UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CAMPUS NATAL
BACHARELADO EM TECNOLOGIA DA INFORMAÇÃO
ESTRUTURA DE DADOS BÁSICAS I

ALISON HEDIGLIRANES DA SILVA
FELIPE MORAIS DA SILVA

ANÁLISE EMPÍRICA: ALGORITMOS DE BUSCA

NATAL/RN
03/2018

ALISON HEDIGLIRANES DA SILVA
FELIPE MORAIS DA SILVA

ANÁLISE EMPÍRICA: ALGORITMOS DE BUSCA

Trabalho acadêmico apresentado ao Ph.D. Selan Rodrigues Dos Santos, ministrante da disciplina de Estrutura de Dados Básicas I, da Universidade Federal do Rio Grande do Norte, campus Natal, como uma das atividades avaliativas da primeira unidade.

NATAL/RN
03/2018

RESUMO

Este relatório descreve a análise empírica realizada sobre algoritmos de busca para a disciplina de Estrutura de Dados Básicas I, na UFRN, ministrada pelo professor Ph.D Selan Rodrigues. O referido documento apresentará uma breve contextualização acerca dos algoritmos de busca; os conceitos necessários para a compreensão geral do assunto; os materiais e métodos utilizados; os resultados obtidos; as discussões realizadas; e, por fim, as devidas considerações finais.

Palavras-chave: Estrutura de Dados. Análise Empírica. Algoritmos de Busca.

SUMÁRIO

1 INTRODUÇÃO

A busca de informação em arranjos é um problema computacional muito recorrente por possuir uma alta importância. No geral, existem diversos algoritmos de busca que possuem vantagens e desvantagens conforme o cenário de aplicação.

Sendo assim, como é possível mensurar o custo computacional de um algoritmo de busca? Quais cenários são mais favoráveis para determinado algoritmo?

Essas perguntas podem ser respondidas por meio da análise de complexidade dos algoritmo, de forma teórica e prática. Ou seja, é possível analisar a eficiência desses algoritmos não apenas de forma matemática como de forma empírica e, ainda, comparar os resultados.

Assim, o objetivo deste relatório é, em primeiro momento, apresentar alguns algoritmos de busca para, posteriormente, submetê-los a testes com o intuito de obter resultados para traçar uma discussão acerca desses. Os algoritmos que serão analisados são: (i) busca linear iterativa; (ii) busca binária iterativa e recursiva; (iii) busca ternária iterativa e recursiva; (iv) busca de salto; (v) busca Fibonacci.

O referido trabalho, na seção 2, apresenta uma série de conceitos fundamentais para a compreensão do relatório; na seção 3, os materiais e métodos do experimento; na seção 4, os resultados obtidos com base na metodologia seguida; na seção 5, as discussões que podem ser realizadas com base nos resultados; e, na seção 6, as devidas considerações finais.

2 FUNDAMENTAÇÃO TEÓRICA

Esta seção irá descrever os conceitos necessários para compreender o conteúdo que será abordado nas próximas seções.

O conceito elementar numa discussão de análise empírica de algoritmos é, em primeiro momento, o do próprio “algoritmo”. Conforme Cormen et.al. (2009), um algoritmo é um procedimento bem definido que, a partir de um conjunto de valores de entrada (chamados de instância), é possível gerar um conjunto de valores de saída (chamados de solução). O conjunto de pares instância/solução é chamado de problema computacional.

Assim, é possível visualizar um algoritmo como uma ferramenta usada na resolução de problemas computacionais. Desta forma, um problema computacional estabelece os pares de instância e solução enquanto que o algoritmo especifica os procedimentos necessários para alcançar a relação de entrada e saída esperada.

2.1 O Estudo de Algoritmos

Ao estudar algoritmos é necessário levar em consideração dois aspectos: (i) corretude; (ii) complexidade.

Um algoritmo é dito correto quando possui um comportamento capaz de gerar as saídas esperadas com base nas entradas fornecidas (WIKIPEDIA, 2018b). Ainda, existe uma distinção entre corretude parcial, a qual necessita do retorno de uma solução, e corretude total, a qual, adicionalmente, exige que o algoritmo termine.

A complexidade, por sua vez, é medida com base na quantidade de recursos (tempo de execução, armazenamento, dentre outros) usados para realizar uma tarefa específica (GÁCS & LOVÁSZ, 1999).

2.1.1 NOTAÇÃO GRANDE-O

Para mensurar a complexidade algorítmica, é comum utilizar a notação Grande-O (do inglês, “Big-O”) para denotar o limite superior.

Definindo-se formalmente, conforme MIT (2018), suponha que $f(n)$ e $g(n)$ são duas funções definidas sobre algum subconjunto dos números Reais. Assim, diz-se que $f(n) = O(g(n))$ para $n \to \infty$ se e somente se existe uma constante N e C tal que $|f(n)| \leq C\,|g(n)|$ para todo $n > N$.

Então, se a é um número real, tem-se que $f(n) = O(g(n))$ para $n \to a$ se e somente se existirem constantes $d > 0$ e C tais que $|f(n)| \leq C\,|g(n)|$ para todo n com $|n-a| < d$.

Na Figura/Tabela abaixo, é possível visualizar uma lista de classes de funções que são encontradas comumente ao analisar a complexidade de algoritmos.

Figura 1 - Classes de funções recorrentes

Notação	Nome
$O(1)$	Constante
$O(log(n))$	Logarítmica
$O((log(n))^c)$	Poli Logarítmica
$O(n)$	Linear
$O(n^2)$	Quadrática
$O(n^c)$	Polinomial
$O(c^n)$	Exponencial

Fonte: http://web.mit.edu/16.070/www/lecture/big_o.pdf

Além disso, existem as notações Omega, Theta e Pequeno-o (do inglês, "Little-o") que, mesmo não sendo utilizadas no decorrer do referido relatório, é necessário às apresentar para melhor compreensão. Abaixo, seguem as notações, conforme Cathyatseneca (2018):

- **Omega:** $f(n) = \Omega(g(n))$ se e somente se, para uma constante C e N, $|f(n)| \geq C\,|g(n)|$ para $n \geq N$;
- **Theta:** $f(n) = \Theta(g(n))$ se e somente se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$;
- **Pequeno-O:** $f(n) = o(g(n))$ se e somente se $f(n) = O(g(n))$ e $f(n) \neq \Theta(g(n))$.

2.2 Problemas de Busca

Os problemas de busca são problemas computacionais representados por uma relação binária entre todos os pares instância-solução e, de maneira geral, consistem na tentativa de encontrar um objeto x em uma estrutura y (WIKIPEDIA, 2018f)

A seguir, serão explicados os seguintes algoritmos de busca: (i) busca linear; (ii) busca binária; (iii) busca ternária; (iv) busca de salto; (v) busca Fibonacci.

2.2.1 Busca Linear

A busca linear é um método utilizado para encontrar um valor alvo dentro de um arranjo. O algoritmo irá percorrer todo o arranjo comparando cada elemento com o valor alvo e, caso encontre um elemento igual, irá retornar sua posição (WIKIPEDIA, 2018e)

Assim, dado um arranjo A com n elementos, tem-se as posições $A[i],\ A[i+1],\ \ldots,\ A[n-1]$ para $0 \le i < n \in N^*$. Para encontrar um valor x nesse arranjo, compare cada i-ésimo elemento com x e, caso os dois valores sejam iguais, retorne o i em que a comparação foi verdadeira.

Note que, no melhor caso, o valor procurado estará na primeira posição a ser comparada ($A[0]$), independentemente do tamanho do arranjo. Por isso, sua complexidade será $O(1)$, ou seja, constante. No pior caso, o valor procurado não estará no arranjo e todas suas respectivas posições serão comparadas, isto é, se o arranjo possui n elementos, serão realizadas n comparações. Desta forma, neste caso o algoritmo possui complexidade linear ou $O(n)$ (GEEKSFORGEEKS, 2018).

2.2.2 Busca Binária

A busca binária é um método utilizado para encontrar um valor dentro de um arranjo ordenado. A ideia por trás do algoritmo segue o princípio de dividir para conquistar.

Tome um arranjo ordenado A com n elementos tais que $A[i] \le A[i+1] \le \ldots \le A[n-1]$ para $0 \le i < n \in N^*$. Para encontrar o elemento x em A verifique se $A[\frac{n}{2}]$ é igual a x. Caso seja, o valor fora encontrado na posição

$i = \frac{n}{2}$ do arranjo. Caso o valor não seja encontrado, é necessário verificar se x é maior ou menor que $A\,[\frac{n}{2}]$. Se $x > A[\frac{n}{2}]$, então basta repetir todo o processo descrito anteriormente no intervalo de $A[j]$ a $A[k]$ tal que $j, k \in [\frac{n}{2} + 1,\ n)$ e $j < k$, ou seja, de $A[\frac{n}{2} + 1]$ a $A[n - 1]$, inclusive. Analogamente, se $x < A[\frac{n}{2}]$, o processo será repetido no intervalo de $j, k \in [0,\ \frac{n}{2})$, ou seja, de $A[0]$ a $A[\frac{n}{2} - 1]$, inclusive. Note que, a cada subdivisão de intervalos n será menor.

