

Algoritmizace

1. Co je algoritmizace a proč je důležitá?

- Definice algoritmu – jakákoli posloupnost kroků vedoucí k řešení úlohy.
- Příklad obecného algoritmu (např. recept na čaj nebo třídění knih v knihovně).
- Proč je dobré znát základy algoritmizace v programování – efektivita, škálovatelnost, debugging.

Co je algoritmizace?

Algoritmizace je proces navrhování a tvorby algoritmů – systematických postupů k řešení problémů. V programování znamená schopnost rozložit problém na logické kroky, které vedou k požadovanému výsledku.

Definice:

↗ *Algoritmus je konečná, jednoznačná a systematická posloupnost kroků, která vede k vyřešení daného problému.*

Podívejme se na **jednoduchý algoritmus na uvaření čaje** ve formě vývojového diagramu:

- 1 Start
- 2 Zahřej vodu
- 3 Pokud je voda dostatečně horká? 🔥
 - Ano → Pokračuj
 - Ne → Čekej
- 4 Vlož sáček čaje do hrnku
- 5 Zalij horkou vodou
- 6 Počkej několik minut
- 7 Vyjměte sáček a zamíchej
- 8 Konec ☕

Proč je dobré se věnovat algoritmizaci při učení programování?

- ✓ Lepší pochopení programování – Programování není jen o psaní kódu, ale hlavně o logickém myšlení a řešení problémů.
- ✓ Schopnost efektivního řešení problémů – Pokud umíš dobře navrhnout algoritmus, umíš rozložit složitý problém na menší části a řešit ho efektivněji.
- ✓ Výkon a optimalizace – Správná algoritmizace pomáhá psát efektivní programy, které běží rychleji a spotřebovávají méně paměti.
- ✓ Lepší debugging – Když víš, jak algoritmus funguje, snáze odhalíš chyby ve svém kódu.
- ✓ Příprava na technické pohovory – Algoritmy jsou klíčové v hiring procesech na pozice vývojářů.

Obecný algoritmus – co to znamená?

💡 Obecný algoritmus je takový algoritmus, který lze aplikovat na různé problémy bez ohledu na konkrétní data nebo jazyk programování. Jedná se o univerzální postup řešení úlohy, který můžeme přizpůsobit různým situacím.

Vlastnosti obecného algoritmu

Každý správný algoritmus by měl mít tyto klíčové vlastnosti:

- ✓ Konečnost – Algoritmus musí mít konečný počet kroků.
- ✓ Jednoznačnost – Každý krok musí být přesně definovaný.
- ✓ Vstup – Algoritmus přijímá nějaké vstupy (čísla, seznamy, texty atd.).
- ✓ Výstup – Algoritmus produkuje výstup (řešení úlohy).
- ✓ Efektivita – Měl by být optimalizovaný pro co nejlepší výkon.

Příklad obecného algoritmu – hledání největšího čísla v seznamu

1 Začátek

2 Vezmi první číslo jako největší

3 Procházej všechna čísla v seznamu:

Pokud najdeš číslo větší než aktuální největší, aktualizuj hodnotu

4 Po skončení seznamu máš největší číslo

5 Konec

Tento algoritmus funguje pro jakýkoliv seznam čísel a není závislý na konkrétní implementaci v programovacím jazyce.

Proč jsou obecné algoritmy užitečné?

- ✓ Mohou se znova použít – místo psaní kódu od nuly lze stejný princip použít pro více problémů.
- ✓ Pomáhají při přemýšlení o řešení – programátoři nemusí znát syntaxi všech jazyků, ale pokud rozumí algoritmům, zvládnou psát kód kdekoli.
- ✓ Jsou optimalizovatelné – lze je vylepšit a přizpůsobit různým potřebám.

Tohle je základní myšlenka algoritmizace – hledání správných postupů k řešení problémů. 🚀

2. Základní stavební kameny algoritmů

- Logické operátory (and, or, not) – jejich význam a použití.
- Podmínky ve větvení kódu (if, elif, else) – ukázka rozhodovacího stromu.
- Smyčky (for, while) – rozdíl mezi for a while, kdy použít který.

💡 Krátké cvičení:

- Napište program, který rozhodne, zda číslo je sudé nebo liché.
- Vytvořte smyčku, která projde čísla od 1 do 10 a vypíše jen ta, která jsou dělitelná třemi.

Logické operátory (and, or, not) – význam a použití

Logické operátory se v programování používají k vyhodnocování **pravdivostních podmínek** (true/false). Jsou základním stavebním kamenem **rozhodování** v kódu, například v podmírkách (if) nebo smyčkách (while).

Vizualizace pomocí pravdivostní tabulky

A	B	A and B	A or B	not A
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

1 and (logické A)

- ◆ Vrátí True, pokud oba operandy jsou True.
- ◆ Jinak vrátí False.

📌 Příklad:

```
python
x = True
y = False
print(x and y) # Výstup: False
```

📌 Reálný scénář:

- Chceme, aby uživatel měl přístup pouze tehdy, pokud je přihlášený a má správná oprávnění.

```
python

is_logged_in = True
has_permission = False

if is_logged_in and has_permission:
    print("Máš přístup!")
else:
    print("Přístup zamítnut.")
```

📌 Výstup: "Přístup zamítnut." (protože `has_permission` je `False`).

2 or (logické NEBO)

- ◆ Vrátí `True`, pokud alespoň jeden operand je `True`.
- ◆ Vrátí `False`, pokud oba jsou `False`.

📌 Příklad:

```
python

x = True
y = False
print(x or y) # Výstup: True
```

📌 Reálný scénář:

- Pokud má uživatel **bud' oprávnění, nebo je administrátor**, povolit akci.

```
python

has_permission = False
is_admin = True

if has_permission or is_admin:
    print("Máš přístup!")
else:
    print("Přístup zamítnut.")
```

📌 Výstup: "Máš přístup!" (protože `is_admin` je `True`).

3 not (logická negace, OPAK)

◆ Převrátí hodnotu:

- `not True` → `False`
- `not False` → `True`

📌 Příklad:

```
python
x = True
print(not x) # Výstup: False
```

Copy Edit

📌 Reálný scénář:

- Pokud uživatel NENÍ přihlášený, zobrazit login.

```
python
is_logged_in = False

if not is_logged_in:
    print("Prosím, přihlas se.")
```

Copy Edit

📌 Výstup: "Prosím, přihlas se." (protože `not False` je `True`).

Praktické cvičení pro juniorky

💡 Úkol 1:

Napište program, který ověří, zda má uživatel **buď oprávnění, nebo je administrátor**, a na základě toho rozhodne o přístupu.

💡 Úkol 2:

Doplňte podmínku do kódu tak, aby program vypsal `"Jsi teenager"`, pokud je uživateli mezi 13 a 19 lety.

```
python
age = int(input("Zadej svůj věk: "))

if ____:
    print("Jsi teenager.")
else:
    print("Nejsi teenager.")
```

Copy Edit

(Doplň chybějící logický výraz!)



Shrnutí

- ✓ and – Obě podmínky musí být True, jinak výsledek False.
- ✓ or – Stačí, když je alespoň jedna podmínka True.
- ✓ not – Převrací logickou hodnotu.

Tyto operátory jsou klíčové pro rozhodování v kódu a tvoří základ podmínek a řízení toku programu.

💡 Úkol: Přístupový systém s negací

Napište program, který ověří, zda uživatel NENÍ zablokován a může se přihlásit.

✓ Podmínky:

- Pokud je uživatel blokovaný (`is_blocked = True`), zobrazí se: "Přístup zamítnut!"
- Pokud není blokovaný (`is_blocked = False`), zobrazí se: "Vítej, můžeš se přihlásit!"

📝 Doplň chybějící část kódu:

```
python                                ⌂ Copy ⌂ Edit

is_blocked = True # Změň na False a sleduj, co se stane

if _____:
    print("Vítej, můžeš se přihlásit!")
else:
    print("Přístup zamítnut!")
```

❓ Otázka na zamýšlení pro juniorky:

Co by se stalo, kdybychom nepoužili `not` a místo toho přepisovali podmínu jinak?

Python podporuje **idiomatický zápis**, který je čistší a čitelnější. Místo `is_blocked == False` se běžně používá **negace (not)**, tedy:

 **Doporučené a idiomatické řešení:**

```
python
```

```
if not is_blocked:  
    print("Vítej, můžeš se přihlásit!")  
else:  
    print("Přístup zamítнут!")
```

Copy Edit

 Proč je `not is_blocked` lepší než `is_blocked == False`?

