Výpočetní složitost algoritmů

Složitost algoritmu

- Počítače pracují rychle, ale ne nekonečně rychle. Provedení každé instrukce trvá nějakou (i když velmi krátkou) dobu.
- Stejný problém může řešit více různých algoritmů a doba výpočtu (daná hlavně počtem provedených instrukcí) může být pro různé algoritmy různá.
- Algoritmy bychom chtěli mezi sebou porovnávat a zvolit si ten lepší.
- Algoritmy můžeme naprogramovat a změřit čas výpočtu. Tím zjistíme jak dlouho trvá výpočet na konkrétních datech, na kterých algoritmus testujeme.
- Chtěli bychom mít i nějakou přesnější představu o tom, jak dlouho bude trvat výpočet na všech možných vstupních datech.

Složitost algoritmu

- Doba výpočtu je ovlivněna mnoha faktory, např.:
  - použitý algoritmus
  - množství vstupních dat
  - použitý hardware (důležitá může být např. taktovací frekvence procesoru)
  - použitý programovací jazyk — a jeho konkrétní implementace (překladač/interpreter)
  - . . .

- Pokud potřebujeme řešit problém pro „malá“ vstupní data, doba výpočtu je většinou zanedbatelná.

- S narůstajícím množstvím vstupních dat (velikosti vstupu) může doba výpočtu růst, někdy velmi výrazně.

Složitost algoritmu

- **Časová složitost algoritmu** — jak závisí doba výpočtu na množství vstupních dat
- **Paměťová** (resp. **prostorová**) **složitost algoritmu** — jak závisí množství použité paměti na množstí vstupních dat

Poznámka: Přesné definice těchto pojmů budou uvedeny za chvíli.

Poznámka:

- Existují i další typy výpočetní složitosti, kterými se nebudeme zabývat (např. komunikační složitost).

Složitost algoritmu

Vezměme si nějaký konkrétní stroj vykonávající nějaký algoritmus — např. stroj RAM, Turingův stroj, ...

Budeme předpokládat, že pro daný stroj $\mathcal{M}$ máme nějak definované pro libovolný vstup w z množiny všech vstupů In následující dvě funkce:

- $time_{\mathcal{M}} : In \rightarrow \mathbb{N}$ — vyjadřuje dobu výpočtu stroje $\mathcal{M}$ nad vstupem w
- $space_{\mathcal{M}} : In \rightarrow \mathbb{N}$ — vyjadřuje množství paměti použité strojem $\mathcal{M}$ při výpočtu nad vstupem w

Poznámka: Předpokládáme, že výpočet stroje $\mathcal{M}$ nad libovolným vstupem w se po konečném počtu kroků zastaví.

Složitost algoritmu

Příklad:

- Jednopáskový Turingův stroj $\mathcal{M}$:
  - $\mathit{time}_{\mathcal{M}}(w)$ — počet kroků, které vykoná $\mathcal{M}$ při výpočtu nad vstupem w
  - $\mathit{space}_{\mathcal{M}}(w)$ — počet políček navštívených na pásce během výpočtu nad vstupem w
- Stroj RAM:
  - $\mathit{time}_{\mathcal{M}}(w)$ — počet kroků, které vykoná daný stroj RAM při výpočtu nad vstupem w
  - $\mathit{space}_{\mathcal{M}}(w)$ — počet buněk poměti, které byly použity během výpočtu nad vstupem w (bylo do nich něco zapsáno nebo z nich bylo čteno)

Velikost vstupu

Pro různé vstupy provede program různý počet instrukcí.

Pokud chceme počet provedených instrukcí nějak analyzovat, je vhodné si zavést pojem **velikost vstupu**.

Typicky je velikost vstupu číslo, které udává, jak je daná instance „velká“ (čím větší číslo, tím větší instance).

Poznámka: Velikost vstupu si v daném konkrétním případě můžeme definovat, jak chceme a jak je to pro další analýzu výhodné.

Co přesně zvolíme jako velikost vstupu není předem dáno, ale z podstaty zadaného problému většinou nějak přirozeně vyplývá, co za velikost vstupu zvolit.

Velikost vstupu

Příklady:

- Pro problém „Třídění“, kde vstupem je sekvence čísel $a_1, a_2, \ldots, a_n$ a výstupem jsou tato čísla setříděná, můžeme vzít jako velikost vstupu hodnotu n.
- Pro problém „Prvočíselnost“, kde vstupem je přirozené číslo x, a kde se ptáme, zda x je prvočíslo, můžeme vzít jako velikost vstupu počet bitů čísla x.

  (Jinou možností by bylo vzít jako velikost vstupu přímo hodnotu x.)

Velikost vstupu

Někdy je vhodné popsat velikost vstupu pomocí více čísel.

Například u problémů, kde vstupem je graf, můžeme definovat velikost vstupu jako dvojici čísel n, m, kde:

- n – počet vrcholů grafu
- m – počet hran grafu

Poznámka: Jinou možností by bylo definovat velikost vstupu jako jediné číslo $n + m$.

Velikost vstupu

Obecně můžeme pro libovolný problém definovat velikost vstupu následovně:

- Pokud je vstupem slovo w z nějaké obecedy Σ:
  délka slova w
- Pokud je vstupem sekvence bitů (tj. slovo z abecedy $\{0,1\}$):
  počet bitů v této sekvenci
- Pokud je vstupem přirozené číslo x:
  počet bitů nutných k zápisu čísla x

Časová složitost

Chceme analyzovat konkrétní algoritmus (jeho konkrétní implementaci).

Zajímá nás, kolik instrukcí se provede, pokud algoritmus dostane vstup velikosti $0, 1, 2, 3, 4, \ldots$.

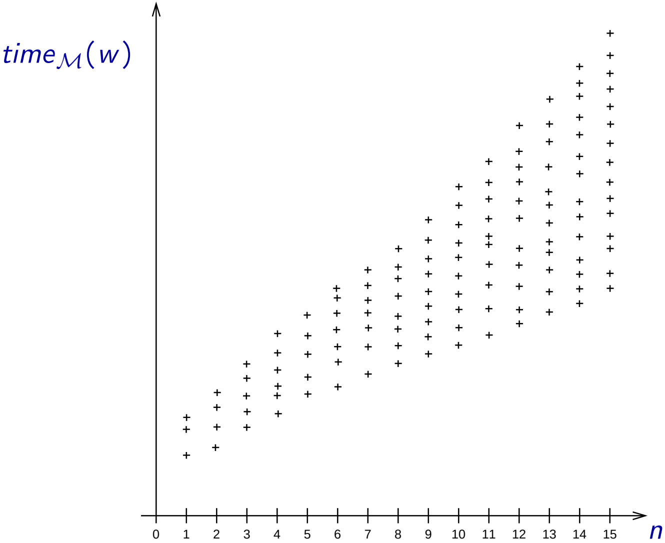
Je zřejmé, že i pro vstupy, které mají stejnou velikost, může být počet provedených instrukcí různý.

Označme si velikost vstupu $w \in In$ jako $size(w)$.

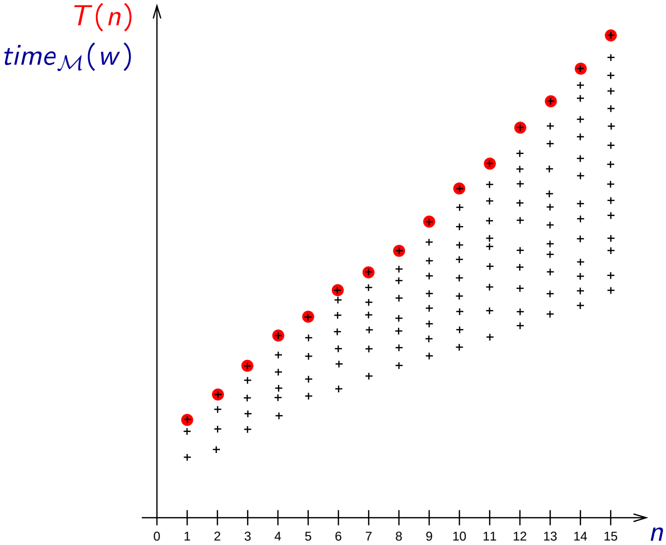
Nyní definujme následující funkci $T : \mathbb{N} \to \mathbb{N}$ takovou, že pro $n \in \mathbb{N}$ je

$$T(n) = \max\{\, time_{\mathcal{M}}(w) \mid w \in In,\ size(w) = n \,\}$$

Časová složitost v nejhorším případě



Časová složitost v nejhorším případě



Časová a prostorová složitost v nejhorším případě

Takto definované funkci $T(n)$ (tj. funkci, která pro daný algoritmus a danou definici velikosti vstupu přiřazuje každému přirozenému číslu n maximální počet instrukcí, které algoritmus provede, pokud dostane vstup velikosti n) se říká **časová složitost algorimu v nejhorším případě**.

$$T(n) = \max\{\, time_{\mathcal{M}}(w) \mid w \in In,\, size(w) = n \,\}$$

Analogicky můžeme definovat **prostorovou (paměťovou) složitost algoritmu v nejhorším případě** jako funkci $S(n)$, kde as a function $S(n)$ where:

$$S(n) = \max\{\, space_{\mathcal{M}}(w) \mid w \in In,\, size(w) = n \,\}$$

Časová složitost v průměrném případě

Kromě časové složitosti v nejhorším případě má smysl zkoumat i časovou složitost **v průměrném případě**.

