Legal Tech: I

Jan Černý, Tomáš Knap

crimczech

Charles University

Version 1.0, 2026

 CC BY-NC-ND 4.0

Anonymní feedback



Naskenujte telefonem nebo klikněte: dvě minuty, bez jména

CC BY-NC-ND 4.0

Funkce

Pojmenovaný a parametrizovaný podprogram

CC BY-NC-ND 4.0

Funkce: syntaxe

Funkce

Pojmenovaný parametrizovaný podprogram.

```
def je_zletily(vek):                # def, jméno, parametry v závorkách, dvojtečka
    return vek >= 18                # return vrátí hodnotu volajícímu a funkci ukončí

print(je_zletily(17))               # volání: jméno a argumenty v závorkách
print(je_zletily(42))
```

```
False
True
```

- `def` funkci **definuje**, nic se zatím neprovede; tělo proběhne až při **volání** `je_zletily(17)`
- **Parametr** `vek` je proměnná, do které se při volání uloží předaná hodnota (**argument**)
- Jméno je sloveso nebo otázka (`spocitej_dph()`, `je_zletily()`), malými písmeny s podtržítky

None: hodnota „nic“

None

Hodnota „nic“: jediná svého typu `NoneType`; není to `0`, `""` ani `False`.

```
>>> vysledek = print("Ahoj")              # print() vypíše, ale nic nevrací
Ahoj
>>> print(vysledek)
None
>>> vysledek is None                      # na None se ptáme přes is, ne ==
True
```

- `None` vrací funkce, která nemá co vrátit: `print()`, `.append()`, `.sort()`; proto `x = lhuty.sort()` uloží `None`
- Ptáme se `if hodnota is None:`; REPL `None` nevypisuje, `print(None)` ano

return a print nejsou totéž

```
cena = 1000

def dph(cena):
    return cena * 0.21              # vrátí hodnotu, se kterou lze dál počítat

def vypis_dph(cena):
    print(cena * 0.21)              # jen vypíše; vrací None

s_dph = cena + dph(cena)            # 1210.0
s_dph = cena + vypis_dph(cena)      # vypíše 210.0 a pak TypeError: int + None
```

- `return` **předá výsledek** zpět tam, odkud byla funkce zavolána; `print()` ho jen ukáže člověku
- Funkce bez `return` (nebo s `return` bez hodnoty) vrací `None`
- `return` funkci ihned ukončí: co je za ním, se už neprovede; ve větvích `if` může být víc `return`

CC BY-NC-ND 4.0

Typové anotace ve funkcích

```python
def vypocti_urok(jistina: int, sazba: float) -> float:
    """Úrok v Kč za jeden rok. Sazba je desetinné číslo, 0.05 znamená 5 %."""
    return jistina * sazba

def prvni(smlouvy: list[str]) -> str | None:  # vrací text, nebo None pro prázdný seznam
    if len(smlouvy) == 0:
        return None
    return smlouvy[0]
```

- Za parametrem dvojtečka a typ, za závorkou `->` a typ výsledku; `-> None` u funkcí, které nic nevracejí
- Python anotace **nekontroluje**, jako u proměnných; editor podle nich napovídá a čtenář vidí smlouvu funkce bez čtení těla
- Text v uvozovkách hned pod `def` je **docstring**, popis funkce pro lidi; konvence kurzu: každá funkce má anotace i docstring

CC BY-NC-ND 4.0

Pojmenované argumenty a výchozí hodnoty

```
def vypocti_urok(jistina: int, sazba: float = 0.05, roky: int = 1) -> float:
    return jistina * sazba * roky
```

```
>>> vypocti_urok(100_000)                    # sazba i roky z výchozích hodnot
5000.0
>>> vypocti_urok(100_000, 0.08)              # poziční: podle pořadí
8000.0
>>> vypocti_urok(100_000, roky=3)            # pojmenovaný: podle jména
15000.0
```

- **Výchozí hodnota** `sazba: float = 0.05` platí, když argument nezadáme; takové parametry jsou až na konci
- **Pojmenovaný argument** `roky=3` říká, kam hodnota patří; na pořadí pak nezáleží
- Poziční argumenty patří před pojmenované; bez povinné `jistina` skončí volání `TypeError`

CC BY-NC-ND 4.0

args: libovolný počet argumentů

```
def soucet_lhut(*lhuty: int) -> int:      # uvnitř funkce je lhuty n-tice
    celkem = 0
    for lhuta in lhuty:
        celkem += lhuta
    return celkem
```

```
>>> soucet_lhut(15, 30, 60, 90)
195
>>> lhuty = [15, 30, 60]
>>> soucet_lhut(*lhuty)                  # hvězdička i při volání: rozbalí seznam na argumenty
105
```

- Hvězdička před parametrem posbírá **všechny poziční argumenty** do n-tice; jméno `args` je jen zvyk
- Tak fungují `print("Lhůta", 15, "dnů")` nebo `max(3, 7, 1)`: berou, kolik jim dáme
- Hvězdička při **volání** dělá opak: rozbalí seznam na jednotlivé argumenty, třeba `print(*smlouvy)`

kwargs: libovolné pojmenované argumenty



```
def vytvor_zaznam(**udaje: str) -> dict[str, str]:   # uvnitř funkce je udaje slovník
    return udaje
```

```
>>> vytvor_zaznam(jmeno="Anna", mesto="Brno")
{'jmeno': 'Anna', 'mesto': 'Brno'}
>>> klient = {"jmeno": "Petr", "mesto": "Praha"}
>>> vytvor_zaznam(**klient)              # dvě hvězdičky při volání rozbalí slovník
{'jmeno': 'Petr', 'mesto': 'Praha'}
```

- Dvě hvězdičky posbírají **všechny pojmenované argumenty**, které funkce nezná, do slovníku; jméno `kwargs` (*keyword arguments*) je zvyk
- Pořadí v hlavičce: běžné parametry (povinné, pak s výchozí hodnotou), `*args`, `**kwargs`
- V kurzu je spíš potkáte v dokumentaci cizích funkcí, než že byste je psali; stačí vědět, co znamenají

CC BY-NC-ND 4.0

Životnost a rozsah platnosti proměnných

```
sazba = 0.21                        # globální: vznikne při spuštění, žije do konce programu

def s_dph(cena: float) -> float:
    dan = cena * sazba              # dan a cena jsou lokální: vzniknou při volání
    return cena + dan               # a po return zaniknou

print(s_dph(1000))                  # 1210.0
print(dan)                          # NameError: name 'dan' is not defined
```

- **Lokální** proměnná vzniká uvnitř funkce (parametry i přiřazení v těle) a **zaniká s návratem**; zvenku není vidět
- **Globální** proměnná vzniká mimo funkce; funkce ji může **číst**, ale přiřazení uvnitř funkce vytvoří novou lokální stejného jména
- Každé volání má své lokální proměnné: `s_dph(1000)` a `s_dph(2000)` si do `dan` navzájem nevidí

CC BY-NC-ND 4.0

Globální proměnné jsou zlo

Funkce dostává data parametry a vrací je přes return. Vždy minimalizujte externí vlivy na funkci.

CC BY-NC-ND 4.0

Knihovny

Hotový kód, který si do programu jen naimportujete

CC BY-NC-ND 4.0

Knihovna: hotové funkce od někoho jiného

Knihovna (modul)

Soubor s funkcemi a hodnotami, které napsal někdo jiný a které si do programu **naimportujete**, místo abyste je psali znovu.

- Odmocnina, náhodné číslo, dnešní datum, čtení CSV: to vše už někdo napsal, otestoval a opravil
- **Standardní knihovna** přichází s Pythonem: `math`, `random`, `datetime`, `csv`, `json` a stovky dalších; nic se neinstaluje
- Cizí knihovny se stahují příkazem `pip install jmeno`; vlastní soubor `pomocne.py` je také modul: `import pomocne`
- Co která knihovna umí: docs.python.org/3/library

CC BY-NC-ND 4.0

import: zpřístupni knihovnu pod jejím jménem

```
>>> import math                          # od teď je knihovna dostupná pod jménem math
>>> math.sqrt(16)                       # tečka: z knihovny math vezmi funkci sqrt
4.0
>>> math.pi                             # knihovna může obsahovat i hodnoty
3.141592653589793
>>> sqrt(16)                            # bez jména knihovny Python funkci nezná
NameError: name 'sqrt' is not defined
```

- `import jmeno` na **začátku souboru**, jednou; pak všechno z knihovny voláme jako `jmeno.funkce()`
- Tečka funguje jako u metod: `text.upper()` je „metoda textu“, `math.sqrt()` je „funkce z knihovny math“
- Čtenář hned vidí, odkud jméno pochází: `sqrt` není naše funkce, ale z `math`

CC BY-NC-ND 4.0

from a as: kratší jména

```
from math import sqrt, pi              # jen vybraná jména, pak se píšou bez předpony
print(sqrt(16) * pi)                   # 12.566370614359172

import random as rnd                   # přejmenování při importu
print(rnd.randint(1, 6))               # hod kostkou: pokaždé jiné číslo od 1 do 6
```

- `from knihovna import jmeno` šetří psaní, ale čtenář už nevidí, odkud `sqrt` je; hodí se pro pár často volaných funkcí
- `import knihovna as zkratka` je obyčejný `import` pod jiným jménem; některé zkratky jsou zvyk (`import numpy as np`)
- `from knihovna import *` nepište: do programu se vysype vše a nikdo neví, co odkud přišlo

CC BY-NC-ND 4.0

Připomenutí matematiky

Logaritmus a polynom: dva pojmy, které potřebujeme pro složitost

CC BY-NC-ND 4.0

Logaritmus

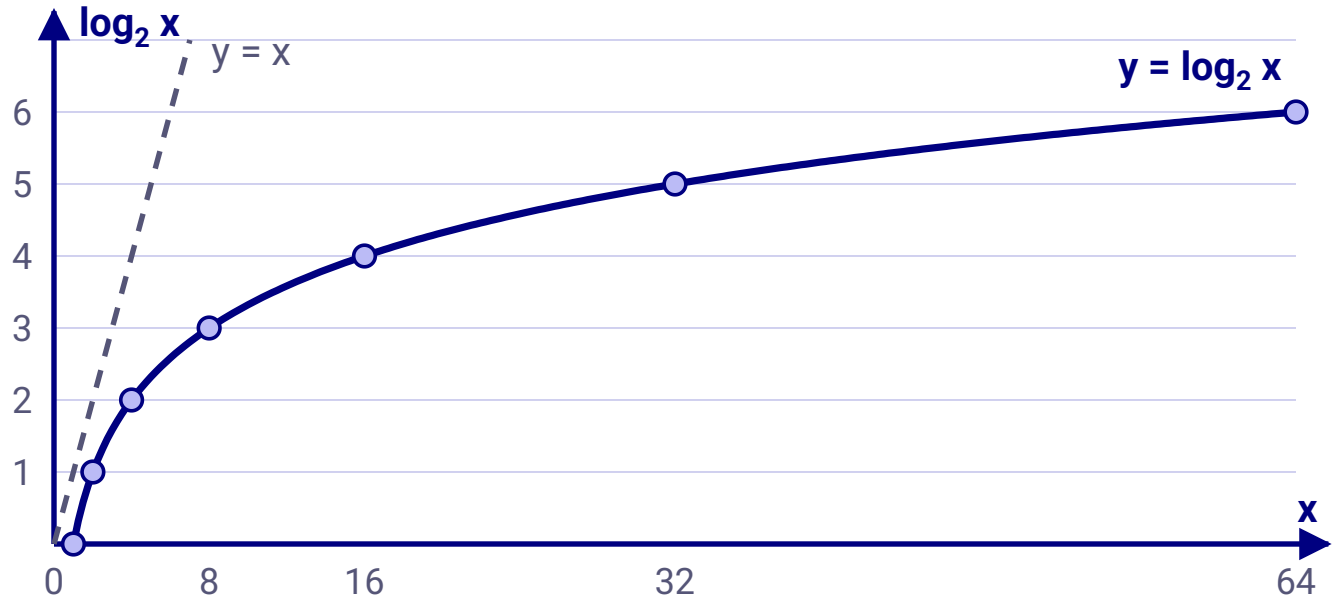
- $\log_2 x = y$ znamená $2^y = x$: logaritmus je **opak mocniny**. Mocnina říká „dvojku umocni desetkrát“, logaritmus „kolikrát umocnit dvojku, aby vyšlo 1024“
- $\log_a 1 = 0$ pro každé a
- Roste velmi pomalu
- Základ může být i jiný (10, e); v informatice skoro vždy 2 a změna základu mění výsledek jen násobkem

$$a^b = c$$

$$\log_a c = b$$

CC BY-NC-ND 4.0

Graf logaritmu



zdvojnásobení x přidá k $\log_2 x$ vždy jen jedničku: 16 → 4, 32 → 5, 64 → 6

- Křivka se **rychle zplošťuje**: každé zdvojnásobení x přidá k $\log_2 x$ jen jedničku
- Pro srovnání přímka $y = x$: z grafu vyletí už u $x = 7$, logaritmus se na 6 dostane až u $x = 64$

