Relatório

Karine Piacentini Coelho da Costa[1]

May 17, 2019

[1]karinepcdc@ufrn.br

Contents

Chapter 1

Introdução

Esse relatório apresenta uma análise de complexidade empírica para diferentes algoritmos de busca. Problemas de busca em um arranjo sequencial se resumem em procurar um dado valor chave k em um conjunto de valores previamente armazenados em um arranjo V passado como entrada do problema. Caso o valor seja encontrado neste arranjo, então o programa deve retornar o índice da localização de k em V, caso contrário deve retornar -1. Este estudo está interessado em problemas de busca em que os elementos do arranjo estão ordenados em *ordem crescente* e são analisados apenas o pior cenário da busca. São considerados os seguintes algoritmos de busca: busca linear; busca binária (versões iterativas e recursiva); busca ternária (versões iterativas e recursiva); *jump search*; e busca de Fibonacci.

O primeiro objetivo desse estudo é determinar qual dos dois algoritmos lineares selecionados são mais eficientes (a busca linear ou a *jump search*). O segundo objetivo é determinar qual implementação é mais eficiente, a recursiva ou iterativa. O terceiro, é determinar como o tamanho da partição influência nos algoritmos de busca não lineares. O quarto é determinar a partir de que momento algoritmos de classe de complexidade diferentes se diferenciam, comparando a busca linear com a binária. Por fim, o quinto objetivo procura determinar se existe diferentes categorias de cenários de pior caso para o algoritmo de busca de Fibonacci.

Chapter 2

Metodologia

Nesta seção descrevemos os materiais e a metodologia utilizados para obteção dos resultados apresentados na seção 3.

2.1 Características técnicas

Os algorítmos de buscas foram implementados na linguagem C++ e o compilador utilizado foi o g++ (versão 10.0.0). O computador onde as simulações foram realizadas possui as seguintes características:

- MacBook Pro (2014)
- Processador: 2.5 Ghz Intel Core i7
- memória: 16 GB 1600 MHz DDR3
- Sistema operaciona: Mojave 10.14.3

2.2 Algoritmos

Dado um arranjo sequencial V, cujos elementos estão ordenados em *ordem crescente* (sem repetição) e um valor chave k, os algoritmos aqui descritos têm como objetivo retornar o índice da localização de k em V, caso este valor não esteja presente no arranjo, devem retornar -1.

2.2.1 Busca linear

A busca linear varre o arranjo do primeiro ao último elemento comparando, a cada passo da varredura, o valor selecionado no arranjo com a chave procurada. Se encontrar a chave, interrompe a busca e retorna o índice atual, se não, a varredura continua até o

fim do arranjo. Seu pior caso, por tanto, é quando o valor procurado é maior ou igual ao último elemento do arranjo.

Algoritmo 1: Busca linear

Entrada: Vetor V, chave k e limites de busca esquerdo l e direito r (inclusive).
Saída: Índice da ocorrência de k em V; ou -1 caso não exista k em V.
`/* Precondição:` $l \leq r$; $l, r \geq 0$; V `em ordem crescente. */`

```
1 Função buscaLin(V: arranjo de inteiros; l: inteiro; r: inteiro; k:
   inteiro): inteiro
2 |  var i: inteiro
3 |  para i ← l até r faça
4 |  |  se V[i] == k então
5 |  |  |  retorna i
6 |  |  fim
7 |  fim
8 |  retorna −1
9 fim
```

2.2.2 Busca binária

Na busca binária, particiona-se o arranjo em dois, selecionando o elemento do meio. Caso este seja igual ao valor procurado, interrompe-se a busca retornado o valor do índice encontrado, caso contrário, uma comparação é feita a fim de determinar se o valor é maior ou menor do que o elemento selecionado, determinando assim em qual metade deve-se fazer a busca novamente. A nova busca repete o mesmo procedimento até que o elemento seja encontrado ou, caso a partição analizada seja igual a zero e o valor não tiver sido encontrado, o algoritmo retorna -1. O pior caso é aquele em que k não pertence ao arranjo ou é o último elemento buscado.