- ✓ Čitelnější – Snadno pochopíš, že negujeme stav.
- ✓ Idiomatický Python – V Pythonu je `not is_blocked` považováno za "Pythonic" styl.
- ✓ Minimalistický kód – Méně psaní, lepší srozumitelnost.

Porovnání dvou verzí

 **Méně doporučená verze (dlouhá)**

```
python
```

```
if is_blocked == False:
```

Copy Edit

- Funguje, ale je zbytečně ukecaná.

 **Doporučená verze (krátká a čistá)**

```
python
```

```
if not is_blocked:
```

Copy Edit

- Kratší, čitelnější, více odpovídá standardům Pythonu.

Takže řešení je správné, ale lepší je použít negaci `not`. 😊

Proč se v podmírkách používají dvě rovná se (==)?

V programování jedno `=` a dvě `==` mají zcela odlišné významy:

Jedno `=` znamená přiřazení hodnoty

- Používáme ho k uložení hodnoty do proměnné.
- Například:

```
python  
  
age = 25 # Přiřazení hodnoty 25 do proměnné age
```

◆ Říkáme: "Proměnná `age` teď obsahuje hodnotu `25`."

Dvě `==` znamenají porovnání hodnot

- Používáme je v podmírkách ke kontrole, zda dvě hodnoty jsou si rovny.
- Například:

```
python  
  
if age == 25:  
    print("Je ti 25 let!")
```

◆ Ptáme se: "Je `age` rovno `25`?"

Co by se stalo, kdybychom omylem použili = místo == ?

💡 Pokud bychom omylem napsali:

```
python
```

Copy Edit

```
if age = 25: # Chyba! ❌
```

🚀 Python by zobrazil chybu, protože = je přiřazení, nikoli porovnání.

Příklad rozdílu mezi = a ==

```
python
```

Copy Edit

```
x = 10      # Přiřazení hodnoty 10 do proměnné x
y = 20      # Přiřazení hodnoty 20 do proměnné y

if x == y: # Porovnání, zda x a y jsou stejné
    print("Hodnoty jsou stejné.")
else:
    print("Hodnoty jsou různé.") # Výstup: "Hodnoty jsou různé."
```

Shrnutí

Operátor	Význam	Příklad	Výstup
=	Přiřazení hodnoty	x = 5	x nyní obsahuje hodnotu 5
==	Porovnání (rovnost)	x == 5	True, pokud x je rovno 5, jinak False

Úkol: Kontrola věku pro registraci

Máme systém, kde se uživatel **NESMÍ** registrovat, pokud je mu méně než 18 let.

Podmínky:

- Pokud je uživateli méně než 18 let (`age < 18`), zobrazit: "Registrace zamítnuta!"
- Pokud je uživateli **18 a více**, zobrazit: "Registrace úspěšná!"

Doplň chybějící část kódu tak, aby využívala `not`:

python

 Copy  Edit

```
age = int(input("Zadej svůj věk: "))

if _____ (age < 18):
    print("Registrace úspěšná!")
else:
    print("Registrace zamítnuta!")
```

Použití přímého porovnání (lepší varianta):

python

 Copy  Edit

```
if age >= 18:
    print("Registrace úspěšná!")
else:
    print("Registrace zamítnuta!")
```

✓ Lepší čitelnost – rovnou vidíme, že kontrolujeme, jestli je věk **18 a více**.

✓ Čistší syntaxe, žádná zbytečná negace.

Kdy použít `not` a kdy přímé porovnání?

- `not` se hodí, když chceme negovat logickou hodnotu (`True` / `False`).
- Když pracujeme s čísly, často je lepší použít přímé porovnání, protože je přehlednější.

?

Kdy použít `not` a kdy přímé porovnání?

- `not` se hodí, když chceme negovat logickou hodnotu (`True` / `False`).
 - Když pracujeme s čísly, často je lepší použít přímé porovnání, protože je přehlednější.
-

Shrnutí

- ✓ `if age >= 18:` je lepší varianta, protože je přímá a intuitivní.
- ✓ `if not (age < 18):` sice funguje, ale je méně čitelná.

Podmínky ve větvení kódu (`if`, `elif`, `else`) – Rozhodovací strom

Podmínky v programování umožňují větvení kódu na základě splnění určitých kritérií.

V Pythonu používáme `if`, `elif` a `else` k rozhodování, jaký kód se vykoná.

1 `if` – základní podmínka

- Pokud podmínka platí (`True`), provede se blok kódu.
- Pokud neplatí (`False`), kód se přeskočí.

💡 Příklad:

```
python
age = 20
if age >= 18:
    print("Můžeš volit!")
```

✓ Výstup: "Můžeš volit!" (protože `age` je 20, což je více než 18).

2 if + else – dvě možnosti (ano/ne)

- `if` vyhodnotí první podmítku.
- `else` se vykoná, pokud první podmínka neplatí.

📌 Příklad:

```
python Copy Edit
age = 16

if age >= 18:
    print("Můžeš volit!")
else:
    print("Nemůžeš volit.")
```

✓ Výstup: "Nemůžeš volit." (protože `age` je 16, což je méně než 18).

3 if + elif + else – více podmínek

- `elif` znamená „jinak pokud“.
- Umožňuje přidat další podmínky mezi `if` a `else`.

📌 Příklad:

```
python Copy Edit
age = 16

if age >= 18:
    print("Můžeš volit!")
elif age >= 15:
    print("Můžeš řídit kolo na silnici.")
else:
    print("Můžeš jezdit na kole jen po chodníku.")
```

✓ Výstup: "Můžeš řídit kolo na silnici."

● Program vyhodnotí první pravdivou podmítku a přeskočí ostatní.

Praktické cvičení pro juniorky:

 Úkol: Rozšiřte tento program tak, aby přidal další pravidlo:

- Pokud je uživateli **21 a více**, vypíše "Můžeš pít alkohol v USA!"
-

Shrnutí

- ✓ `if` – základní podmínka.
- ✓ `elif` – další možnosti, když `if` neplatí.
- ✓ `else` – pokud neplatí žádná z předchozích podmínek.

 Podmínky `if` a `elif` se vyhodnocují postupně shora dolů.

 Pokud věk splňuje první pravdivou podmínsku, zbývající `elif` se už **nezkoumají**.

Co se stane při `age = 16` ?

- `if age >= 18:` →  (16 není ≥ 18)
- `elif age >= 15:` →  (16 je ≥ 15) → Tato podmínka je splněna, takže se provede a všechny další `elif` se přeskočí!
- `elif age >= 21:` →  (nikdy se k ní nedostane)
- `else:` →  (také se přeskočí, protože `elif age >= 15` bylo True)

Je potřeba **nejprve** otestovat `age >= 21`, protože je to nejvyšší věková hranice.

 Správná verze kódu:

```
python

age = 16

if age >= 21:
    print("Můžeš pit alkohol v USA!")
elif age >= 18:
    print("Můžeš volit!")
elif age >= 15:
    print("Můžeš řidit kolo na silnici.")
else:
    print("Můžeš jezdit na kole jen po chodníku.")
```

 Copy  Edit

✓ Tímto způsobem se správně otestují nejvyšší věkové hranice jako první

✓ Podmínky se vyhodnocují logicky a postupně

 **Úkol: Klasifikace známek pomocí `elif`**

Napište program, který klasifikuje známky studentů na základě číselného hodnocení.

 Podmínky:

- 90 a více → "Výborně (A)"
- 80–89 → "Chvalitebně (B)"
- 70–79 → "Dobře (C)"
- 60–69 → "Dostatečně (D)"
- Méně než 60 → "Nedostatečně (F)"

✖ Doplň chybějící části v kódu:

```
python Copy Edit

score = int(input("Zadej svůj počet bodů: "))

if score >= 90:
    print("Výborně (A)")
elif _____:
    print("Chvalitebně (B)")
elif _____:
    print("Dobře (C)")
elif _____:
    print("Dostatečně (D)")
else:
    print("Nedostatečně (F)")
```

Bonusová otázka pro juniorky:

- ◆ Co se stane, pokud někdo zadá skóre vyšší než 100 nebo záporné číslo?
- ◆ Jak byste program rozšířili, aby takové hodnoty odmítl?

✓ Správná verze kódu:

```
python Copy Edit

score = int(input("Zadej svůj počet bodů: "))

if score >= 90:
    print("Výborně (A)")
elif score >= 80:
    print("Chvalitebně (B)")
elif score >= 70:
    print("Dobře (C)")
elif score >= 60:
    print("Dostatečně (D)")
else:
    print("Nedostatečně (F)")
```

Vysvětlení:

- ✓ Podmínky se vyhodnocují shora dolů.
- ✓ Pokud `score >= 90`, provede se první `if` a zbytek se přeskočí.
- ✓ Pokud `score` není alespoň 90, program pokračuje dál a kontroluje další `elif`.
- ✓ Zajištěno správné pořadí podmínek, takže každý rozsah bodů má správnou klasifikaci.

