API

Enuniado

Ao final, apresente um relatório explicando:

- A estrutura da classe criada.
- A lógica utilizada nos métodos.
- As diferenças entre os algoritmos de ordenação implementados e em quais casos um é mais eficiente que o outro.

Estrutura da classe criada

Criamos a classe Matriz com alguns atributos que iremos utilizar durante a execução dos seus métodos.

A classe Matriz recebe como argumento a quantidade de itens que a matriz de inteiros poderá receber e a quantidade de itens que cada vetor dentro da matriz poderá armazenar.

Imaginamos essa Matriz bidimensional como uma tabela com linhas e colunas, você perceberá que o código reflete isso.

Construtor

O método construtor se encarrega de criar a matriz que a classe Matriz utilizará para armazenar os valores

```java
public Matriz(int matrizSize, int vetorSize) {
    this.integers = new int[matrizSize][vetorSize];
    this.matrizSize = matrizSize;
    this.vetorSize = vetorSize;

    this.positionsAmount = this.vetorSize * this.matrizSize;
}
```

Método add

O método add é utilizado para adicionar inteiros em posições livres dentro dos vetores da matriz bidimensional. Se a matriz estiver cheia, o método notifica a situação e não executa a operação.

A lógica desse método basicamente se resume em percorrer cada vetor dentro da matriz bidimensional e adicionar o item na primeira posição livre que encontrar.

```java
public void add(int item) {
    if (this.amountOfItemsAdded >= this.positionsAmount) {
        System.out.println(x:"Matriz is full");
        return;
    }

    for (int index = 0; index <= this.getLastIndex(); index++) {
        for (int vetorIndex = 0; vetorIndex < this.integers[index].length; vetorIndex++)
        {
            if (this.integers[index][vetorIndex] != 0)
                continue;

            this.integers[index][vetorIndex] = item;
            this.amountOfItemsAdded++;
            return;

        }
    }
}
```

Método remove

O método remove serve para remover items adicionados nos vetores da matriz bidimensional.

A lógica envolvida nesse método é bastante simples, é preciso informar em qual vetor, e em qual posição do vetor, está o item que se deseja remover.

Antes de tudo é feito algumas verificações para garantir que o os valores passados como argumento são válidos e que realmente existe um elemento naquela posição.

Passando as verificações, o item é removido e o vetor reorganizado.

```java
public void remove(int row, int column) {
    if (row > this.getLastIndex()) {
        System.out.println(String.format(format:"Row '%d' is not exists", row));
        return;
    }

    if (this.integers[row][column] == 0) {
        System.out.println(String.format(format:"Element of column '%d' row '%d' is not
        exists", column, row));
        return;
    }

    this.integers[row][column] = 0;

    this.rearrangeArray();
}
```

@Override toString()

Esse método serve para retornar a matriz bidimensional de uma forma mais legível ao usuário e em forma de tabela com linhas e colunas.

Utilizamos um construtor de string para construir o texto que mostra de forma formatada a matriz bidimensional.

Esse texto é formado percorrendo cada vetor da matriz bidimensional e adicionando ao texto final o items da posição atual da iteração.

no final usamos o metodo toString() pra formatar a saída.
O @Override serve para sobrescrevermos o método toString já existente da classe Object

```java
@Override
public String toString() {
    StringBuilder sb = new StringBuilder();

    for (int i = 0; i <= this.getLastIndex(); i++) {
        for (int j = 0; j < this.integers[i].length; j++) {
            sb.append(this.integers[i][j]);
            if (j < this.integers[i].length - 1) {
                sb.append(str:", ");
            }
        }
        sb.append(str:"\n");
    }

    return sb.toString();
}
```

bubbleSort

O método bubbleSort serve para ordenar a matriz bidimensional de forma completa, esse método ordena linhas e colunas da “tabela”.

O bubbleSort irá iterar sobre um laço while, e a cada iteração ele irá iterar em um laço for até que o indice atual (que sempre é incrementado a cada iteração do for) for menor que o total de posições disponíveis na amtriz vezes o total de posições disponíveis do vetor, fazemos isso para percorrer todos os elementos da matriz

depois, baseado em linhas e colunas, vamos trocando os números, fazendo os maiores irem para o final e os menores para o começo.

Tem uma variável que é a chave para que tudo isso funcione, a swapped (troca), ela é iniciada como false a cada iteração do laço while, e definida como true a cada troca feita dentro do laço for. Essa é a forma de sabermos que ouve troca e que o processo precisa continuar até que não precise mais haver trocas de reordenação.

```java
public void bubbleSort() {
    boolean swapped;

    do {
        swapped = false;
        int totalLength = this.integers.length * this.integers[0].length;

        for (int i = 0; i < totalLength - 1; i++) {
            int rowIndex1 = i / this.integers[0].length;
            int colIndex1 = i % this.integers[0].length;
            int rowIndex2 = (i + 1) / this.integers[0].length;
            int colIndex2 = (i + 1) % this.integers[0].length;

            if (this.integers[rowIndex1][colIndex1] > this.integers[rowIndex2][colIndex2]) {
                int temp = this.integers[rowIndex1][colIndex1];
                this.integers[rowIndex1][colIndex1] = this.integers[rowIndex2][colIndex2];
                this.integers[rowIndex2][colIndex2] = temp;

                swapped = true;
            }
        }
    } while (swapped);
}
```

mergeSort

O método mergeSort tem a função de ordenar a tabela, mas ele ordena baseado em uma opção que diz a ele se ele deve ordenar as linhas ou as colunas da tabela.

