Universidade Federal do Ceará

Campus Sobral

Engenharia da Computação

Vanessa Carvalho do Nascimento (Mat.: 471584)

Análise de Algoritmos de Busca

UNIVERSIDADE FEDERAL DO CEARÁ

04 de Setembro de 2023

Conteúdo

1 Análises

Este trabalho se propõe a analisar aspectos de tempo e memória dos 6 algoritmos de busca a seguir:

1. Busca binária $\rightarrow O(\log n)$
2. Busca Ternária $\rightarrow O(\log n)$
3. Busca Linear (Versão comum) $\rightarrow O(n)$
4. Busca Linear (Versão melhorada) $\rightarrow O(n)$
5. Busca Quadrática (com contagem de repetição de elementos)$\rightarrow O(n^2)$
6. Busca Cúbica $\rightarrow O(n^3)$

Para executar os códigos foi utilizado um computador com as seguintes especificações:

$\Rightarrow$ Placa gráfica GeForce MX 350 (NVIDIA).

$\Rightarrow$ Processador Intel Core i5

$\Rightarrow$ 8GB de RAM DDR4 de 2133Mhz

$\Rightarrow$ SSD de 256 GB NVMe

Para medir o tempo foi utilizada a função *time_np*() da biblioteca *time*, que mede o tempo em nanossegundos. Já para o cálculo do total de memória consumido foi utilizada a biblioteca *tracemalloc*, especificamente as funções *start()* e *get_traced_memory()*.

A seguir serão apresentados os detalhes das execuções.

1.1 Busca binária $\rightarrow O(\log n)$

A Figura 1 apresenta a implementação do algoritmo Busca Binária, que possui complexidade $O(\log n)$.

```
def PesquisaBinaria(x, vet, e, d):
    meio = int((e + d) / 2)
    if vet[meio] == x:
        return meio
    if e>=d:
        return -1
    elif vet[meio]<x:
        return PesquisaBinaria(x, vet, meio + 1, d);
    else:
        return PesquisaBinaria(x, vet, e, meio - 1);
```

Figura 1: Implementação do algoritmo busca binária.

Nesse algoritmo, o primeiro valor a ser analisado é o do meio. Assim, dividiu-se a análise em dois casos:

- Busca pelo valor do meio do vetor
- Busca pelo primeiro valor do vetor

O primeiro caso é o melhor cenário possível e deverá ser executado com um tempo total extremamente baixo, pois exige mínimos cálculos. Já o segundo, por se tratar de busca por um valor no extremo do vetor, deve executar em um tempo consideravelmente maior, já que se trata do pior caso.

1.1.1 Busca pelo valor do meio do vetor

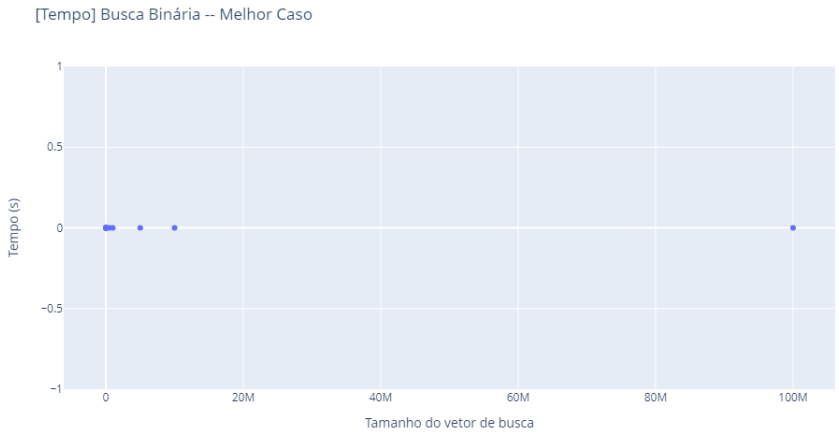

Figura 2: Tempo consumido na execução da busca binária no melhor caso.

