Análise da Busca Binária e outros Algoritmos de Busca

Giovanni Lucas O. da Silva[1]

[1]Instituto Federal de Brasília Campus Taguatinga (IFB)
72146-050 – Brasília – DF – Brazil

giovanni61540@estudante.ifb.edu.br

Abstract. *This work aims to analyze three common search algorithms: binary search, sequential search, and optimized sequential search. Both an empirical approach, through practical experimentation, and an analytical analysis will be used to understand the performance and efficiency of these algorithms in different scenarios. By combining these approaches, we hope to gain a comprehensive understanding of the strengths and weaknesses of each search algorithm, allowing us to make informed decisions when choosing the most suitable algorithm for search situations.*

Resumo. *Este trabalho tem como objetivo analisar três algoritmos de busca comuns: busca binária, busca sequencial e busca sequencial otimizada. Tanto uma abordagem empírica, através de experimentação prática, quanto uma análise analítica serão utilizadas para compreender o desempenho e a eficiência desses algoritmos em diferentes cenários. Ao combinar essas abordagens, esperamos obter uma compreensão abrangente dos pontos fortes e fracos de cada algoritmo de busca, permitindo-nos tomar decisões informadas ao escolher o algoritmo mais adequado para situações de busca.*

1. Introdução

Algoritmos de busca desempenham um papel crucial na localização de elementos específicos dentro de conjuntos de dados, estas coleçôes de dados fazem parte do dia a dia da maioria das informações ligadas a tecnologia. A escolha de tipos mais otimizados de busca para cada caso influencia diretamente o desempenho de sistemas que os utilizam pois "Algoritmos diferentes criados para resolver o mesmo problema muitas vezes são muito diferentes em termos de eficiência". [Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein 2009] [1]

A eficiência de um algoritmo de busca é frequentemente medida em termos de sua complexidade temporal (tempo de execução) e complexidade espacial (uso de memória). Algoritmos mais eficientes permitem buscas rápidas, mesmo em conjuntos de dados muito grandes.

Neste relatório técnico, investigaremos e compararemos três algoritmos de busca amplamente utilizados: busca binária, busca sequencial e busca sequencial otimizada. O objetivo é compreender e verificar o desempenho desses algoritmos, tanto por meio de análise prática: implementando-os e comparando seus tempos de execução em diferentes conjuntos de dados, quanto por análise teórica: calculando a complexidade temporal e por fim analisar se os tempos na prática refletem os tempos encontrados matematicamente assim como se a complexidade ciclomática tem alguma relação com a assintótica.

2. Conceitos preliminares

É de fundamental importância compreender os conceitos básicos por trás de cada algoritmo de busca que será analisado. A missão destes algoritmos é encontrar um elemento específico dentro de um conjunto de dados, seguindo uma estratégia definida.

2.1. Busca Sequencial

A busca sequencial é um algoritmo que tem como base percorrer de forma linear uma série de dados analisando elemento a elemento [Ashutosh Krishna 2022] [2], como se em um armário de gavetas fossemos abrindo uma por uma, da primeira à última sem pular nenhuma até achar o elemento desejado. Caso o elemento esteja na última gaveta ou não esteja na lista de elementos, analisaremos todas as gavetas anteriores para daí finalizar a procura.

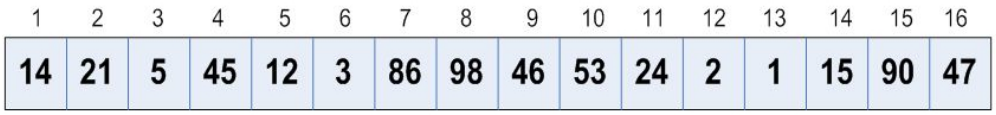


Figura 1. Vetor de números

Vamos assumir o vetor de exemplo da figura 1 onde o número 90 será buscado e queremos sua posição, começamos pelo primeiro elemento do vetor e verificamos se ele é o elemento que estamos buscando, caso não, continuamos comparando cada elemento do vetor com o valor que estamos procurando (neste caso, 90). Se o elemento atual for igual a 90, terminamos a busca e retornamos a posição do elemento.

Se percorrermos todo o vetor e não encontrarmos o valor 90, indicamos que ele não está presente no vetor, neste exemplo, o vetor contém vários elementos, e o número 90 está na penúltima posição. O algoritmo de busca sequencial percorre cada elemento até encontrar o valor 90, e então retorna o índice onde ele foi encontrado.

2.2. Busca Sequencial Otimizada

A otimização em algoritmos de busca visa melhorar a eficiência do processo de busca, reduzindo o número de comparações necessárias e as técnicas comuns de otimização incluem a poda de estados irrelevantes, o uso de heurísticas para guiar a busca e a adaptação dinâmica do algoritmo com base nas características dos dados. A busca sequencial otimizada é uma variação da busca sequencial que visa reduzir o número de comparações necessárias, reorganizando a lista ao ordenar os elementos de forma crescente a fim de melhorar a busca:

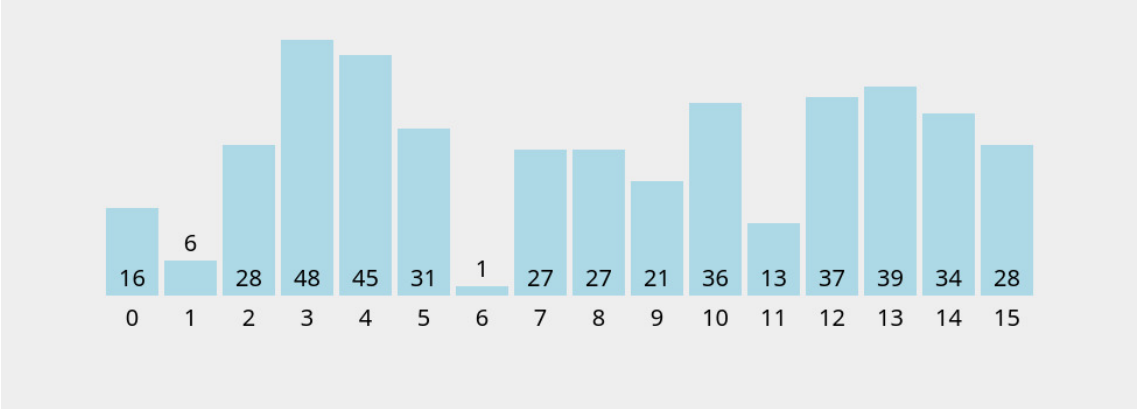


Figura 2. Vetor de números aleatórios

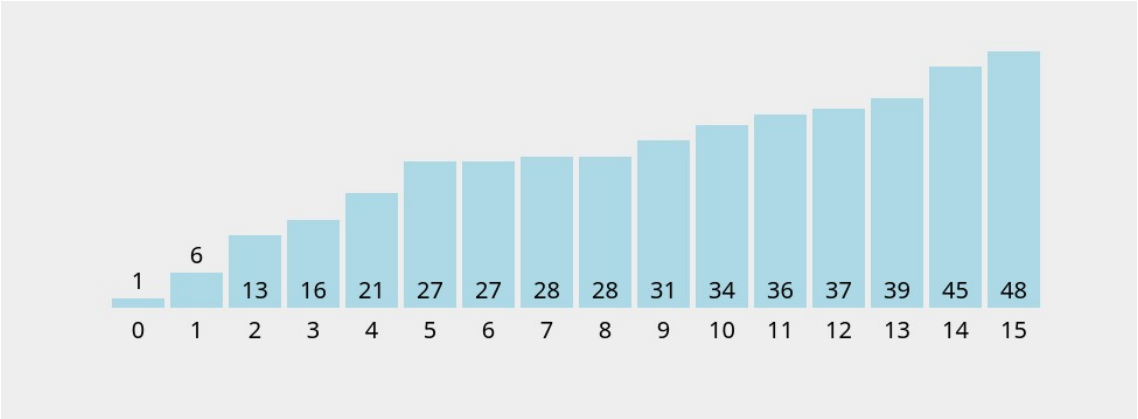


Figura 3. Vetor de números após ordenação

O vetor depois de ordenado é submetido à busca sequencial para encontrar um elemento especifico e uma das características dessa implementação é de que no caso de o elemento que estamos procurando for menor que o elemento que estamos comparando, é possível presumir que o elemento não está presente na lista, isto é, não está dentro de nenhuma gaveta adiante, então podemos parar de procurar antes mesmo de analisar qualquer outra gaveta, gerando assim mais rápidez para analizar os casos onde o número que estamos procurando não é um dos últimos. "Em suma, a busca sequencial é melhorada quando sabemos que a lista está ordenada somente no caso em que não encontramos o item". [Instituto de Matemática e Estatística 2019] [3]

2.3. Busca Binária

"Binary search is to algorithms what a wheel is to mechanics: It is simple, elegant, and immensely important." [Udi Manber 1989] [4]

A busca binária é um algoritmo que funciona analisando o elemento mediano de uma lista, comparando com o número que almejamos encontrar e caso não encontre, ele continua dividindo a lista em duas partes e comparando o elemento alvo com o elemento no meio da sublista ($\lfloor n/2 \rfloor$), que deve estar ordenada desde o início para que a lógica de divisão baseada no número fucione corretamente pois, é o valor deste elemento que será

comparado. Dependendo da comparação, o algoritmo decide em qual metade continuar a busca e esse processo é repetido até que o elemento seja encontrado ou a sublista se torne vazia, por exemplo:

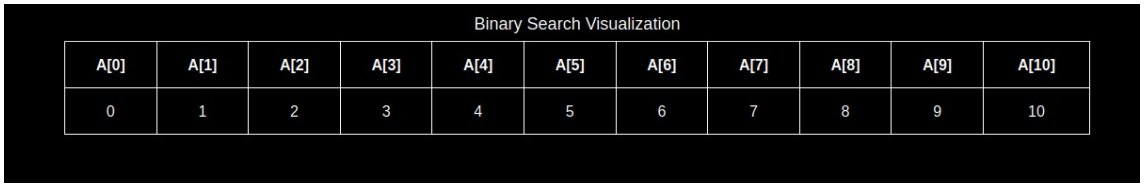


Figura 4. Vetor de números ordenados

Dado este vetor queremos achar o número 1, então analizaremos o elemento do meio e verificamos se o elemento mediano é o qual queremos encontrar:

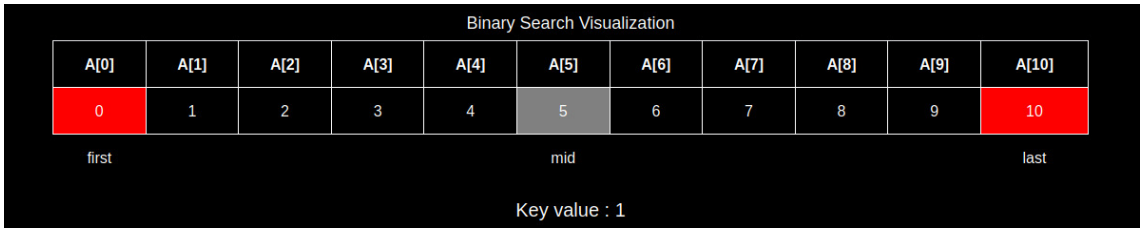


Figura 5. Vetor com as demarcações de início e fim com o elemento do meio

Comparamos o valor com o qual estamos querendo encontrar, neste caso é 5 que é diferente de 1 então vamos delimitar o vetor com base no elemento do meio e analizaremos o vetor esquedo resultante, pois como nosso vetor está ordenado, o elemento que procuramos está à esquerda do elemento analizado.

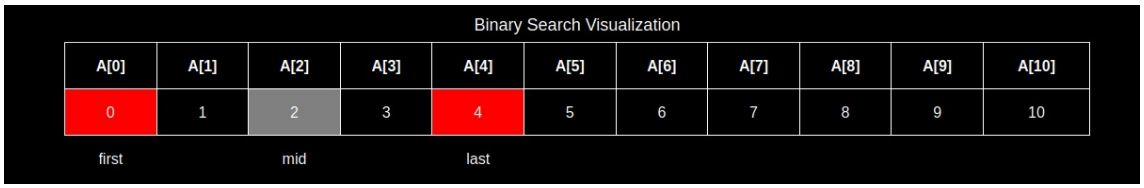


Figura 6. Vetor esquerdo com as demarcações de início e fim com o elemento do meio

Agora o 2 é o elemento central e ainda não chegamos ao nosso item então continuamos o algoritmo.