CC BY-NC-ND 4.0

Polynom: mocniny x s koeficienty

Definice polynomu

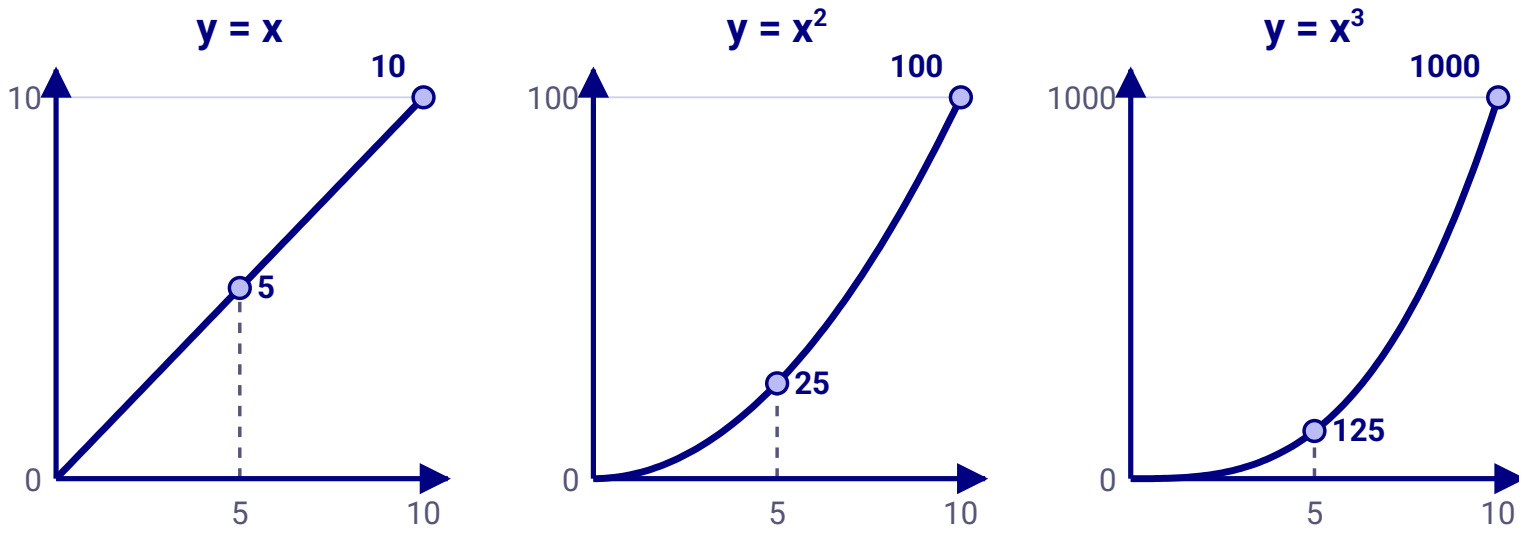
Polynom je součet členů $a_k x^k$, kde a_k je reálné číslo a k celé číslo od nuly výš.

stupeň polynomu je nejvyšší mocnina ($3x^2 + 2x + 1$ má stupeň 2)

Polynom	Stupeň	Hodnota pro $x = 10$
$2x + 1$	1	`21`
x^2	2	`100`
$3x^2 + 2x + 1$	2	`321`, v Pythonu `3 * x**2 + 2 * x + 1`
$x^3 - x$	3	`990`

- Pro velké x rozhoduje nejvyšší mocnina: při $x = 1000$ dá $3x^2$ tři miliony, zbytek jen 2001

Grafy polynomů: stupeň 1, 2 a 3



pro x = 5 a x = 10: dvakrát větší x dá 2×, 4×, 8× větší hodnotu

- Stejná osa x od 0 do 10, ale osa y končí na 10, 100 a 1000: stupeň určuje, **jak rychle hodnota roste**
- Dvojnásobné x dá u stupně 1 dvojnásobek, u stupně 2 čtyřnásobek, u stupně 3 osminásobek: 2^k pro stupeň k

CC BY-NC-ND 4.0

Co je a není polynom

Výraz	Polynom?	Proč
$3x^2 + 2x + 1$	ano	celé nezáporné mocniny x s koeficienty
x^{10}	ano	vysoký stupeň nevadí, pořád jen mocnina x
7	ano	stupeň 0, totéž co $7x^0$
2^x	ne	x je v exponentu, ne v základu: **exponenciála**
$\log_2 x$	ne	logaritmus není mocnina
$\sqrt{x}, 1/x$	ne	mocnina $1/2$ a -1: není celé nezáporné číslo

- Rozdíl je obrovský: pro $x = 100$ je $x^2 = 10\,000$, ale $2^x \approx 1{,}3 \cdot 10^{30}$
- Proto se to rozlišuje: polynomiální počet kroků algoritmu je „zvládnutelný“, exponenciální ne

CC BY-NC-ND 4.0

Složitost

Jak poznat rychlý postup od pomalého, ještě než ho spustíme

CC BY-NC-ND 4.0

Jsou všechny postupy, které řeší stejný problém, stejně dobré?

Algoritmus

Přesný postup jak k řešení nějakého problému.

- Jeden problém, mnoho algoritmů: seřadit seznam, najít paragraf nebo sečíst čísla jde vždy několika způsoby
- Jsou ale všechna korektní řešení stejně dobrá?

CC BY-NC-ND 4.0

Jak měřit efektivitu programu?

- **Změříme čas na stopkách?**
  - Na různých počítačích poběží různě dlouho
- **Spočítáme počet instrukcí?**
  - Lepší: nezávisí rychlosti počítače. Ale jiný jazyk nebe instrukční sadu dá jiný počet.
- **Spočítáme, jak počet kroků roste s velikostí vstupu.**
  - Dáme programu desetkrát víc dat: udělá desetkrát víc kroků? Stokrát? O jeden víc?

CC BY-NC-ND 4.0

Stejný výsledek, jiná cena: 1 + 2 + … + 100

```
celkem = 0
for i in range(1, 101):           # sto sčítání; pro milion čísel milion sčítání
    celkem += i
print(celkem)                     # 5050

n = 100
print(n * (n + 1) // 2)           # 5050: jedno násobení a jedno dělení, pro milion totéž
```

- Oba postupy jsou správné; první udělá **n kroků**, druhý **pořád dva**, ať je n jakékoliv
- Totéž v papírovém světě: paragraf v zákoníku najdete listováním od první strany, nebo jedním pohledem do obsahu
- Legenda o malém Gaussovi: 1 + 100 = 2 + 99 = … = 50 + 51, padesát dvojic po 101, hotovo za minutu

CC BY-NC-ND 4.0

Časová složitost: počet kroků jako funkce velikosti vstupu

Časová složitost

Kolik kroků algoritmus udělá pro vstup velikosti n, zapsáno jako funkce n: třeba n, n^2 nebo $\log_2 n$.

- **Velikost vstupu** n: počet prvků seznamu, počet znaků textu, počet řádků souboru
- **Krok**: jedna jednoduchá operace: porovnání, sčítání, přiřazení, přístup k prvku podle indexu
- Sečtení cyklem: n sčítání, složitost n; vzorec $n(n+1)/2$: dvě operace vždy, složitost **konstantní**
- Nezáleží na jazyce ani počítači, a program nemusíme ani spustit: počítáme z postupu, ne z běhu

CC BY-NC-ND 4.0

Tři dohody, které měření zjednoduší

- **Nejhorší případ.** Hledaný prvek může být první, nebo až poslední; počítáme s tím nejhorším, protože ten nás překvapí
- **Zajímá nás růst pro velká n.** Pro deset prvků je rychlé všechno; rozdíly se ukážou u milionu
- **Konstanty neřešíme.** Program s $2n$ kroky a program s n kroky rostou stejně: zdvojnásobíme vstup, zdvojnásobí se práce. Aditivní konstanta ($n + 1000$) se u velkých n ztratí úplně

CC BY-NC-ND 4.0

O-notace: zápis „roste nejvýš jako“

O-notace

$O(f(n))$ čteme „řádově $f(n)$“: počet kroků pro velká n roste nejvýš jako $f(n)$, až na násobící konstantu.

Ukázka: algoritmus udělá $10n^4 + 1\,000\,000 + 30n^2 + 99n$ kroků.

1. Seřadíme členy podle rychlosti růstu: $10n^4 + 30n^2 + 99n + 1\,000\,000$
2. Pro velká n rozhoduje nejrychleji rostoucí člen, ostatní škrtneme: $10n^4$
3. Násobící konstantu škrtneme také: n^4

Výsledek: algoritmus má složitost $O(n^4)$, čteme „ó od n na čtvrtou“.

CC BY-NC-ND 4.0

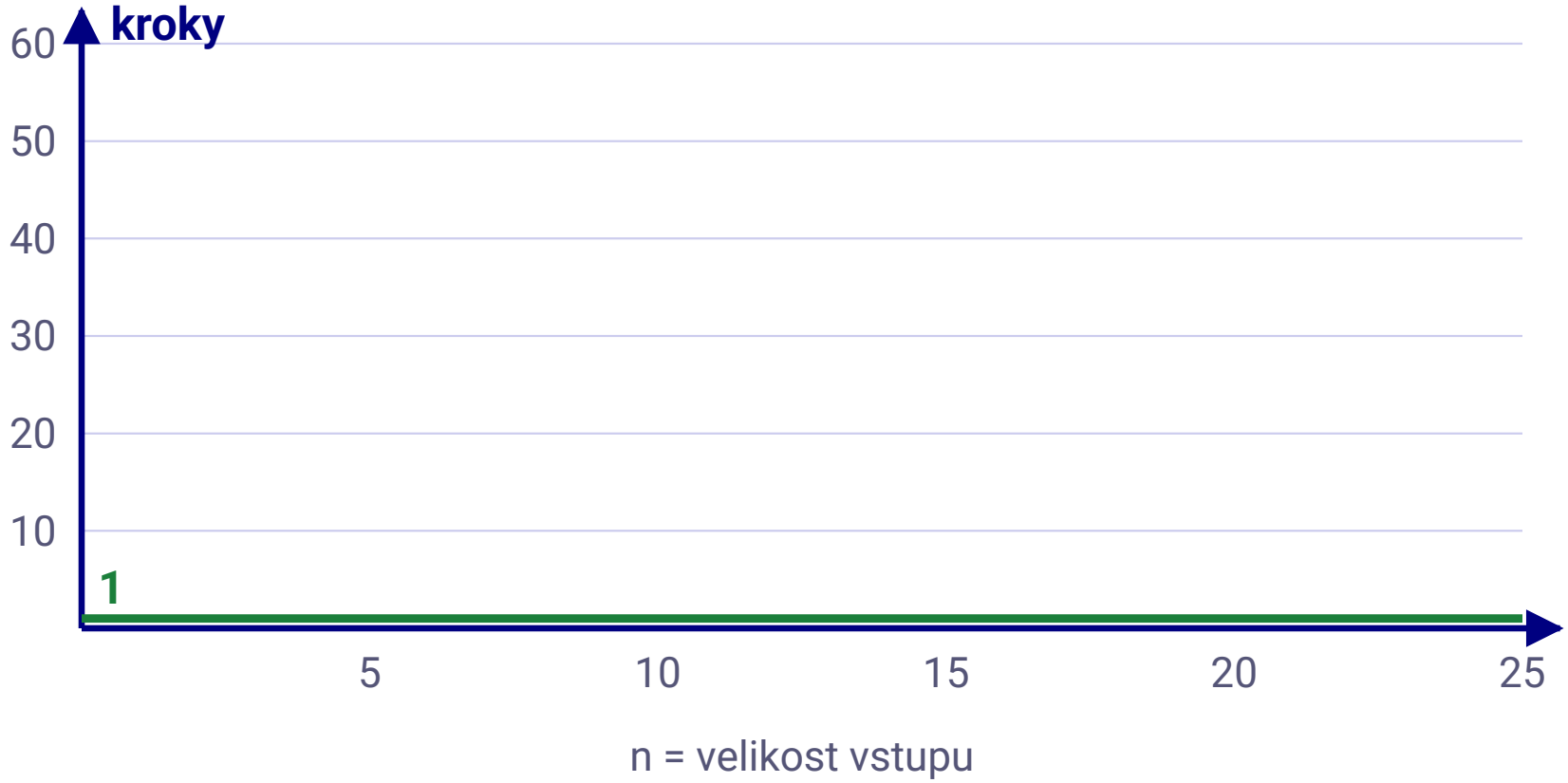
Proč smíme škrtat: hodnoty členů pro rostoucí n

n	$10n^4$	zbytek: $30n^2 + 99n + 1\,000\,000$	$10n^4$ oproti zbytku
10	100 000	1 003 990	desetkrát **menší**
100	1 000 000 000	1 309 900	760× větší
1 000	10 000 000 000 000	31 099 000	320 000× větší