Bonus: Ošetření neplatných vstupů

Pokud chcete zajistit, že skóre je v platném rozsahu 0–100, přidejte kontrolu:

```
python Copy Edit
score = int(input("Zadej svůj počet bodů: "))

if score < 0 or score > 100:
    print("Chyba: Zadej platné skóre mezi 0 a 100.")
elif score >= 90:
    print("Výborně (A)")
elif score >= 80:
    print("Chvalitebně (B)")
elif score >= 70:
    print("Dobře (C)")
elif score >= 60:
    print("Dostatečně (D)")
else:
    print("Nedostatečně (F)")
```

- ✓ Ochrana proti nesmyslným hodnotám (např. -5 nebo 105).

Odpovědi na bonusové otázky

◆ Co se stane, pokud někdo zadá skóre vyšší než 100 nebo záporné číslo?

Pokud v původním programu zadáme:

- Hodnotu vyšší než 100 (např. 120) → Program ji vyhodnotí jako "Výborně (A)" , což není správné, protože skóre by mělo být maximálně 100.
- Záporné číslo (např. -10) → Program ho vyhodnotí jako "Nedostatečně (F)" , což taky není správné, protože skóre nemůže být záporné.

 **Problém:** Program neověruje, zda je vstup v platném rozsahu (0–100), a proto dává nesprávné výsledky.

◆ Jak byste program rozšířili, aby takové hodnoty odmítl?

Musíme přidat kontrolu platnosti vstupu:

- 1 Pokud `score` není v rozsahu 0–100, zobrazí se chybová zpráva.
- 2 Program nebude pokračovat v klasifikaci známky, pokud vstup není platný.

Úkol: Převod teploty do slovního hodnocení

Napište program, který převede teplotu ve stupních Celsia na slovní hodnocení počasí.

Podmínky:

- Více než 30°C → "Horko!"
- 20 až 30°C → "Příjemné počasí."
- 10 až 19°C → "Chladno."
- 0 až 9°C → "Zima."
- Méně než 0°C → "Mrazivo!"

✖ Doplň chybějící části v kódu:

```
python Copy Edit

temperature = int(input("Zadej teplotu ve °C: "))

if _____:
    print("Horko!")
elif _____:
    print("Příjemné počasí.")
elif _____:
    print("Chladno.")
elif _____:
    print("Zima.")
else:
    print("Mrazivo!")
```

❓ Bonusové otázky:

- 1 Co se stane, pokud někdo zadá nesmyslnou hodnotu, jako třeba "abc" ?
- 2 Jak byste program upravili, aby odmítal nečíselné vstupy a vyzval uživatele k zadání správného čísla?

✓ Správné řešení úkolu: Převod teploty na slovní hodnocení

```
python Copy Edit

temperature = int(input("Zadej teplotu ve °C: "))

if temperature > 30:
    print("Horko!")
elif temperature >= 20:
    print("Příjemné počasí.")
elif temperature >= 10:
    print("Chladno.")
elif temperature >= 0:
    print("Zima.")
else:
    print("Mrazivo!")
```

Vysvětlení kódu:

- ✓ Nejprve testujeme nejvyšší teploty ($> 30^{\circ}\text{C}$), aby byla logika správná.
- ✓ `elif temperature >= 20`: zajistí, že $20\text{--}30^{\circ}\text{C}$ bude hodnoceno jako "Příjemné počasí."
- ✓ `elif temperature >= 10`: pokrývá $10\text{--}19^{\circ}\text{C}$
- ✓ `elif temperature >= 0`: zachytí $0\text{--}9^{\circ}\text{C}$
- ✓ Pokud teplota klesne pod 0°C , spustí se `else` a vypíše "Mrazivo!"

Opravená verze s kontrolou nečíselného vstupu

Pokud uživatel zadá text (např. "abc"), program spadne na chybě `ValueError`.

Tuto chybu můžeme ošetřit použitím `try-except`:

```
python Copy Edit  
  
try:  
    temperature = int(input("Zadej teplotu ve °C: "))  
  
    if temperature > 30:  
        print("Horko!")  
    elif temperature >= 20:  
        print("Příjemné počasí.")  
    elif temperature >= 10:  
        print("Chladno.")  
    elif temperature >= 0:  
        print("Zima.")  
    else:  
        print("Mrazivo!")  
  
except ValueError:  
    print("Chyba: Prosím zadej platné číslo.")
```

- ✓ Tato verze zajistí, že program nespadne, pokud uživatel zadá neplatný vstup!
- ✓ Pokud zadá text místo čísla, zobrazí se chybová hláška a program pokračuje dál.

Shrnutí

- 1 Základní verze → Funguje, ale neumí zpracovat nečíselné vstupy.
- 2 Vylepšená verze s `try-except` → Zajišťuje, že uživatel zadá pouze číslo, jinak dostane chybovou zprávu.

◆ Shrnutí: Podmínky ve větvení kódu (`if`, `elif`, `else`)

- ✓ `if` – Používáme, pokud chceme ověřit jednu podmínku. Pokud je podmínka `True`, provede se kód uvnitř bloku.
- ✓ `elif` – Používáme, pokud chceme přidat další podmínky. Funguje jako „jinak pokud“, tedy pokud první `if` neplatí, ale jiná podmínka ano.
- ✓ `else` – Používáme pro všechny ostatní případy, pokud žádná předchozí podmínka nebyla splněna.

◆ Nejčastější chyby při používání `if-elif-else`

✗ Špatné pořadí podmínek

- Například pokud kontrolujeme `elif temperature >= 30` až po `elif temperature >= 10`, tak se nikdy nevyhodnotí vyšší teploty správně.
 - ✓ Řešení: Nejprve testuj **nejvyšší hodnoty!**

✗ Zbytečné používání `== True` nebo `== False`

- Například místo `if is_active == True:` je lepší psát přímo `if is_active:`
 - ✓ Řešení: Používej čisté podmínky bez zbytečných porovnání.

✗ Zapomenutí `else` pro nečekané případy

- ✓ Řešení: Pokud chceme mít jistotu, že program vždy něco vypíše, přidej `else`.

🚀 Bonusové otázky pro ověření pochopení

1 Jaký bude výstup tohoto kódu?

```
python                                ⌂ Copy ⌂ Edit

age = 18

if age > 18:
    print("Jsi dospělý.")
elif age == 18:
    print("Právě jsi dosáhl dospělosti!")
else:
    print("Ještě nejsi dospělý.")
```

➔ Otázka: Která podmínka se vyhodnotí jako `True`?

2 Jaká je chyba v tomto kódu?

python

Copy Edit

```
if temperature >= 0:  
    print("Není mrazivo.")  
elif temperature > 20:  
    print("Je teplo.")  
else:  
    print("Je mrazivo.")
```

→ Otázka: Co se stane, pokud `temperature = 25`? Vypíše se "Je teplo." ?

3 Jak bys opravil tento kód, aby správně určoval, jestli číslo je kladné, záporné nebo nula?

python

Copy Edit

```
number = int(input("Zadej číslo: "))  
  
if number > 0:  
    print("Číslo je kladné.")  
elif number < 0:  
    print("Číslo je záporné.")  
elif number == 0:  
    print("Číslo je nula.")
```

→ Otázka: Funguje tento kód správně? Lze ho napsat jinak?

Odpovědi na bonusové otázky

1 Jaký bude výstup tohoto kódu?

python

 Copy  Edit

```
age = 18

if age > 18:
    print("Jsi dospělý.")
elif age == 18:
    print("Právě jsi dosáhl dospělosti!")
else:
    print("Ještě nejsi dospělý.")
```

Správná odpověď:

 `age == 18`, takže se vykoná druhý blok kódu:

 Výstup: "Právě jsi dosáhl dospělosti!"

✓ Vysvětlení:

- `if age > 18:` → False (18 není větší než 18)
- `elif age == 18:` → True, takže se tato podmínka vykoná.
- `else` se přeskočí, protože `elif` byla pravdivá.

2 Jaká je chyba v tomto kódu?

python

 Copy  Edit

```
if temperature >= 0:
    print("Není mrazivo.")
elif temperature > 20:
    print("Je teplo.")
else:
    print("Je mrazivo.")
```

Chyba:

Podmínka `elif temperature > 20:` se nikdy neprovede, pokud je `temperature` nad 20!

✓ Vysvětlení:

- `if temperature >= 0:` zachytí všechny hodnoty větší než nebo rovné 0, takže když bude `temperature = 25`, vypíše se "Není mrazivo." a program skončí.
- Podmínka `elif temperature > 20:` se nikdy nevyhodnotí, protože už byla splněna `if` podmínka.

