a) Vypsání matice po řádcích

Popis algoritmu:

Vrchní cyklus prochází všechny řádky a vnitřní cyklus prochází všechny bloky v jednom řádku. Ve vnitřním cyklu získáváme pomocí metody read index na pozici i * N/B + j. „i * N/B“ nám udává index začátku řádku na disku a „j“ udává index bloku v daném řádku. Po načtení bloku do paměti jsou vypsány všechny prvky bloku. Na konci vnitřního cyklu procházející všechny bloky jednoho řádku je zalomen řádek, aby se matice vypsala hezky.

Pseudokód:

```
for(int i = 0; i < N; i++){
  for(int j = 0; j < N/B; j++){
    block = read(i * N/B + j);
    for(int x = 0; x < B; x++){
      print(block[x] + " ");
    }
  }
 println("");
}
```

Složitosti:

- Specifikace **paměťové složitosti** je splněna, jelikož máme konstantní počet proměnných (i, j, x) a velikost pole na prvky nepřekračuje M, které musí být větší než B.
- Každý prvek procházíme pouze jednou. Matice je ovšem čtvercová a počet prvků je N^2. **Časová složitost** je tedy $O(N^2)$
- Pro **komunikační složitost** platí, že každý blok čteme právě jednou. Zároveň platí, že blok vždy obsahuje na všech svých indexech číslo matice, nemáme tedy pouze částečně vyplněné bloky. Komunikační složitost je tedy rovna počtu bloků, tedy $O(N/B * N) = O(N^2/B)$

b) Vypsání matice po sloupcích

Nejjednodušší by bylo postupovat s následujícím kódem:

```
for (int i = 0; i < N; i++) {
  for (int j = 0; j < N; j++) {
    print(read((j * N + i) / B)[i % B] + "");
  }
  println("");
}
```

Program čte B krát bloky ve stejném sloupci. Po každém přečtení sloupce se hodnota „i“ zvýší o 1, čímž se zvýší i výsledek operace i % B oproti předchozímu měření o 1. Tím pádem vezmeme další index z pole. Jakmile je B krát přečten blokový sloupec, zvýší se operace i / B o 1, čímž dojde k posunu o další blokový sloupec.

Oproti předchozímu příkladu se časová ani paměťová složitost nemění, ovšem komunikační složitost je $O(N^2)$

Vylepšení:

Vylepšená verze využívá faktu, že M může být o něco větší než B. Přechozí příklad si toho vůbec nevšímal a vždy načítal blok B krát. Vytvoříme si tedy pole K o velikosti M – B, které budeme využívat na uchování části dat na další použití

Náš vylepšený program tedy bude pracovat následovně.

1. Pole M – B je v tomto kroku prázdné. Čteme bloky stejným způsobem jako v kódu nahoře a vždy vypisujeme první prvek přečteného bloku. Můžeme ovšem uložit M / B prvků z načteného pole do pole K. Ukládáme je na index i, který je počet prvků od posledního stavu, kdy bylo pole prázdné. Pokud je M / B větší než 1 (např. 2, 3), můžeme uložit současně více prvků z bloku. Ukládáme je na index i + N * (počet ukládaných čísel z bloku). Ukládáme přitom maximálně tolik čísel, kolik nám od posledního nevypsaného bloku zbývá do konce bloku (můžeme-li uložit 4 čísla v bloku, který má délku 8, ovšem vypsáno bylo už 6 čísel, zbývá uložit pouze 2 čísla).
2. Jakmile dojdeme do bodu, kdy je pole plné, pokračujeme s načítáním dalších bloků a vypisováním prvních indexů. Do pole přitom nic neukládáme. S tím končíme ve chvíli, kdy se dostaneme do posledního bloku, který jsme nezačali ukládat
3. Vypíšeme všechny prvky pole
4. Uložíme poslední index, kde začneme znovu načítat a vracíme se do bodu 1. Končíme v případě, že jsou vypsány všechny prvky

Toto vylepšení nemění časovou složitost. Paměťová složitost dosahuje přesně M. Změní se ovšem komunikační složitost. Ta je potom O (N^2-počet čtení, který je ušetřen polem)

c)

Popis algoritmu:

Jak je v zadání psáno, do paměti se nám v tomto případě vejde alespoň B^2 prvků. Pokud bychom si celou matici rozdělili po sloupcích i rádcích v každém směru B čárami, dostaneme v každém řádku N/B čtverců o počtu prvků B^2. Tohoto faktu můžeme využít tím, že budeme do paměti vždy číst celý tento čtverec.