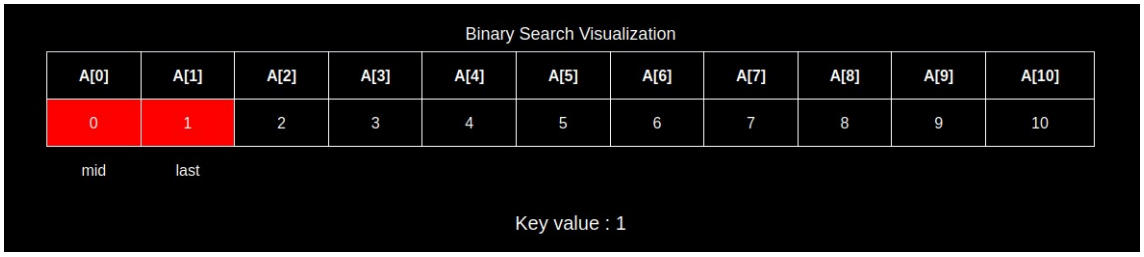


Figura 7. Vetor esquerdo com as demarcações de início e fim com o elemento do meio sendo o 0 pela divisão inteira por 2 ou $\lfloor n/2 \rfloor$

Agora analisando o elemento do meio (A[0]), sabemos que o elemento agora deve estar à direita e analizamos agora o que restou.

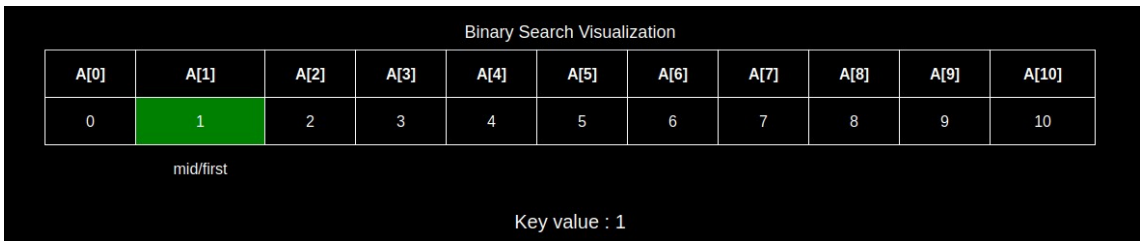


Figura 8. Vetor direito contendo somente o elemento que estavamos buscando na posição A[1]

Com isso a busca acaba e encontramos na segunda posição do vetor o número 1, finalizando assim o algoritmo e retornando sua correta posição.

3. Análise empírica

3.1. Metodologia

Na realização dos experimentos foi adotada a linguagem de programação "C" para a produção dos algoritmos de busca e para o algoritmo de ordenação requerido, utilizando na compilação destes programas a flag de otimização "-O3". Para cada um dos algoritmos foi criado um vetor de tamanho "n" e efetuado um número de buscas "q", onde "n" e "q" podem assumir os valores de 10^2, 10^3, 10^4 e 10^5, gerando 48 resultados de tempo. A população do vetor foi efetuada de forma pseudo-aleatória utilizando funções da linguagem, assim como foi feito ao número a ser avaliado em cada busca.

Os experimentos foram conduzidos utilizando um script "Bash" para realizar a execução dos códigos cinco vezes e realizar a média aritmética dos tempos de execução das linhas de código referentes a cada busca, isto é, na busca sequencial é analisado somente a busca, enquanto na busca sequencial otimizada e binária é tanto o tempo de ordenação do vetor quanto da busca em si.

Para a ordenação dos vetores foi utilizado o algorítimo "Merge Sort" por sua eficiência de ordenação de vetores grandes "Assim, o Mergesort só se torna realmente mais rápido que os algoritmos elementares quando n é suficientemente grande" [Paulo Feofiloff 2019].

3.2. Resultados

Os resultados encontrados foram atribuídos em tabelas para a geração de gráficos para melhor visualização.

Tabela 1. Tabela representando os tempos de execução de cada busca em segundos, relacionado com o tamanho do vetor e com o número de buscas por execução do algoritmo ("q") igual a 100

Número de buscas: 100			
Tamanho do Vetor	**Busca Sequencial**	**Busca Sequencial Otimizada**	**Busca Binária**
10000	0.000335	0.001923	0.001195
100000	0.002370	0.018650	0.011646
1000000	0.024733	0.188817	0.120884
10000000	0.305588	2.122835	1.364604

Tabela 2. Tabela representando os tempos de execução de cada busca em segundos, relacionado com o tamanho do vetor e com o número de buscas por execução do algoritmo ("q") igual a 1000

Número de buscas: 1000			
Tamanho do Vetor	**Busca Sequencial**	**Busca Sequencial Otimizada**	**Busca Binária**
10000	0.002766	0.004344	0.001462
100000	0.024865	0.043234	0.013976
1000000	0.234909	0.448841	0.201750
10000000	2.826622	4.911622	1.714041

Tabela 3. Tabela representando os tempos de execução de cada busca em segundos, relacionado com o tamanho do vetor e com o número de buscas por execução do algoritmo ("q") igual a 10000

Número de buscas: 10000			
Tamanho do Vetor	**Busca Sequencial**	**Busca Sequencial Otimizada**	**Busca Binária**
10000	0.020877	0.027366	0.002373
100000	0.190778	0.228395	0.013546
1000000	1.934093	2.348101	0.131148
10000000	21.404409	24.274155	1.428998

Tabela 4. Tabela representando os tempos de execução de cada busca em segundos, relacionado com o tamanho do vetor e com o número de buscas por execução do algoritmo ("q") igual a 100000

Número de buscas: 100000			
Tamanho do Vetor	**Busca Sequencial**	**Busca Sequencial Otimizada**	**Busca Binária**
10000	0.202421	0.291759	0.010225
100000	1.972691	2.783297	0.021631
1000000	20.558575	17.62991	0.130241
10000000	181.177704	190.807327	1.346202

