



Geometrická intuice



Opět banální příklad

kolikrát se vykoná metoda DoHardWork?

```
for(int i = 0; i < n; i++) {  
    for(int j = i+1; j < n; j++)  
        DoHardWork(i, j);  
}
```

Opět banální příklad

kolikrát se vykoná metoda DoHardWork?

```
for(int i = 0; i < n; i++) {  
    for(int j = i+1; j < n; j++)  
        DoHardWork(i, j);  
}
```

$$g(n) = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n$$

Opět banální příklad

kolikrát se vykoná metoda DoHardWork?

```
for(int i = 0; i < n; i++) {  
    for(int j = i+1; j < n; j++)  
        DoHardWork(i, j);  
}
```

$$g(n) = \frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n \in \Theta(n^2)$$

Ještě jeden banální příklad

kolikrát se vykoná metoda DoHardWork?

```
int c = n;
while (c>0) {
    for (int j = 0; j<c; j++)
        DoHardWork (c, j);
    c = c-1;
}
```

Ještě jeden banální příklad

kolikrát se vykoná metoda DoHardWork?

```
int c = n;  
while (c>0) {  
    for (int j = 0; j<c; j++)  
        DoHardWork (c, j);  
    c = c-1;  
}
```

$$g(n) = \frac{n(n+1)}{2}$$

Ještě jeden banální příklad

kolikrát se vykoná metoda DoHardWork?

```
int c = n;  
while (c>0) {  
    for (int j = 0; j<c; j++)  
        DoHardWork (c, j);  
    c = c-1;  
}
```

$$g(n) = \frac{n(n+1)}{2} \in \Theta(n^2)$$

Zcela neobanální a téměř sofistikovaný příklad

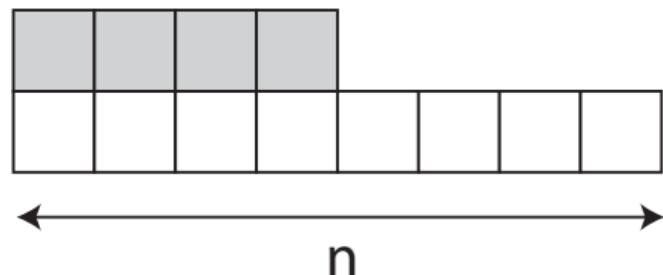
kolikrát se vykoná metoda DoHardWork?

```
int c = n;
while(c>0) {
    for(int j = 0; j<c; j++)
        DoHardWork(c, j);
    c = c/2;
}
```

