Přednáška 2

Asymptotická složitost (časová složitost)

Insert sort – podrobnější analýza algoritmu

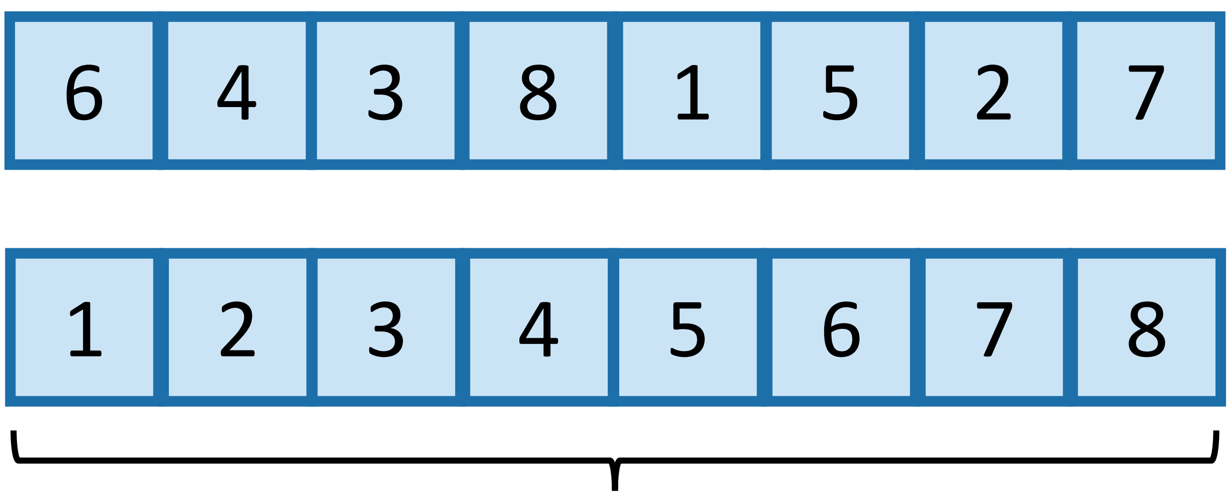
Selection Sort, Bubble Sort – ukázka kódu

Plán

- Třídění (InsertSort)
- Analýza – nejhorší případ
  - Insert Sort (Třídění vkládáním): Jak to funguje?
- Asymptotická složitost (časová složitost) – je náš algoritmus dostatečně rychlý pro naše data?
  - Insertion Sort: Je dostatečně rychlý?
- Selection Sort, Bubble Sort

Řazení

- Důležité – jednoduchý přístup:
  - Seřadíme prvky – budeme potřebovat jen jedno pole – třídění na místě
  - Použité operace – pouze porovnání a přesun prvků
- Stabilita - zachování pořadí prvků se stejnou hodnotou
  - Jednoduché algoritmy většinou ANO
  - Složitější algoritmy většinou NE



Délka posloupnosti (pole) je n

Insert Sort (Třádění vkládáním)

6	4	3	8	5

Začneme tím, že posouváme A [1] na začátek seznamu, dokud nenarazíme menší prvek (nebo už nemůžete dále):

Index pole:

0	1	2	3	4
6	4	3	8	5
4	6	3	8	5

Potom posuneme A[2]:

4	6	3	8	5
3	4	6	8	5

Potom posuneme A[3]:

3	4	6	8	5
3	4	6	8	5

Potom posuneme A[4]:

3	4	6	8	5
3	4	5	6	8

Hotovo!

Celý algoritmus

```
//Vstup: nesetříděné pole <type> [] A
//Výstup: setříděné pole <type> [] A

for (int i = 1; i<n; i++)
{
        t = A[i];
        j = i - 1;
        while (t  < A[j])  {
                A[j+1] = A[j];
                j = j - 1;
                if (j < 0) break;
        }
        A[j+1] = t;
}
```

Celý algoritmus - Python

```
def InsertSort1(A):
  for x in A:
      B = [None for i in range(len(A))]
      for i in range(len(B)):
          if B[i] == None or B[i] > x:
              j = len(B)-1
              while j > i:
                  B[j] = B[j-1]
                  j -= 1
              B[i] = x
              break
       return B
```

```
def InsertSort2(A):
   for i in range(1,len(A)):
      current = A[i]
        j = i-1
        while j >= 0 and A[j] > current:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = current
```

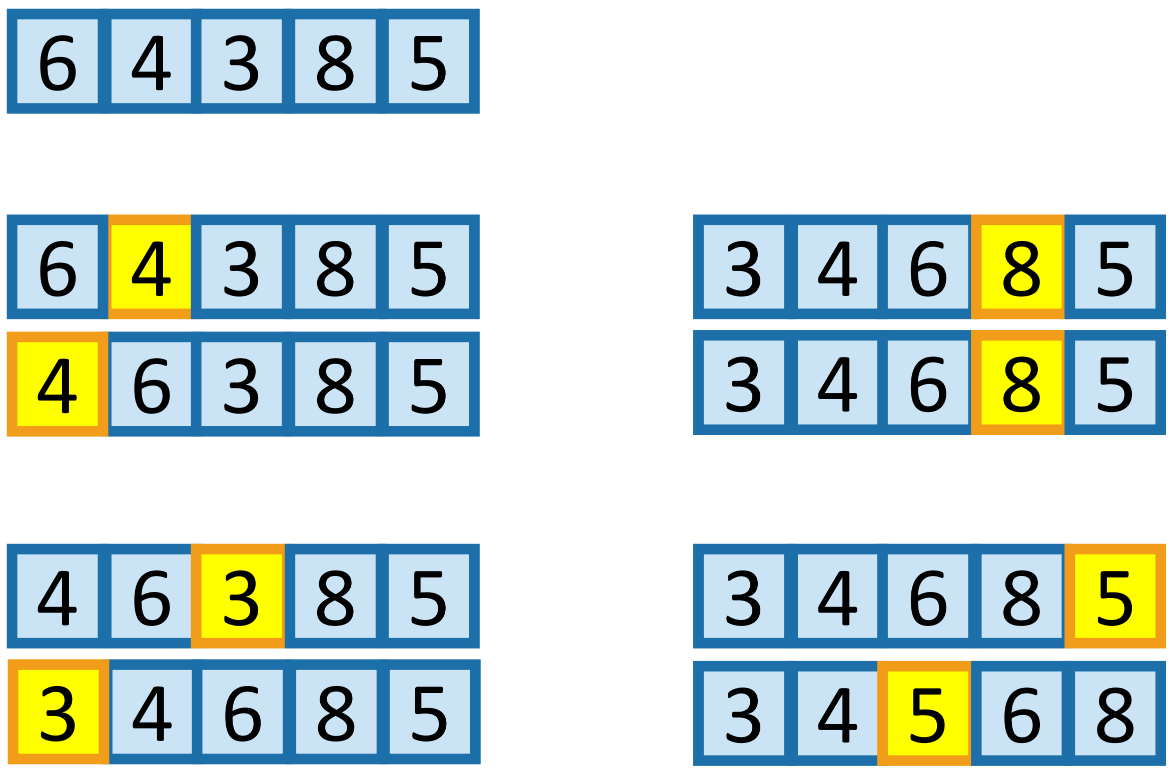
Jedna verze nebo jiná verze

Insert Sort

1. Funguje to – můžeme formálně dokázat?
2. Je to dost rychlé – asymptotická složitost?

Tvrzení: Insert sort „funguje“

- „Důkaz:“ Právě to fungovalo v tomto příkladu:



Seřazeno!

Tvrzení: Insert sort „funguje“

"Důkaz:" Udělal jsem to na spoustě náhodných posloupností a vždy to fungovalo:

```
A = [1,2,3,4,5,6,7,8,9,10]
for trial in range(100):
    shuffle(A)
    InsertionSort(A)
    if is_sorted(A):
        print('YES IT IS SORTED!')
```

```
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
```

```
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
```

```
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
YES IT IS SORTED!
```

Co to znamená, že to „funguje “?

- Stačí, když je to správné pouze na jednom vstupu?
- Stačí, když je to správné na více vstupech?

- Ve naší výuce budeme používat analýzu **nejhoršího případu**:
  - Algoritmus musí být správný na všech možných vstupech.
  - Doba běhu algoritmu je nejhorší možná doba běhu přes všechny vstupy.

Proč to funguje?

- Řekněme, že máme seřazený seznam,

3	4	6	8

  a další prvek 5.

- Vložíme 5 hned po největším prvku, který je stále menší než 5. (Tedy, hned po 4).
- Získáme seřazený seznam:

3	4	5	6	8

Tuto logiku tedy použijeme u každého kroku.

6	4	3	8	5

První prvek [6] tvoří seřazený seznam.

4	6	3	8	5

Správné vložení 4 do seznamu [6] znamená, že se z [4, 6] stane seřazený seznam.

4	6	3	8	5

První dva prvky [4, 6] tvoří seřazený seznam.

3	4	6	8	5

Správné vložení 3 do seznamu [4, 6] znamená, že se z [3, 4, 6] stane seřazený seznam.

3	4	6	8	5

První tři prvky [3, 4, 6] tvoří seřazený seznam.

