Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

# Výučba časovej zložitosti programov na strednej škole

Záverečná práca doplňujúceho pedagogického štúdia

2020
Mgr. Jozef Rajník

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

# Výučba časovej zložitosti programov na strednej škole

Záverečná práca doplňujúceho pedagogického štúdia

| | |
|---|---|
| Študijný program: | 7656 Učiteľstvo akademických predmetov |
| Študijný odbor:: | 2508 Informatika |
| Pracovisko: | Katedra didaktiky matematiky, fyziky a informatiky |
| Školiteľ: | RNDr. Michal Winczer, PhD. |

Bratislava, 2020
Mgr. Jozef Rajník

17982357

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

# ZADANIE ZÁVEREČNEJ PRÁCE

**Meno a priezvisko študenta:** Mgr. Jozef Rajník
**Študijný program:** doplňujúce pedagogické štúdium informatiky (Doplňujúce pedagogické štúdium, iné N st., denná forma)
**Študijný odbor:** učiteľstvo a pedagogické vedy
**Typ záverečnej práce:** Záverečná práca doplňujúceho pedagogického štúdia
**Jazyk záverečnej práce:** slovenský
**Sekundárny jazyk:** anglický

**Názov:** Výučba časovej zložitosti programov na strednej škole
*Teaching time complexity of programs in upper secondary education*

**Anotácia:** Aktuálne cieľové požiadavky na vedomosti a zručnosti maturantov z informatiky vyžadujú, aby maturanti vedeli intuitívne uvažovať o zložitosti algoritmu. Cieľom práce je nájsť vhodné programátorské úlohy, pri ktorých možno dobre uvažovať o časovej zložitosti, a vyvinúť k nim metodiku.

**Cieľ:** Navrhnúť a otestovať spôsob výučby časovej zložitosti u maturantov.

**Vedúci:** RNDr. Michal Winczer, PhD.
**Katedra:** FMFI.KDMFI - Katedra didaktiky matematiky, fyziky a informatiky
**Vedúci katedry:** prof. RNDr. Ivan Kalaš, PhD.

**Dátum zadania:** 28.10.2019

**Dátum schválenia:** 28.10.2019 doc. RNDr. Zuzana Kubincová, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

**Poďakovanie:**

Chcel by som sa poďakovať môjmu školiteľovi Michalovi Winczerovi za cenné rady pri tvorbe tejto práce a za upozornenia na možné úskalia pri testovaní.

Zároveň chcem využiť tento priestor na poďakovanie sa celej Katedre didaktiky matematiky, fyziky a informatiky, ako aj fakulte, vďaka ktorým som mohol absolvovať doplňujúce pedagogické štúdium z informatiky. Štúdium bolo pre mňa veľmi prínosné. Som rád, že som mal tú česť absolvovať predmety so špičkovými pedagógmi a odborníkmi vo svojich oblastiach. Taktiež si odnášam spomienky na mojich kolegov študentov, s ktorými som mohol viesť zaujímavé diskusie naprieč rôznymi predmetmi a vypočuť si ich názory a pohľady na rôzne aspekty učiteľstva. Chcem taktiež prejaviť svoju vďaku za všetku ústretovosť, s ktorou som sa počas štúdia stretol, a porozumenie voči mnohým ťažkostiam, ktoré sú s doplňujúcim pedagogickým štúdiom späté.

Ďakujem vedeniu Gymnázia Grösslingova, na ktorom učím, a všetkým mojim kolegom za sprevádzanie a pomoc. Nedá mi nespomenúť, že som vďačný za to, že som počas pandémie COVID-19 mohol učiť. Otvorilo mi to oči a naučil som vďaka tomu mnohé zručnosti, ktoré, verím, využijem aj pri obnovení kontaktnej výučby. Moja vďaka patrí aj učiteľkám, u ktorých som si mohol vyskúšať metodiku z tejto práce, za výbornú komunikáciu a spoluprácu.

V neposlednom rade ďakujem za podporu mojej rodine, známym, priateľom. Bez ich podpory by som nebol tam, kde stojím teraz.

# Abstrakt

Táto práca poskytuje metodický materiál pre vyučovanie časovej zložitosti zameraný pre maturantov z informatiky. Naším cieľom je ukázať študentom, že možnosti počítača sú obmedzené. Chceme ich naučiť približne odhadnúť čas potrebný na vykonanie programu. Študenti by následne mali byť schopní odhadnúť najväčšiu veľkosť vstupu, ktorý daný program zvládne spracovať za krátky čas. Taktiež by mali byť schopní použiť časovú zložitosť ako argument pri porovnávaní dvoch programov. Všetko toto demonštrujeme na programovacích úlohách a neformálne, bez použitia $O$ notácie.

**Kľúčové slová:** časová zložitosť, porovnávanie algoritmov, priemerný / najhorší prípad, metodika

# Abstract

This work offers a methodical material for teaching time complexity to high school students graduating in informatics. Our goal is to show the students that capabilities of a computer are limited. We aim to teach them to approximately estimate the running time of a computer program. Consequently, the students should be able to estimate the largest size of an input which a given program can process in a short amount of time. Also, the students should manage to use the time complexity as an argument when they compare two programs. We demonstrate all of this using programming problems and informally, without $O$ notation.

**Keywords:** time complexity, algorithm comparison, average / worst case, methodical material

# Úvod

Ako sa uvádza v cieľových požiadavkách na vedomosti a zručnosti maturantov z informatiky, žiak má vedieť „intuitívne uvažovať o zložitosti algoritmu (na úlohách primeranej zložitosti)“[18]. Cieľom tejto práce je vypracovať a otestovať materiál pre výučbu časovej zložitosti na intuitívnej úrovni. Chceme sa v ňom zamerať hlavne na nasledovné výučbové ciele:

- Študent vie, že výpočtové možnosti počítača sú obmedzené a že niektoré programy nevykoná za okamih.
- Študent vie odhadnúť s rádovou presnosťou počet príkazov, ktoré program vykoná v závislosti od veľkosti vstupu
- Študent vie na základe počtu vykonaných príkazov odhadnúť s rádovou presnosťou čas, ktorý bude program bežať pre vstup zadanej veľkosti.
- Študent vie odhadnúť s rádovou presnosťou najväčšiu možnú veľkosť vstupu, na ktorom program skončí do zadaného času.
- Študent vie využiť časovú zložitosť ako kritérium porovnávania dvoch programov.

V kapitole 1 uvádzame prehľad dostupnej literatúry a internetových zdrojov týkajúcich sa časovej zložitosti. Ďalej v kapitole 2 spomenieme tri typy úloh, ktoré sme zvažovali v tvorbe našej práce. Spomedzi nich vyberieme úlohy, v ktorých žiaci analyzujú zložitosť programu riešiaceho nejakú programátorskú úlohu. Pre jednu konkrétnu úlohu uvedieme v kapitole 3 metodiku pre zavedenie časovej zložitosti. Zvyšné úlohy, ktoré sme zvažovali a ktoré možno využiť pre ďalšie precvičovanie, uvedieme v kapitole 4. V kapitole 2 uvedieme dve užitočné programátorské konštrukcie, ktorými si vieme pomôcť pri výučbe časovej zložitosti. Na záver uvedieme v kapitole 6 postrehy z testovania metodiky v dvoch triedach mimo našej školy.

Zároveň kapitola 4 poskytuje prehľad programovacích úloh, pri ktorých je priestor na zaujímavú analýzu časovej zložitosti. Riešenia úloh pokrývajú väčšinu bežných časových zložitostí a tiež možno medzi nimi nájsť aj jednoduché, priamočiare riešenia, ale aj zložité riešenia s užitočnými programátorskými myšlienkami a zložitejšou časovou analýzou.

# Kapitola 1

# Prehľad literatúry

V slovenskej a českej literatúre sa nenachádza veľa materiálov týkajúcich sa výučbe časovej zložitosti u maturantov. Väčšina materiálov pochádza z vysokoškolských kurzov (napr. [2], [4], [14], [9]). Ďalším zdrojom materiálov sú programátorské súťaže, v ktorých sa často vyžaduje analýza časovej zložitosti programu. Slovenský Korešpondenčný seminár z pogramovania má na svojej internetovej stránke dva články o časovej zložitosti [12], [11]. Ďalším materiálom je kniha českého seminára z programovania [1]). Poznámky o časovej zložitosti možno nájsť aj na stránkach gymnázií vo Zvolene [15] a v Nitre [6].

Všetky uvedené materiály využívajú pri opisovaní zložitosti tzv. $O$ notáciu. Taktiež väčšina z úloh, ktoré sa naprieč nimi spomína, presahuje znalosti, ktoré sú vyžadované od maturantov z informatiky. Zo spomínaných materiálov nie je ľahké pripraviť hodinu pre maturantov z informatiky.

Úvodné pozorovania zavádzania konceptu zložitosti na stredných školách v Izraeli potvrdili, že pochopenie tohto pojmu je obťažné [5]. V centrálnom registri záverečných prác sa nenachádzajú žiadne práce, ktoré by sa venovali téme časovej zložitosti.

# Kapitola 2

# Výber typu úloh

V začiatkoch našej práce sme zvažovali tri rôzne smery, ktorými by sme sa mohli uberať pri vyučovaní časovej zložitosti. V nasledujúcich troch sekciách opíšeme tieto smery.

## 2.1 Neprogramátorské úlohy

Ide o úlohy, ktoré vytyčujú nejakú prostredie a isté kroky, ktoré v ňom možno vykonávať. V týchto úlohách sa môžeme dozvedať nejakú informáciu, niekoho presviedčať alebo meniť stav prostredia. Mnohé z nich majú charakter hry. Snažíme sa pritom dosiahnuť cieľ vytýčený v úlohe. Môže ísť o zistenie určitej informácie alebo dosiahnutie konkrétneho stavu. Skúmame pritom počet použitých krokov, ktorý chceme mať spravidla čo najmenší. S mnohými úlohami tohto typu sa možno stretnúť aj v matematických súťažiach. Uvedieme niekoľko príkladov.

**Úloha 2.1.** Kamarát si myslí číslo od 1 do 100. Môžeme sa ho spýtať ľubovoľnú zisťovaciu otázku, na ktorú nám kamarát pravdivo odpovie áno alebo nie. Ako máme postupovať, aby sme uhádli číslo? Koľko najmenej krokov potrebujeme, aby sme mali zaručené, že číslo určite uhádneme?

**Úloha 2.2.** Máme 128 vriec. Spomedzi nich 127 vriec obsahuje po 100 kovových mincí a jedno vrece obsahuje 100 zlatých mincí. Jedna kovová minca váži 10 g a jedna zlatá minca váži 9 g. Mince sú na pohľad nerozlíšiteľné. V jednom kroku môžeme položiť na váhu ľubovoľnú skupinu mincí a zistiť ich celkovú hmotnosť. Ako vieme nájsť vrece so zlatými mincami? Koľko najmenej krokov potrebujeme, aby sme vrece so zlatými mincami zaručene našli?

**Úloha 2.3.** Pexeso sa hrá nasledovne. Harry najskôr rozloží na stôl $2n$ kartičiek pexesa otočených obrázkom dolu. Na každom pexese je práve jeden obrázok. Obrázkov je $n$ a každý z nich je práve na dvoch pexesách. V každom ťahu hráč postupne otočí dve kartičky. Ak sú rovnaké, zoberie si ich. Ak sú rôzne, otočí ich naspäť. Harry sa

snažil pozbierať všetky kartičky na čo najmenej ťahov. Koľko najmenej ťahov potrebuje Harry, ktorý má mimo iného perfektnú pamäť, na to, aby pozbieral všetky kartičky, nech je pexeso na začiatku rozložené akokoľvek? [8]

Často je v úlohách žiaduce opakovať podobné ťahy alebo reagovať na výsledky predošlých ťahov. Preto sa na riešenie takýchto úloh môžeme pozerať ako na navrhnutie algoritmu, ktorý nám opisuje, ako máme vykonávať naše ťahy.

Ako možno vidno, v týchto úlohách možno jasne definovaná „časovú zložitosť“ algoritmu – je to počet krokov, ktoré spravíme. Tiež si môžeme všimnúť, že v úlohách často hrá proti nám istý prvok náhody. Napríklad v úlohe 2.1 môžeme mať šťastie a uhádnuť číslo na prvý pokus, ale tiež môžeme mať smolu a bude nám to trvať dlho. Preto tu môžeme nájsť priestor na diskusiu o najlepšom, najhoršom a priemernom prípade.

V úlohách 2.1 a 2.2 môžeme žiakov hravým spôsobom zoznámiť s myšlienkou binárneho vyhľadávania a logaritmickej časovej zložitosti. Z programátorského hľadiska ide o kombináciu cyklu s podmienkou – postupne v cykle sa pýtame otázky, pričom konkrétne znenie našej otázky volíme na základe odpovedí od kamaráta. Kombinácia cyklu a podmienky tak môže byť pre študentov prístupnejšia v takejto hravej forme, kedy im stačí svoje myšlienky sformulovať slovne a nemusia ich transformovať do programátorských konštrukcií.

Pri úlohách 2.1 a 2.2 môžeme poukázať na to, ako sa v počte potrebných krokov môžu líšiť rôzne algoritmy. Dokonca v úlohe 2.2 je ten rozdiel ešte výraznejší, kde nám so šikovnou myšlienkou stačí jedno váženie.

## 2.2 Vypisovanie hviezdičiek

Tento typ úloh sa spomína aj priamo v požiadavkách maturity z informatiky ako príklad práce so zložitosťou na intuitívnej úrovni [18]. Taktiež sa takáto úloha vyskytla v pilotnom testovaní maturantov Monitor 2004 [20]. V týchto úlohách študenti dostanú napísaný program, ktorý iba vypisuje hviezdičky s využitím programovacích konštrukcií ako sú cyklus a vetvenie. V programoch pritom môžu vystupovať premenné (zväčša len jedna), ktoré ovplyvňujú počet vypísaných hviezdičiek. Úlohou študentov je zväčša zistiť, koľko hviezdičiek program vypíše. V prípade, že program obsahuje premenné, tak treba nájsť závislosť počtu vypísaných hviezdičiek od vstupných parametrov.

```
1 n = int(input())
2 for i in range(n):
3     for j in range(i):
4         print('*')
```

Program 2.1: Príklad programu vypisujúceho hviezdičky

Ako nedostatok týchto úloh možno vnímať absenciu motivácie u študentov. Úlohy tohto typu môžu pôsobiť ako samoúčelné a môže sa stať, že študenti ich budú vnímať odtrhnuté od praktického sveta. Na druhej strane, práve vďaka ich samoúčelnosti sa môžeme pri takýchto úloha sústrediť práve na časovú zložitosť a študenti nebudú rozptyľovaní programovaním a porozumením algoritmov.

## 2.3 Programátorské úlohy

Posledným spôsobom je ilustrovať otázky ohľadom časovej zložitosť na skutočných programoch, ktoré budú riešiť nejaký programátorský problém. Ide o typ úloh, s akým sa študenti pripravujúci sa na maturitu často stretávajú. Preto by im nemali byť cudzie. Tiež pri nich študenti vedia experimentovať a merať čas svojho programu na rôznych vstupoch. Rizikom je, že programátorská časť pridáva na komplexnosti riešených problémov a študenti môžu mať väčšie ťažkosti s pochopením ako pri predošlých typoch úloh.

Môže sa stať, že študenti budú mať riešenia s inou asymptotickou zložitosťou. V takom prípade vie učiteľ nadviazať otázkou, ako by sme vedeli porovnať, ktoré riešenie je lepšie.

Je tu priestor poukázať aj na iné aspekty ovplyvňujúce čas programov, napr. ak si nenačítam súbor do poľa a zakaždým ním prechádzam, tak je to pomalšie.

V našej práci sme si zvolili zameranie na programátorské úlohy. Našou hlavnou motiváciou bola skutočnosť, že práve s takýmto typom úloh sa študenti pri príprave na maturitu stretávajú najčastejšie. Analýzu časovej zložitosti možno tak zaradiť po rôznych úlohách, ktoré so študentmi bežne na seminároch programujeme.

# Kapitola 3

# Metodika

Východiskovým bodom našej metodiky je demonštrácia otázok ohľadom časovej zložitosti na programátorských úlohách. Na vybranej úlohe môžu študenti uvidieť, že počítač nie je schopný všetkého a v niektorých situáciách mu môže výpočet trvať veľa času. Niekedy až toľko, že odhadovaný čas programu je príliš veľký.

Riešenie otázok časovej zložitosti ilustrujeme na jednej úlohe. Programovanie vybranej úlohy možno zakomponovať do precvičovania práce s jednorozmernými poľami a súbormi.

## Základné informácie

- **Cieľová skupina:** študenti gymnázia pripravujúci sa na maturitu (3. alebo 4. ročník).
- **Časová dotácia:** 90 min (najlepšie dvojhodinovka seminára) + 45 min na precvičenie.
- **Pomôcky:**
  - projektor;
  - tabuľa;
  - počítač pre každého študenta, príp. do dvojice;
  - zadanie úloh pre študentov;
  - súbory so vstupmi pre študentov.

## Vstupné požiadavky na študentov

1. Študent vie vytvoriť pole potrebnej veľkosti, načítať doňho hodnoty a pristúpiť na miesta poľa.

2. Študent vie identifikovať opakujúci sa vzor a zostaviť for cyklus sledujúci tento vzor.
3. Študent vie použiť vnorený cyklus.
4. Študent vie zistiť, či počas vykonávania cyklu bola aspoň raz splnená podmienka (napr. pomocou premennej).
5. Študent vie v poli čísel spočítať počet výskytov jednotlivých čísel (napr. spočítať počet padnutí čísel 1 až 6 pri hádzaní hracou kockou).
6. Študent vie prečítať údaje zo súboru a uložiť ich do poľa.