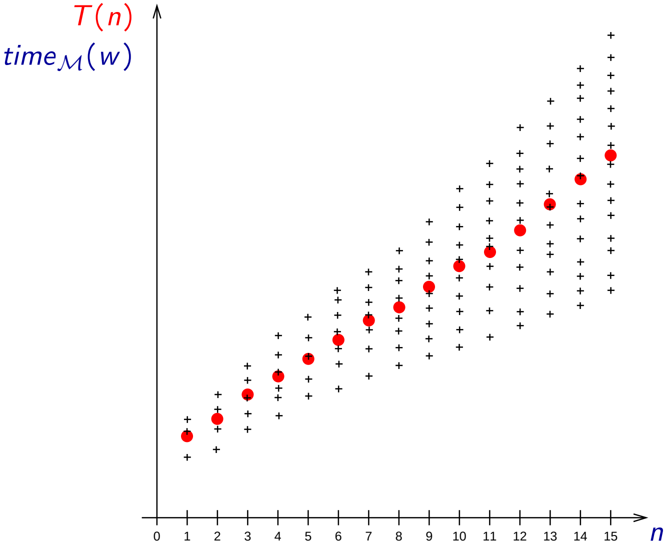
V tomto případě $T(n)$ nedefinujeme jako maximum, ale jako aritmetický průměr z hodnot

$$\{\, time_{\mathcal{M}}(w) \mid w \in In,\ size(w) = n \,\}$$

- Určit časovou složitost v průměrném případě je většinou těžší než určit časovou složitost v nejhorším případě.
- Často se tyto dvě funkce příliš neliší, někdy je ale rozdíl významný.

Poznámka: Zkoumat složitost v **nejlepším případě** většinou moc smysl nemá.

Časová složitost v průměrném případě



Výpočetní složitost algoritmu

Z definice vidíme, že jak časová, tak prostorová, složitost algoritmu jsou funkce, jejíchž přesné hodnoty závistí nejen na daném algoritmu *Alg*, ale také na následujících věcech:

- na stroji $\mathcal{M}$, na kterém algoritmus *Alg* běží,
- na definici doby výpočtu $\mathit{time}_{\mathcal{M}}(w)$ a množství použité paměti $\mathit{space}_{\mathcal{M}}(w)$ algoritmu *Alg* na stroji $\mathcal{M}$ pro vstup $w \in \mathit{In}$,
- na definici velikosti vstupu (tj. definici funkce *size*).

Výpočetní složitost algoritmu

Přesné určení doby výpočtu nebo množství použité paměti může být extrémně komplikované.

Většinou se při analýze výpočetní složitosti algoritmu používá celá řada zjednodušení:

- Většinou se neanalyzuje, jak závisí doba výpočtu nebo množství použité paměti na konkrétních vstupních datech, ale pouze, jak závistí na **velikosti vstupu**, tj. na množství těchto dat.
- Funkce vyjadřující, jak roste doba výpočtu nebo množství použité paměti v závislosti na velikosti vstupu, se nepočítají přesně — počítají se **odhady** těchto funkcí.
- Odhady těchto funkcí se vyjadřují pomocí tzv. **asymptotické notace** — např. se řekne, že časová složitost algoritmu MergeSort je $O(n \log n)$, zatímco časová složitost algoritmu BubbleSort je $O(n^2)$.

Časová složitost algoritmu

Příklad analýzy časové složitosti algoritmu **bez** použití asymptotické notace:

- Takto podrobně se analýza výpočetní složitosti algoritmu téměř nikdy **nedělá** — je to příliš pracné a komplikované.
- Uvidíme tak ale, co vše je při použití asymptotické notace zanedbáno a o kolik je analýza s použitím asymptotické notace jednodušší.
- Budeme počítat s konstantami $c_0, c_1, \ldots, c_k$, které udávají dobu trvání jednotlivých instrukcí — nebudeme počítat s konkrétními čísly.

Doba výpočtu

Řekněme, že máme algoritmus reprezentován ve formě grafu řídícího toku:

- Každé instrukci (tj. každé hraně) přiřadíme hodnotu udávající, jak dlouho trvá provedení této instrukce.
- Provedení různých instrukcí může trvat různou dobu.
- Pro jednoduchost předpokládejme, že provedení té samé instrukce trvá pokaždé stejnou dobu — hodnota přiřazená dané instrukci je číslo z množiny $\mathbb{R}_+$ (množina nezáporných reálných čísel).

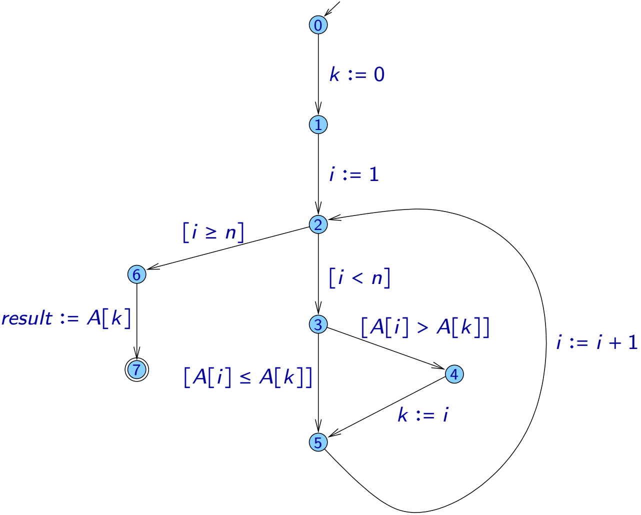
Doba výpočtu

Příklad:

Algoritmus: Nalezení největšího prvku v poli

```
FIND-MAX ($A$, $n$):
    $k := 0$
    for $i := 1$ to $n - 1$ do
        if $A[i] > A[k]$ then
            $k := i$
    return $A[k]$
```

Doba výpočtu



Instr.	doba
$k := 0$	c_0
$i := 1$	c_1
$[i < n]$	c_2
$[i \geq n]$	c_3
$[A[i] \leq A[k]]$	c_4
$[A[i] > A[k]]$	c_5
$k := i$	c_6
$i := i + 1$	c_7
$result := A[k]$	c_8

Doba výpočtu

Příklad: Doby provedení jednotlivých instrukcí by mohly být třeba:

Instr.	označení	doba
$k := 0$	c_0	4
$i := 1$	c_1	4
$[i < n]$	c_2	10
$[i \geq n]$	c_3	12
$[A[i] \leq A[k]]$	c_4	14
$[A[i] > A[k]]$	c_5	12
$k := i$	c_6	5
$i := i + 1$	c_7	6
$result := A[k]$	c_8	5

Pro konkrétní vstup w, např. pro $w = ([3, 8, 4, 5, 2], 5)$, bychom mohli výpočet odsimulovat a určit konkrétní dobu výpočtu $t(w)$.

Časová složitost algoritmu

Předpokládáme vstupy tvaru (A, n), kde A je pole a n počet prvků tohoto pole (přičemž $n \geq 1$).

Jako velikost vstupu (A, n) zvolme n.

Uvažujme nyní o nějaké jednom vstupu $w = (A, n)$ velikosti n:

- Dobu výpočtu $t(w)$ nad vstupem w můžeme vyjádřit jako

  $$t(w) = c_0 \cdot m_0(w) + c_1 \cdot m_1(w) + \cdots + c_8 \cdot m_8(w),$$

  kde $m_0, m_1, \ldots, m_8$ jsou funkce udávající, kolikrát je daná instukce při výpočtu nad vstupem w provedena.

Časová složitost algoritmu

Instr.	doba	počet provedení	hodnota $m_i(w)$
$k := 0$	c_0	$m_0(w)$	1
$i := 1$	c_1	$m_1(w)$	1
$[i < n]$	c_2	$m_2(w)$	$n - 1$
$[i \geq n]$	c_3	$m_3(w)$	1
$[A[i] \leq A[k]]$	c_4	$m_4(w)$	$n - 1 - \ell$
$[A[i] > A[k]]$	c_5	$m_5(w)$	ℓ
$k := i$	c_6	$m_6(w)$	ℓ
$i := i + 1$	c_7	$m_7(w)$	$n - 1$
$result := A[k]$	c_8	$m_8(w)$	1

ℓ — počet průchodů cyklem, kdy platí $A[i] > A[k]$ (zjevně je $0 \leq \ell < n$)

Časová složitost algoritmu

Dosazením do

$$t(w) \;=\; c_0 \cdot m_0(w) + c_1 \cdot m_1(w) + \;\cdots\; + c_8 \cdot m_8(w),$$

dostaneme

$$t(w) \;=\; d_1 \;+\; d_2 \cdot (n-1) \;+\; d_3 \cdot (n-1-\ell) \;+\; d_4 \cdot \ell,$$

kde

$$d_1 = c_0 + c_1 + c_3 + c_8 \qquad d_3 = c_4$$
$$d_2 = c_2 + c_7 \qquad d_4 = c_5 + c_6$$

Po úpravě je

$$t(w) \;=\; (d_2 + d_3) \cdot n \;+\; (d_4 - d_3) \cdot \ell \;+\; (d_1 - d_2 - d_3)$$

Poznámka: $t(w)$ není časová složitost, ale doba výpočtu pro konkrétní vstup w

Časová složitost algoritmu

Například pokud budou doby provedení jednotlivých instrukcí následující:

Instr.	označení	doba
$k := 0$	c_0	4
$i := 1$	c_1	4
$[i < n]$	c_2	10
$[i \geq n]$	c_3	12
$[A[i] \leq A[k]]$	c_4	14
$[A[i] > A[k]]$	c_5	12
$k := i$	c_6	5
$i := i + 1$	c_7	6
$result := A[k]$	c_8	5

bude $d_1 = 25$, $d_2 = 16$, $d_3 = 14$ a $d_4 = 17$.

V takovém případě je $t(w) = 30n + 3\ell - 5$.

Pro konkrétní vstup $w = ([3, 8, 4, 5, 2], 5)$ je $n = 5$ a $\ell = 1$, takže $t(w) = 30 \cdot 5 + 3 \cdot 1 - 5 = 148$.