- Pro malé n vyhrává milionová konstanta; od n v řádu stovek je zbytek vedle $10n^4$ zanedbatelný a s rostoucím n to je jen horší
- O-notace popisuje právě toto chování „pro dost velká n“; na malých vstupech může vyhrát i algoritmus s horší složitostí

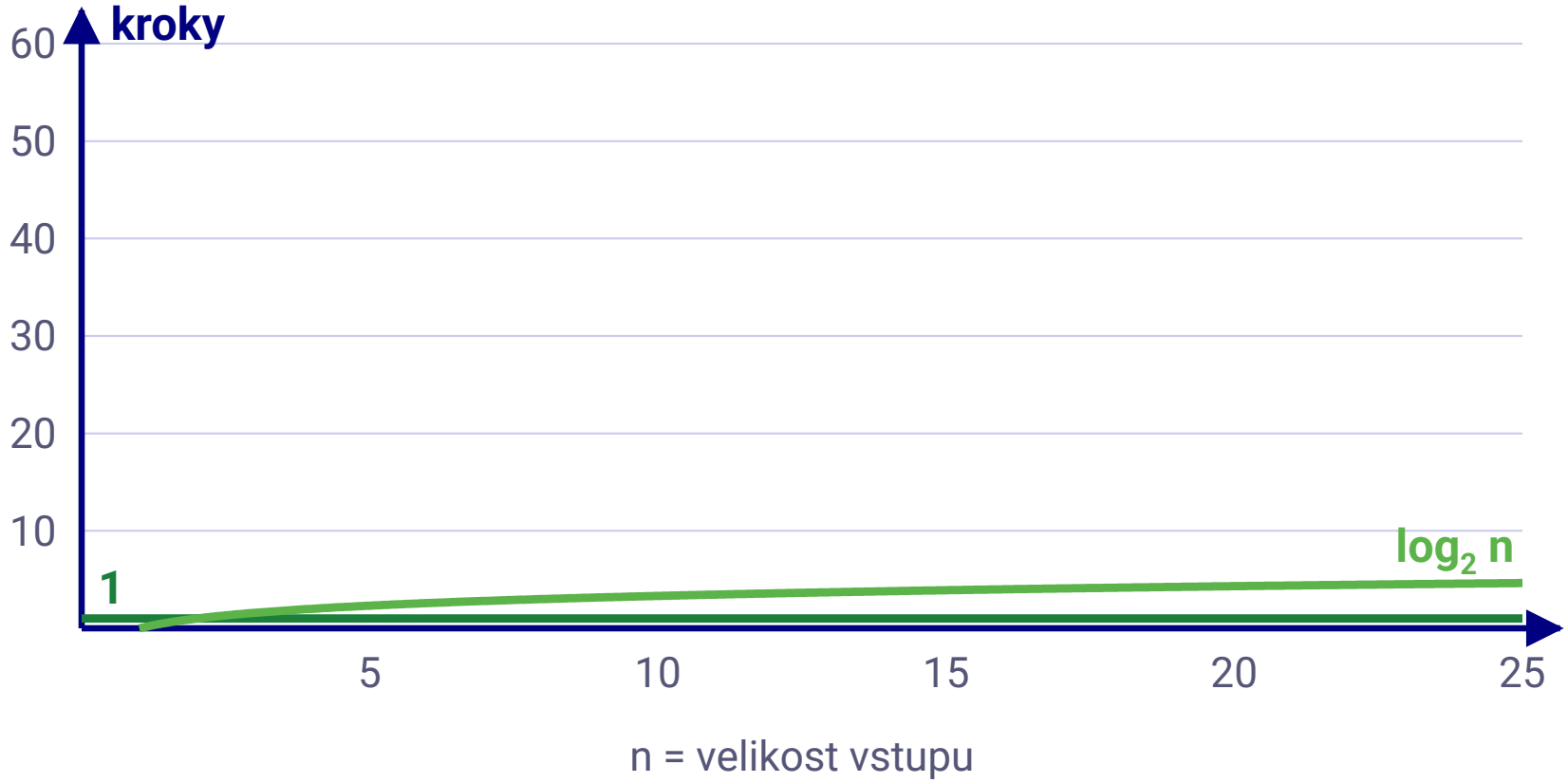
CC BY-NC-ND 4.0

Řády růstu: konstantní, O(1)



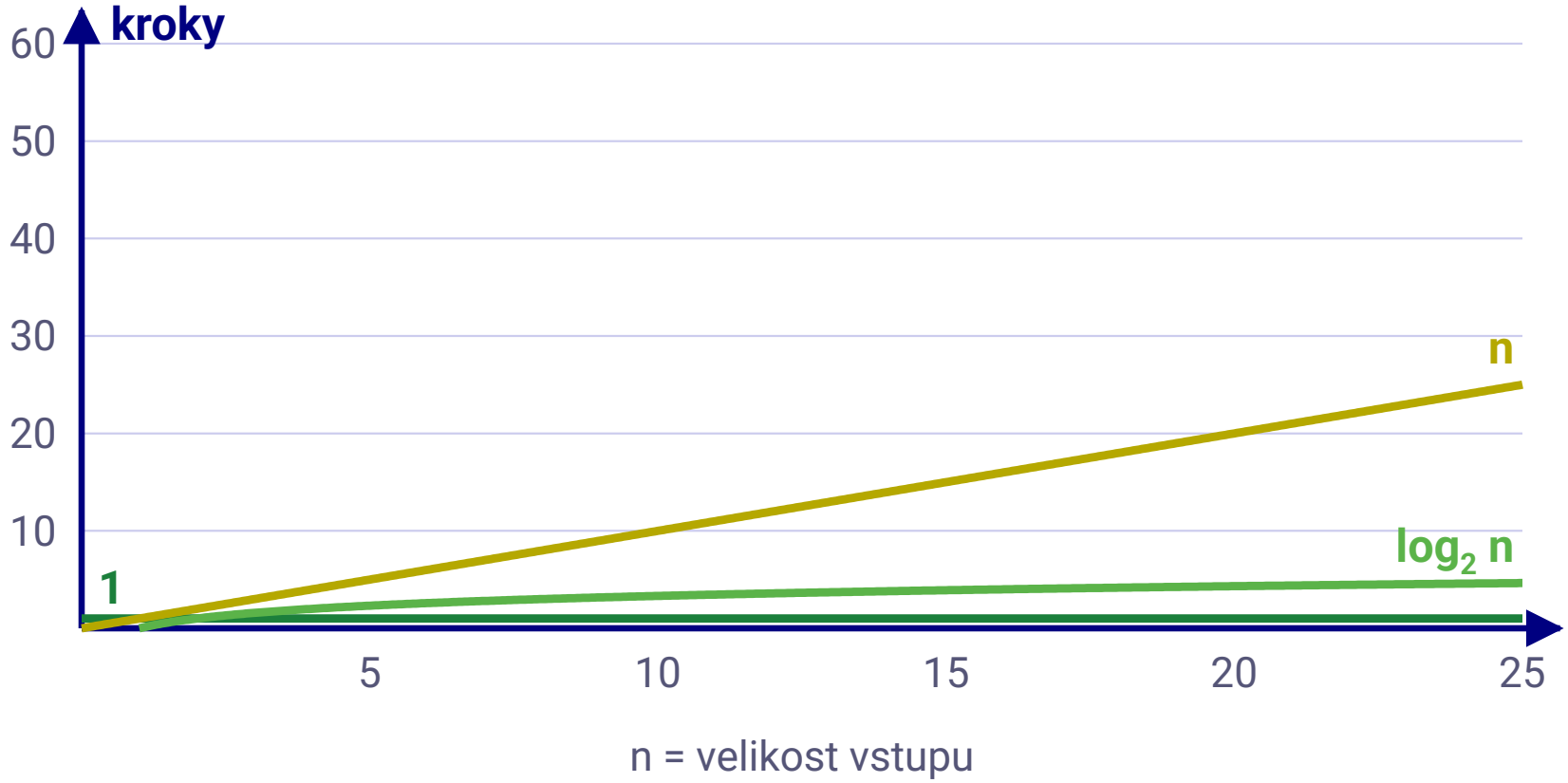
CC BY-NC-ND 4.0

Řády růstu: logaritmická, O(log n)



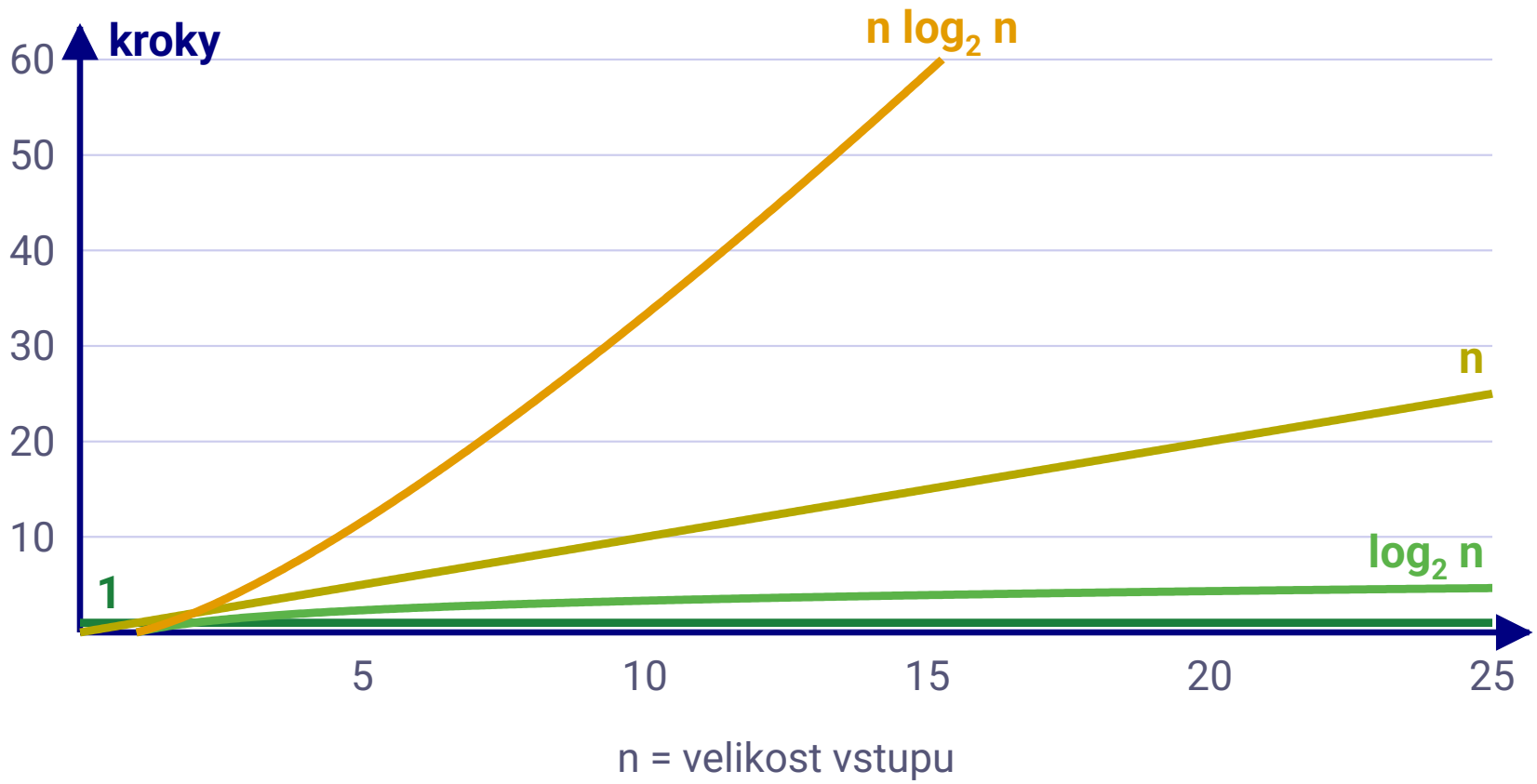
CC BY-NC-ND 4.0

Řády růstu: lineární, O(n)