Opravená verze:

```
python Copy Edit

temperature = int(input("Zadej teplotu: "))

if temperature > 20:
    print("Je teplo.")
elif temperature >= 0:
    print("Není mrazivo.")
else:
    print("Je mrazivo.")
```

- ✓ Nejprve kontrolujeme `temperature > 20`, aby se správně vyhodnotilo, zda je teplo.

Jak bys opravil tento kód, aby správně určoval, jestli číslo je kladné, záporné nebo nula?

```
python Copy Edit

number = int(input("Zadej číslo: "))

if number > 0:
    print("Číslo je kladné.")
elif number < 0:
    print("Číslo je záporné.")
elif number == 0:
    print("Číslo je nula.")
```

Chyba v kódu:

-  Poslední `elif` je zbytečný, protože pokud číslo není ani kladné, ani záporné, musí to být nula.

Opravená verze:

```
python Copy Edit

number = int(input("Zadej číslo: "))

if number > 0:
    print("Číslo je kladné.")
elif number < 0:
    print("Číslo je záporné.")
else:
    print("Číslo je nula.") # Použijeme `else`, protože zbyla jen jedna možnost.
```

- ✓ `else` je zde čistší řešení, protože zachytí všechny zbývající případy.

📌 Shrnutí:

- 1 Podmínky se vyhodnocují shora dolů – pokud něco platí, další `elif` se už nezkoumá.
- 2 Nezapomínej na správné pořadí podmínek – jinak se některé nikdy neprojdou.
- 3 Používej `else` tam, kde už jiná možnost není možná – šetří to řádky kódu.

◆ Smyčky v Pythonu: `for` vs. `while` – rozdíl a použití

Smyčky umožňují opakování provádění kódu. V Pythonu máme dva hlavní typy smyček:

- 1 `for` smyčka – Používá se, když víme, kolikrát se má kód opakovat.
 - 2 `while` smyčka – Používá se, když nevíme přesný počet opakování, ale záleží na podmínce.
-

1 `for` smyčka – iterace přes sekvenci

✓ Používá se, když dopředu známe počet opakování nebo chceme procházet kolekce (seznamy, řetězce, `range`).

📌 Příklad 1: Výpis čísel od 1 do 5

```
python
```

Copy Edit

```
for i in range(1, 6):
    print(i)
```

◆ Výstup:

Copy Edit

```
1
2
3
4
5
```

📌 Jak to funguje?

- `range(1, 6)` generuje čísla 1, 2, 3, 4, 5.
- Smyčka "projde každé číslo" a vypíše ho.

📌 Příklad 2: Iterace přes seznam

```
python  
  
fruits = ["jablko", "banán", "třešeň"]  
  
for fruit in fruits:  
    print(fruit)
```

[Copy](#) [Edit](#)

◆ Výstup:

```
nginx  
  
jablko  
banán  
třešeň
```

[Copy](#) [Edit](#)

📌 Jak to funguje?

- Smyčka prochází seznam a do proměnné `fruit` ukládá jednotlivé hodnoty.

2 while smyčka – běží, dokud platí podmínka

- ✓ Používá se, když nevíme předem, kolikrát se má kód spustit.
- ✓ Podmínka se kontroluje při každé iteraci – když už neplatí, smyčka se zastaví.

📌 Příklad 1: Počítání do 5 pomocí while

```
python  
  
x = 1  
while x <= 5:  
    print(x)  
    x += 1 # Zvýšení x o 1
```

[Copy](#) [Edit](#)

◆ Výstup:

```
1  
2  
3  
4  
5
```

[Copy](#) [Edit](#)

📌 Jak to funguje?

- `while x <= 5` říká: "Pokud `x` je menší nebo rovno 5, pokračuj".
 - Každé opakování se `x` zvětší o 1 → Jakmile `x = 6`, podmínka už neplatí, takže smyčka končí.
-

📌 Příklad 2: Opakované dotazování, dokud není zadán správný vstup

```
python Copy Edit
password = ""

while password != "tajneheslo":
    password = input("Zadej heslo: ")

print("Přihlášení úspěšné!")
```

◆ Jak to funguje?

- Smyčka se bude opakovat, dokud uživatel **nezadá správné heslo**.
- Jakmile zadá `"tajneheslo"`, smyčka skončí a vypíše se `"Přihlášení úspěšné!"`.

◆ Kdy použít `for` a kdy `while`?

Smyčka	Kdy ji použít?	Příklad
<code>for</code>	Když víme počet opakování	Projít seznam, generovat čísla s <code>range()</code>
<code>while</code>	Když nevíme předem počet opakování	Čekání na správný vstup, nekonečné smyčky

✓ Pokud víme počet iterací → `for`

✓ Pokud se podmínka mění během běhu programu → `while`

🚀 Praktické cvičení pro juniorky

💡 Úkol 1 (pro `for`)

Napište smyčku, která vypíše všechna sudá čísla od 2 do 20.

💡 Úkol 2 (pro `while`)

Napište program, který se ptá uživatele na PIN **dokud nezadá správný kód "1234"**.

💡 Úkol 1 (pro `for`): Výpis sudých čísel od 2 do 20

python

Copy Edit

```
for number in range(2, 21, 2):
    print(number)
```

◆ Výstup:

Copy Edit

```
2
4
6
8
10
12
14
16
18
20
```

📌 Vysvětlení:

- `range(2, 21, 2)` generuje čísla od 2 do 20 s krokem 2 → sudá čísla.
- Smyčka `for` projde všechna čísla a vypíše je.

💡 Úkol 2 (pro `while`): Zadání PINu

python

Copy Edit

```
pin = ""

while pin != "1234":
    pin = input("Zadej PIN: ")

print("Správný PIN! Přístup povolen.")
```

◆ Výstup: (Záleží na tom, co uživatel zadá.)

```
yaml
```

Copy Edit

```
Zadej PIN: 0000
Zadej PIN: 4321
Zadej PIN: 1234
Správný PIN! Přístup povolen.
```

Vysvětlení:

- Smyčka se opakuje, dokud uživatel nezadá "1234".
- Pokud zadá špatný PIN, smyčka se znovu ptá.
- Jakmile je správně "1234", program skončí a zobrazí se potvrzení.

Bonusová otázka pro vylepšení kódu:

Jak byste zajistili, aby uživatel mohl zadat PIN maximálně 3x, než se program ukončí?

Vysvětlení `range(2, 21, 2)`

V Pythonu funkce `range(start, stop, step)` generuje posloupnost čísel podle tří parametrů:

```
python
```

Copy Edit

```
range(start, stop, step)
```

- `start` → Počáteční hodnota (zde 2).
- `stop` → Konečná hodnota (ale ne včetně!) (zde 21, takže skončí u 20).
- `step` → Krok, jakým se čísla zvětšují (zde 2, takže přeskočí lichá čísla).

Jak funguje `range(2, 21, 2)`?

Generuje čísla:

```
2, 4, 6, 8, 10, 12, 14, 16, 18, 20
```

Copy Edit

Proč končí u 20?

- Hodnota `stop = 21` znamená, že čísla budou generována do 21, ale bez 21 → tedy maximálně 20.
- Python nikdy nezahrnuje `stop` hodnotu do výsledného seznamu.

◆ Další příklady `range()` pro pochopení

📌 Výpis čísel od 1 do 5:

```
python
for i in range(1, 6):
    print(i)
```

Copy Edit

→ Výstup: 1, 2, 3, 4, 5 (protože `stop=6` se už nevypíše).

📌 Výpis každého třetího čísla mezi 10 a 30:

```
python
for i in range(10, 31, 3):
    print(i)
```

Copy Edit

→ Výstup: 10, 13, 16, 19, 22, 25, 28

📌 Počítání pozpátku:

```
python
for i in range(10, 0, -1):
    print(i)
```

Copy Edit

→ Výstup: 10, 9, 8, 7, 6, 5, 4, 3, 2, 1



🚀 Shrnutí

- ✓ `range(start, stop, step)` generuje čísla od `start` do `stop-1` s krokem `step`.
- ✓ Pokud nenastavíme `step`, výchozí hodnota je 1 (`range(1, 6) → 1, 2, 3, 4, 5`).
- ✓ `step` může být záporný, což znamená počítání zpět (`range(10, 0, -1)`).

✓ Alternativní řešení s `if` a `%` (modulus)

python

Copy Edit

```
for number in range(1, 21): # Projdeme všechna čísla od 1 do 20
    if number % 2 == 0: # Pokud je číslo dělitelné 2 beze zbytku (sudé)
        print(number)
```

◆ Výstup:

```
2
4
6
8
10
12
14
16
18
20
```

Copy Edit

📌 Jak to funguje?