Algoritmo 2: Busca binária iterativa

Entrada: Vetor V, chave k e limites de busca esquerdo l e direito r (inclusive).
Saída: Índice da ocorrência de k em V; ou -1 caso não exista k em V.

```
/* Precondição: l ≤ r; l, r ≥ 0; V em ordem crescente.                    */
1  Função buscaBin_it(V: arranjo de inteiros; l: inteiro; r: inteiro; k:
     inteiro): inteiro
2      var m: inteiro /* último valor da primeira metade do arranjo       */
3
4      enquanto r ≥ l faça
5          m ← (l + r)/2
6          se k == V[m] então
7              retorna m
8          senão se k < V[m] então
9              r ← m − 1
10         senão
11             l ← m + 1
12         fim
13     fim
14     retorna −1
15 fim
```

Algoritmo 3: Busca binária recursiva

Entrada: Vetor V, chave k e limites de busca esquerdo l e direito r (inclusive).
Saída: Índice da ocorrência de k em V; ou -1 caso não exista k em V.
`/* Precondição:` $l \leq r$; $l, r \geq 0$; V `em ordem crescente. */`

```
1  Função buscaBin_rec(V: arranjo de inteiros; l: inteiro; r: inteiro; k:
     inteiro): inteiro
2      var m: inteiro /* último valor da primeira metade do arranjo */
3
4      se r < l então
5          retorna −1
6      senão
7          m ← (l + r)/2
8          se k == V[m] então
9              retorna m
10         senão se k < V[m] então
11             retorna buscaBin_rec(V,l,m − 1,k)
12         senão
13             retorna buscaBin_rec(V,m + 1,r,k)
14         fim
15     fim
16 fim
```

2.2.3 Busca ternária

A busca ternária se assemelha a busca binária, com a diferença que divide o arranjo em 3 partes, selecionando o maior elemento do primeiro terço do arranjo t_1 e o maior do segundo terço t_2. Em seguida, verifica se t_1 ou t_2 é o valor procurado k. Caso seja, retorna o índice do elemento no arranjo, caso não, realiza o mesmo procedimento no terço que possivelmente contém o valor procurado, ou seja, se $k < t_1$, faz a busca ternária no primeiro terço, se $t_1 < k < t_2$ faz a busca no segundo terço, senão faz a busca no último terço. Novamente, o pior caso é aquele em que k não pertence ao arranjo ou é o último elemento buscado.

Algoritmo 4: Busca ternária iterativa

Entrada: Vetor V, chave k e limites de busca esquerdo l e direito r (inclusive).
Saída: Índice da ocorrência de k em V; ou -1 caso não exista k em V.
`/* Precondição:` $l \leq r$; $l, r \geq 0$; V `em ordem crescente. */`

```
1  Função buscaTer_it(V: arranjo de inteiros; l: inteiro; r: inteiro; k:
     inteiro): inteiro
2      var t1: inteiro /* último valor do primeiro terço do arranjo    */
3      var t2: inteiro /* último valor do segundo terço do arranjo     */
4
5      enquanto r ≥ l faça
6          t1 ← l + (r − l)/3
7          t2 ← r − (r − l)/3
8
9          se k == V[t1] então
10             retorna t1
11         senão se k == V[t2] então
12             retorna t2
13         senão se k < V[t1] então
14             r ← t1 − 1
15         senão se k < V[t2] então
16             l ← t1 + 1
17             r ← t2 − 1
18         senão
19             l ← t2 + 1
20         fim
21     fim
22     retorna −1
23 fim
```

Algoritmo 5: Busca ternária recursiva

Entrada: Vetor V, chave k e limites de busca esquerdo l e direito r (inclusive).
Saída: Índice da ocorrência de k em V; ou -1 caso não exista k em V.
`/* Precondição:` $l \leq r$; $l, r \geq 0$; V `em ordem crescente. */`

```
1  Função buscaTer_rec(V: arranjo de inteiros; l: inteiro; r: inteiro; k:
     inteiro): inteiro
2      var t1: inteiro /* último valor do primeiro terço do arranjo     */
3      var t2: inteiro /* último valor do segundo terço do arranjo      */
4
5      se r < l então
6          retorna −1
7      senão
8          t1 ← l + (r − l)/3
9          t2 ← r − (r − l)/3
10
11         se k == V[t1] então
12             retorna t1
13         senão se k == V[t2] então
14             retorna t2
15         senão se k < V[t1] então
16             retorna buscaTer_rec(V,l,t1 − 1,k)
17         senão se k < V[t2] então
18             retorna buscaTer_rec(V,t1 + 1,t2 − 1,k)
19         senão
20             retorna buscaTer_rec(V,t2 + 1,r,k)
21         fim
22     fim
23 fim
```