# Výchovno vzdelávacie ciele hodiny

1. Študent vie, že výpočtové možnosti počítača sú obmedzené a že niektoré programy nevykoná za okamih.
2. Študent vie odhadnúť s rádovou presnosťou počet príkazov, ktoré program vykoná v závislosti od veľkosti vstupu.
3. Študent vie na základe počtu vykonaných príkazov odhadnúť s rádovou presnosťou čas, ktorý bude program bežať pre vstup zadanej veľkosti.
4. Študent vie odhadnúť s rádovou presnosťou najväčšiu možnú veľkosť vstupu, na ktorom program skončí do zadaného času.
5. Študent vie využiť časovú zložitosť ako kritérium porovnávania dvoch programov.

## Kostra hodiny a časový plán

Uvedieme jednotlivé časti našej hodiny, ktoré budeme opisovať v nasledujúcich sekciách tejto kapitoly. Ku každej z nich uvádzame aj odporúčaný čas.

1. 5 min: Predstavenie zadania úlohy (sekcia 3.1).
2. 25 min: Riešenie a programovanie úlohy formou podľa výberu učiteľa (sekcia 3.2).
3. 15 min: Spoločné prejdenie riešení úlohy (riešenia a návody k ním uvádzame v sekcii 3.3).
4. 5 min: Úvodná diskusia o časovej zložitosti (sekcia 3.4).

5. 10 min: Odhad závislosti počtu vykonaných príkazov od veľkosti vstupu na konkrétnom riešení (sekcia 3.5).

6. 5 min: Odhad času behu konkrétneho programu na základe odhadnutého počtu príkazov (sekcia 3.5).

7. 10 min: Odhad času behu pre zvyšné programy (sekcia 3.6).

8. 10 min: Porovnanie programov z pohľadu časovej zložitosti (sekcia 3.7).

9. 5 min: rezerva pre riešenie problémov a odpovedanie na otázky.

Precvičenie na iných úlohách realizujeme na ďalšej hodine (sekcia 3.8).

## 3.1 Zadanie úlohy

**Úloha 3.1.** Vláda vymyslela nový systém na podpisovanie petícií. Každý občan štátu má pridelený jednoznačný identifikátor, čo je prirodzené číslo menšie ako 1 000 000. Keď niekto v systéme dokončí petíciu, systém pošle vláde súbor, ktorý obsahuje identifikátory podpísaných ľudí, v každom riadku jeden.

Vzniklo podozrenie, že kvôli chybe v systéme niektoré petície obsahovali duplicitné podpisy. Vláda si vás najala, aby ste toto podozrenie skontrolovali.

Vašou úlohou je napísať program, ktorý dostane súbor s podpismi petície a zistí, či petícia obsahuje duplicitný podpis (aspoň jeden). Ak áno, vypíše identifikátor človeka, ktorého podpis je v petícii dvakrát. V prípade, že duplicitných podpisov je viacero, stačí vypísať jeden z nich. Pokiaľ petícia duplicitný podpis neobsahuje, program vypíše, že je korektná.

## 3.2 Riešenie úlohy

Študenti ako súčasť zadania dostanú niekoľko súborov s podpismi. Počty podpisov v súbore sú rôzne a rastú až k jednému miliónu. Pre šikovnejších študentov máme pripravené aj súbory s petíciami z inej krajiny, kde identifikačné čísla môžu byť ešte väčšie.

Je dobré, keď si študenti úlohu skúsia sami vyriešiť. Môžeme k riešeniu úlohy pristupovať podľa našich zvykov. Napríklad

- môže dať úlohu vyriešiť študentom samostatne, príp. v dvojiciach;
- môže najprv spolu so študentmi prejsť kľúčové myšlienky a nechá ich previesť povedané myšlienky do programu.

Je však prospešné ponechať študentom nejakú voľnosť pri vymýšľaní riešenia, aby sa mohli objaviť rôzne prístupy, o ktorých sa dá potom diskutovať.

Ako študenti postupne dokončujú svoj program a spúšťajú ho na rôznych súboroch, tak si môžu všimnúť, že pri veľkých súboroch program neskončí hneď, ale trvá mu to niekoľko sekúnd, respektíve jeho skončenia sa ani nedočkajú. Pokiaľ máme ešte čas, kým riešenie úlohy naprogramuje zvyšok triedy, môžeme týchto študentov inštruovať, nech popremýšľajú, či by vedeli vytvoriť taký program, ktorý by zvládol vyriešiť aj veľké súbory.

## 3.3 Riešenia

So študentmi si prejdeme riešenia úloh, na ktoré prišli, či už pri samostatnom programovaní alebo spoločnom v rámci triedy. Uvádzame tri princípy riešenia úlohy. Neočakávame, že študenti prídu na všetky z nich. Väčšina študentov pri testovaní prišla len na riešenie A. Preto k riešeniam uvádzame aj niekoľko otázok, ktorými možno študentov na riešenie naviesť. Je totiž dobré ukázať študentom aj iné riešenie úlohy. Študenti tak môžu uvidieť, že ten istý problém môže byť vyriešený rôzne rýchlymi programami, čo posilňuje dôvod, prečo je užitočné sa časovou zložitosťou zaoberať.

Pokiaľ študenti prídu len na riešenie A, odporúčame si spoločne so študentmi prejsť riešenie B, obzvlášť keď študenti nevedia usporiadať pole čísel. Vyhneme sa tak problému, ako usporiadať čísla podľa veľkosti a hlavne problému, ako dlho usporadúvanie trvá. Riešenie B je tiež podobné úlohe s počítaním hodov kocky, s ktorou sa mohli už študenti stretnúť.

```
1  nazov_suboru = input('Zadajte nazov suboru s peticiou: ')
2  subor = open(nazov_suboru)
3  n = int(subor.readline())
4  podpisy = [int(riadok) for riadok in subor]
5  korektna = True
6  for i in range(n):
7      for j in range(i):
8          if podpisy[i] == podpisy[j]:
9              korektna = False
10             print('V peticii bol najdeny duplicitny podpis s id', podpisy[i])
11 if korektna:
12     print('Peticia je korektna.')
```

Program 3.1: Hľadanie duplicitného podpisu pomocou porvnávania každej dvojice podpisov

## Riešenie A: porovnávanie každých dvoch podpisov

Najviac priamočiare riešenie je založené na porovnávanie každého podpisu s každým. Môžeme to spraviť napríklad tak, že si najprv obsah súboru načítame do poľa. Potom v dvoch for cykloch prejdeme cez všetky dvojice indexov $i$ a $j$. Pre každú dvojicu skontrolujeme, či sa $i$-ty podpis líši od $j$-teho podpisu. Pri tomto riešení si musíme dať pozor na to, aby sme nechtiac neporovnali dva podpisy na rovnakom mieste. Program 3.1 to zaručuje tým, že skúša len prípady, kedy $i < j$.

Tento princíp možno naprogramovať vo viacero modifikáciách. Napr. jeden z cyklov môže byť nahradený prechádzaním súboru alebo funkciou na zistenie, či sa číslo nachádza v poli.

## Riešenie B: pomocou počítania počtu výskytov.

Súborom s podpismi prejdeme len raz. Miesto toho, aby sme si podpisy ukladali do poľa, tak spočítame, koľkokrát sa tam ktorý podpis nachádza. Na to si zavedieme pole veľkosti $m + 1$, kde $m$ je maximálna možná hodnota identifikátora občana. Jeho prvky nastavíme na nuly. Pri prechádzaní súboru s podpismi zvyšujeme počítadlá pre počet výskytov jednotlivých podpisov. Keď prídeme na podpis, ktorý má už zvýšené počítadlo na 1, tak prehlásime, že sme našli duplicitný podpis.

```
1  nazov_suboru = input('Zadajte nazov suboru s peticiou: ')
2  subor = open(nazov_suboru)
3  pocet = [0] * 10000000
4  korektna = True
5  for riadok in subor:
6      id = int(riadok)
7      if pocet[id] >= 1:
8          korektna = False
9          print('V peticii bol najdeny duplicitny podpis s id', id)
10     pocet[id] += 1
11 if korektna:
12     print('Peticia je korektna.')
```

Program 3.2: Hľadanie duplicitného podpisu pomocou počítania počtu výskytov

**Návod.** Keď chceme skontrolovať petíciu, tak vlastne chceme nájsť podpis, ktorý sa v nej vyskytuje dvakrát (resp. viackrát), prípadne ukázať, že taký podpis nie je. Čo ak by sme teda pre každý podpis spočítali, koľkokrát sa v petícii vyskytuje? Skúste si spomenúť, počítali sme už niekedy počet výskytov nejakých čísel? Počítali sme koľkokrát padlo ktoré číslo na pri hádzaní kockou. Nevedeli by sme podobný princíp využiť aj tu?

## Riešenie C: pomocou usporiadania

Študenti môžu prísť ešte s jedným spôsobom riešenia úlohy, a to pomocou usporiadania. Totiž keď máme identifikátory podpisov usporiadané vzostupne, rovnaké identifikátory sa budú nachádzať vedľa seba. Stačí nám potom iba jeden prechod poľom. Pre študentov môže byť problém, ako usporiadať čísla v poli, zvlášť, keď sa s tým ešte nestretli. Neočakávame, že by študenti mali vedieť triediace algoritmy, napokon to nie je ani v požiadavkách na maturitu [18]. Môžeme však študentom prezradiť, že pre usporiadanie poľa môžu použiť vstavanú, príp. knižničnú funkciu. V jazyku Python môžeme použiť funkciu `sorted(pole)`, ktorá dostane za parameter pole a vráti nové, usporiadané pole, alebo metódu `sort()`, ktorú môžeme použiť na pole (v programe napíšeme `pole.sort()`) a utriedi nám priamo zadané pole. Môžeme zvážiť, že miesto toho, aby sme študentom priamo povedali, ktorú funkciu môžu použiť, môžeme ich nechať vyhľadať správnu funkciu na internete.

```
1  nazov_suboru = input('Zadajte nazov suboru s peticiou: ')
2  subor = open(nazov_suboru)
3  podpisy = [int(riadok) for riadok in subor]
4  subor.close()
5  podpisy.sort()
6  korektna = True
7  for i in range(len(podpisy) - 1):
8      if podpisy[i] == podpisy[i + 1]:
9          korektna = False
10         print('V peticii bol najdeny duplicitny podpis s id', podpisy[i])
11 if korektna:
12     print('Peticia je korektna.')
```

Program 3.3: Hľadanie duplicitného podpisu po usporiadaní podpisov

**Návod.** V riešení A si podpisy uložíme do poľa tak, ako sa nachádzajú v súbore. Keď si zoberieme nejaký podpis a hľadáme, či sa tam vyskytuje na inom mieste, tak nám toto hľadanie trvá dlho, lebo musíme prejsť celým poľom. Nevedeli by sme si uložiť podpisy v nejakej šikovnejšej podobe? Aby nám takéto vyhľadávanie išlo rýchlejšie alebo aby sme vedeli v tejto podobne podpis rýchlo vyhľadať? Skúste sa zamyslieť nad situáciami z bežného života. Keď potrebujete vy alebo zamestnanec v niektorom konkrétnom povolaní rýchlo nájsť číslo (alebo iný údaj, napr. meno) medzi veľa údajmi, akým spôsobom zvyknú byť tieto údaje uložené.

S takýmito situáciami sa môžeme stretnúť, keď hľadáme telefónne číslo v našom adresári na mobile alebo keď lekár hľadá niekoho zdravotnú kartu. Vo väčšine takýchto situáciách sú potrebné údaje usporiadané. Tu si študenti vedia uvedomiť, že usporiadanie nám vie túto úlohu uľahčiť. Keď si zoberieme nejaký podpis, tak ak sa tam nachádza duplicitne, tak jeho duplikáty sa musia nachádzať vedľa neho.

## 3.4 Diskusia o časovej zložitosti programu

Po tom, čo sme si so študentmi prešli ich riešenia (príp. riešenie) úlohy, prejdeme k diskusii so študentmi. Predpokladáme, že väčšina študentov prišla k riešeniu pomocou porovnávaním v dvoch cykloch. Tak to bolo aj pri všetkých našich testovaniach. Preto sa budú nasledovné úvahy o časovej zložitosti týkať programu, ktorý porovnáva všetky dvojice podpisov. Preto ani nie je potrebné, aby žiaci mali riešení viacero. Na základe výsledkov testovaní očakávame, že väčšina študentov bude mať len riešenie A.

Pokiaľ študenti programovali sami, príp. v skupinkách, tak sa ich spýtame, či skúšali spúšťať program na veľkých súboroch a či si niečo pritom všimli. Študenti tu povedia, že na veľkých súboroch program nevypísal výsledok. Program 3.1 môžeme demonštračne spustiť na vhodne veľkom vstupe, aby tento problém videli všetci študenti. Takto si vieme poradiť aj v prípade, keď nikto z nich väčšie vstupy neskúšal. Ideálne je vybrať taký vstup, na ktorom program skončí, rádovo tak za niekoľko sekúnd, aby si študenti nemysleli, že program nič nevypisuje kvôli chybe. Tento vstup je potrebné vybrať pred hodinou, keďže presný čas behu programu sa medzi počítačmi líši. Pre náš počítač je vhodný súbor `5000.txt`, na ktorom program beží asi 4 sekundy.

So študentmi prejdeme do diskusie, v ktorej prídeme k potrebe o odhadovaní času behu programu. Uvedieme dva možné smery diskusie prispôsobené tomu, či sa medzi študentmi objavili aj iné riešenia ako riešenia A.

Motiváciu pre určovanie časovej zložitosti zakladáme na potrebe odhadnúť čas behu programu. Keď si zoberieme dostatočne veľký súbor, napr. s milión podpismi, tak program 3.1 neskončí za pár sekúnd. Má vôbec zmysel čakať? Ak by program mal bežať niekoľko hodín, tak to zvládneme počkať (môžeme nechať počítač bežať cez noc). Ak by to malo byť niekoľko rokov, tak program veľmi použiteľný nie je. Budú to minúty, hodiny, dni, roky? Alebo ešte dlhšie? Vieme tento čas aspoň nejako rádovo odhadnúť?

Pokiaľ majú študenti viacero riešení, tak vieme ich motiváciu zložiť aj na potrebe porovnať riešenia: Máme tu dva programy, ktoré riešia tú istú úlohu. Vieme ich nejako porovnať? Ktorý program ju vyrieši rýchlejšie? Iste, že to vieme zmerať tak, že tie programy pustíme na rovnakom počítači. To však nie je veľmi praktické, ak jednému programu to trvá deň a druhému rok. Vedeli by sme to spraviť aj bez toho? Iba na základe toho, ako je program napísaný?

## 3.5 Odhad času behu programu

Po uvedení potreby odhadnúť čas behu programu pristúpime k samotnému odhadovaniu. Z čoho by sme sa vedeli odraziť? Máme dve základné východiská, od ktorých sa vieme odraziť a to:

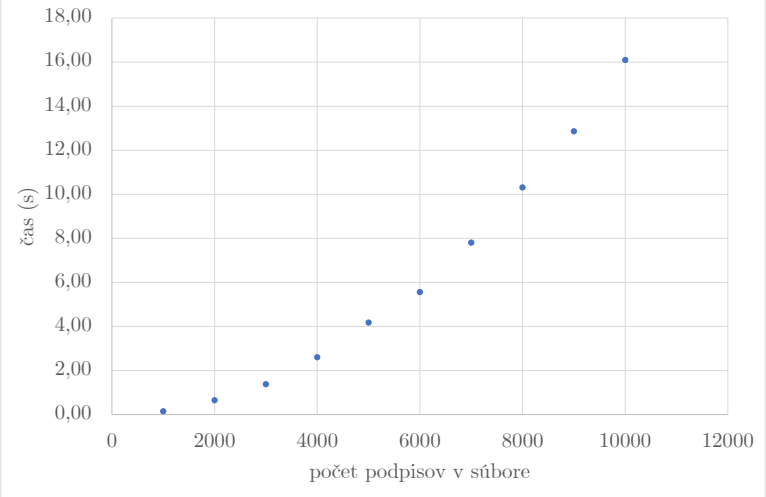
1. ako vyzerá kód nášho programu;

2. koľko trvá beh programu na vstupoch, kde beží dostatočne malý čas.

Študentom dáme za úlohu, nech odmerajú, ako dlho im beží program. Prípadne vieme túto časť spraviť aj v rámci spoločnej demonštrácie. Časy behu programu na malých vstupoch uvádzame v tabuľke 3.1. Odporúčame odmerať čas behu programu aspoň na niektorých vstupoch, aby sme mali reálne údaje, od ktorých sa vieme odraziť. Pokiaľ chceme, môžeme z odmeraných časov zostrojiť aj graf, napríklad ako na obrázku 3.1. Podotýkame, že presný čas behu programu záleží od mnohých faktorov, aj od samotného počítača, na ktorom sme program spúšťali. Preto ich treba brať len orientačne a keď na hodine použijeme iný počítač, môžeme očakávať iné hodnoty. Z toho dôvodu si treba dať pozor, aby sme nekombinovali namerané časy na rôznych počítačoch.

| $n$ | 10 | 50 | 100 | 500 | 1000 | 5 000 | 10 000 |
|---|---|---|---|---|---|---|---|
| Čas (s) | 0, 0008 | 0, 0007 | 0, 002 | 0, 04 | 0, 15 | 4 | 15 |

Tabuľka 3.1: Časy behu programu 3.1 na vstupoch s $n$ podpismi




Obr. 3.1: Graf závislosti času behu programu 3.1 od veľkosti vstupu

Ako ovplyvňuje kód nášho programu to, ako dlho bude bežať? Čo spôsobuje dlhý beh programu? Študenti môžu aj sami navrhnúť, že môže ísť o počet vykonaných príkazov či inštrukcií, resp. môžu dať podobný nápad, od ktorého sa dá odraziť. Pokiaľ študenti nevedia ako, môžeme ich na to naviesť.

1. Ako sme mohli vidieť na demonštrácii, program bežal na väčších vstupoch dlhšie a dlhšie.

2. Čo také počítač vlastne robí?