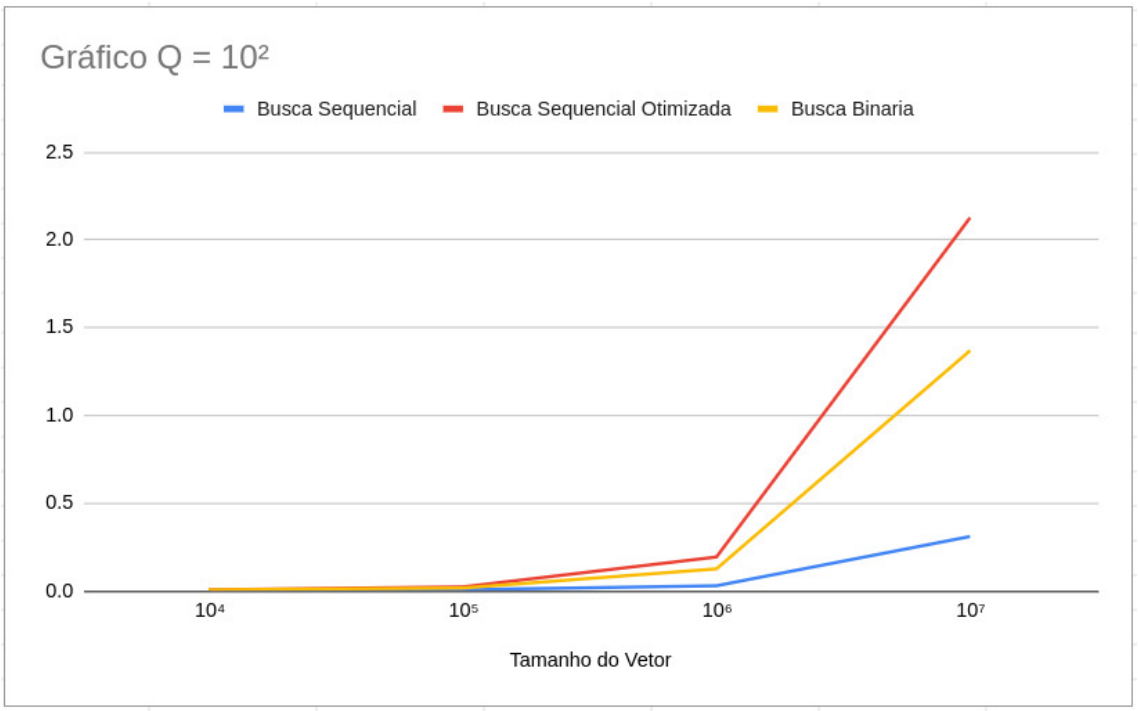


Figura 9. Gráfico com 100 buscas pelo vetor.

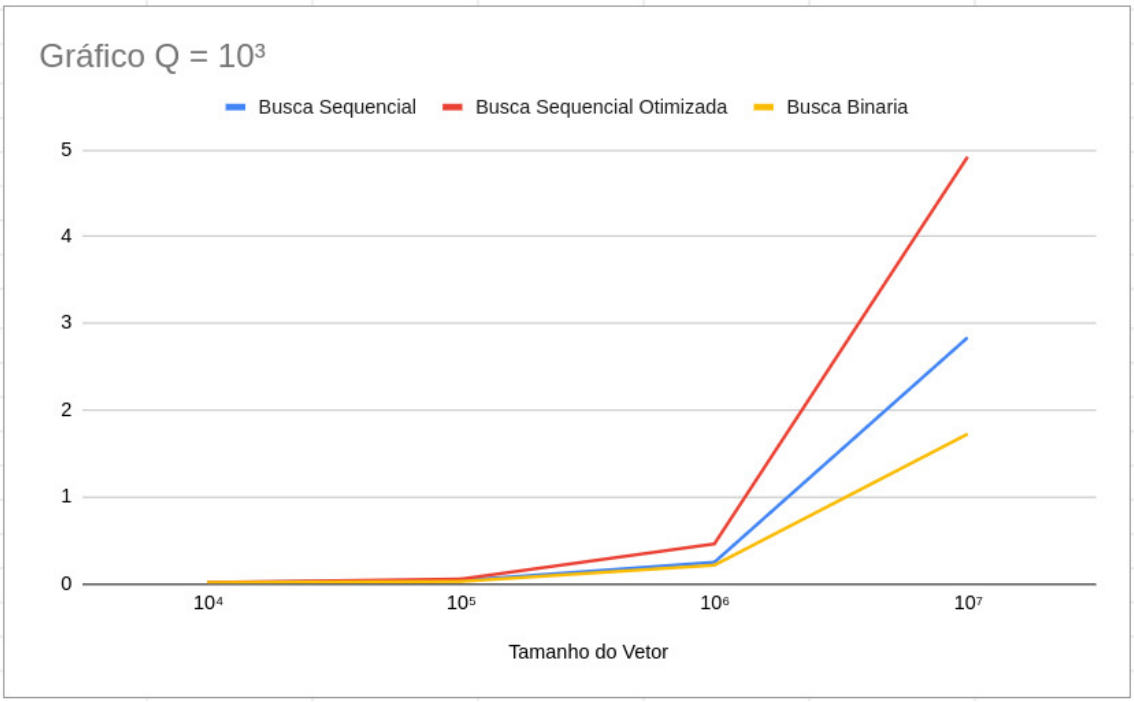


Figura 10. Gráfico com 1000 buscas pelo vetor.