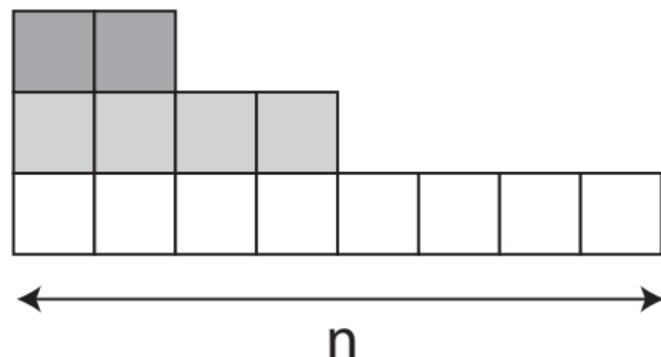
Geometrická intuice



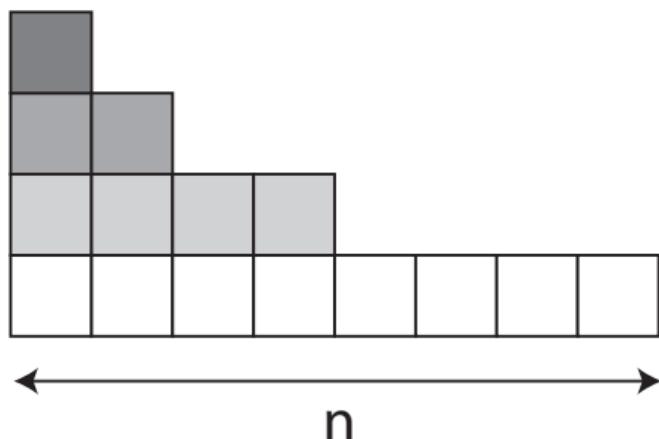
Geometrická intuice



Geometrická intuice



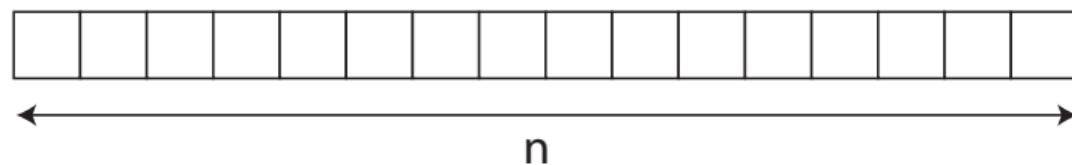
Geometrická intuice



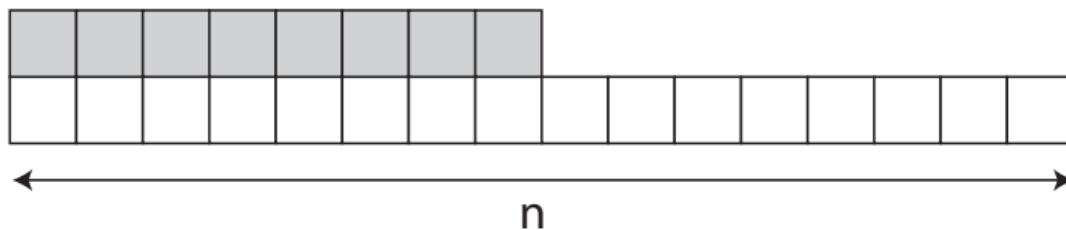
Geometrická intuice



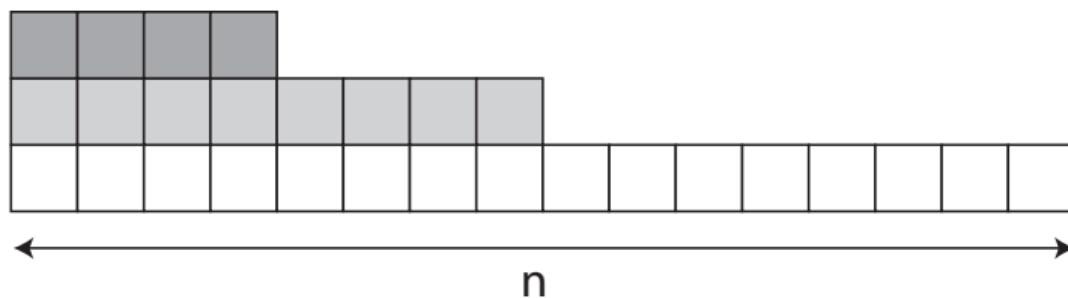
Větší geometrická intuice



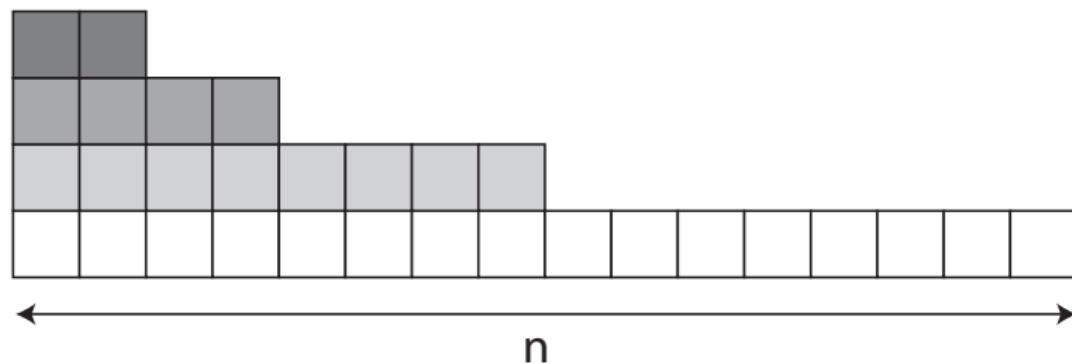
Větší geometrická intuice



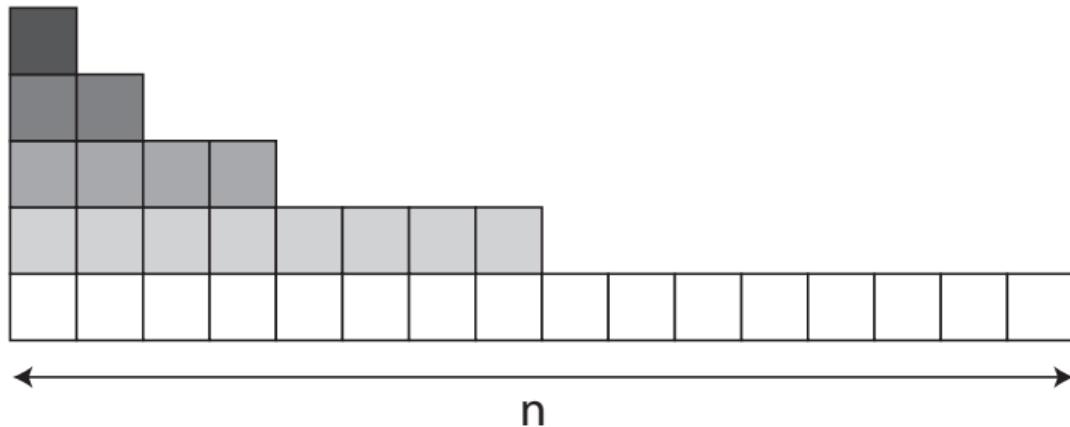
Větší geometrická intuice



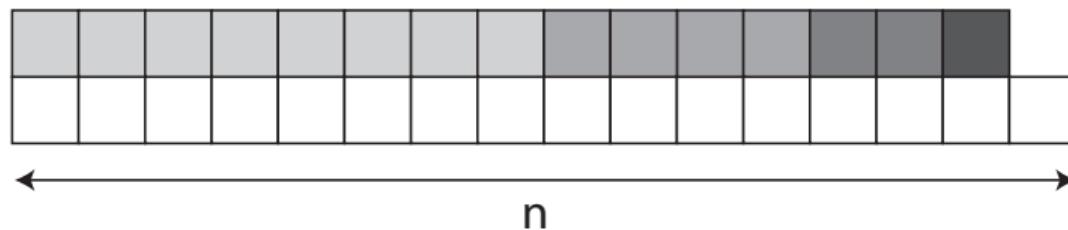
Větší geometrická intuice



Větší geometrická intuice



Větší geometrická intuice



Zcela neobanální a téměř sofistikovaný příklad

kolikrát se vykoná metoda DoSomething()?

```
int c = n;  
while(c>0) {  
    for(int j = 0; j<c; j++)  
        DoSomething(i, j);  
    c = c/2;  
}
```

$g(n) = 2n - 1$, pokud $n = 2^k$, tj. každé dělení je beze zbytku

Zcela neobanální a téměř sofistikovaný příklad

kolikrát se vykoná metoda DoSomething()?

```
int c = n;  
while(c>0) {  
    for(int j = 0; j<c; j++)  
        DoSomething(i, j);  
    c = c/2;  
}
```

$g(n) = 2n - 1$, pokud $n = 2^k$, tj. každé dělení je beze zbytku
pokud $n \neq 2^k$, potom $g(n) < 2n - 1$, a tudíž $g(n) \in \mathcal{O}(n)$

Splnitelnost logické formule

Mějme logickou formuli o n argumentech

Splnitelnost logické formule

Mějme logickou formuli o n argumentech

Problém

Existuje taková sada argumentů, pro které je logická formule pravdivá?