Časová složitost algoritmu

Pro které vstupy velikosti n bude výpočet trvat nejdéle (tj. které vstupy představují nejhorší případ), může záviset na detailech implementace a přesných hodnotách konstant:

Doba výpočtu algoritmu $\textsc{Find-Max}$ pro vstup $w = (A, n)$ velikosti n:

$$t(w) = (d_2 + d_3) \cdot n + (d_4 - d_3) \cdot \ell + (d_1 - d_2 - d_3)$$

- Pokud $d_3 \geq d_4$ — nejhorší jsou případy, kdy má ℓ co nejmenší hodnotu $\ell = 0$ — například vstupy tvaru $[0, 0, \ldots, 0]$ nebo třeba $[n,\ n-1,\ n-2,\ \ldots,\ 2,\ 1]$
- Pokud $d_3 \leq d_4$ — nejhorší jsou případy, kdy má ℓ co největší hodnotu $\ell = n - 1$ — například vstupy tvaru $[0, 1, \ldots, n-1]$

Časová složitost algoritmu

Časová složitost $T(n)$ algoritmu Find-Max v nejhorším případě je tedy dána následovně:

- Pokud $d_3 \geq d_4$:

$$T(n) = (d_2 + d_3) \cdot n + (d_1 - d_2 - d_3)$$

- Pokud $d_3 \leq d_4$:

$$\begin{aligned} T(n) &= (d_2 + d_3) \cdot n \ + \ (d_4 - d_3) \cdot (n - 1) \ + \ (d_1 - d_2 - d_3) \\ &= (d_2 + d_4) \cdot n \ + \ (d_1 - d_2 - d_4) \end{aligned}$$

Příklad: Pro $d_1 = 25$, $d_2 = 16$, $d_3 = 14$ a $d_4 = 17$ bude

$$\begin{aligned} T(n) &= (16 + 17) \cdot n \ + \ (25 - 16 - 17) \\ &= 33n - 8 \end{aligned}$$

Časová složitost algoritmu

V obou případech (ať už $d_3 \geq d_4$ nebo $d_3 \leq d_4$) bude časová složitost algoritmu Find-Max funkce tvaru

$$T(n) = an + b$$

kde a a b jsou nějaké konstanty, jejichž přesné hodnoty závisí na délce trvání jednotlivých instrukcí.

Poznámka: Konkrétně bychom tyto konstanty mohli vyjádřit jako

$$a = d_2 + \max\{d_3, d_4\} \qquad\qquad b = d_1 - d_2 - \max\{d_3, d_4\}$$

Například

$$T(n) = 33n - 8$$

Časová složitost algoritmu

Pokud bychom se spokojili s tím, že časová složitost algoritmu Find-Max je nějaká funkce tvaru

$$T(n) = an + b,$$

kde by nás ale nezajímaly konkrétní hodnoty konstant a a b, celá analýza mohla být výrazně jednodušší.

- Ve skutečnosti ani většinou nechceme vědět, jak přesně funkce $T(n)$ vypadá (obecně to může být nějaká velmi komplikovaná funkce), a stačilo by nám, že víme, že hodnoty funkce $T(n)$ „zhruba“ odpovídají hodnotám nějaké funkce $S(n) = an + b$, kde a a b jsou nějaké konstanty.

Časová složitost algoritmu

U dané funkce $T(n)$ vyjadřující časovou nebo paměťovou složitost se tak většinou spokojíme s jejím přibližným vyjádřením — **odhadem**, kde

- zanedbáme méně významné členy
  (např. ve funkci $T(n) = 15n^2 + 40n - 5$ zanedbáme členy $40n$ a -5 a místo původní funkce budeme uvažovat jen o funkci $T(n) = 15n^2$),
- zanedbáme konstanty, kterými se násobí
  (např. místo funkce $T(n) = 15n^2$ budeme uvažovat
  o funkci $T(n) = n^2$)
- konstanty v exponentech ignorovat nebudeme — například je podstatný rozdíl mezi funkcemi $T_1(n) = n^2$ a $T_2(n) = n^3$.
- bude nás zajímat, jak se funkce $T(n)$ chová pro „velké“ hodnoty n, chování na malých hodnotách budeme ignorovat

Rychlost růstu funkcí

Program zpracovává vstup velikosti n.
Předpokládejme, že pro vstup velikosti n provede $T(n)$ operací, a že provedení jedné operace trvá $1\,\mu\mathrm{s}$ $(10^{-6}\,\mathrm{s})$.

	n							
$T(n)$	20	40	60	80	100	200	500	1000
n	$20\,\mu$s	$40\,\mu$s	$60\,\mu$s	$80\,\mu$s	0.1 ms	0.2 ms	0.5 ms	1 ms
$n \log n$	$86\,\mu$s	0.213 ms	0.354 ms	0.506 ms	0.664 ms	1.528 ms	4.48 ms	9.96 ms
n^2	0.4 ms	1.6 ms	3.6 ms	6.4 ms	10 ms	40 ms	0.25 s	1 s
n^3	8 ms	64 ms	0.216 s	0.512 s	1 s	8 s	125 s	16.7 min.
n^4	0.16 s	2.56 s	12.96 s	42 s	100 s	26.6 min.	17.36 hod.	11.57 dní
2^n	1.05 s	12.75 dní	36560 let	$38.3\cdot10^{9}$ let	$40.1\cdot10^{15}$ let	$50\cdot10^{45}$ let	$10.4\cdot10^{136}$ let	–
$n!$	77147 let	$2.59\cdot10^{34}$ let	$2.64\cdot10^{68}$ let	$2.27\cdot10^{105}$ let	$2.96\cdot10^{144}$ let	–	–	–

Rychlost růstu funkcí

Uvažujme 3 algoritmy se složitostmi $T_1(n) = n$, $T_2(n) = n^3$, $T_3(n) = 2^n$. Náš počítač zvládne v reálném čase (kolik jsme ochotni počkat) 10^{12} kroků.

Složitost	Velikost vstupu
$T_1(n) = n$	10^{12}
$T_2(n) = n^3$	10^4
$T_3(n) = 2^n$	40

Rychlost růstu funkcí

Uvažujme 3 algoritmy se složitostmi $T_1(n) = n$, $T_2(n) = n^3$, $T_3(n) = 2^n$. Náš počítač zvládne v reálném čase (kolik jsme ochotni počkat) 10^{12} kroků.

Složitost	Velikost vstupu
$T_1(n) = n$	10^{12}
$T_2(n) = n^3$	10^4
$T_3(n) = 2^n$	40

Nyní počítač 1000 násobně zrychlíme. Zvládne tedy 10^{15} kroků.

Složitost	Velikost vstupu	Nárust
$T_1(n) = n$	10^{15}	1000×
$T_2(n) = n^3$	10^5	10×
$T_3(n) = 2^n$	50	+10

Asymptotická notace

V následujícím se zaměříme na funkce typu $f : \mathbb{N} \rightarrow \mathbb{R}$, kde:

- Hodnota $f(n)$ nemusí být definovaná pro všechny hodnoty $n \in \mathbb{N}$, ale musí existovat nějaká konstanta n_0 taková, že hodnota $f(n)$ je definovaná pro všechna $n \in \mathbb{N}$ taková, že $n \geq n_0$.

  Příklad: Funkce $f(n) = \log_2(n)$ není definovaná pro $n = 0$, ale pro všechna $n \geq 1$ už definovaná je.

- Musí existovat taková konstanta n_0, že pro všechny hodnoty $n \in \mathbb{N}$, kde $n \geq n_0$, platí $f(n) \geq 0$.

  Příklad: Pro funkci $f(n) = n^2 - 25$ platí $f(n) \geq 0$ pro všechna $n \geq 5$.

Asymptotická notace

Vezměme si libovolnou funkci $g : \mathbb{N} \rightarrow \mathbb{R}$. Zápisy $O(g)$, $\Omega(g)$, $\Theta(g)$, $o(g)$ a $\omega(g)$ označují **množiny funkcí** typu $\mathbb{N} \rightarrow \mathbb{R}$, kde:

- $O(g)$ – množina všech funkcí, které rostou nejvýše tak rychle jako g
- $\Omega(g)$ – množina všech funkcí, které rostou alespoň tak rychle jako g
- $\Theta(g)$ – množina všech funkcí, které rostou stejně rychle jako g
- $o(g)$ – množina všech funkcí, které rostou pomaleji než funkce g
- $\omega(g)$ – množina všech funkcí, které rostou rychleji než funkce g

Poznámka: Toto nejsou definice! Ty následují na následujících slidech.

- O – velké „O“
- Ω – velké řecké písmeno „omega“
- Θ – velké řecké písmeno „theta“
- o – malé „o“
- ω – malé „omega“

Asymptotická notace – symbol O

Neformálně:

$O(g)$ – množina všech funkcí, které rostou nejvýše tak rychle jako g

Jak formálně definovat, kdy platí $f \in O(g)$?

První pokus:

- porovnat hodnoty funkcí

$$(\forall n \in \mathbb{N})(f(n) \leq g(n))$$

Problém: Neumožňuje zanedbat konstanty, např. není pravda, že $(\forall n \in \mathbb{N})(3n^2 \leq 2n^2)$.

Asymptotická notace – symbol O

Neformálně:

$O(g)$ – množina všech funkcí, které rostou nejvýše tak rychle jako g

Jak formálně definovat, kdy platí $f \in O(g)$?

Druhý pokus:

- přenásobit funkci g nějakou dostatečně velkou konstantou c

$$(\exists c > 0)(\forall n \in \mathbb{N})(f(n) \leq c \cdot g(n))$$

Problém: Nerovnost nemusí ani po přenásobení libovolně velkou konstantou platit pro malé hodnoty n.

Například funkce $g(n) = n^2$ očividně roste rychleji než funkce $f(n) = n + 5$. Ovšem bez ohledu na to, jak velkou zvolíme konstantu c, pro $n = 0$ nikdy nebude platit $n + 5 \leq c \cdot n^2$.