3. Keď si uvedomíme, že počítač vlastne len vykonáva príkazy, ktoré sme mu zadali, tak zrejme tých príkazov bude robiť veľa.

Pristúpime k samotnému spočítaniu príkazov. Pre lepšiu názornosť budeme robiť úvahy pre vstupný súbor s fixným počtom podpisov, konkrétne 1 000 000. Počítanie príkazov robíme spoločne so študentmi, pričom im vieme napomáhať návodnými otázkami.

- Koľko opakovaní spraví cyklus `for i in range(len(podpisy)):`?
- Koľko príkazov sa spraví v jednom kroku cyklu `for j in range(i)`?
- Koľko opakovaní spraví cyklus `for j in range(i)`, ak je $i = 0$? Koľko ak je $i$ rovné 1, 2, 3? Vieme

## Počet vykonaných príkazov

Odhad počtu vykonaných príkazov môže vyzerať asi takto:

1. 1 príkaz: načítanie počtu podpisov `n = int(input())`.
2. 1 príkaz: vytvorenie poľa pre podpisy `podpisy = [0] * n`.
3. $10^6$ príkazov: načítanie podpisov do poľa pomocou cyklu.
4. 1 príkaz: priradenie `korektna = True`.
5. $3 \cdot 10^6(10^6 - 1)/2$ príkazov: kontrola duplikátov pomocou dvoch for cyklov:
   - V jednom opakovaní cyklu `for j in range(i)` sa vykonajú najviac 3 príkazy.
   - Cyklus `for j in range(i)` sa vykoná postupne 0-krát (pre $i = 0$), 1-krát ($i = 1$), 2-krát ($i = 2$), 3-krát ($i = 3$), ..., $(10^6 - 1)$-krát ($i = 10^6 - 1$). Spolu sa teda vykoná $0 + 1 + 2 + 3 + \cdots + 10^6 - 1 = 10^6(10^6 - 1)/2$ opakovaní.
6. 2 príkazy: kontrola podmienky a prípadný výpis.

Spolu sme teda napočítali

$$10^6 + 3 \cdot 10^6(10^6 - 1)/2 + 5 = \frac{3}{2} \cdot 10^{12} + \frac{1}{2} \cdot 10^6 + 5 \text{ príkazov.}$$

Očakávame, že na mnohých miestach v programe budú od študentov rôzne návrhy na to, koľko príkazov sa v skutočnosti vykoná. Hlavne vie ísť o miesta:

- V kroku 2 samotné vytvorenie poľa vie trvať dlho, pokiaľ je pole veľké. Preto by sme mali rátať $10^6$ príkazov miesto jedného. (To sa aj v Pythone naozaj deje.)

- V kroku 5 sa v jednom opakovaní vnútorného cyklu väčšinou vykoná iba podmienka, preto niektorí študenti môžu chcieť v jednom opakovaní cyklu započítať iba jeden príkaz.

- For cyklus môže v sebe obsahovať nejaké skryté príkazy – napr. nejaký čas zaberie zvýšenie premennej cyklu.

Pokiaľ majú študenti iné návrhy, prejdeme si ich s nimi a ich výsledné počty príkazov si napíšeme na tabuľu. Pokiaľ nespravili výraznú chybu v odhadovaní počtu príkazov, výsledné počty príkazov by mali byť rádovo rovnaké, teda okolo $10^{12}$ príkazov. Najväčšie rozdiely by sa mali pohybovať zhruba v 10 % rozsahu.

Študenti môžu ešte navrhnúť pridať do programu 3.1 príkaz `break`, ak už nájdeme duplicitné podpisy, čo vie program urýchliť. Pri tomto návrhu vieme dať študentom nasledovné otázky na rozmyslenie: Ako veľmi týmto urýchlime beh programu na korektnej petícii? Ako asi odhadujete, že pridanie príkazu `break` urýchli program? Aký bude efekt pri kontrole korektnej petície? Skúste si to domerať na Vašich programoch. Pri správnej petícii si tým nepomôžeme. Pri nekorektnej petícii si tak v priemernom prípade program dvakrát urýchlime. K rádovej zmene teda tak nedôjde.

Diskusiu o počte uzavrieme tým, že nepotrebujeme presný počet príkazov, ale uspokojíme sa iba s rádovým odhadom. Je nám teraz jedno, či bude program bežať hodinu alebo dve. Ide nám o to zistiť, či pobeží pár sekúnd, pár hodín, pár dní, pár rokov alebo ďaleko viac. Študentov sa teda spýtame, či by na základe doterajších úvah vedeli stanoviť približnú hodnotu počtu príkazov. Tá by mala byť okolo $10^{12}$ príkazov. Poukážeme potom na to, že pri tomto odhadovaní nezáleží na tom, či započítame mimo cyklov +5, +1 alebo +11 príkazov. Dokonca ani na tom, či vytvorenie poľa započítame za 1 príkaz alebo $10^6$. Stále dostaneme rádovo $10^{12}$ príkazov, čo vieme demonštrovať aj na iných výsledkoch od žiakov.

Keď sme už určili, koľko príkazov počítač vykoná pri kontrole $10^6$ podpisov, môžeme naše úvahy zovšeobecniť a spraviť ich všeobecne pre vstupný súbor obsahujúci $n$ podpisov. So študentmi úvahy zopakujeme a prídeme k tomu, že počítač vykoná približne $n^2$ príkazov. Pri našich úvahách vlastne iba nahradíme $10^6$ premennou $n$.

## Odhad času

Keď máme odhadnutý počet príkazov, pokúsime sa odhadnúť čas, ktorý bude na ich vykonanie program potrebovať. Čo na to potrebujeme? Potrebujeme odhadnúť, ako dlho trvá počítaču vykonať jeden príkaz. Resp. koľko príkazov spraví za sekundu alebo inú jednotku času. Ktorý údaj počítača by nám to vedel prezradiť? Ak ste si niekedy pozerali minimálne systémové požiadavky na spustenie niektorej hry, čo za údaje ste sledovali? Ktoré údaje ovplyvňujú, ako rýchlo bude bežať hra na počítači.

Študenti môžu navrhovať veľkosť RAM, no tá nám priamo o čase nič nehovorí, hoci ho do veľkej miery ovplyvňuje – hlavne pri komplexnejších programoch akými sú napr. počítačové hry. Užitočným parametrom je frekvencia procesora. Ako veľká býva? Môžeme študentov nechať pohľadať na internete či v nastaveniach alebo pre ušetrenie času im môžeme rovno prezradiť, že súčasné počítače majú frekvenciu procesora rádovo niekoľko GHz. Študentov sa spýtame, čo tá jednotka znamená. Hertze by mali poznať už z fyziky. Jeden gigahertz znamená miliardu za sekundu, v tomto prípade ide o $10^9$ počítačových operácií za sekundu. Teda inými slovami jednu operáciu vykoná počítač rádovo za $10^{-9}$ sekundy.

Poznamenáme, že ako vyplynie z neskoršieho postupu, tento odhad nie je veľmi presný. Zanedbáva totiž, že počítač nepracuje v jazyku Python, ale v oveľa jednoduchšom jazyku. Jeden príkaz jazyka Python sa tak môže niekedy interpretovať aj ako desiatky, možno aj stovky strojových inštrukcií. Ďalším faktorom je, že náš program nemá procesor celý pre seba. Na počítači máme spustený operačný systém, mnohé procesy na pozadí a možno aj otvorené iné aplikácie. Tieto faktory vedia značne znížiť počet príkazov jazyka Python vykonaných za sekundu.

Študentov nemusíme s týmto problémom zaťažovať hneď teraz. Môžeme pokojne (pokiaľ sami nejaké vylepšenia nenavrhnú) pokračovať s odhadom $10^9$ príkazov za sekundu, ktorý neskôr opravíme.

## Záverečný odhad

S týmto údajom, môžeme odhadnúť, že náš program bude bežať pre $n = 10^6$ asi $10^3$, príp. $10^4$ (ak sme mali menší odhad na počet príkazov za sekundu) sekúnd. To zodpovedá rádovo zopár hodinám v tom horšom prípade. To vyzerá, že náš program nie je úplne bezvýznamný a s miernou dávkou trpezlivosti sa výsledku dočkáme.

## Overenie odhadu

Zišlo by sa ale ešte náš odhad overiť. Odhadnime, ako dlho by mal program bežať podľa nášho odhadu na tých vstupoch, na ktorých si ho vieme odmerať. Tieto odhady pre $10^9$ operácií za sekundu možno nájsť v tabuľke. Opäť pripomenieme, že nemusíme merať čas na všetkých súboroch, stačí vybrať aj dva.

Necháme študentov, nech zhodnotia presnosť nášho odhadu, nech skúsia nájsť, v čom náš odhad nebol dokonalý a ako by sa dal vylepšiť.

1. Myslíte si, že nám frekvencia procesora hovorí priamo o počte príkazov programovacieho jazyka Python vykonaných za sekundu alebo pôjde o iné operácie.

2. Pracuje každý procesor v jazyku Python alebo má vlastný jazyk?

| $n$ | 10 | 50 | 100 | 500 | 1000 | 5 000 | 10 000 |
|---|---|---|---|---|---|---|---|
| Odhad | $10^{-7}$ | $2,5 \cdot 10^{-6}$ | $10^{-5}$ | $2,5 \cdot 10^{-4}$ | $10^{-3}$ | $0,025$ | $0,1$ |
| Skutočnosť | $8 \cdot 10^{-4}$ | $7 \cdot 10^{-4}$ | $2 \cdot 10^{-3}$ | $4 \cdot 10^{-2}$ | $1,5 \cdot 10^{-1}$ | 4 | 15 |
| Oprava | $10^{-5}$ | $2,5 \cdot 10^{-4}$ | $10^{-3}$ | $2,5 \cdot 10^{-2}$ | $10^{-1}$ | $2,5$ | 10 |

Tabuľka 3.2: Porovnanie odhadovaných časov behu programu 3.1 na vstupoch s $n$ podpismi s nameranými časmi. Všetky časové údaje sú v sekundách.

3. Má náš program celý počítač sám pre seba alebo sa oň musí deliť s nejakými inými programami alebo procesmi, ktoré tiež využívajú procesor?

4. Viete povedať, ako veľmi sme sa pomýlili pri našom odhade?

5. Neoplatí sa pri našej chybe pozrieť skôr na to, koľkokrát je horší náš odhad ako o koľko je horší?

Na odmeraných časoch si môžeme všimnúť, že sme urobili chybu približne o dva rády. Skutočné časy sú s výnimkou $n = 10$ vždy rádovo 100-krát dlhšie. So študentmi tak vieme prísť na to, že sme neodhadli dobre počet príkazov vykonaných za sekundu. Lepší odhad by bol uvažovať $10^7$ príkazov jazyka Python za sekundu. Pomocou neho vieme získať vylepšené odhady, ktorý uvádzame v poslednom riadky tabuľky 3.2.

Na záver ešte poznamenáme, že hodnota $10^7$ sa môže líšiť v závislosti od druhu počítača a programovacieho jazyku. Nemalo by však ísť o veľký rádový skok. Azda najväčší rozdiel možno dosiahnuť použitím rýchlejšieho programovacieho jazyka (napr. C++ alebo Pascal), kedy sa počet vykonaných príkazov za sekundu pohybuje skôr okolo $10^8$.

## 3.6 Odhad zložitosti zvyšných programov

Keď už máme so študentmi spravené aj aspoň jedno z riešení B, C, spravíme aj preň odhad zložitosti.

### Odhad zložitosti riešenia B

Odhadneme, koľko príkazov vykoná program 3.2 pri súbore, ktorý obsahuje $n$ podpisov.

1. 1 príkaz: načítanie názvu súboru,

2. 1 príkaz: otvorenie súboru,

3. $m$ príkazov: vytvorenie poľa.

| Počet podpisov: | 10 | 100 | 1 000 | 10 000 | 100 000 | 1 000 000 |
|---|---|---|---|---|---|---|
| Priemerný čas behu (v s): | 0,07 | 0,07 | 0,07 | 0,08 | 0,2 | 1,1 |

4. najviac $5n$ príkazov: cyklus pre čítanie zo súboru sa zopakuje $n$-krát a pri každom opakovaní vykoná najviac 5 príkazov:
   (a) 1 príkaz: pretypovanie riadku na číslo,
   (b) najviac 3 príkazy: kontrola podmienky a prípadné priradenie a výpis,
   (c) 1 príkaz: zvýšenie počítadla,
5. najviac 2 príkazy: kontrola podmienky a prípadný výpis.

Môžeme teda odhadnúť, že program vykoná spolu najviac $5n + m + 4$ príkazov. Opäť pripomenieme, že sa odhady vytvorené študentmi môžu líšiť v jednotlivých konštantách. Zo skúseností je očakávateľné, že študenti skôr započítajú vytvorenie poľa ako jeden príkaz.

Podľa nášho odhadu by mal náš program vykonať na súbore s $10^6$ podpismi približne $5 \cdot 10^6 + 10^7 + 4 \sim 10^7$ príkazov. Podľa našich zistení, že počítač zvládne vykonať asi $10^7$ príkazov za sekundu, by náš program mal skontrolovať tento súbor za rádovo niekoľko sekúnd. Priemerný čas behu programu pri našom meraní bol 1,1 sekundy (pozri aj tabuľku 3.3).

Pokiaľ študenti do odhadov nezapočítali čas potrebný na vytvorenie poľa, vie sa to teraz prejaviť pri odhadu času malých súborov. Ak študenti odhadujú, že počet príkazov bude rádovo niekoľkokrát $n$, program by mal pri malých vstupoch bežať krátko. Avšak keď odmeriame časy behu programu pre veľkosť vstupu postupne 10, 100 a 1000, môžeme si všimnúť, že je približne rovnaký. To nesedí s odhadom, že by sa mal s rastúcim $n$ lineárne zvyšovať. Na tom si môžu študenti všimnúť, že aj pri malých súboroch niečo stále dlho trvá. Tým miestom je práve vytváranie značne veľkého poľa.

## Odhad zložitosti riešenia C

Problémom pri odhadovaní zložitosti riešenia C, je určiť, koľko príkazov sa vykoná pri volaní triediacej funkcie, o ktorej nevieme, ako funguje. Môžeme študentov nechať vyhľadať odpoveď na internete. Pritom sa hodí informácia, že anglický pojem pre časovú zložitosť je *time complexity*. Druhou možnosťou je priamo povedať študentom, že utriedenie poľa obsahujúce $n$ prvkov vyžaduje približne $n \cdot \log_2 n$ operácií. Treťou možnosťou je nechať študentov tento údaj odhadnúť z nameraných časov programu.

Môžu sa objaviť otázky ohľadom základu logaritmu. V literatúre sa základ logaritmu väčšinou neuvádza, hoci z analýzy mnohých algoritmov možno dostať logaritmus

| Počet podpisov: | 10 | 100 | 1 000 | 10 000 | 100 000 | 1 000 000 |
|---|---|---|---|---|---|---|
| Priemerný čas behu (v s): | 0,07 | 0,07 | 0,07 | 0,08 | 0,2 | 1,1 |

Tabuľka 3.3: Čas behu programu 3.2 na rôznych veľkostiach vstupu

| Počet podpisov: | 10 | 100 | 1 000 | 10 000 | 100 000 | 1 000 000 |
|---|---|---|---|---|---|---|
| Priemerný čas behu (v s): | $6 \cdot 10^{-4}$ | $3 \cdot 10^{-4}$ | 0,001 | 0,01 | 0,14 | 1,7 |

Tabuľka 3.4: Čas behu programu 3.3 na rôznych veľkostiach vstupu

so základom 2. Keď si však uvedomíme, že $\log n = \log_2 n / \log_2 10$, teda desiatkový logaritmus je približne $0,3$-násobok dvojkového, tak zmena základu logaritmu je v podstate iba zmena konštanty. Rádový odhad nám teda neovplyvní. Preto môžeme úvahy o základe vynechať a predísť tak komplikáciám s ťažším chápaním logaritmov u študentov.

Jeden zo spôsobov odhadu časovej zložitosti je nasledovný:

1. 3 príkazy: načítanie názvu súboru, jeho otvorenie a zatvorenie,
2. $n$ príkazov: načítanie obsahu súboru do poľa,
3. $n \log n$ príkazov: usporiadanie poľa,
4. najviac $3(n-1)$ príkazov vo for cykle,
5. najviac 2 príkazy pri záverečnom výpise.

Spolu teda program 3.3 vykoná najviac $n \log n + 4n + 2$ príkazov. Teda na vstupe s $n$ podpismi vykoná približne $n \log n$ príkazov.

Podľa nášho odhadu na vstupe s $n = 10^6$ petíciami, by mal program 3.3 vykonať $10^6 \log 10^6 = 6 \cdot 10^6$ príkazov. Preto by jeho beh mal trvať približne zopár sekúnd. To korešponduje aj s nameranými časmi, ktoré uvádzame v tabuľke 3.4.

## 3.7 Porovnanie časových zložitostí riešení

Keď už máme dva programy riešiace ten istý problém, môžeme sa spýtať študentov, či by vedeli zhodnotiť, ktorý je lepší. Hlavné rozdiely sú nasledovné:

a) Program 3.2 zvládne za niekoľko sekúnd skontrolovať aj súbory obsahujúce milión podpisov na rozdiel od programu 3.1.

b) Takisto zvládne vstupy s milión podpismi spracovať aj program 3.3, ktorý je v praxi len o niečo pomalší ako program 3.2.

c) Na menších vstupoch je rýchlejší program 3.1, no program 3.2 je stále dostatočne rýchly.

d) Programy 3.1 a 3.3 na menších vstupoch spotrebujú menej pamäte.

Od študentov očakávame, že zvládnu objaviť prvé dva rozdiely. Porovnanie na základe pamäte ani nemusíme uvádzať, je to samostatné kritérium porovnávania programov, ktoré táto metodika nepokrýva.

## 3.8 Úlohy na precvičenie

Na ďalšej hodine zadáme študentom niekoľko úloh na precvičenie odhadovania času behu programu.

