UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
INSTITUTO METRÓPOLE DIGITAL
BOAS PRÁTICAS DE PROGRAMAÇÃO
2024.2

Estudo dirigido 05 – Desempenho e Otimização de Código e Padrões de Projeto
Relatório da Unidade 02

Thaynan Paulo Fernandes Bezerra de Mendonça

1. INTRODUÇÃO

Esse trabalho visa à criação de um sistema de busca para itens de um inventário de uma pequena loja virtual.

Como a quantidade de produtos desse inventário pode variar, podendo ter poucos itens até milhares, as diferentes possibilidades de busca podem impactar no desempenho.

No caso deste trabalho teremos um vetor (array) de inteiros contendo o código (id) de produtos na linguagem de programação *Java*. Cada busca verifica a existência do id no vetor e, caso ocorra, retorna a sua posição. Há duas possibilidades de busca: **Buscar Linear** ou **Busca Binária**. Também será realizada uma análise de cada busca para diferentes tamanhos de vetores, bem como, os possíveis *trade-offs* para o uso de cada busca.

Para finalizar, utilizou-se nesse trabalho o **Padrão *Strategy*** na qual se utiliza uma família de algoritmos de tal forma que se pode intercambiá-los de forma facilitada.

2. DESCRIÇÃO DO PADRÃO STRATEGY

Para a aplicação do padrão strategy no código temos a interface *SearchStrategy* que implementa a função busca de forma genérica.

As duas classes de busca (*BinarySearchStrategy* e *LinearSearchStrategy*) implementam o método dessa interface garantindo que tenham a mesma estrutura. Consequentemente as duas classes têm as mesmas entradas e saídas, sendo da mesma família, o que garante a execução comum.

No código foi criado a classe Estratégia (equivalente a InventorySearchContext solicitada no roteiro) que implementa uma variável do tipo SearchStrategy que pode ser tanto linear quanto binária, possibilitando o intercâmbio entre as classes de busca de forma mais célere.

3. IMPLEMENTAÇÃO DOS CÓDIGOS

Nesse tópico apresentaremos os códigos das classes e da interface importantes para esse projeto com uma breve explicação.

3.1 Interface (Classe) SearchStrategy

O código da classe consiste apenas na função busca que será implementada nas classes de busca linear e binária:

```java
/*
 * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
 * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
 */
package com.mycompany.inventario;

/**
 *
 * @author Thaynan
 */
public interface SearchStrategy {
    int busca(int id, int[] vetor);
}
```

3.2 Classe LinearSearchStrategy

Essa classe faz uma busca linear. Nesse tipo de busca pesquisa-se o número através do vetor partindo do primeiro elemento (vetor[0]) até o último elemento (vetor[i]). Nesse caso não há necessidade de ordenamento.

Além disso, como implementa a interface SearchStrategy precisa obrigatoriamente ter o método public int busca(int id, int[] vetor). O retorno da busca é:

1) O valor da posição (i+1) quando encontrado.
2) O valor -1 se não for encontrado o id na pesquisa;

Segue abaixo o código:

```java
package com.mycompany.inventario;
/**
 *
 * @author Thaynan
 */
public class LinearSearchStrategy implements SearchStrategy {
    @Override
    public int busca(int id, int[] vetor) {
        for(int i = 0; i < vetor.length ; i++) {
            if(vetor[i] == id) {
                return i+1;             // vetor começa de 0 até n-1 então a posição será i+1
            }
        }
     return -1;
    }
}
```

3.3 Classe BinarySearchStrategy

A classe BinarySearchStrategy realiza a busca binária. Esse tipo busca baseia-se na verificação a partir da metade do vetor, visando maior celeridade. Entretanto é preciso inicialmente ordenar o vetor, o que pode em alguns casos prejudicar o desempenho.