O to se stará metoda *readBlockOfBlocks*, které jako argumenty předáváme souřadnice těchto bloků o straně B a konstanty B, N a dvourozměrné pole memory, které reprezentuje naši paměť. Metoda nic nevrací, naplní ovšem paměť bloky na disku.

Aby mohlo být memory naplněno, je potřeba převést souřadnice čtvercových bloků na souřadnice bloků na disku. O to se stará metoda *getDriveIndex*. Metoda *writeBlockOfBlock* poté slouží k zapsání čtvercového bloku na disk.

Algoritmus funguje na principu, který můžeme vidět v metodě *rotateInnerBlock*. Matice je procházena pomocí cyklů, kdy se v každém cyklu otočí pouze část matice. V prvním cyklu se otočí celý vnější okruh matice, tedy prvky nejblíže hranám matice. V dalším cyklu se otočí celý okruh o 1 prvek blíže ke středu a tak to pokračuje, dokud není celá matice otočená. Jelikož ovšem nemůžeme pracovat s jednotlivými prvky naší matice, tato metoda nám nestačí na vyřešení úlohy. Využijeme tedy faktu, že pokud rozdělíme matici do čtvercových bloků o straně B, poté budou všechny otočené obrazy prvků, které se nacházely v jednom bloku obsaženy v obrazu původního bloku. Tento celý blok se někam přesune a je vzhledem k původnímu bloku otočené o 90°. Jeho prvky jsou ovšem oproti původním prvkům v rámci tohoto otočeného bloku otočené oproti původnímu bloku také o 90°. Z tohoto vyplývá, že pokud načteme čtvercový blok do paměti, otočíme ho pomocí metody rotateInnerBlock a až poté ho přesuneme na správné místo v matici pomocí metody writeBlockOfBlock, tyto prvky budou otočeny, jako bychom rovnou otáčeli celou maticí NxN. O otáčení v rámci čtvercového bloku se tedy stará metoda rotateInnerBlock a o otočení tohoto bloku v rámci celé matice se stará metoda main.

Otáčení funguje na principu, že se vytvoří čtverec mezi čtyřmi prvky a tento čtverec si vymění své vrcholy pomocí hodinových ručiček. Těchto čtverců se vytvoří q pro každý cyklus r. První vrchol se uloží do pomocné proměnné a zbytek se prohodí, kdy na konci prohazování je poslední vrchol inicializován pomocí hodnoty v pomocné proměnné. Metoda rotateInnerBlock používá při svých výpočtech pomocnou metodu temp. V případě, že by to byl problém, je možné proměnné prohazovat i pomocí sčítání a odčítání. Byl by to ovšem další náklad na výkon aplikace, navíc v zadání je napsáno, že můžeme mít malý počet konstantních proměnných. Jak ovšem vyřešit problém s pomocnou proměnnou v případě pohybování čtvercových bloků, kdy používáme stejný způsob prohazování? V zadání je řečeno, že počet místa na disku je neomezený. Jednoduše tedy první blok uložíme za matici na disk. Následně prohodíme bloky a při posledním prohození přečteme za maticí uložený blok a vložíme ho zpět do matice.

Zároveň je třeba zmínit, co se stane v případě, že je matice lichá. V normální matici bychom otočili všechny prvky a střed bychom otáčet nemuseli, jelikož by se jednalo o jedno číslo. My ovšem máme místo normálních prvků čtvercové bloky, kde je potřeba otočit i čtvercový středový blok. Samotný algoritmus s tímto nepočítá a je tedy potřeba na konci běhu algoritmu zkontrolovat, zda je počet čtvercových bloků v matici lichý, nebo není. Pokud ano, otočíme vnitřní blok ručně.