2.2.4 *Jump search*

Na *Jump search*, uma varredura em saltos de tamanho m ($0 < m < n$, com n o tamanho do vetor) é realizada. Na primeira iteração, o m-ésimo elemento é comparado com o valor buscado k, caso seja igual, retorna-se m, se for maior, uma busca linear é realizada neste bloco do arranjo, caso contrário, a varredura passa para o $(m+1)$-ésimo elemento onde se procede da mesma maneira. Caso a busca chegue ao último elemento do vetor sem encontrar o valor chave, o valor de retorno é -1. Aqui o pior caso é quando o valor procurado é maior ou igual ao último elemento do arranjo.

Algoritmo 6: *Jump search*

Entrada: Vetor V, chave k e limites de busca esquerdo l e direito r (inclusive).
Saída: Índice da ocorrência de k em V; ou -1 caso não exista k em V.

```
/* Precondição:  l ≤ r; l, r ≥ 0; V em ordem crescente.          */
1  Função buscaJump(V: arranjo de inteiros; l: inteiro; r: inteiro; k:
     inteiro): inteiro
2      var m: inteiro
3      var p: inteiro /* tamanho do salto                          */
4
5      p ← √(r − l + 1)
6      m ← l + p
7      enquanto m ≤ r faça
8          se k == V[m] então
9              retorna m
10         senão se k < V[m] então
11             retorna buscaLin(V, m − p, m − 1, k)
12         fim
13         m ← m + p
14     fim
15     se m > r e V[r] > k então
16         retorna buscaLin(V, m − p, r, k)
17     fim
18     retorna −1
19 fim
```

2.2.5 Busca de Fibonacci

A busca de Fibonacci procede da mesma forma que a busca binária, mas particiona o arranjo em duas partes de tamanhos diferentes. O tamanho da primeira partição é o i-ésimo número da série de Fibonacci $F(i)$, tal que $Fib(i+2)$ é o primeiro número na série maior ou igual ao tamanho do arranjo n. O pior caso é aquele em que k não pertence ao arranjo ou é o último elemento buscado.

Algoritmo 7: Busca de Fibonacci

Entrada: Vetor V, chave k e limites de busca esquerdo l e direito r (inclusive).
Saída: Índice da ocorrência de k em V; ou -1 caso não exista k em V.

```
/* Precondição:  l ≤ r; l,r ≥ 0; V em ordem crescente.                    */
```

```
1  Função buscaBin_it(V: arranjo de inteiros; l: inteiro; r: inteiro; k:
     inteiro): inteiro
2      var size: inteiro
3      var i: inteiro
4      var i_fib1: inteiro
5      var fib1: inteiro
6      var Fib: arranjo de inteiros
7      size ← r − l + 1
       /* Calcula a serie de Fibonacci até o i-ésimo termo,
       onde F(i) <= size                                                   */
8      Fib[0] ← 0
9      Fib[1] ← 1
10     i ← 1
11     enquanto Fib[i] < size faça
12         i ← i + 1
13         Fib[i] ← Fib[i − 1] + Fib[i − 2]
14     fim
15
16     i_fib1 ← i − 2
17     enquanto l < r faça
18         fib1 ← l + F[i_fib1]
19         se k == V[fib1] então
20             retorna fib1
21         senão se k < V[fib1] então
               /* Novos tamanhos das partições à esquerda                 */
22             r ← fib1 − 1
23             i_fib1 = i_fib1 − 2
24             se i_fib1 < 0 então
25                 i_fib1 ← 0
26             fim
27         senão
               /* Novos tamanhos das partições à direita                  */
28             l ← fib1 + 1
29             i_fib1 ←  i_fib1 − 1
30             se i_fib1 < 0 então
31                 i_fib1 ← 0
32             fim
33         fim
34     fim
35     retorna −1
36 fim
```

2.3 Cenários das simulações

As simulações foram feitas buscando um valor em um conjunto *ordenado crescente* considerando o pior caso apenas. Portanto, para todos os algoritmos selecionados, o cenário de pior caso escolhido foi a busca de um valor que não pertence ao conjunto de busca e é maior que o maior elemento neste.