Esse processo de subdivisão deve se repetir enquanto $j < k$ for uma condição verdadeira. Assim, quando $k = j + 1$, o intervalo de $A[j]$ a $A[k]$, para $j, k \in (j\ ,\ j + 1]$ possuirá somente um elemento, $A[j]$, que, caso não seja igual a x, então é possível concluir que o valor procurado não se encontra no arranjo.

Se, na busca binária, o valor procurado estiver na posição $i = \frac{n}{2}$, tem-se o melhor caso, pois será realizada somente uma comparação (independentemente do tamanho de A) e, portanto, a complexidade será $O(1)$ ou linear. No pior dos casos, aquele em que o elemento x não está em A, serão realizadas $floor(log_2(n) + 1)$ comparações, ou seja, a complexidade é $O(log\, n)$ ou logarítmica (WIKIPEDIA, 2018a).

2.2.3 Busca Ternária

A busca ternária é um método utilizado para encontrar um valor dentro de um arranjo ordenado. A ideia por trás do algoritmo segue o princípio de dividir para conquistar e é similar à busca binário, porém, ao dividir um arranjo em duas partes, divide-o em três.

Tome um arranjo ordenado A com n elementos tais que $A[i] \leq A[i + 1] \leq ... \leq A[n - 1]$ para $0 \leq i < n \in N^*$. Analogamente à busca binária, para encontrar um valor x em A é necessário verificar, inicialmente, se $x = A[\frac{n}{3}]$ ou $x = A[\frac{2n}{3}]$. Assim, caso x não seja igual a nenhum desses valores, basta verificar em qual intervalo esse se encontra e repetir o processo nos subintervalos respectivos até haver somente um único elemento que, caso seja diferente de x, então o valor não será encontrado no arranjo.

No melhor dos casos o valor será um dos elementos que dividem arranjo em três blocos e, por isso, tem complexidade linear ou $O(1)$, já que não irá importar o tamanho de A. No pior dos casos, onde x não está em A, a complexidade é logarítmica, isto é, $O(log\ n)$ (WIKIPEDIA, 2018g).

2.2.4 Busca de Salto

A busca de salto é um método utilizado para encontrar um valor dentro de um arranjo ordenado. O algoritmo irá percorrer o arranjo a um passo pré-estabelecido e ao perceber que o valor procurado está entre um intervalo definido entre dois saltos, aplica-se a busca linear no intervalo menor.

Dado um arranjo A com n elementos tais que $A[i] \leq A[i+1] \leq ... \leq A[n-1]$ para $0 \leq i < n \in N^*$, $p = \sqrt{n}$ é a constante de salto. Assim, para encontrar x em A, itera-se cada elemento de A ao passo p até encontrar um valor em que $x < A[p \cdot k]$, para $1 \leq k \leq \sqrt{n}$. Após encontrada a posição que satisfaz essa condição, utiliza-se a busca linear no intervalo de $A[p \cdot (k-1)]$ a $A[p \cdot k]$.

O melhor caso é quando o elemento procurado está na primeira posição do vetor já que, independentemente do tamanho de A, será constante a complexidade, isto é, $O(1)$. Por outro lado, o pior caso é aquele em que o elemento não se encontra no arranjo. Assim, em suma, serão necessários $2\sqrt{n}$ iterações, para não encontrar o valor, ou seja, a complexidade é $O(\sqrt{n})$ (WIKIPEDIA, 2018d).

2.2.5 Busca Fibonacci

A busca binária é um método utilizado para encontrar um valor dentro de um arranjo ordenado. A ideia por trás do algoritmo segue o princípio de dividir para conquistar.

De forma geral, a busca Fibonacci utiliza os números da sequência de Fibonacci como índice para encontrar um valor em um arranjo. Ainda, a sequência de Fibonacci é definida recursivamente da seguinte forma:

$F(1) = 1$

$F(2) = 1$

$F(n) = F(n-1) + F(n-2)$

Tome um arranjo ordenado A com n elementos tais que $A[i] \leq A[i+1] \leq ... \leq A[n-1]$ para $0 \leq i < n \in N^*$. Para encontrar um valor x

em A é necessário encontrar o menor número da sequência de Fibonacci $F(k)$ tal que $F(k) \geq n$. Se for um índice válido, use $j = F(k-2)$ como índice e verifique se $A[j] = x$, caso seja, o algoritmo é finalizado e j é retornado. Caso a condição falhe, será necessário verificar se $x > A[j]$ ou se $x < A[j]$ para descobrir um subintervalo em que x possa estar, semelhante à busca binária. Ao se obter um novo intervalo, aplica-se novamente o processo até então descrito até obter um intervalo que possua somente um único elemento.

O melhor caso ocorre quando X. Já o pior caso ocorre quando x não se encontra no arranjo, configurando complexidade $O(log(n))$, isto é, logarítmica (WIKIPEDIA, 2018c).

3 METODOLOGIA

Esta seção irá apresentar os materiais e as metodologias utilizadas para a execução dos experimentos.

Os algoritmos foram implementados em *C++* e compilados com o *G++*, no computador com as seguintes características (Quadro 1):

Quadro 1 - Configurações usadas nos experimentos

Processador	Intel Core i7 3612QM @ 2.10 GHz
Memória	8 GB DDR3
Placa-mãe	Dell 0RHTCK
Tipo de sistema	64 bits
Sistema	Windows 10
Sistema do Bash	Ubuntu 16.04.4 LTS
Compilador	G++ 5.4.0

Fonte: Autoria própria.

Quanto aos algoritmos de busca, todas as assinaturas das funções operam sobre um intervalo descrito por meio de dois ponteiros, na implementação propriamente dita. Entretanto, para apresentar a lógica por meio de pseudocódigo, optou-se por fazer algumas alterações na assinatura das funções para fins didáticos (a implementação real dos algoritmos pode ser vista no Apêndice A).

O primeiro dos algoritmos a ser implementado foi a busca linear iterativa (Algoritmo 1). Além disso, leia o tipo de variável “número” como qualquer tipo numérico, seja inteiro ou decimal.

Algoritmo 1 - Busca linear iterativa

```
função busca-linear(A: arranjo de números, valor: número):inteiro
    var n: inteiro ← tam A   # tamanho do vetor
    var i: inteiro ← 0       # contador

    # enquanto houver valores a serem comparados
    enquanto i < n faça
        # se encontrar o valor, retorne seu índice
        se A[i] == valor então
            retorna i

        # incrementa o contador
        i ← i + 1
```

```
        # caso não encontre ninguém no arranjo
        retorna -1
```

Fonte: Autoria própria.

O segundo algoritmo foi a busca binária iterativa (Algoritmo 2), seguido de sua versão recursiva (Algoritmo 3).

Algoritmo 2 - Busca binária iterativa

```
função busca-binária-iterativa(A: arranjo de números, valor: número):inteiro
        var n: inteiro ← tam A   # tamanho do vetor
        var i: inteiro ← 0       # índice inicial
        var f: inteiro ← n       # índice final

        # enquanto houver valores a serem comparados
        enquanto i < (f-1) faça
                n ← n/2 # divide tamanho pela metade

                # se encontrar o valor no índice central do subintervalo
                se A[i+n] == valor então
                        retorna n # retorne seu índice

                # se o estiver à esquerda do centro do subintervalo
                se valor < A[i+n] então
                        f ← i + n + 1 # ignora o bloco da direita

                # se o estiver à direita do centro do subintervalo
                senão
                        i ← i + n + 1 # ignora o bloco da direita

        # caso não encontre ninguém no arranjo
        retorna -1
```

Fonte: Autoria própria.

A implementação da busca binária recursiva (Algoritmo 3) necessitou de uma auxiliar para manter a mesma assinatura das demais funções de busca com o intuito de otimizar os testes.

Algoritmo 3 - Busca binária recursiva

```
função busca-binária-recursiva(A: arranjo de números, valor: número) : inteiro
        var n: inteiro ← tam A   # tamanho do vetor
        retorna busca-binária-recursiva-aux(A, valor, 0, n)

função busca-binária-recursiva-aux(A: arranjo de números, valor: número, início:
inteiro, final: inteiro) : inteiro
        var n: inteiro ← final - início       # tamanho do vetor
        var i: inteiro ← n/2                  # índice central

        # se não houver intervalo válido
        se (início >= final) então
                retorna -1 # elemento não encontrado
```

```
      # se encontrar o valor, no índice central do subintervalo
      se A[i + início] == valor então
            retorna i + início; # retorne seu índice

      # se o valor estiver à esquerda do índice central
      senão se valor < A[i + início] então
            faça busca-binária-recursiva-aux(A, valor, início, início + i)

      # se o valor estiver à direita do índice central
      senão
            faça busca-binária-recursiva-aux(A, valor, início + i + 1, final)
```

Fonte: Autoria própria.

O terceiro algoritmo foi a busca ternária iterativa (Algoritmo 4) e recursiva (Algoritmo 5).