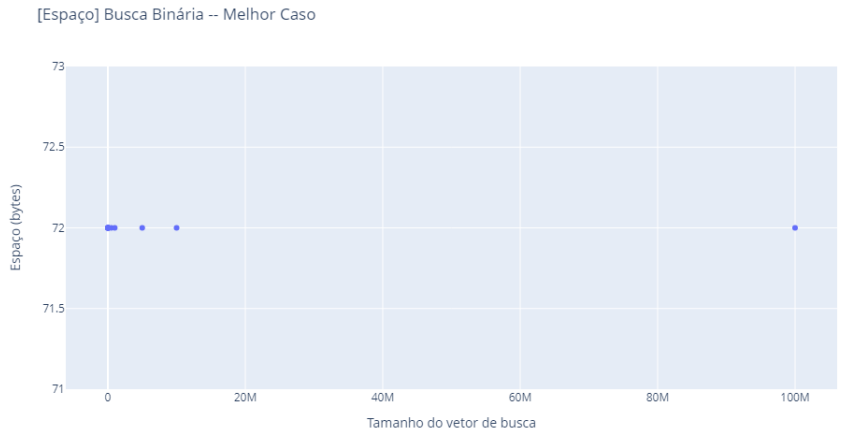

Figura 3: Espaço consumido na execução da busca binária no melhor caso.

1.1.2 Busca pelo último valor do vetor

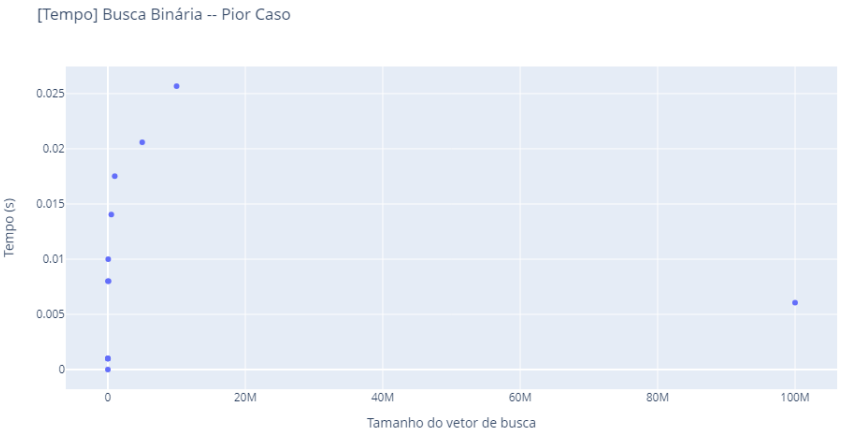

Figura 4: Tempo consumido na execução da busca binária no pior caso.

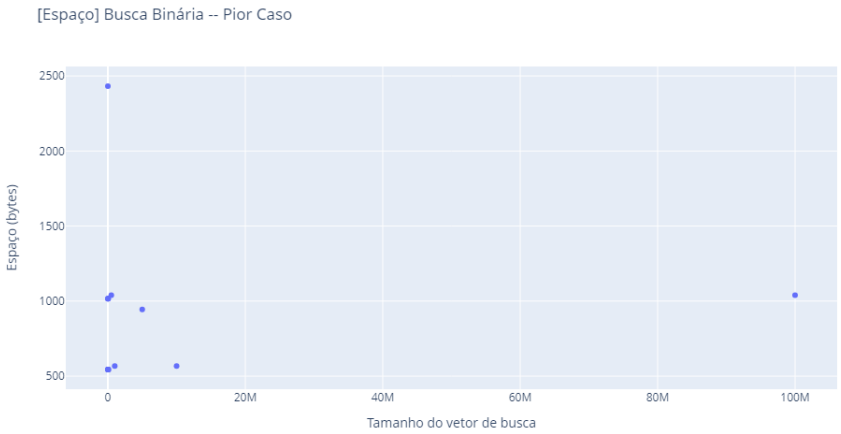

Figura 5: Espaço consumido na execução da busca binária no pior caso.

1.2 Busca ternária $\rightarrow O(\log n)$

A Figura 6 apresenta a implementação do algoritmo Busca Ternária, que possui complexidade $O(\log n)$.

```
def BuscaTernaria(vet, n, x):
    inicio = 0
    fim = n - 1
    while inicio <= fim:
        meio_esquerdo = math.floor(inicio + (fim - inicio) / 3)
        meio_direito = math.floor(fim - (fim - inicio) / 3)
        if vet[meio_esquerdo] == x:
                return meio_esquerdo;
        elif vet[meio_direito] == x :
            return meio_direito
        elif vet[meio_esquerdo] > x:
            fim = meio_esquerdo - 1
        elif vet[meio_direito] < x:
            inicio = meio_direito + 1
        else:
            inicio = meio_esquerdo + 1
            fim = meio_direito - 1
    return -1
```

Figura 6: Implementação do algoritmo busca ternária.