2.4 metodologia

Para comparar os diferentes algoritmos de busca, simulações do tempo de execução para diferentes tamanhos de arranjos de entrada foram feitas. Um vetor de inteiros longos de tamanho 10^8 preenchido com números pares em ordem crescente foi utilizado para gerar as amostras. Para os resultados obtidos na seção 3, de 200 a 300 amostras do vetor foram utilizadas com tamanhos variando de 10 até 10^8, com crescimento linear. Para os gráficos ampliados na seção 4, 900 amostras foram utilizadas.

2.4.1 Simulações de tempo de execução

Utilizou-se a biblioteca Chronos com precisão de nanosegundos para medir o tempo antes e depois da execução de cada algoritmo de busca. Para suavizar as flutuações temporais, o tempo levado em cada amostra foi medido 100 vezes e apenas a média progressiva foi registrada. A fórmula da média temporal progressiva foi utilizada para evitar erros de arredondamento e é dada pela seguinte fórmula recusiva:

$$\begin{aligned} M_0 &= 0, \\ M_k &= M_{k-1} + \frac{x_k - M_{k-1}}{k}, \end{aligned} \tag{2.1}$$

onde x_k é o tempo mensurado para a k-ésima execução e $M_{k=m}$ corresponde a média aritmética.

Chapter 3

Resultados

Os resultados obtidos com as simulações estão descrito abaixo.

3.1 Busca linear

Na figura abaixo está o resultado da simulação do algoritmo de busca linear em arranjos sequenciais de tamanhos diferentes, onde o tempo de execução médio foi medido. Fazendo um ajuste dos pontos da curva conseguimos determinar que o comportamento assintótico é linear.

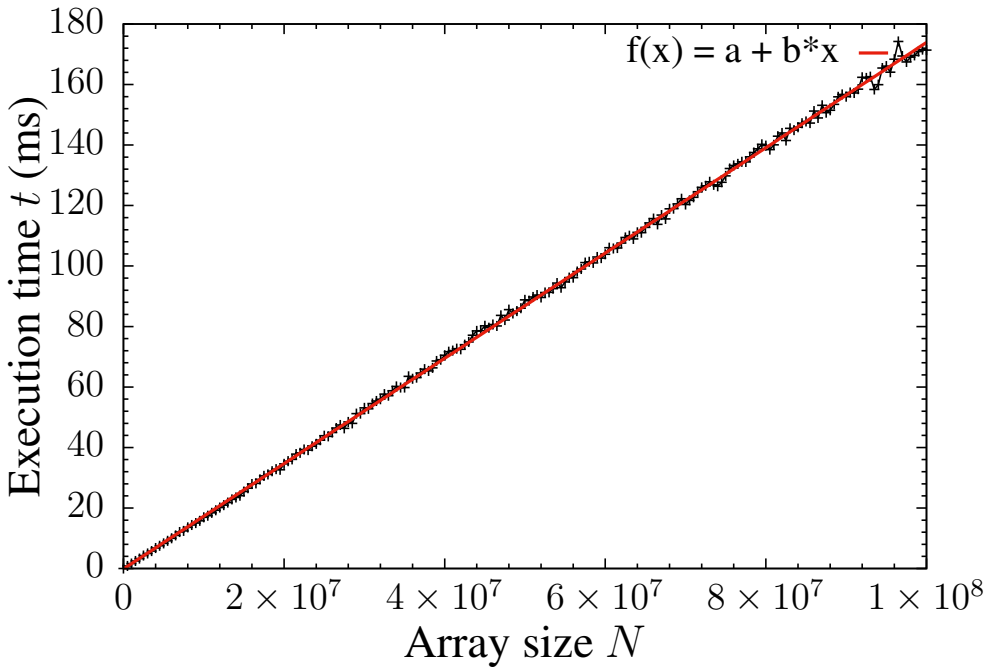

Figure 3.1: Comportamento assintótico da busca linear em relação ao tempo de execução médio (em milisegundos) a medida que o tamanho do arranjo sequencial N aumenta.

3.2 Busca binária

Os resultados da simulação do algoritmo de busca binária em suas versões iterativa e recursiva em arranjos sequenciais de tamanhos diferentes seguem abaixo. Fazendo um ajuste dos pontos da curva de ambas as implementações conseguimos determinar que o comportamento é logarítmico.