Adicionamos comentários para facilitar a explicação a nível de código, mas basicamente, ele utiliza métodos privados da classe Matriz para fazer a ordenação baseado no valor recebido como argumento no parâmetro option

```java
// Método principal para ordenar a matriz usando Merge Sort
public void mergeSort(MergeSortOptions option) {
    if (option == MergeSortOptions.ROW) {
        // Ordena as linhas individualmente
        for (int i = 0; i < this.integers.length; i++) {
            mergeSortRow(this.integers[i], left:0, this.integers[i].length - 1);
        }
    } else if (option == MergeSortOptions.COLUMN) {
        // Ordena as colunas individualmente
        for (int j = 0; j < this.integers[0].length; j++) {
            // Extrai uma coluna, ordena e coloca de volta na matriz
            int[] column = extractColumn(this.integers, j);
            mergeSortRow(column, left:0, column.length - 1);
            insertColumn(this.integers, column, j);
        }
    }
}
```

```java
// Método Merge Sort para um array unidimensional (linha ou coluna)
private void mergeSortRow(int[] array, int left, int right) {
    if (left < right) {
        int middle = left + (right - left) / 2;

        mergeSortRow(array, left, middle);
        mergeSortRow(array, middle + 1, right);

        merge(array, left, middle, right);
    }
}
```

```java
// Função auxiliar para mesclar duas sub-partes de um array
private void merge(int[] array, int left, int middle, int right) {
    int n1 = middle - left + 1;
    int n2 = right - middle;

    int[] leftArray = new int[n1];
    int[] rightArray = new int[n2];

    // Copia os dados para arrays temporários
    for (int i = 0; i < n1; i++) {
        leftArray[i] = array[left + i];
    }
    for (int j = 0; j < n2; j++) {
        rightArray[j] = array[middle + 1 + j];
    }

    // Mescla os arrays temporários de volta ao array original
    int i = 0, j = 0;
    int k = left;
    while (i < n1 && j < n2) {
        if (leftArray[i] <= rightArray[j]) {
            array[k] = leftArray[i];
            i++;
        } else {
```

```java
            array[k] = rightArray[j];
            j++;
        }
        k++;
    }

    // Copia os elementos restantes de leftArray (se houver)
    while (i < n1) {
        array[k] = leftArray[i];
        i++;
        k++;
    }

    // Copia os elementos restantes de rightArray (se houver)
    while (j < n2) {
        array[k] = rightArray[j];
        j++;
        k++;
    }
}
```

```java
// Extrai uma coluna da matriz como um array
private int[] extractColumn(int[][] matrix, int colIndex) {
    int[] column = new int[matrix.length];
    for (int i = 0; i < matrix.length; i++) {
        column[i] = matrix[i][colIndex];
    }
    return column;
}

// Insere uma coluna de volta na matriz após a ordenação
private void insertColumn(int[][] matrix, int[] column, int colIndex) {
    for (int i = 0; i < matrix.length; i++) {
        matrix[i][colIndex] = column[i];
    }
}
```

Acima você pode ver cada parte do mergeSort e como fizemos para ele funcionar em uma matriz bidimensional de inteiros, ficou bastante performático 😃

A lógica do mergeSort é utilizar da recursividade para quebrar um vetor grande em duas partes e depois juntas essas partes para criar um vetor inteiro novamente, porém agora ordenado. É uma estratégia bastante performática pois não utiliza de laços, mas sim de condicionais em cima de listas pequenas, no final temos o vetor inteiro ordenado corretamente.
A recursividade é a peça chave dessa estratégia, pois o método que à utiliza chama ele próprio sempre passando a próxima parte do vetor maior quebrada ao meio, até que ele não consiga mais fazer essa divisão, quando isso acontece ele resolve tudo que ficou pendente.

mergeSort vs bubbleSort

A estratégia de merge sort é dividir pra conquistar, é recomendado usar em grandes listas de dados, também é importante notar que essa estratégia tem um nível de complexidade bem elevado.
O bubble sort é uma estratégia melhor aplicada em contextos onde existem pequenos conjuntos de dados previamente ordenados, o legal dessa estratégia é que a complexidade dela é bem mais baixa que a do merge sorte pois se baseia em comparação direta, por isso não faz sentido usar merge sorte sempre.

Merge Sort: Ideal para ordenar uma grande lista de registros de usuários em um sistema onde a eficiência é crucial.

Bubble Sort: Útil em um aplicativo de aprendizado onde um usuário precisa entender os conceitos básicos de ordenação em uma lista pequena de números.