Algoritmo 4 - Busca ternária iterativa

```
função busca-ternária-iterativa(A: arranjo de números, valor: número):inteiro
      var n: inteiro ← tam A   # tamanho do vetor
      var i: inteiro ← 0       # índice inicial
      var f: inteiro ← n       # índice final

      # enquanto houver valores a serem comparados
      enquanto i < f faça
            n ← n/3 # divide tamanho por 3

            # se encontrar o valor no índice n/3
            se A[i+n] == valor então
                  retorna i + n # retorne seu índice

            # se encontrar o valor no índice 2n/3
            se A[i+2*n] == valor então
                  retorna i + n # retorne seu índice

            # se o estiver no bloco mais a esquerda
            se valor < A[i+n] então
                  f ← i + n + 1 # ajusta intervalo

            # se o estiver no bloco mais a direita
            senão se valor < A[i+n] então
                  i ← i + 2*n + 1 # ajusta o intervalo

            # se o estiver no bloco central
            senão
                  f ← i + n;
                  i ← i + n + 1 # ajusta o intervalo

      # caso não encontre ninguém no arranjo
      retorna -1
```

Fonte: Autoria própria.

Observe no Algoritmo 5 que a mesma estratégia em Algoritmo 3 foi usada para manter a assinatura da função padrão.

Algoritmo 5 - Busca ternária recursiva

```
função busca-ternária-recursiva(A: arranjo de números, valor: número) : inteiro
      var n: inteiro ← tam A   # tamanho do vetor
      retorna busca-binária-recursiva-aux(A, valor, 0, n)

função busca-ternária-recursiva-aux(A: arranjo de números, valor: número,
início: inteiro, final: inteiro) : inteiro
      var n: inteiro ← final - início      # tamanho do vetor
      var i: inteiro ← n/3                 # índice ternário

      # se não houver intervalo válido
      se (início >= final) então
            retorna -1 # elemento não encontrado

      # se encontrar o valor, no índice central do subintervalo
      se A[i + início] == valor então
            retorna i + início; # retorne seu índice

      # se encontrar o valor, no índice central do subintervalo
      se A[2*i + início] == valor então
            retorna 2*i + início; # retorne seu índice

      # se o valor estiver no bloco mais a esquerda
      se valor < A[i + início] então
            faça busca-ternária-recursiva-aux(A, valor, início, início + i)

      #se o valor estiver no bloco mais a direita
      senão se valor > A[2*i + início] então
            faça busca-ternária-recursiva-aux(A, valor, início + 2*i + 1,
final)

       # se o valor estiver no bloco central
      senão
            faça busca-ternária-recursiva-aux(A, valor, início + i + 1, início
+ 2*i)
```

Fonte: Autoria própria.

O quarto algoritmo foi a busca de salto iterativa (Algoritmo 6). Note que, após definidos os intervalos de busca com base no salto, o algoritmo de busca linear (Algoritmo 1) é realizado no respectivo intervalo e, como ele já foi explicado, sua implementação foi omitida.

Algoritmo 6 - Busca de salto iterativa

```
função busca-de-salto-iterativa(A: arranjo de números, valor: número):inteiro
      var n: inteiro ← tam A   # tamanho do vetor
```

```
    var m: inteiro ← sqrt(n) # tamanho do salto
    var i: inteiro ← 0       # índice inicial
    var f: inteiro ← m       # índice final

    # enquanto houver valores a serem comparados
    enquanto A[f-1] < valor faça
        f ← f + m   # incrementa o final do intervalo
        i ← i + m   # incrementa o início do intervalo

        # se o início do intervalo for maior que n
        se f > n então
            retorna -1 # elemento não existe no vetor

        # se o final do intervalo for maior que n
        se f > n então
            f ← n

        # se o estiver à esquerda do centro do subintervalo
        se valor < A[i+n] então
            f ← i + n + 1 # ignora o bloco da direita

        # se o estiver à direita do centro do subintervalo
        senão
            i ← i + n + 1 # ignora o bloco da direita

    # após definido o intervalo de busca
    retorna busca-linear(busque valor em A de i a f)
```

Fonte: Autoria própria.

Por fim, o quarto algoritmo foi a busca Fibonacci iterativa (Algoritmo 7).

Algoritmo 7 - Busca Fibonacci iterativa

```
função busca-fibonacci-iterativa(A: arranjo de números, valor: número):inteiro
    var n  : inteiro ← tam A       # tamanho do vetor
    var f1 : inteiro ← 0           # (n-2)ésimo termo de fibonacci
    var f2 : inteiro ← 1           # (n-1)ésimo termo de fibonacci
    var fn : inteiro ← f1 + f2     # n-ésimo termo de fibonacci
    var lim: inteiro ← -1          # verificador de limite de índices

    # encontrar o menor termo de fibonacci maior que n
    enquanto fn < n faça
        f1 ← f2
        f2 ← fn
        fn ← f1 + f2

    # enquanto houver elementos a serem comparados
    enquanto a faça
        var val ← min (lim + f1, n - 1)

        # se valor for menor que o valor na posição val
        # cada elemento volta dois termos na sequência fibonacci
        se A[val] > valor então
```

```
                f3 ← f1
                f2 ← f2 - f1
                f1 ← f3 - f2

        # se valor for maior que o valor na posição val
        # cada elemento volta um termo na sequência fibonacci
        senão se A[val] < valor então
                f3  ← f2
                f2  ← f1
                f1  ← f3 - f2
                lim ← val

        # se o valor for igual ao valor na posição val
        senão
                retorna val # retorna o índice

    # caso o elemento não seja encontrado
    retorna -1
```

Fonte: Autoria própria.

Para executar os experimentos e testar esses algoritmos, foi implementado uma função *main* para tal. De forma resumida, a função irá executar cada função de busca uma quantidade pré-estabelecida de vezes para obter sua média relativa a cada amostra, isto é, tamanho de arranjo. O pseudocódigo pode ser visto com mais detalhes abaixo no Algoritmo 8.

Algoritmo 8 - Função main

```
função main () : inteiro
    var qtd-amostras : inteiro
    var tam-amostras : inteiro
    var funções-de-busca : arranjo de funções

    leia(qtd-amostras) # lê quantidade de amostras
    leia(tam-amostras) # lê o tamanho de incremento de amostras

    # instancia um vetor de tamanho (qtd-amostras * tam-amostras)
    # de inteiros longos com valores ordenados
    instanciar-vetor(qtd-amostras * tam-amostras)

    enquanto houver amostras faça
        para x em funções-de-busca faça
            var media : decimal ← 0
            repita MEDIA vezes
                media ← (media + calcular-tempo(x))/MEDIA
            imprima(media)

    # fim do main
    retorna 0;
```

Fonte: Autoria própria.

Além disso, Algoritmo 8 recebe duas entradas (número de amostras e tamanho das amostras) e retorna uma saída relativa aos tempos em nanossegundos de cada função para uma amostra específica. Cada execução de uma função em particular para uma amostra específica é de MEDIA vezes. Nos testes realizados, MEDIA é igual a 100.

Para a realização propriamente dita da análise empírica, foram utilizadas 25 amostras, cada uma crescendo ao passo de 40 milhões, ou seja, na 25ª amostra, o tamanho do arranjo seria de um bilhão de elementos. Além disso, foram considerados os piores casos, isto é, aquele em que o elemento procurado não está no vetor.

Após executado o programa, todos os resultados foram gravados em um arquivo .csv dividido em oito colunas (o qual pode ser visto no Apêndice B) onde a primeira era sobre o tamanho da amostra e as demais eram sobre o tempo de cada função em ms.

Por fim, para gerar os gráficos expostos nos resultados, foi utilizado o software gnuplot, colocando o tempo do algoritmo em função do tamanho das amostras.

4 RESULTADOS

Esta seção irá apresentar os resultados obtidos após a execução de 25 amostras de cada um dos seguintes algoritmos: (i) busca linear iterativa; (ii) busca binária iterativa e recursiva; (iii) busca ternária iterativa e recursiva; (iv) busca de salto iterativo; (v) busca fibonacci iterativa.

Foram testados somente os piores casos para cada algoritmo, ou seja, quando o elemento procurado não se encontra no arranjo, e os dados não mostraram muita variância de tempo à medida que o tamanho das amostras ia crescendo (exceto na busca linear).

Antes de mais nada, é importante ressaltar que a partir de uma quantidade de amostras bem grande, o tempo de execução dos algoritmos possuiu alguns picos devido à quantidade de memória utilizada nesse processo (por volta de 98% do total de memória). Segue abaixo, na Tabela 1, os dados do experimento.