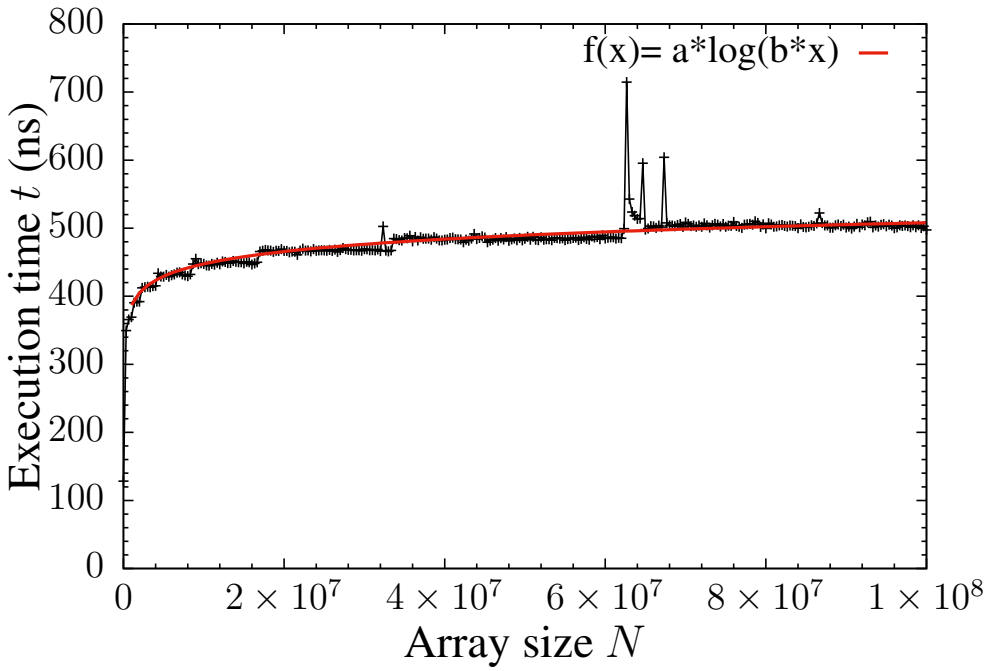

Figure 3.2: Comportamento assintótico da implementação iterativa da busca binária em relação ao tempo de execução médio (medido em nanosegundos).

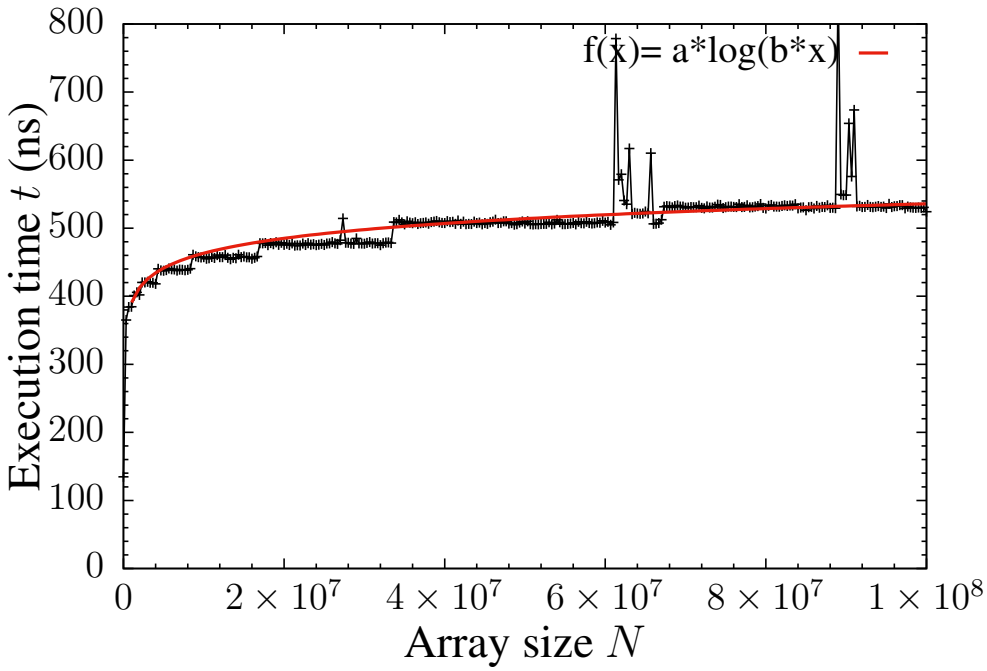

Figure 3.3: Comportamento assintótico da implementação recursiva da busca binária em relação ao tempo de execução médio (medido em nanosegundos).

