Doprovodný soubor k zápočtovému programu

Uživatelská dokumentace

Jedná se o program, který implementuje a, po uživatelském vstupu, vyhodnotí základní maticové operace.

Byl vyvinut v programovacím jazyce Python verze 3.9.0. Není potřeba žádná externí knihovna.

Příkazy

- *matrix* – pro tvorbu matice
- *show* – vypíše všechny dosud vytvořené matice
- *exit* – zastaví program

Operace

Operace zadáváme v upraveném prefixovém zápisu – *operace(X)* pro unární a *operace(X, Y)* pro binární operaci – je nutné oddělit členy binární operace čárkou.

Unární: *rank*, *ref*, *rref*, *inverse*, *lu*, *transpose*

Binární: *+*, *-*, ***

Návod k užívání programu

Po spuštění programu se vypíše shrnutý návod k použití.

```
Type 'matrix' to create one, type 'show' to know what you've created and 'exit' to stop the program.

To apply binary operation, type 'name_of_binary_operation(X, Y)'
To apply unary operation, type 'name_of_unary_operation(X)'

You can choose from these operations {+, -, *, rank, ref, rref, inverse, lu, transpose}

Example:
'matrix' ... A = [ 1 2 ]  =>  '+(A, A)'  =>  B = [ 2 4 ]
```

Vypišme příkaz *matrix*, čímž se dostaneme do dialogu, který nám ji pomůže vytvořit.

```
matrix
Enter dimensions:
m =
```

Nejprve zadejme legální dimenze své matice - tedy přirozená čísla (*m* pro počet řádků, *n* pro počet sloupců matice).

```
m = 3
n = 2
[ - - ] enter values:
```

Poté zadejme číselné hodnoty pro každý řádek. Po správném ukončení dialogu bychom měli vidět naši matici s přiřazeným písmenem.

```
[ - - ] enter values: 1 2
[ - - ] enter values: 3 4
[ - - ] enter values: 5 6

    [ 1 2 ]
A = [ 3 4 ]
    [ 5 6 ]
```

Řekněme, že chceme vytvořit matici B, která se rovná transponované A. Vypišme tedy unární operaci *transpose* na matici A.

```
transpose(A)

B = [ 1 3 5 ]
    [ 2 4 6 ]
```

Nyní proveďme maticový součin mezi maticí A vynásobenou skalárem 2 a maticí B.

```
*(*(2, A), B)

    [ 10 22  34 ]
C = [ 22 50  78 ]
    [ 34 78 122 ]
```

Pro ukončení programu vypišme *exit*.

Programátorská dokumentace

Program je rozdělen do tří tříd – Matrix, Operations, InputLoop.
Vytvořené matice jsou ukládány do datové struktury *dictionary* pod svým písmenem a počet vytvořených matic se zapisuje do proměnné *count*.
Jediná externě importovaná funkce je ze základní knihovny *fractions* – funkce Fraction.

class Matrix

- objekt definován počtem řádků, sloupců, seznamovou reprezentací a popř. jménem

class Operations

- zde jsou uloženy zmíněné algoritmy pro maticové operace a jejich pomocné funkce

Addition (pro matice *m×n* je časová i prostorová složitost *O(m*n)*) – Vezme dva objekty a znaménko operace (+ nebo -). Zkontroluje, jestli jsou oba objekty maticí – pokud jsou – zkontroluje, jestli mají správné rozměry (pokud nejsou, vrátí error zprávu), zkopíruje seznamovou reprezentaci první matice a ke každému prvku přičte prvek (ze stejných indexů) z matice druhé. Vrátí objekt *Matrix* s nově vytvořeným seznamem. Pokud oba objekty nejsou maticí, ale jsou numerickou hodnotu, vrátím jejich součet.