- 1 Smyčka `for` projde všechna čísla od 1 do 20.
- 2 Každé číslo se ověří podmínkou `if number % 2 == 0`:
 - `%` (modulus) vrací zbytek po dělení.
 - Pokud `number % 2 == 0`, znamená to, že číslo je sudé (např. `4 % 2 = 0`).
 - Pokud by výsledek byl 1, číslo je liché (např. `5 % 2 = 1`).
- 3 Pokud podmínka platí, číslo se vypíše.

Co kdybychom chtěli místo sudých čísel vypsat lichá čísla?

Stačí změnit podmítku:

```
python

for number in range(1, 21):
    if number % 2 != 0: # Pokud číslo není dělitelné 2
        print(number)
```

 Copy  Edit

→ Výstup: 1, 3, 5, 7, 9, 11, 13, 15, 17, 19

Shrnutí

- ✓ Použití `range(2, 21, 2)` je efektivnější, protože přímo generuje sudá čísla.
- ✓ Použití `if number % 2 == 0` je univerzálnější, protože můžeme filtrovat i jiná čísla (lichá, dělitelná 3, 5...).
- ✓ Modulus `%` je užitečný v mnoha situacích, např. při kontrole sudých/lichých čísel, dělitelnosti, cyklických operacích.

ÚKOL S PINEM:

Správná verze s maximálně 5 pokusy

```
python

pocet_opakovani = 0
pin = ""

while pin != "1234" and pocet_opakovani < 5:
    pin = input("Zadej PIN: ")
    pocet_opakovani += 1 # Zvýšení počtu pokusů

if pin == "1234":
    print("Správný PIN! Přístup povolen.")
else:
    print("Přístup zamítnut. Překročen maximální počet pokusů.")
```

 Copy  Edit

Jak to funguje?

✓ Podmínka ve `while` smyčce:

- Smyčka se opakuje, pokud:
 - PIN je nesprávný (`pin != "1234"`)
 - Počet pokusů je menší než 5 (`pocet_opakovani < 5`)

✓ Zvýšení počtu pokusů (`pocet_opakovani += 1`)

- Každé zadání PINu zvýší počet pokusů.
- Pokud uživatel zadá správný PIN, smyčka se zastaví.

✓ Po smyčce program rozhodne:

- Pokud byl zadáný PIN `"1234"` → "Přístup povolen"
- Pokud uživatel překročil 5 pokusů → "Přístup zamítnut."

Úkol: Hra "Uhádni slovo" s omezeným počtem pokusů

Napište program, kde musí uživatel uhodnout tajné slovo.

- Má 5 pokusů.
- Pokud slovo uhodne, program okamžitě skončí s gratulací.
- Pokud neuhodne, program mu oznamí zbývající počet pokusů.
- Pokud vyčerpá všechny pokusy, program zobrazí "Konec hry. Správné slovo bylo: ..."

Doplňte chybějící části kódu:

```
python Copy Edit
tajne_slovo = "python"
pokusy = 0
max_pokusy = 5

while _____ and _____:
    tip = input("Zadej tajné slovo: ")
    _____ # Zvýšení počtu pokusů

    if _____:
        print("Gratulace! Uhodl jsi správné slovo.")
        _____ # Ukončení smyčky
    else:
        print("Špatně. Zbývajici pokusy:", _____)

if _____:
    print("Konec hry. Správné slovo bylo:", tajne_slovo)
```

Bonusová otázka:

Jak byste kód upravili, aby program ignoroval velká a malá písmena (`"Python"`, `"PYTHON"`, `"python"` jsou správné)?

Správné řešení úkolu: Hra "Uhádni slovo"

```
python Copy Edit

tajne_slovo = "python"
pokusy = 0
max_pokusy = 5

while pokusy < max_pokusy and tajne_slovo != tip:
    tip = input("Zadej tajné slovo: ")
    pokusy += 1 # Zvýšení počtu pokusů

    if tip == tajne_slovo:
        print("Gratulace! Uhodl jsi správné slovo.")
        break # Ukončení smyčky
    else:
        print("Špatně. Zbývající pokusy:", max_pokusy - pokusy)

if tip != tajne_slovo:
    print("Konec hry. Správné slovo bylo:", tajne_slovo)
```

Vysvětlení kódu

✓ Podmínka `while`

- Smyčka běží, pokud `pokusy` jsou menší než 5 a uživatel zatím neuhodl slovo.

✓ Zvýšení počtu pokusů (`pokusy += 1`)

- Každý neúspěšný pokus přidává +1.

✓ Kontrola správného slova (`if tip == tajne_slovo`)

- Pokud uživatel zadá `"python"`, program vypíše `"Gratulace!"` a použije `break`, aby ukončil smyčku.

✓ Pokud uživatel vyčerpá pokusy (`if tip != tajne_slovo`)

- Program oznámí konec hry a zobrazí správné slovo.

✓ Opravená verze s inicializací tip na začátku

```
python Copy Edit

tajne_slovo = "python"
pokusy = 0
max_pokusy = 5
tip = "" # Přidání inicializace proměnné

while pokusy < max_pokusy and tajne_slovo != tip:
    tip = input("Zadej tajné slovo: ").lower() # Převedení na malá písmena
    pokusy += 1

    if tip == tajne_slovo:
        print("Gratulace! Uhodl jsi správné slovo.")
        break
    else:
        print("Špatně. Zbývající pokusy:", max_pokusy - pokusy)

if tip != tajne_slovo:
    print("Konec hry. Správné slovo bylo:", tajne_slovo)
```

✓ Verze s .upper()

```
python Copy Edit

tajne_slovo = "PYTHON" # Tajné slovo uložené ve velkých písmenech
pokusy = 0
max_pokusy = 5
tip = "" # Inicializace

while pokusy < max_pokusy and tajne_slovo != tip:
    tip = input("Zadej tajné slovo: ").upper() # Převede vstup na VELKÁ písmena
    pokusy += 1

    if tip == tajne_slovo:
        print("Gratulace! Uhodl jsi správné slovo.")
        break
    else:
        print("Špatně. Zbývající pokusy:", max_pokusy - pokusy)

if tip != tajne_slovo:
    print("Konec hry. Správné slovo bylo:", tajne_slovo)
```

📌 Jaký je rozdíl mezi `.lower()` a `.upper()`?

Funkce	Co dělá?	Příklad vstupu "Python"	Výstup
<code>.lower()</code>	Převede text na malá písmena	<code>"Python".lower()</code>	<code>"python"</code>
<code>.upper()</code>	Převede text na VELKÁ písmena	<code>"Python".upper()</code>	<code>"PYTHON"</code>

- ✓ Obě verze jsou správné, jen záleží, jak uložíme `tajne_slovo`.
- ✓ Pokud použiješ `.upper()`, je lepší mít `tajne_slovo = "PYTHON"`.
- ✓ Pokud použiješ `.lower()`, je lepší mít `tajne_slovo = "python"`.

3. Matematické operace v algoritmech

- Absolutní hodnota – co znamená a proč je užitečná.
- Medián – vysvětlení + jednoduchý algoritmus pro jeho nalezení.

💡 Krátké cvičení:

- Vypočítejte absolutní hodnotu libovolného čísla bez použití vestavěné funkce.
- Najděte medián v seznamu [7, 2, 5, 3, 9].

📌 Absolutní hodnota – co znamená a proč je užitečná?

Absolutní hodnota čísla je vzdálenost tohoto čísla od nuly na číselné ose.

📌 Matematicky:

- Pokud je číslo **kladné nebo nula**, absolutní hodnota se **nemění**.
- Pokud je číslo **záporné**, absolutní hodnota se **převede na kladné**.

✓ Příklady absolutní hodnoty

Číslo	Absolutní hodnota
5	5
-5	5
0	0
-12	12
8	8

Jak na absolutní hodnotu v Pythonu?

 V Pythonu se používá vestavěná funkce `abs()`

python

 Copy  Edit

```
print(abs(5))    # Výstup: 5
print(abs(-5))   # Výstup: 5
print(abs(0))    # Výstup: 0
```

Proč je absolutní hodnota užitečná?

 1 Měření vzdálenosti mezi čísly

- Když porovnáváme dvě hodnoty, rozdíl je **vždy kladný**, a proto se používá absolutní hodnota.

python

 Copy  Edit

```
vzdalenost = abs(-10 - 5) # Výstup: 15
```

 2 Výpočet rozdílu mezi dvěma body (např. souřadnice)

- Např. chceme zjistit, jak daleko jsou dvě souřadnice na ose X.

python

 Copy  Edit

```
x1 = -3
x2 = 7
vzdalenost = abs(x2 - x1) # Výstup: 10
print("Vzdálenost:", vzdalenost)
```

 3 Práce s fyzikálními veličinami (např. rychlosť, teplota, síla)

- V reálném světě záporné hodnoty často nemají smysl (např. rychlosť nemôže byť záporná).