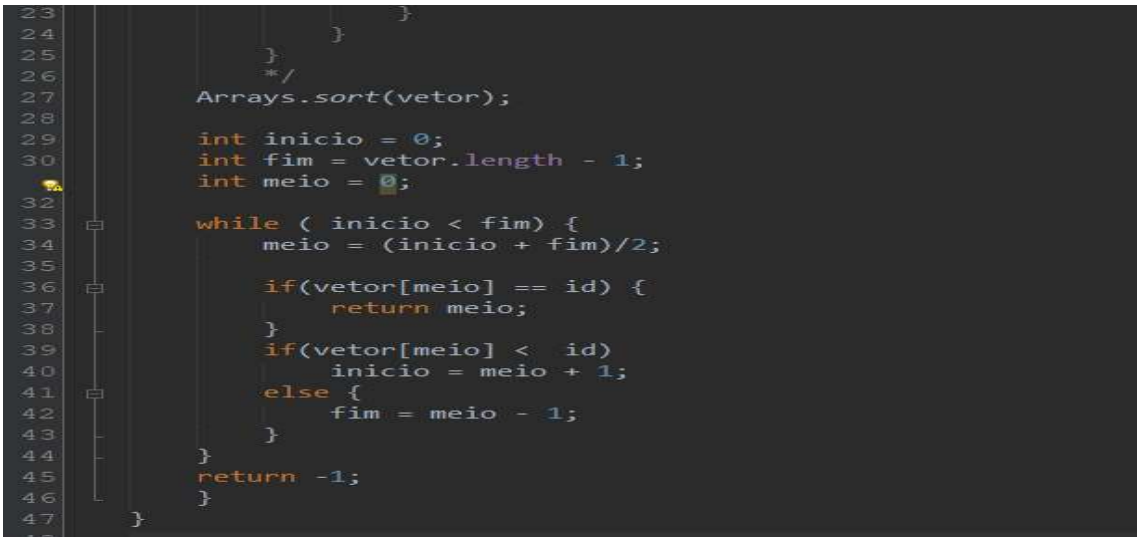
No código do projeto inicialmente realizei a operação de ordenamento usando um método próprio, contudo, a biblioteca java.util.Arrays permite o ordenamento com apenas uma função. Acabei optando por deixar ambas possibilidades de ordenamento, caso fosse preciso verificar o desempenho.

O algoritmo baseia-se na comparação no id com valor do vetor que corresponde à posição central (metade). A cada ciclo o valor de início ou fim do vetor é atualizado dependendo se esse ponto médio é maior ou menor que o id pesquisado.

Tal qual na classe da busca linear, o retorno será:

1) da posição i+1, se encontrado
2) do valor -1 se o id não for achado.

```
5   package com.mycompany.inventario;
6   import java.util.Arrays;
7
8   /**
9    *
10   * @author Thaynan
11   */
12  public class BinarySearchStrategy implements SearchStrategy {
13
14      public int busca(int id, int[] vetor) {
15          /*
16          int trocar = 0;
17          for(int i = 0; i < vetor.length; i++ ) {
18              for(int j = 0; j < i; j++ ) {
19                  if( vetor[i] < vetor[j]) {
20                      trocar = vetor[i];
21                      vetor[i] = vetor[j];
22                      vetor[j] = trocar;
23                  }
24              }
25          }
26          */
27      Arrays.sort(vetor);
```



```
                    }
                }
            }
            */
        Arrays.sort(vetor);

        int inicio = 0;
        int fim = vetor.length - 1;
        int meio = 0;

        while ( inicio < fim) {
            meio = (inicio + fim)/2;

            if(vetor[meio] == id) {
                return meio;
            }
            if(vetor[meio] <  id)
                inicio = meio + 1;
            else {
                fim = meio - 1;
            }
        }
        return -1;
        }
    }
```

3.4 **Classe Estrategia**

Na classe estratégia temos a implementação das possibilidades de busca. No caso temos a criação de uma variável do tipo SearchStrategy() que será utilizada para escolher o tipo de busca.