Splnitelnost logické formule

Mějme logickou formuli o n argumentech

Problém

Existuje taková sada argumentů, pro které je logická formule pravdivá?

Příklad:

$n = 3$;

$$f_1(a, b, c) = (a \& \& b) \parallel (a \& \& c)$$

je splnitelná: $a=true$, $b=true$, $c=true$

Splnitelnost logické formule

Mějme logickou formuli o n argumentech

Problém

Existuje taková sada argumentů, pro které je logická formule pravdivá?

Příklad:

$n = 3$;

$$f_1(a, b, c) = (a \& \& b) \parallel (a \& \& c)$$

je splnitelná: $a=true$, $b=true$, $c=true$

$$f_2(a, b, c) = (a \& \& b) \& \& (!b)$$

není splnitelná

Dveře a páky



Možné řešení:

- testovat všechny kombinace argumentů
- pokud je formule splněna, vracíme `true`
- pokud se vyčerpají všechny možnosti, vracíme `false`
- rekurzivní algoritmus na generování všech možností:
 - nastaví na n -té místo `true` a testuje všechny možnosti na pozicích $n + 1$ a dále
 - nastaví na n -té místo `false` a testuje všechny možnosti na pozicích $n + 1$ a dále

SplitInost - algoritmus

```
bool Expression(bool[] v) {  
    ...  
}  
  
bool TestExpression(bool[] v, int n) {  
    if (n == v.Length)  
        return Expression(v);  
    else {  
        v[n] = true;  
        if (TestExpression(v, n+1))  
            return true;  
        v[n] = false;  
        if (TestExpression(v, n+1))  
            return true;  
        return false;  
    }  
}
```

Složitost vs. výkon

Počet testovaných možností: 2^n , složitost $\Theta(2^n)$

Složitost vs. výkon

Počet testovaných možností: 2^n , složitost $\Theta(2^n)$

Předpokládejme počítač, který dokáže vyřešit v přijatelném čase problém pro $n = 100$

Složitost vs. výkon

Počet testovaných možností: 2^n , složitost $\Theta(2^n)$

Předpokládejme počítač, který dokáže vyřešit v přijatelném čase problém pro $n = 100$
jedno vyhodnocení výrazu: t_1

Složitost vs. výkon

Počet testovaných možností: 2^n , složitost $\Theta(2^n)$

Předpokládejme počítač, který dokáže vyřešit v přijatelném čase problém pro $n = 100$
jedno vyhodnocení výrazu: t_1

Nový počítač, 8x rychlejší, jedno vyhodnocení výrazu: $t_1/8$

Složitost vs. výkon

Počet testovaných možností: 2^n , složitost $\Theta(2^n)$

Předpokládejme počítač, který dokáže vyřešit v přijatelném čase problém pro $n = 100$
jedno vyhodnocení výrazu: t_1

Nový počítač, 8x rychlejší, jedno vyhodnocení výrazu: $t_1/8$

Otázka

Jak velký problém (pro jaké n) dokáže nový počítač vyřešit?

Složitost vs. výkon

$$t_1 2^{100} = \frac{t_1}{8} 2^{n_2}$$

Složitost vs. výkon

$$t_1 2^{100} = \frac{t_1}{8} 2^{n_2}$$

$$t_1 2^{100} = t_1 2^{-3} 2^{n_2}$$

Složitost vs. výkon

$$t_1 2^{100} = \frac{t_1}{8} 2^{n_2}$$

$$t_1 2^{100} = t_1 2^{-3} 2^{n_2}$$

$$t_1 2^{100} = t_1 2^{n_2 - 3}$$

Složitost vs. výkon

$$t_1 2^{100} = \frac{t_1}{8} 2^{n_2}$$

$$t_1 2^{100} = t_1 2^{-3} 2^{n_2}$$

$$t_1 2^{100} = t_1 2^{n_2 - 3}$$

$$100 = n_2 - 3$$

Složitost vs. výkon

$$t_1 2^{100} = \frac{t_1}{8} 2^{n_2}$$

$$t_1 2^{100} = t_1 2^{-3} 2^{n_2}$$

$$t_1 2^{100} = t_1 2^{n_2 - 3}$$

$$100 = n_2 - 3$$

$$n_2 = 103$$

Složitost vs. výkon

$$t_1 2^{100} = \frac{t_1}{8} 2^{n_2}$$

$$t_1 2^{100} = t_1 2^{-3} 2^{n_2}$$

$$t_1 2^{100} = t_1 2^{n_2 - 3}$$

$$100 = n_2 - 3$$

$$n_2 = 103$$

- 8x rychlejší počítač dokáže v tomto případě vyřešit problém o 3% větší
- na rychlosti počítače opravdu moc nezáleží

Složitost vs. výkon

$$t_1 2^{200} = \frac{t_1}{8} 2^{n_2}$$

Složitost vs. výkon

$$t_1 2^{200} = \frac{t_1}{8} 2^{n_2}$$

$$t_1 2^{200} = t_1 2^{-3} 2^{n_2}$$

$$t_1 2^{200} = t_1 2^{n_2 - 3}$$

$$200 = n_2 - 3$$

$$n_2 = 203$$

Složitost vs. výkon

$$t_1 2^{200} = \frac{t_1}{8} 2^{n_2}$$

$$t_1 2^{200} = t_1 2^{-3} 2^{n_2}$$

$$t_1 2^{200} = t_1 2^{n_2 - 3}$$

$$200 = n_2 - 3$$

$$n_2 = 203$$

- 8x rychlejší počítač vyřešit problém **o 3 větší** (tentokrát o 1.5%)
- čím větší je problém, tím menší relativní efekt má zrychlení počítače

Složitost vs. výkon

Otázka

Pokud máme problém velikosti $n = 1000$, jak rychlý bychom potřebovali počítač?