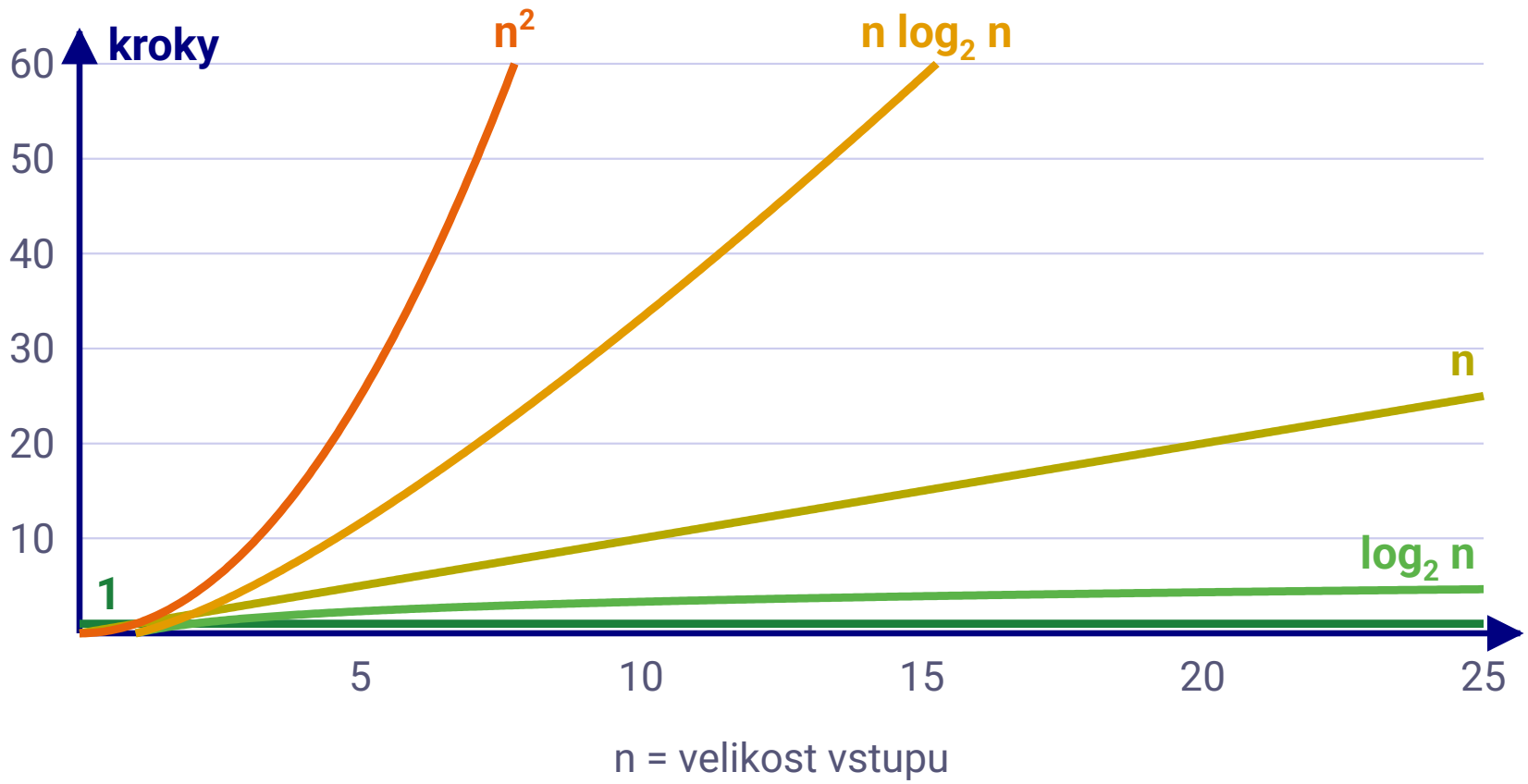


CC BY-NC-ND 4.0

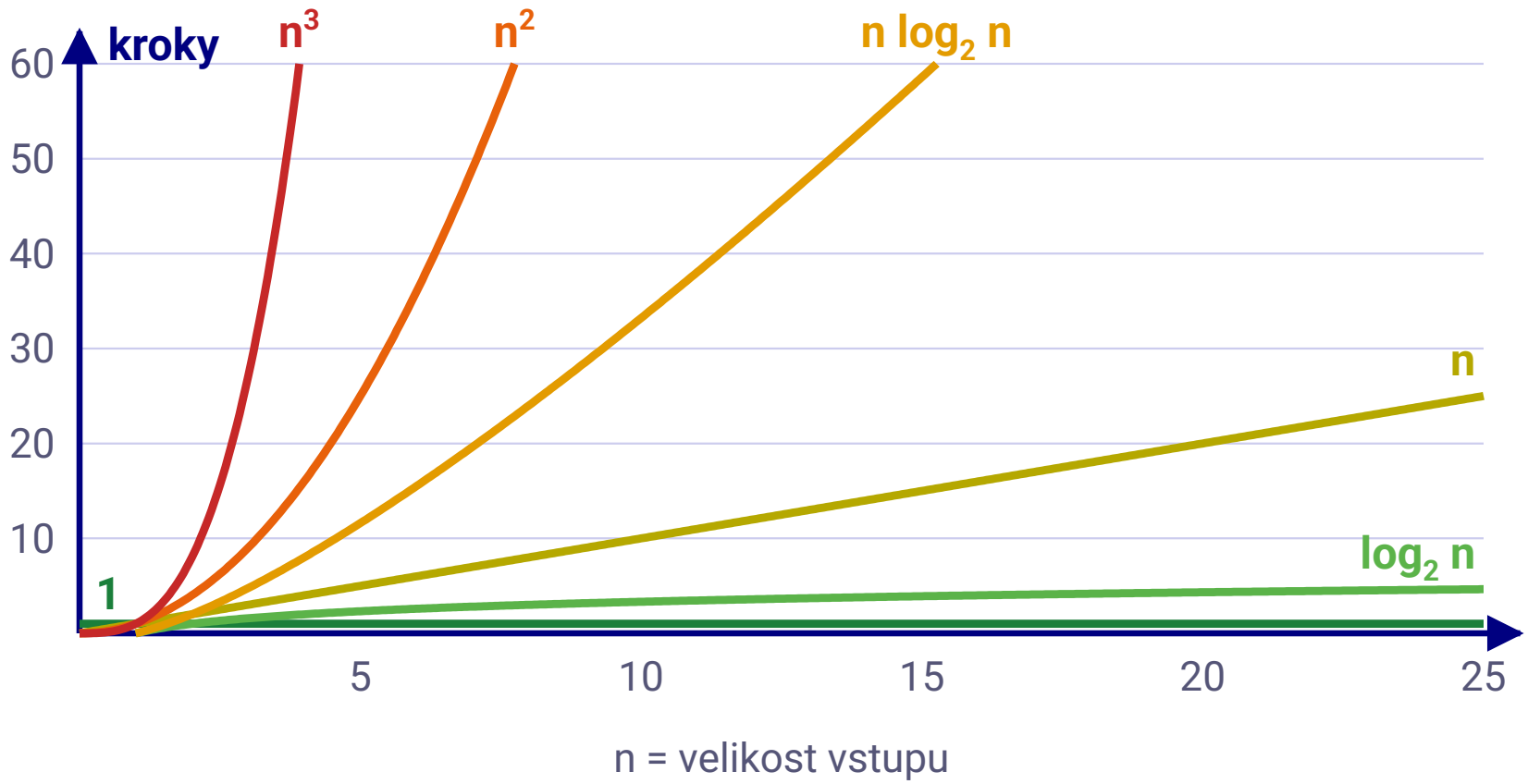
Řády růstu: n log n, O(n log n)



Řády růstu: kvadratická, O(n²)

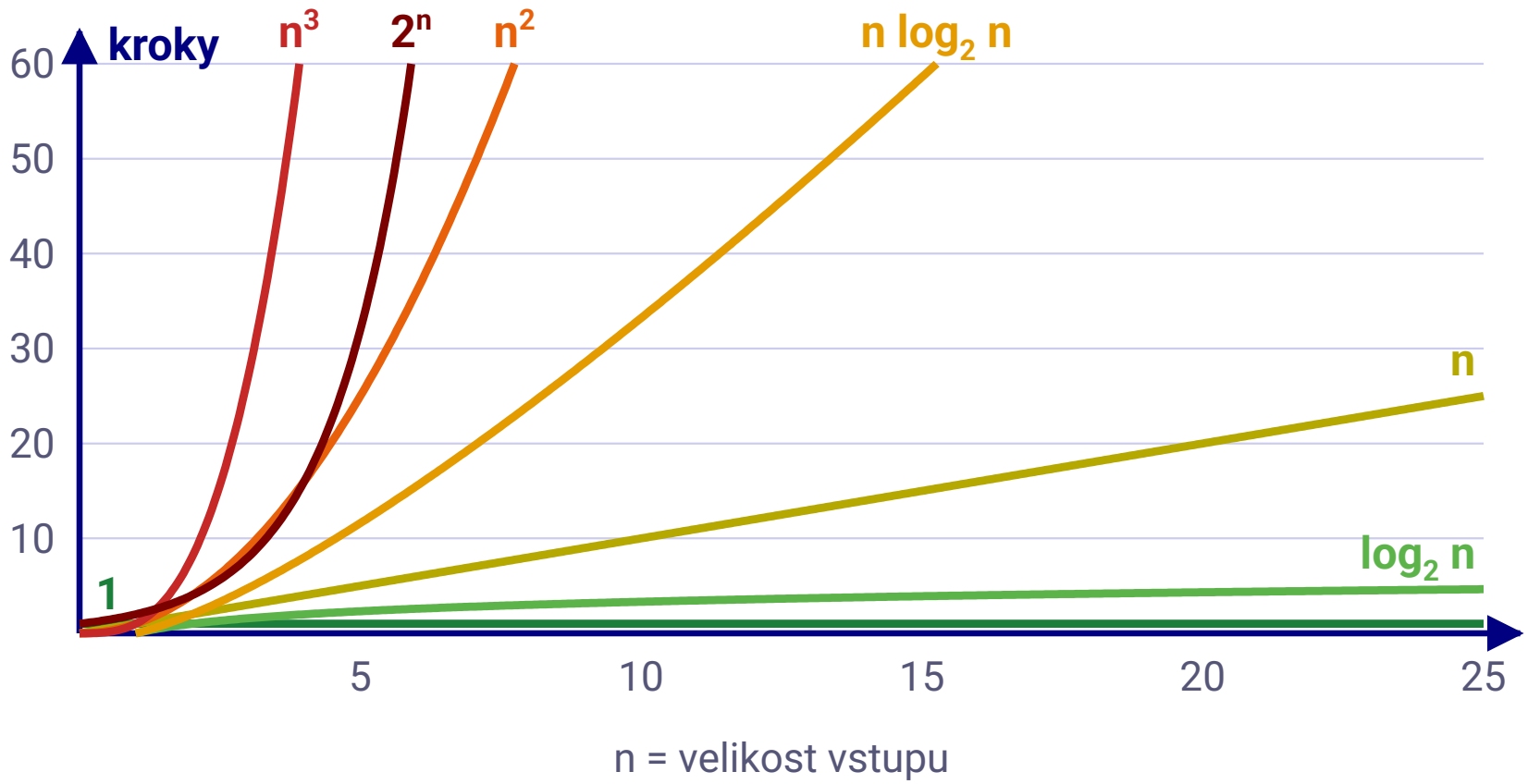


Řády růstu: kubická, O(n³)