3	4	6	8	5

Správné vložení 8 do seznamu [3, 4, 6] znamená, že se z [3, 4, 6, 8] stane seřazený seznam.

3	4	6	8	5

První čtyři prvky [3, 4, 6, 8] tvoří seřazený seznam.

3	4	5	6	8

Správné vložení 5 do seznamu [3, 4, 6, 8] znamená, že se z [3, 4, 5, 6, 8] stane seřazený seznam.

HOTOVO!

Použijeme důkaz indukcí ...

Matematická indukce je jedna ze základních důkazových metod, která se obvykle používá, chceme-li dokázat, že nějaké tvrzení či matematická věta platí pro všechna přirozená čísla.

Důkaz matematickou indukcí spočívá ve dvou krocích:

1. Tvrzení dokážeme pro $n = 1$

2. Předpokládáme, že tvrzení platí pevně zvolené $n = k$.

Z tohoto indukčního předpokladu dokážeme, že platí i pro $n = k + 1$.

(viz doprovodný text v Moodle)

Nástin důkazu indukcí

Poznámka: v Pythonu lze zapsat: x[:j] = x[0 : j]
Příklad:
a=[6,7,5,2,9]
print(a[: 3])
Výsledek: [6, 7, 5]

Nechť A je seznam délky n

- Induktivní hypotéza:
  - **`A[:i+1]`** bude seřazeno na konci i-té iterace (vnější smyčky).
- Základní případ (i = 0):
  - **`A[:1]`** bude seřazeno na konci 0-té iterace. ✓
- Inductivní krok:
  - Pro libovolné 0 <k <n, jestliže pro i = k - 1 platí indukční hypotéza, potom také platí pro i = k.
  - Tudíž, jestliže A[: k] je setříděno v kroku k-1, potom A[: k+1] je setříděno v kroku k
- Závěr:
  - Induktivní hypotéza platí pro i = 0, 1,…, n-1.
  - Zejména platí pro i = n-1.
  - Na konci n-1 –té iterace (tudíž na konci běhu algoritmu), `A[:n] = A` je sežazené pole.
  - To jsme chtěli! ✓

Postupujeme pomocí této logiky

4	6	3	8	5

První dva prvky [4, 6] tvoří seřazený seznam.

3	4	6	8	5

Správné vložení 3 do seznamu [4, 6] znamená, že se z [3, 4, 6] stane seřazený seznam.

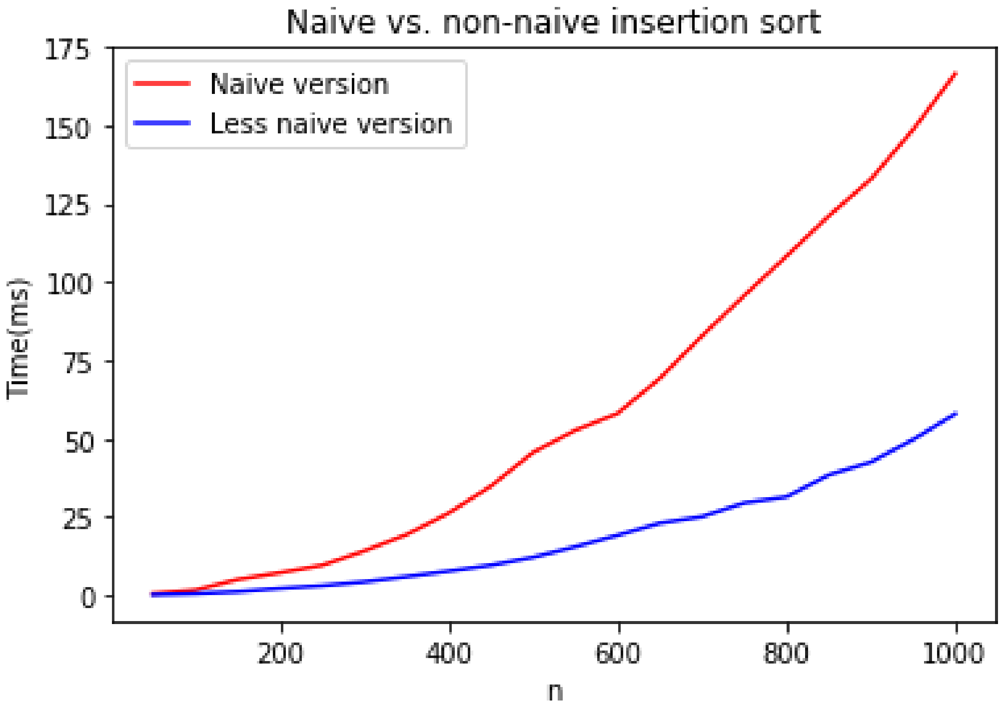
To byla iterace i = 2.

Asymptotická složitost algoritmu (časová složitost)

- **Zjednodušení: obvykle místo asymptotická složitost algoritmu říkáme pouze *složitost algoritmu* (*časová složitost*).**

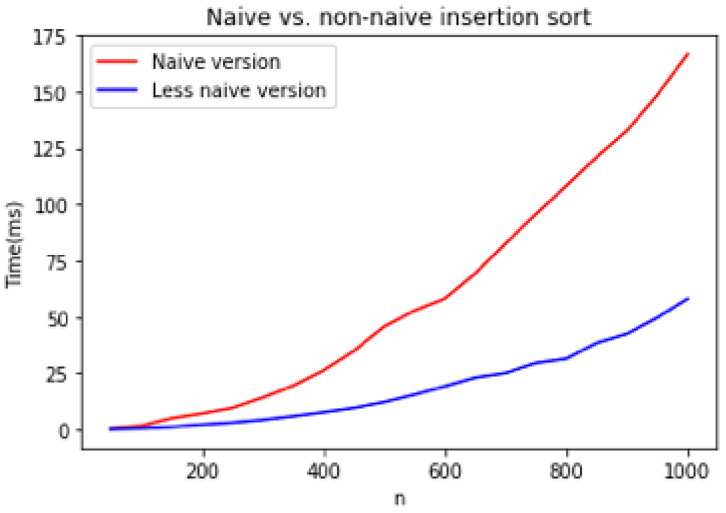
Jak rychlé je řazení Insert sort?

- Takto rychlý:



Problémy s touto odpovědí?

- „Stejný“ algoritmus může být pomalejší nebo rychlejší v závislosti na implementaci.
- Může to být také pomalejší nebo rychlejší v závislosti na hardwaru, na kterém jej provozujeme.



Jak rychlý je Insert sort?

- Spočítejme počet operací!

```
def InsertionSort(A):
    for i in range(1,len(A)):
        current = A[i]
        j = i-1
        while j >= 0 and A[j] > current:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = current
```

Spočítáme operace *…

- $2n^2 - n \ - 1$ přiřazení
- $2n^2 - n \ - 1$ inkrementování/dekrementování
- $2n^2 - 4n + 1$ porovnání
- …

* Nebudeme teď věnovat pozornost těmto vzorcům, na nich teď nezáleží. Ukážeme si to ještě na příkladu.

Problémy s touto odpovědí?

- Je to velmi zdlouhavé!
- Abych to mohl použít k pochopení doby běhu, potřebuji vědět, jak dlouho každá operace trvá, plus spoustu dalších věcí …

```
def InsertionSort(A):
   for i in range(1,len(A)):
      current = A[i]
      j = i-1
      while j >= 0 and A[j] > current:
         A[j+1] = A[j]
         j -= 1
      A[j+1] = current
```

Budeme používat …

- **Velké-Oh notaci!**
- Poskytuje nám smysluplný způsob, jak hovořit o době chodu algoritmu, nezávisle na programovacím jazyce, výpočetní platformě atd., a aniž bychom museli počítat všechny operace jako v předchozím případě.
- Poskytuje na jednoduchý způsob, jak vystihnout chování algoritmu při různé velikosti vstupu (různých *n*)

Hlavní myšlenka :

Zaměříme se na to, jak se doba běhu mění s *n* (velikost vstupu).

(Budeme věnovat pozornost pouze největší funkci n, která se ve funkci objeví (říkáme ji také řád růstu funkce))

Nějaké příklady …

Počet operací	Asymptotická doba běhu
$\frac{1}{10} \cdot n^2 + 100$	$O(n^2)$
$0.063 \cdot n^2 - .5\, n + 12.7$	$O(n^2)$
$100 \cdot n^{1.5} - 10^{10000}\sqrt{n}$	$O(n^{1.5})$
$11 \cdot n \log(n) + 1$	$O(n \log(n))$

Říkáme, že tento algoritmus je „asymptoticky rychlejší“ než ostatní.

Proč je to dobrý nápad?

- Předpokládejme, že doba běhu algoritmu je :

$$T(n) = 10n^2 + 3n + 7 \quad \text{ms}$$

Tento konstantní faktor 10 hodně závisí na mé výpočetní platformě …

Pak nám zbývá výraz n^2! To je to, co je smysluplné.

Na těchto výrazech nižšího řádu ve skutečnosti moc nezáleží, jak se *n* (velikost vstupu) zvětšuje.

Výhody a nevýhody asymptotické časové složitosti

Výhody:

- Abstrahuje od problémů specifických pro hardware a programovací jazyk.
- Díky analýze algoritmu je mnohem přehlednější a zvládnutější.
- Umožňuje nám smysluplně porovnat, jak budou algoritmy fungovat na velkých vstupech.