Otázka

Pokud máme problém velikosti $n = 1000$, jak rychlý bychom potřebovali počítač?

$$t_1 2^{100} = \frac{t_1}{k} 2^{1000}$$

Otázka

Pokud máme problém velikosti $n = 1000$, jak rychlý bychom potřebovali počítač?

$$t_1 2^{100} = \frac{t_1}{k} 2^{1000}$$

$$k 2^{100} = 2^{1000}$$

Složitost vs. výkon

Otázka

Pokud máme problém velikosti $n = 1000$, jak rychlý bychom potřebovali počítač?

$$t_1 2^{100} = \frac{t_1}{k} 2^{1000}$$

$$k 2^{100} = 2^{1000}$$

$$k = \frac{2^{1000}}{2^{100}}$$

Složitost vs. výkon

Otázka

Pokud máme problém velikosti $n = 1000$, jak rychlý bychom potřebovali počítač?

$$t_1 2^{100} = \frac{t_1}{k} 2^{1000}$$

$$k 2^{100} = 2^{1000}$$

$$k = \frac{2^{1000}}{2^{100}}$$

$$k = 2^{900} = 8.457 \times 10^{270}$$

Složitost vs. výkon

Otázka

Pokud máme problém velikosti $n = 1000$, jak rychlý bychom potřebovali počítač?

$$t_1 2^{100} = \frac{t_1}{k} 2^{1000}$$

$$k 2^{100} = 2^{1000}$$

$$k = \frac{2^{1000}}{2^{100}}$$

$$k = 2^{900} = 8.457 \times 10^{270}$$

- nemůžeme čekat, že časem pomůže rychlejší počítač
- nepomůže ani Mooreův zákon: i kdyby se výkon skutečně každé dva roky zdvojnásobil, budeme čekat na dostatečně výkonný stroj **1800 let**

Praktické určení složitosti

```
int Example(int n) {  
    int result = 0;  
    for (int i = 0; i < n; i++)  
        for (int j = 0; j < n; j++)  
            if (i == j/2)  
                result += Math.Sin(i*j);  
    return result;  
}
```

Praktické určení složitosti

```
int Example(int n) {  
    int result = 0;  
    for (int i = 0; i < n; i++)  
        for (int j = 0; j < n; j++)  
            if (i == j/2)  
                result += Math.Sin(i*j);  
    return result;  
}
```

- výpočet funkce sin se provádí $\Theta(n)$ krát

Praktické určení složitosti

```
int Example(int n) {  
    int result = 0;  
    for (int i = 0; i < n; i++)  
        for (int j = 0; j < n; j++)  
            if (i == j/2)  
                result += Math.Sin(i*j);  
    return result;  
}
```

- výpočet funkce sin se provádí $\Theta(n)$ krát
- algoritmus je ale $\Theta(n^2)$, protože podmínka se vyhodnocuje $\Theta(n^2)$ krát!

Praktické určení složitosti

```
int Example(int n) {  
    int result = 0;  
    for (int i = 0; i < n; i++)  
        for (int j = 0; j < n; j++)  
            if (i == j/2)  
                result += Math.Sin(i*j);  
    return result;  
}
```

- výpočet funkce sin se provádí $\Theta(n)$ krát
- algoritmus je ale $\Theta(n^2)$, protože **podmínka** se vyhodnocuje $\Theta(n^2)$ krát!
- **není možné její vyhodnocení zanedbat**, protože má vliv na celkovou složitost algoritmu!

Praktické určení výpočetní složitosti

```
bool ContainsDuplicates(int[] a, int n) {  
    for (int i = 0; i < n; i++)  
        for (int j = i + 1; j < n; j++)  
            if (a[i] == a[j])  
                return true;  
    return false;  
}
```

Praktické určení výpočetní složitosti

```
bool ContainsDuplicates(int[] a, int n) {  
    for (int i = 0; i < n; i++)  
        for (int j = i + 1; j < n; j++)  
            if (a[i] == a[j])  
                return true;  
    return false;  
}
```

- složitost pro nejlepší data $\Theta(1)$ (pokud jsou první dva prvky stejné)

Praktické určení výpočetní složitosti

```
bool ContainsDuplicates(int[] a, int n) {  
    for (int i = 0; i < n; i++)  
        for (int j = i + 1; j < n; j++)  
            if (a[i] == a[j])  
                return true;  
    return false;  
}
```

- složitost pro nejlepší data $\Theta(1)$ (pokud jsou první dva prvky stejné)
- složitost pro nejhorší data $\Theta(n^2)$ (pokud data neobsahují duplikáty)

Praktické určení výpočetní složitosti

```
bool ContainsDuplicates(int[] a, int n) {  
    for (int i = 0; i < n; i++)  
        for (int j = i + 1; j < n; j++)  
            if (a[i] == a[j])  
                return true;  
    return false;  
}
```

- složitost pro nejlepší data $\Theta(1)$ (pokud jsou první dva prvky stejné)
- složitost pro nejhorší data $\Theta(n^2)$ (pokud data neobsahují duplikáty)
- složitost v průměru $\Theta(?)$ (záleží na tom, jaká jsou průměrná data. Náhodná? V jakém rozsahu?)

Praktické určení složitosti

```
bool ContainsDuplicates2(int[] a) {
    MyCollection c = new MyCollection();

    c.Add(a[0]);
    for (int i = 1; i < a.Length; i++)  {
        if (c.Contains(a[i]))
            return true;
        c.Add(a[i]);
    }
    return false;
}
```

Praktické určení složitosti

```
bool ContainsDuplicates2(int[] a) {  
    MyCollection c = new MyCollection();  
  
    c.Add(a[0]);  
    for (int i = 1; i < a.Length; i++) {  
        if (c.Contains(a[i]))  
            return true;  
        c.Add(a[i]);  
    }  
    return false;  
}
```

- složitost v nejhorším případě? $\Theta(n)$?