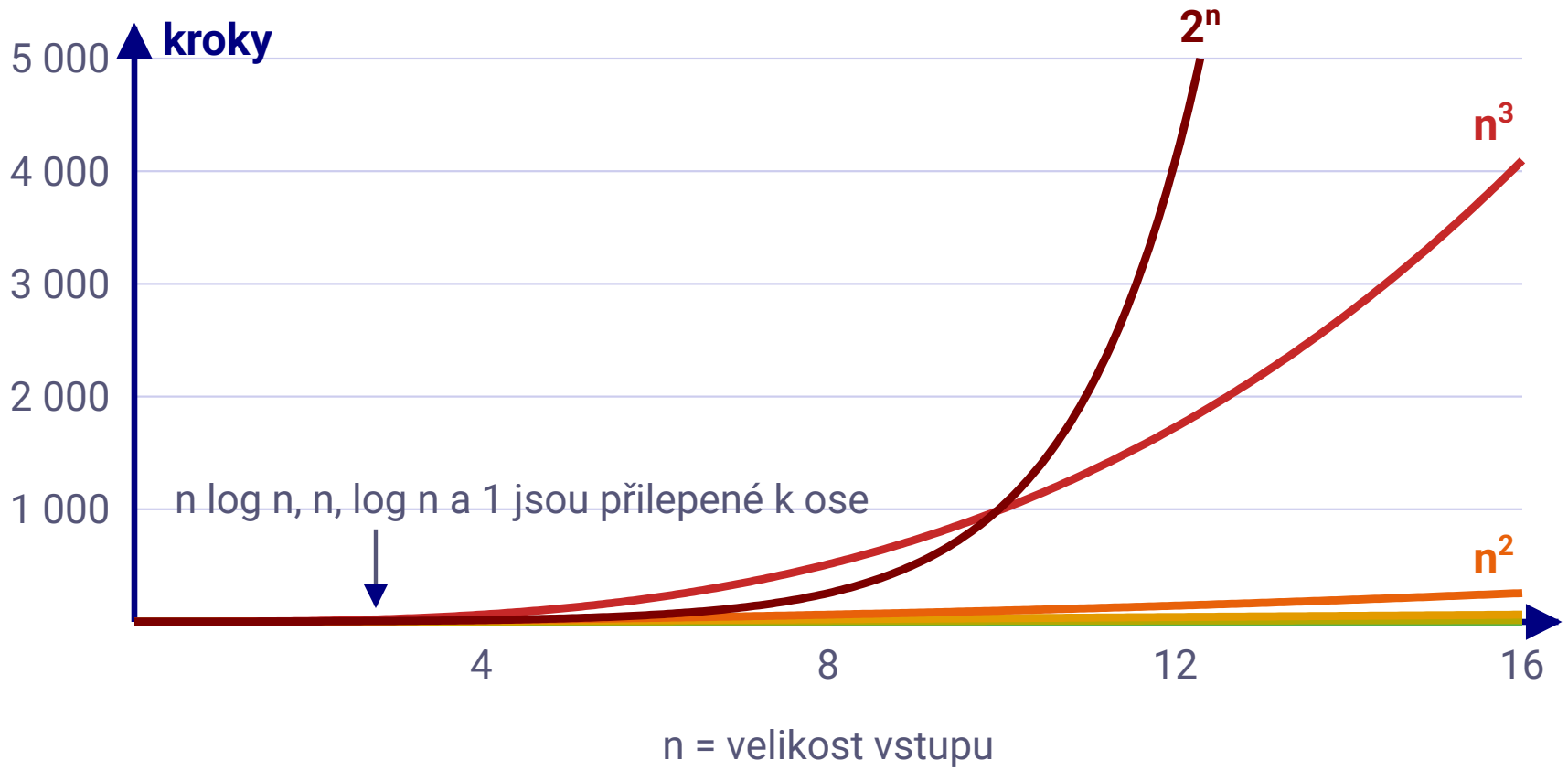
CC BY-NC-ND 4.0

Řády růstu: exponenciální, $O(2^n)$



CC BY-NC-ND 4.0

Totéž z "větší dálky"



- Stačí n do 16 a 2^n předběhne n^3; všechno pomalejší než n^2 se přilepilo k ose
- Jakýkoliv polynom nakonec exponenciála přeroste, a od toho okamžiku už se nic nedohání

Polynomiální je zvládnutelné, exponenciální ne

Pravidlo

Složitost $O(n^k)$ pro pevné k je **polynomiální** a považujeme ji za použitelnou. $O(2^n)$ a horší je **exponenciální**: použitelná jen pro malé vstupy.

n	n^2	n^3	2^n	2^n kroků při miliardě kroků za sekundu
10	100	1 000	1 024	okamžitě
30	900	27 000	asi 1 miliarda	1 sekunda
50	2 500	125 000	asi 10^{15}	13 dní
100	10 000	1 000 000	asi 10^{30}	$4 \cdot 10^{13}$ let, tisíckrát víc než stáří vesmíru

- Kubický algoritmus pro sto prvků zvládne počítač za milisekundu; exponenciální se ze sta prvků nevzpamatuje nikdy

CC BY-NC-ND 4.0

Příklad: hledání v seřazeném seznamu

Zadání

Máme **seřazený** seznam čísel a hledané číslo. Najděte index, na kterém hledané číslo leží, nebo zjistěte, že v seznamu není.

```
paragrafy = [140, 173, 205, 206, 207, 209, 329]      # seřazeno vzestupně
hledane = 207                                        # očekávaný výsledek: index 4
```

- Co je **velikost vstupu** n?
  - Počet prvků seznamu: zde 7, v praxi klidně milion

CC BY-NC-ND 4.0

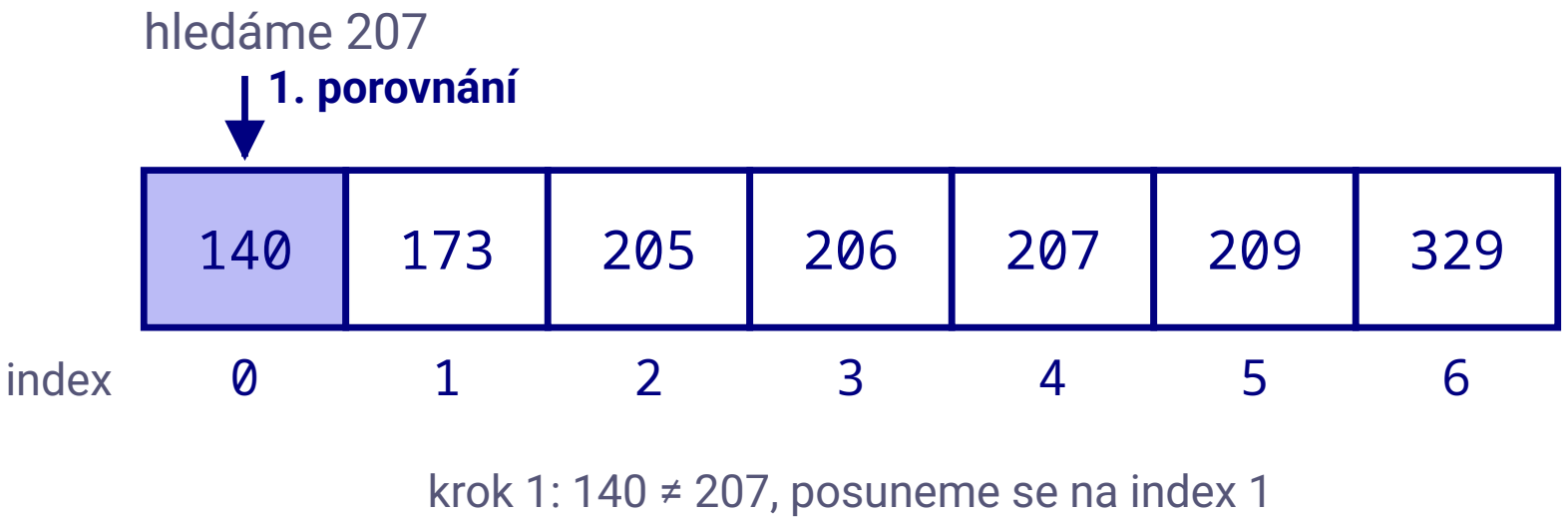
První přístup: jít postupně od začátku

```python
def najdi_postupne(seznam: list[int], hledane: int) -> int | None:
    """Vrátí index hledaného čísla, nebo None, když v seznamu není."""
    for index in range(len(seznam)):          # indexy 0, 1, 2, … až po poslední
        if seznam[index] == hledane:
            return index                      # našli jsme, dál nehledáme
    return None                               # prošli jsme vše a nic
```

```
>>> paragrafy = [140, 173, 205, 206, 207, 209, 329]
>>> najdi_postupne(paragrafy, 207)            # pět porovnání
4
>>> print(najdi_postupne(paragrafy, 999))     # sedm porovnání a teprve pak odpověď
None
```

- Otázka: **jakou to má časovou složitost?** Kolik porovnání uděláme v nejhorším případě pro seznam délky n?

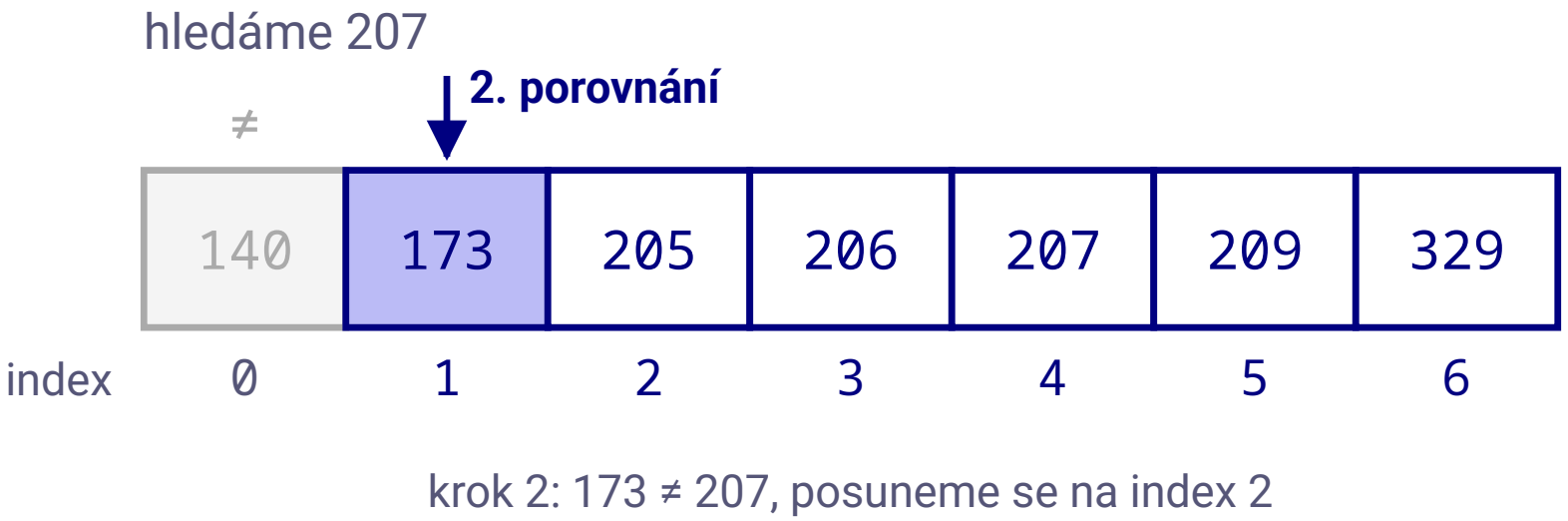
Postupné hledání: krok 1



- Začínáme na indexu 0: porovnáme 140 s hledaným 207, nerovná se, posuneme se o jeden index