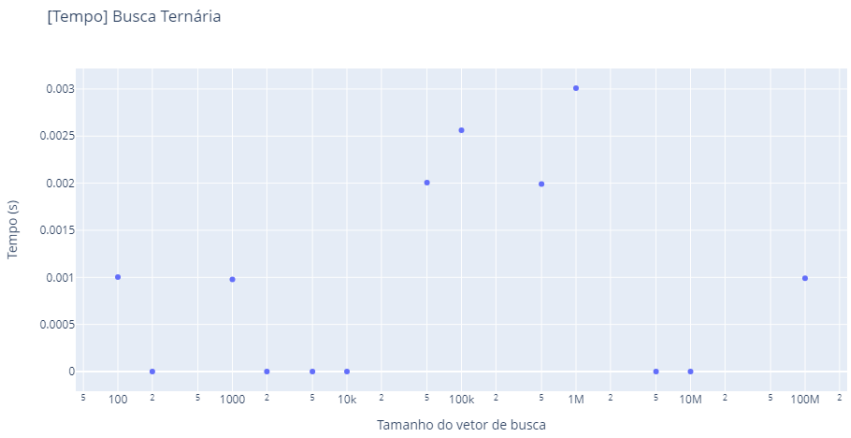

Figura 7: Tempo consumido na execução da busca ternária.

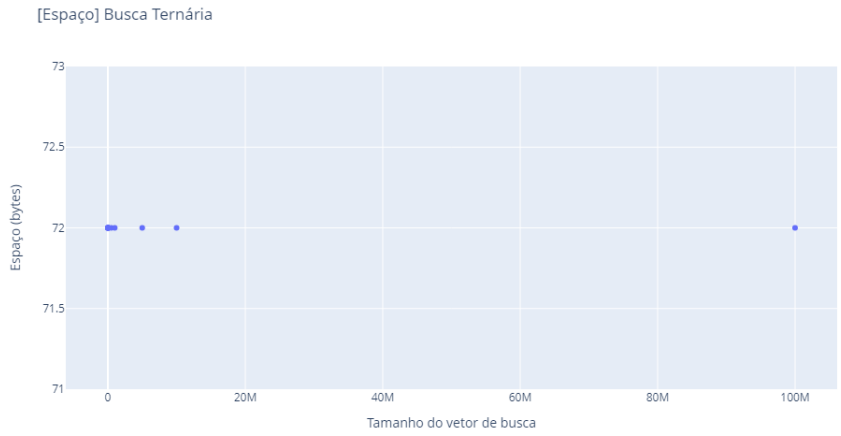

Figura 8: Espaço consumido na execução da busca ternária.

1.3 Busca sequencial (versão comum) $\rightarrow O(n)$

A Figura 9 apresenta a implementação da versão comum algoritmo de Busca Sequencial, que possui complexidade de $O(n)$.

```
def BuscaSequencialv1(x, vet):
    indice=-1;
    for i in range(len(vet)):
        if vet[i] == x:
            indice = i;

    return indice;
```

Figura 9: Implementação do algoritmo busca sequencial (versão comum).

1.3.1 Vetor de busca ordenado

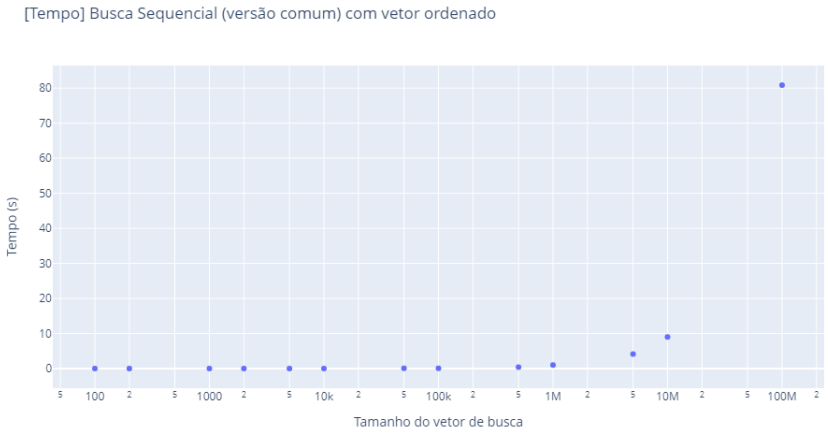

Figura 10: Tempo consumido na execução da busca sequencial (versão comum) com vetor ordenado.