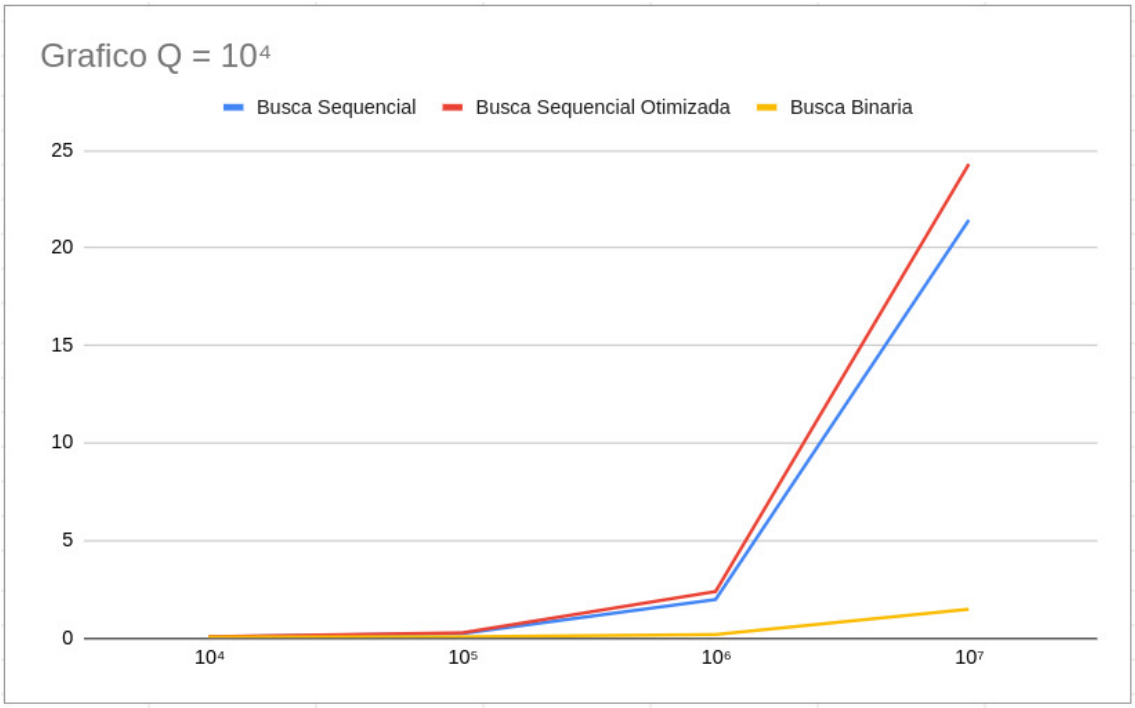


Figura 11. Gráfico com 10000 buscas pelo vetor.

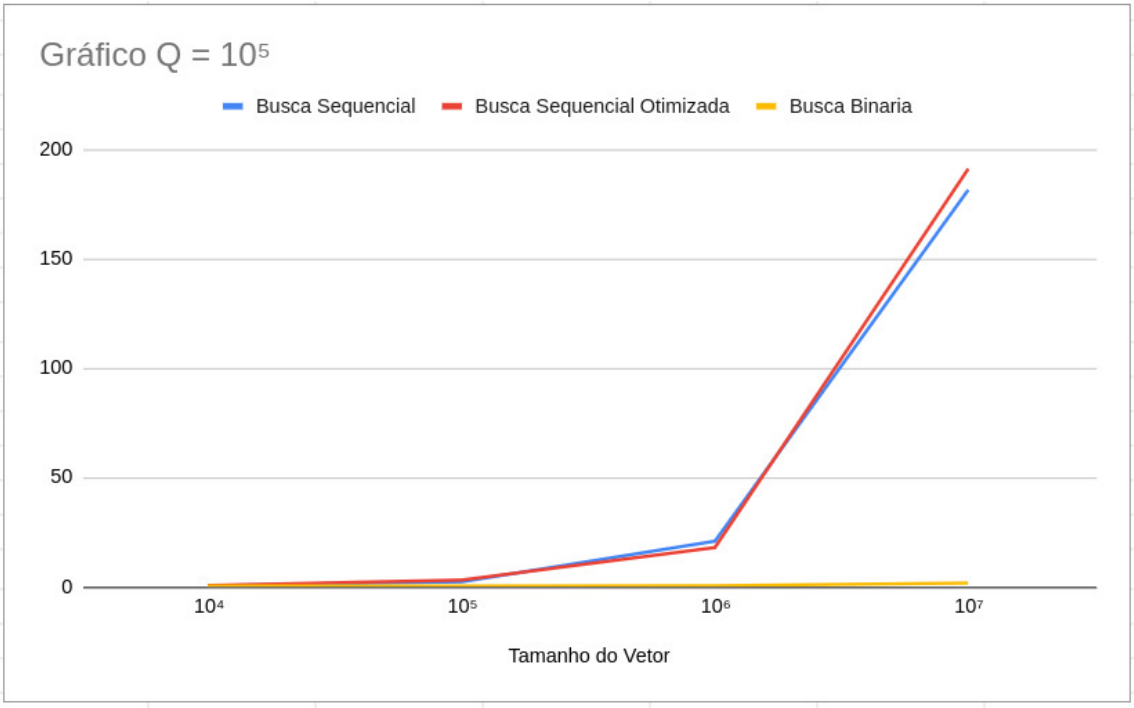


Figura 12. Gráfico com 100000 buscas pelo vetor.

Observando as tabelas e os gráficos podemos inferir o comportamento dos algorítmos: No caso do algorítmo de busca sequencial, a linha de tempo cresce linearmente conforme o tamanho do vetor aumenta. Isso porque, no pior cenário, é necessário verificar todos os elementos do vetor o que liga o tempo de execução ao tamanho do vetor

utilizado, comparado aos outros ele desempenha bem em vetores pequenos e ao longo que o vetor cresce ele começa a piorar em relação aos outros.

Já na sequencial otimizada a eficiência é melhorada em relação à busca sequencial simples, especialmente se o elemento estiver próximo ao início do vetor. No entanto o tempo utilizado para ordenar o vetor aumenta seu tempo para vetores pequenos e, no pior caso, se o elemento estiver próximo ao final do vetor ou não estiver presente, a complexidade ainda será a mesma que a simples.

É possivel visualizar que ao aumentar o tamanho do vetor as duas buscas começam a se aproximar tanto que podemos enchergar que a busca otimizada passa a simples quando se trata de vetores grandes, mostrando que sua eficiência em vetores maiores é superior a simples mesmo com o tempo de ordenação.