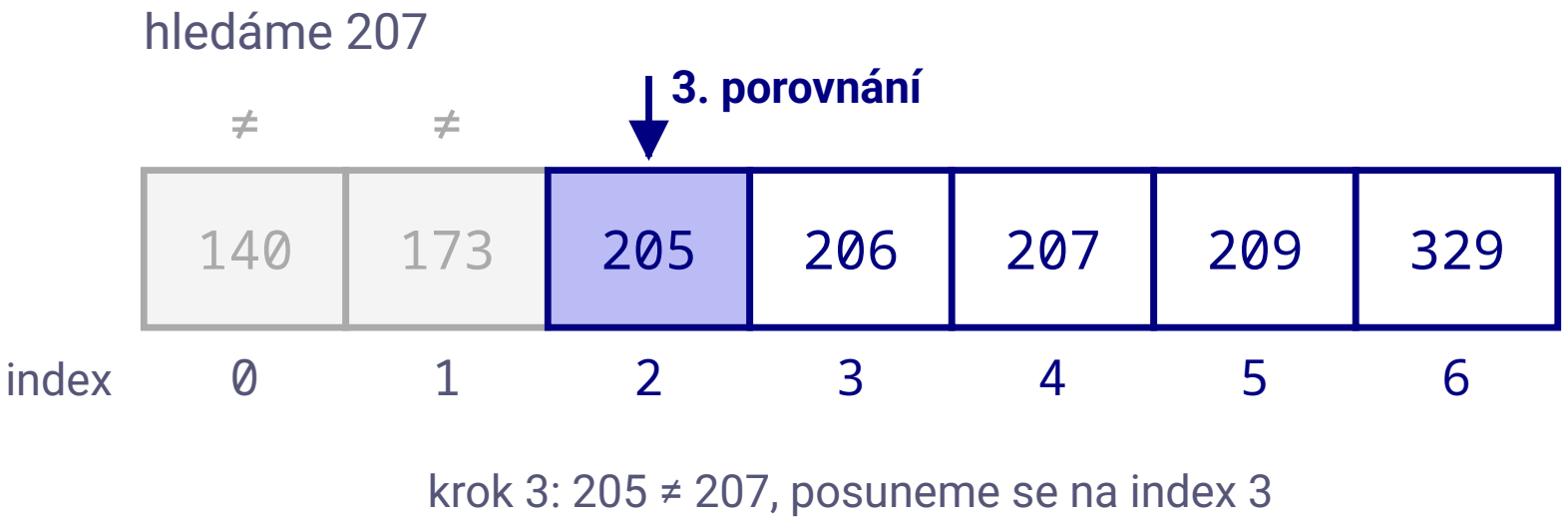
CC BY-NC-ND 4.0

Postupné hledání: krok 2



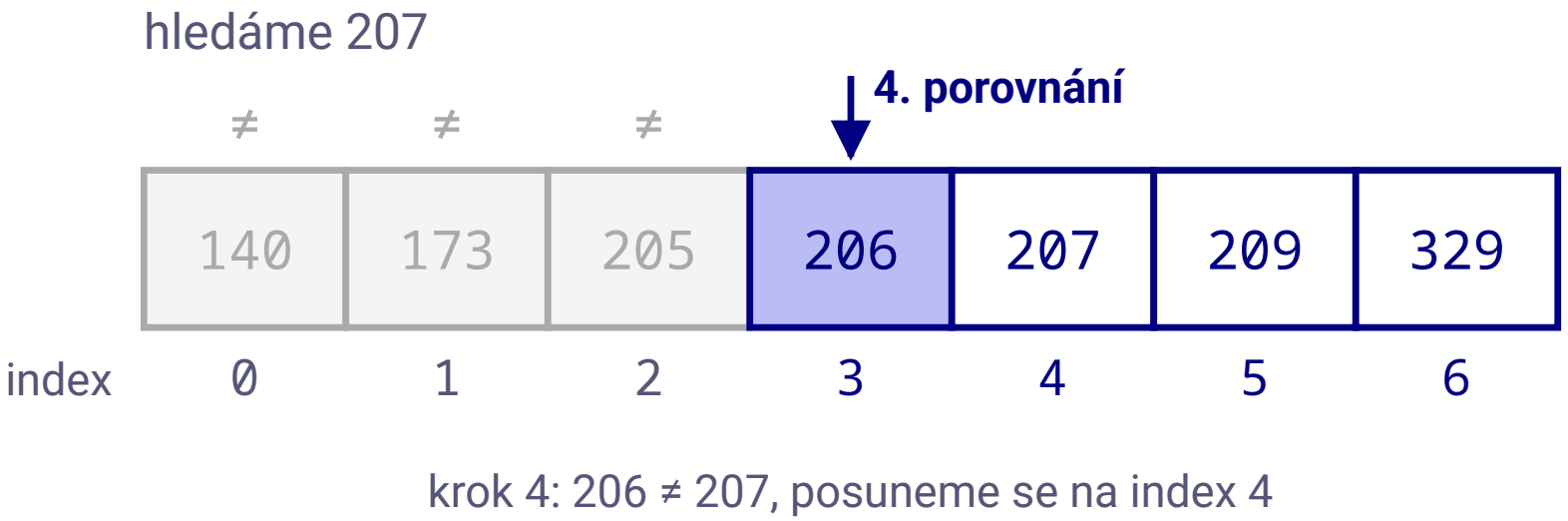
- 173 také není 207; dvě porovnání, a víme jen to, že hledané číslo není na indexech 0 a 1

Postupné hledání: krok 3



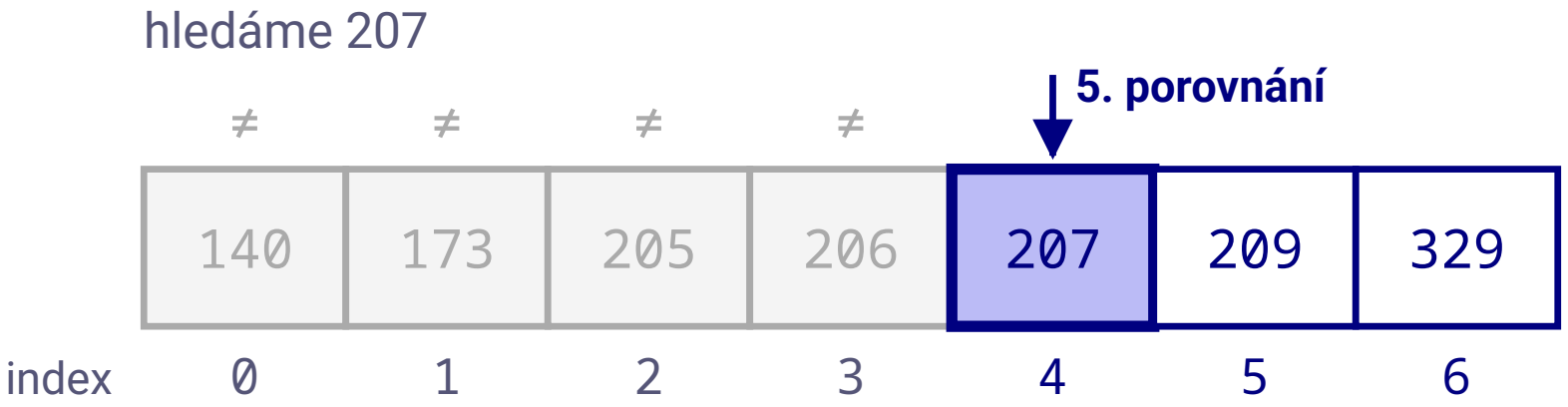
- 205 není 207; každé porovnání vyřadí **jediný prvek**, zbytek seznamu zůstává neprozkoumaný

Postupné hledání: krok 4



- 206 není 207; čtyři porovnání za námi a stále nevíme, jestli hledané číslo v seznamu vůbec je

Postupné hledání: krok 5

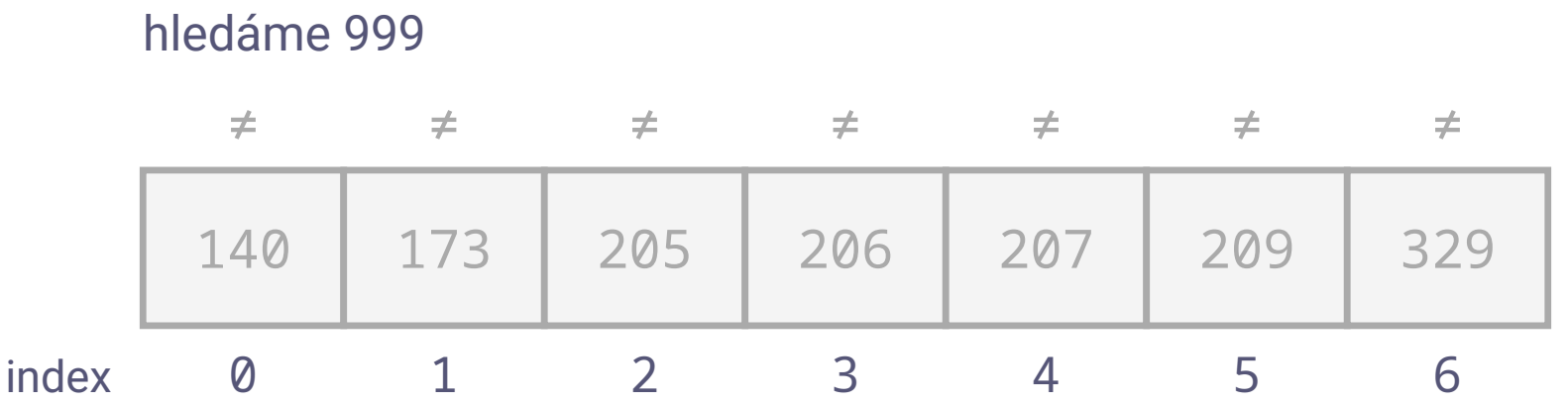


krok 5: 207 = 207, nalezeno na indexu 4 po 5 porovnáních

- 207 = 207: **nalezeno na indexu 4** po pěti porovnáních; kdyby bylo hledané číslo 329, bylo by jich sedm

CC BY-NC-ND 4.0

Postupné hledání je lineární: O(n)



všech sedm porovnání dopadlo ≠, teprve teď víme, že 999 v seznamu není

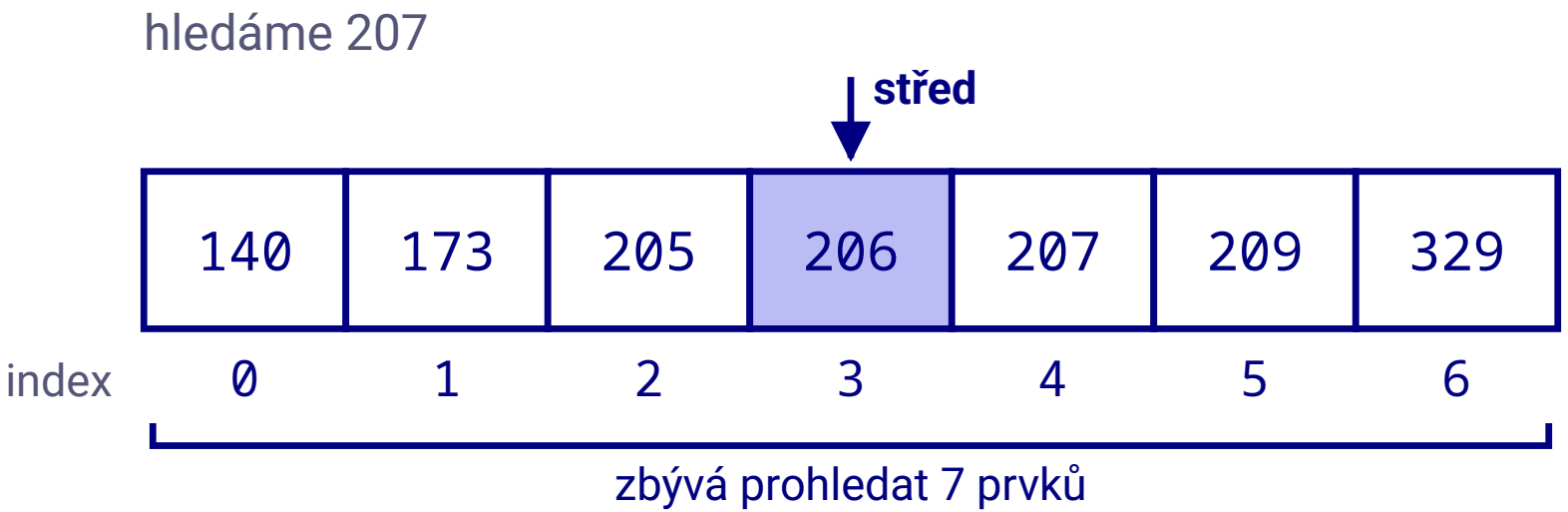
- Nejhorší případ: hledané číslo je **poslední**, nebo v seznamu **vůbec není**; pak ho porovnáme se všemi n prvky
- Počet kroků roste přímo úměrně s n: desetkrát delší seznam, desetkrát víc porovnání; milion prvků, milion porovnání
- Že je seznam seřazený, jsme **vůbec nevyužili**

CC BY-NC-ND 4.0

Jde to rychleji?