Equivale a classe InventorySearchContext solicitada no roteiro.

```
package com.mycompany.inventario;

/**
 * @author Thaynan
 */

public class Estrategia {
    private SearchStrategy estrategia;

    public Estrategia(SearchStrategy estrategia) {
        this.estrategia = estrategia;
    }

    public SearchStrategy getEstrategia() {
        return estrategia;
    }

    public void setEstrategia(SearchStrategy estrategia) {
        this.estrategia = estrategia;
    }

    public int buscar (int id, int[] vetor) {
        return estrategia.busca(id, vetor);
    }

}
```

3.5 Classe Inventário

É a classe que se encontra a função *main*. Foi apresentada para mostrar como ocorre:

1) A criação do vetor com diferentes tamanhos
2) O intercâmbio entre as buscas e;
3) Apresentação dos resultados dependendo se os valores foram encontrados ou não.

```java
package com.mycompany.inventario;


import java.util.Random;

/**
 *
 * @author Thaynan
 */
public class Inventario{

    public static void main(String[] args) {

        int id = 32;
        int tamanho = 50;
        int[] vetor = new int[tamanho];

        Random random = new Random();

        for(int i = 0; i < tamanho; i++) {
        vetor[i] = random.nextInt(70);
    }

        //Estrategia estrategia = new Estrategia (new BinarySearchStrategy());
        Estrategia estrategia = new Estrategia (new LinearSearchStrategy());
```

```java
        //Estrategia estrategia = new Estrategia (new BinarySearchStrategy());
        Estrategia estrategia = new Estrategia (new LinearSearchStrategy());

        int posicao = estrategia.buscar(id, vetor);

        for(int i = 0; i < vetor.length - 1; i++ ) {
            System.out.println(vetor[i] + ", " );
        }

        if (estrategia.buscar(id, vetor) != -1) {
            System.out.println("Encontrado na posicao: " + posicao);
        }
        else {
            System.out.println("Não encontrado!");
        }

    }
}
```

A mudança na estratégia ocorre através da inicialização da variável estratégia que pode ser através dos dois tipos de classes (*BinarySearchStrategy* e *LinearSearchStrategy*).

```java
//Estrategia estrategia = new Estrategia (new BinarySearchStrategy());
Estrategia estrategia = new Estrategia (new LinearSearchStrategy());
```

4. ANÁLISE DE DESEMPENHO

Faremos alguns testes com vetores com tamanho 50, 5.000 e 100.000. Para separar os tipos de busca foi colocado parte dos valores apresentados. As imagens na qual os números aparecem em ordem consistem na apresentação da busca binária, uma vez que essa busca exige o ordenamento.

Além disso, optou-se por casos em que a busca não encontra o id pesquisado. Esse é o pior caso possível para ambos tipos de busca, pois precisa ir até o último elemento para verificar a existência.

Na Busca Linear a complexidade é de *O(n)* nessa situação, em que n é o tamanho do vetor. Ou seja, na pior das hipóteses ele terá que percorrer todo o vetor (até n).

Já na Busca Binária, como cada etapa é dividida pela metade, a complexidade no pior caso é *O(log n)*. Entretanto, deve-se levar em consideração o tempo para ordenamento, que prejudica o desempenho.

O uso do Padrão *strategy* faz com os testes tornem-se mais fáceis de serem replicados, alterando apenas a inicialização dos métodos anteriormente apresentados.

4.1 **Teste com vetor de tamanho 50**

```java
 * @author Thaynan
 */
public class Inventario{

    public static void main(String[] args) {

        int id = 32;
        int tamanho = 50;
        int[] vetor = new int[tamanho];

        Random random = new Random();

        for(int i = 0; i < tamanho; i++) {
        vetor[i] = random.nextInt(35);
    }
```

Nesse cenário temos o vetor de tamanho 50 com o intervalo

de valores de 0 a 35. O id a ser encontrado é o 32.