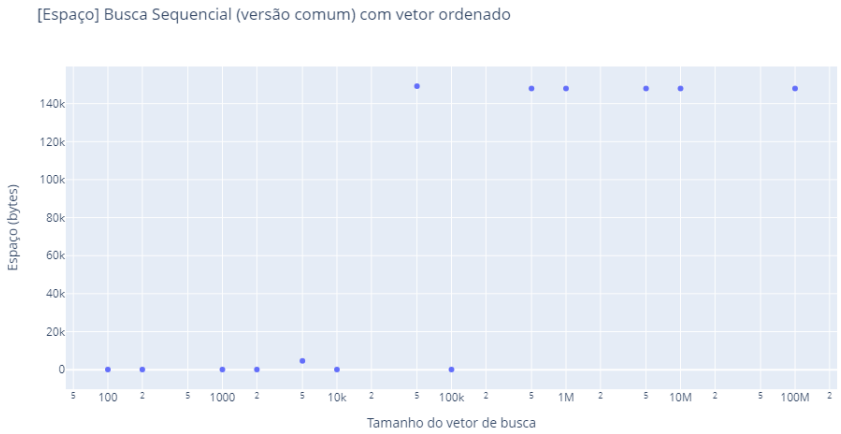

Figura 11: Espaço consumido na execução da busca sequencial (versão comum) com vetor ordenado.

1.3.2 Vetor de busca não-ordenado

Figura 12: Tempo consumido na execução da busca sequencial (versão comum) com vetor não-ordenado.

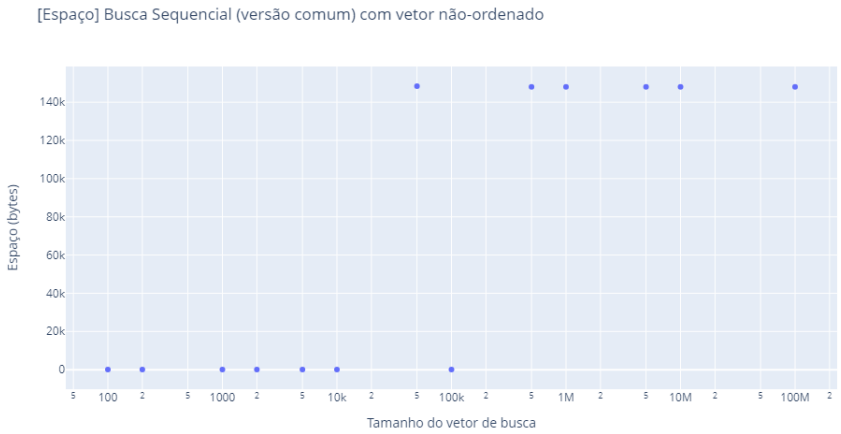

Figura 13: Espaço consumido na execução da busca sequencial (versão comum) com vetor não-ordenado.

1.4 Busca sequencial (versão melhorada) $\rightarrow O(n)$

A Figura 14 apresenta a implementação da versão melhorada do algoritmo de Busca Sequencial, que possui complexidade de $\Rightarrow O(\log n)$. Trata-se de uma versão mais otimizada do algoritmo anterior de busca sequencial, já que durante a análise sequencial, caso o número procurado seja encontrado, o algoritmo já retorna a posição, não percorrendo, assim, os elementos restantes.

```
def BuscaSequencialv2(x, vet):
    for i in range(len(vet)):
        if vet[i] == x:
            return i;
    return -1
```

Figura 14: Implementação do algoritmo busca sequencial (versão melhorada).

1.4.1 Vetor de busca ordenado

Figura 15: Tempo consumido na execução da busca sequencial (versão melhorada) com vetor ordenado.

Figura 16: Espaço consumido na execução da busca sequencial (versão melhorada) com vetor ordenado.

1.4.2 Vetor de busca não-ordenado

Figura 17: Tempo consumido na execução da busca sequencial (versão melhorada) com vetor não-ordenado.

Figura 18: Espaço consumido na execução da busca sequencial (versão melhorada) com vetor não-ordenado.