CC BY-NC-ND 4.0

Půlení: podívej se doprostřed (krok 1)

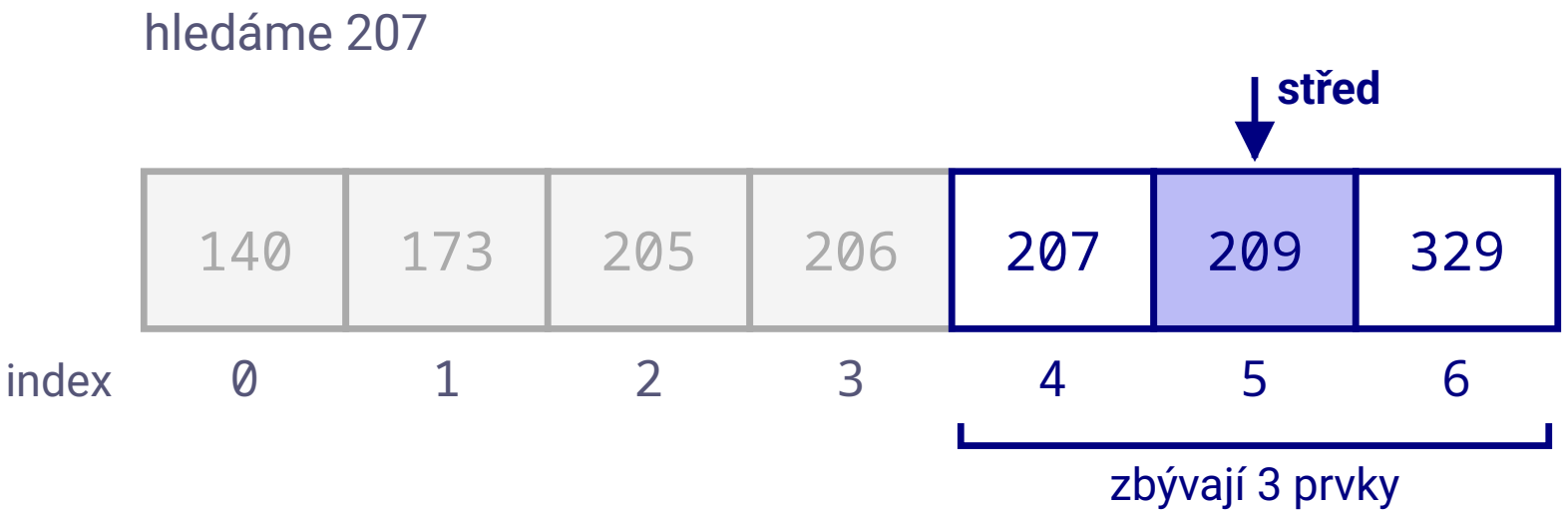


krok 1: střed je 206 < 207, hledané číslo může ležet jen vpravo; indexy 0 až 3 vyřazeny

- Prostřední prvek je 206. Hledáme 207, a to je **větší**: v seřazeném seznamu tedy může ležet jen **vpravo** od středu
- Jedním porovnáním jsme vyřadili polovinu seznamu: indexy 0 až 3 už nás nezajímají

CC BY-NC-ND 4.0

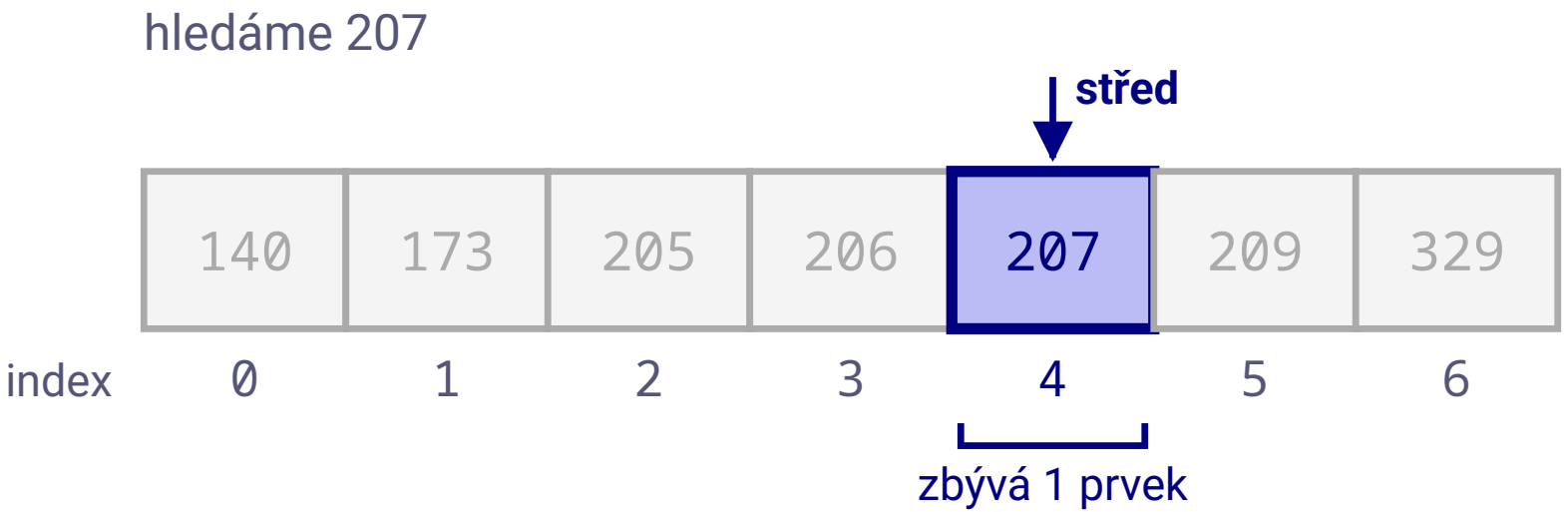
Půlení: podívej se doprostřed zbytku (krok 2)



krok 2: střed je 209 > 207, hledané číslo může ležet jen vlevo; indexy 5 a 6 vyřazeny

- Ze zbývajících tří prvků je prostřední 209. Hledáme 207, a to je **menší**: může ležet jen **vlevo** od středu
- Druhé porovnání vyřadilo další polovinu zbytku: zbývá jediný prvek

Půlení: poslední prvek (krok 3)

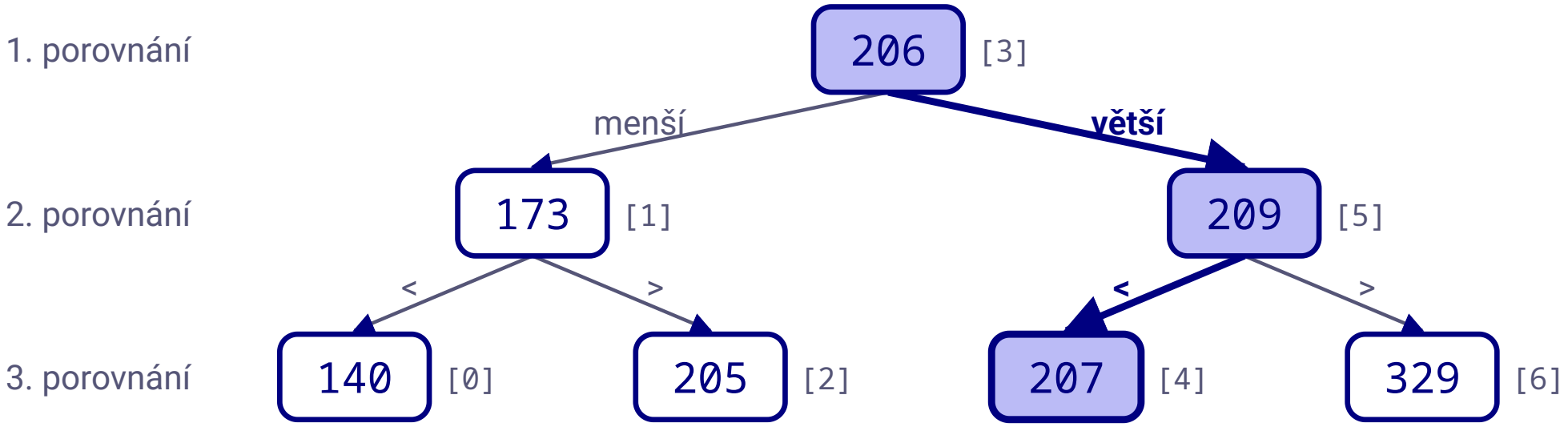


krok 3: střed je 207 = 207, nalezeno na indexu 4; tři porovnání místo pěti

- Zbyl jediný prvek, 207 = 207: **nalezeno na indexu 4** po třech porovnáních; postupně jsme jich potřebovali pět
- Kdyby zbylý prvek nebyl ten hledaný, nemáme kam dál půlit: číslo v seznamu **není**

CC BY-NC-ND 4.0

Půlení jako binární strom



hledáme 207: vpravo, vlevo, nalezeno; tři patra pro sedm prvků, protože $\log_2 8 = 3$

- Kořen je prostřední prvek, vlevo jsou menší, vpravo větší; hledání je **cesta od kořene dolů**, v každém uzlu jedno porovnání
- Pater je tolik, kolikrát jde n vydělit dvěma, a víc porovnání než pater být nemůže; seznam zůstává seznamem, strom je jen obrázek rozhodování

Jakou to má časovou složitost?

Kolikrát jde n vydělit dvěma, než zbude jediný prvek?

CC BY-NC-ND 4.0

Půlení je logaritmické: O(log n)

po kroku	0	1	2	3	…	k
zbývá prvků	n	$n/2$	$n/4$	$n/8$	…	$n/2^k$

- Končíme, když zbude jeden prvek: $n/2^k = 1$, tedy $2^k = n$ a $k = \log_2 n$; přesně proto jsme logaritmus opakovali
- Milion prvků: postupně až milion porovnání, půlením **20**; miliarda prvků: **30**
- Podmínka: seznam musí být **seřazený**; řazení stojí $O(n \log n)$ a vyplatí se, když pak hledáme mnohokrát

CC BY-NC-ND 4.0

Datové struktury a složitost

Stejné operace, jiné vnitřnosti, jiná cena

CC BY-NC-ND 4.0

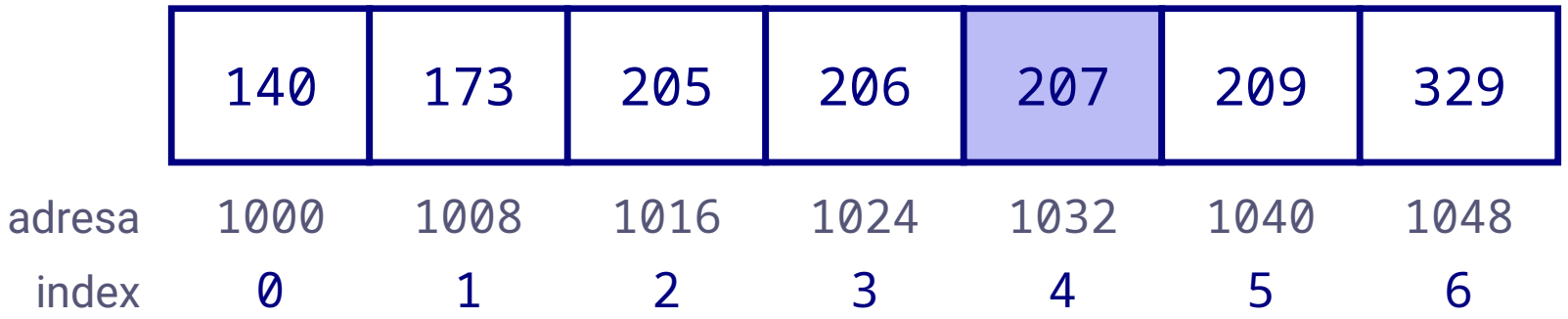
Pole a spojový seznam: dva způsoby, jak držet prvky za sebou

- Obě struktury dělají navenek **totéž**: drží prvky v daném pořadí, jako `list` v Pythonu
- Liší se tím, **jak jsou uložené v paměti**, a to rozhoduje o ceně každé operace
- Operace, které po nich chceme:
  - najdi prvek na indexu i
  - přidej prvek na konec
- Otázka pro každou operaci: jakou mají časovou složitost, když je prvků n?

CC BY-NC-ND 4.0

Pole: souvislý kus paměti

```
paragrafy = [140, 173, 205, 206, 207, 209, 329]
```



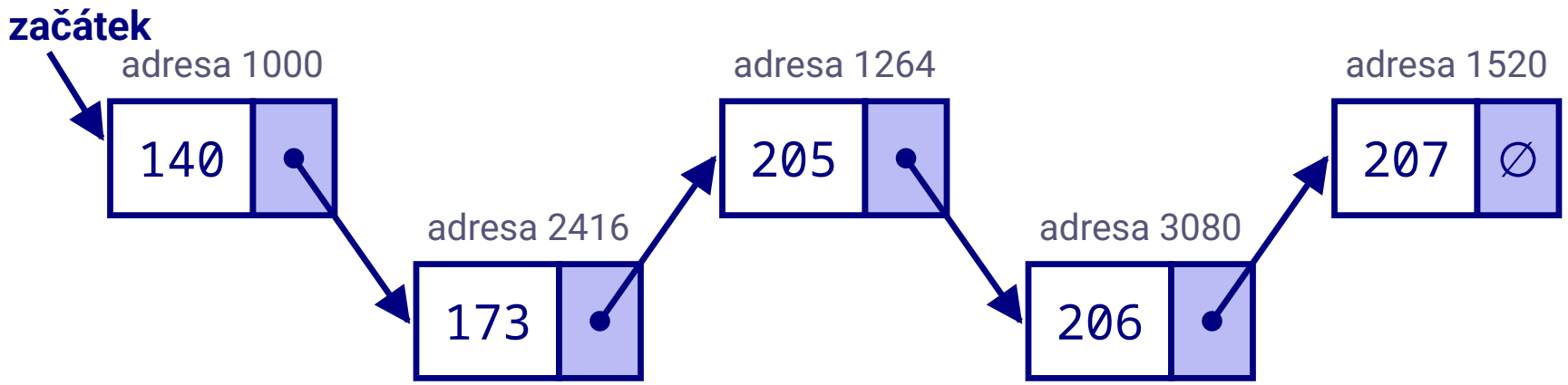
každý prvek zabírá 8 bajtů, všechny leží těsně za sebou

adresa prvku i = 1000 + 8 · i, tedy prvek 4 leží na 1000 + 8 · 4 = 1032

Prvky leží v paměti **těsně za sebou** a každý je stejně velký; adresu prvku i spočítáme jako začátek + i · velikost

CC BY-NC-ND 4.0

Spojový seznam: prvky rozházené po paměti, spojené odkazy



uzel = hodnota a adresa dalšího uzlu; prvek na indexu 4 najdeme až po čtyřech šipkách

- Každý prvek (uzel) nese hodnotu a **adresu dalšího uzlu**; struktura si pamatuje jen adresu prvního (a posledního)

Otázka: jakou složitost mají jednotlivé operace?