**Úloha 3.2.** Nasledovný program načíta číslo $n$ a zistí o ňom, či je prvočíslo.

```
from math import sqrt, floor

n = int(input())
prvocislo = True
for delitel in range(2, floor(sqrt(n))):
    if n % delitel == 0:
        prvocislo = False
if prvocislo and n > 1:
    print('Je prvocislo')
else:
    print('Nie je prvocislo')
```

Bez toho, aby ste program spustili, rádovo odhadnite:

1. Ako dlho bude program bežať, ak mu zadáme $n = 100$?
2. Ako dlho bude program bežať, ak mu zadáme $n = 1\,000\,000$?
3. Aké najväčšie môže byť zadané číslo $n$, aby beh programu trval najviac niekoľko sekúnd?

**Riešenie.** Program obsahuje jeden for cyklus, ktorý sa zopakuje $\lfloor\sqrt{n}\rfloor$-krát. V každom opakovaní sa vykonajú najviac dva príkazy. Okrem neho tam máme 5 príkazov, ktoré sa vykonajú najviac raz. Preto vidíme, že spolu je príkazov rádovo $\sqrt{n}$. Použijeme náš vytvorený odhad, že počítač vykoná za sekundu rádovo $10^7$ príkazov jazyka Python. To znamená, že

a) Pri vstupe $n = 100$ bude program bežať rádovo $\sqrt{100}/10^7 = 10^{-6}$ s.

b) Pri vstupe $n = 1\,000\,000$ bude program bežať rádovo $\sqrt{10^6}/10^7 = 10^{-4}$ s.

c) Aby program skončil za niekoľko sekúnd, tak by mal vykonať najviac $10^7$ príkazov. Preto musí platiť $\sqrt{n} \leq 10^7$, z čoho dostaneme $n \leq 10^{14}$. Tým teda máme odhad, že najväčšie $n$, kedy bude program skončí do niekoľkých sekúnd, je $10^{14}$. (Na našom počítači program bežal $3,8$ s.)

**Úloha 3.3.** Nasledovný program načíta pole $n$ čísel zo súboru a zistí, či sa medzi nimi nachádzajú tri za sebou idúce čísla. (Napr. tri za sebou idúce čísla sa nachádzajú v číslach 15, 43, 17, 3, 42, 67, 8, 44, 46, 47.)

```
1  nazov_suboru = input('Zadajte nazov suboru s peticiou: ')
2  subor = open(nazov_suboru)
3  n = int(subor.readline())
4  pole = [int(riadok) for riadok in subor]
5  je_tam = False
6  for i in range(n):
7      for j in range(n):
8          for k in range(n):
9              if pole[k] - pole[j] == 1 and pole[j] - pole[i] == 1:
10                 je_tam = True
11 if je_tam:
12     print('Medzi cislami SU tri za sebou iduce cisla')
13 else:
14     print('Medzi cislami NIE SU tri za sebou iduce cisla')
```

Bez toho, aby ste program spustili, rádovo odhadnite:

1. Ako dlho bude program bežať, ak mu zadáme súbor obsahujúci $n = 100$ čísel?
2. Ako dlho bude program bežať, ak mu zadáme súbor obsahujúci $n = 1\,000\,000$ čísel?
3. Aký najväčší môže byť počet čísel v súbore, aby beh programu trval najviac rádovo niekoľko sekúnd?

**Riešenie.** Cyklus `for k in range(n):` sa zopakuje $n$-krát a v každom opakovaní vykoná najviac dva príkazy. To máme zatiaľ $2n$ príkazov. Týchto $2n$ príkazov sa zopakuje $n$-krát v cykle `for j in range(n):`, čo dáva $2n^2$ príkazov. Toto všetko sa zopakuje ešte $n$-krát cyklom `for i in range(n):`, z čoho vychádza $2n^3$ príkazov. Okrem toho máme ešte malý počet príkazov, ktoré sa vykonajú raz, a načítanie obsahu súboru do poľa obsahujúce $n$ príkazov. V oboch prípadoch ide o zanedbateľný počet príkazov, tak môžeme odhadnúť, že počet vykonaných príkazov programu bude rádovo $n^3$. To znamená, že

1. Pre $n = 100$ bude program bežať rádovo $100^3/10^7 = 0,1$ s.
2. Pre $n = 10^6$ bude program bežať rádovo $(10^6)^3/10^7 = 10^{11}$ s, čo je asi 3 000 rokov.
3. Z podmienky $n^3/10^7 \leq 1$ vieme získať $n \leq \sqrt[3]{10^7} \approx 215$. Teda aby program bežal najviac rádovo niekoľko sekúnd, v súbore sa môže nachádzať rádovo zopár stoviek čísel.

## 3.9 Alternatívy a modifikácie

Metodiku možno rôznymi spôsobmi upraviť. Prvým prvkom, ktorý možno zmeniť, je samotná úloha. Miesto úlohy 3.1 možno vybrať inú, primeranú schopnostiam a zručnostiam študentov. Pri výbere je vhodné vybrať úlohu tak, aby čo najlepšie podporila u študentov motiváciu. Prehľad ďalších úloh uvádzame v nasledovnej kapitole. Vhodnými kandidátmi sú úlohy 4.1 a 4.6. Pri talentovaných študentov môžeme zvážiť aj úlohu 4.7.

Viaceré riešenia úlohy si možno prejsť so študentmi v rôznych častiach hodiny. V metodike opisujeme priebeh vhodný pre situáciu, kedy všetci žiaci až na zopár možných výnimiek majú len jedno riešenie, a to riešenie A. Tento priebeh možno použiť aj pri rozmanitých riešeniach – najprv si analyzujeme jedno a potom si prejdeme riešenie od iných študentov a vyšetríme to. Môžeme však už pred začatím diskusie si prejsť všetky riešenia študentov a od začiatku viesť diskusiu v duchu potreby porovnať dva rôzne programy riešiaci ten istý problém.

Veľký priestor pre voľnosť je v spôsobe precvičovania. Na precvičenie tiež môžeme vybrať úlohu zo sady úloh v kapitole 4. Precvičovanie nemusí prebiehať formou, kedy študenti dostanú hotové programy, ktoré majú analyzovať. Môžeme im zadať tieto programy najskôr naprogramovať. Taktiež precvičovanie práce s časovou zložitosťou môže prebiehať priebežne počas celého seminára. Pri vybraných úlohách, ktoré so študentmi programujeme, sa vieme po ich naprogramovaní spýtať študentov na analýzu ich zložitosti.

Tri vyučovacie hodiny, ktoré opisujeme, možno rozdeliť do dvojhodinoviek aj iným spôsobom. Úlohu 3.1 možno so študentmi vyriešiť v rámci samostatnej dvojhodinovke. Môžeme na nej naznačiť, že pri veľkých vstupoch môže program trvať príliš dlho. Potom nasledujúcu dvojhodinovku začneme prejdením si riešení úlohy 3.1, pokračujeme diskusiou o časovej zložitosti a dvojhodinovku zakončíme precvičením.

# Kapitola 4

# Ďalšie úlohy

Priestor na diskusiu o časovej zložitosti vzniká pri mnohých úlohách. My sme predstavili jednu dvojhodinovku, na ktorej sa študenti zoznámia s konceptom časovej zložitosti. Precvičovanie odhadovania času behu programu však nemusí prebiehať v bezprostrednej nadväznosti. Možno sa k týmto otázkam vrátiť pri rôznych úlohách.

Úlohy z tejto kapitoly nemusíme zadávať študentom naprogramovať. Pokiaľ chceme overiť určovanie zložitosti, stačí nám zadať študentom hotový program, pre ktorý majú odhadnúť časovú zložitosť. Konkrétnejšie, môžeme k zvolenému programu zadať nasledovné otázky:

1. Odhadnite, ako dlho bude program bežať, ak mu zadáme na vstup $n = 10^6$. Je dobré rovnako sa spýtať aj na iné hodnoty $n$, napr. $n = 10^3$, $n = 10^{12}$.

2. Odhadnite, akú najväčšiu hodnotu $n$ môžeme programu zadať, aby skončil rádovo do niekoľkých sekúnd.

V tabuľke 4.1 uvádzame prehľad, ktoré úlohy vyžadujú od študentov ktoré programátorské koncepty. V tabuľke používame nasledovné skratky:

- P označuje koncept, ktorý je potrebný na vyriešenie úlohy.
- O označuje odporúčaný koncept – pokiaľ ho Študenti neovládajú, môžu mať pri riešení úlohe ťažkosti alebo učiteľ bude musieť úlohu upraviť.
- R značí rozširujúci koncept, ktorý nie je nutný pre pochopenie úlohy. Je potrebný len pre rozširujúce riešenie, ktoré možno bez väčšej ujmy vypustiť, často spolu s jedným riešením úlohy.

Taktiež možno v tabuľke 4.1 vidieť časové zložitosti programov, ktoré vedia úlohu vyriešiť.

Každej úlohe je venovaná samostatná sekcia, v ktorej uvádzame podrobnejšie vstupné požiadavky na študentov ako aj metodické poznámky. Následne uvedieme niekoľko riešení spolu s analýzou ich časovej zložitosti.

| | Úloha | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| for cyklus | P | P | P | P | P | P | P | P |
| podmienka v cykle | P | P | P | P | P | P | P | P |
| while cyklus | O | – | – | – | O | R | – | – |
| pole | – | – | R | P | – | P | P | – |
| vnorený cyklus | – | – | P | – | – | P | P | – |
| súbor | – | – | – | O | – | P | O | – |
| Časová zložitosť | $n$, $\sqrt{n}$, $\log n$ | $n$, $\sqrt{n}$, $\log n$ | $n^2$, $n\sqrt{n}$, $n \log n$, $n \log \log n$ | $n \cdot s$, $n + s$ | $n$, $\log n$ | $n^2$, $n$, $n \log n$ | $n^3$, $n^2$, $n$ | $a^n$ |

Tabuľka 4.1: Prehľad obsiahnutých konceptov a časových zložitostí úloh kapitoly 4

Podotkneme, že z mnohých programov v tejto kapitole možno zostaviť aj úlohy, v ktorých žiaci zisťujú, koľko hviezdičiek vypíše program. Stačí pre to zobrať štruktúru niektorých z programov a nahradiť potrebné príkazy vypísaním hviezdičky.

V riešeniach úloh uvádzame kvôli stručnosti načítanie čísel do poľa pomocou *list comprehension*:

```
pole = [int(riadok) for riadok in subor]
```

Ide o skrátený zápis, ktorý je ekvivalentný nasledovnému napĺňaniu poľa:

```
pole = []
for riadok in subor:
        pole.append(int(riadok))
```

## 4.1 Najväčší nepárny deliteľ

**Úloha 4.1.** Pre zadané číslo $n$ nájdite najväčšieho nepárneho deliteľa čísla $n$.

**Príbeh k úlohe.** Miladka vyhrala v lotérii. Výhru si však nechce nechať pre seba. Preto usporiada večierok, na ktorý zavolá svoje kamarátky. Každú kamarátku volá spolu s jej manželom. Na večierku sa Miladka rovnomerne rozdelí so svojou výhrou so všetkými zúčastnenými, teda rovnako veľa dostane Miladka, každá z jej kamarátok a každý z ich manželov.

Napíšte pre Miladku program, ktorému Miladka zadá celé číslo – výhru v centoch. Program vypíše, koľko najviac ľudí sa môže zúčastniť večierka (vrátane Miladky). Peniaze nemožno deliť viac ako na centy a každý zúčastnený musí dostať na cent presne rovnako.

*Príklady:*

- Miladka vyhrá 60 centov. Najviac sa môže hostiny zúčastniť 15 ľudí: 7 párov a Miladka. Každý dostane po 4 centy.
- Miladka vyhrá 16 centov. Nič sa nedá robiť, nemá si koho zavolať. Večierka sa teda zúčastní sama a všetkých 16 centov jej ostane.
- Miladka vyhrá 36 700 160 centov. Najviac sa môže hostiny zúčastniť 35 ľudí: 17 párov a Miladka. Každý dostane 1 048 576 centov.

*Poznámka.* Úloha je zadaná v centoch, aby sme nemuseli premieňať eurá na centy a naopak.

**Požiadavky na študentov.** Študent vie

- získať číselný vstup od užívateľa a prezentovať mu výstup;
- identifikovať opakujúci sa vzor a zostaviť for cyklus sledujúci tento vzor;
- použiť podmienku v kombinácii s cyklom;
- využiť premennú na postupnú aktualizáciu výsledku programu počas cyklu;
- zistiť, či jedno číslo delí druhé;
- zistiť, či je číslo nepárne;
- využiť while cyklus pre vopred neznámy počet opakovania príkazov (pre riešenie 4.1.3).

**Metodické poznámky.** Táto úloha si zaslúži svoju pozornosť pre svoju jednoduchosť, ako možno vidieť aj z požiadaviek na študenta. Spomedzi všetkých našich úloh nevyžaduje prácu s poľom, ani vnorené použitie cyklov či nejaký zložitejší programátorský koncept. Napriek tomu na nej možno ilustrovať dve zaujímavé zložitosti programov – odmocninovú a logaritmovú. Výhody logaritmickej zložitosti možno veľmi pekne ilustrovať v jazyku Python, v ktorom môžeme bez problémov zadať veľmi veľké čísla, s ktorými si program 4.4 poradí v okamihu. Pri analýze programu 4.4 sa dotkneme aj otázky najhoršieho prípadu. To môže študentom robiť problémy pri analýze zložitosti, keďže na väčšine vstupov môže náš program skončiť veľmi rýchlo. Napr. program 4.4 spraví len jedno opakovanie cyklu na každom nepárnom čísle.

Táto úloha bola zadaná v Korešpondenčnom seminári z programovania [13]. Na stránke úlohy možno po prihlásení odovzdať program riešiaci úlohu, ktorý bude automaticky otestovaný. Žiaci si tak môžu sami otestovať, či sa im úlohu podarilo vyriešiť správne a taktiež aj ako rýchle je ich riešenie. Upozorňujeme však, že zadanie úlohy je mierne komplikovanejšie, nakoľko vyžaduje, aby program vypísal odpoveď pre viacero zadaných čísel. Taktiež v práci uvádzame odlišný príbeh pre prípad, žeby delenie

výhry pôsobilo na žiakov lepšie ako delenie čokolády, pri ktorej nie je moc reálne, aby obsahovala miliardy a viac dielikov.

### 4.1.1 Riešenie skúšaním všetkých čísel

Priamočiary prístup spočíva v tom, že postupne skúšame čísla menšie alebo rovné $n$, či sú nepárne a či delia $n$ a vyberieme z nich to najväčšie. Programy sa môžu líšiť v konkrétnych detailoch. Miesto zisťovania, či je číslo nepárne, nám stačí sa iba pohybovať po nepárnych číslach. To môžeme dosiahnuť zmenou kroku for cyklu alebo aj vlastnou premennou, ktorú zvyšujeme / znižujeme o dva. Ďalším rozdielom je smer, z ktorého čísla skúšame. Jednou možnosťou je skúšať čísla vzostupne ako v programe 4.1. V tomto prípade si musíme pamätať doposiaľ najväčšieho nájdeného deliteľa v premennej. Druhou možnosťou je skúšať čísla zostupne od $n$ ako v programe 4.2. Vtedy nám stačí skúšanie ukončiť hneď, keď nájdeme nepárneho deliteľa, lebo vieme, že musí byť najväčší.

```
1 n = int(input())
2 najvacsi = 1
3 for delitel in range(1, n + 1, 2):
4     if n % delitel == 0:
5         if delitel > najvacsi:
6             najvacsi = delitel
7 print(najvacsi)
```

Program 4.1: Hľadanie najväčšieho nepárneho deliteľa vzostupne

```
1 n = int(input())
2 for delitel in range(n, 0, -1):
3     if delitel % 2 == 1 and n % delitel == 0:
4         print(delitel)
5         break
```

Program 4.2: Hľadanie najväčšieho nepárneho deliteľa zostupne

Najskôr určíme časovú zložitosť programu 4.1. Ten sa skladá z troch príkazov a for cyklu, ktorý spraví $n/2$ opakovaní. V každom opakovaní sa vykonajú najviac dva príkazy: kontrola podmienky a zmena premennej `najvacsi`. Spolu teda program 4.1 vykoná najviac $3 + n$ príkazov, čo je rádovo $n$ príkazov.

Pri určovaní časovej zložitosti programu 4.2 narazíme na jednu prekážku. Nevieme na prvý pohľad určiť, koľkokrát sa v ňom vykoná for cyklus, lebo v ňom máme príkaz break. Ak je $n$ nepárne, tak sa celý cyklus zopakuje len raz. Naopak, pokiaľ hľadaný najväčší nepárny deliteľ je malý, tak cyklus bude robiť veľa opakovaní. Konkrétne, ak $n$ je mocnina dvojky (a teda najväčší nepárny deliteľ čísla $n$ je 1), tak for cyklus prejde všetkými číslami od $n$ až po 1. Spraví tak $n$ opakovaní.

Keď chceme dávať záruky, ako rýchlo náš program skončí, tak musíme uvažovať najhorší prípad. Teda prípad, kedy cyklus spraví $n$ opakovaní. V každom opakovaní

spraví jeden príkaz – kontrolu podmienky. Výpis spraví iba v jednom opakovaní cyklu – v tom poslednom. Spolu s načítaním vstupu teda máme $n+1$ príkazov. Program 4.2 teda vykoná rádovo najviac $n$ príkazov.

Oba programy 4.1 aj 4.2 spravia rádovo rovnako príkazov. Odmeraním časov (ako aj možno vidieť v tabuľke 4.2) si môžeme všimnúť, že program 4.1 je dvakrát rýchlejší.