Tabela 1 - Tempo de execução dos algoritmos de busca

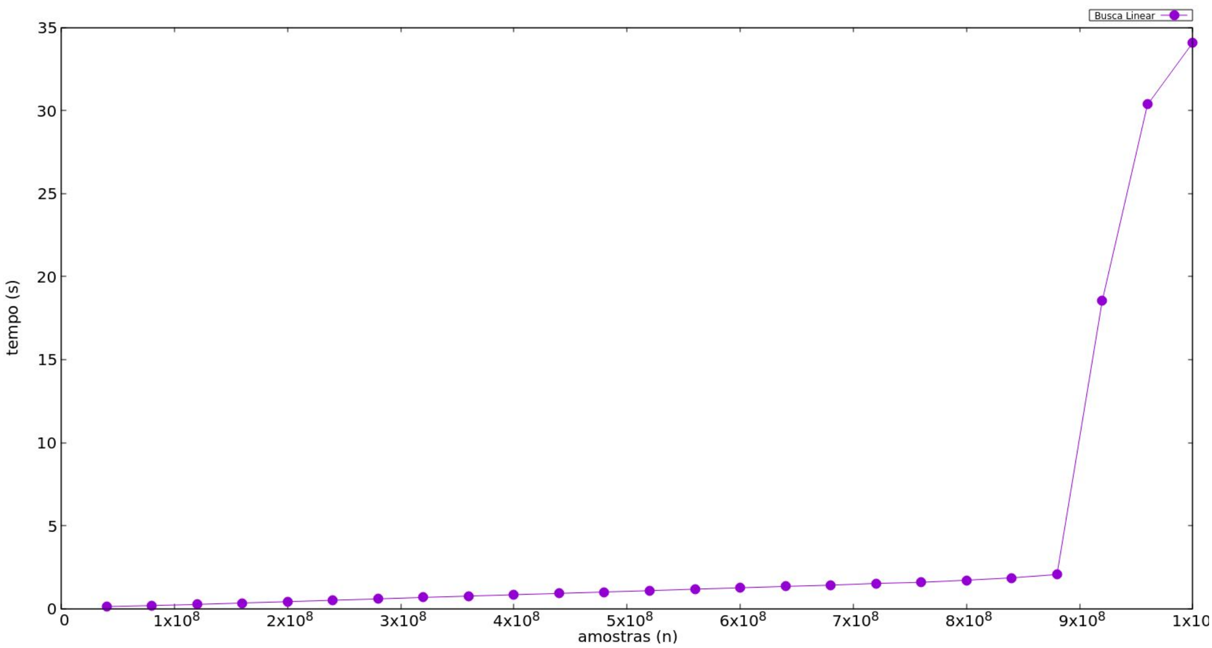
Tamanho da amostra (em milhões)	**Busca linear** (s)	**Busca binária iterativa** (ns)	**Busca binária recursiva** (ns)	**Busca ternária iterativa** (ns)	**Busca ternária recursiva** (ns)	**Busca de salto** (ms)	**Busca Fibonacci** (ns)
40	0,13	1,22	1,39	1,25	1,35	0,11	2,37
80	0,18	1,14	1,59	1,34	1,34	0,16	1,65
120	0,26	1,15	1,31	1,15	1,24	0,19	2,41
160	0,34	1,15	1,36	1,14	1,28	0,24	1,54
200	0,42	1,15	1,35	1,15	1,27	0,26	1,78
240	0,50	1,22	1,36	1,18	1,29	0,29	1,57
280	0,59	1,18	1,37	1,26	1,28	0,30	1,90
320	0,68	1,16	1,36	1,18	1,26	0,41	1,57
360	0,76	1,17	1,33	1,16	1,25	0,35	1,59
400	0,84	1,14	1,38	1,18	1,27	0,44	1,56
440	0,92	1,16	1,36	1,26	1,25	0,45	1,85
480	1,00	1,27	1,48	1,27	1,36	0,44	2,01
520	1,09	1,31	1,33	1,17	1,27	0,51	2,11

560	1,17	1,14	1,34	1,15	1,24	0,53	2,25
600	1,25	1,15	1,36	1,15	1,27	0,49	1,55
640	1,34	1,16	1,35	1,17	1,36	0,52	1,64
680	1,42	1,20	1,36	1,22	1,33	0,53	1,65
720	1,52	1,17	1,38	1,18	1,31	0,61	1,80
760	1,58	1,20	1,36	1,15	1,32	0,67	1,75
800	1,72	1,17	1,34	1,16	1,28	0,58	2,32
840	1,86	1,16	1,32	1,11	1,29	0,61	2,36
880	2,06	1,28	1,51	1,32	1,46	0,76	2,21
920	18,54	1,31	1,47	1,15	1,41	4,42	3,77
960	30,38	1,33	1,51	1,50	1,44	6,55	2,95
1000	34,10	1,20	1,39	1,20	1,32	4,11	3,03

Fonte: Autoria própria

Analisando-se algoritmo a algoritmo, o primeiro a ser observado é a busca linear. No Gráfico 1, é possível visualizar seu comportamento, isto é, o crescimento do tempo em relação ao tamanho das amostras. Como esperado, seu tempo de execução cresceu de forma linear até certo ponto, porém teve picos em tamanhos amostrais maiores que 920 milhões.

Gráfico 1 - Busca linear

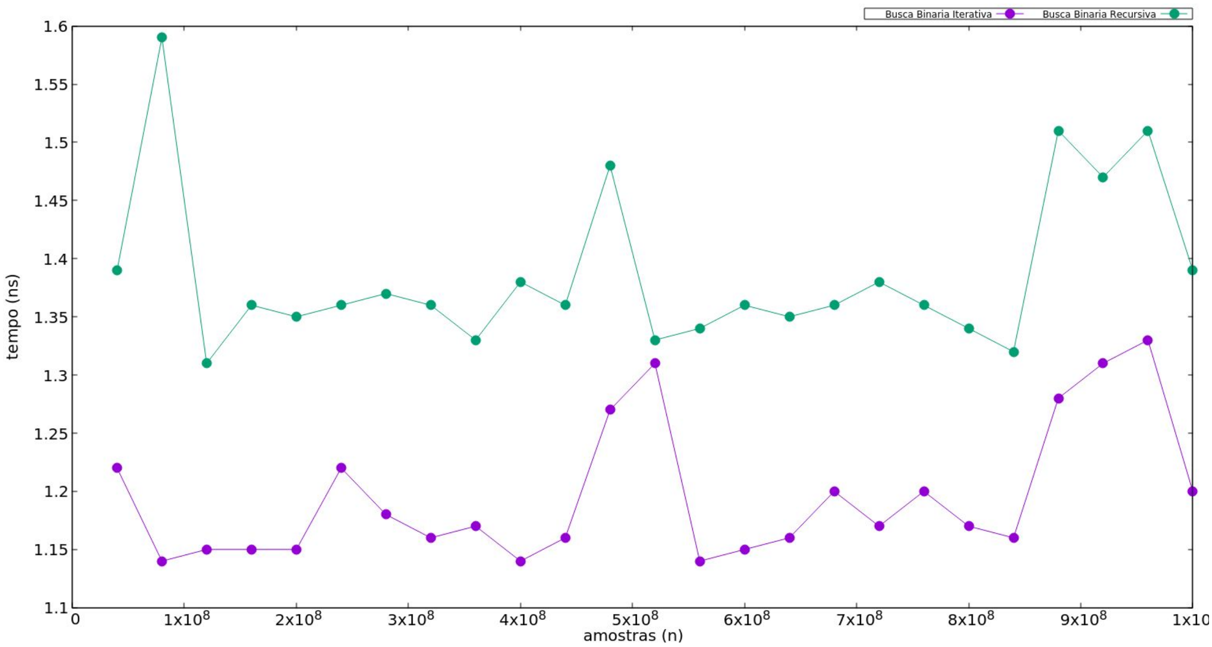


Fonte: Autoria própria

Passando para os algoritmos de busca binária, é possível ver, no Gráfico 2, que as implementações iterativa e recursiva, no geral, tiveram um excelente desempenho, sendo perceptível a diferença analisando na escala dos nanossegundos. Além disso, a abordagem iterativa mostrou um desempenho superior à recursiva.

Note que, por se tratar de uma escala de tempo muito pequena, os valores se comportam de forma um pouco aleatória se comparados com $O(log\ n)$. Isso ocorre pois, numa escala de tempo tão pequena, as variações internas do computador têm maior influência sobre os resultados.