Multiplication (pro matice *m×k* a *k×n* je časová složitost *O(m*n*k)* a prostorová *O(m*n)*) – Vezme dva objekty, zjistí, jestli jsou oba matice (pokud je jeden matice a druhý numerická hodnota, vrátím matici vynásobenou skalárem; pokud jsou oba numerickými hodnotami, vynásobím je mezi sebou) a zkontroluje, jestli mají vhodné rozměry – pokud nemají, vrátím error zprávu.
Ve třech cyklech postupně přiděluji sumy, dle definice maticového součinu, na místa prvků ve výsledné matici, kterou pak vrátím jako *Matrix* objekt.
Nepoužil jsem Strassenův algoritmus, který je asymptoticky lepší, protože pro malé rozměry matic je pomalejší.

REF (pro matici *m×n* je časová složitost *O(m*n*min(m, n))*) – Na vstupu dostane objekt *Matrix*, popř. *int expand* (s hodnotou 2, kdyby matice byla v rozšířeném tvaru) anebo *boolean* hodnotu *lu* (*True*, pokud je matice rozkládána). Zavede indexy *i, j=0, 0*, které určují pozici prvku a *r=0* pro rank matice.
Dokud je *i* menší než počet řádků matice a *j* menší než počet sloupců, najde pivot a přeskočí nulové podsloupečky pomocí funkce *_skip*, která prohodí řádek, ve kterém je pivot, a přiřadí proměnné *j* číslo sloupce, ve kterém pivot našla (pokud ho nenajde, vrátí hodnotu *-1*). Poté každému řádku pod pivotem nuluje prvky ve stejném sloupci pomocí funkce *_reduce*. Přičteme proměnným *i*, *j*, *r* jedničku (posuneme se o řádek dolů a sloupec doprava a zvýšíme hodnotu rank) – pak se vrací na začátek *while* cyklu. Po skončení cyklu vrátí *tuple* (objekt *Matrix*, proměnná *r*).

RREF (pro matici *m×n* je časová složitost *O(m*n*min(m, n))*) – Na vstupu dostane objekt *Matrix*, popř. *boolean* hodnotu *inREF* (*True* pokud je v odstupňovaném tvaru).

Funguje podobně jako funkce *REF*, ale pivot je vydělen tak, aby vznikla jednička a použije funkci *_reduce* jak na řádky pod pivotem, tak i nad pivotem. Pokud je *inREF=True*, tak redukuje pouze řádky nad pivotem. Vrací objekt *Matrix*.

Inversion (časová složitost *O(n^3)*) – Dostane objekt Matrix. Provede operaci REF na jejím rozšířeném tvaru (nastaví *expand=2*) a uloží si její rank – díky němu zkontroluje, jestli má matice plnou hodnost. Poté pustí operaci RREF s tím, že nastaví *boolean inREF* na *True*. Vrátí druhou půlku rozšířené matice.

LU decomposition (časová složitost *O(n^3)*) – Dostane *Matrix*, zkontroluje rozměry. Pustí *REF* na její rozšířený tvar s tím, že nastaví *lu=True*. Vrátí *tuple* objektů *Matrix* (levá strana rozšířené matice - lower triangular, pravá – upper triangular).
Funkce funguje pouze pro regulární matice.

Transposition (časová složitost *O(m*n)*) – Dostane *Matrix*. Vytvoří prázdný seznam *c*, do kterého *n*-krát přidá další seznam, kde *m*-krát použije *append* pro prvky ze sloupce, který zrovna „převrací v řádek.“ Vrátí *Matrix* s novou maticovou reprezentací *c* a prohozenými dimenzemi.

_skip (časová složitost v nejhorším případě *O(m*n)*) – Přijme seznam reprezentující matici, nynější řádek, nynější sloupec, počet řádků a počet sloupců. Dokud je prvek na indexu, určeným nynějším řádkem a sloupcem, roven nule, posune index o řádek dolu. Pokud se index dostane přes poslední řádek, posune se o sloupec doprava a nastaví index řádku znovu na počáteční. Pokud je po *while* cyklu řádek s pivotem jiný, než který byl na začátku, řádky vymění. Vrátí index sloupce, ve kterém je pivot.
Pokud pivot nenajde vrátí hodnotu *-1*.

_expand (časová složitost *O(n^2)*) – Dostane seznam reprezentující čtvercovou matici a její dimenzi. Pro každý řádek *n*-krát připojí: *1* (pokud se číslo v pořadí přidávaného prvku rovná číslu řádku, do kterého přidáváme), jinak *0*.