### 4.1.2 Riešenie skúšaním čísel po odmocninu

Program 4.1 možno značne vylepšiť nasledovnou úvahou. Môžeme sa pozrieť na to, ako hľadáme všetkých deliteľov nejakého čísla. Keďže nevieme, čím je deliteľné a čím nie je, tak musíme postupne skúšať všetky čísla. Avšak keď zistíme, že číslo $n$ je deliteľné číslom $d$, tak vieme, že $n$ bude mať aj deliteľa $n/d$. Delitele sa nám teda vyskytujú v takýchto dvojiciach. Preto keď budeme skúšať deliteľov po dvojiciach $d$ a $n/d$, tak nám stačí skúšať hodnoty $d$ len po $\sqrt{n}$.

```
from math import floor, sqrt

n = int(input())
najvacsi = 1
for delitel in range(1, floor(sqrt(n)) + 1):
    if n % delitel == 0:
        if delitel % 2 == 1 and delitel > najvacsi:
            najvacsi = delitel
        druhy_delitel = n // delitel
        if druhy_delitel % 2 == 1 and druhy_delitel > najvacsi:
            najvacsi = druhy_delitel
print(najvacsi)
```

Program 4.3: Hľadanie najväčšieho nepárneho deliteľa vzostupne po $\sqrt{n}$

Program 4.3 obsahuje for cyklus, ktorý sa zopakuje $\sqrt{n}$-krát. V jednom opakovaní sa vykoná najviac 6 príkazov. Spolu program 4.3 vykoná najviac $6\sqrt{n}+3$ príkazov, čo je rádovo $\sqrt{n}$ príkazov.

### 4.1.3 Riešenie postupným delením dvomi

Hľadanie deliteľov si tiež vieme predstaviť cez prvočíselný rozklad čísla. Každý deliteľ čísla $n$ vznikne tak, že vyberieme z jeho prvočíselného rozkladu nejaké prvočísla a vynásobíme ich. Nepárny deliteľ môže vzniknúť len násobením nepárnych prvočísel. Ak má byť najväčší, tak musí vzniknúť vynásobením všetkých nepárnych prvočísel. Inými slovami, dostaneme ho tak, že z prvočíselného rozkladu čísla $n$ odstránime všetky párne prvočísla, teda všetky dvojky.

```
n = int(input())
while n % 2 == 0:
    n //= 2
print(n)
```

Program 4.4: Hľadanie najväčšieho nepárneho deliteľa pomocou postupného delenia dvomi

Keďže v programe 4.4 používame cyklus while, tak opäť počet vykonaných príkazov môže byť veľmi premenlivý. Preto aj pri analýze tohto programu sa pozrieme na počet vykonaných príkazov v najhoršom prípade, teda vtedy, keď $n$ bude mocninou dvojky. Vtedy totiž vydelíme $n$ dvomi najväčší možný počet krát. Ak $n = 2^k$, tak číslo $n$ vydelíme dvomi presne $k$-krát. Číslo $k$ teda vieme vyjadriť ako $k = \log_2 n$. Program 4.4 teda vykoná najviac $2 + \log_2 n$ príkazov, čo je rádovo $\log_2 n$ príkazov.

Poznamenáme, že v praxi sa často uvádza zložitosť bez základu logaritmu, teda $\log n$. Je to preto, lebo $\log_2 n = \log n / \log 2$, teda zmena základu ovplyvňuje iba konštantu, nie rádový odhad.

Tu si môžeme všimnúť, že program 4.4 je veľmi rýchly. Na vstupoch s $n \leq 2^{45}$, na ktorých sme merali čas predošlých programov, beží takmer nepozorovateľne krátko. Dokonca tak krátko, že ťažko odpozorovať od zmeraných časov nejaké rozdiely v správaní. Nárast času sa prejaví až u podstatne väčších vstupoch. Pre $n = 2^{200}$ sme odmerali čas $10^{-4}$ s.

Poďme si odhadnúť, na akom vstupe bude program 4.4 bežať rádovo sekundu. To by musel vykonať rádovo $10^7$ príkazov, teda by muselo platiť $\log_2 n \sim 10^7$, z čoho dostaneme, že $n \sim 2^{10\,000\,000}$. Ak si skúsime také veľké $n$ zadať, zistíme, že náš odhad nie je dobrý. Taktiež vieme experimentálne zistiť, že program 4.4 bude bežať 2 s pre $n \sim 2^{50\,000}$. Nepresnosť nášho odhadu spôsobilo to, že práca s veľkými číslami (ako napr. ich delenie dvomi) tiež trvá nejaký čas – nie jednu inštrukciu, ako sme predpokladali doteraz, ale viacero inštrukcií v závislosti od počtu cifier čísla.

| $n$ | prog. 4.1 | prog. 4.2 | prog. 4.3 | prog. 4.4 |
|---|---|---|---|---|
| $2^5$ | $2 \cdot 10^{-5}$ | $2 \cdot 10^{-5}$ | $2 \cdot 10^{-5}$ | $1 \cdot 10^{-5}$ |
| $2^{10}$ | $10^{-4}$ | $2 \cdot 10^{-4}$ | $2 \cdot 10^{-5}$ | $1 \cdot 10^{-5}$ |
| $2^{15}$ | $4 \cdot 10^{-3}$ | $9 \cdot 10^{-3}$ | $6 \cdot 10^{-5}$ | $1 \cdot 10^{-5}$ |
| $2^{20}$ | $2 \cdot 10^{-1}$ | $4 \cdot 10^{-1}$ | $3 \cdot 10^{-4}$ | $2 \cdot 10^{-5}$ |
| $2^{25}$ | 6 | 10 | $2 \cdot 10^{-3}$ | $2 \cdot 10^{-5}$ |
| $2^{30}$ | 200 | 400 | $10^{-2}$ | $2 \cdot 10^{-5}$ |
| $2^{45}$ | 200 | 400 | 3 | $2 \cdot 10^{-5}$ |
| $2^{50\,000}$ | $10^{15\,000}$ | $10^{15\,000}$ | $10^{7\,500}$ | 2 |

Tabuľka 4.2: Porovnanie časov v sekundách pre programy riešiace úlohu 4.1

## 4.2 Zisťovanie prvočíselnosti

**Úloha 4.2.** Pre zadané kladné celé číslo $n$ zistite, či je prvočíslom.

**Požiadavky na študentov.** Študent vie

- získať číselný vstup od užívateľa a prezentovať mu výstup;
- identifikovať opakujúci sa vzor a zostaviť for cyklus sledujúci tento vzor;
- použiť podmienku v kombinácii s cyklom;
- využiť premennú na postupnú aktualizáciu výsledku programu počas cyklu;
- zistiť, či jedno číslo delí druhé;
- použiť matematické funkcie pre odmocninu a prevod reálneho čísla na celé 4.1.3).

**Metodické poznámky.** Ide o celkom známu algoritmickú úlohu, ktorá sa zvykne pri vyučovaní informatiky objavovať. Úloha je tiež jednoduchá a mnohé atribúty zdieľa s úlohou 4.1. Tiež študenti tu môžu ľahko otestovať, na akých veľkých vstupoch dosiahne ich program svoje časové limity. Rovnako sa tu môžu stretnúť s odmocninovou zložitosťou a jej rozdielom oproti lineárnej zložitosti. Lineárne riešenie je obťažnosťou primerané pre maturantov, tak by nemali mať s ním problémy. K zlepšeniu na odmocninové stačí jednoduchá úprava na základe nie až tak obťažnej myšlienky (čo zväčša neplatí pri neskorších úlohách).

### 4.2.1 Riešenie skúšaním všetkých deliteľov

Priamočiare riešenie spočíva vo vyskúšaní čísel od 2 menších ako zadané číslo $n$. Pokiaľ medzi nimi nájdeme číslo, ktoré delí $n$, tak $n$ nie je prvočíslom. V opačnom prípade prvočíslom je (ak je väčšie ako 1).

```
1 n = int(input())
2 prvocislo = True
3 for delitel in range(2, n):
4     if n % delitel == 0:
5         prvocislo = False
6 if prvocislo and n > 1:
7     print('Je prvocislo')
8 else:
9     print('Nie je prvocislo')
```

Program 4.5: Zisťovanie prvočíselnosti skúšaním všetkých čísel menších ako $n$

Program 4.5 vykoná rádovo $n$ príkazov. Možno ho ešte vylepšiť tým, že budeme skúšať okrem dvojky iba nepárne čísla. Dostaneme tak polovičný počet príkazov, hoci stále ich bude rádovo $n$. V praxi sa to prejaví tým, že dostaneme približne dvakrát

```
from math import sqrt, floor

n = int(input())
prvocislo = True
for delitel in range(2, floor(sqrt(n))):
    if n % delitel == 0:
        prvocislo = False
if prvocislo and n > 1:
    print('Je prvocislo')
else:
    print('Nie je prvocislo')
```

Program 4.6: Zisťovanie prvočíselonosti skúšaním čísel po $\sqrt{n}$

tak rýchly program. Na toto vylepšenie sú schopní prísť študenti aj sami. Pokiaľ im ukážeme aj riešenie s odmocninovou časovou zložitosťou, tak môžeme poukázať na to, že ide o výrazne lepšie zrýchlenie ako iba na polovicu času.

### 4.2.2 Riešenie skúšaním deliteľov po odmocninu

Predchádzajúce riešenie možno zrýchliť rovnakým princípom ako pri riešení 4.1.2 úlohy 4.1. Keďže delitele každého celého čísla $n$ možno usporiadať do párov, z ktorých je jeden najviac $\sqrt{n}$, tak nám stačí vyskúšať čísla väčšie ako 2 a nanajvýš rovné $\sqrt{n}$.

Časová zložitosť programu 4.5 je $n$, teda dokáže za pár sekúnd overiť čísla do $10^7$. Program 4.6 má zložitosť $\sqrt{n}$, teda dokáže do pár sekúnd preveriť čísla až do $10^{14}$. Na tejto úlohe je pozoruhodné, aký veľký rozdiel spravila celkom drobná úprava kódu vychádzajúca z užitočného pozorovania.

Poznamenáme, že existujú aj ďaleko rýchlejšie algoritmy, ktoré dokážu zistiť prvočíselnosť čísel obsahujúcich stovku cifier. Ich náročnosť však výrazne presahuje požiadavky na maturanta.

## 4.3 Počet prvočísel menších ako zadané číslo

**Úloha 4.3.** Pre zadané číslo $n$ zistite, koľko je prvočísel menších ako $n$.

**Požiadavky na študentov.** Študent vie

- získať číselný vstup od užívateľa a prezentovať mu výstup;
- identifikovať opakujúci sa vzor a zostaviť for cyklus sledujúci tento vzor;
- použiť podmienku v kombinácii s cyklom;
- využiť premennú na postupnú aktualizáciu výsledku programu počas cyklu;
- zistiť, či jedno číslo delí druhé;

- zistiť, či číslo je prvočíslo;
- použiť v cykle pred tým naprogramovaný program či už pomocou vlastnej funkcie alebo vnoreného cyklu;
- vytvoriť pole potrebnej veľkosti, nastaviť jeho počiatočné hodnoty a pristupovať k prvkom poľa (riešenie 4.3.2);
- využiť pole a údaje v ňom pri návrhu algoritmu (riešenie 4.3.2).

**Metodické poznámky.** Ide o už náročnejšiu úlohu, v ktorej sa opakovane využíva nejaký algoritmus. Je to spôsob ako ukázať kvadratickú zložitosť (ak nekončíme testovanie prvočíselnosti pri odmocnine), ktorý nevyžaduje od študentov ovládať pole. Tiež tu možno dostať celkom zaujímavú zložitosť $n\sqrt{n}$. Myšlienka využitia programu na zisťovanie prvočíselnosti v cykle však môže byť náročnejšia. Pri analýze zložitosti Eratostenovho sita sa študenti môžu stretnúť so zložitosťou $n \log n$, príp. $n \log \log n$. Avšak táto analýza presahuje matematický aparát maturantov.

### 4.3.1 Riešenie zisťovaním prvočíselnosti o každom čísle

Toto riešenie vieme zostaviť priamym použitím programu na zisťovanie prvočíselnosti. Prejdeme v cykle všetkými číslami od 2 po $n-1$ a o každom zistíme, či je prvočíslo.

```
1  from math import floor, sqrt
2
3  n = int(input())
4  pocet = 0
5  for i in range(2, n):
6      prvocislo = True
7      for delitel in range(2, floor(sqrt(i)) + 1):
8          if i % delitel == 0:
9              prvocislo = False
10     if prvocislo:
11         pocet += 1
12 print('Pocet prvocisel mensich ako', n, ':', pocet)
```

Program 4.7: Počítanie prvočísel testovaním každého čísla

Program 4.7 $(n-2)$-krát zopakuje test prvočíselnosti, ktorý vykoná rádovo $\sqrt{i}$ krokov. Jeho časová zložitosť bude teda rádovo $\sqrt{2} + \sqrt{3} + \cdots + \sqrt{n-1}$. Keďže platí

$$n\sqrt{n} \sim \frac{n-1}{2}\sqrt{\frac{n-1}{2}} \leq \sqrt{2} + \sqrt{3} + \cdots + \sqrt{n-1} \leq (n-2)\sqrt{n-1} \sim n\sqrt{n},$$

tak môžeme usúdiť, že program 4.7 vykoná rádovo $n\sqrt{n}$ príkazov.

V programe 4.7 môžeme, samozrejme, naprogramovať zisťovanie prvočíselnosti aj ako v programe 4.5, teda bez skrátenia s odmocninou. V takom prípade by sme tu dostali rádovo $1 + 2 + \cdots + n - 1 \sim n^2$ príkazov.

### 4.3.2 Eratostenovo sito

Eratostenovo sito využíva nasledovnú myšlienku: ak už zistíme, že nejaké číslo je prvočíslo, tak môžeme z testovania vylúčiť všetky jeho násobky. Budeme si teda udržiavať pole `prvocislo`, v ktorom budeme mať pre každé číslo zapamätané, či ešte je kandidátom na prvočíslo alebo už nie. Postupne budeme prechádzať všetkými číslami a ak natrafíme na doposiaľ nevylúčeného kandidáta, prehlásime ho za prvočíslo – ak by totiž mal náš kandidát nejakého netriviálneho deliteľa, tak by sme ho vylúčili. Po nájdení prvočísla vylúčime spomedzi našich kandidátov všetky jeho násobky.

```
1 n = int(input())
2 prvocislo = [False, False] + [True] * (n - 2)
3 pocet = 0
4 for i in range(2, n):
5     if prvocislo[i]:
6         pocet += 1
7         for x in range(2*i, n, i):
8             prvocislo[x] = False
9 print('Pocet prvocisel mensich ako', n, ':', pocet)
```

Program 4.8: Eratostenovo sito

Mimo for cyklu zo 4. riadku sa vykoná $3 + n$ príkazov vrátane vytvorenia poľa. For cyklus `for i in range(2, n):` sa zopakuje $n$ krát. V niektorých opakovaniach sa vykoná iba jeden príkaz – kontrola podmienky. Niektoré opakovania však vykonajú viac ako jeden príkaz – tie, pri ktorých sme našli prvočíslo. Aby sme si to uľahčili, tak najskôr spravíme veľmi hrubý horný odhad – budeme predpokladať, že sme v každom opakovaní cyklu našli prvočíslo. Cyklus `for x in range(2*i, n, i):` sa v jednom takom opakovaní vykoná rádovo $(n/i)$-krát. To znamená, že program 4.8 vykoná najviac rádovo

$$\frac{n}{2} + \frac{n}{3} + \cdots + \frac{n}{n-1} \leq n\left(\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}\right) = nH_n$$

príkazov. Súčet $1/1 + 1/2 + \cdots + 1/n$ sa zvykne označovať $H_n$ a nazýva sa *harmonické číslo*. Možno o ňom zistiť, že je rádovo rovné $\log n$ (presnejšie $\ln n$). Môžeme skonštatovať, že program 4.8 vykoná rádovo $n \log n$ príkazov. To znamená, že sme takto schopní do pár sekúnd nájsť všetky prvočísla zhruba do miliónu.

V skutočnosti možno odhad časovej zložitosti rádovo zlepšiť s využitím ešte pokročilejších znalostí. Bez predchádzajúceho zjednodušenia program 4.8 vykoná najviac rádovo

$$\frac{n}{2} + \frac{n}{3} + \frac{n}{5} + \frac{n}{7} + \cdots + \frac{n}{p} = n\left(\frac{1}{2} + \frac{1}{3} + \frac{1}{5} + \frac{1}{7} + \cdots + \frac{1}{p}\right)$$

príkazov, kde $p$ je najväčšie prvočíslo, ktoré je menšie ako $n$. Súčet prevrátených prvočísel menších ako $n$ možno rádovo odhadnúť pomocou $\log \log n$, teda program 4.8 vykoná celkovo rádovo najviac $n \log \log n$ krokov [3].

```
nazov_suboru = input('Zadajte nazov suboru: ')
subor = open(nazov_suboru)
n = int(subor.readline())
pole = [int(riadok) for riadok in subor]
pocet = [0] * 100
for x in pole:
    pocet[x] += 1
for i in range(100):
    print(i, pocet[i])
```

Program 4.9: Počítanie výskytov pomocou vlastných počítadiel

## 4.4 Počet výskytov prvku v poli

**Úloha 4.4.** Talentovej súťaže sa zúčastnilo 100 súťažiacich, ktorí mali pridelené čísla od 0 po 99. O postupe do ďalšieho kola rozhodujú diváci posielaním hlasov. Hlasy divákov sú zaznamenané v súbore, ktorý obsahuje v prvom riadku počet hlasov a v každom ďalšom riadku číslo súťažiaceho, ktorý dostal hlas. Napíšte program, ktorý pre každého súťažiaceho vypíše, koľko hlasov dostal.

**Požiadavky na študentov.** Študent vie

- načítať čísla zadané užívateľom do poľa;
- vytvoriť pole potrebnej veľkosti, nastaviť jeho počiatočné hodnoty a pristupovať k prvkom poľa;
- využiť pole pre počítanie počtu výskytov prvkov;
- spočítať výskyty prvku poľa pomocou funkcie `count`.