 4 Řešení podmínek (např. tolerance odchylky)

- Když porovnáváme dvě čísla s určitou tolerancí, **absolutní hodnota nám pomůže zjistit rozdíl**.

python

 Copy  Edit

```
skutecna_teplota = 22
ocekavana_teplota = 20

if abs(skutecna_teplota - ocekavana_teplota) <= 2:
    print("Teplota je v přijatelné toleranci.")
```

🚀 Praktické cvičení pro juniorky

💡 Úkol 1:

Napište program, který přijme číslo od uživatele a vypíše jeho absolutní hodnotu.

💡 Úkol 2:

Vytvořte program, který porovná dvě čísla a zjistí jejich **vzdálenost** na číselné ose.

📌 Může `abs()` pracovat s `float` číslami?

Ano! Funkce `abs()` funguje nejen pro celá čísla (`int`), ale také pro desetinná čísla (`float`).

✅ Příklad s `float`:

python

Copy Edit

```
print(abs(-3.14)) # Výstup: 3.14
print(abs(7.89)) # Výstup: 7.89
print(abs(-0.001)) # Výstup: 0.001
```

✓ Funguje úplně stejně jako pro `int` – převede záporné číslo na kladné.

📌 Jaký je datový typ výsledku `abs()`?

- 📌 Pokud vstupní hodnota je `int`, vrátí `int`.
- 📌 Pokud vstupní hodnota je `float`, vrátí `float`.

python

Copy Edit

```
print(type(abs(-5))) # Výstup: <class 'int'>
print(type(abs(-3.14))) # Výstup: <class 'float'>
```

◆ Kdy se `abs()` s `float` čísla používá v praxi?

✓ 1 Výpočet vzdáleností ve fyzice a matematice

- Pokud počítáme rozdíl mezi dvěma souřadnicemi, výsledná vzdálenost by měla být vždy kladná.

```
python Copy Edit  
  
x1 = 2.5  
x2 = -4.8  
vzdalenost = abs(x2 - x1) # Výstup: 7.3  
print("Vzdálenost:", vzdalenost)
```

✓ 2 Odchylky a tolerance v inženýrství a vědě

- Pokud měříme například **teplotní odchylku**, chceme pracovat jen s kladnými hodnotami.

```
python Copy Edit  
  
teplota_ideal = 20.0  
teplota_real = 18.7  
  
odchylka = abs(teplota_real - teplota_ideal)  
print("Odchylka je:", odchylka) # Výstup: 1.3
```

✓ 3 Výpočty v ekonomii a statistice

- Např. výpočet rozdílu mezi dvěma kurzy měn nebo změnou ceny akcií.

```
python Copy Edit  
  
stara_cena = 250.75  
nova_cena = 245.30  
  
zmena_ceny = abs(nova_cena - stara_cena)  
print("Změna ceny:", zmena_ceny) # Výstup: 5.45
```

📌 Shrnutí

- ✓ Funkce `abs()` funguje pro `int` i `float` čísla.
- ✓ Výstupní hodnota má stejný datový typ jako vstup (`int` → `int`, `float` → `float`).
- ✓ Využívá se ve fyzice, matematice, statistice, ekonomii a programování.

✓ Opravená verze programu s `round()`

```
python Copy Edit
vstup = float(input("Napiš číslo: "))
print("Absolutní hodnota:", round(abs(vstup), 4))
```

→ Výstup:

```
yaml Copy Edit
Napiš číslo: -3.14159265
Absolutní hodnota: 3.1416
```

- ✓ `round(abs(vstup), 4)` zajistí, že výstup bude mít 4 čísla za desetinnou čárkou.

✗ Alternativní způsob: Formátovaný výstup (`f-string`)

Pokud chceme hezčí výstup, můžeme použít `f-string`:

```
python Copy Edit
vstup = float(input("Napiš číslo: "))
print(f"Absolutní hodnota: {abs(vstup):.4f}")
```

→ Výstup je stejný jako předtím, ale v hezčím formátu:

```
yaml Copy Edit
Napiš číslo: -2.718281
Absolutní hodnota: 2.7183
```

- ✓ `:.4f` znamená 4 desetinná místa.
- ✓ Výstup vždy zobrazí 4 desetinná místa, i kdyby číslo mělo méně.

◆ Jaký je rozdíl mezi `round()` a `f-string`?

Metoda	Co dělá?	Příklad <code>abs(-3.14159265)</code>
<code>round(x, 4)</code>	Zaokrouhlí číslo, ale může odstranit nulová desetinná místa	<code>3.1416</code>
<code>f"{x:.4f}"</code>	Vždy zobrazí přesně 4 desetinná místa (i když jsou nuly)	<code>3.1416</code> nebo <code>2.0000</code>

💡 Úkol: Rozdíl teplot pomocí absolutní hodnoty

Napište program, který:

- ✓ Požádá uživatele o dvě teploty (např. ranní a odpolední).
- ✓ Vypočítá absolutní rozdíl mezi těmito teplotami.
- ✓ Výstup zobrazí zaokrouhlený na 2 desetinná místa.

📌 Doplňte chybějící části kódu:

```
python Copy Edit

teplota_rano = float(input("Zadej ranní teplotu: "))
teplota_odpoledne = float(input("Zadej odpolední teplotu: "))

rozdil = _____ # Spočítej absolutní rozdíl mezi teplotami

print(f"Teplotní rozdíl je: {_____}") # Zaokrouhlí na 2 desetinná místa
```

🚀 Bonusová otázka:

Jak byste kód upravili, aby program fungoval i pro teploty pod nulou?

✓ Správné řešení úkolu: Rozdíl teplot pomocí absolutní hodnoty

```
python Copy Edit

teplota_rano = float(input("Zadej ranní teplotu: "))
teplota_odpoledne = float(input("Zadej odpolední teplotu: "))

rozdil = abs(teplota_odpoledne - teplota_rano) # Spočítáme absolutní rozdíl

print(f"Teplotní rozdíl je: {rozdil:.2f}") # Zaokrouhlení na 2 desetinná místa
```

📌 Vysvětlení kódu

- ✓ Použití `abs()` zajistí, že rozdíl je vždy kladný, bez ohledu na to, zda bylo ráno tepleji nebo chladněji.
- ✓ Zaokrouhlení `:.2f` znamená, že výstup bude na dvě desetinná místa.

◆ Příklad běhu programu:

yaml

Copy Edit

```
Zadej ranní teplotu: -5.3
Zadej odpolední teplotu: 10.7
Teplotní rozdíl je: 16.00
```

→ Funguje i pro teploty pod nulou! *🔥

📌 Medián – vysvětlení a algoritmus pro jeho nalezení

◆ Co je medián?

- ✓ Medián je prostřední hodnota v setříděné řadě čísel.
- ✓ Pokud je počet prvků lichý, medián je prostřední číslo.
- ✓ Pokud je počet prvků sudý, medián je průměr dvou prostředních čísel.

✓ Příklad mediánu v praxi

◆ Pokud máme lichý počet čísel:

- 📌 Data: [3, 1, 7, 5, 9]
- 📌 Seřadíme: [1, 3, 5, 7, 9]
- 📌 Medián je prostřední číslo: 5

◆ Pokud máme sudý počet čísel:

- 📌 Data: [4, 1, 6, 2, 8, 10]
- 📌 Seřadíme: [1, 2, 4, 6, 8, 10]
- 📌 Dvě prostřední čísla jsou 4 a 6
- 📌 Medián = $(4 + 6) / 2 = 5.0$



✓ Algoritmus pro nalezení mediánu

- 1 Seřad' seznam čísel.
- 2 Zjisti délku seznamu (n).
- 3 Pokud n je liché, medián je prostřední prvek.
- 4 Pokud n je sudé, medián je průměr dvou prostředních prvků.

✓ Jednoduchý Python kód pro nalezení mediánu

python

Copy Edit

```
def najdi_median(cisla):
    cisla.sort()  # Seřadíme čísla vzestupně
    n = len(cisla)  # Délka seznamu
    stred = n // 2  # Index prostředního prvku

    if n % 2 == 1:  # Pokud je počet čísel lichý
        return cisla[stred]
    else:  # Pokud je počet čísel sudý
        return (cisla[stred - 1] + cisla[stred]) / 2

# Testovací data
print(najdi_median([3, 1, 7, 5, 9]))      # Výstup: 5
print(najdi_median([4, 1, 6, 2, 8, 10]))    # Výstup: 5.0
```

◆ Jak tento kód funguje?