Praktické určení složitosti

```
bool ContainsDuplicates2(int[] a) {  
    MyCollection c = new MyCollection();  
  
    c.Add(a[0]);  
    for (int i = 1; i < a.Length; i++) {  
        if (c.Contains(a[i]))  
            return true;  
        c.Add(a[i]);  
    }  
    return false;  
}
```

- složitost v nejhorším případě? $\Theta(n)$?
- záleží na tom jak dlouho trvají metody `Add()` a `Contains()`!

Praktické určení složitosti

```
bool ContainsDuplicates2(int[] a) {  
    MyCollection c = new MyCollection();  
  
    c.Add(a[0]);  
    for (int i = 1; i < a.Length; i++) {  
        if (c.Contains(a[i]))  
            return true;  
        c.Add(a[i]);  
    }  
    return false;  
}
```

- složitost v nejhorším případě? $\Theta(n)$?
- záleží na tom jak dlouho trvají metody `Add()` a `Contains()`!
- čas jejich vykonání **může (a nemusí)** záviset na tom, kolik prvků už je v kolekci `c`!

Složitost: big picture



Složitost: big picture

Tady jsou funkce,
které rostou
rychle



Tady jsou funkce,
které rostou
pomalu

Složitost: big picture



Tady jsou funkce,
které rostou
rychle



Tady jsou funkce,
které rostou
pomalu

Složitost: big picture



Tady jsou funkce,
které rostou
rychle

$$f(n) = 2^n$$

$$f(n) = n^3$$

$$f(n) = n^2$$

$$f(n) = n * \log(n)$$

$$f(n) = n$$

$$f(n) = \sqrt{n}$$

$$f(n) = 1$$

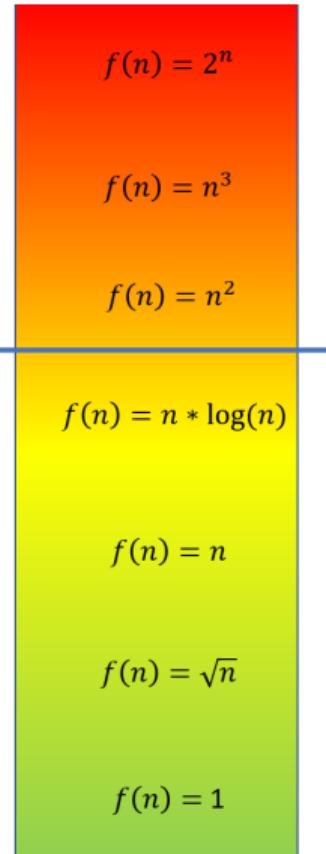


Tady jsou funkce,
které rostou
pomalu

Složitost: big picture



Tady jsou funkce,
které rostou
rychle



Tady jsou funkce,
které rostou
pomalu

Složitost: big picture



Tady jsou funkce,
které rostou
rychle

$$f(n) = 2^n$$

$$f(n) = n^3$$

$$f(n) = n^2$$

$$f(n)$$

$$f(n) = n * \log(n)$$

$$f(n) = n$$

Množina
funkcí
 $O(f(n))$

$$f(n) = \sqrt{n}$$

$$f(n) = 1$$

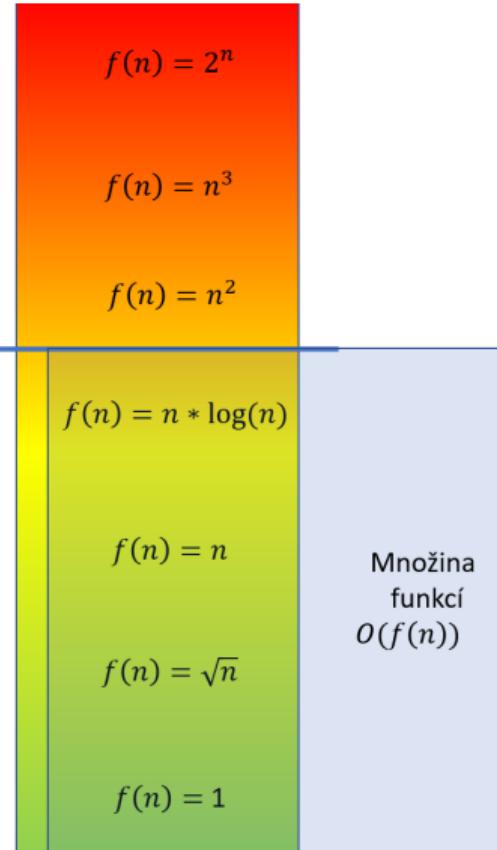


Tady jsou funkce,
které rostou
pomalu

Složitost: big picture



Tady jsou funkce,
které rostou
rychle



Tady jsou funkce,
které rostou
pomalu

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

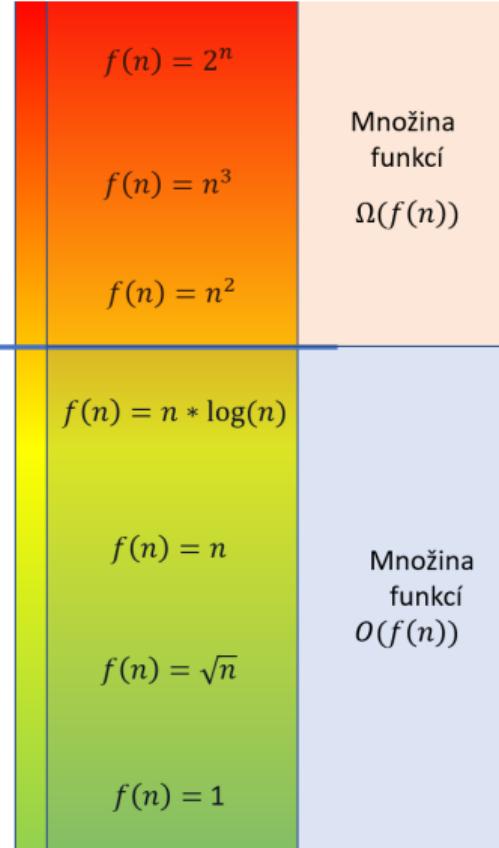
$g(n)$ se nachází
někde v množině
v nejhorším případě
rostě tak rychle jako
 $f(n)$