A linha de tempo cresce devagar a medida que cresce o vetor, o que significa que a busca binária é muito mais eficiente em vetores grandes em comparação com a busca sequencial simples ou otimizada. Mesmo no pior caso, a quantidade de comparações necessárias aumenta muito mais lentamente do que no caso de uma busca linear.

3.3. Complexidade ciclomática

O valor da complexidade ciclomática obtida automaticamente pelo site SonarCloud gerou que a busca sequencial têm o valor ciclomático 7 e a otimizada 20, já a busca binária deu 24 de valor ciclomático, analisando estes resultados empíricos fornecidos por cada implementação é possível notar que a medida que o algoritmo fica mais complexo isto é, ele tem a possibilidade de percorrer mais caminhos, ele não altera o resultado final de desempenho dos algoritmos. "A complexidade ciclomática é definida como a medição da quantidade de lógica de decisão em uma função de código-fonte [NIST235]. Simplificando, quanto mais decisões precisarem ser tomadas no código, mais complexas serão." [Microsoft 2022] [6]

Vale resaltar que a busca sequencial otimizada tem uma métrica ciclomática mais próxima da busca binária e que para os vetores analizados ela não apresenta um real acréscimo em sua perfomace, é possível também ponderar que pelo gráfico dela estar se aproximando da busca sequencial a cada aumento do vetor, para vetores realmente grandes, ela pode superá-lá em algum momento, isso poderá ficar mais claro após a analise assintótica destes algoritmos.

3.4. Código

O código utilizado nestes experimentos se encontram neste endereço: `https://github.com/Giovanni-LOS/Binary_Search`.

3.5. Ambiente experimental

As especificações da máquina podem impactar o desempenho dos algoritmos implementados, especialmente o tempo de execução e o uso de memória. As especificações de hardware e software utilizadas para a execução dos algoritmos de busca e ordenação:

Todos os testes foram executados e realizados em um notebook com o sistema operacional Arch Linux x86_64 com processador Intel i7-1165G7 (8) @ 4.700 GHz de décima primeira geração.

4. Análise dos algoritmos

A análise dos algoritmos envolve o estudo e a avaliação do desempenho dos algoritmos, e neste trabalho será feito utilizando a notação assintótica, que é utilizada para descrever o comportamento temporal de um algoritmo em termos de sua eficiência e complexidade. Esta notação nos ajuda a ter uma aproximação do que se pode encontrar na realidade, pois "ela considera o comportamento de funções no limite" [Flavio moura 2023], o real desempenho vai depender da capacidade de cada computador.

As classes básicas de eficiência que utilizaremos para analisar algoritmos são listadas a seguir em ordem crescente de complexidade em função do tamanho "n"da entrada:

Tabela 5. Complexidade de diferentes classes de funções

Classe	Nome
1	constante
$\log n$	logarítmica
n	linear
$n \log n$	linearítmica
n^2	quadrática
n^3	cúbica
n^k	polinomial (k $\geq$ 1 e finito)
a^n	exponencial (a $\geq$ 2)
$n!$	fatorial

4.1. Busca Sequencial

O pseudocódigo utilizado para a criação do programa e o código criado são respectivamente:

```
Para i de 0 até n-1 faça:
    Se vetor[i] == x então
        Retorne i
Retorne -1
```

```
int seqSearch( int array[], int len, int num ) {

        for(int i = 0; i<len; i++) {

                if(array[i] == num) {

                        return 1;
                }
        }
        return -1;
}
```

Ao analisar a complexidade de tempo podemos chegar as sequintes conclusões: no melhor caso o elemento que queremos comparar estará na primeira posição logo na execução teremos 1 comparação logo a complexidade é O(1).

$$\text{Número de Comparações no Melhor Caso} = 1$$

Portanto, a complexidade do pior caso é:

$$O(1)$$

O pior caso ocorre quando o elemento x está na última posição do vetor ou não está presente no vetor. Nesse caso, o número de operações é igual ao número de elementos no vetor n.

$$\text{Número de Comparações no Pior Caso} = n$$

Portanto, a complexidade do pior caso é:

$$O(n)$$

4.2. Merge Sort

Para a ordenação dos vetores dos próximos algoritmos, foi utilizado um tipo de algoritmo de ordenação chamado Merge Sort, ele foi implementado na forma serial simples:

```
1  void merge(int *array, int *array1, int *array2, int num) {
2
3          int size_array1 = num / 2;
4          int size_array2 = num - size_array1;
5          int i = 0, j = 0, k = 0;
6
7          for(;j < size_array1 && k < size_array2; i++) {
8                  if( array1[j] <= array2[k] ) {
9                          array[i] = array1[j++];
10                 } else {
11                         array[i] = array2[k++];
12                 }
13         }
14
15         while( j < size_array1 ) {
16                 array[i++] = array1[j++];
17         }
18
19         while( k < size_array2 ) {
20                 array[i++] = array2[k++];
21         }
```

```
22 }
23
24 void mergeSort( int *array, int num ) {
25
26         int mid;
27         if( num > 1 ) {
28                 mid = num / 2;
29
30                 int *array1 = malloc(sizeof(int) * mid);
31                 if(array1 == NULL) {
32                         exit(-1);
33                 }
34                 int *array2 = malloc(sizeof(int)*num-mid);
35                 if(array2 == NULL) {
36                         exit(-1);
37                 }
38                 int i;
39
40                 for (i = 0; i<mid; i++) {
41                         array1[i] = array[i];
42                 }
43
44                 for(i = mid; i<num; i++) {
45                         array2[i - mid] = array[i];
46                 }
47
48                 mergeSort(array1, mid);
49                 mergeSort(array2, num - mid);
50                 merge(array,array1,array2,num);
51
52                 free(array1);
53                 free(array2);
54         }
55 }
```

A complexidade deste algoritmo é $O(n \log n)$, segundo o artigo “Performance analysis of merge sort algorithms” [Sonia Kuwelkar and Joella Lobo 2020],onde n é o tamanho do vetor, e como a etapa de ordenação está ligada às próximas buscas ela será acrescentada na analise final.