Nevýhody:

- Dává smysl pouze v případě, že n je velké (i ve srovnání s konstantními faktory).

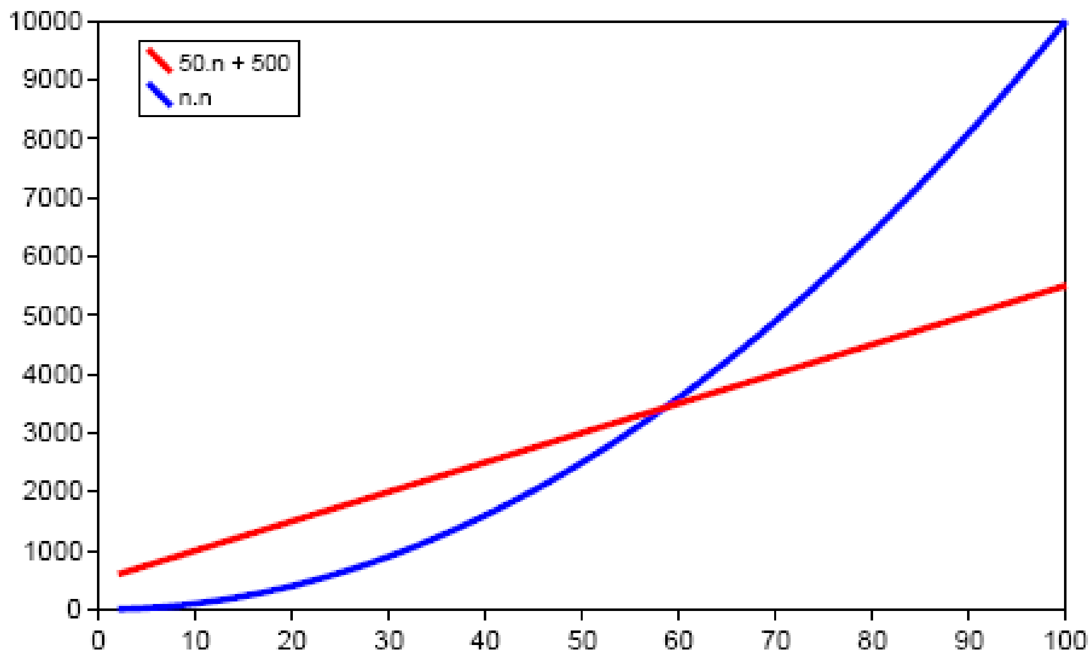
1000000000 n

je “lepší” než n^2 ?!?!

Výhody a nevýhody asymptotické analýzy

Pro malá *n* nemusí být nutně algoritmus s nižší složitostí rychlejší(výhodnější)

např. n^2 vs. $50n + 500$



vyslovuje se „big-oh of…“ nebo někdy „ach …”

Neformální definice pro O(…)

- Nechť $T(n)$, $g(n)$ jsou funkce definované na množině kladných celých čísel.
  - Představte si $T(n)$ jako dobu běhu: je kladná a rostoucí na n.

- Řekneme, že “$T(n)$ je $O\big(g(n)\big)$” jestliže:

pro dostatečně velká n (tj. pro n > n_0 ,)

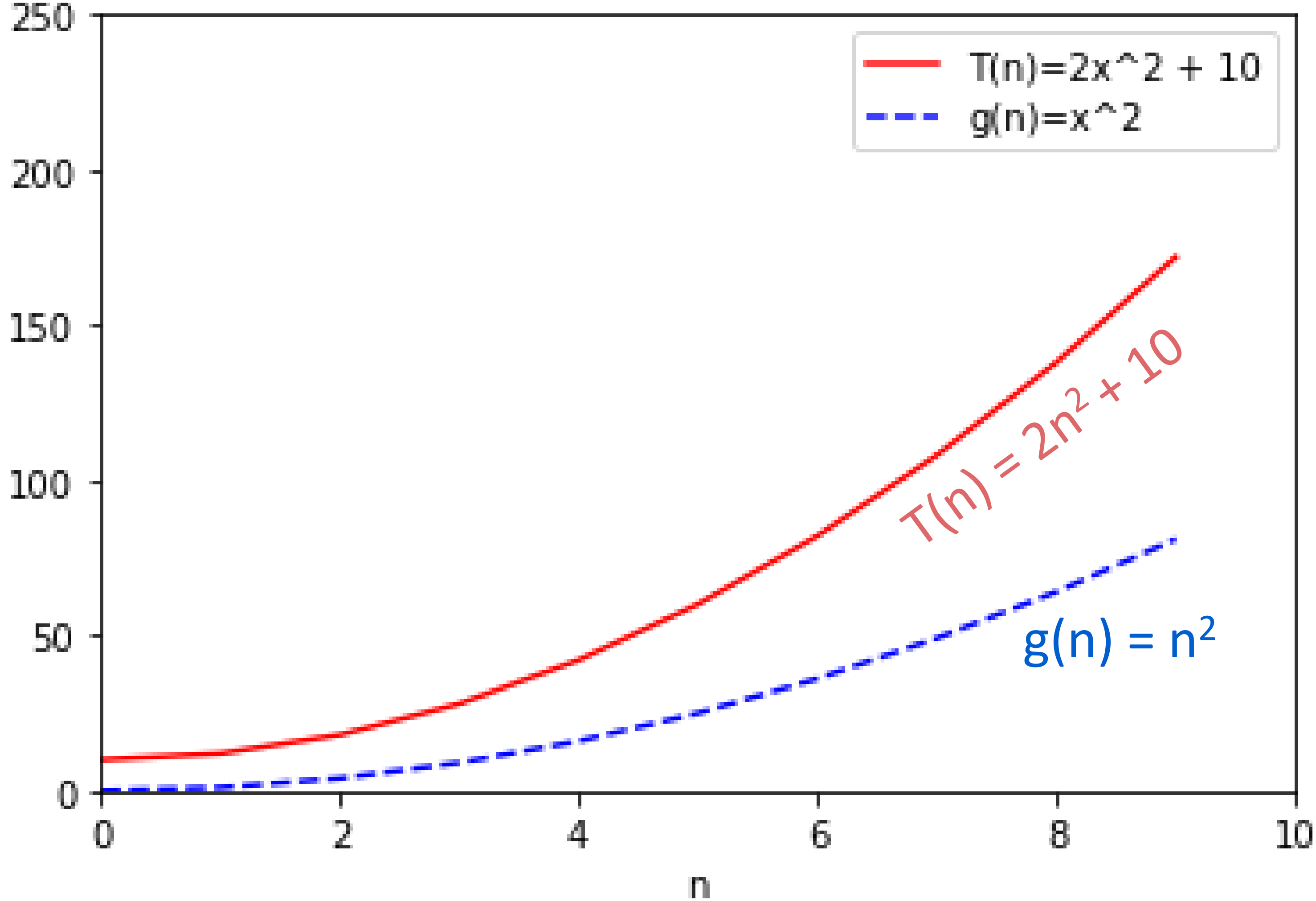
$T(n)$ je menší nebo rovna násobku nějaké konstanty a funkce $g(n)$.

„Konstanta“ zde znamená „nějaké číslo, které nezávisí na n“.