Operace	Pole	Spojový seznam
prvek na indexu i	?	?
přidání na konec	?	?

- Pro každou buňku: co se musí v paměti stát? Účastní se toho jeden prvek, nebo všech n?

CC BY-NC-ND 4.0

Odpověď: stejné operace, jiná cena

Operace	Pole	Spojový seznam
prvek na indexu i	$O(1)$: výpočet adresy	$O(n)$: projít až n odkazů
přidání na konec	$O(n)$: překopírovat	$O(1)$: nový uzel za poslední

- Pythonový `list` je pole: proto je `seznam[i]` okamžité, `.append()` rychlé a `.insert(0, x)` nebo `.pop(0)` u dlouhých seznamů pomalé
- Žádná struktura není nejlepší na všechno; vybíráme podle operací, které program dělá nejčastěji

CC BY-NC-ND 4.0

Množina a slovník: proč je in tak rychlé?

```
>>> seznam = list(range(1_000_000))
>>> mnozina = set(seznam)
>>> 999_999 in seznam                    # projde milion prvků: O(n)
True
>>> 999_999 in mnozina                   # jeden pohled do přihrádky: O(1)
True
```

- Minule: množina i slovník si z prvku (klíče) spočítají číslo a podle něj ho uloží do **přihrádky**; dnes se podíváme dovnitř
- Klíčová ingredience je **hashovací funkce**

CC BY-NC-ND 4.0

Množina a slovník: proč je in tak rychlé?

Hashovací funkce

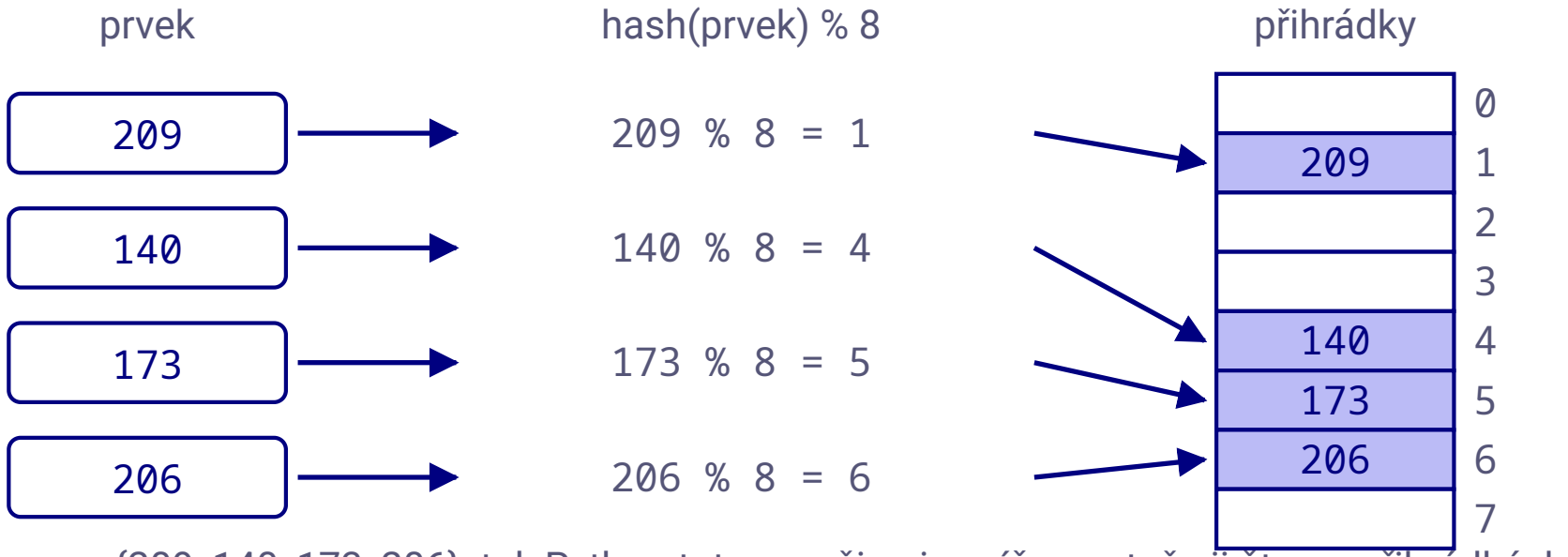
Funkce, která z libovolné hodnoty (čísla, textu, n-tice) rychle spočítá **číslo**. Stejná hodnota dá vždy stejné číslo, různé hodnoty skoro vždy různá čísla.

```
print(hash(205))                # 205: u malých celých čísel je hash číslo samo
print(hash("krádež"))           # velké číslo, třeba -4877261459163364063; při dalším spuštění jiné
print(hash((2026, 5)))          # i n-tice má hash, proto může být v množině
hash([1, 2])                    # TypeError: unhashable type: 'list'
```

- Číslo přihrádky: `hash(hodnota) % pocet_prihradek`; s osmi přihrádkami jde 205 do přihrádky 5
- Proto musí být prvek množiny a klíč slovníku **neměnný**: kdyby se po uložení změnil, ležel by ve špatné přihrádce

CC BY-NC-ND 4.0

Množina a slovník: proč je in tak rychlé?

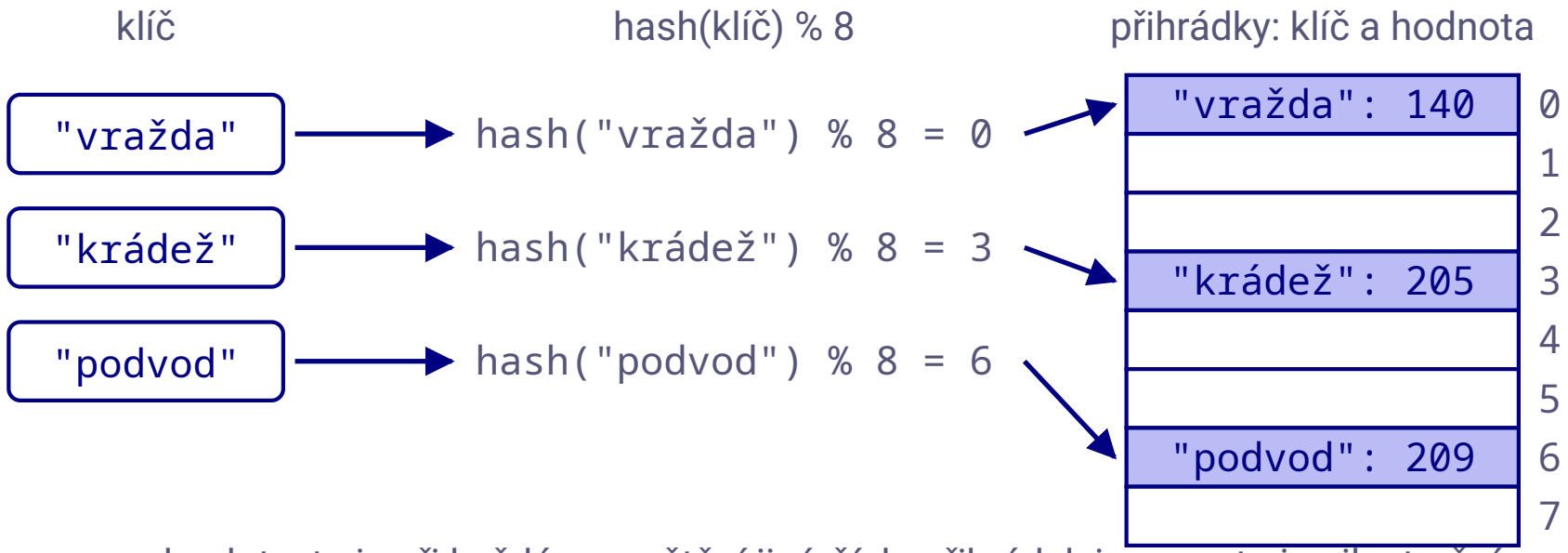


{209, 140, 173, 206}: tak Python tuto množinu i vypíše, protože ji čte po přihrádkách

- Vložení: hash, zbytek po dělení počtem přihrádek, uložit: **stejný počet kroků**, ať je prvků kolik chce
- Dotaz `205 in mnozina`: 205 % 8 = 5, v přihrádce 5 leží 173, tedy **není**; nic dalšího neprocházíme
- Výpis jde po přihrádkách: proto minule `{205, 173, 205, 140}` vypsalo `{140, 205, 173}`

CC BY-NC-ND 4.0

Slovník uvnitř: stejné přihrádky, navíc hodnota



hash textu je při každém spuštění jiný, čísla přihrádek jsou proto jen ilustrační

- Hash se počítá jen z **klíče**, do přihrádky jde klíč i hodnota; proto musí být neměnný jen klíč
- `tresty["podvod"]`: hash klíče, přihrádka, porovnat klíč, vrátit hodnotu; stejně kroků pro tři i milion dvojic
- Pořadí vložení si slovník pamatuje zvlášť, proto se vypisuje v pořadí, v jakém jsme dvojice přidali

CC BY-NC-ND 4.0

Složitost základních operací: list, set, dict

Operace	`list`	`set`	`dict`
je tam? `x in s`	$O(n)$	$O(1)$	$O(1)$ podle klíče
přidání `.append(x)`, `.add(x)`, `d[k] = v`	$O(1)$	$O(1)$	$O(1)$
odebrání `.remove(x)`, `del d[k]`	$O(n)$	$O(1)$	$O(1)$
prvek podle indexu `s[i]`	$O(1)$	nelze	nelze
hodnota podle klíče `d[k]`	nelze	nelze	$O(1)$
projít vše cyklem `for`	$O(n)$	$O(n)$	$O(n)$

- Pravidlo z minula má teď důvod: mnoho dotazů „je tam?“ znamená **množinu**, hledání podle klíče **slovník**, pořadí a indexy **seznam**

CC BY-NC-ND 4.0

Co za rychlost platíme: nevýhody množiny a slovníku

- **Bez indexu a pořadí**: v množině nejde „třetí prvek“ ani výřez; slovník drží pořadí vložení, ale `d[2]` je klíč 2, ne třetí dvojice
- **Jen neměnné prvky a klíče**: seznam ani slovník do množiny nevložíte a klíčem být nemůže; místo seznamu n-tice
- **Víc paměti**: přihrádek musí být víc než prvků, jinak přibývají kolize; část paměti je vždy prázdná
- $O(1)$ **platí v průměru**: při smůle (mnoho kolizí) se hledá déle; Python proto přihrádky včas zvětší a prvky přerozdělí, což samo stojí $O(n)$
- **Žádné „nejbližší větší“**: pro dotaz „první paragraf nad 200“ je lepší seřazený seznam a půlení

CC BY-NC-ND 4.0

P vs. NP: problém za milion dolarů

P a NP

P: problémy, které umíme v polynomiálním čase **vyřešit**. **NP**: problémy, u kterých umíme navržené řešení v polynomiálním čase alespoň **ověřit**.

- Sudoku: vyplnit mřížku je těžké, zkontrolovat vyplněnou snadné. Rozvrh bez kolizí, rozdělení majetku napůl, nejkratší trasa přes sto měst: totéž
- Co umíme vyřešit, umíme i ověřit, takže každý problém z P je i v NP; otázka za milion dolarů zní, **zda to platí i naopak**
- Problém milénia Clayova institutu: otevřený od roku 1971; většina informatiků věří, že P ≠ NP, dokázat to nikdo neumí

CC BY-NC-ND 4.0

Proč na P vs. NP záleží i právníkům

- Kdyby P = NP, šlo by rychle **najít** cokoliv, co umíme rychle **ověřit**: každý důkaz, každý rozvrh, každé heslo
- Dnešní šifrování (bankovnictví, datové schránky, elektronické podpisy) stojí na tom, že rozložit velké číslo na prvočísla neumíme rychle, ale ověřit součin ano
- Řada praktických úloh je „NP-těžká“: rozvrh jednacích síní, rozdělení majetku, trasa doručovatele. Řešíme je přibližně, nebo jen pro malá n
- Když program „nestíhá“, nemusí to být chyba programátora: některé problémy rychlé řešení možná **nemají**

CC BY-NC-ND 4.0

Děkujeme za pozornost

CC BY-NC-ND 4.0