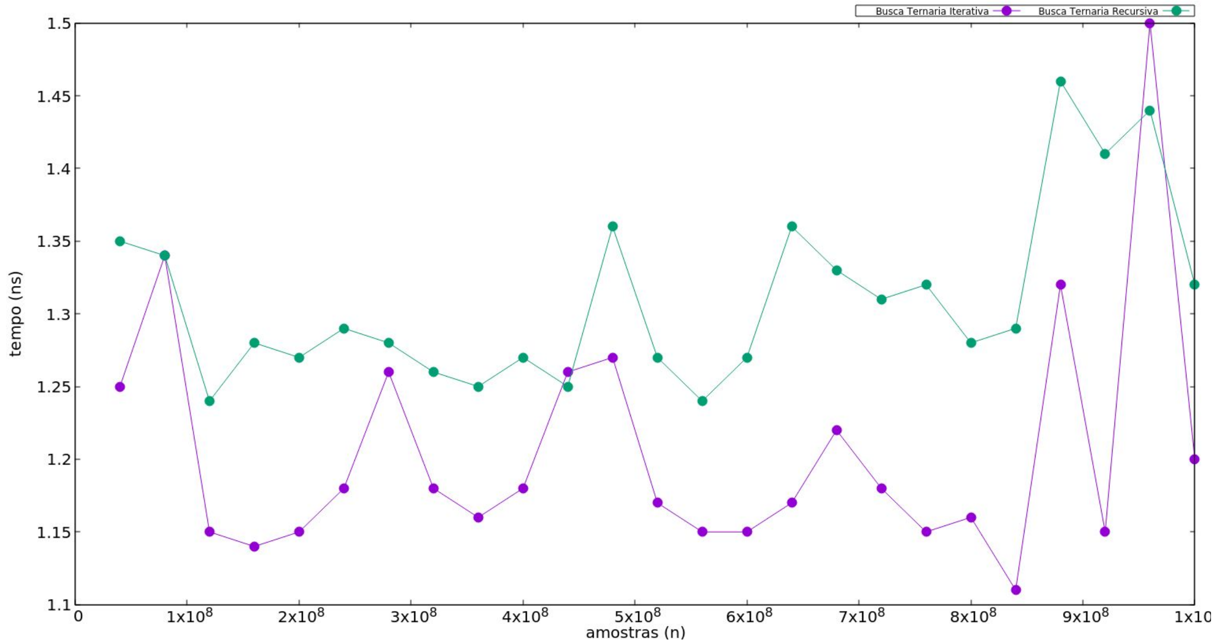
Gráfico 2 - Busca binária



Fonte: Autoria própria

Os algoritmos de busca ternária (iterativa e recursiva), tiveram desempenho semelhante à busca binária e a análise é análoga. Conforme o Gráfico 3, é possível visualizar que, na maioria dos casos, a abordagem iterativa se mostrou mais eficiente.

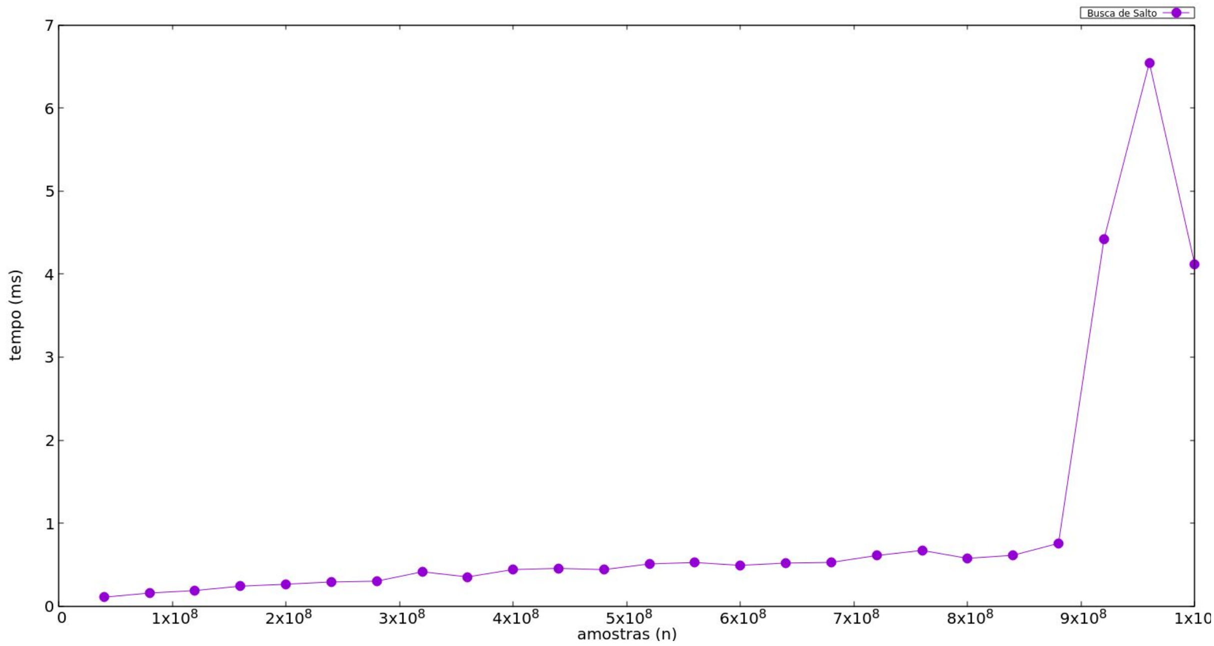
Gráfico 3 - Busca ternária



Fonte: Autoria própria

Quanto à busca de salto (Gráfico 4), seu crescimento aparenta ser linear, embora devesse ser $O(\sqrt{n})$. A partir de amostras maiores que 920 milhões de elementos, a busca de salto (similar à linear), apresenta um pico ocasionado por uma alta utilização da memória. No geral, a busca de salto é mais eficiente que a linear, porém menos eficiente do que as funções de busca.

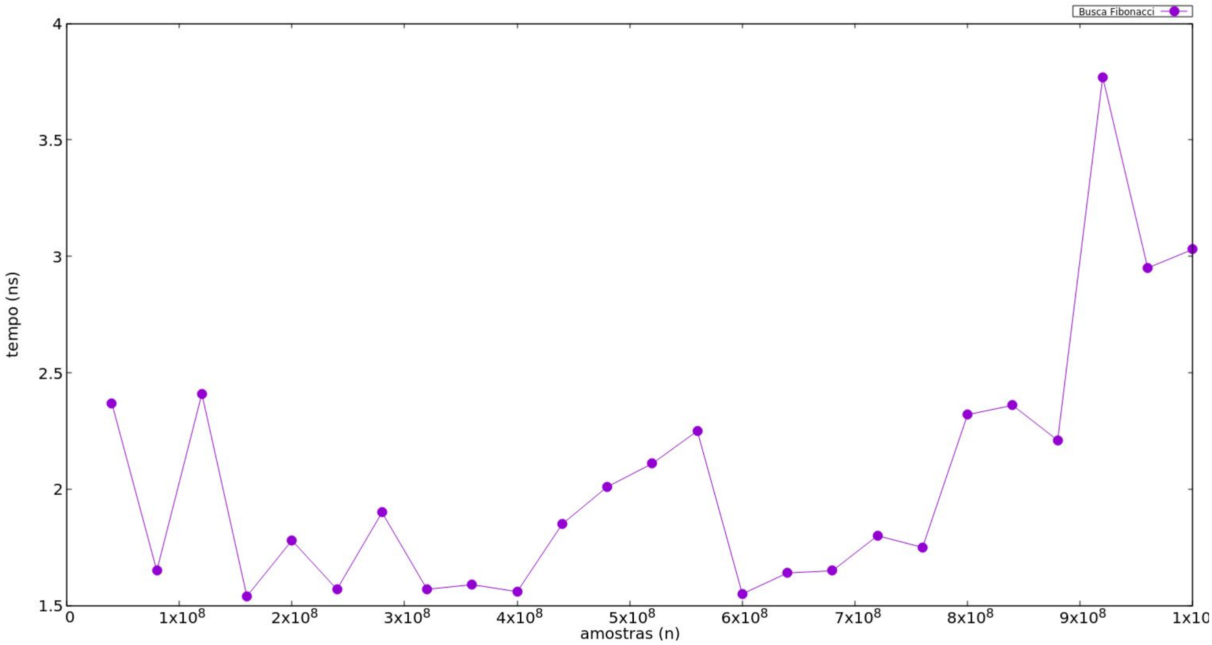
Gráfico 4 - Busca de salto



Fonte: Autoria própria

No que se refere à busca Fibonacci, observando o Gráfico 5, percebe-se que as medições estão dispostas de forma mais aleatória. Isso ocorre pois, além dos fatores relativos ao computador, a própria natureza do algoritmo permite perceber que um valor não está no arranjo de forma mais rápida em alguns casos específicos.

Gráfico 5 - Busca Fibonacci

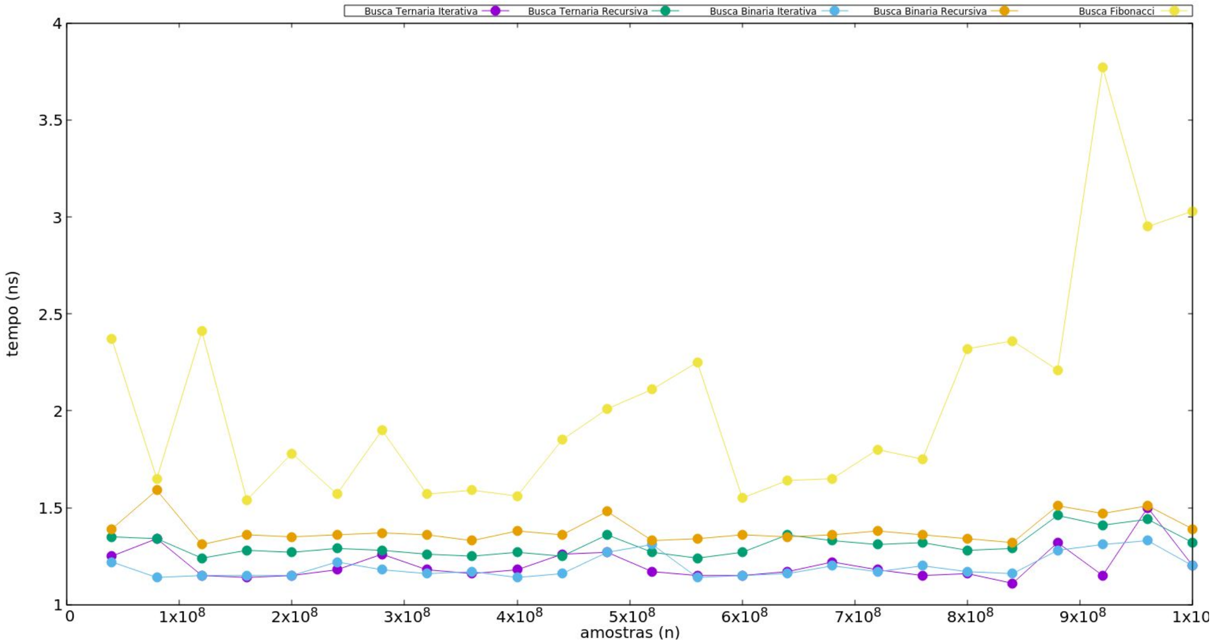


Fonte: Autoria própria

Desta forma, é perceptível que o algoritmo menos eficiente é a busca linear, seguido da busca de salto. Quanto aos demais algoritmos, nota-se que as versões iterativas são superiores às recursivas, na maioria dos casos. No Gráfico 6, têm-se todos os algoritmos de busca $O(log\ n)$ sendo possível perceber que, em todas as amostras, a busca Fibonacci se mostrou inferior.