3.3 Busca ternária

Nesta subseção, apresentamos os gráficos de tempo de execução em função do tamanho do arranjo para a busca ternária em suas versões iterativa e recursiva. Fazendo um ajuste dos pontos da curva de ambas as implementações conseguimos determinar que o comportamento é logarítmico.

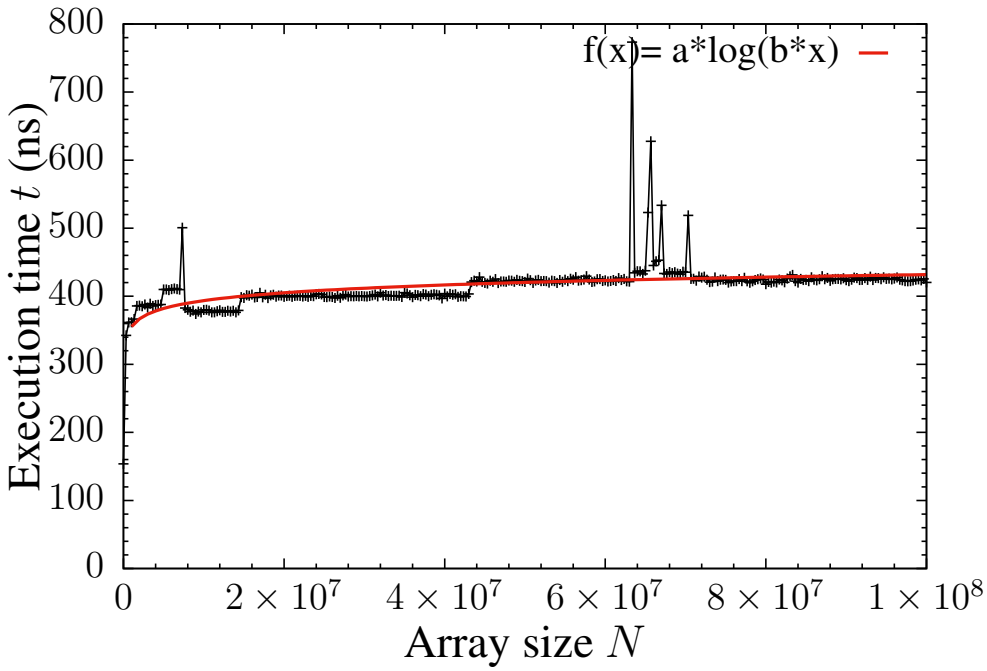

Figure 3.4: Comportamento assintótico da implementação iterativa da busca ternária em relação ao tempo de execução médio (medido em nanosegundos).

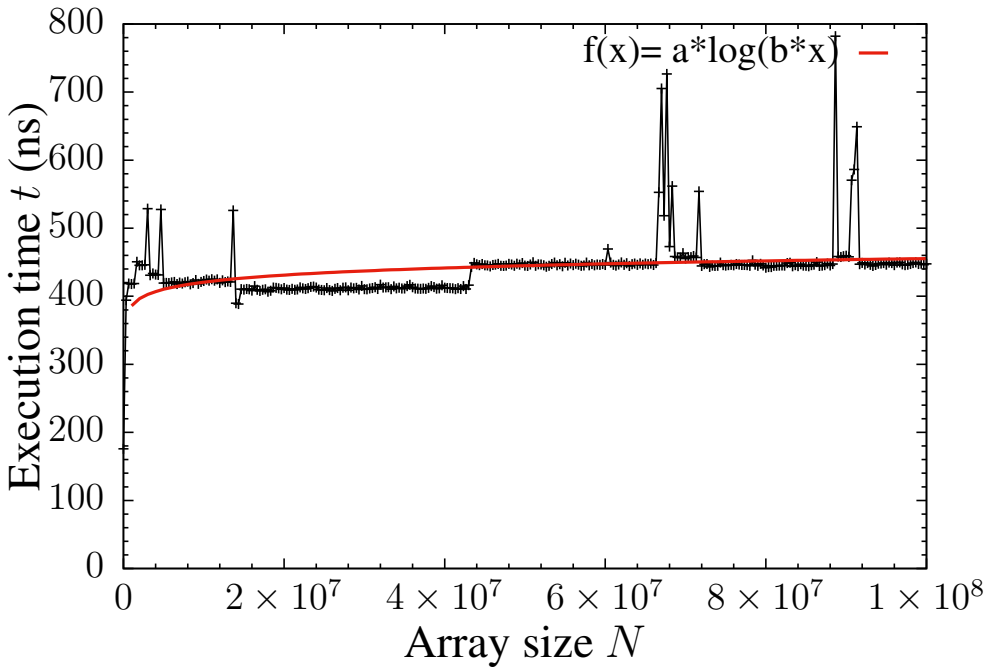

Figure 3.5: Comportamento assintótico da implementação recursiva da busca ternária em relação ao tempo de execução médio (medido em nanosegundos).