Složitost: big picture



Tady jsou funkce,
které rostou
rychle

$f(n)$



Tady jsou funkce,
které rostou
pomalu

$g(n) \in O(f(n))$
 $g(n)$ se nachází
někde v množině
v nejhorším případě
rostě tak rychle jako
 $f(n)$

Složitost: big picture

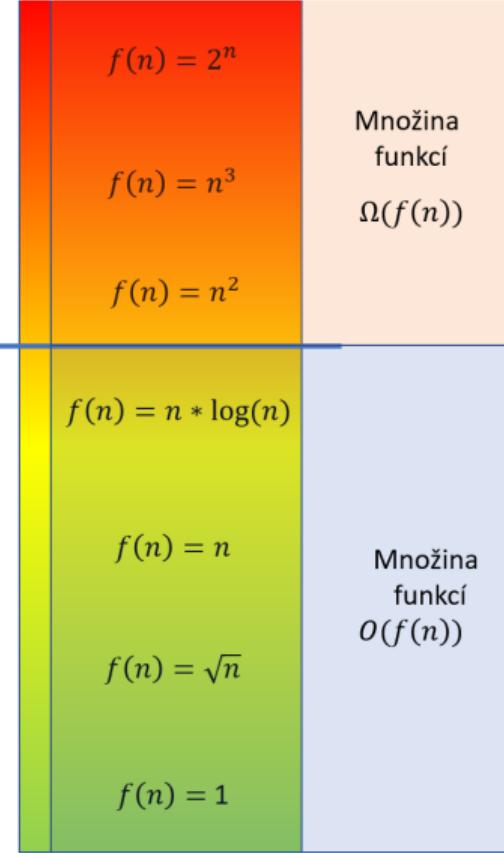


Tady jsou funkce,
které rostou
rychle

$f(n)$



Tady jsou funkce,
které rostou
pomalu



$g(n) \in \Omega(f(n))$

$g(n)$ se nachází
někde v množině
v nejlepším případě
roste tak rychle jako
 $f(n)$

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

$g(n)$ se nachází
někde v množině
v nejhorším případě
roste tak rychle jako
 $f(n)$

Složitost: big picture

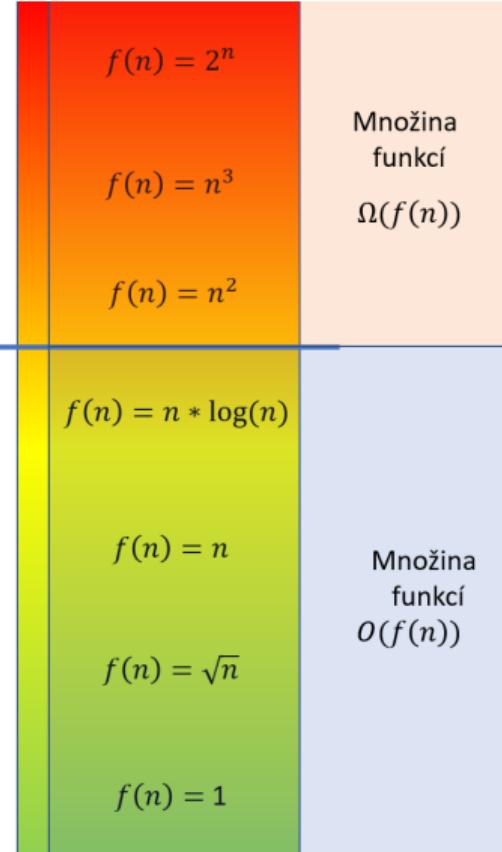


Tady jsou funkce, které rostou rychle

$f(n)$



Tady jsou funkce, které rostou pomalu



$g(n) \in \Omega(f(n))$

$g(n)$ se nachází někde v množině v nejlepším případě roste tak rychle jako $f(n)$

$\Theta(f(n)) = \Omega(f(n)) \cap O(f(n))$

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

$g(n)$ se nachází někde v množině v nejhorším případě roste tak rychle jako $f(n)$

Zdroj zmatení



Zdroj zmatení

V nejhorším případě...

V nejlepším případě...

V nejhorším případě...

V nejlepším případě...

Zdroj zmatení

V nejhorším případě...
V nejlepším případě...

V nejhorším případě...
V nejlepším případě...

O funkci $g(n)$ je dokázána příslušnost do množiny $O(f(n))$, případně $\Omega(f(n))$.

Nachází se někde v množině, nevíme kde.

Nejhorší a nejlepší se vztahují k **nejistotě polohy** v množině.

Zdroj zmatení

V nejhorším případě...
V nejlepším případě...

O funkci $g(n)$ je dokázána příslušnost do množiny $O(f(n))$, případně $\Omega(f(n))$.

Nachází se někde v množině, nevíme kde.

Nejhorší a nejlepší se vztahují k **nejistotě polohy** v množině.

V nejhorším případě...
V nejlepším případě...

Některé algoritmy se chovají různě pro různá data.

Chování lze popsat různými funkcemi, podle charakteru dat.

Nejhorší a nejlepší se vztahují k **charakteru dat**, zda vedou na příznivou nebo nepříznivou složitost.