Dentre a busca binária e a busca ternária, os resultados não são 100% conclusivos para apontar qual das duas abordagens é mais superior. De qualquer forma, é necessário levar em consideração que, a quantidade de subdivisões do intervalo dado não representou performance significativa.

Gráfico 6 - Comparação entre busca binária, ternária e Fibonacci



Fonte: Autoria própria

Por fim, os resultados foram próximos do esperado e as principais conclusões são: (i) em escalas de medição de tempo pequenas, os dados podem possuir muito ruído; (ii) algoritmos $O(n)$ podem se tornar inviáveis para uma quantidade de amostras alta; (iii) as abordagens iterativas, no geral, são mais eficientes que as recursivas.

5 DISCUSSÃO

Esta seção irá discutir os resultados obtidos na seção anterior com o intuito de relacionar a teoria com a prática.

De forma geral, os resultados obtidos foram, até certo ponto, esperados, como por exemplo: (i) a busca linear é o algoritmo menos eficiente; (ii) as abordagens iterativas são mais eficientes que as recursivas; (iii) os algoritmos de dividir para conquistar são mais eficientes. Entretanto, os ruídos e picos nos gráficos tiveram de ser estudados para serem melhor compreendidos.

Vale ressaltar que, por mais que a busca linear seja o algoritmo menos eficiente, ele é o único, dentre os apresentados, que não necessita de um arranjo ordenado. Isso se torna uma vantagem nos cenários em que ordenar arranjos se torna inviável devido a entrada e saída de dados de forma incontrolável, por exemplo.

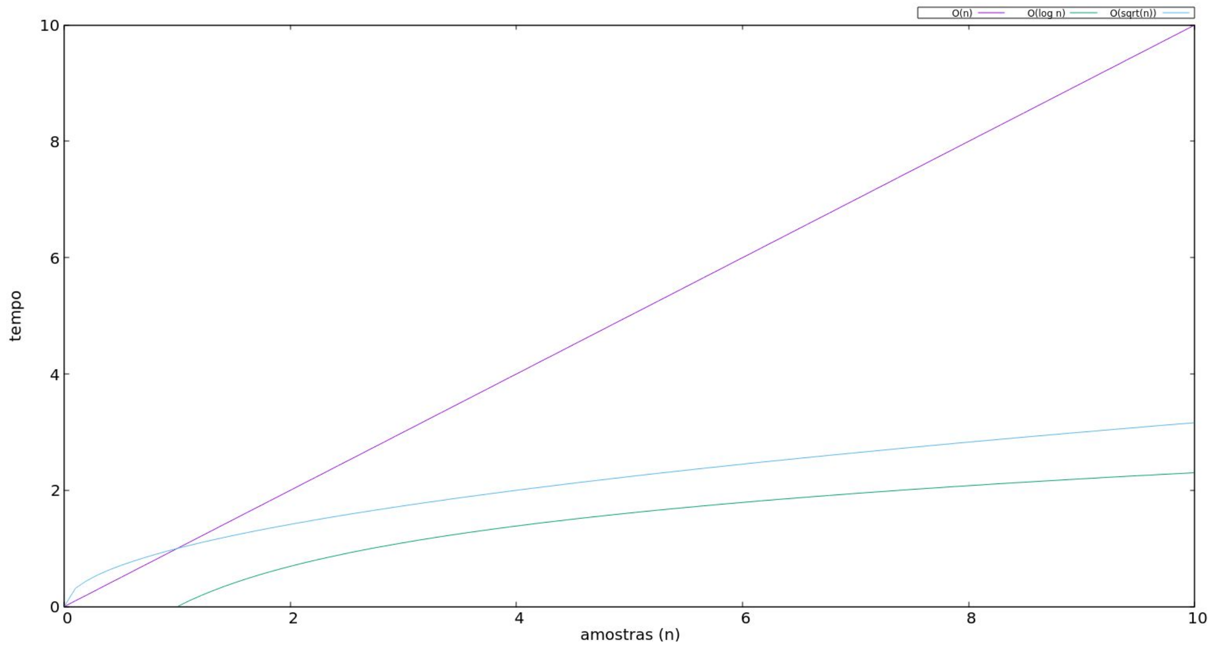
Com base nos experimentos, para arranjos ordenados, os melhores algoritmos são as abordagens iterativas da busca binária e busca ternária. Note que, na busca ternária, o arranjo foi dividido em três partes e, embora seja razoável pensar que isso é o suficiente para tornar o algoritmo mais rápido, não houve nenhum ganho de performance significativo. Isso ocorre pois, embora seja possível trabalhar sobre intervalos menores, a quantidade de comparações realizadas é maior.

Seguindo esse raciocínio, no caso extremo, se um arranjo de tamanho n fosse dividido n partes, seria necessário realizar n comparações, no pior caso, para encontrar determinado valor no arranjo, ou seja, seria igual à busca linear.

No Gráfico 7, é possível visualizar o comportamento assimptótico das funções matemáticas relacionadas aos algoritmos de busca analisados. Por exemplo, a busca linear é $O(n)$; a busca de salto é $O(\sqrt{n})$; as buscas binária, ternária e Fibonacci são $O(log\ n)$.

Assim, conhecer a classe de funções que a complexidade de um algoritmo pertence permite, entre outras coisas, criar estimativas para cenários específicos e escolher a alternativa mais eficiente.

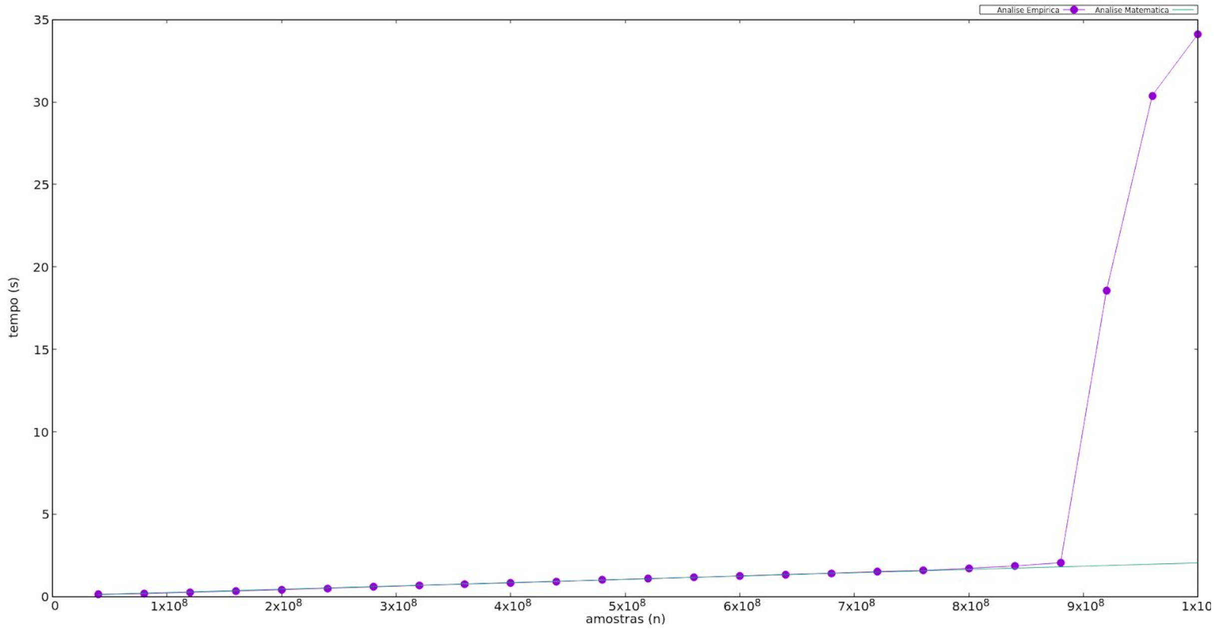
Gráfico 7 - Comparação das funções relativas aos algoritmos de busca



Fonte: Autoria própria

Entretanto, por mais que a análise matemática seja útil para estimar o custo computacional de um algoritmo, a análise empírica é necessária para ter uma visão mais precisa já existem outras variáveis que influenciam no resultado, como processador, placa-mãe, memória, sistema, dentre outros. Por exemplo, no Gráfico 8, a estimativa do tempo de execução ficou muito inferior do tempo real de execução.