Příklad

$2n^2 + 10 = O(n^2)$

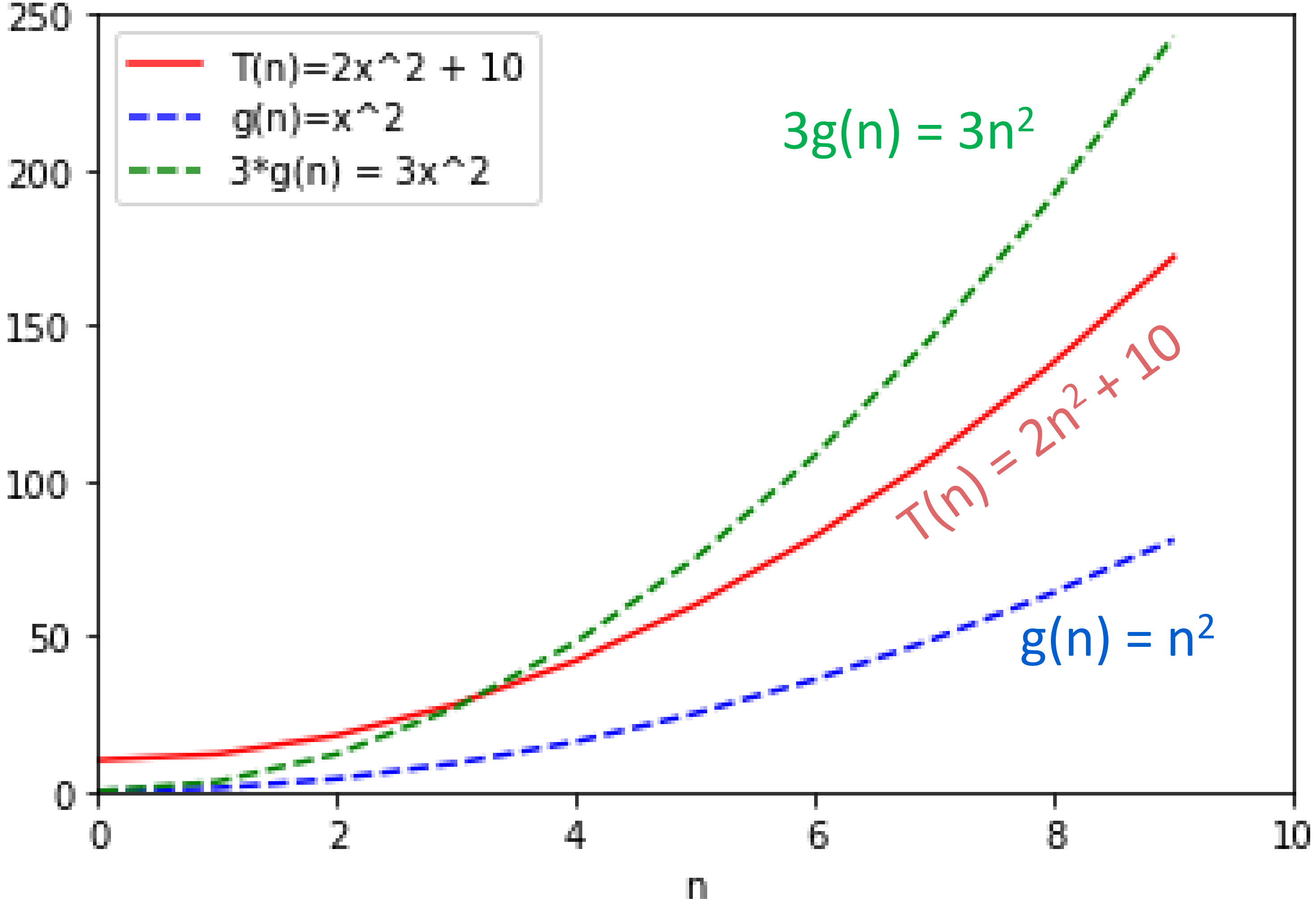
pro dostatečně velké n je $T(n)$ nanejvýš násobek nějaké konstanty a funkce $g(n)$.



Příklad

$2n^2 + 10 = O(n^2)$

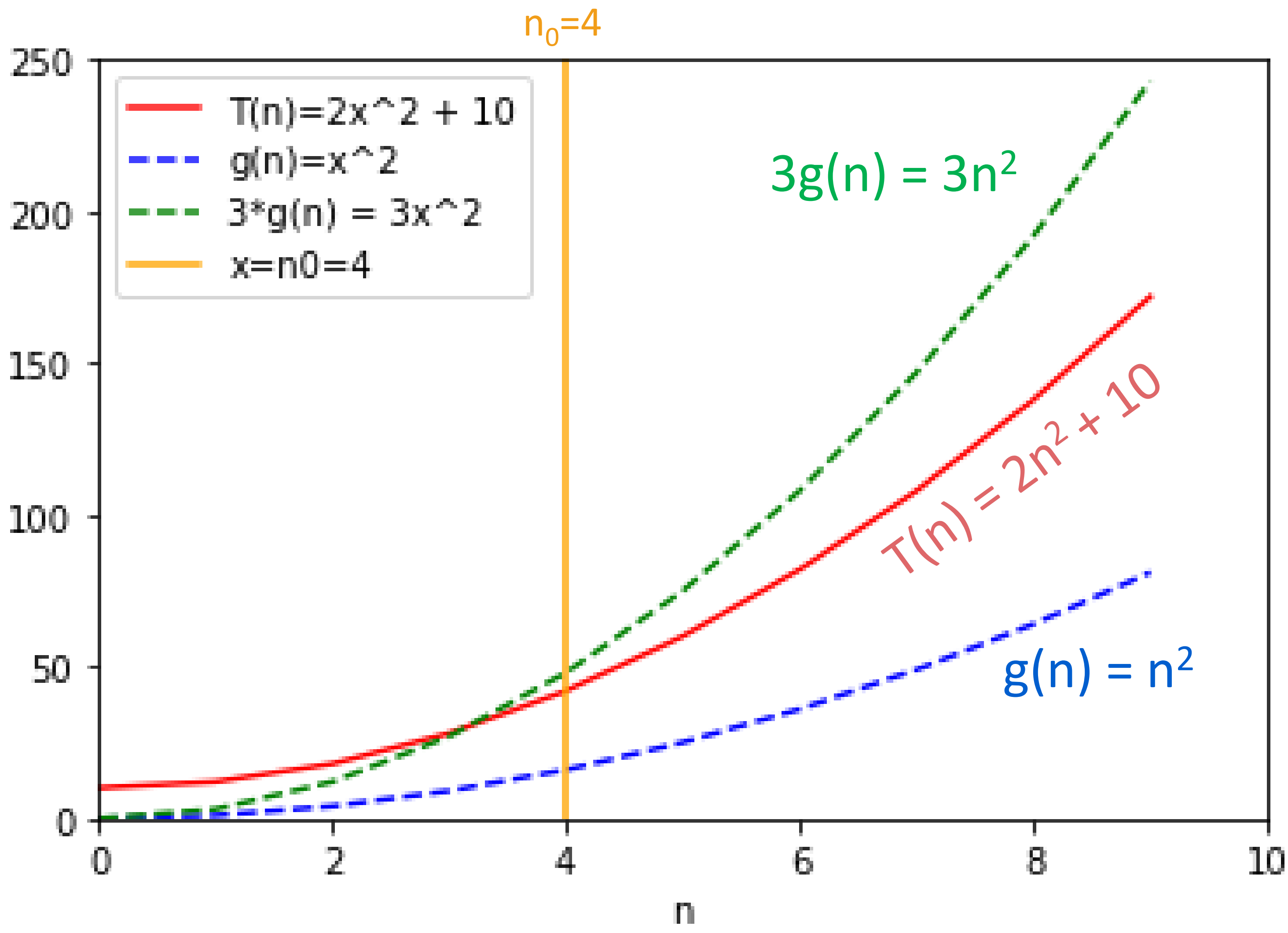
pro dostatečně velké n je $T(n)$ nanejvýš násobek nějaké konstanty a funkce $g(n)$.



Příklad

$2n^2 + 10 = O(n^2)$

pro dostatečně velké n je $T(n)$ nanejvýš násobek nějaké konstanty a funkce $g(n)$.



Formální definice O(...)

- Nechť $T(n)$, $g(n)$ jsou funkce definované na množině kladných celých čísel.
  - Představte si $T(n)$ jako dobu běhu: je kladná a rostoucí na n.

- Formálně,

$$T(n) = O\big(g(n)\big)$$

"právě tehdy, když"

$$\iff$$

"pro všechna"

$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$

$$T(n) \leq c \cdot g(n)$$

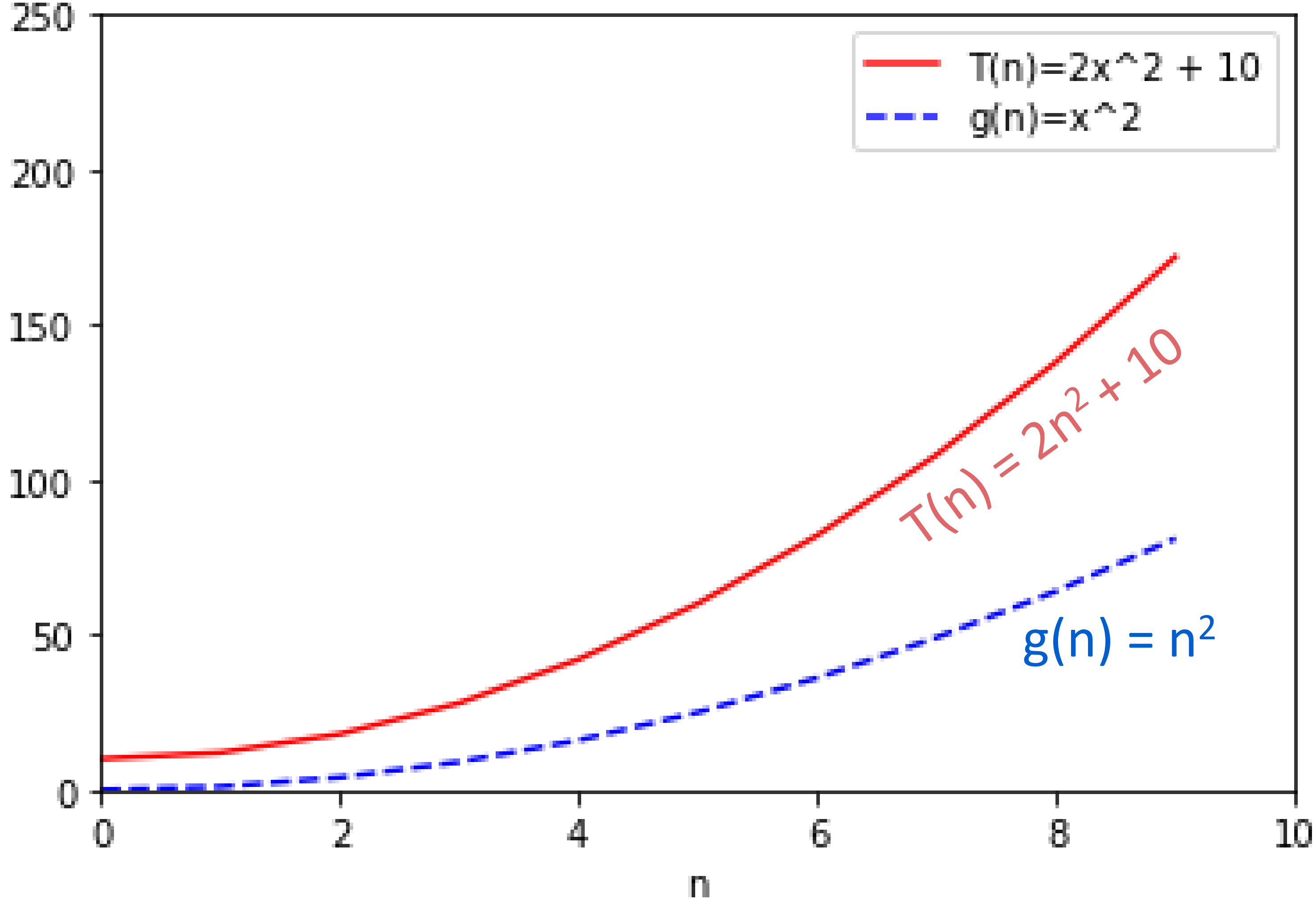
"existuje"