Asymptotická notace – symbol O

Neformálně:

$O(g)$ – množina všech funkcí, které rostou nejvýše tak rychle jako g

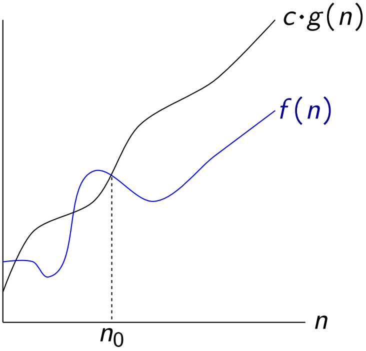
Jak formálně definovat, kdy platí $f \in O(g)$?

Třetí pokus:

- nerovnost nemusí platit pro všechna n, stačí, že bude platit pro všechny „dostatečně velké“ hodnoty n

$$(\exists c > 0)(\exists n_0 \geq 0)(\forall n \geq n_0)(f(n) \leq c \cdot g(n))$$

Asymptotická notace – symbol O



Definice

Vezměme si libovolnou funkci $g : \mathbb{N} \to \mathbb{R}$. Pro funkci $f : \mathbb{N} \to \mathbb{R}$ platí $f \in O(g)$ právě tehdy, když

$$(\exists c > 0)(\exists n_0 \geq 0)(\forall n \geq n_0)\big(f(n) \leq c \cdot g(n)\big).$$

Asymptotická notace – symbol O

Poznámky:

- c je kladné reálné číslo (tj. $c \in \mathbb{R}$ a $c > 0$)
- n_0 a n jsou přirozená čísla (tj. $n_0 \in \mathbb{N}$ a $n \in \mathbb{N}$)

Asymptotická notace – symbol O

Příklad: Vezměme si funkce $f(n) = 2n^2 + 3n + 7$ a $g(n) = n^2$.

Chceme ukázat $f \in O(g)$, tj. $f \in O(n^2)$:

- **Postup 1:**

  Zvolme například $c = 3$.

  $$c \cdot g(n) = 3n^2 = 2n^2 + \tfrac{1}{2}n^2 + \tfrac{1}{2}n^2$$

  Potřebujeme najít takové n_0, aby pro každé $n \geq n_0$ platilo současně

  $$2n^2 \geq 2n^2 \qquad \tfrac{1}{2}n^2 \geq 3n \qquad \tfrac{1}{2}n^2 \geq 7$$

  Snadno ověříme, že například $n_0 = 6$ vyhovuje těmto požadavkům.

  Pak pro každé $n \geq 6$ platí $c \cdot g(n) \geq f(n)$:

  $$cg(n) = 3n^2 = 2n^2 + \tfrac{1}{2}n^2 + \tfrac{1}{2}n^2 \geq 2n^2 + 3n + 7 = f(n)$$

Asymptotická notace – symbol O

Příklad, kde $f(n) = 2n^2 + 3n + 7$ a $g(n) = n^2$:

- **Postup 2:**

  Zvolme $c = 12$.

  $$c \cdot g(n) = 12n^2 = 2n^2 + 3n^2 + 7n^2$$

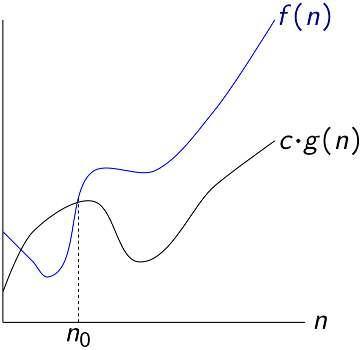
  Potřebujeme najít takové n_0, aby pro každé $n \geq n_0$ platilo současně

  $$2n^2 \geq 2n^2 \qquad 3n^2 \geq 3n \qquad 7n^2 \geq 7$$

  Uvedené vztahy zjevně platí pro $n_0 = 1$, takže pro každé $n \geq 1$ platí $f(n) \leq c \cdot g(n)$:

  $$c \cdot g(n) = 12n^2 = 2n^2 + 3n^2 + 7n^2 \geq 2n^2 + 3n + 7 = f(n)$$

Asymptotická notace – symbol Ω



Definice

Vezměme si libovolnou funkci $g : \mathbb{N} \rightarrow \mathbb{R}$. Pro funkci $f : \mathbb{N} \rightarrow \mathbb{R}$ platí $f \in \Omega(g)$ právě tehdy, když

$$(\exists c > 0)(\exists n_0 \geq 0)(\forall n \geq n_0)\big(c \cdot g(n) \leq f(n)\big).$$

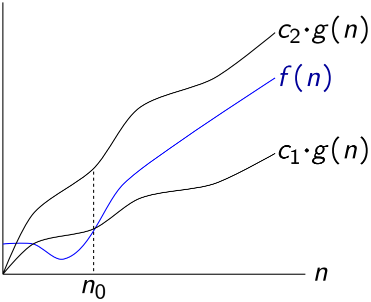
Asymptotická notace – symbol Ω

Není těžké zdůvodnit, že platí následující tvrzení:

Pro libovolné funkce f a g platí:

$$f \in O(g) \qquad \text{právě tehdy, když} \qquad g \in \Omega(f)$$

Asymptotická notace – symbol Θ



Definice

Vezměme si libovolnou funkci $g : \mathbb{N} \to \mathbb{R}$. Pro funkci $f : \mathbb{N} \to \mathbb{R}$ platí $f \in \Theta(g)$ právě tehdy, když

$$(\exists c_1 > 0)(\exists c_2 > 0)(\exists n_0 \geq 0)(\forall n \geq n_0)\big(c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\big).$$

Asymptotická notace – symbol Θ

Z definice Θ snadno vyplývá následující:

Pro libovolné funkce f a g platí:

$f \in \Theta(g)$ právě tehdy, když $f \in O(g)$ a $f \in \Omega(g)$

$f \in \Theta(g)$ právě tehdy, když $f \in O(g)$ a $g \in O(f)$

$f \in \Theta(g)$ právě tehdy, když $g \in \Theta(f)$

Asymptotická notace – symboly o and ω

Definice

Vezměme si libovolnou funkci $g : \mathbb{N} \to \mathbb{R}$. Pro funkci $f : \mathbb{N} \to \mathbb{R}$ platí $f \in o(g)$ právě tehdy, když

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = 0$$

Definice

Vezměme si libovolnou funkci $g : \mathbb{N} \to \mathbb{R}$. Pro funkci $f : \mathbb{N} \to \mathbb{R}$ platí $f \in \omega(g)$ právě tehdy, když

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = +\infty$$

Asymptotická notace

Pro libovolné funkce f a g platí následující tvrzení:

Jestliže existuje hodnota $c \geq 0$ taková, že

$$\lim_{n \to +\infty} \frac{f(n)}{g(n)} = c$$

pak $f \in O(g)$.

Jestliže existuje hodnota $c \geq 0$ taková, že

$$\lim_{n \to +\infty} \frac{g(n)}{f(n)} = c$$

pak $f \in \Omega(g)$.

Asymptotická notace

Zjevně platí:

- Pokud $f \in o(g)$, pak $f \in O(g)$.
- Pokud $f \in \omega(g)$, pak $f \in \Omega(g)$.

Asymptotická notace

Na asymptotickou notaci se můžeme dívat jako na určitý druh porovnání **rychlosti růstu** funkcí:

$f \in O(g)$ — rychlost růstu f "$\leq$" rychlost růstu g

$f \in \Omega(g)$ — rychlost růstu f "$\geq$" rychlost růstu g

$f \in \Theta(g)$ — rychlost růstu f "$=$" rychlost růstu g

$f \in o(g)$ — rychlost růstu f "$<$" rychlost růstu g

$f \in \omega(g)$ — rychlost růstu f "$>$" rychlost růstu g

Poznámka:

- Existují dvojice funkcí f a g takové, že

  $$f \notin O(g) \qquad \text{a} \qquad g \notin O(f),$$

  například

  $$f(n) = n^2 \qquad\qquad g(n) = \begin{cases} n & \text{pokud } n \mod 2 = 1 \\ n^3 & \text{jinak} \end{cases}$$

Asymptotická notace

- O funkci f řekneme, že je:

  lineární, pokud $f(n) \in \Theta(n)$
  kvadratická, pokud $f(n) \in \Theta(n^2)$
  kubická, pokud $f(n) \in \Theta(n^3)$
  polynomiální, pokud $f(n) \in O(n^k)$ pro nějaké $k > 0$
  exponenciální, pokud $f(n) \in O(c^{n^k})$ pro nějaké $c > 1$ a $k > 0$
  logaritmická, pokud $f(n) \in \Theta(\log n)$
  polylogaritmická, pokud $f(n) \in \Theta(\log^k n)$ pro nějaké $k > 0$

- $O(1)$ je množina všech **omezených** funkcí, tj. funkcí jejichž funkční hodnoty jsou shora omezeny nějakou konstantou.

- Exponenciální funkce se v asymptotické notaci často uvádí ve tvaru $2^{O(n^k)}$, protože potom již nemusíme uvažovat různé základy mocniny.

Asymptotická notace

Obecně platí:

- jakákoliv polylogaritmická funkce roste pomaleji než jakákoli polynomiální funkce
- jakákoli polynomiální funkce roste pomaleji než jakákoli exponenciální funkce
- při porovnávání polynomiálních funkcí n^k a n^ℓ stačí porovnat hodnoty k a ℓ
- při porovnávání polylogaritmických funkcí $\log^k n$ a $\log^\ell n$ stačí porovnat hodnoty k a ℓ
- při porovnávání exponenciálních funkcí $2^{p(n)}$ a $2^{q(n)}$ stačí porovnat polynomy $p(n)$ a $q(n)$.

Asymptotická notace

Tvrzení

Předpokládejme, že a a b jsou nějaké konstanty takové, že $a > 0$ a $b > 0$, a k a ℓ jsou nějaké libovolné konstanty, kde $k \geq 0$, $\ell \geq 0$ a $k \leq \ell$.

Uvažujme funkce

$$f(n) = a \cdot n^k \qquad\qquad g(n) = b \cdot n^\ell$$

Pro každé takové funkce f a g platí $f \in O(g)$:

Důkaz: Zvolme $c = \frac{a}{b}$.