Gráfico 8 - Comparação entre análise matemática e análise empírica da busca linear



Fonte: Autoria própria.

Em conclusão, a análise empírica é importante, também, para mensurar o custo computacional de forma mais precisa

6 CONSIDERAÇÕES FINAIS

No referido trabalho fora visto o que são algoritmos de busca, quais seus respectivos algoritmos e como podem ser classificados no que diz respeito à complexidade. Além disso, foram apresentados os materiais e métodos utilizados na realização dos experimentos, bem como seus respectivos resultados. Ao final, fora feita uma discussão acerca dos resultados referentes à análise empírica dos algoritmos propostos.

De maneira geral, com base nas 25 amostras (que variam de 400 milhões a 1 bilhão de elementos) usadas no experimento foi possível concluir que: (i) no geral, abordagens iterativas são mais eficientes que as recursivas; (ii) No quesito eficiência, a busca linear e ternária são superiores, seguidas da busca Fibonacci, busca de salto e busca linear; (iii) a análise matemática ajuda a prever o comportamento dos algoritmos quando submetidos a uma quantidade amostral muito alta; (iv) a análise empírica mensura o comportamento dos algoritmos com maior precisão e, eventualmente, pode ser muito discrepante da análise matemática.

REFERÊNCIAS

CATHYATSENECA. **Big-O, Little-O, Theta, Omega**. Disponível em: <https://cathyatseneca.gitbooks.io/data-structures-and-algorithms/analysis/notations.html>. Acesso em: 20 mar. 2018.

CORMEN, Thomas H. et al. **Introduction to Algorithms**. 3. ed. Massachusetts: The Mit Press, 2009.

GÁCS, Peter; LOVÁSZ, László. **Complexity of Algorithms**. 1999. Disponível em: <http://web.cs.elte.hu/~lovasz/complexity.pdf>. Acesso em: 20 mar. 2018.

GEEKSFORGEEKS. **Linear Search.** Disponível em: <https://www.geeksforgeeks.org/linear-search/>. Acesso em: 20 mar. 2018.

MIT. **Big O notacion**. Disponível em: <http://web.mit.edu/16.070/www/lecture/big_o.pdf>. Acesso em: 20 mar. 2018.

WIKIPEDIA. **Binary Search**. Disponível em: <https://en.wikipedia.org/wiki/Binary_search_algorithm>. Acesso em: 20 mar. 2018.

WIKIPEDIA. **Correctness (computer science)**. Disponível em: <https://en.wikipedia.org/wiki/Correctness_(computer_science)>. Acesso em: 20 mar. 2018.

WIKIPEDIA. **Fibonacci Search Technique**. Disponível em: <https://en.wikipedia.org/wiki/Fibonacci_search_technique>. Acesso em: 20 mar. 2018.

WIKIPEDIA. **Jump Search**. Disponível em: <https://en.wikipedia.org/wiki/Jump_search>. Acesso em: 20 mar. 2018.

WIKIPEDIA. **Linear Search**. Disponível em: <https://en.wikipedia.org/wiki/Linear_search>. Acesso em: 20 mar. 2018.

WIKIPEDIA. **Search Problem**. Disponível em: <https://en.wikipedia.org/wiki/Search_problem>. Acesso em: 20 mar. 2018.

WIKIPEDIA. **Ternary Search**. Disponível em: <https://en.wikipedia.org/wiki/Ternary_search>. Acesso em: 20 mar. 2018.

APÊNDICE A - Código Fonte em C++ do Programa Implementado

Nome do arquivo: search.h

```
#ifndef SEARCH_H
#define SEARCH_H

#include <math.h>

namespace edb{

	#define NOT_FOUND -1

	/**
	Procura um elemento dentro de um intervalo

	@param início do intervalo
	@param fim do intervalo
	@param a ser procurado
	@return índice da primeira ocorrência do valor (-1 não
encontrado)
	*/
	template <typename T>
	int linearSearch(T *, T *, T );

	/**
	Procura um elemento dentro de um intervalo

	@param início do intervalo
	@param fim do intervalo
	@param a ser procurado
	@return índice da primeira ocorrência do valor (-1 não
encontrado)
	*/
	template <typename T>
	int iteBinarySearch(T *, T *, T);

	//função usada por recBinarySearch(T *, T *, T)
```

```
    template <typename T>
    int recBinarySearch(T *,T *,T, T *);

    /**
    Procura um elemento dentro de um intervalo

    @param início do intervalo
    @param fim do intervalo
    @param a ser procurado
    @return índice da primeira ocorrência do valor (-1 não
encontrado)
    */
    template <typename T>
    int recBinarySearch(T *, T *, T);

    /**
    Procura um elemento dentro de um intervalo

    @param início do intervalo
    @param fim do intervalo
    @param a ser procurado
    @return índice da primeira ocorrência do valor (-1 não
encontrado)
    */
    template <typename T>
    int iteTernarySearch(T *, T *, T);

    //função usada por iteTernarySearch(T *, T *, T);
    template <typename T>
    int recTernarySearch(T *, T *, T, T *);

    /**
    Procura um elemento dentro de um intervalo

    @param início do intervalo
    @param fim do intervalo
    @param a ser procurado
    @return índice da primeira ocorrência do valor (-1 não
encontrado)
    */
    template <typename T>
```

```
	int recTernarySearch(T *, T *, T);

	/**
	Procura um elemento dentro de um intervalo

	@param início do intervalo
	@param fim do intervalo
	@param a ser procurado
	@return índice da primeira ocorrência do valor (-1 não
encontrado)
	*/
	template <typename T>
	int jumpSearch(T *, T *, T);

	/**
	Procura um elemento dentro de um intervalo

	@param início do intervalo
	@param fim do intervalo
	@param a ser procurado
	@return índice da primeira ocorrência do valor (-1 não
encontrado)
	*/
	template <typename T>
	int fibSearch(T *, T *, T);
}

#include "search.inl"

#endif
```