1.5 Busca quadrática (com contagem de repetição elementos) $\rightarrow O(n^2)$

A Figura 9 apresenta a implementação da versão comum algoritmo de Busca Sequencial, que possui complexidade de $O(\log n)$.

```
def BuscaQuadratica(numeroProcurado,vet):
    contador = 0
    posicao = -1
    entrou = False
    for i in range(len(vet)):
        for j in range(i,len(vet)):
            if vet[i] == numeroProcurado:
                if not entrou:
                    posicao = i;
                    if vet[j] == numeroProcurado:
                        contador+=1;

        if (contador > 0):
            entrou = True

    print("Posicao: " + str(posicao) + " - contador de repeticao: " + str(contador));
```

Figura 19: Implementação do algoritmo busca quadrática (com contagem de repetição de elementos)

.

1.5.1 Vetor de busca ordenado

O algoritmo foi executado usando vetores de busca com tamanho de até 100000 elementos, por conta do alto tempo de processamento.

Figura 20: Tempo consumido pela execução do algoritmo busca quadrática em um vetor ordenado.

Figura 21: Espaço consumido pela execução do algoritmo busca quadrática em um vetor ordenado.

1.5.2 Vetor de busca não-ordenado

O algoritmo foi executado usando vetores de busca com tamanho de até 50000 elementos, por conta do alto tempo de processamento.

Figura 22: Tempo consumido pela execução do algoritmo busca quadrática em um vetor não-ordenado.

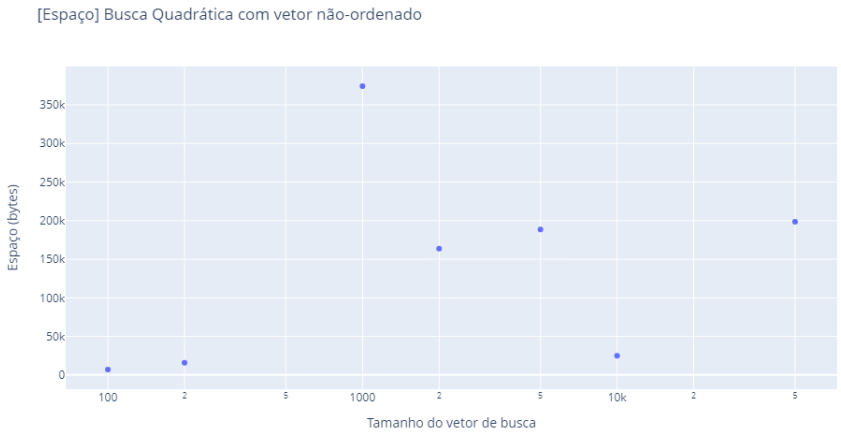

Figura 23: Espaço consumido pela execução do algoritmo busca quadrática em um vetor não-ordenado.

1.6 Busca cúbica $\rightarrow O(n^3)$

A Figura 24 apresenta a implementação do algoritmo de Busca cúbica, que possui complexidade $O(n^3)$. Trata-se do algoritmo com maior custo de processamento dentre os analisados.

```
def BuscaCubica(numeroProcurado,vet):

    posicao = -1

    for i in range(len(vet)):
        for j in range(len(vet)):
            for l in range(len(vet)):
                if (vet[i] == numeroProcurado and vet[j] == numeroProcurado and vet[l] == numeroProcurado):
                    posicao = i



    print("Posição: "+str(posicao))
```

Figura 24: Implementação do algoritmo Busca Cúbica.

1.6.1 Vetor de busca ordenado

O algoritmo foi executado usando vetores de busca com tamanho de até 5000 elementos, por conta do alto tempo de processamento.

Figura 25: Tempo consumido pela execução do algoritmo busca cúbica em um vetor ordenado.

Figura 26: Espaço consumido pela execução do algoritmo busca cúbica em um vetor ordenado.

1.6.2 Vetor de busca não-ordenado

O algoritmo foi executado usando vetores de busca com tamanho de até 1000 elementos, por conta do alto tempo de processamento.

Figura 27: Tempo consumido pela execução do algoritmo busca cúbica em um vetor não-ordenado.

Figura 28: Espaço consumido pela execução do algoritmo busca cúbica em um vetor não-ordenado.

2 Comparações de tempo

A seguir serão apresentadas algumas comparações de tempo entre os algoritmos analisados utilizando vetores de busca ordenados.