Busca Linear

```
0,
0,
19,
18,
10,
15,
Nao encontrado!
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  2.100 s
Finished at: 2025-01-08T06:25:30-03:00
------------------------------------------------------------------------
```

Busca Binária

```
29,
30,
31,
33,
33,
Nao encontrado!
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  2.202 s
Finished at: 2025-01-08T06:28:12-03:00
------------------------------------------------------------------------
```

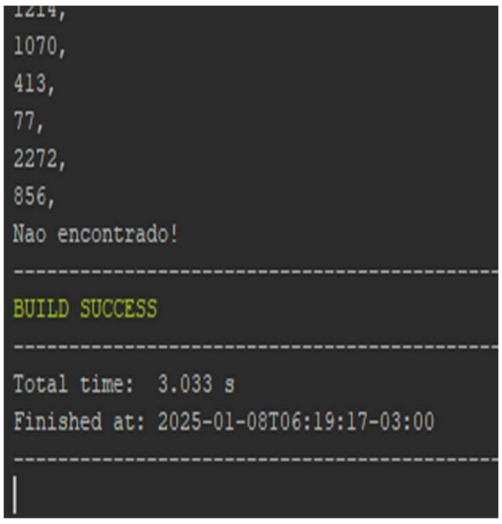
O tempo total da busca linear acaba sendo um pouco menor. A hipótese mais plausível é que a necessidade de ordenamento prejudica o desempenho para vetores de tamanho pequeno.

4.2 Teste com vetor de tamanho 5000

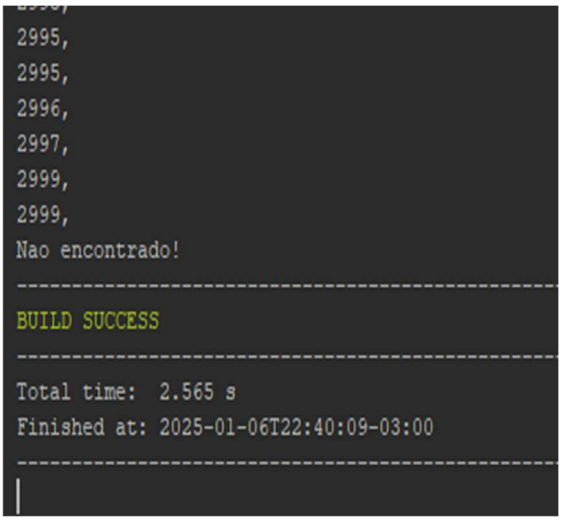
```java
public class Inventario{

    public static void main(String[] args) {

        int id = 32;
        int tamanho = 5000;
        int[] vetor = new int[tamanho];

        Random random = new Random();

        for(int i = 0; i < tamanho; i++) {
        vetor[i] = random.nextInt(3000);
    }

```

Nesse cenário temos o vetor de tamanho 5000 com o intervalo de valores de 0 a 3000. O id a ser encontrado continua sendo o valor 32.

Busca Linear



Busca Binária



Nesse caso, a busca binária acaba se tornando mais rápida. O crescimento do vetor faz com que o tempo de ordenamento acabe sendo compensado pela celeridade da busca a cada metade.

4.3 Teste com vetor de tamanho 100.000

```java
 * @author Thaynan
 */
public class Inventario{

    public static void main(String[] args) {

        int id = 32;
        int tamanho = 100000;
        int[] vetor = new int[tamanho];

        Random random = new Random();

        for(int i = 0; i < tamanho; i++) {
        vetor[i] = random.nextInt(30000);
    }
```

Nesse cenário temos o vetor de tamanho 100000 com o intervalo de valores de 0 a 40.000 (A imagem acima apresenta valor 30.000, mas a execução foi com 40.000 conforme resultado da busca binária). O id a ser encontrado continua sendo o valor 32.