Nome do arquivo: search.inl

```
namespace edb{

	//Retorna o índice de um elemento num intervalo
	template <typename T>
	int linearSearch(T *first, T *last, T value){
		auto _first = first; // primeira posição do arranjo

		//enquanto houver valores a serem comparados
```

```
        while (first != last){
            //verifica se um valor x é igual a value
            if (*first == value)
                return first - _first; //retorna o índice
            first++; //incrementa o ponteiro
        }

        return NOT_FOUND; //caso value não seja encontrado
    }

    //Retorna o índice de um elemento num intervalo
    template <typename T>
    int iteBinarySearch(T *first, T *last, T value){

        auto _first = first;  //primeira posição do arranjo
        int i = last - first; //tamanho do vetor

        //enquanto houver intervalos de, no mínimo, 1 valor
        while (first < last - 1){
            //divide o tamanho pela metade para
            //obter o índice de comparação
            i /= 2;

            //se encontrar o elemento
            if (first[i] == value)
                return first + i - _first; //retorna o
índice

            //se value é menor que o elemento inspecionado
            else if (value < first[i] )
                //restringe o intervalo para a primeira
metade
                last = first + i + 1;

            //se value é maior que o elemento inspecionado
            else
                //restringe o intervalo para a segunda
metade
                first += i + 1;
        }
```

```
        return NOT_FOUND; //caso value não seja encontrado
    }

    //Retorna o índice de um elemento num intervalo
    template <typename T>
    int recBinarySearch(T *first, T *last, T value){
        return recBinarySearch(first,last,value, first);
    }

    template <typename T>
    int recBinarySearch(T *first, T *last, T value, T *_first){
        //se não há elemento para ser verificado
        if (first >= last)
            return NOT_FOUND; //value não foi encontrado

        int i = (last - first)/2; //metade do tamanho do array

        //se value for encontrado no indice i
        if (first[i] == value)
            return first + i - _first; //retorna o índice

        //se value é menor que o elemento inspecionado
        else if (value < first[i] )
            //restringe o intervalo para a primeira metade
            recBinarySearch(first, first + i, value, _first);

        //se value é maior que o elemento inspecionado
        else
            //restringe o intervalo para a segunda metade
            recBinarySearch(first + i + 1, last, value,
_first);
    }

    //Retorna o índice de um elemento num intervalo
    template <typename T>
    int iteTernarySearch(T *first, T *last, T value){
        auto _first = first; //primeira posição
        int i = last - first;//tamanho do array

        //enquanto houverem elementos a serem comparados
        while (first < last){
```

```
                i /= 3; //divide o array em 3

                //se value for encontrado no índice i
                if (first[i] == value){
                     return first + i - _first; //retorna o
índice

                //se value for encontrado no índice 2i
                }else if (first[2*i] == value){
                     return first + 2 * i - _first; //retorna o
índice

                //se o valor está dentro do primeiro 1 terço do
vetor
                }else if (value < first[i]){
                     //ajusta intervalo para o terço
correspondente
                     last = first + i;

                //se o valor está dentro do ultimo 1 terço do
vetor
                }else if (value > first[2*i]){
                     //ajusta intervalo para o terço
correspondente
                     first += 2*i + 1;

                //se o valor está dentro do 1 terço do meio
                } else{
                     //ajusta intervalo para o terço
correspondente
                     first += i + 1;
                     last = first + i;
                }
           }

           return NOT_FOUND; //caso value não seja encontrado
      }

      //Retorna o índice de um elemento num intervalo
      template <typename T>
      int recTernarySearch(T *first, T *last, T value){
```

```
        return recTernarySearch(first, last, value, first);
    }

    //Retorna o índice de um elemento num intervalo
    template <typename T>
    int recTernarySearch(T *first, T *last, T value, T *_first){
        //se não há elemento para ser verificado
        if (first >= last)
            return NOT_FOUND; //value não foi encontrado

        int i = (last - first)/3; //1 terço do tamanho do array

        //se value está no índice i
        if (first[i] == value)
            return first + i - _first; //retorna o índice

        //se value está no índice 2i
        else if (first[2*i] == value)
            return first + 2 * i - _first; //retorna o índice

        //se o valor está dentro do primeiro 1 terço do vetor
        else if (value < first[i])
            //ajusta intervalo para o terço correspondente
            recTernarySearch(first, first + i, value, _first);

        //se o valor está dentro do ultimo 1 terço do vetor
        else if (value > first[2*i])
            //ajusta intervalo para o terço correspondente
            recTernarySearch(first + 2*i + 1, last, value,
_first);

        //se o valor está dentro do 1 terço do meio
        else
            //ajusta intervalo para o terço correspondente

            recTernarySearch (first + i + 1 , first + 2*i ,
value, _first);
    }

    //Retorna o índice de um elemento num intervalo
```

```
template <typename T>
int jumpSearch(T *first, T *last, T value){
	int n = last - first; //tamanho do array
	int m = sqrt(n); //salto
	int f = 0; //indice anterior ao salto
	int l = m; //índice do salto

	//para evitar ajustes de limite inferiores no array
	//basta retornar NOT_FOUND caso value seja menor que o
	//menor valor do vetor
	if (value < *first)
		return NOT_FOUND; //value não foi encontrado

	//enquanto houver elementos a serem comparados
	while (first[l - 1] < value){
		l += m; //incrementa last do novo intervalo
		f += m; //incrementa first do novo intervalo

		//se novo first for maior q o tamanho do vetor
		if (f >= n)
			return NOT_FOUND; //value não foi encontrado

		//se novo last for maior que tamanho do vetor
		if (l >= n)
			l = n; //ajusta-se os limites
	}

	//faz uma busca linear no subvetor
	return (linearSearch(first + f, first + l, value) + f);
}

//Retorna o índice de um elemento num intervalo
template <typename T>
int fibSearch(T *first, T *last, T value){
	int f1 = 0;				//(n-2)ésimo termo de
fibonacci
	int f2 = 1;				//(n-1)ésimo termo de
fibonacci
	int f3 = f1 + f2;			//n-ésimo termo de fibonacci
	int i = last - first; //tamanho do vetor
```

```
        //verificador se os indices estão dentro do vetor
        int lim = -1;

        //encontrar o menor termo de fibonnaci maior
        //que o tamanho do array
        while(f3 < i){
            f1 = f2;
            f2 = f3;
            f3 = f1 + f2;
        }

        //enquanto houver elementos a serem comparados
        while(f3 > 1){
            int val = std::min(lim+f1,i-1); //menor elemento

            //se value for menor que o valor no índice,
            //cada elemento volta dois termo na sequencia de
fibonnaci
            if(first[val] > value){
                f3 = f1;
                f2 -= f1;
                f1 = f3 - f2;

            //se value for maior que o valor no índice
            //cada elemento volta um termo na sequencia de
fibonacci
            }else if(first[val] < value){
                f3 = f2;
                f2 = f1;
                f1 = f3 - f2;
                lim = val;

            //se for igual ao valor no índice
            }else{
                return val;//retorna o índice
            }
        }

        return NOT_FOUND; //caso value não seja encontrado
    }
}
```

Nome do arquivo: main.cpp

```
#include <iostream>
#include <chrono>
#include <iomanip>

#include "search.h"

#define N_AVERAGE 100
#define PRECISION 4

typedef int (*SearchFunction)(long int *, long int*, long int);

int main(){

	//array de funções
	SearchFunction functions[] = {edb::linearSearch,
								  edb::iteBinarySearch,
								  edb::recBinarySearch,

edb::iteTernarySearch,

edb::recTernarySearch,
								  edb::jumpSearch,
								  edb::fibSearch};

	int s,			//aux para calcular tamanho
		n_samples, //quantidade de amostras
		s_samples; //tamanho das amostras
	long int *v;
	long int value = s_samples*n_samples;

	//ler quantidade e tamanho das amostras
	std::cin >> n_samples >> s_samples;

	//aloca o array
	v = new long int[n_samples * s_samples];

	//inicia o vetor com valores ordenados
	for (int i = 0; i < n_samples * s_samples; i++)
		v[i] = i;
```

```
		//para cada amostra
		for (int i = 1; i <= n_samples; i++){
			s = i * s_samples;

			//imprime o tamanho da amostra
			std::cout << s << ", ";

			//simula o pior caso para cada função
			//ou seja, o valor não é encontrado
			for (auto & e : functions){
				double time = 0;

				//executa a função N_AVERAGE vezes e em seguida
				//calcula a média de tempo das execuções
				for (int i = 0; i < N_AVERAGE; i++){
					//inicia o cronometro
					auto start =
std::chrono::steady_clock::now();

					//executa as funções
					int x = e(v, v + s, value);

					//finaliza o cronometro
					auto end = std::chrono::steady_clock::now();

					//calcula tempo decorrido em nano segundos
					time += (std::chrono::duration <double,
std::milli> (end-start).count()) / N_AVERAGE;
				}

				//imprime uma linha com os tempos das funções
				std::cout << time <<
std::setprecision(PRECISION)<<", ";
			}

			//pula linha
			std::cout << "END" << std::endl;
		}

		return 0;
}
```

APÊNDICE B - Resultado das Medições de Tempo(ms) por amostra em CSV

```
40000000, 128.351, 0.00122, 0.00139, 0.00125, 0.00135, 0.10834, 0.00237
80000000, 184.0873, 0.00114, 0.00159, 0.00134, 0.00134, 0.15842, 0.00165
120000000, 255.6437, 0.00115, 0.00131, 0.00115, 0.00124, 0.18827, 0.00241
160000000, 340.9632, 0.00115, 0.00136, 0.00114, 0.00128, 0.24092, 0.00154
200000000, 421.2197, 0.00115, 0.00135, 0.00115, 0.00127, 0.26408, 0.00178
240000000, 504.38, 0.00122, 0.00136, 0.00118, 0.00129, 0.29186, 0.00157
280000000, 587.3728, 0.00118, 0.00137, 0.00126, 0.00128, 0.3011, 0.0019
320000000, 675.4817, 0.00116, 0.00136, 0.00118, 0.00126, 0.41379, 0.00157
360000000, 755.5171, 0.00117, 0.00133, 0.00116, 0.00125, 0.35202, 0.00159
400000000, 836.848, 0.00114, 0.00138, 0.00118, 0.00127, 0.44102, 0.00156
440000000, 916.8449, 0.00116, 0.00136, 0.00126, 0.00125, 0.45434, 0.00185
480000000, 1004.626, 0.00127, 0.00148, 0.00127, 0.00136, 0.44049, 0.00201
520000000, 1086.28, 0.00131, 0.00133, 0.00117, 0.00127, 0.50886, 0.00211
560000000, 1174.555, 0.00114, 0.00134, 0.00115, 0.00124, 0.52536, 0.00225
600000000, 1254.589, 0.00115, 0.00136, 0.00115, 0.00127, 0.49187, 0.00155
640000000, 1340.725, 0.00116, 0.00135, 0.00117, 0.00136, 0.51882, 0.00164
680000000, 1417.726, 0.0012, 0.00136, 0.00122, 0.00133, 0.52793, 0.00165
720000000, 1518.072, 0.00117, 0.00138, 0.00118, 0.00131, 0.60987, 0.0018
760000000, 1584.362, 0.0012, 0.00136, 0.00115, 0.00132, 0.67129, 0.00175
800000000, 1715.998, 0.00117, 0.00134, 0.00116, 0.00128, 0.57541, 0.00232
840000000, 1855.383, 0.00116, 0.00132, 0.00111, 0.00129, 0.61248, 0.00236
880000000, 2057.223, 0.00128, 0.00151, 0.00132, 0.00146, 0.75638, 0.00221
920000000, 18540.74, 0.00131, 0.00147, 0.00115, 0.00141, 4.41719, 0.00377
960000000, 30376.66, 0.00133, 0.00151, 0.0015, 0.00144, 6.54526, 0.00295
1000000000, 34103.21, 0.0012, 0.00139, 0.0012, 0.00132, 4.11399, 0.00303
```