2.0.1 Busca Sequencial: versão comum vs versão melhorada

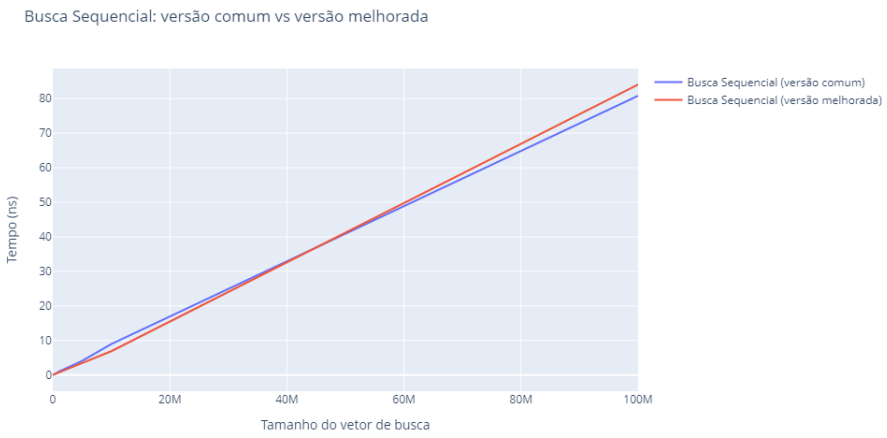

Figura 29: Busca Sequencial: versão comum vs versão melhorada.

Percebe-se que a modificação efetuada em relação ao algoritmo Busca Sequencial (versão comum) contribuiu positivamente reduzindo o tempo de execução conforme o tamanho do vetor de busca aumenta.

2.0.2 Busca Quadrática vs Busca Cúbica

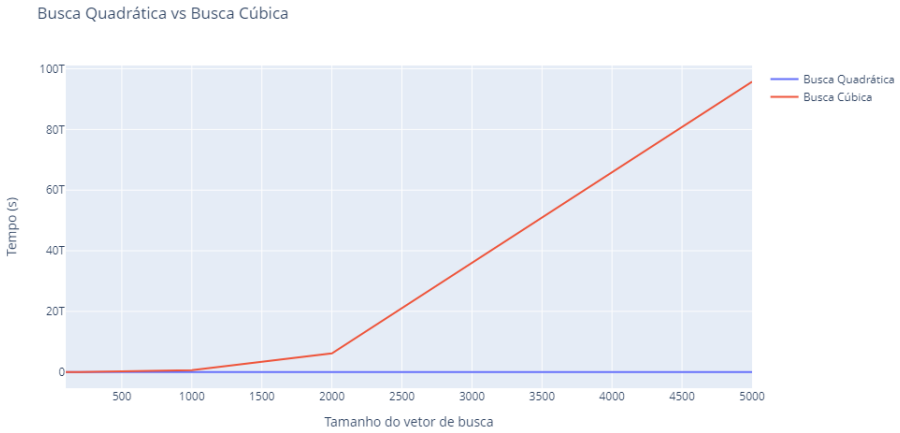

Figura 30: Busca Quadrática vs Busca Cúbica.

2.0.3 Busca Quadrática vs Busca Binária

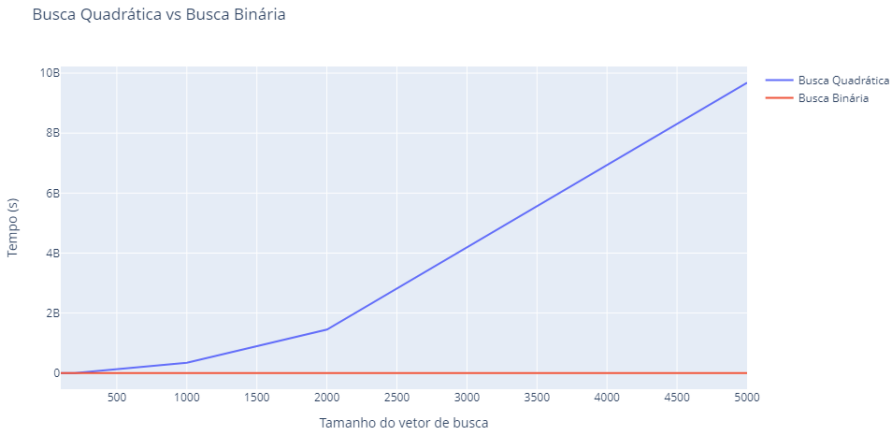

Figura 31: Busca Quadrática vs Busca Binária.