4.3. Busca Sequencial Otimizada

O pseudocódigo utilizado para a criação do programa e o código criado são respectivamente:

```
Para i de 0 até n-1 faça:
    se array[i] > num:
        interrompe o laço
    se array[i] == num:
        Retorna 1
Retorna -1
```

```
int seqSearchOpt( int array[], int len, int num ) {

        for(int i = 0; i<len; i++) {

                if( array[i] > num ) {
                        break;
                }

                if( array[i] == num ) {

                        return 1;
                }
        }
        return -1;
}
```

A busca sequencial otimizada em si não muda muito da busca sequencial, a parte que mais se difere é que pelo vetor já estar ordenado previamente podemos interromper a busca assim que for comparado um número maior do o que estamos comparando, mas se analisarmos o melhor e pior caso temos que: no melhor caso o elemento está na primeira posição então faremos 1 comparação somente.

$$\text{Número de Comparações no Melhor Caso} = 1$$

Portanto, a complexidade do melhor caso é:

$$O(1)$$

O pior caso ocorre igualmente quando o elemento x está na última posição do vetor ou não está presente no vetor. Nesse caso, o número de operações é igual ao número de elementos no vetor n.

$$\text{Número de Comparações no Pior Caso} = n$$

Portanto, a complexidade do pior caso é:

$$O(n)$$

Adicionar a interrupção quando o número procurado é menor que o número comparado pode melhorar a eficiência prática no caso médio, mas a complexidade assintótica no pior caso não muda.

4.4. Busca Binária

O pseudocódigo utilizado para a criação do programa e o código criado são respectivamente:

```
Função busca_binaria_recursiva(vetor, alvo, início, fim):
  Se início > fim:
        Retorne -1

  meio = início + (fim - início)

  Se vetor[meio] == alvo:
  Retorne meio
  Senão Se vetor[meio] < alvo:
  Retorne busca_binaria_recursiva(vetor, alvo, meio + 1, fim)
  Senão:
  Retorne busca_binaria_recursiva(vetor, alvo, início, meio - 1)
```

```
int binary_search(int array[],int sPtr, int ePtr, int num){

        if( sPtr > ePtr ) {
                return -1;
        }

        int mPtr = sPtr + ((ePtr - sPtr) / 2);
        if( array[mPtr] == num ) {
                return 1;
        } else if( array[mPtr] < num ) {
          return binary_search( array, mPtr + 1, ePtr,num);
        } else {
          return binary_search( array, sPtr, mPtr - 1 ,num);
        }

}
```

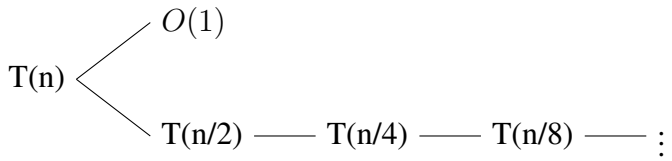
A implementação da busca binária foi feita de forma recursiva, isto é, a função durante sua execução chama a si mesma até que ela chegue na resposta ou chegue no caso base e retorne finalizando assim a recursão.

Para sua analise seria possível utilizar o método da substituição para encontrar seu tempo assintótico isto é: “O método de substituição para resolver recorrências envolve duas etapas: 1. Arriscar um palpite para a forma da solução, 2. Usar indução para determinar as constantes e mostrar que a solução funciona.” [Thomas H. Cormen, Charles

E. Leiserson, Ronald L. Rivest, and Clifford Stein 2009] [1], mas é difícil achar um bom palpite e por isso podemos usar o método da árvore de recursão:

"Em uma árvore de recursão, cada nó representa o custo de um único subproblema em algum lugar no conjunto de invocações de função recursiva. Somamos os custos em cada nível da árvore para obter um conjunto de custos por nível e depois somamos todos os custos por nível para determinar o custo total de todos os níveis da recursão." [Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein 2009] [1]

Podemos analizar que a cada vez que a função é chamada ela verifica o elemento do meio e caso ele não for o elemento procurado ela divide no meio o vetor e chama a si mesma novamente:



$$T(n) = T(n/2) + O(1)$$

Com isso podemos perceber que essa árvore tem $log(n)$ níveis cada um custando $O(1)$ por conta da comparação, somando todos os custos temos que o algoritmo terá tempo $O(log\ n)$. Podemos agora confirmar usando o método mestre para relações recursivas: "O método mestre fornece uma receita para resolver recorrências da forma $T(n) = aT\left(\frac{n}{b}\right) + \Theta(n^k)$" [Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein 2009] [1]

onde:

- $a \geq 1$ é o número de subproblemas em cada etapa,
- $b > 1$ é o fator pelo qual o tamanho do problema é reduzido,
- $\Theta(n^k), k > 0$ é o custo adicional fora da recursão.

O Teorema Mestre fornece três casos para determinar a complexidade assintótica de $T(n)$:

1. Se $a > b^k$, então:
$$T(n) = \Theta\left(n^{\log_b a}\right)$$

2. Se $a = b^k$, então:
$$T(n) = \Theta\left(n^k \log n\right)$$

3. Se $a < b^k$, então:
$$T(n) = \Theta(n^k)$$

No caso da Busca binária temos que $a = 1, b = 2, k = 0$, logo ele cai no segundo caso onde $1 = 2^0$, sendo assim, substituimos e chegamos que a busca binária tem tempo $\Theta\ log(n)$.

5. Discussão

Partindo destas informações podemos discorrer sobre o comportamento real e o esperado destes testes: em todas as buscas são realizadas "q"procuras no vetor de tamanho "n", isto

é: Para a busca sequencial será assintoticamente no pior caso $q \cdot O(n)$, para a otimizada será $O(n\ log\ n) + q \cdot O(n)$ pois o vetor sofreu uma previa ordenação e a busca binária será $O(n\ log\ n) + q \cdot O(log\ n)$ pelo mesmo motivo, podemos pensar que temos agora 2 tempos para analisar nos algoritmos de busca otimizada e binária, o tempo de ordenação e o tempo de busca, como uma é em função de n e a outra em função de n e q, temos que, se q for menor que n, então a análise das funções de complexidade se mantém de certa forma e caso q seja maior ou igual a n a função se altera. Vejamos melhor como isso se aplica:

Quando q é menor do que n, o impacto de q nas funções de complexidade é reduzido em comparação com os termos que dependem diretamente de n. Vamos revisar as três funções sob a condição de que $q < n$.