Vzhledem k tomu, že pro $n \geq 1$ zjevně platí $n^k \leq n^\ell$ (protože $k \leq \ell$), tak pro $n \geq 1$ platí

$$c \cdot g(n) = \frac{a}{b} \cdot g(n) = \frac{a}{b} \cdot b \cdot n^\ell = a \cdot n^\ell \geq a \cdot n^k = f(n)$$

Asymptotická notace

Tvrzení

Pro libovolná $a, b > 1$ a libovolné $n > 0$ platí

$$\log_a n = \frac{\log_b n}{\log_b a}$$

Důkaz: Z $n = a^{\log_a n}$ plyne $\log_b n = \log_b(a^{\log_a n})$.
Protože $\log_b(a^{\log_a n}) = \log_a n \cdot \log_b a$, dostáváme $\log_b n = \log_a n \cdot \log_b a$, z čehož plyne výše uvedený závěr. $\square$

Z toho důvodu se při použití asymptotické notace základ logaritmu obvykle vynechává: například místo $\Theta(n \log_2 n)$ můžeme napsat $\Theta(n \log n)$.

Asymptotická notace

Příklady:

$n \in O(n^2)$
$1000n \in O(n)$
$2^{\log_2 n} \in \Theta(n)$
$n^3 \notin O(n^2)$
$n^2 \notin O(n)$
$n^3 + 2^n \notin O(n^2)$

$n^3 \in O(n^4)$
$0.00001n^2 - 10^{10}n \in \Theta(10^{10}n^2)$
$n^3 - n^2 \log_2^3 n + 1000n - 10^{100} \in \Theta(n^3)$
$n^3 + 1000n - 10^{100} \in O(n^3)$
$n^3 + n^2 \notin \Theta(n^2)$
$n! \notin O(2^n)$

Asymptotická notace

Pro libovolné tři funkce f, g a h platí:

- jestliže $f \in O(g)$ a $g \in O(h)$, pak $f \in O(h)$
- jestliže $f \in \Omega(g)$ a $g \in \Omega(h)$, pak $f \in \Omega(h)$
- jestliže $f \in \Theta(g)$ a $g \in \Theta(h)$, pak $f \in \Theta(h)$

Asymptotická notace

- Pro libovolnou funkci f a libovolnou konstantu $c > 0$ platí:
  - $c \cdot f \in \Theta(f)$

- Pro libovolné dvě funkce f, g platí:
  - $\max(f, g) \in \Theta(f + g)$
  - pokud $f \in O(g)$, pak $f + g \in \Theta(g)$

- Pro libovolné čtyři funkce f_1, f_2, g_1, g_2 platí:
  - pokud $f_1 \in O(f_2)$ a $g_1 \in O(g_2)$, pak $f_1 + g_1 \in O(f_2 + g_2)$ a $f_1 \cdot g_1 \in O(f_2 \cdot g_2)$
  - pokud $f_1 \in \Theta(f_2)$ a $g_1 \in \Theta(g_2)$, pak $f_1 + g_1 \in \Theta(f_2 + g_2)$ a $f_1 \cdot g_1 \in \Theta(f_2 \cdot g_2)$

Asymptotická notace

Jak bylo uvedeno, výrazy $O(g)$, $\Omega(g)$, $\Theta(g)$, $o(g)$ a $\omega(g)$ označují určité množiny funkcí.

V odborných textech se však někdy používají tyto výrazy i v poněkud odlišném významu:

- zápis $O(g)$, $\Omega(g)$, $\Theta(g)$, $o(g)$ nebo $\omega(g)$ nereprezentuje danou množinu funkcí, ale **nějakou** funkci z dané množiny.

Tato konvence se používá zejména v zápisu rovnic nebo nerovnic.

Příklad: $3n^3 + 5n^2 - 11n + 2 = 3n^3 + O(n^2)$

Při použití této konvence je tedy možné například psát $f = O(g)$ místo $f \in O(g)$.

Složitost algoritmů

Řekněme, že bychom chtěli analyzovat časovou složitost $T(n)$ nějakého algoritmu, který se skládá z instrukcí $I_1, I_2, \ldots, I_k$:

- Předpokládejme, že doby provedení jednotlivých instrukcí jsou $c_1, c_2, \ldots, c_k$, tj. doba provedení instrukce I_i je dána konstantou c_i.
- Předpokládejme, že In je množina všech možných vstupů pro daný algoritmus.

  Zaveďme si pro každou instrukci I_i odpovídající funkci

  $$m_i : In \rightarrow \mathbb{N}$$

  udávající, kolikrát se provede instrukce I_i při výpočtu nad daným vstupem, tj. hodnota $m_i(w)$ udává, kolikrát se provede instrukce I_i při výpočtu nad vstupem w.

Složitost algoritmů

- Celková doba výpočtu nad vstupem w:

$$t(w) = c_1 \cdot m_1(w) + c_2 \cdot m_2(w) + \cdots + c_k \cdot m_k(w).$$

- Připomeňme, že $T(n) = \max\{\, t(w) \mid size(w) = n \,\}$.

- Pro každou z funkcí $m_1, m_2, \ldots, m_k$ můžeme nadefinovat odpovídající funkci $f_i : \mathbb{N} \to \mathbb{R}$, kde

$$f_i(n) = \max\{\, m_i(w) \mid size(w) = n \,\}$$

  tj. $f_i(n)$ je maximum z počtu provedení instrukce I_i pro všechny vstupy velikosti n.

- Zjevně platí $T \in O(f_1 + f_2 + \cdots + f_k)$.

- Připomeňme si, že pokud $f_j \in O(f_i)$, pak $c_i \cdot f_i + c_j \cdot f_j \in O(f_i)$.

- Pokud tedy pro některou funkci f_i platí, že pro všechny f_j, kde $j \neq i$, je $f_j \in O(f_i)$, pak

$$T \in O(f_i).$$

Složitost algoritmů

- Zjevně také platí, že pro libovolnou z funkcí $f_1, f_2, \ldots, f_k$ je $T \in \Omega(f_i)$.
- Při analýze celkové časové složitosti $T(n)$ se tedy většinou můžeme omezit pouze na analýzu počtu provedení nejčastěji prováděné instrukce I_i, tj. zkoumání toho, jak rychle roste funkce $f_i(n)$, protože platí
  $$T \in \Theta(f_i).$$
- Pro ostatní instrukce I_j stačí ověřit, že
  $$f_j \in O(f_i),$$
  tj. není pro ně nutné přesně zjišťovat, jak rychle rostou, ale jen to, že rostou nanejvýš tak rychle jako f_i.

Složitost algoritmů

Příklad:

Algoritmus: Nalezení největšího prvku v poli

```
FIND-MAX ($A, n$):
    $k := 0$
    for $i := 1$ to $n - 1$ do
        if $A[i] > A[k]$ then
            $k := i$
    return $A[k]$
```

Složitost algoritmů

Při analýze složitosti algoritmu Find-Max jsme zjistili, že časová složitost daného algoritmu v nejhorším případě je

$$f(n) = an + b\,.$$

Kdybychom to nechtěli takto podrobně zjišťovat a spokojili se s hrubším odhadem, mohli jsme určit, že časová složitost tohoto algoritmu je $\Theta(n)$, protože:

- Algoritmus obsahuje jediný cyklus, který se pro vstup velikosti n provede vždy právě $(n-1)$ krát, tj. počet průchodů cyklem je v $\Theta(n)$.
- V rámci jednoho průchodu cyklem se provede několik instrukcí, jejichž počet je shora i zdola omezen nějakými konstantami nezávislými na velikosti vstupu. Doba provedení jedné iterace cyklu je tedy v $\Theta(1)$.
- Ostatní instrukce se provedou jednou. Čas, který se stráví jejich prováděním, je v $\Theta(1)$.

Složitost algoritmů

Pokusme se analyzovat časovou složitost následujícího algoritmu:

Algoritmus: Třídění přímým vkládáním

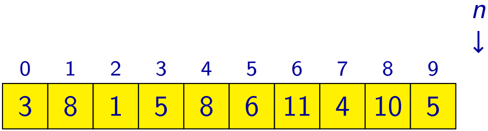
```
INSERTION-SORT (A, n):
    for $j := 1$ to $n - 1$ do
        $x := A[j]$
        $i := j - 1$
        while $i \geq 0$ and $A[i] > x$ do
            $A[i + 1] := A[i]$
            $i := i - 1$
        $A[i + 1] := x$
```

Tj. chceme najít funkci $T(n)$ takovou, že časová složitost algoritmu INSERTION-SORT v nejhorším případě je v $\Theta(T(n))$.