Busca Linear

```
2796,
17751,
761,
5287,
23725,
16392,
Nao encontrado!
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  7.716 s
Finished at: 2025-01-06T22:48:36-03:00
------------------------------------------------------------------------
```

Busca Binária

```
39996,
39997,
39997,
39998,
39998,
39999,
39999,
Nao encontrado!
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time:  7.746 s
Finished at: 2025-01-06T22:53:53-03:00
------------------------------------------------------------------------
```

Nesse caso, a busca linear volta a ter vantagem, mas quase imperceptível. Então, para vetores extremamente grandes ambas estratégias equivalem, possivelmente por conta do enorme custo para ordenar vetores tão

grandes.

Nesse caso a busca por melhoria na busca implique na busca de outras estruturas de dados como tabela Hash ou árvores balanceadas;

5. DISCUSSÃO SOBRE ESTRUTURA DE DADOS

O vetor (array) é uma estrutura de dados que armazena objetos do mesmo tipo em posições contíguas de memória.

Além disso, aplicações que tenham os seguintes requisitos podem utilizar essa estrutura.

1) O tamanho fixo;
2) Acesso rápido através de índice, uma vez que os dados estão armazenados de forma contígua.

Aplicações com essas características podem usar essa estrutura de dados.

Além disso, o uso do vetor permite duas formas de consulta de dados: A busca Linear ou a Busca Binária. Como já foi dito anteriormente a busca linear pesquisa até o final do vetor enquanto a binária há o ordenamento e posterior “quebra” do vetor em metades cada vez menores.

A análise dos casos mostra que há perdas para o algoritmo de busca Binária para vetores de tamanho muito pequenos ou extremamente grandes, quase equivalente às duas estratégias para vetores grandes.

Possivelmente a necessidade de ordenamento prejudica o desempenho no caso dos pequenos (mais rápido pesquisar diretamente o vetor do que o ordenamento) e grandes (ordenar acaba levando tanto tempo que prejudica o ganho da pesquisa por metades). O tamanho do vetor que seria o ponto de virada para a perda de desempenho seria conseguido apenas com novos testes.

Para finalizar, temos algumas outras estruturas de dados que podem ser utilizadas para armazenamento de objetos que seriam mais adequadas que o vetor, dependendo dos requisitos do sistema:

1) Para o caso de sistemas que o tamanho armazenado seja variável, a lista encadeada é mais indicada.
2) Para o caso de sistemas com diferença de prioridade inserção/remoção pilhas, filas e listas podem ser utilizadas.

Logo, o uso do vetor dependerá dos requisitos de sistema do caso concreto.

6. Conclusão

A análise de casos permitiu a avaliação das buscas linear e binária em vetores. A necessidade de ordenamento prejudica o desempenho da busca binária no caso de pequenos e grandes vetores. A escolha do algoritmo dependerá das necessidades do sistema, como por exemplo memória disponível e tamanho do vetor.

Além disso, podemos verificar algumas sugestões de melhorias para ambos os tipos de busca. Para a Busca Linear termos:

1) Uso da tabela Hash para ganhar tempo na indexação.
2) Ordenamento anterior dos itens do vetor permitiria o encerramento da busca quando o valor pesquisado fosse menor que o último elemento pesquisado no vetor (depois desse valor não faz mais sentido pesquisar). Entretanto, o ordenamento levaria aos mesmos problemas de desempenho da busca Binária.

Para o caso da busca binária as sugestões seriam:

1) Usar a busca iterativa em vez da recursiva para economizar memória.
2) Outros tipos de estruturas de dados mais complexas como árvores rubro negras, grafos são opções para melhorar o desempenho.

7. REFERÊNCIAS

https://www.softplan.com.br/tech-writers/descomplicando-o-strategy/#:~:text=O%20Strategy%20%C3%A9%20um%20padr%C3%A3o,clientes%20que%20fazem%20uso%20dele. acessado em 06-01-2025 as 15:52

Algoritmos de busca sequencial e binaria - MundoJS acessado em 08-01-2025 as 12:00

Qual é a principal vantagem de usar um vetor array? acessado em 07-01-2025 as 18:30