**Metodické poznámky.** Ide o celkom štandardnú úlohu, ktorú možno nájsť v jednoduchej verzii aj ako motivačnú úlohu pri predstavovaní poľa – počítame počet hodov hracej kocky [10]. Keď sa už študenti stretli s funkciou `count`, tak môžu navrhnúť pomerne jednoduché riešenie tejto úlohy s využitím tejto funkcie ako v programe 4.10.

So študentmi tu môžeme skúmať, aký je rozdiel medzi vlastným programovaním algoritmu 4.9 a používaním funkcie dostupnej v programovacom jazyku (program 4.10). Môžeme skúmať, ako na rýchlosť programu vplýva nielen počet hlasov (teda číslo $n$), ale aj počet súťažiacich. Počet súťažiacich môžeme zväčšiť alebo nahradiť parametrom $s$.

### 4.4.1 Riešenie pomocou vlastných počítadiel

Program 4.9 vykoná na vstupe s $n$ hlasmi a $s$ súťažiacimi najviac $n + s$ príkazov.

### 4.4.2 Riešenie pomocou funkcie count

```
1 nazov_suboru = input('Zadajte nazov suboru: ')
2 subor = open(nazov_suboru)
3 n = int(subor.readline())
4 pole = [int(riadok) for riadok in subor]
5 for i in range(100):
6     pocet = pole.count(i)
7     print(i, pocet)
```

Program 4.10: Počítanie výskytov pomocou funkcie `count`

Program 4.10 vykoná rádovo $ns$ príkazov. Vidíme, že počet vykonaných príkazov rastie lineárne s rastúcim $n$. Na druhej strane, hodnota $s$ môže byť veľmi veľká. To znamená, že v praxi očakávame, že program 4.10 bude o niekoľko rádov pomalší, zhruba toľkokrát aké je $s$.

Zaujímavosťou je, že v programovacom jazyku Python zrejme nebudeme pozorovať, že program 4.10 beží až $s$-krát pomalšie ako program 4.9. Pri našom meraní nám vyšlo, že program 4.10 bol iba 3-krát pomalší ako program 4.9. Tento rozdiel je spôsobený tým, že funkcia `count` je optimalizovaná, aby bežala rýchlo. Nič to však nemení na tom, že s rastúcim $s$ bude lineárne rásť aj čas behu programu 4.10, pričom čas behu programu 4.9 to nebude výrazne ovplyvňovať, pokým bude $s < n$.

## 4.5 Najväčší spoločný deliteľ

**Úloha 4.5.** Napíšte program, ktorý načíta prirodzené čísla $a$, $b$ a vypíše ich najväčšieho spoločného deliteľa.

**Požiadavky na študentov.** Študent vie

- získať číselný vstup od užívateľa a prezentovať mu výstup;
- identifikovať opakujúci sa vzor a zostaviť for cyklus sledujúci tento vzor;
- použiť podmienku v kombinácii s cyklom;
- využiť premennú na postupnú aktualizáciu výsledku programu počas cyklu;
- zistiť, či jedno číslo delí druhé;
- prerušiť vykonávanie cyklu pomocou príkazu break (pre riešenie 4.5.2);
- využiť while cyklus pre vopred neznámy počet opakovania príkazov (pre riešenie 4.5.3);
- pozná euklidov algoritmus na zisťovanie najväčšieho spoločného deliteľa (pre riešenie 4.5.3).

**Metodické poznámky.** Táto úloha je tiež význačná svojou jednoduchosťou – nevyžaduje použitie poľa. Avšak pre úvahy o časovej zložitosti má mnohé úskalia. Študenti o Euklidovom algoritme nemusia vedieť, keďže sa nenachádza ani v inovovanom štátnom vzdelávacom programe [17], ani v maturitných požiadavkách na matematiku [19]. Jeho pochopenie vie byť pre študentov náročné a môže tak zabrať veľa času. Taktiež aj analýza jeho časovej zložitosti je výrazne zložitejšia v porovnaní s taktiež logaritmickým riešením 4.1.3 úlohy 4.1. Stále nám však táto úloha umožňuje skúmať so študentmi hranice lineárneho riešenia.

### 4.5.1 Postupné skúšanie deliteľov zdola

Priamočiary spôsob, ako nájsť najväčšieho spoločného deliteľa, je postupné skúšanie všetkých čísel od 1 po menšie zo zadaných čísel $a$, $b$ (program 4.11), príp. v opačnom poradí (program 4.12).

```
a = int(input('Zadaj prve cislo: '))
b = int(input('Zadaj druhe cislo: '))
nsd = 1
for d in range(2, min(a, b)):
    if a % d == 0 and b % d == 0:
        nsd = d
print('NSD(a, b) =', nsd)
```

Program 4.11: Hľadanie najväčšieho spoločného deliteľa zdola

### 4.5.2 Postupné skúšanie deliteľov zhora

Priamočiarou analýzou možno zistiť, že programy 4.12 aj 4.11 vykonajú najviac rádovo $n$ príkazov.

```
a = int(input('Zadaj prve cislo: '))
b = int(input('Zadaj druhe cislo: '))
for d in range(min(a, b), 0, -1):
    if a % d == 0 and b % d == 0:
        print('NSD(a, b) =', d)
        break
```

Program 4.12: Hľadanie najväčšieho spoločného deliteľa zhora

### 4.5.3 Euklidov algoritmus

Pokiaľ študentom zvykneme učiť Euklidov algoritmus, tak sa tu otvára príležitosť na diskusiu o jeho časovej zložitosti.

```
a = int(input('Zadaj prve cislo: '))
b = int(input('Zadaj druhe cislo: '))
while b > 0:
    a, b = b, a % b
print('NSD(a, b) =', a)
```

Program 4.13: Euklidov algoritmus pre hľadanie najväčšieho spoločného deliteľa

Analýza časovej zložitosti Euklidovho algoritmu nie je priamočiara. Na prvý pohľad nie je zjavné, pri ktorých číslach vykoná while cyklus najviac opakovaní. Môžeme to zistiť spätnou analýzou. Pozrieme sa na jeden krok Euklidovho algoritmu, do ktorého vstúpili čísla $a$, $b$ (pričom $a > b$) a vystúpili z neho čísla $a' = b$ a $b' = a \bmod b$. Pri spätnom postupe budeme uvažovať, že čísla $a'$ a $b'$ poznáme a chceme určiť, aké najmenšie mohli byť čísla $a$ a $b$. Pri $b$ nemáme inú možnosť ako $b = a'$. Keďže $a$ má byť väčšie ako $b$, tak najbližšie číslo, ktoré dáva po delení $b$ zvyšok $b'$ je $a = a' + b'$.

| $a$ | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 |
|---|---|---|---|---|---|---|---|---|---|---|
| $b$ | 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 |

Tabuľka 4.3: Spätná konštrukcia najhoršieho vstupu pre Euklidov algoritmus

Prehľadnú spätnú rekonštrukciu najhorších vstupov možno vidieť v tabuľke 4.3. Môžeme si všimnúť, že všetky tieto vstupy nám vytvárajú jednu postupnosť. Prvými členmi sú $F_0 = 0$ a $F_1 = 1$. Každý ďalší člen je súčtom dvoch predchádzajúcich, teda $F_{n+2} = F_{n+1} + F_n$ pre všetky nezáporné celé čísla $n$. Ide tak o známu *Fibonacciho postupnosť*.

Môžeme sa presvedčiť, že Euklidov algoritmus vykoná na vstupe $a = F_n$, $b = F_{n-1}$ naozaj $n$ opakovaní. O Fibonacciho číslach je známy rádový odhad $F_n \sim \varphi^n$, kde $\varphi = (1 + \sqrt{5})/2 \approx 1,51$ je zlatý rez. Preto na taktom to vstupe Euklidov algoritmus vykoná rádovo $\log_\varphi n$ opakovaní while cyklu, teda aj rádovo $\log n$ príkazov.

## 4.6 Prienik dvoch množín

**Úloha 4.6.** Každý zamestnanec firmy má jedinečné identifikačné číslo – prirodzené číslo menšie ako 1 000 000. Firma zorganizovala pre svojich zamestnancov dve vzdelávacie konferencie. Pre každú konferenciu má v separátnom súbore zoznam zamestnancov, ktorí sa na konferencii zúčastnili. Firma ide zamestnancom vyplácať odmeny, a preto potrebuje zistiť, kto sa zúčastnil na oboch konferenciách.

Napíšte pre firmu program, ktorý dostane názvy dvoch súborov s účastníkmi konferencie a vypíše, koľko zamestnancov firmy sa zúčastnilo na oboch konferenciách. Upravte program tak, aby do tretieho súboru zapísal zoznam zamestnancov, ktorí sa zúčastnili na oboch konferenciách

**Požiadavky na študentov.** Študent vie

- vie prečítať údaje z dvoch súborov;
- vie zapísať výsledok programu do súboru;
- identifikovať opakujúci sa vzor a zostaviť for cyklus sledujúci tento vzor;
- použiť podmienku v kombinácii s cyklom;
- vie vytvoriť pole potrebnej veľkosti, zapisovať a čítať jeho hodnoty;
- vie zistiť, či sa daný prvok vyskytuje v poli (pre riešenie 4.6.2);
- usporiadať pole čísel (pre riešenie 4.6.3).

**Metodické poznámky.** Úlohu možno zaradiť k úlohám na prácu so súborom. Je do značnej miery podobná úlohe 3.1. Podobne aj tu máme tri riešenia – jedno založené na porovnávaní každej dvojice prvkov (teraz však je každý prvok z iného poľa), druhé využíva pole ako počítadlá výskytov pre jednotlivé hodnoty prvkov a tretie využíva usporiadanie polí. Úloha 4.6 je náročnejšia tým, že je v nej potrebné využívať dva súbory.

### 4.6.1 Kvadratické riešenie

Podobne ako riešenie A úlohy 3.1, program 4.14 vykoná rádovo $n^2$ príkazov.

```
1  nazov_suborov = input('Zadaj nazov suborov: ')
2  subor_1 = open(nazov_suborov + 'a.txt')
3  subor_2 = open(nazov_suborov + 'b.txt')
4  vystup = open(nazov_suborov + 'v.txt', 'w')
5  cisla1 = [int(riadok) for riadok in subor_1]
6  cisla2 = [int(riadok) for riadok in subor_2]
7  i = 0
8  j = 0
9  pocet = 0
10 for c in cisla1:
11     nachadza_sa = False
12     for d in cisla2:
13         if c == d:
14             nachadza_sa = True
15     if nachadza_sa:
16         pocet += 1
17 print('Na oboch konferenciach sa zucastnilo', pocet, 'zamestnancov.')
```

Program 4.14: Riešenie pomocou porovnávania každej dvojice čísel

### 4.6.2 Riešenie pomocou počítadiel

Informácie o tom, ktoré čísla sa nachádzajú v prvom súbore si uložíme šikovnejším spôsobom. Miesto toho, aby sme si čísla zaradom uložili do poľa, tak si vytvoríme pole veľkosti 1 000 000, ktoré bude mať pre každé možné číslo zamestnanca vlastné počítadlo, koľkokrát sa v súbore vyskytlo. Keďže zo zadania vieme, že počet výskytov čísla zamestnanca môže byť len 0 alebo 1 (zamestnanec buď bol na konferencii, alebo nie), preto môžeme používať hodnoty `True` a `False`.

Keď budeme prechádzať druhým súborom, tak budeme vedieť veľmi rýchlo skontrolovať, či sme dané číslo videli v prvom súbore – stačí sa nám pozrieť do poľa `vyskytol_sa` na správne miesto (na miesto s indexom čísla zamestnanca).

```
1  nazov_suborov = input('Zadaj nazov suborov: ')
2  subor_1 = open(nazov_suborov + 'a.txt')
3  subor_2 = open(nazov_suborov + 'b.txt')
4  vystup = open(nazov_suborov + 'v.txt', 'w')
5  m = 1000000
6  vyskytol_sa = [False] * m
7  pocet = 0
8  for riadok in subor_1:
9      cislo = int(riadok)
10     vyskytol_sa[cislo] = True
11 for riadok in subor_2:
12     cislo = int(riadok)
13     if vyskytol_sa[cislo]:
14         print(cislo, file=vystup)
15         pocet += 1
16 print('Na oboch konferenciach sa zucastnilo', pocet, 'zamestnancov.')
```

Program 4.15: Riešenie pomocou poľa indikátorov

Program 4.15 obsahuje mimo cyklov 7 príkazov a jedno vytvorenie poľa veľkosti $m$, ktoré pozostáva z $m$ príkazov. Prvý cyklus `for riadok in subor_1:` vykoná $n$ opakovaní a v každom dva príkazy. Druhý cyklus `for riadok in subor_2:` vykoná ďalších $n$ opakovaní a v každom najviac 4 príkazy. Spolu teda program 4.15 vykoná najviac $6n + m + 7$ príkazov. To je rádovo $n$, avšak ak je $n$ malé, tak ho preváži hodnota $m$.

### 4.6.3 Riešenie pomocou usporiadania

Podobne, ako pri úlohe 3.1, aj tu nám vie pomôcť usporiadanie si poľa. Keď máme zoznamy čísel zo súborov utriedené vzostupne, stačí nám nimi súbežne prechádzať. Zakaždým sa posunieme v tom zozname, kde sa nachádzame na menšom čísle. Pokiaľ sa dostaneme na rovnaké číslo v oboch zoznamoch, tak ho započítame a posunieme sa o krok ďalej v oboch zoznamoch.

Určiť časovú zložitosť programu 4.16 bude o niečo náročnejšie, keďže sa tu vyskytuje cyklus while. Koľkokrát sa tento cyklus zopakuje? Môžeme si všimnúť, že jeho vykonávanie riadia dve premenné `i` a `j`. Cyklus sa skončí, keď obe tieto premenné

```
nazov_suborov = input('Zadaj nazov suborov: ')
subor_1 = open(nazov_suborov + 'a.txt')
subor_2 = open(nazov_suborov + 'b.txt')
vystup = open(nazov_suborov + 'v.txt', 'w')
cisla1 = [int(riadok) for riadok in subor_1]
cisla2 = [int(riadok) for riadok in subor_2]
cisla1.sort()
cisla2.sort()
i = 0
j = 0
pocet = 0
while i < len(cisla1) and j < len(cisla2):
    if cisla1[i] < cisla2[j]:
        i += 1
    elif cisla1[i] > cisla2[j]:
        j += 1
    else:
        print(cisla1[i], file=vystup)
        i += 1
        j += 1
        pocet += 1
print('Na oboch konferenciach sa zucastnilo', pocet, 'zamestnancov.')
```

Program 4.16: Riešenie pomocou usporiadania zoznamov

dosiahnu hodnotu $n$, teda počet čísel v jednom súbore. Uvedomme si, že v každom opakovaní cyklu sa aspoň jedna z premenných `i`, `j` zvýši o 1. To znamená, že cyklus while sa môže vykonať najviac $2n$-krát. V jednom opakovaní vykoná najviac 6 príkazov, čo dáva spolu $12n$ príkazov pre cyklus while.

Mimo cyklu while máme 8 príkazov, dve načítania zo súboru do poľa obsahujúce $n$ príkazov a dve usporiadania poľa pozostávajúce z rádovo $n \log n$ príkazov. Spolu tak dostávame, že program vykoná približne $2n \log n + 14n + 8$. Logaritmus pri základe 2 okolo čísla $10^6$ je asi 20 a s rastúcim $n$ bude rásť. Preto člen $2n \log n$ bude pre veľké $n$ väčší ako $14n$. Tak môžeme člen $14n$ zanedbať, čím dostaneme rádovo $n \log n$ príkazov.

## 4.7 Súvislý úsek s najväčším súčtom

**Úloha 4.7.** Zmrzlinárka Zuzka si chce otvoriť stánok so zmrzlinou. Jej kamarát je špičkový odborník v ekonomike a vypočítal jej pre každý z $n$ nasledujúcich dní, koľko by zarobila alebo stratila, ak by mala v ten deň otvorený stánok. Ak však už raz stánok zatvorí, tak ho nemôže otvoriť znovu. Preto chce starostlivo vybrať čas (súvislý úsek dní), kedy bude mať otvorený stánok. Chce si predsa zarobiť čo najviac.

Napíšte pre Zuzku program, ktorý načíta postupnosť $n$ celých čísel. Každé číslo reprezentuje zárobok v daný deň; v prípade, že číslo je záporné, tak ide o stratu. Program nájde, koľko najviac si vie Zuzka zarobiť. (napríklad pri postupnosti $-24$, $-17$, 47, $-10$, 42, 18, $-2$, $-3$, 1, $-25$, $-3$, 14 si vie zarobiť najviac 97, a to od 3. po 6. deň.)

Program vypíše najväčší možný súčet, ktorý možno dostať sčítaním súvislého úseku postupnosti.

**Požiadavky na študentov.** Študent vie

- získať číselný vstup od užívateľa a prezentovať mu výstup;
- identifikovať opakujúci sa vzor a zostaviť for cyklus sledujúci tento vzor;
- použiť podmienku v kombinácii s cyklom;
- využiť premennú na postupnú aktualizáciu výsledku programu počas cyklu;
- nájsť najväčší prvok v poli;
- zistiť súčet prvkov v poli;
- rozdeliť si problém na menšie, čiastkové problémy;
- spojiť riešenie viacerých čiastkových problémov (počítanie súčtu, hľadanie najväčšieho prvku) do väčšieho programu.

**Metodické poznámky.** Túto úlohu možno vyriešiť viacerými programami s rôznou časovou zložitosťou. Ide dokonca aj o príklad úlohy, kde môžeme nájsť kubickú zložitosť. Úloha vie byť pre maturantov príliš náročná. Jej riešenie v lineárnom čase si dokonca vyžaduje náročnú algoritmickú myšlienku. Úloha je vhodná pre študentov, ktorí sa venujú informatickým súťažiam.