Složitost algoritmů

Příklad: Výpočet algoritmu Insertion-Sort pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$



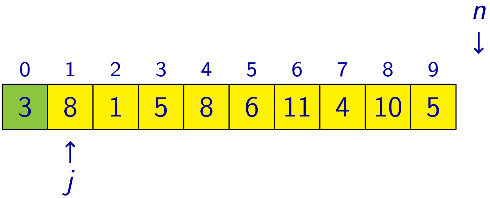
0	1	2	3	4	5	6	7	8	9
3	8	1	5	8	6	11	4	10	5

$x = ?$

Složitost algoritmů

Příklad: Výpočet algoritmu INSERTION-SORT pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$



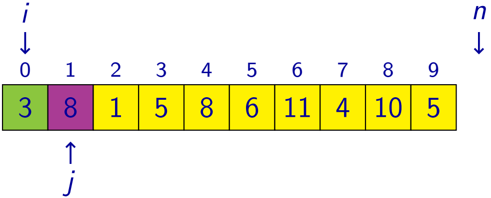
0	1	2	3	4	5	6	7	8	9
3	8	1	5	8	6	11	4	10	5

$x = ?$

Složitost algoritmů

Příklad: Výpočet algoritmu INSERTION-SORT pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$

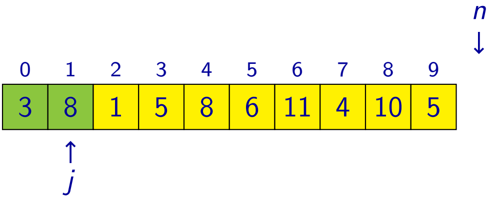


$x = 8$

Složitost algoritmů

Příklad: Výpočet algoritmu INSERTION-SORT pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$



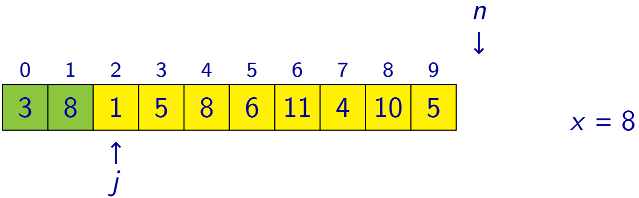
0	1	2	3	4	5	6	7	8	9
🟩 3	🟩 8	🟨 1	🟨 5	🟨 8	🟨 6	🟨 11	🟨 4	🟨 10	🟨 5

$x = 8$

Složitost algoritmů

Příklad: Výpočet algoritmu INSERTION-SORT pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$



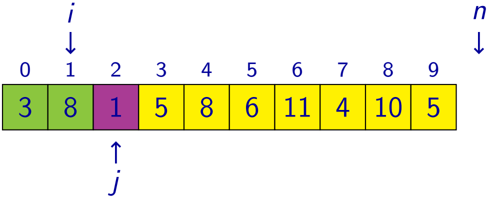
0	1	2	3	4	5	6	7	8	9
3	8	1	5	8	6	11	4	10	5

$x = 8$

Složitost algoritmů

Příklad: Výpočet algoritmu INSERTION-SORT pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$



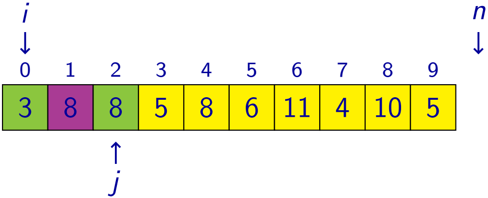
0	1	2	3	4	5	6	7	8	9
3	8	1	5	8	6	11	4	10	5

$x = 1$

Složitost algoritmů

Příklad: Výpočet algoritmu INSERTION-SORT pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$



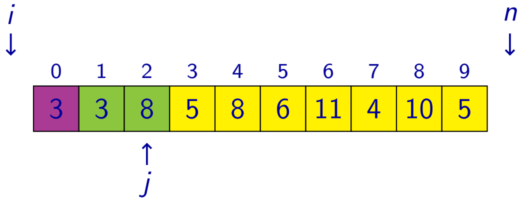
0	1	2	3	4	5	6	7	8	9
3	8	8	5	8	6	11	4	10	5

$x = 1$

Složitost algoritmů

Příklad: Výpočet algoritmu INSERTION-SORT pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$

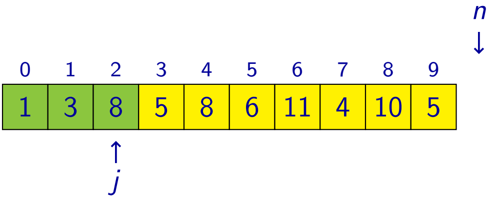


$x = 1$

Složitost algoritmů

Příklad: Výpočet algoritmu Insertion-Sort pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$



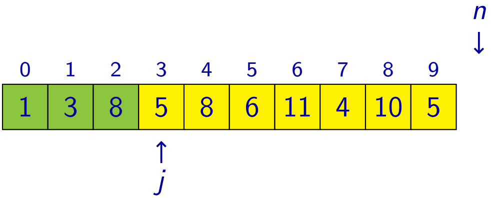
0	1	2	3	4	5	6	7	8	9
1	3	8	5	8	6	11	4	10	5

$x = 1$

Složitost algoritmů

Příklad: Výpočet algoritmu INSERTION-SORT pro vstup

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



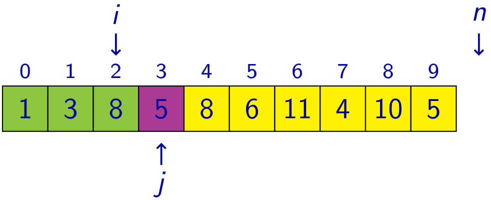
0	1	2	3	4	5	6	7	8	9
1	3	8	5	8	6	11	4	10	5

$x = 1$

Složitost algoritmů

Příklad: Výpočet algoritmu INSERTION-SORT pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$



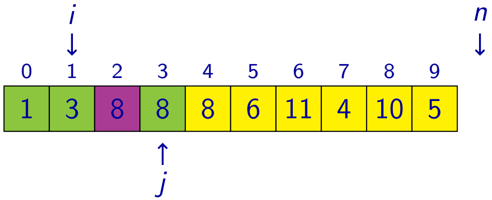
0	1	2	3	4	5	6	7	8	9
1	3	8	5	8	6	11	4	10	5

$x = 5$

Složitost algoritmů

Příklad: Výpočet algoritmu Insertion-Sort pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$



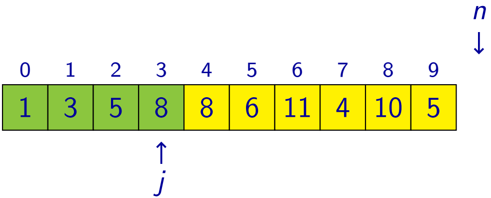
0	1	2	3	4	5	6	7	8	9
1	3	8	8	8	6	11	4	10	5

$x = 5$

Složitost algoritmů

Příklad: Výpočet algoritmu Insertion-Sort pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\; n = 10.$$



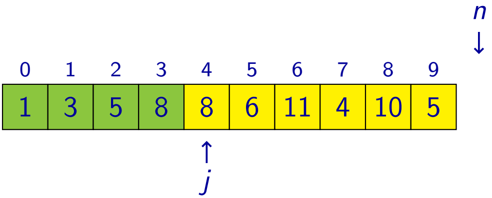
0	1	2	3	4	5	6	7	8	9
1	3	5	8	8	6	11	4	10	5

$x = 5$

Složitost algoritmů

Příklad: Výpočet algoritmu Insertion-Sort pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$



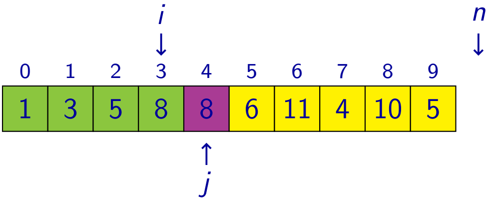
0	1	2	3	4	5	6	7	8	9
1	3	5	8	8	6	11	4	10	5

$x = 5$

Složitost algoritmů

Příklad: Výpočet algoritmu Insertion-Sort pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$



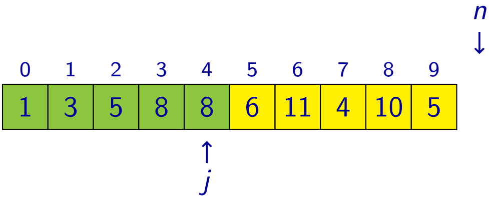
0	1	2	3	4	5	6	7	8	9
1	3	5	8	8	6	11	4	10	5

$x = 8$

Složitost algoritmů

Příklad: Výpočet algoritmu INSERTION-SORT pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$



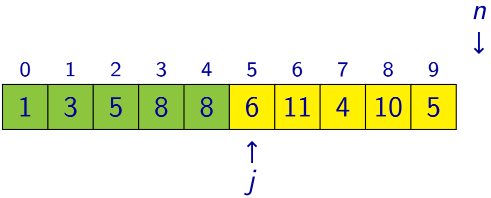
0	1	2	3	4	5	6	7	8	9
1	3	5	8	8	6	11	4	10	5

$x = 8$

Složitost algoritmů

Příklad: Výpočet algoritmu Insertion-Sort pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$



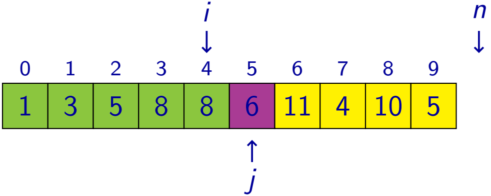
0	1	2	3	4	5	6	7	8	9
1	3	5	8	8	6	11	4	10	5

$x = 8$

Složitost algoritmů

Příklad: Výpočet algoritmu INSERTION-SORT pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$

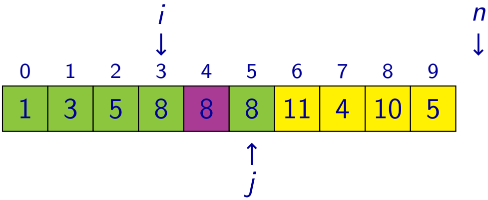


$x = 6$

Složitost algoritmů

Příklad: Výpočet algoritmu INSERTION-SORT pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$



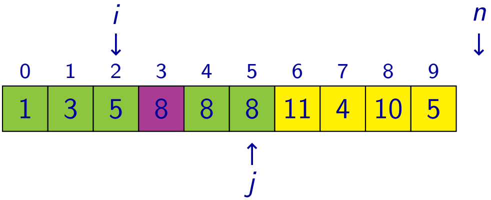
0	1	2	3	4	5	6	7	8	9
1	3	5	8	8	8	11	4	10	5

$x = 6$

Složitost algoritmů

Příklad: Výpočet algoritmu Insertion-Sort pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$



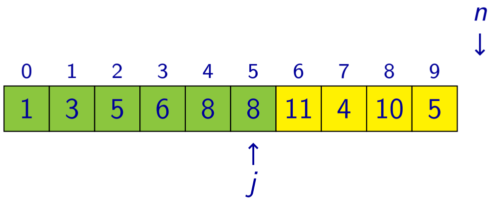
0	1	2	3	4	5	6	7	8	9
1	3	5	8	8	8	11	4	10	5