Sekvenční vyhledávání prvku v seřazeném poli

```
bool Contains(int[] data, int x) {  
    for (int i = 0; i < data.Length; i++) {  
        if (data[i] == x)  
            return true;  
        if (data[i] > x)  
            return false;  
    }  
    return false;  
}
```

Sekvenční vyhledávání prvku v seřazeném poli

```
bool Contains(int[] data, int x) {  
    for (int i = 0; i < data.Length; i++) {  
        if (data[i] == x)  
            return true;  
        if (data[i] > x)  
            return false;  
    }  
    return false;  
}
```

$g(n) = \text{počet obrátek for smyčky}$
 $n = \text{data.Length}$

Poslední příklad

Sekvenční vyhledávání prvku v seřazeném poli

```
bool Contains(int[] data, int x) {  
    for (int i = 0; i < data.Length; i++) {  
        if (data[i] == x)  
            return true;  
        if (data[i] > x)  
            return false;  
    }  
    return false;  
}
```

$g(n)$ = počet obrátek for smyčky
 n = data.Length

Nejlepší případ:
 $g_1(n) = 1$

Sekvenční vyhledávání prvku v seřazeném poli

```
bool Contains(int[] data, int x) {  
    for (int i = 0; i < data.Length; i++) {  
        if (data[i] == x)  
            return true;  
        if (data[i] > x)  
            return false;  
    }  
    return false;  
}
```

$g(n)$ = počet obrátek for smyčky
 n = data.Length

Nejlepší případ:
 $g_1(n) = 1$

Hledaný prvek
je hned na
začátku

Sekvenční vyhledávání prvku v seřazeném poli

```
bool Contains(int[] data, int x) {  
    for (int i = 0; i < data.Length; i++) {  
        if (data[i] == x)  
            return true;  
        if (data[i] > x)  
            return false;  
    }  
    return false;  
}
```

$g(n)$ = počet obrátek for smyčky
 n = data.Length

Nejlepší případ:
 $g_1(n) = 1$

Hledaný prvek
je hned na
začátku

Nejhorší případ:
 $g_2(n) = n$

Sekvenční vyhledávání prvku v seřazeném poli

```
bool Contains(int[] data, int x) {  
    for (int i = 0; i < data.Length; i++) {  
        if (data[i] == x)  
            return true;  
        if (data[i] > x)  
            return false;  
    }  
    return false;  
}
```

$g(n)$ = počet obrátek for smyčky
 n = `data.Length`

Nejlepší případ:
 $g_1(n) = 1$

Hledaný prvek
je hned na
začátku

Nejhorší případ:
 $g_2(n) = n$

Hledaný prvek v
poli vůbec není

Nezávislé vlastnosti

Nejhorší/nejlepší **případ vstupních dat**

$$g_1(n) = 1$$



$$g_2(n) = n$$

Nezávislé vlastnosti

Nejhorší/nejlepší **případ vstupních dat**

$$g_1(n) = 1$$



$$g_2(n) = n$$



Nezávislé vlastnosti

Nejhorší/nejlepší případ vstupních dat

$$g_1(n) = 1$$



$$g_2(n) = n$$

Nejhorší/nejlepší rychlosť v rámci množiny funkcií



Nezávislé vlastnosti

Nejhorší/nejlepší případ vstupních dat

$$g_1(n) = 1$$



$$g_2(n) = n$$

Nejhorší/nejlepší rychlosť v rámci množiny funkcií

$g_1(n) \in O(1) \Rightarrow$ je to rychlé



Nezávislé vlastnosti

Nejhorší/nejlepší případ vstupních dat

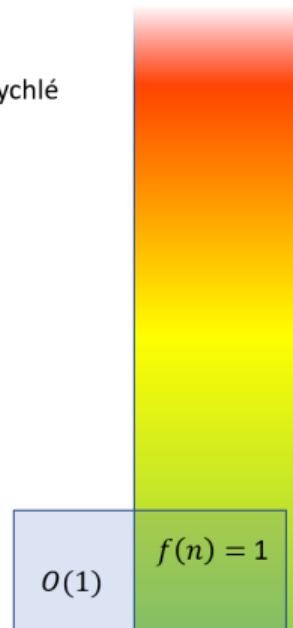
$$g_1(n) = 1$$



$$g_2(n) = n$$

Nejhorší/nejlepší rychlosť v rámci množiny funkcií

$g_1(n) \in O(1) \Rightarrow$ je to rychlé



Nezávislé vlastnosti

Nejhorší/nejlepší případ vstupních dat

$$g_1(n) = 1$$



$$g_2(n) = n$$

Nejhorší/nejlepší rychlosť v rámci množiny funkcií

$g_1(n) \in O(1) \Rightarrow$ je to rychlé

$g_1(n) \in \Omega(1) \Rightarrow ?$



$O(1)$	$f(n) = 1$
--------	------------

Nezávislé vlastnosti

Nejhorší/nejlepší případ vstupních dat

$$g_1(n) = 1$$



$$g_2(n) = n$$

Nejhorší/nejlepší rychlosť v rámci množiny funkcií

$g_1(n) \in O(1) \Rightarrow$ je to rychlé

$g_1(n) \in \Omega(1) \Rightarrow ?$

$g_1(n) \in O(n^5) \Rightarrow ?$



$O(1)$	$f(n) = 1$
--------	------------

Nezávislé vlastnosti

Nejhorší/nejlepší případ vstupních dat

$$g_1(n) = 1$$



$$g_2(n) = n$$

Nejhorší/nejlepší rychlosť v rámci množiny funkcií

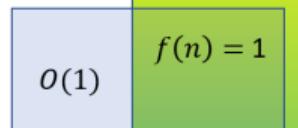
$g_1(n) \in O(1) \Rightarrow$ je to rychlé