### 4.7.1 Kubické riešenie

Priamočiare riešenie je vyskúšať všetky súvislé úseky. To znamená, skúsiť pre každú pozíciu začiatku úseku všetky možné pozície konca a pre každý súvislý úsek určený začiatkom a koncom určiť jeho súčet. Ak je súčet väčší ako doposiaľ nájdený maximálny súčet, tak si aktualizujeme údaje o maximálnom súčte.

```
1  n = int(input())
2  pole = []
3  for i in range(n):
4      pole.append(int(input()))
5  maximum = -10**9
6  for zaciatok in range(n):
7      for koniec in range(zaciatok, n):
8          sucet = 0
9          for i in range(zaciatok, koniec + 1):
10             sucet += pole[i]
11         if sucet > maximum:
12             maximum = sucet
13             max_zac = zaciatok
14             max_kon = koniec
15 print('Zuznka si moze zarobit najviac', maximum)
```

```
16 print('Stanok si ma otvorit od dna', max_zac + 1, 'po den', max_kon + 1)
```

Program 4.17: Riešenie počítaním súčtu každého súvislého úseku

Odhad časovej zložitosti je pri tejto úlohe o niečo náročnejší. Najprv ukážeme veľmi hrubý odhad. Program obsahuje tri vnorené cykly. Cyklus `for i in range(zaciatok, koniec + 1):` sa vykoná najviac $n$-krát a opakuje sa v ňom jeden príkaz. V cykle `for koniec in range(zaciatok, n):` sa nám okrem spomenutých $n$ príkazov vykoná najviac 5 ďalších príkazov. Týchto $n+5$ príkazov sa nám zopakuje opäť najviac $n$-krát. Máme zatiaľ odhad $n(n+5)$ príkazov. Tie sa nám ešte opakujú v cykle `for zaciatok in range(n):` s práve $n$ opakovaniami. Dostávame tak, že v programe 4.17 sa vykoná najviac $n^2(n+5)+5$ príkazov, čo je rádovo $n^3$ príkazov.

S takto hrubým odhadom sa môžeme uspokojiť. Môžeme sa však presvedčiť o tom, že náš odhad je rádovo presný tým, že urobíme podobný odhad zdola – teda odhadneme aspoň koľko príkazov sa v programe isto vykoná. Budeme uvažovať len tie opakovania, kde $zaciatok < n/2$. Tých je približne $n/2$. Pre každé z nich budeme v cykle `for koniec in range(zaciatok, n):` uvažovať tie opakovania, kde platí $koniec > zaciatok + n/2$. Keďže $zaciatok + n/2 < n/2 + n/2 = n$, tak týchto opakovaní bude aspoň $n/2$. V týchto vybraných opakovaniach prvých dvoch cyklov nám platí $koniec - zaciatok \geq n/2$. To znamená, že v každom takomto opakovaní bude mať cyklus `for i in range(zaciatok, koniec + 1):` aspoň $n/2$ opakovaní. Dostali sme tak, že každý z troch cyklov spraví aspoň $n/2$ opakovaní. Dohromady teda vieme povedať, že program 4.17 vykoná aspoň $(n/2)^3 = n^3/8$ príkazov. Vidíme, že opäť sme dostali rádovo kubickú časovú zložitosť.

## Kvadratické riešenie – priebežné počítanie súčtu

Predchádzajúce riešenie možno pomerne priamočiaro vylepšiť. Nie je totiž potrebné počítať súčet každý raz znovu a znovu nanovo, keď zväčšíme skúmaný úsek o jedno miesto. Stačí nám k predošlému súčtu úseku pripočítať pridané číslo.

```
1  n = int(input())
2  pole = []
3  for i in range(n):
4      pole.append(int(input()))
5  maximum = -10**9
6  for zaciatok in range(n):
7      sucet = 0
8      for koniec in range(zaciatok, n):
9          sucet += pole[koniec]
10         if sucet > maximum:
11             maximum = sucet
12             max_zac = zaciatok
13             max_kon = koniec
14 print('Zuznka si moze zarobit najviac', maximum)
15 print('Stanok si ma otvorit od dna', max_zac + 1, 'po den', max_kon + 1)
```

Program 4.18: Riešenie pomocou priebežného počítania súčtu

Časovú zložitosť programu 4.18 môžeme určiť podobným spôsobom ako pri programe 4.17, teda horným a dolným odhadom. Druhým spôsobom je určiť presne počet príkazov nasledovne: Cyklus `for koniec in range(zaciatok, n):` obsahuje 5 príkazov. Počet opakovaní tohto cyklu závisí od hodnoty premennej *zaciatok*. Celkovo sa vykoná $n+(n-1)+(n-2)+\cdots+2+1 = n(n+1)/2$ opakovaní cyklu `for koniec in range(zaciatok, n):`. Mimo cyklov sa vykoná rádovo $n$ príkazov, čo je zanedbateľné. Spolu sa teda program 4.18 vykoná rádovo $n^2$ príkazov.

## Lineárne riešenie – prefixové súčty

Túto úlohu možno vyriešiť ešte rýchlejšie. Aby sme vedeli rýchlo počítať súčty súvislých úsekov poľa `pole`, využijeme koncept známy pod menom *prefixové súčty*. Vytvoríme si pole `prefix`, ktoré bude obsahovať na $(i+1)$-vej pozícii hodnotu $prefix[i+1] = = pole[0] + pole[1] + \cdots + pole[i]$. Súčet súvislého úseku od *zaciatok* do *koniec* vieme potom určiť ako

$$pole[zaciatok] + pole[zaciatok+1] + \ldots + pole[koniec] = \\ = prefix[koniec] - prefix[zaciatok-1].$$

Hodnota $prefix[koniec] - prefix[zaciatok-1]$ bude pritom maximálna vtedy, keď $prefix[koniec]$ bude mať najväčšiu možnú hodnotu a $prefix[zaciatok-1]$ bude mať najmenšiu možnú hodnotu. Teda nám stačí len určiť maximálnu a minimálnu hodnotu v poli prefixových súčtov.

```
n = int(input())
pole = []
prefix = [0]
for i in range(n):
    pole.append(int(input()))
    prefix.append(prefix[-1] + pole[-1])
maximum = -10**9
minimum = 10**9
for i in range(n):
    if prefix[i] > maximum:
        maximum = prefix[i]
        max_kon = i
    if prefix[i] < minimum:
        minimum = prefix[i]
        max_zac = i
print('Zuznka si moze zarobit najviac', prefix[max_kon] - prefix[max_zac])
print('Stanok si ma otvorit od dna', max_zac + 1, 'po den', max_kon)
```

Program 4.19: Riešenie pomocou prefixových súčtov

Cyklus zo 4. riadku sa vykoná $n$-krát a v každom opakovaní vykoná dve operácie. Cyklus z 9. riadku sa vykoná tiež $n$-krát a v každom opakovaní vykoná najviac 6 operácií. Spolu teda máme najviac $8n + 7$ operácií, teda časová zložitosť programu 4.19 je rádovo $n$.

## 4.8 Zisťovanie hesla

**Úloha 4.8.** Nasledujúci program načíta celé číslo $n$. Náhodne si vygeneruje $n$-znakové heslo zložené z malých písmen anglickej abecedy. Následne sa ho snaží uhádnuť generovaním všetkých možností.

```
1 n = int(input())
2 abeceda = 'abcdefghijklmnopqrstuvwxyz'
3 heslo = ''
4 for i in range(n):
5     heslo += random.choice(abeceda)
6
7 for pokus in vsetky_slova(abeceda, n):
8     if pokus == heslo:
9         print('Heslo je', pokus)
```

Program 4.20: Genrovanie všetkých hesiel

Pritom cyklus `for pokus in vsetky_slova(abeceda, n):` skúša všetky možné $n$-písmenové slová zložené z malých písmen anglickej abecedy (teda tých, ktoré sú uložené v reťazci `abeceda`). To, ako presne spraviť takýto cyklus, nie je podstatné.

1. Ako dlho bude programu trvať nájsť heslo skladajúce sa z ôsmich znakov?
2. Ako dlho bude programu trvať nájsť heslo skladajúce sa zo šestnástich znakov?
3. Aspoň koľko znakov má mať heslo, aby ho program hľadal aspoň rok?
4. Zvolíme si bezpečnejšie heslo, ktoré môže okrem malých písmen anglickej abecedy obsahovať aj veľké písmená (spolu teda 52 znakov). Ako dlho bude trvať programu uhádnuť 8-znakové slovo teraz? V akom vzťahu očakávate, že bude tento čas v porovnaní s odpoveďou na otázku 1?
5. Možné znaky ešte rozšírime a budeme používať znaky slovenskej abecedy, pomocné symboly ako otáznik, pomlčka a pod. Povedzme, že máme spolu 100 možných znakov. Ako dlho bude teraz trvať vyskúšanie 8-znakových hesiel?

**Požiadavky na študentov.** Študent vie

- získať číselný vstup od užívateľa a prezentovať mu výstup;
- použiť podmienku v kombinácii s cyklom;

**Metodické poznámky.** Táto úloha dáva otázku časovej zložitosti do praktického kontextu. Bezpečnosť mnohých informatických konceptov – ako napríklad hesiel – sa odvíja od skutočnosti, že počítače pracujú s obmedzenou rýchlosťou. Hoci teoreticky sú počítače schopné zistiť heslo do k účtu, trvalo by im to (pri dostatočnom zabezpečení) niekoľko rokov, príp. ešte viac. Študenti si pri nej môžu skúsiť odhadnúť, ako dlho môže trvať prelomenie rôzne silných hesiel.

Pri tejto úlohe sa študenti môžu stretnúť s exponenciálnou časovou zložitosťou a môžu tu spoznať jej veľké nevýhody – ako sa len pri malom náraste vstupu výrazne zvýši čas behu programu. Úskalím tejto úlohy je však to, že skúšanie hesiel využíva pre maturantov príliš pokročilé myšlienky. Treba preto použiť niektorú hotovú funkciu. Príp. možno úlohu modifikovať dodaním informácie, ako dlho počítaču trvá overenie jedného hesla.

## Riešenie

Najprv vypočítame, koľko je $n$-písmenových slov zložených z malých písmen anglickej abecedy. Na každé z $n$ miest si môžeme nezávisle vybrať jedno z 26 písmen malej anglickej abecedy. Spolu teda máme $26^n$ možností pre výber slova (ide o variácie s opakovaním). Cyklus `for pokus in vsetky_slova(abeceda, n):` sa teda vykoná $26^n$-krát. V každom opakovaní sa vykonajú najviac dva príkazy. Pred týmto cyklom máme ešte cyklus `for i in range(n):`, ktorý sa zopakuje $n$-krát. Oproti $26^n$ príkazov ide však o veľmi malé číslo, preto tieto príkazy môžeme zanedbať. Máme tak odhad, že program vykoná približne $26^n$ príkazov. To odpovedá času behu rádovo $26^n/10^7$ s.

Na základe toho vieme odhadnúť, že 8-znakové slová bude program skúšať asi 20 000 s, čo je asi 6 hodín (otázka 1). Podobne dostaneme, že 16-znakové slová bude program skúšať rádovo $4 \cdot 10^{15}$ s, čo je asi sto miliónov rokov (otázka 2).

Ak chceme, aby skúšanie hesla trvalo aspoň rok, čo je rádovo $365 \cdot 24 \cdot 60 \cdot 60 \sim 3 \cdot 10^7$ s, tak musí platiť

$$\begin{aligned} 26^n/10^7 &\geq 3 \cdot 10^7, \\ 26^n &\geq 3 \cdot 10^{14}, \\ n \geq \log_{26} 3 \cdot 10^{14} = \frac{\log 3 \cdot 10^{14}}{\log 26} &\approx 10. \end{aligned}$$

Teda pri 10-znakovom hesle bude program bežať asi rok a pri viac znakových ešte dlhšie.

Keď budeme pracovať s heslami obsahujúce 52 možných znakov, dostaneme spolu $52^n$ možných hesiel, ktoré bude program skúšať. Jeho čas behu bude teda rádovo $52^n/10^7$. Keď dáme ten to čas do pomeru s časom z úlohy 1 pre $n = 8$, dostaneme

$$\frac{52^8/10^7}{26^8/10^7} = \frac{2^8 \cdot 26^8}{26^8} = 2^8 = 256.$$

Teda čas behu narastie až 256-krát. Vychádza to tak rádovo na $5 \cdot 10^6$ sekúnd, čo je asi 60 dní.

Ak máme v úlohe 5 až 100 možných znakov, tak máme $100^n$ možných slov. To zodpovedá času $100^n/10^7$. Z toho pre $n = 8$ dostaneme rádovo $10^9$ s, čo je asi 30 rokov.

Táto úloha ilustruje, aký vplyv na bezpečnosť hesla majú faktory ako jeho dĺžka a množina, z ktorej vyberáme doňho znaky.

V skutočnosti zisťovanie hesiel neprebieha takto jednoducho. Overenie jedného hesla trvá výrazne pomalšie ako kontrola jednej podmienky v Pythone.

# Kapitola 5

# Užitočné metódy

V tejto kapitole uvedieme niekoľko užitočných metód, ktoré nemusia byť známe každému učiteľovi a vedia nám uľahčiť prácu. Prípadne s nimi možno spraviť hodinu prístupnú žiakom, ktorí nepracovali so súbormi.

## 5.1 Meranie času behu programu

Pre názornejšiu prácu s časom behu programu je vhodné použiť nástroj, ktorý čas behu presne odmeria. Meranie času využíva funkciu, ktorá vráti systémový čas – čas od 1. 1. 1970 0:00. Touto funkciou si zistíme čas na začiatku meraného úseku a na konci a následne vypíšme ich rozdiel. Samozrejme, odmerané časy si uložíme do premenných. Pokiaľ načítavame vstup od užívateľa, musíme si dať pozor, aby sme čakanie na vstup od užívateľa nezahrnuli do meraného úseku, čo by nám predĺžilo výsledný čas.

```
1 from time import time
2
3 zaciatok = time()
4 # ...
5 # program
6 # ...
7 koniec = time()
8 print('Program bezal', koniec - zaciatok, 's')
```

Program 5.1: Meranie času v jazyku Python

```
program meranie_casu;
uses DateUtils, sysutils;
var
  zaciatok, koniec: TDateTime;

begin
  zaciatok := Now;
  // ...
  // program
  // ...
  koniec := Now;
  writeln('Program bezal ', MilliSecondsBetween(koniec, zaciatok)/1000:0:3, ' s');
  readln();
end.
```

Program 5.2: Meranie času v jazyku Pascal

```
#include <chrono>

int main() {
        chrono::steady_clock::time_point zaciatok, koniec;
        zaciatok = chrono::steady_clock::now();
        // ...
        // program
        // ...
        koniec = chrono::steady_clock::now();
        long duration = chrono::duration_cast<std::chrono::microseconds>(koniec -
    zaciatok).count();
        std::cout << "Program trval " << duration << " mikrosekund" << std::endl;
```

Program 5.3: Meranie času v jazyku C++

```
long zaciatok = System.nanoTime();
// ...
// program
// ...
long koniec = System.nanoTime();
System.out.print("Program bezal ");
System.out.print((double)(koniec - zaciatok)/1000000000);
System.out.println(" s");
```

Program 5.4: Meranie času v jazyku Java

## 5.2 Presmerovanie vstupu

Mnohé úlohy v našej práci spracovávajú veľa čísel. Aby sme dosiahli beh programu rádovo v sekundách, väčšinou treba zadať tisíce až milióny čísel. To nie je možné spraviť ručne. Takto veľké vstupné údaje musíme prečítať zo súboru. Pokiaľ študenti vedia čítať zo súboru, tak si s tým poradia. Ak nie, existuje spôsob ako sa čítaniu zo súboru vyhnúť. Študenti tým pádom môžu písať programy tak, že budú využívať príkazy na čítanie z konzoly. Presmerovanie vstupu je proces, ktorým zariadime, aby v situácii, keď program má čítať vstup z konzoly, tak si ho prečíta z určeného súboru.

Mnohé programovacie jazyky poskytujú túto možnosť. Väčšinou ide o niekoľko príkazov, ktoré stačí dať na začiatok programu a iba v nich upraviť názov súboru. Pri používaní presmerovania nie je potrebné, aby žiaci rozumeli, čo presne použité príkazy robia. Vedia ich brať, ako „čarovnú formulku“, ktorá presmeruje čítanie vstupu z konzoly do súboru. Dôležité je však vedieť, že súbor musí byť v rovnakom priečinku ako spúšťaný program, pokiaľ používame relatívnu cestu k súboru. Pri niektorých programátorských prostrediach je ťažké odsledovať, na ktorom mieste sa program nachádza. Nemusí sa to totiž nachádzať v rovnakom priečinku ako zdrojový kód. Preto vtedy odporúčame použiť absolútnu cestu.

## Python

```
1 import sys
2 sys.stdin = open('subor.txt')
```

Miesto, z ktorého sa číta štandardne vstup, je v Pythone určené premennou `stdin` (skratka zo standard input) z modulu `sys`. Prepísaním hodnoty tejto premennej na súbor zariadime, že program bude vstup čítať zo súboru. Výhodou Pythonu je, že sa nekompiluje do spustiteľného súboru, ale rovno sa interpretuje. Súbor so vstupom teda stačí mať v rovnakom priečinku ako zdrojový súbor s koncovkou .py.

Ďalšou výhodou je, že ak potrebujeme v ceste k súboru ísť do iného priečinka, tak môžeme medzi priečinkami používať znaky \ aj /. Treba si však dať pozor na to, že znak \ je súčasťou mnohých systémových znakov (napr. \n pre koniec riadku), a preto ho treba v reťazcoch písať ako \\.

Žiakov treba upozorniť na to, aby príkazy na presmerovanie súboru dali naozaj na začiatok svojho programu. Pri skúšaní hodín sa nám stávalo, že žiaci presmerovali vstup až po načítaní prvého údaju, čo viedlo k nefungujúcim programom.