_reduce (s maticí *m×n* je časová složitost *O(n)*) – Dostane seznam, index řádku a sloupce s pivotem, index řádku, který redukuji a počet sloupců matice, popř. informaci o tom, jestli mám brát v potaz, že matice podléhá *LU dekompozici*.
Zvolí koeficient *alpha*, kterým vynásobí prvek z řádku s pivotem a odečte jej od každého prvku redukovaného řádku tak, aby vznikla nula přímo pod pivotem.
Pokud matice podléhá rozkladu *LU* a při redukování se dostane do druhé poloviny rozšířené matice (upper triangular) a na diagonálu nebo dál, místo odečítání se přičítá. V druhé polovině rozšířené matice se před diagonálou neprovádí žádná operace.

_check – Vezme dva nebo jeden input a vrátí *True*, jestli je maticí, jinak *False*.

_scalar – Vezme jednu matici a jeden skalár a vrátí *tuple*, kde je skalár na první pozici.

class InputLoop

- v inicializaci třídy se vypíší úvodní informace o programu a zavolá se metoda *loop*, která běží, dokud je *boolean inProcess = 1*.

create_matrix (časová složitost při správném zadávání hodnot je *O(m×n)*) – Nejdříve se zeptá na dimenze (zavolá funkci *get_dims*, která vrací *tuple* (*int m*, *int n*) – pokud jsou zadány špatně, znovu zavolá sama sebe). Pak vrátí *Matrix* se seznamem, který vrátí funkce *get_values* – vytvoří prázdný seznam *c* s *m* řádky, kde pro každý připojí *n* pomlček (estetická funkce) a načte vstup. Pokud je vstup kratší než *n*, doplním ho nulami. Pokud je nějaká hodnota v řádku nenumerická, funkce se znovu zavolá. Postupně se nahrazují pomlčky v seznamu *c*.

exp_eval (časová složitost je primárně určena tím, jaké matice ve výrazu figurují a jaká operace se na ně zavolá – tzn. výraz s maticí n×n a nejnáročnější operací je *O(n^3)*) – Dostane lowercase *string* bez mezer. Pokud nemá závorky, zavolá funkci *command*. Pokud má závorky, zkontroluje jejich správnost pomocí funkce *symbol* s mezerou coby parametr (funkce *symbol* primárně hledá čárku, která rozděluje prvky účastnící se binární operace – zde je potřeba kontrolovat i *counter* závorek, který by měl mít bilanci 0, když symbol najde – aby se zabránilo nalezení dřívější čárky, která neodděluje členy vnější operace).
Následně do *result* uloží to, co vrátí funkce *evaluate* (která vezme dva parametry – začátek intervalu a konec+1 – najde otevřenou závorku – tím určí operaci a rekurzivně ji zavolá na obsah závorky. U binárních operací nejdřív najde index čárky a zavolá obsah závorky před a po ní; kvůli operaci *LU* se mimo jiné zjišťuje hloubka zanoření pomocí proměnné *d.*. Pokud v intervalu závorku nenajde, vrátí ho.) Po dokončení rekurze se *result* uloží pomocí *store* funkce.

store (časová složitost *O(m*n)*, prostorová *O(n)*) – Dostane seznam, nebo *tuple* a každému prvku v něm (maximálně jsou dva) přiřadí písmeno, pod kterým ho uloží do *slovníku* a vizualizuje výslednou matici pomocí funkce *visual*.

visual (časová složitost *O(m*n)*, prostorová *O(n)*) – Dostane Matrix. Vytvoří seznam jedniček, ve kterém zaznamená nejdelší hodnotu (převedenou na zlomek) v každém sloupci. Vypíše jednotlivé řádky matice (pokud se dostane na prostřední řádek, vypíše i jméno matice) – přitom pro každý sloupec využije seznamu maximálních délek (pro mezery), který ještě koriguje proměnnou *z*, která přičte 0 – pokud je délka hodnoty prvku sudá, nebo 1 – lichá.