"taková že (such that)"

Příklad

$2n^2 + 10 = O(n^2)$

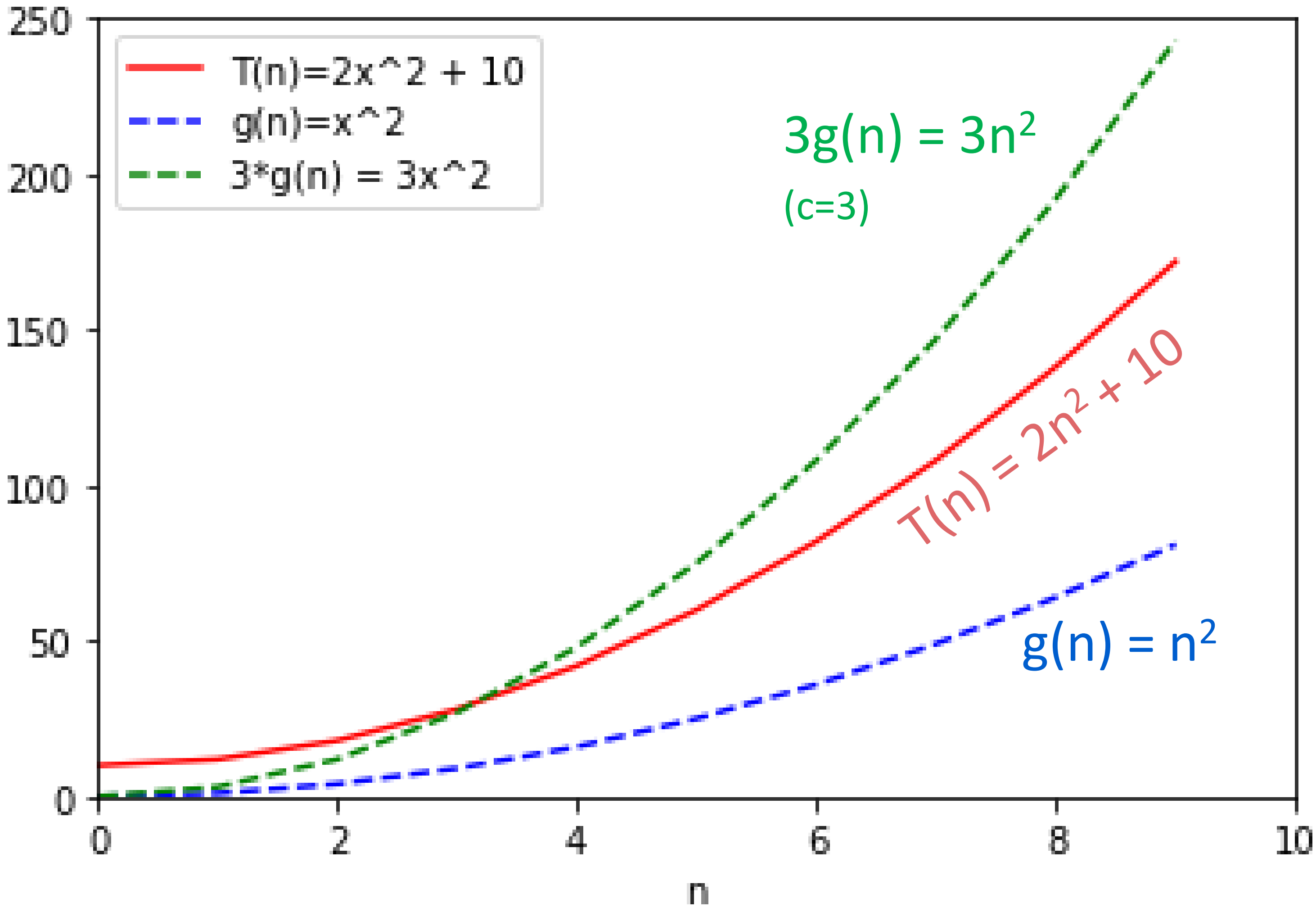
$$T(n) = O\big(g(n)\big)$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$T(n) \leq c \cdot g(n)$$



Příklad

$2n^2 + 10 = O(n^2)$

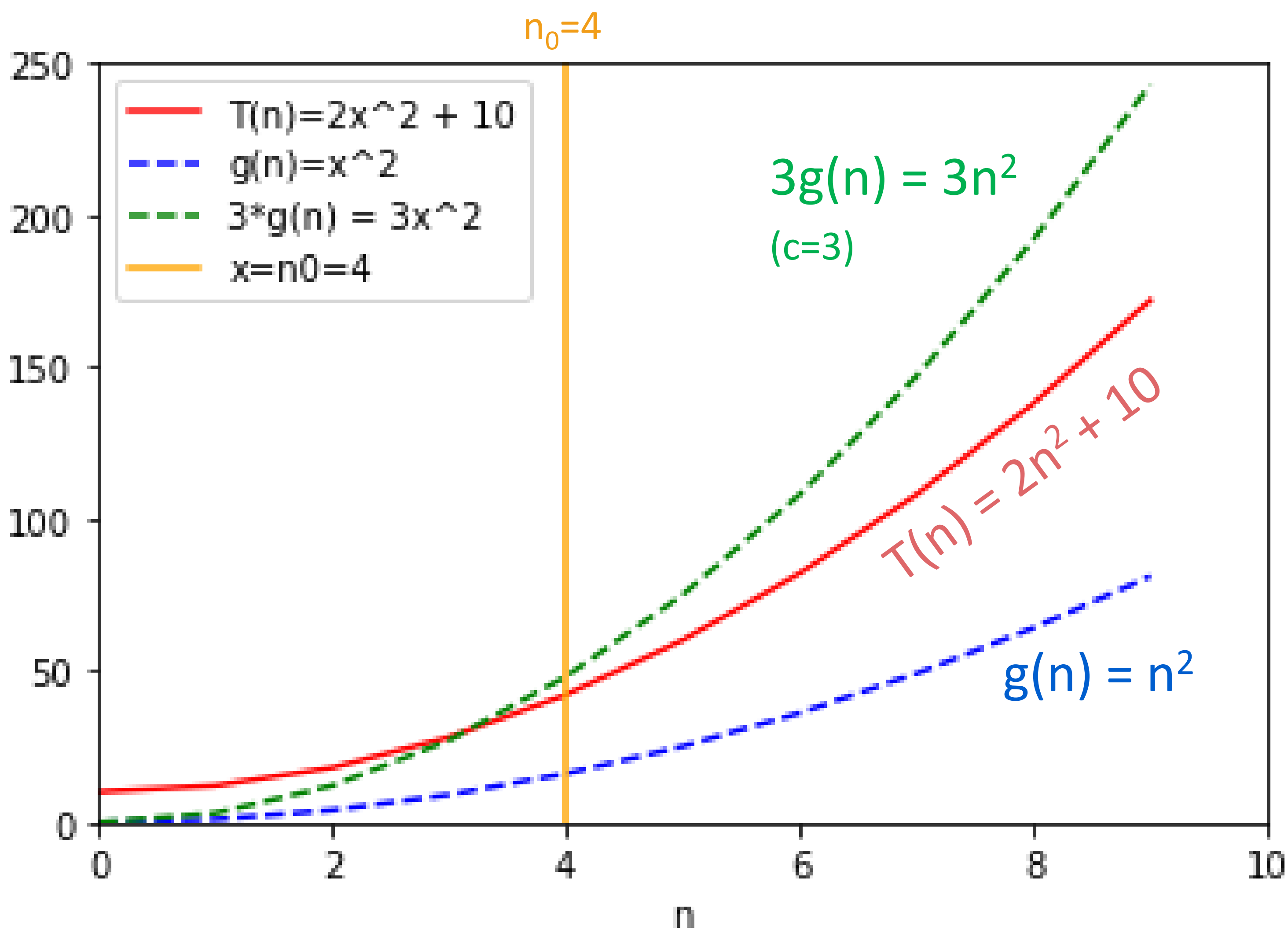
$$T(n) = O\big(g(n)\big) \iff \exists c, n_0 > 0 \ \ s.t. \ \ \forall n \geq n_0, \ T(n) \leq c \cdot g(n)$$