Busca sequencial este caso, a complexidade é:

$$q \cdot O(n)$$

Como $q < n$, a complexidade permanece linear em relação a n, mas com um fator de escala menor. No entanto, a notação Big-O ignora constantes multiplicativas, então a complexidade final ainda é:

$$O(n)$$

Busca Sequencial Otimizada neste caso, a função:

$$O(n \log n) + q \cdot O(n)$$

Quando $q < n$, o termo $q \cdot O(n)$ é menor do que $O(n^2)$ e menor do que $O(n \log n)$ para valores grandes de n. Portanto, a complexidade dominante é:

$$O(n \log n)$$

Busca binária com função:

$$O(n \log n) + q \cdot O(\log n)$$

Mesmo que $q < n$, o termo $q \cdot O(\log n)$ é muito menor do que $O(n \log n)$, especialmente à medida que n cresce. Assim, a complexidade final permanece:

$$O(n \log n)$$

Em resumo, quando q é menor que n, o impacto de q é menor e não altera a complexidade dominante das funções, que é determinada principalmente pelos termos envolvendo n, agora vamos analisar com o n igual a q.

Se q for igual a n, então a análise das funções de complexidade muda da seguinte maneira, quando $q = n$, a função $q \cdot O(n)$ se transforma em:

$$q \cdot O(n) = n \cdot O(n) = O(n^2)$$

Portanto, a complexidade final é $O(n^2)$.

Substituindo q por n, na função da otimizada se torna:

$$O(n \log n) + q \cdot O(n) = O(n \log n) + n \cdot O(n) = O(n \log n) + O(n^2)$$

Entre $O(n \log n)$ e $O(n^2)$, o termo $O(n^2)$ domina, então a complexidade final é:

$$O(n^2)$$

Na Binária, a função se transforma em:

$$O(n \log n) + q \cdot O(\log n) = O(n \log n) + n \cdot O(\log n) = O(n \log n) + O(n \log n)$$

Como ambos os termos têm a mesma complexidade, a complexidade final continua sendo:

$$O(n \log n)$$

Portanto, quando $q = n$, a função $q \cdot O(n)$ se torna $O(n^2)$, e qualquer função contendo esse termo será dominada por $O(n^2)$, exceto quando comparada a uma função que já é menor, como $O(n \log n)$.

Correlacionando estes dados com os resultados alcançados podemos notar que a analise assintótica representa como os algoritmos se comportam na realidade, sua similaridade com os graficos reforça que a implementação dos algoritmos foi executada de forma correta.

6. Considerações Finais

Com isso podemos afirmar que para um número pequeno de buscas no vetor, a medida que este cresce, os códigos que fazem uso do ordenamento do vetor se tornam piores pois o tempo de ordenação fica alto e não compensa pela quantidade de buscas, já quando as buscas começam a aumentar o tempo de busca fica bem melhor para a busca binária, já a busca otimizada começa a se aproximar da busca sequencial mas até o final dos experimentos ela não a supera, isso se deve ao tempo de ordenação não compensar o suficiente até o final, em comparação ao tempo economizado na busca, caso o vetor já esteja ordenado a busca otimizada e binária vão bem melhores pelo fato de não precisar calcular o preço de ordenação na soma do tempo destes algoritmos.

Podemos assim dizer que para vetores pequenos a busca sequencial é uma escolha razoável caso o vetor não esteja ordenado, já quando trabalhamos com vetores grandes que necessitam de ordenação, podemos trabalhar com a busca binária pois é um algoritmo

bem mais rapido que a busca sequencial otimizada. Esta forma aprimorada da sequencial não chegou a se destacar em nenhum dos testes efetuados e se olharmos pela analise assintótica, ela só se sai melhor quando o tamanho do vetor e a quantidade de buscas é alto, superando assim o tempo gasto ordenando o vetor, más mesmo assim aprensenta um desempenho pior que a busca binária, sendo ela a que mais se destaca em termos de vetores grandes.

E agora podemos ponderar também que não tem correspondencia entre a analise assintótica e a ciclomática, enquanto uma está analisando desempenho a outra está mais focada na complexidade estrutural do código tendo mais afinidade com a parte de engenharia de software.

Referências

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. 3rd ed. The MIT Press, 2009.

[2] Ashutosh Krishna. *Search Algorithms: Linear and Binary Search Explained*. 2024. Disponível em: https://www.freecodecamp.org/news/search-algorithms-linear-and-binary-search-explained/. Acesso em 2 de agosto de 2024.

[3] Instituto de Matemática e Estatística, Universidade de São Paulo. *Busca Sequencial*. 2019. Disponível em: https://panda.ime.usp.br/panda/static/pythonds_pt/05-OrdenacaoBusca/BuscaSequencial.html. Acesso em 2 de agosto de 2024.

[4] Udi Manber. *Introduction to Algorithms*. Addison-Wesley, 1989. ISBN 978-0201120370.

[5] Paulo Feofiloff. *Merge Sort*. 2024. Disponível em: https://www.ime.usp.br/~pf/algoritmos/aulas/mrgsrt.html. Acesso em 2 de agosto de 2024.

[6] Microsoft, Métricas de Código - Complexidade Ciclomática, 2022, https://learn.microsoft.com/pt-br/visualstudio/code-quality/code-metrics-cyclomatic-complexity?view=vs-2022, Acessado em: 5 de agosto de 2024.

[7] Sonia Kuwelkar and Joella Lobo. Performance analysis of merge sort algorithms. *ResearchGate*, 2020. URL: https://www.researchgate.net/profile/Sonia-Kuwelkar/publication/343434465_Performance_Analysis_of_Merge_Sort_Algorithms/links/60f860ed2bf3553b29029346/Performance-Analysis-of-Merge-Sort-Algorithms.pdf.

[8] Flávio Moura. Aulas 05 e 06 de PAA 2023-1. 2023. URL: http://flaviomoura.info/files/paa-2023-1-aulas05e06.pdf, Acessado em: 07-08-2024.