3.4 *Jump search*

Na figura abaixo temos o resultado da simulação do algoritmo *jump search* em arranjos sequenciais de tamanhos diferentes, onde o tempo de execução médio foi medido. Fazendo um ajuste dos pontos da curva conseguimos determinar que o comportamento assintótico é linear.

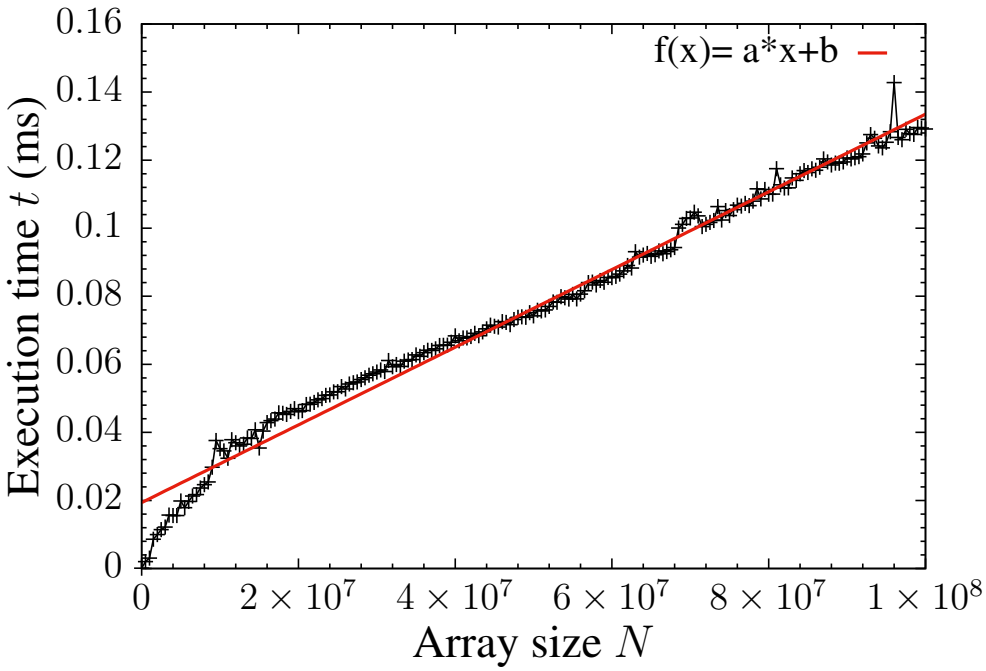

Figure 3.6: Comportamento assintótico da *jump search* em relação ao tempo de execução médio (medido em milisegundos) a medida que o tamanho do arranjo sequencial N aumenta.

3.5 Busca de Fibonacci

Na figura 3.7 é possível verificar o resultado da simulação do algoritmo de busca de Fibonacci em arranjos sequenciais de tamanhos diferentes, onde o tempo de execução médio foi medido. Fazendo um ajuste dos pontos da curva conseguimos determinar que o comportamento assintótico é logarítmo.