Příklad

$$2n^2 + 10 = O(n^2)$$

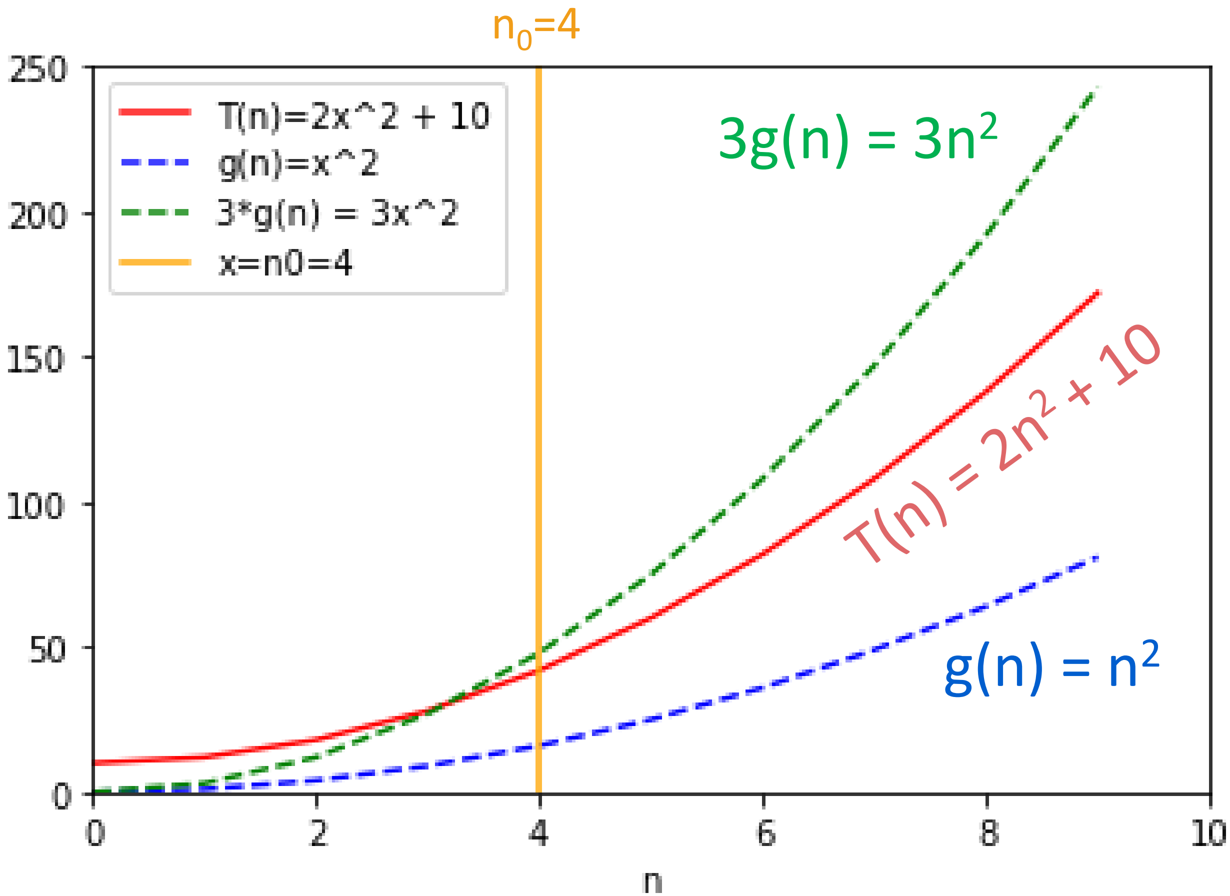
$$T(n) = O\big(g(n)\big)$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \ \ s.t. \ \ \forall n \geq n_0,$$
$$T(n) \leq c \cdot g(n)$$



Příklad

$2n^2 + 10 = O(n^2)$

$$T(n) = O\big(g(n)\big)$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$T(n) \leq c \cdot g(n)$$



Formálně:

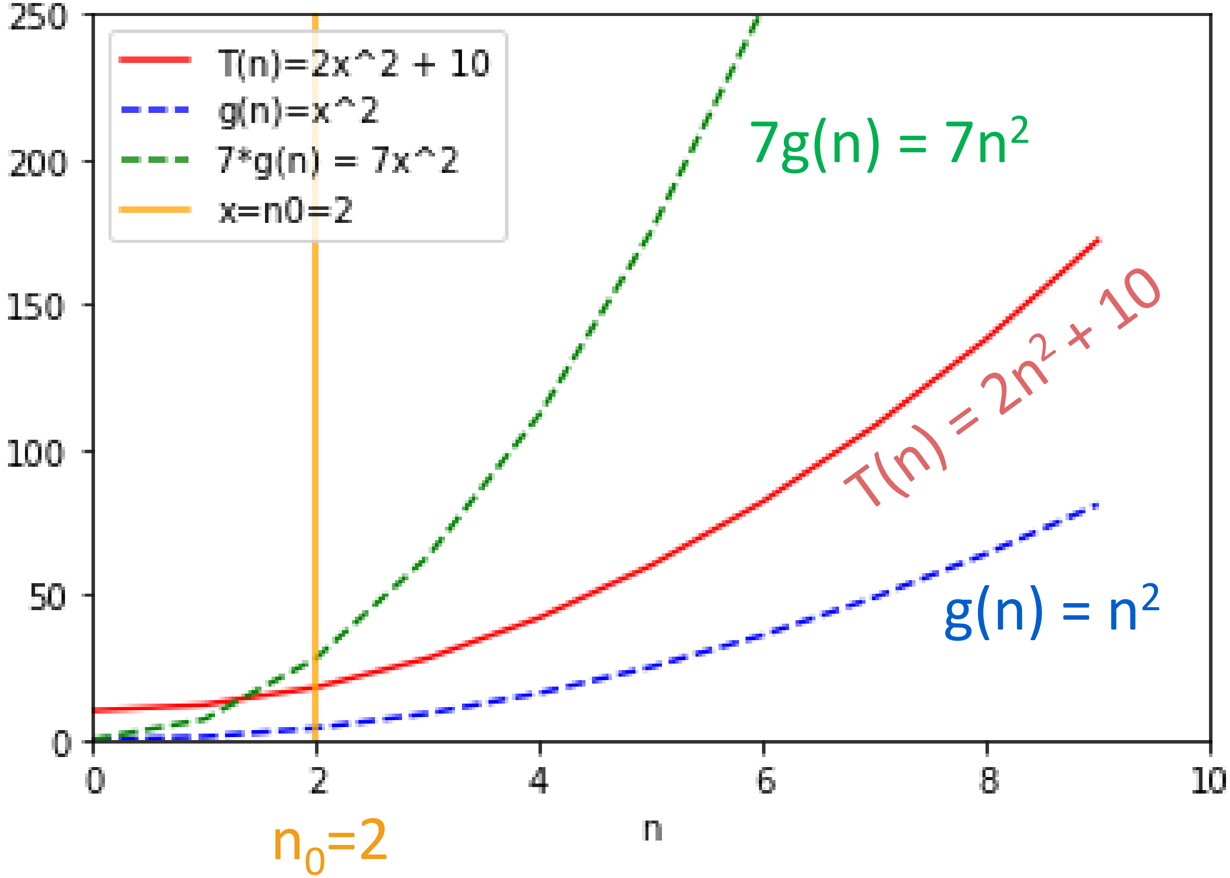
- Zvolíme c = 3
- Zvolíme n_0 = 4
- Potom:

$$\forall n \geq 4,$$
$$2n^2 + 10 \leq 3 \cdot n^2$$

Jiný příklad

$$2n^2 + 10 = O(n^2)$$

$$T(n) = O\big(g(n)\big)$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$T(n) \leq c \cdot g(n)$$



Formálně:

- Zvolíme c = 7
- Zvolíme n_0 = 2
- Potom:

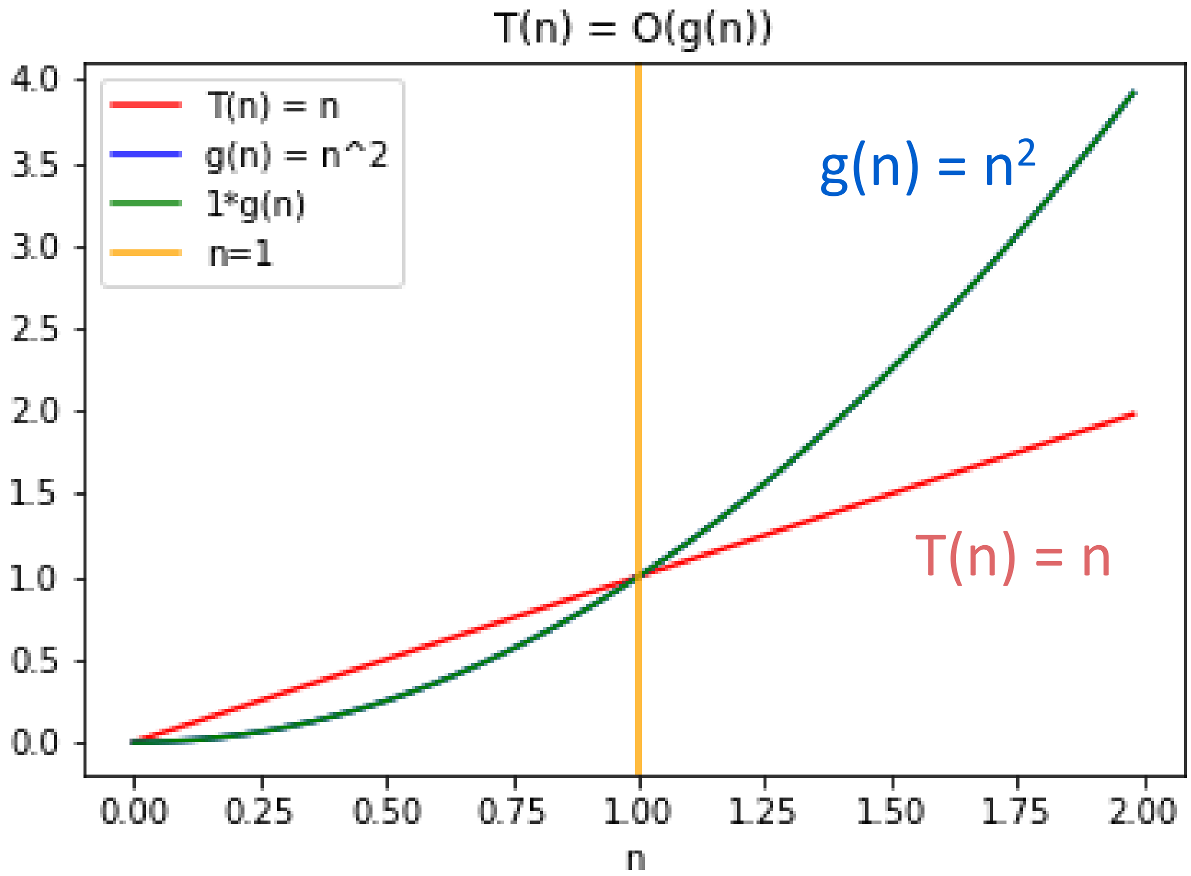
$$\forall n \geq 2,$$
$$2n^2 + 10 \leq 7 \cdot n^2$$

Neexistuje „správná“ volba c a n_0

O(…) je horní mez :

$$n = O(n^2)$$

$$T(n) = O(g(n))$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$T(n) \leq c \cdot g(n)$$



- Zvolíme c = 1
- Zvolíme n_0 = 1
- Potom

$$\forall n \geq 1,$$
$$n \leq n^2$$

$\Omega(\ldots)$ znamená dolní hranici

- Řekneme, že "$T(n)$ je $\Omega\big(g(n)\big)$" jestliže pro dostatečně velká n, $T(n)$ je nejméně tak velká jako funkce $g(n)$ vynásobená zvolenou konstantou.

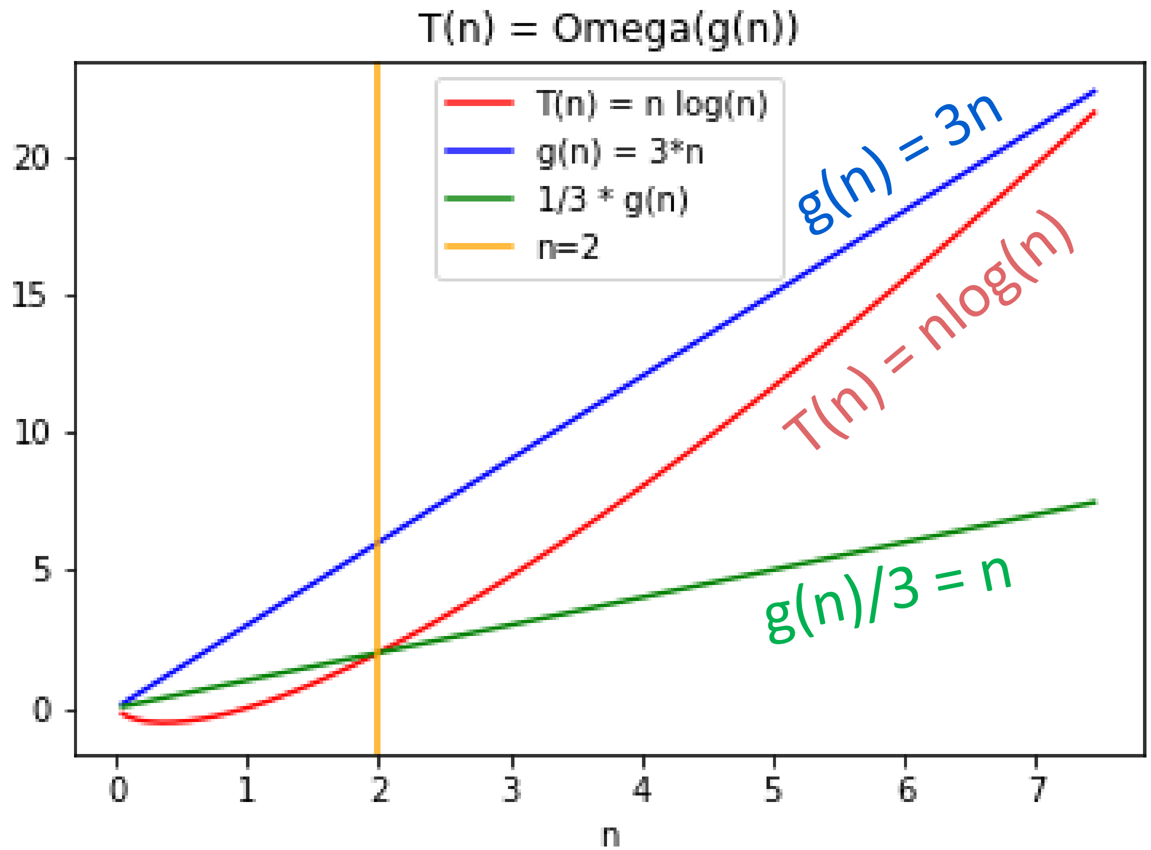
- Formálně,

$$T(n) = \Omega\big(g(n)\big)$$
$$\Longleftrightarrow$$
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$c \cdot g(n) \leq T(n)$$

Příklad

$n \log_2(n) = \Omega(3n)$

$$T(n) = \Omega\big(g(n)\big)$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$c \cdot g(n) \leq T(n)$$



- Zvolíme c = 1/3
- Zvolíme n_0 = 2
- Potom

$$\forall n \geq 2,$$

$$\frac{3n}{3} \leq n \log_2(n)$$

$\Theta(\ldots)$ znamená obojí!

- Řekneme, že "$T(n)$ je $\Theta(g(n))$" jestliže platí zároveň:

$$T(n) = O\big(g(n)\big)$$

a

$$T(n) = \Omega\big(g(n)\big)$$

$$T(n) = O\big(g(n)\big)$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \ \ s.t. \ \ \forall n \geq n_0,$$
$$T(n) \leq c \cdot g(n)$$

Dokažmě, že: n^2 není O(n)

- Důkaz sporem:
- Předpokládejme, že $n^2 = O(n)$.
- Potom existují nějaká kladná čísla c a n_0 taková, že:

$$\forall n \geq n_0, \qquad n^2 \leq c \cdot n$$

- Podělíme obě strany číslem n:

$$\forall n \geq n_0, \qquad n \ \leq c$$

- To ale není pravda!!!
- Co když řekneme, že $n_0 = \mathrm{c} + 1$?
    - Potom pro $n \geq c + 1$ vychází $n \ \leq c$
- Spor!

Podobné příklady

- Abychom dokázali, že T(n) = O(g(n)), musíme přijít s c a n_0 takovými, že splňují definici.

- Abychom dokázali, že T(n) NENÍ O(g(n)), jeden ze způsobů je **důkaz sporem**:
  - Předpokládejme (abychom se později dostali ke sporu), že „něco“ nám dá *c* a an n_0 takové, že definice ***je*** splněna.
  - Poté ukážeme, že to „něco“ nám dává nepravdivou informaci a tím „odvodíme“ **spor**.

tedy

- Řekneme, že $p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$ je polynom stupně $k \geq 1$.

- Potom:
  1. $p(n) = O\left(n^k\right)$
  2. $p(n)$ není $O\left(n^{k-1}\right)$

Další příklady

- $n^3 + 3n = O(n^3 - n^2)$
- $n^3 + 3n = \Omega(n^3 - n^2)$
- $n^3 + 3n = \Theta(n^3 - n^2)$

- 3^n **NENÍ** $O(2^n)$
- $\log_2(n) = \Omega(\ln(n))$
- $\log_2(n) = \Theta(2^{\log\log(n)})$

Zkuste sami provést důkazy!