## C++

Premserovanie vstupu v jazyku C++ je veľmi priamočiare. Stačí pridať jeden príkaz. Odporúčame používať absolútnu cestu, lebo nie v každom vývojovom prostredí je jasné, z ktorej cesty sa program spúšťa. Pri programovaní vo Windowse musíme miesto spätných lomiek \ použiť dvojité spätné lomky \\.

```
1 freopen("C:\\Users\\Jano\\programy\\vstup.txt","r",stdin);
```

## Pascal

Pri presmerovaní vstupu v Pascale musíme vykonať viacero krokov. Po prvé, samotný proces otvárania súboru pozostáva z viacerých príkazov, ktoré potrebujeme umiestniť na začiatok hlavného programu. Po druhé, pokiaľ chceme mať na konci programu

príkaz `readln();`, aby sa konzolový program hneď neskončil, tak potrebujeme presmerovanie vstupu vrátiť. Preto si potrebujeme odložiť pôvodnú hodnotu premennej `Input` (reprezentujúcu konzolu) do premennej, z ktorej premennú `Input` na konci programu obnovíme.

Tiež je bezpečnejšie použiť absolútnu cestu. Tá má formát rovnaký ako v jazyku C++.

```
var
  subor, staryvstup: TextFile;

begin
  AssignFile(subor, 'C:\\Users\\Jano\\programy\\vstup.txt');
  Reset(subor);
  staryvstup := Input;
  Input := subor;
  // ...
  // program
  // ...
  Input := staryvstup;
  readln();
end.
```

## Java

Presmerovanie vstupu v Jave prebieha pomocou metódy `setIn` triedy `System`. Metóda berie ako argument inštanciu triedy `FileInputStream`, do ktorej konštruktora dáme cestu k súboru. Túto triedu musíme importovať tým, že navrch zdrojového súboru pridáme `import java.io.FileInputStream;`. V Jave odporúčame používať absolútnu cestu, keďže projekty v Jave väčšinou pozostávajú z viacerých priečinkov a je náročné pamätať si, do ktorého priečinku treba umiestniť súbory.

V prípade, keď zadaný súbor neexistuje, konštruktor triedy `FileInputStream` hádže výnimku `FileNotFoundException`. To je problém, lebo spracovávanie výnimiek študenti nemusia vedieť. Aby sme nemuseli pripisovať do hlavičky metódy `main` deklaráciu, že vyhadzuje výnimku, tak výnimku odchytíme try-catch blokom. Takto dostaneme samostatne použiteľný kus kódu, ktorý stačí len pridať na začiatok programu.

Odporúčame do catch bloku dať najvšeobecnejšiu výnimku `Exception`. Pri testovaní sa nám totiž stalo, že niektorým žiakom hádzalo inú výnimku a toto prepísanie problém vyriešilo.

```
import java.io.FileInputStream;

// ...

try {
        System.setIn(new FileInputStream("C:\\Users\\Jano\\programy\\vstup.txt"));
} catch (Exception e) {
        System.err.println("Subor so vstupom nebol najdeny. Nacitavam z konzoly.");
}
```

# Kapitola 6

# Testovanie

Metodiku sme vypracovali na základe skúseností z učenia na gymnáziu Grösslingová 18 v Bratislave. Spočiatku bola založená na úlohe 4.6. Túto úlohu sme riešili so študentmi septimy na dvojhodinovke seminára z informatiky a so študentmi sexty na dvojhodinovke informatiky. V oboch prípadoch išlo o triedy s rozšíreným vyučovaním matematiky a informatiky. Úlohu riešili v rámci témy práca zo súbormi.

Na hodine v septime študenti riešili úlohu sami, pričom sme im radili, keď potrebovali s niečím pomôcť. Väčšina študentov naprogramovala riešenie A. Po jeho spoločnom vysvetlení sme analyzovali jeho časovú zložitosť. Do diskusie sa študenti zapájali. Pri diskusii, ako by sa dala úloha vyriešiť rýchlejšie, žiaci prišli s riešením C pomocou usporiadania. V závere hodiny sme si so študentmi prešli hlavnú myšlienku riešenia B.

V sexte všetci študenti naprogramovali riešenie A, tiež samostatne s našimi usmerneniami. Potom sme v diskusii so študentmi odhadovali čas behu ich programu. Študenti sa zapájali. Pri diskusii, ako by sa dal program zrýchliť, navrhovali len drobné vylepšenia, ktoré nemenili kvadratickú zložitosť (pridanie príkazu break, opakované prechádzanie súborov miesto uloženia údajov do poľa). Pre tieto riešenia sme odhadli, že stále budú mať rádovo kvadratický počet príkazov, preto sme ich ani neprogramovali.

Pre nedostatok času už nezvýšil čas na ukázanie rýchleho riešenia. Jeden študent sa spýtal otázku, ako by sa dali skontrolovať aj veľké súbory. Študentom sme ponechali priestor, nech môžu nad tým sami pouvažovať a na ďalšej hodine sme si spolu s nimi prešli a zanalyzovali riešenie B.

## 6.1 Testovanie na inej škole

Testovanie bolo uskutočnené na Gymnáziu Ľudovíta Štúra v Trenčíne na seminári z informatiky pre 3. ročník v informatickej triede. Pri komunikácii s učiteľkou sme zistili, že žiaci v tom čase ešte neovládali prácu so súbormi. Z toho dôvodu sme vymenili úlohu 4.6 za 3.1. Dostali sme tak do metodiky úlohu, pre ktorú nie je nutné ovládať prácu

so súbormi a ktorá je jednoduchšia tým, že obsahuje len jeden zoznam údajov. Načítavanie veľkých zoznamov sme riešili pomocou presmerovania. Študenti programovali v jazyku Java.

V prvej časti hodiny študenti programovali samostatne, s miernym usmernením tých, ktorí potrebovali. Časté otázky a problémy študentov boli v súvislosti s presmerovaním štandardného vstupu z konzoly do súboru. Všetkým študentom sa podarilo naprogramovať riešenie A s kvadratickou zložitosťou. Študenti nemali problémy s vnoreným cyklom. Jedna študentka naprogramovala aj riešenie B pomocou počítania výskytov.

Najprv sme si so študentmi prešli odhad časovej zložitosti pre riešenie A. Potom študentka, ktorá mala naprogramované riešenie B, vysvetlila ostatným svoje riešenie. Následne ostatní študenti spoločne s našimi pomocnými otázkami určili časovú zložitosť riešenia B.

Celých prvých 45 minút sme venovali programovaniu a začiatok druhej hodiny sme venovali zhrnutiu riešenia. Na konci druhej hodiny nám nezostal čas na precvičenie. Po hodine sme zhodnotili, že priestor na precvičenie na dvojhodinovke nie je, pokiaľ chceme dať žiakom priestor na vyriešenia a naprogramovanie úlohy a taktiež priestor na analýzu a porovnanie viacerých riešení.

Konštatujeme, že čas na riešenie môžeme skrátiť. Nemusíme čakať, dokým všetci napíšu odladený program. Budeme čakať len dovtedy, kým všetci študenti nemajú kostru programu a potom požiadame, nech si žiaci doladia chyby po dvojiciach. Popri tom pomôžeme tým, čo nezvládnu ich programy doladiť ani za pomoci suseda. Na základe toho sme čas na riešenie úlohy v našej metodike skrátili.

## 6.2 Druhé testovanie

Druhé testovanie sme uskutočnili na Piaristickom gymnáziu Jozefa Braneckého v Trenčíne na tretiackom seminári z informatiky všeobecného zamerania. Hodina prebiehala online cez videokonferenciu. Študenti v tom čase nemali ešte iné programátorské zručnosti ako tie, ktoré sa uvádzajú v inovovanom štátnom vzdelávacom programe pre informatiku [16] – boli na konci učebnice Programujeme v Pythone [7].

Týmto testovaním sme chceli vyskúšať metodiku v náročnejšom prostredí, kde žiaci nie sú veľmi skúsení v programovaní. Preto sme aj pridali do našej sady úloh úlohu 4.1. Predpoklady úlohy 4.1 študenti spĺňali. Vďaka tejto úlohe môže našu metodiku využiť učitelia aj so študentami, ktorí neovládajú polia.

Študentom sme nechali najprv 10 minút nad úlohou popremýšľať samostatne. Potom sa ukázalo, že študenti vôbec nevedia, ako začať, tak sme prešli na spoločné riešenie úlohy. Rozdelili sme si jej riešenie a programovanie na viacero častí. V každej časti sme

komunikovali s iným študentom a navádzali sme ho pomocnými otázkami.

1. Zobrali sme si konkrétny vstup ($n = 40$) a najprv bez programovania študenti povedali riešenie.
2. Stanovili sme si, že chceme vyskúšať všetky čísla od 1 do $n$ – všetky možné počty centov, ktoré môže dostať jeden účastník večierka.
3. Napísali sme cyklus, ktorý pre každé číslo $i$ od 1 do $n$ vypísal podiel $n/i$.
4. Upravili sme program tak, aby vypisoval podiel len vtedy, ak je celočíselný.
5. Upravili sme program tak, aby vypisoval podiel len vtedy, keď je nepárny.
6. Upravili sme program, aby sa po vypísaní prvého čísla zastavil. Prvé vypísané číslo zodpovedaná najmenšej možnej hodnote jedného diela výhry, čo teda koreš-ponduje s najväčším možným počtom zúčastnených.

Potom sme pristúpili k diskusii o časovej zložitosti, ako sme uviedli v kapitole 3. Museli sme do nej pridať aj diskusiu o najhoršom prípade a odhalenie, že najhorší prípad predstavujú mocniny dvojky. Odhadli sme, ako dlho by program bežal pri zadaní $n = 2^{30}$. Kvôli nedostatku času sme sa nedostali k iným riešeniam úlohy, ani k precvičeniu si analýzy časovej zložitosti na inom programe.

Po hodine sme zaslali žiakom krátky dotazník, ktorý vyplnilo 7 z 10 zúčastnených študentov. Uvádzame znenie jeho otázok. K uzavretým otázkam uvádzame aj štatistiku odpovedí v tabuľke 6.1.

1. Ako bola pre Teba téma hodiny zaujímavá (1 – nudná, 5 – fascinovala ma)?
2. Ako bola pre Teba téma náročná na pochopenie (1 – strácal(a) som sa, 5 – pohodička)?
3. Pokladáš tému, o ktorej sme sa bavili, za dôležitú? (1 – načo mi bude, 5 – dôležitá)
4. Chceš vyzdvihnúť na hodine alebo téme niektoré veci, ktoré sa Ti páčili?
5. Ktoré veci sa Ti na hodine alebo téme nepáčili? Bolo niečo pre Teba nezrozumiteľné?

Z priebehu hodiny usudzujeme, že úloha 4.1, ktorú sme vybrali, nebola pre študentov známa. Z toho dôvodu sme strávili pomerne veľa času jej riešením. Celú dvojhodinovku sme sa tak venovali vlastne iba jednému riešeniu tejto úlohy, čo bolo monotónne. To spomenul aj jeden študent v dotazníku. Preto je dobré vybrať pre analýzu typovo takú úlohu, ktorú študenti poznajú. Pokiaľ sa tak nestane, tak vo fáze riešenia úlohy ju odporúčame riešiť spolu so študentmi.

| Otázka | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1. Ako bola pre Teba téma hodiny zaujímavá? | 0 | 0 | 2 | 4 | 1 |
| 2. Ako bola pre Teba téma náročná na pochopenie? | 0 | 1 | 1 | 3 | 2 |
| 3. Pokladáš tému, o ktorej sme sa bavili, za dôležitú? | 0 | 0 | 4 | 2 | 1 |

Tabuľka 6.1: Počet jednotlivých odpovedí na prvé tri otázky dotazníka

# Záver

V práci sme zhromaždili niekoľko programátorských úloh, ktoré poskytujú podklady pre analýzu časovej zložitosti programov. Väčšinu z úloh možno vyriešiť rôznymi spôsobmi s rôznou časovou zložitosťou. Naprieč všetkými úlohami sa možno stretnúť s najbežnejšími triedami časovej zložitosti.

Spomedzi týchto úloh sme vybrali jednu, ku ktorej sme pripravili metodiku, ako možno na hodinách so študentmi stredných škôl pripravujúcich sa na maturitu riešiť otázky týkajúce sa časovej zložitosti. Túto metodiku sme otestovali pri výučbe vlastných žiakov a aj v dvoch triedach na iných školách.

Hoci je výučba časovej zložitosti na stredných školách náročná, o čom sme sa presvedčili aj pri písaní práce, aj pri testovaní; konštatujeme, že jej výučba u maturantov je možná. Naša práca spracováva túto tému názornejšie a prístupnejšie pre maturantov. Pre relevantnejšie zhodnotenie efektívnosti našej metodiky by bolo potrebné spraviť rozsiahlejšie testovania v triedach.

Naša práca taktiež ponecháva priestor pre ďalší výskum v tejto oblasti. Možno skúmať, či a ako sa dá vyučovať časová zložitosť na iných typoch úloh, ako sme ich opísali v kapitole 2 – na úlohách o vypisovaní hviezdičiek a na neprogramovacích úlohách.

# Obsah

# Literatúra


[1] Veselý a kol. *Programátorské kuchařky*. TMATFYZPRESS„ 2011. ISBN: 78-80-7378-181-1.

[2] Andrej Blaho. Algoritmy a dátové štruktúry (stránka predmetu). On line. získané 8. 6. 2020 z adresy `http://struct.input.sk/01.html`.

[3] Geeks for Geeks. How is the time complexity of sieve of eratosthenes is $n \cdot \log(\log(n))$. On line. získané 8. 6. 2020 z adresy https://www.geeksforgeeks.org/how-is-the-time-complexity-of-sieve-of-eratosthenes-is-nloglogn/.

[4] Michal Forišek. Early beta verzia skrípt z adŠ. On line. získané 8. 6. 2020 z adresy `http://foja.dcs.fmph.uniba.sk/ads/skripta/02zlozitost.pdf`.

[5] Judith Gal-Ezer and Ela Zur. The concept of "algorithm efficiency" in the high school cs curriculum. 2002.

[6] Nitra Gymnázium, Párovská 1. Mt spravnosý algoritmu, vypoctova zlozitosý a efektıvnosý algoritmu a programu. On line. získané 8. 6. 2020 z adresy `http://www.gymparnr.edu.sk/obsah/predmety/subory/informatika/spravnost_zlozitost.pdf`.

[7] Peter Kučera. *Programujeme v Pythone*. 2016. ISBN: 978-80-972320-4-7.

[8] Korešpondenčný matematický seminár. 8. úloha 1. zimnej série 2013/2017. On line. získané 8. 6. 2020 z adresy `https://www.ksp.sk/ulohy/zadania/1450/`.

[9] Stanislav Palúch. Algoritmy a ich zložitosť. On line. získané 8. 6. 2020 z adresy `https://frcatel.fri.uniza.sk/users/paluch/Prezentacie/GrafPrez_02.pdf`.

[10] Jaroslav Výbošťok Peter Kučera. *Programujeme v Pythone 2*. 2017. ISBN: 978-80-972779-1-8.

[11] Korešpondenčný seminár z programovania. Úvod do teórie časovej zložitosti. On line. získané 25. 10. 2018 z adresy `https://www.ksp.sk/kucharka/zlozitost1/`.

[12] Korešpondenčný seminár z programovania. Čo je to časová zložitosť? Online. získané 25. 10. 2018 z adresy `https://www.ksp.sk/kucharka/zlozitost0/`.

[13] Korešpondenčný seminár z programovania. Črieda pažravých vedúcich, 1. úloha 2. kola 1. časti 35. ročníka. On line. získané 8. 6. 2020 z adresy `https://www.ksp.sk/ulohy/zadania/1450/`.

[14] Ján Guniš Valentína Gunišová. Niekoľko slov o efektivite algoritmov. On line. získané 8. 6. 2020 z adresy `https://di.ics.upjs.sk/informatika_na_zs_ss/studijny_material/programovanie/teoria/efektivi.pdf`.

[15] Gymnázium Ľudovíta Štúra Zvolen. Výpočtová zložitosť. On line. získané 8. 6. 2020 z adresy `http://www.gymzv.sk/~vyuka/inf_pro/vyp_zlozitost/vyp_zlozit_na_web.htm`.

[16] Štátny pedagogický ústav. Inovovaný štátny vzdelávací program – informatika – gymnázium so štvorročným a päťročným vzdelávacím programom, 2015. Online, získané 25. 10. 2018 z adresy `https://www.statpedu.sk/files/articles/dokumenty/inovovany-statny-vzdelavaci-program/informatika_g_4_5_r.pdf`.

[17] Štátny pedagogický ústav. Inovovaný štátny vzdelávací program – matematika – gymnázium so štvorročným a päťročným vzdelávacím programom, 2015. Online, získané 25. 10. 2018 z adresy `https://www.statpedu.sk/files/articles/dokumenty/inovovany-statny-vzdelavaci-program/matematika_g_4_5_r.pdf`.

[18] Štátny pedagogický ústav. Cieľové požiadavky na vedomosti a zručnosti maturantov z informatiky, 2016. Online, získané 25. 10. 2018 z adresy `www.statpedu.sk/files/articles/nove_dokumenty/cieloQve-poziadavky-pre-mat-skusky/informatika.pdf`.

[19] Štátny pedagogický ústav. Cieľové požiadavky na vedomosti a zručnosti maturantov z matematiky, 2016. Online, získané 25. 10. 2018 z adresy `https://www.statpedu.sk/files/sk/svp/maturitne-skusky/platne-od-sk-r-2018/2019/cp_matematika_2019.pdf`.

[20] NÚCEM Štátny pedagogický ústav. Monitor 2004, pilotné testovanie maturantov, informatika, test i-1. On line. získané 8. 6. 2020 z adresy `https://www.nucem.sk/dl/3300/11_Test_I-1.pdf`.

# Príloha

V prílohe pripájame:

- programy z kapitol 3 a 4;
- zadanie a vstupné súbory pre žiakov spomínané v kapitole 3;
- programy pre generovanie vstupných súborov k úlohám 3.1, 4.4 a 4.6.