- ✓ Seřadí čísla (`sort()`)
- ✓ Najde prostřední index (`n // 2`)
- ✓ Vybere prostřední číslo nebo vypočítá průměr dvou prostředních čísel

🚀 Praktický úkol pro juniorky

💡 Úkol:

- 1 Napište program, který přijme libovolný počet čísel od uživatele a najde jejich medián.
- 2 Ověřte, že funguje správně pro sudý i lichý počet čísel.

Ano! `sort()` je vestavěná metoda v Pythonu, která se používá k seřazení seznamu (`list`) čísel nebo řetězců vzestupně (ve výchozím nastavení).

- ✓ Nepotřebuje žádné knihovny – je součástí standardního Pythonu.
 - ✓ Funguje přímo na seznamu (`list`) a mění jeho obsah na místě.
-

📌 Jak funguje `sort()`?

◆ Seřazení číselného seznamu

```
python                                ⌂ Copy ⌂ Edit

cisla = [5, 2, 8, 1, 9]
cisla.sort() # Seřazení vzestupně
print(cisla)
```

→ Výstup: [1, 2, 5, 8, 9]

◆ Seřazení seznamu textů

```
python                                ⌂ Copy ⌂ Edit

slova = ["Banán", "Jablko", "Hruška"]
slova.sort()
print(slova)
```

→ Výstup: ['Banán', 'Hruška', 'Jablko'] (řadí podle abecedy)

◆ Co když chci řadit sestupně?

Použijeme `sort(reverse=True)`:

```
python                                ⌂ Copy ⌂ Edit

cisla = [5, 2, 8, 1, 9]
cisla.sort(reverse=True) # Seřazení sestupně
print(cisla)
```

→ Výstup: [9, 8, 5, 2, 1] 

◆ `sorted()` vs. `sort()` – jaký je rozdíl?

- ✓ `sort()` mění seznam přímo (`in-place sorting`).
- ✓ `sorted()` vrací nový seznam, aniž by měnil původní.

python

Copy Edit

```
cisla = [5, 2, 8, 1, 9]

# Použití sorted() – vrátí nový seznam
novy_seznam = sorted(cisla)
print(novy_seznam) # [1, 2, 5, 8, 9]
print(cisla)       # [5, 2, 8, 1, 9] (původní seznam zůstává stejný)

# Použití sort() – změní seznam přímo
cisla.sort()
print(cisla) # [1, 2, 5, 8, 9]
```

📌 Shrnutí

Funkce	Co dělá?	Změní původní seznam?
<code>list.sort()</code>	Seřadí seznam na místě	Ano
<code>sorted(list)</code>	Vrátí nový seřazený seznam	Ne

- ✓ Pokud chceš zachovat původní seznam, použij `sorted()`.
- ✓ Pokud chceš přímo změnit seznam, použij `sort()`.

✓ Výsledné řešení: Nalezení mediánu z čísel zadaných uživatelem

```
python Copy Edit  
  
def najdi_median():  
    # Požádáme uživatele o čísla a rozdělime je podle mezery  
    cisla = list(map(float, input("Zadej čísla oddělená mezerou: ").split()))  
  
    # Seřazení seznamu čísel  
    cisla.sort()  
  
    # Délka seznamu  
    n = len(cisla)  
    stred = n // 2 # Index prostředního prvku  
  
    # Pokud je počet čísel lichý  
    if n % 2 == 1:  
        median = cisla[stred]  
    else: # Pokud je počet čísel sudý  
        median = (cisla[stred - 1] + cisla[stred]) / 2  
  
    print(f"Medián je: {median:.2f}") # Výstup zaokrouhlený na 2 desetinná místa  
  
# Spuštění programu  
najdi_median()
```



📌 Jak tento kód funguje?

- ✓ Uživatel zadává čísla jako text, která se rozdělí (`split()`).
- ✓ Každé číslo se převeze na `float (map(float, ...))`, takže funguje i pro desetinná čísla.
- ✓ Seznam se seřadí (`sort()`).
- ✓ Najdeme prostřední index (`n // 2`).
- ✓ Lichý počet čísel → Medián je prostřední číslo.
- ✓ Sudý počet čísel → Medián je průměr dvou prostředních čísel.
- ✓ Výstup je zaokrouhlený na 2 desetinná místa.

Lichý počet čísel:

yaml

Copy Edit

Zadej čísla oddělená mezerou: 4 1 7 9 3
Medián je: 4.00

✓ Seřazený seznam: [1, 3, 4, 7, 9] → prostřední číslo je 4.

Sudý počet čísel:

yaml

Copy Edit

Zadej čísla oddělená mezerou: 8 2 6 10 4 12
Medián je: 6.00

✓ Seřazený seznam: [2, 4, 6, 8, 10, 12] → medián = $(6+8)/2 = 7.00$.

✓ Rozbor kódu krok za krokem

python

Copy Edit

```
def najdi_median():
```

- ✓ Vytváříme funkci `najdi_median()`, která bude obsahovat celý výpočet mediánu.
- ✓ Funkce se volá až na konci kódu (`najdi_median()`), takže se spustí, když ji zavoláme.

python

 Copy  Edit

```
cisla = list(map(float, input("Zadej čísla oddělená mezerou: ").split()))
```

📌 Co tady děláme?

1 `input("Zadej čísla oddělená mezerou: ")`

- Požádáme uživatele, aby zadal čísla oddělená mezerou (např. `"4 1 7 9 3"`).

2 `.split()`

- Rozdělí text (string) podle mezer na jednotlivé části.
- `"4 1 7 9 3".split() → ["4", "1", "7", "9", "3"]` (seznam stringů).

3 `map(float, ...)`

- Každou položku převedeme na číslo (`float`).
- `map(float, ["4", "1", "7", "9", "3"]) → [4.0, 1.0, 7.0, 9.0, 3.0]`.

4 `list(map(...))`

- Výsledný **map objekt** převedeme na **seznam** (`list`).
- Výsledný seznam čísel: `[4.0, 1.0, 7.0, 9.0, 3.0]`.

```
cisla.sort()
```

Copy Edit

✓ Seřadíme seznam čísel vzestupně.

✓ Seznam [4.0, 1.0, 7.0, 9.0, 3.0] se po seřazení stane [1.0, 3.0, 4.0, 7.0, 9.0].

```
python
```

Copy Edit

```
n = len(cisla)  
stred = n // 2
```

📌 Co tady děláme?

1 len(cisla) → Určuje, kolik čísel je v seznamu.

- Například: [1.0, 3.0, 4.0, 7.0, 9.0] → n = 5.

2 n // 2 → Vypočítáme index prostředního čísla.

- n = 5 → stred = 5 // 2 = 2 (index 2 v seznamu).
- Pozor! V Pythonu indexování začíná od 0, takže třetí číslo (index 2) je 4.0.

```
python
```

Copy Edit

```
if n % 2 == 1:  
    median = cisla[stred]
```

📌 Co se tady děje?

✓ Kontrolujeme, zda n je liché (% 2 == 1).

✓ Pokud ano, medián je prostřední číslo.

◆ Příklad:

Seznam [1.0, 3.0, 4.0, 7.0, 9.0] (n = 5, lichý počet)

✓ stred = 2, takže medián = cisla[2] = 4.0.

```
python
```

Copy Edit

```
else:  
    median = (cisla[stred - 1] + cisla[stred]) / 2
```

👉 Co se tady děje?

- ✓ Pokud `n` je sudé, spočítáme průměr dvou prostředních čísel.

◆ Příklad:

Seznam `[1.0, 2.0, 4.0, 6.0, 8.0, 10.0]` ($n = 6$, sudý počet)

✓ `stred = 6 // 2 = 3` (index `3`) → číslo `6.0`.

✓ Druhé prostřední číslo je `cisla[stred - 1]`, což je `4.0`.

✓ Medián je `(4.0 + 6.0) / 2 = 5.0`.

```
python
```

Copy Edit

```
najdi_median()
```

👉 Spuštění programu

- ✓ Zavolá naši funkci, aby se program spustil.

◆ Shrnutí, co program dělá

- 1 Uživatel zadá čísla oddělená mezerou.
- 2 Program převede vstup na seznam čísel (`float`).
- 3 Seřadí čísla (`sort()`).
- 4 Najde prostřední číslo (pokud je lichý počet), nebo spočítá průměr dvou prostředních (pokud je sudý počet).
- 5 Vypíše medián zaokrouhlený na 2 desetinná místa.