$x = 6$

Složitost algoritmů

Příklad: Výpočet algoritmu INSERTION-SORT pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$



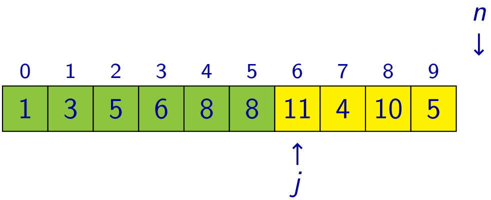
0	1	2	3	4	5	6	7	8	9
1	3	5	6	8	8	11	4	10	5

$x = 6$

Složitost algoritmů

Příklad: Výpočet algoritmu Insertion-Sort pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$



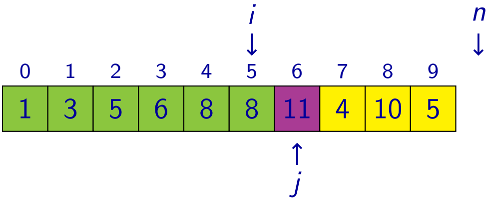
0	1	2	3	4	5	6	7	8	9
1	3	5	6	8	8	11	4	10	5

$x = 6$

Složitost algoritmů

Příklad: Výpočet algoritmu INSERTION-SORT pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$



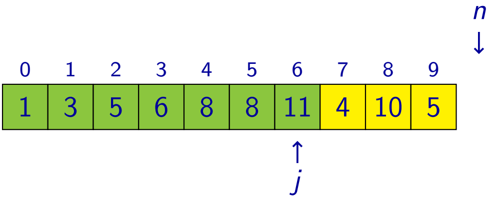
0	1	2	3	4	5	6	7	8	9
1	3	5	6	8	8	11	4	10	5

$x = 11$

Složitost algoritmů

Příklad: Výpočet algoritmu INSERTION-SORT pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$



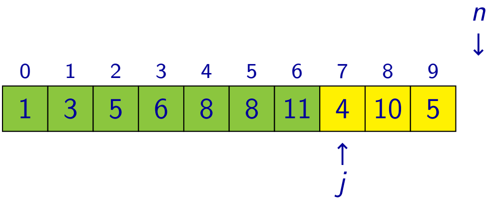
0	1	2	3	4	5	6	7	8	9
1	3	5	6	8	8	11	4	10	5

$x = 11$

Složitost algoritmů

Příklad: Výpočet algoritmu Insertion-Sort pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$



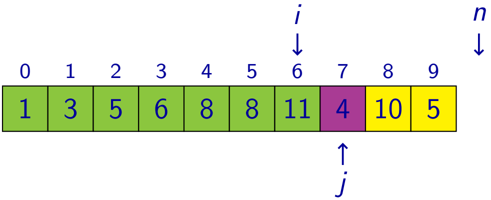
0	1	2	3	4	5	6	7	8	9
1	3	5	6	8	8	11	4	10	5

$x = 11$

Složitost algoritmů

Příklad: Výpočet algoritmu INSERTION-SORT pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$



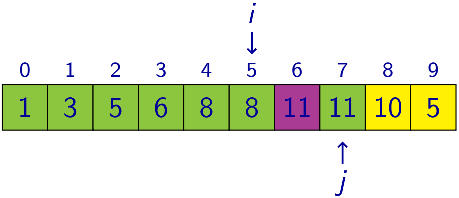
0	1	2	3	4	5	6	7	8	9
1	3	5	6	8	8	11	4	10	5

$x = 4$

Složitost algoritmů

Příklad: Výpočet algoritmu INSERTION-SORT pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$



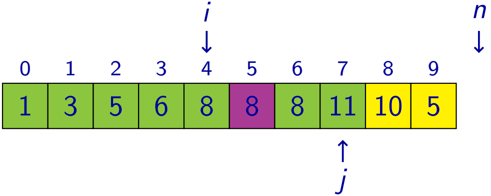
0	1	2	3	4	5	6	7	8	9
1	3	5	6	8	8	11	11	10	5

$x = 4$

Složitost algoritmů

Příklad: Výpočet algoritmu Insertion-Sort pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$

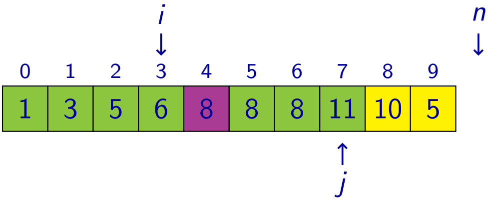


$x = 4$

Složitost algoritmů

Příklad: Výpočet algoritmu INSERTION-SORT pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$



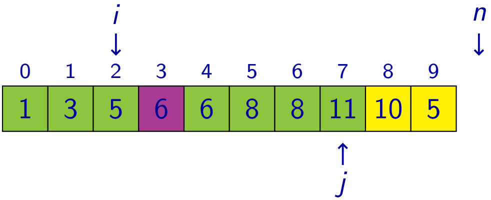
0	1	2	3	4	5	6	7	8	9
1	3	5	6	8	8	8	11	10	5

$x = 4$

Složitost algoritmů

Příklad: Výpočet algoritmu INSERTION-SORT pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$



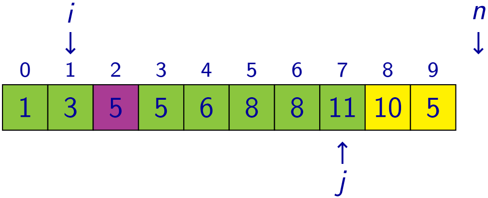
0	1	2	3	4	5	6	7	8	9
1	3	5	6	6	8	8	11	10	5

$x = 4$

Složitost algoritmů

Příklad: Výpočet algoritmu INSERTION-SORT pro vstup

$$A = [\, 3, 8, 1, 5, 8, 6, 11, 4, 10, 5 \,],\ n = 10.$$



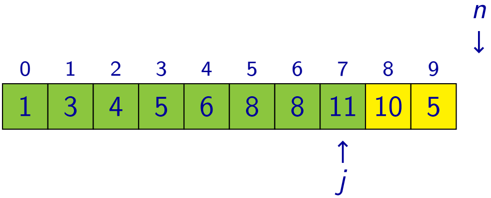
0	1	2	3	4	5	6	7	8	9
1	3	5	5	6	8	8	11	10	5

$x = 4$

Složitost algoritmů

Příklad: Výpočet algoritmu Insertion-Sort pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$



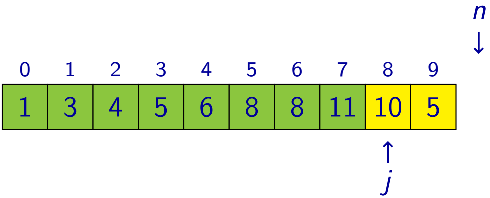
0	1	2	3	4	5	6	7	8	9
1	3	4	5	6	8	8	11	10	5

$x = 4$

Složitost algoritmů

Příklad: Výpočet algoritmu Insertion-Sort pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$



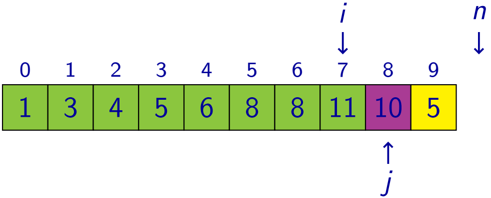
0	1	2	3	4	5	6	7	8	9
1	3	4	5	6	8	8	11	10	5

$x = 4$

Složitost algoritmů

Příklad: Výpočet algoritmu INSERTION-SORT pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$

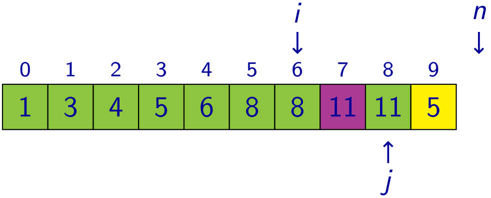


$x = 10$

Složitost algoritmů

Příklad: Výpočet algoritmu INSERTION-SORT pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$



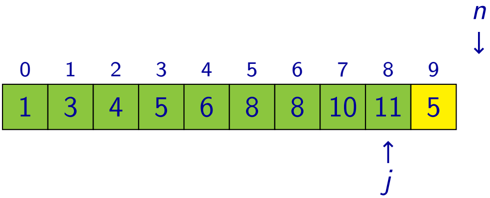
0	1	2	3	4	5	6	7	8	9
1	3	4	5	6	8	8	11	11	5

$x = 10$

Složitost algoritmů

Příklad: Výpočet algoritmu Insertion-Sort pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$



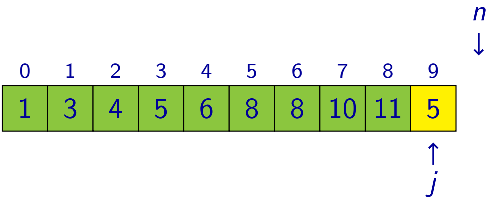
0	1	2	3	4	5	6	7	8	9
1	3	4	5	6	8	8	10	11	5

$x = 10$

Složitost algoritmů

Příklad: Výpočet algoritmu INSERTION-SORT pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$



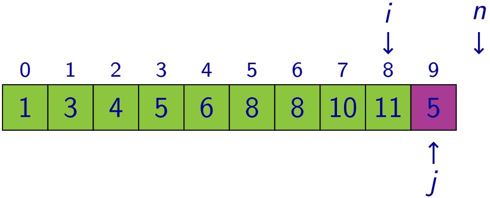
0	1	2	3	4	5	6	7	8	9
1	3	4	5	6	8	8	10	11	5

$x = 10$

Složitost algoritmů

Příklad: Výpočet algoritmu INSERTION-SORT pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$



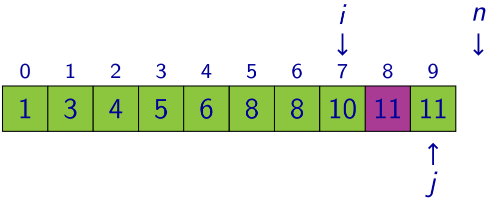
0	1	2	3	4	5	6	7	8	9
1	3	4	5	6	8	8	10	11	5