$g_1(n) \in \Omega(1) \Rightarrow ?$

$g_1(n) \in O(n^5) \Rightarrow ?$

$g_1(n) \notin \Omega(n^5) \Rightarrow ?$



Nezávislé vlastnosti

Nejhorší/nejlepší případ vstupních dat

$$g_1(n) = 1$$



$$g_2(n) = n$$

Nejhorší/nejlepší rychlosť v rámci množiny funkcií

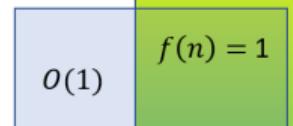
$g_1(n) \in O(1) \Rightarrow$ je to rychlé

$g_1(n) \in \Omega(1) \Rightarrow ?$

$g_1(n) \in O(n^5) \Rightarrow ?$

$g_1(n) \notin \Omega(n^5) \Rightarrow ?$

$g_2(n) \notin O(1) \Rightarrow ?$



Nezávislé vlastnosti

Nejhorší/nejlepší případ vstupních dat

$$g_1(n) = 1$$



$$g_2(n) = n$$

Nejhorší/nejlepší rychlosť v rámci množiny funkcií

$g_1(n) \in O(1) \Rightarrow$ je to rychlé

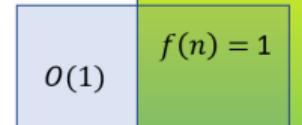
$g_1(n) \in \Omega(1) \Rightarrow ?$

$g_1(n) \in O(n^5) \Rightarrow ?$

$g_1(n) \notin \Omega(n^5) \Rightarrow ?$

$g_2(n) \notin O(1) \Rightarrow ?$

$g_2(n) \in \Omega(1) \Rightarrow ?$



Nezávislé vlastnosti

Nejhorší/nejlepší případ vstupních dat

$$g_1(n) = 1$$



$$g_2(n) = n$$

Nejhorší/nejlepší rychlosť v rámci množiny funkcií

$g_1(n) \in O(1) \Rightarrow$ je to rychlé

$g_1(n) \in \Omega(1) \Rightarrow ?$

$g_1(n) \in O(n^5) \Rightarrow ?$

$g_1(n) \notin \Omega(n^5) \Rightarrow ?$

$g_2(n) \notin O(1) \Rightarrow ?$

$g_2(n) \in \Omega(1) \Rightarrow ?$

$g_2(n) \in O(n) \Rightarrow ?$



Nezávislé vlastnosti

Nejhorší/nejlepší případ vstupních dat

$$g_1(n) = 1$$



$$g_2(n) = n$$

Nejhorší/nejlepší rychlosť v rámci množiny funkcií

$g_1(n) \in O(1) \Rightarrow$ je to rychlé

$g_1(n) \in \Omega(1) \Rightarrow ?$

$g_1(n) \in O(n^5) \Rightarrow ?$

$g_1(n) \notin \Omega(n^5) \Rightarrow ?$



$g_2(n) \notin O(1) \Rightarrow ?$

$g_2(n) \in \Omega(1) \Rightarrow ?$

$g_2(n) \in O(n) \Rightarrow ?$

$g_2(n) \in \Omega(n) \Rightarrow$ je to pomalé
(pomalejší než $g_1(n)$)

Nezávislé vlastnosti

Nejhorší/nejlepší případ vstupních dat

$$g_1(n) = 1$$

$$g_2(n) = n$$

Nejhorší/nejlepší rychlosť v rámci množiny funkcií

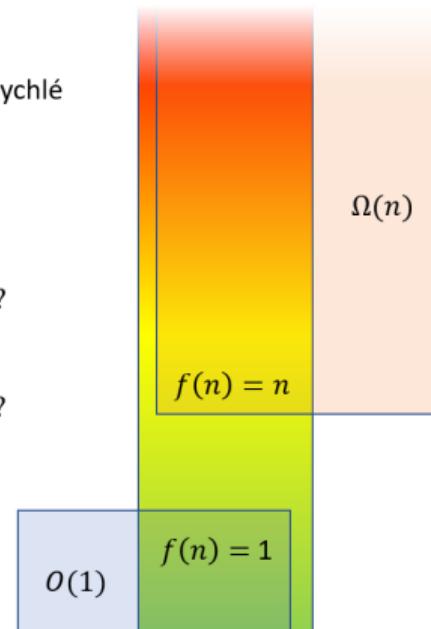


$g_1(n) \in O(1) \Rightarrow$ je to rychlé

$g_1(n) \in \Omega(1) \Rightarrow ?$

$g_1(n) \in O(n^5) \Rightarrow ?$

$g_1(n) \notin \Omega(n^5) \Rightarrow ?$



$g_2(n) \notin O(1) \Rightarrow ?$

$g_2(n) \in \Omega(1) \Rightarrow ?$

$g_2(n) \in O(n) \Rightarrow ?$

$g_2(n) \in \Omega(n) \Rightarrow$ je to pomalé
(pomalejší než $g_1(n)$)

Finální (mnemotechnická?) poznámka

Dobré zprávy sdělujeme v \mathcal{O} -notaci

- funkce se vyskytuje v oblasti pomalu rostoucích, pod určitým limitem

Finální (mnemotechnická?) poznámka

Dobré zprávy sdělujeme v \mathcal{O} -notaci

- funkce se vyskytuje v oblasti pomalu rostoucích, pod určitým limitem

Špatné zprávy sdělujeme v Ω -notaci

- funkce se vyskytuje v oblasti rychle rostoucích, nad určitým limitem

Finální (mnemotechnická?) poznámka

Dobré zprávy sdělujeme v \mathcal{O} -notaci

- funkce se vyskytuje v oblasti pomalu rostoucích, pod určitým limitem

Špatné zprávy sdělujeme v Ω -notaci

- funkce se vyskytuje v oblasti rychle rostoucích, nad určitým limitem

Přesné zprávy sdělujeme v Θ -notaci

- nároky té věci rostou s velikostí vstupu stejně rychle jako $f(x)$