📌 Proč používám `map(float, input().split())`?

- ✓ Efektivnější zápis – místo psaní `for` cyklu převede všechna čísla najednou.
- ✓ Kompaktní a přehledné – méně řádků kódu.
- ✓ Přímá konverze vstupu – zajistí, že všechna čísla jsou `float`, i když uživatel zadá desetinná čísla.

◆ Jak by vypadal kód bez `map()`?

python

Copy Edit

```
vstup = input("Zadej čísla oddělená mezerou: ").split()
cisla = [] # Prázdný seznam

for prvek in vstup:
    cisla.append(float(prvek)) # Převedeme každý prvek na float
```

📌 Tento kód dělá to samé, ale je delší!

- `input().split()` rozdělí text na seznam stringů.
- `for` cyklus projde každý prvek a převede ho  `float`.

✓ Jak funguje `map(float, input().split())`?

python

Copy Edit

```
cisla = list(map(float, input("Zadej čísla oddělená mezerou: ").split()))
```

- ✓ `input("Zadej čísla: ")` → uživatel zadá např. "4 1 7 9 3"
- ✓ `.split()` → rozdělí text na seznam: ["4", "1", "7", "9", "3"]
- ✓ `map(float, ...)` → převede každý prvek na `float` → [4.0, 1.0, 7.0, 9.0, 3.0]
- ✓ `list(...)` → převede map objekt zpět na seznam (`list`)

◆ Shrnutí: Výhody `map()`

Metoda	Výhody	Nevýhody
✓ <code>map(float, input().split())</code>	Krátký, efektivní zápis	Pro začátečníky méně intuitivní
✗ <code>for</code> cyklus	Srozumitelnější pro začátečníky	Delší kód, více proměnných

- ✓ Pokud rozumíš `map()`, je to kratší a efektivnější varianta.
- ✓ Pokud chceš kód čitelnější pro začátečníky, můžeš použít `for` smyčku.

✗ `map()` – co to je?

`map()` je vestavěná funkce v Pythonu, která aplikuje funkci na každý prvek v iterovatelném objektu (například seznamu) a vrátí nový objekt `map`.

✗ `Syntaxe map()`

python

Copy Edit

```
map(funkce, iterovatelný_objekt)
```

- ✗ `funkce` – Funkce, která se aplikuje na každý prvek seznamu.
- ✗ `iterovatelný_objekt` – Seznam (`list`), n-tice (`tuple`), množina (`set`), nebo jiný iterovatelný objekt.
- ✓ `map()` vrátí `map objekt`, který je nutné převést na `list()`, pokud chceme výsledek jako seznam.

✓ Příklad 1: Použití `map()` pro převod stringů na `int`

python

Copy Edit

```
cisla = ["1", "2", "3", "4"]
cisla_int = list(map(int, cisla)) # Převede každý string na int
print(cisla_int) # Výstup: [1, 2, 3, 4]
```

- ✓ `map(int, cisla)` převede všechny hodnoty v seznamu `cisla` na `int`.
- ✓ `list(...)` zajistí, že výsledek je seznam (`list`).

✓ Příklad 2: Použití `map()` s `float` (pro náš příklad s mediánem)

python

Copy Edit

```
vstup = input("Zadej čísla oddělená mezerou: ") # Např. "3.5 2.1 4.8"
cisla = list(map(float, vstup.split())) # Každé číslo převedeme na float
print(cisla)
```

- ✓ `input().split()` rozdělí řetězec podle mezer.
- ✓ `map(float, vstup.split())` převede každý prvek na `float`.
- ✓ Výstup: [3.5, 2.1, 4.8]

✓ Příklad 3: Použití `map()` s vlastní funkcí

python

Copy Edit

```
def dvojnasobek(x):
    return x * 2

cisla = [1, 2, 3, 4, 5]
dvojnasobky = list(map(dvojnasobek, cisla))
print(dvojnasobky) # Výstup: [2, 4, 6, 8, 10]
```

- ✓ Každé číslo v seznamu se vynásobí 2 pomocí funkce `dvojnasobek(x)`.

📌 Kdy použít `map()` místo `for` smyčky?

Metoda	Výhody	Nevýhody
✓ <code>map()</code>	Kratší a efektivnější kód	Méně intuitivní pro začátečníky
✗ <code>for</code> cyklus	Snadněji pochopitelné	Delší kód

✓ Pokud potřebuješ rychle upravit seznam, `map()` je super volba.

✗ Pokud potřebuješ složitější logiku, `for` cyklus může být lepší.

📌 Co je `split()`?

`split()` je metoda v Pythonu, která rozdělí řetězec (`string`) na seznam (`list`) podle zadaného oddělovače (separátora).

Ve výchozím nastavení odděluje mezery, ale můžeme použít i jiný znak (např. tečku `.`).

✓ Základní syntaxe `split()`

```
python                                ⌂ Copy ⌂ Edit

text = "Toto je věta"
slova = text.split()  # Rozdělí podle mezer (výchozí nastavení)
print(slova)
```

✓ Výstup: `['Toto', 'je', 'věta']`

✗ Pokud `split()` nemá argument, rozdělí řetězec podle mezer.

✓ Jak rozdělit řetězec podle jiného znaku? (např. tečky .)

Pokud chceme rozdělit podle teček, musíme `split(".")` předat jako argument tečku (.).

```
python
```

Copy Edit

```
text = "www.google.com"
casti = text.split(".")
print(casti)
```

✓ Výstup: ['www', 'google', 'com']

✓ Příklad: Rozdělení IP adresy podle teček

```
python
```

Copy Edit

```
ip_adresa = "192.168.1.1"
casti = ip_adresa.split(".")
print(casti)
```

✓ Výstup: ['192', '168', '1', '1']

✓ Jak rozdělit podle více znaků najednou?

Například, pokud chceme rozdělit podle mezery i tečky současně, `split()` to neumí přímo, ale můžeme použít `re.split()` z knihovny `re` (regular expressions).

```
python
```

Copy Edit

```
import re

text = "www. google .com"
casti = re.split(r'\s|\.', text) # Rozdělí podle mezer i teček
print(casti)
```

✓ Výstup: ['www', '', 'google', '', 'com']

✗ `\s` znamená mezera a `|` značí "nebo" → tedy dělíme podle mezery nebo tečky.

✓ Jak odstranit prázdné položky po rozdělení?

Pokud `split()` vytvoří prázdné položky, můžeme je odstranit pomocí list comprehension:

```
python
```

Copy Edit

```
casti = [cast for cast in text.split(".") if cast] # Odstraní prázdné položky
```

✗ Shrnutí

Použití	Syntaxe	Výstup
Základní <code>split()</code> (mezera)	<code>"Toto je věta".split()</code>	<code>['Toto', 'je', 'věta']</code>
Rozdělení podle tečky	<code>"www.google.com".split(".")</code>	<code>['www', 'google', 'com']</code>
Rozdělení podle více znaků (mezera + tečka)	<code>'re.split(r'\s</code> <code>if cast]</code>	<code>., text)'</code>
Odstranění prázdných položek	<code>[cast for cast in text.split(".")</code> <code>if cast]</code>	Bez prázdných položek

4. Datové struktury – Seznam, Fronta, Zásobník

- **Seznam (list)** – lineární kolekce dat.
- **Zásobník (stack)** – LIFO (Last In, First Out).
- **Fronta (queue)** – FIFO (First In, First Out).

💡 Krátké cvičení:

- Implementujte zásobník v Pythonu pomocí seznamu (`push`, `pop`).
- Vytvořte simulaci fronty v obchodě.

5. Třídění a efektivita algoritmů

- BubbleSort vs. QuickSort – proč BubbleSort není efektivní, ale proč je dobré ho znát.
- Porovnání složitostí ($O(n^2)$ vs. $O(n \log n)$).
- Základy efektivity algoritmů – co je Big O notation.

💡 Krátké cvičení:

- Ručně projděte BubbleSort na malém seznamu [4, 2, 7, 1].
- Vysvětlete, proč je QuickSort rychlejší než BubbleSort na velkých datech.

6. Rekurze – když se funkce volá sama

- Kdy a proč použít rekurzi.
- Výpočet faktoriálu nebo Fibonacciho posloupnosti pomocí rekurze.



Krátké cvičení:

- Implementujte rekurzivní funkci pro faktoriál.
- Napište program, který spočítá Fibonacciho číslo rekurzivně i iterativně.

7. Debugging a testování algoritmů

- Jak efektivně hledat chyby v algoritmech.
- Použití print() vs. debugger.
- Automatizované testování pomocí assert.



Krátké cvičení:

- Najděte chybu v jednoduchém algoritmu a opravte ho.