$x = 5$

Složitost algoritmů

Příklad: Výpočet algoritmu INSERTION-SORT pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$



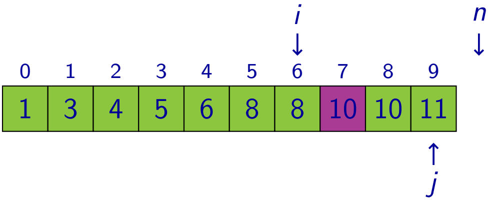
0	1	2	3	4	5	6	7	8	9
1	3	4	5	6	8	8	10	11	11

$x = 5$

Složitost algoritmů

Příklad: Výpočet algoritmu INSERTION-SORT pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$



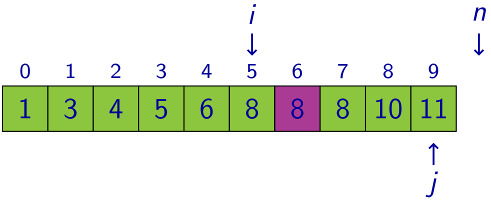
0	1	2	3	4	5	6	7	8	9
1	3	4	5	6	8	8	10	10	11

$x = 5$

Složitost algoritmů

Příklad: Výpočet algoritmu Insertion-Sort pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$



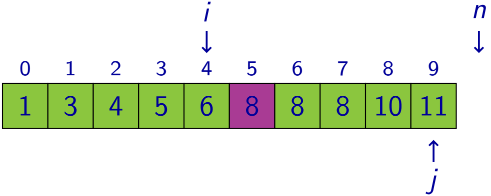
0	1	2	3	4	5	6	7	8	9
1	3	4	5	6	8	8	8	10	11

$x = 5$

Složitost algoritmů

Příklad: Výpočet algoritmu Insertion-Sort pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$



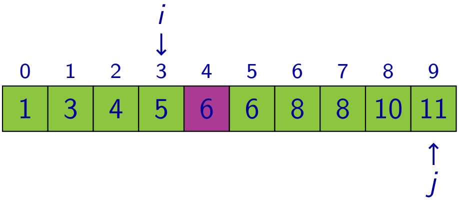
0	1	2	3	4	5	6	7	8	9
1	3	4	5	6	8	8	8	10	11

$x = 5$

Složitost algoritmů

Příklad: Výpočet algoritmu INSERTION-SORT pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$



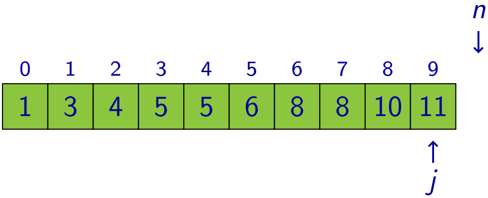
0	1	2	3	4	5	6	7	8	9
1	3	4	5	6	6	8	8	10	11

$x = 5$

Složitost algoritmů

Příklad: Výpočet algoritmu Insertion-Sort pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$



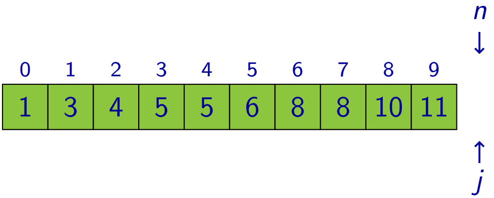
0	1	2	3	4	5	6	7	8	9
1	3	4	5	5	6	8	8	10	11

$x = 5$

Složitost algoritmů

Příklad: Výpočet algoritmu Insertion-Sort pro vstup

$$A = [\,3, 8, 1, 5, 8, 6, 11, 4, 10, 5\,],\ n = 10.$$



0	1	2	3	4	5	6	7	8	9
1	3	4	5	5	6	8	8	10	11

$x = 5$

Složitost algoritmů

Algoritmus: Třídění přímým vkládáním

```
INSERTION-SORT ($A$, $n$):
    for $j := 1$ to $n - 1$ do
        $x := A[j]$
        $i := j - 1$
        while $i \geq 0$ and $A[i] > x$ do
            $A[i + 1] := A[i]$
            $i := i - 1$
        $A[i + 1] := x$
```

Složitost algoritmů

Uvažujme vstupy velikosti n:

- Vnější cyklus **for** se provede $n - 1$ krát.
  (Proměnná j nabývá hodnot $1, 2, \ldots, n - 1$.)

- Vnitřní cyklus **while** se pro danou hodnotu j provede maximálně j krát.
  (Proměnná i nabývá hodnot $j - 1, j - 2, \ldots, 1, 0$.)

- Existují vstupy, pro které platí, že pro každou hodnotu j od 1 do $n - 1$ se vnitřní cyklus **while** provede právě j krát.

- V nejhorším případě se tedy cyklus **while** provede celkem m krát, kde

$$m = 1 + 2 + \cdots + (n - 1) = (1 + (n - 1)) \cdot \tfrac{n-1}{2} = \tfrac{1}{2}n^2 - \tfrac{1}{2}n$$

- Celková časová složitost algoritmu $\textsc{Insertion-Sort}$ v nejhorším případě je tedy $\Theta(n^2)$.

Složitost algoritmů

V předchozím případě jsme přesně spočítali celkový počet průchodů cyklem **while**.

Obecně to není vždy možné spočítat takto přesně nebo to může být hodně komplikované. Pokud nás zajímá jen asymptotický odhad, tak to často ani není nutné.

Složitost algoritmů

Pokud bychom například neuměli spočítat součet aritmetické posloupnosti, mohli bychom provést analýzu následovně:

- Vnější cyklus **for** se neprovede více než n krát, vnitřní cyklus **while** se při každé iteraci vnějšího cyklu provede maximálně n krát. Celkově se tedy vnitřní cyklus provede maximálně n^2 krát.

  Platí tedy $T \in O(n^2)$.

- Pro některé vstupy se při posledních $\lfloor n/2 \rfloor$ průchodech cyklem **for** provede cyklus **while** alespoň $\lceil n/2 \rceil$ krát.

  Pro některé vstupy se tedy cyklus **while** provede alespoň $\lfloor n/2 \rfloor \cdot \lceil n/2 \rceil$ krát.

  $\lfloor n/2 \rfloor \cdot \lceil n/2 \rceil \geq (n/2 - 1) \cdot (n/2) = \frac{1}{4}n^2 - \frac{1}{2}n$

  Platí tedy $T \in \Omega(n^2)$.

Složitost algoritmů

- Zatím jsme uvažovali, že provedení dané instrukce trvá vždy stejně dlouho bez ohledu na to, s jakými hodnotami pracuje.
- Při použití asymptotických odhadů tedy doba trvání jednotlivých instrukcí nehrála roli a důležité bylo pouze to, kolikrát se daná instrukce při běhu algoritmu provede.
- Například při použití strojů RAM jako výpočetního modelu to odpovídá počítání počtu provedených instrukcí, tj. doba trvání provedení jedné instrukce je 1.

  Tato se označuje jako použití tzv. **jednotkové míry**.
- Odhady časové složitosti v jednotkové míře odpovídají době běhu na skutečných počítačích za předpokladu, že operace, které provádí stroj RAM, může skutečný počítač provést v konstantním čase.

  To platí, pokud čísla, se kterými algoritmus pracuje, jsou malá (vejdou se např. do 32 nebo 64 bitů).

Složitost algoritmů

- Pokud by stroj RAM pracoval s „velkými“ čísly (např. 1000 bitovými), bude odhad časové složitosti v jednotkové míře nerealistický v tom smyslu, že výpočet na skutečném počítači bude trvat mnohem déle.
- Proto se při analýze časové složitosti algoritmů, u kterých se předpokládá práce s velkými čísly, používá tzv. **logaritmická míra**, kdy je doba provedení jedné instrukce úměrná počtu **bitových operací**, které je třeba pro provedení dané instrukce provést.
- Doba trvání instrukce je tedy závislá na aktuálních hodnotách jejích operandů.
- Například doba provádění instrukcí sčítání a odčítání je rovna součtu počtů bitů jejich operandů.
- Doba provádění instrukcí násobení a dělení je rovna součinu počtů bitů jejich operandů.

Složitost algoritmů

Poznámka: Zápisem $\mathit{blen}(x)$ označme počet bitů v binárním zápise přirozeného čísla x.

Platí

$$\mathit{blen}(x) = \max\left(1, \lceil \log_2(x+1) \rceil\right)$$

Prostorová (paměťová) složitost algoritmů

- Zatím jsme se zajímali o čas, který potřebujeme k výpočtu
- Někdy bývá kritickou velikost paměti potřebné k provedení výpočtu.

V případě strojů RAM opět můžeme i z hlediska množství použité paměti rozlišovat mezi použitím jednotkové a logaritmické míry:

Množstvím paměti stroje RAM $\mathcal{M}$ použitým pro vstup w rozumíme buď počet buněk paměti nebo počet bitů paměti, které stroj $\mathcal{M}$ během svého výpočtu nad vstupem w použije.

Definice

Prostorová složitost stroje RAM $\mathcal{M}$ (v nejhorším případě) je funkce $S : \mathbb{N} \to \mathbb{N}$, kde $S(n)$ udává maximální množství paměti použité strojem $\mathcal{M}$ pro vstupy délky n.

Prostorová (paměťová) složitost algoritmů

- Pro konkrétní problém můžeme mít dva algoritmy takové, že jeden má menší prostorovou složitost a druhý zase časovou složitost.
- Je-li časová složitost algoritmu v $O(f(n))$ je i prostorová v $O(f(n))$ (počet buněk navštívených RAMem nemůže být řádově větší než počet kroků, protože v každém kroku použije nejvýše tři buňky paměti — nejvýše dvě pro čtení a nejvýše jednu pro zápis).