Pseudokód:

```
void main(){
int[][] memory = new int[B][B];
  for (int i = 0; i < N / B / 2; i++) {
    for (int j = i; j < N / B - i - 1; j++) {
      readBlockOfBlocks(i, j, N, B, memory);
      rotateInnerBlock(B, memory);
      for (int x = 0; x < B; x++) {
        write(N * N / B + x, memory[x]);
      }

      readBlockOfBlocks(N / B - 1 - j, i, N, B, memory);
      rotateInnerBlock(B, memory);
      writeBlockOfBlocks(i, j, N, B, memory);

      readBlockOfBlocks(N / B - 1 - i, N / B - 1 - j, N, B, memory);
      rotateInnerBlock(B, memory);
      writeBlockOfBlocks(N / B - 1 - j, i, N, B, memory);

      readBlockOfBlocks(j, N / B - 1 - i, N, B, memory);
      rotateInnerBlock(B, memory);
      writeBlockOfBlocks(N / B - 1 - i, N / B - 1 - j, N, B, memory);

      for (int x = 0; x < B; x++) {
        memory[x] = read(N * N / B + x);
      }
      writeBlockOfBlocks(j, N / B - 1 - i, N, B, memory);
    }
  }
  if(N / B % 2 == 1){
    readBlockOfBlocks(N / B / 2, N / B / 2, N, B, memory);
    rotateInnerBlock(B, memory);
    writeBlockOfBlocks(N / B / 2, N / B / 2, N, B, memory);
  }
}
}

void rotateInnerBlock(int B, int[][] memory) {
  int temp;
  for (int r = 0; r < B / 2; r++) {
    for (int q = r; q < B - r - 1; q++) {
      temp = memory[r][q];
      memory[r][q] = memory[B - 1 - q][r];
      memory[B - 1 - q][r] = memory[B - 1 - r][B - 1 - q];
      memory[B - 1 - r][B - 1 - q] = memory[q][B - 1 - r];
      memory[q][B - 1 - r] = temp;
    }
  }
}

public static void readBlockOfBlocks(int i, int j, int N, int B, int[][] memory) {
  for (int x = 0; x < B; x++) { //radek B bloku v B^2 bloku
     memory[x] =read(getDriveIndex(i, j, N, B, x));
```

```
  }
}

public static void writeBlockOfBlocks(int i, int j, int N, int B, int[][] memory) {
  for (int x = 0; x < B; x++) { //radek B bloku v B^2 bloku
    write(getDriveIndex(i, j, N, B, x), memory[x]);
  }
}

public static int getDriveIndex(int i, int j, int N, int B, int x) {
  return j + (B * i + x) * N / B;
}
```

Složitosti:

- Paměťová složitost je $O(B^2 + \text{konstantní místo na proměnné})$, jelikož je při každém čtení nahrán do paměti celý čtvercový blok. Paměťovou složitost pro disk nemá smysl řešit, jelikož má neomezenou velikost, každopádně jedná se o $\mathbf{O(N^2 + B^2)}$ pro matici a místo pro jeden čtvercový blok za maticí
- Program vykoná N^2/B^2 přeskupení čtvercových bloků a zároveň musí každý blok vnitřně otočit. To zabere dalších B^2 kroků, tedy pro všechny $N^2/B^2 * B^2$ tedy N^2 kroků. Celková složitost je tedy $O(N^2 + N^2/B^2) = \mathbf{O(N^2 * (1 + 1/B))}$. Každému prvku se sice algoritmus věnuje pouze jednou, ovšem je potřeba počítat i s tím, že otáčíme matici způsobem nejdřív vnitřní matice čtvercového bloku a až poté její zapsání na disk, je zde tedy přidaných N^2/B^2
- V našem případě je každý blok přečten a zapsán právě jednou a do každého bloku je právě jednou zapsáno. Výjimku tvoří bloky, které zapisujeme za matici, ty čteme i zapisujeme dvakrát. Toto můžeme vyjádřit jako O(2*(N*N/B + k)) dvojka je zde kvůli tomu, že mluvíme jak o čtení, tak o psaní. N je počet řádků matice a N/B počet bloků v jednom řádku. *2k* (po roznásobení závorky 2k) poté značí počet čtení a psaní za matici nutné k prohození bloků. Při každém cyklu je nutné použít pomocnou proměnnou tolikrát, kolik je v daném cyklu prohazovaných prvků – 1. Prohazovaných prvků je v každém cyklu právě tolik, kolik je prvků na jednom řádku matice, které budeme prohazovat – 1.
  Pro matici:
  1 2 3 4
  5 **6** 7 8
  9 0 1 2
  3 4 5 6
  Jsem tučně zvýraznil čísla, která musíme načíst do pomocné proměnné, abychom mohli algoritmus provést. Cykly budou dva. První řádek cyklu je 1 2 3 4 a druhý 6 7. Délka prvního cyklu je 4, druhého 2. Pomocnou proměnnou používáme tedy 3 + 1 krát tedy 4 krát. Pokud je N sudé, pak je počet použití pomocné proměnné součtem: 1 + 3 + … + N – 1, pokud liché, tak 2 + 4 + … + N – 1. První posloupnost můžeme sečíst pomocí vzorce $\underline{(N+1)^2/2}$ a druhou posloupnost jako 2n(n+1)/2, tedy $\underline{n(n+1).}$ Proměnná k ve vzorci $\mathbf{O(2*(N*N/B + k))}$ je tedy závislá na paritě čísla N a nabývá právě jedné podtržené hodnoty.