K zamyšlení, poznámka

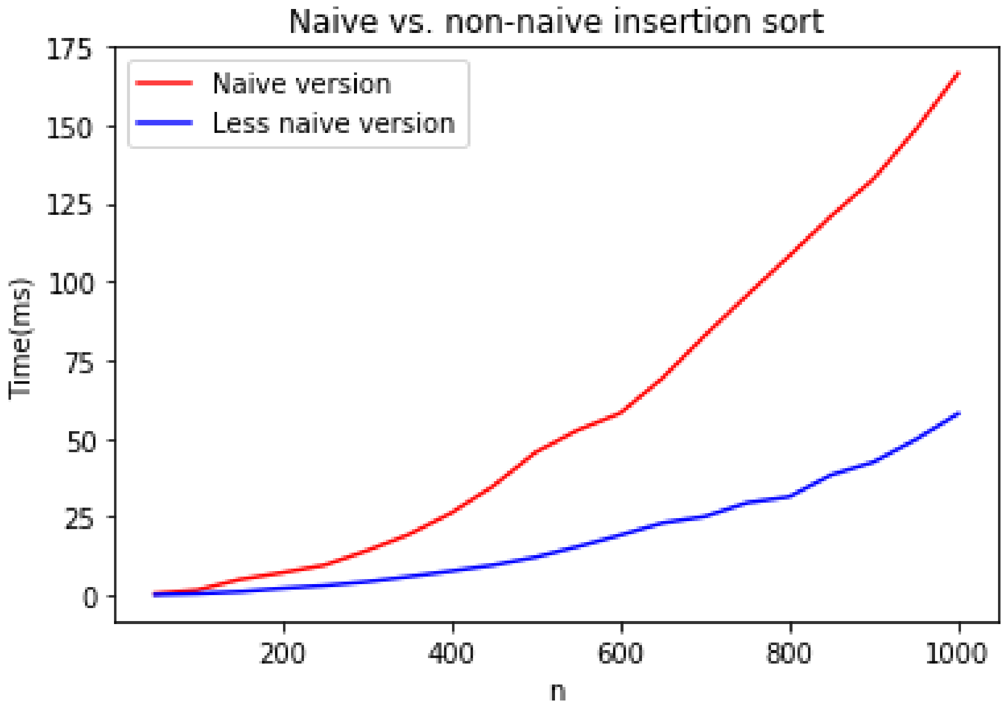
- Existují funkce f, g takové, že ANI f = O(g) ani f = Ω(g)?
- Existují nějaké ne-klesající funkce f, g takové, že shora tvrzení platí?

Poznámka

- Asymptotická notace
- Zanedbáváme nižší řády a multiplikativní konstanty, ale musíme být přitom opatrní.
- Zde probíráme praktické algoritmy, takže nemusíme volit $n \geq n_0 = 2^{10000000}$.

Zpět k insert sortu

1. Pracuje?
2. Je rychlý?

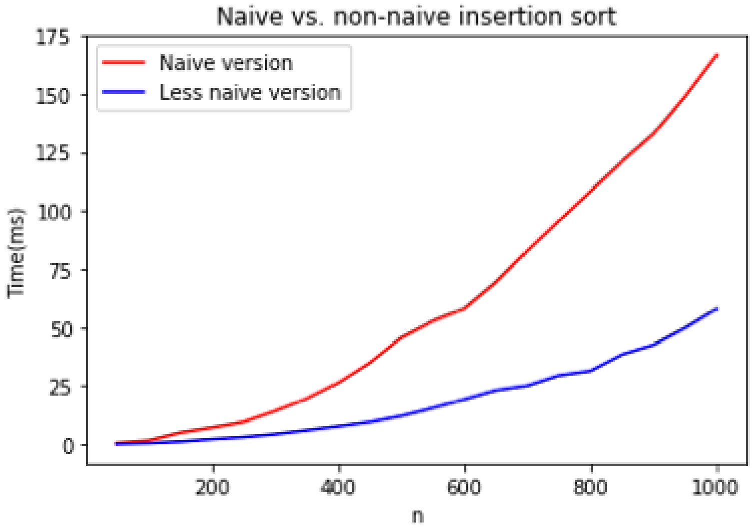


Insertion Sort: doba běhu

- Počet operací byl:
    - $2n^2 - n - 1$ přiřazení
    - $2n^2 - n - 1$ inkrement/dekrement
    - $2n^2 - 4n + 1$ porovnání
    - …

- Doba běhu je $O(n^2)$

Podívej se na pseudokód a přesvědč se o tom!



To se zdá rozumné

Insert sort: doba běhu

Často můžeme odhadnou dobu běhu z pseudokódu, například bychom odhadli…

```
def InsertionSort(A):
    for i in range(1,len(A)):
        prvek = A[i]
        j = i-1
        while j >= 0 and A[j] > prvek:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = prvek
```

n-1 iteraci ve vnějším cyku

V nejhorším případě přibližně n iterací v tomto vnitřním cyklu

Tedy vnitřní cyklus se opakuje n x, tedy dostáváme O(n^2).”

Závěr

Insert sort je algoritmus, který správně seřadí libovolné n-rozměrné pole v čase $O(n^2)$.

Dle řádu funkce dostaly některé složitosti i své jméno

- O(1) –konstantní
- O(log N) –logaritmická
- O(N) –lineární
- O(N log N) –lineárně-logaritmická (supralineární)
- O(N^2) –kvadratická
- O(N^3) –kubická
- Obecně O(N^X) –polynomiální
- Obecně O(X^N) –exponenciální
- O(N!) –faktoriálová

Pro představu: Máme počítač s rychlostí 1 megaFLOPS. Jak dlouho bude počítat?

složitost / N	**10**	**20**	**40**	**60**	**500**	**1000**
log_2 N	3,3 μs	4,3 μs	5 μs	5,8 μs	9 μs	10 μs
N	10 μs	20 μs	40 μs	60 μs	0,5 ms	1 ms
N log_2 N	33 μs	86 μs	0,2 ms	0,35 ms	4,5 ms	10 ms
N^2	0,1 ms	0,4 ms	1,6 ms	3,6 ms	0,25 s	1 s
N^3	1 ms	8 ms	64 ms	0,2 s	125 s	17 min
N^4	10 ms	160 ms	2,56 s	13 s	17 hod	11,6 dnů
2^N	1 ms	1 s	12,7 dnů	36000 let	10^{137} let	10^{287} let
N!	3,6 s	77000 let	10^{34} let	10^{68} let	10^{1110} let	10^{2554} let

„Domácí úkol": Zjistěte, jaký výkon má současný nejrychlejší superpočítač. Jak dlouho by řešil algoritmus s exponenciální složitostí pro N=1000?

Prostorová (paměťová) složitost

- Složitost prostorová – spotřeba paměti, diskového prostoru v závislosti na vstupních datech.
- Lze analyzovat podobně, jako jsem to dělali u časové složitosti.
- Časová složitost je to, o co nám většinou jde. Nebudeme se prostorovou (paměťovou) složitostí dále zabývat.

Další algoritmy

- Selection sort (Řazení výběrem)
- Bubble sort (Metoda přímé výměny)

Selection Sort

- Selection sort (Řazení výběrem)
- Snaha minimalizovat počet časově náročných operací – přesunů
- V každém kroku i algoritmu se vybere nejmenší prvek posloupnosti A[i]..A[n−1]a vymění se s prvkem A[i]
- Algoritmus

```
for(int i=0; i<n-1; i++) {
        vyber nejmenší prvek A[k] z A[i]..A[n-1]
        vyměň A[k] s A[i]
}
```

Selection Sort – ukázka kódu

```
// Vstup: nesetříděné pole <type> [] A
// Výstup: setříděnépole <type> [] A

for ( int i=0; i<n-1; i++) {
        k=i; x=A[i];
        for( int j=i+1; j<n; j++) {
                if( A[j] <x) {
                        k=j; x=A[j];
                }
        }
        if( k!=i) {
                t=A[k]; A[k]=A[i]; A[i]=t;
        }
}
```

Vlastnosti algoritmu Selection Sort

- Složitost?

$O(n^2)$

- Co se zlepšilo?
  - počet přesunů
- Za jakou cenu?
  - zvýšení počtu porovnání

Bubble Sort (metoda přímé výměny)

- Bubble sort
- Založený na postupném porovnání a případné záměně sousedních prvků
- Prvky s nízkým klíčem postupně „probublávají” na začátek pole (možný je i opačný postup, kdy prvky s vysokou hodnotou „probublávají“ na konec pole).

Bubble Sort – příklad kódu

```
// Vstup: nesetříděné pole <type> [] A
//Výstup: setříděné pole <type> [] A
for ( int i = 1; i < n-1; i++) {
		for (int j = n-1; j >= i; j--)	{
				if(A[j-1] > A[j])
				{
						t = A[j-1];
						A[j-1] = A[j];
						A[j] = t;
				}
		}
}
```

Bubble Sort

- Vlastnosti
- Složitost: $O(n^2)$
- Co jsme ušetřili?
  - nic! – bublinkové třídění je ve většině případů nejpomalejší
- Kdy je dobré?
  - když je pole téměř setříděno

Konec – co jsme probrali

- Pojem asymptotická složitost algoritmů
- Algoritmy patřící do skupiny kvadratických algoritmů, které správně seřadí libovolné n-rozměrné pole v čase $O(n^2)$ (říkáme také se složitostí $O(n^2)$):
  - Insert sort – analyzovali jsme podrobněji
  - Selection sort
  - Bubble sort