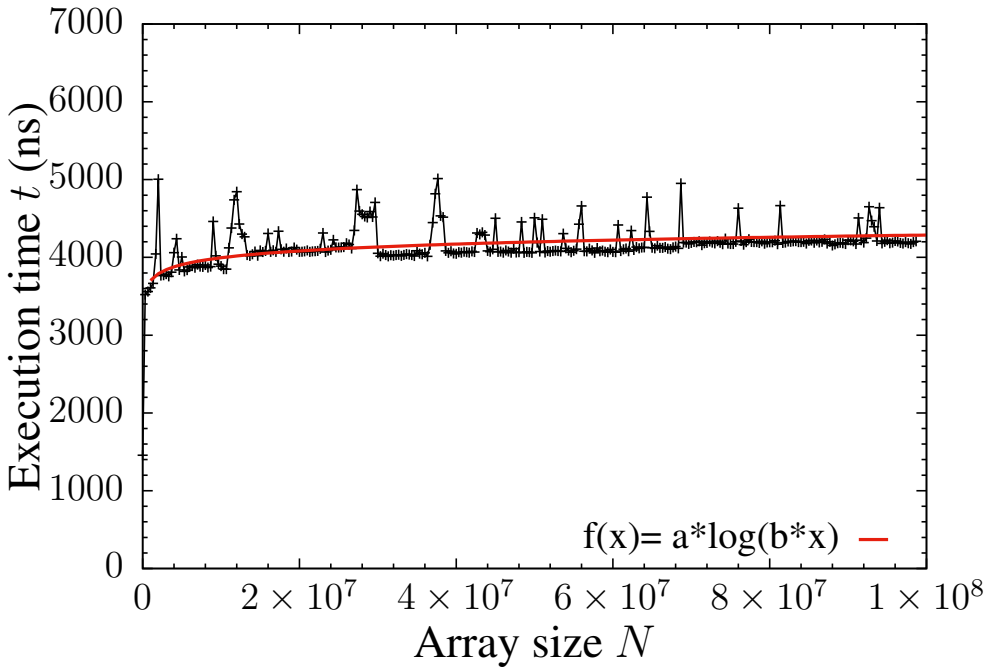

Figure 3.7: Comportamento assintótico da busca de Fibonacci em relação ao tempo de execução médio (medido em milisegundos) a medida que o tamanho do arranjo sequencial N aumenta.

Chapter 4

Discussão

Nesta seção, são analidos os resultados obtidos a fim de responder as perguntas que motivaram este estudo.

4.1 Algoritmos linear mais eficiente

A primeira pergunta a ser respondida é qual dos dois algoritmos lineares simulados é mais eficiente. Na figura 4.1, estão os resultados da busca linear e da *jump search*. Claramente, a *jump search* é mais eficiente, chegando a ser mil vezes mais rápida para o maior tamanho de arranjo simulado. Na ampliação mostrada na figura 4.2 podemos observar melhor a diferença de eficiência entre os dois algoritmos.

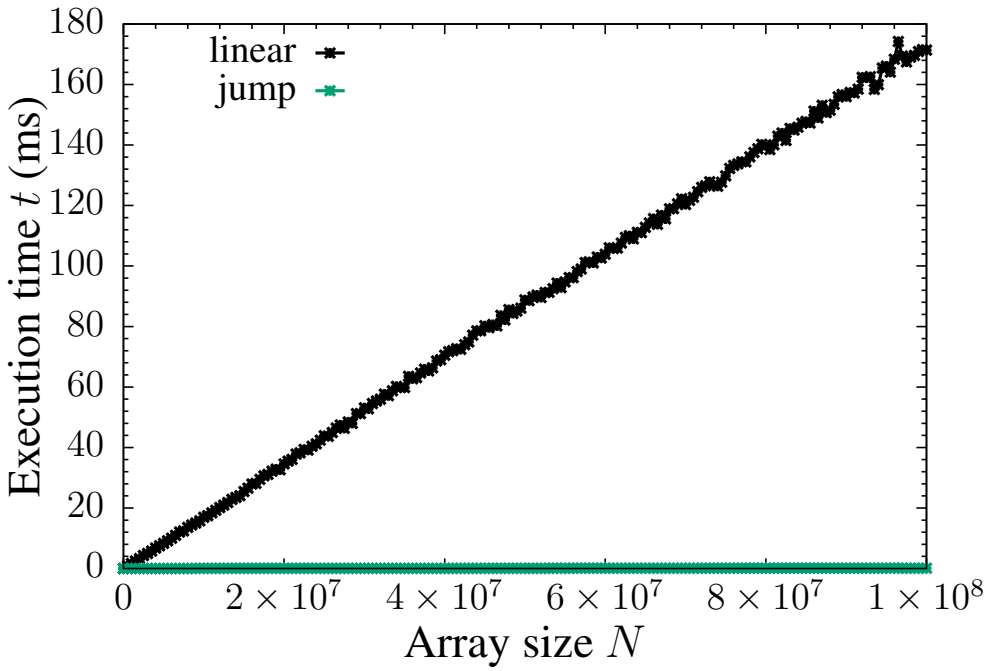

Figure 4.1: Comparação do tempo de execução médio das simuluções dos algoritmos lineares: busca linear (preto) e *jump search* (verde).

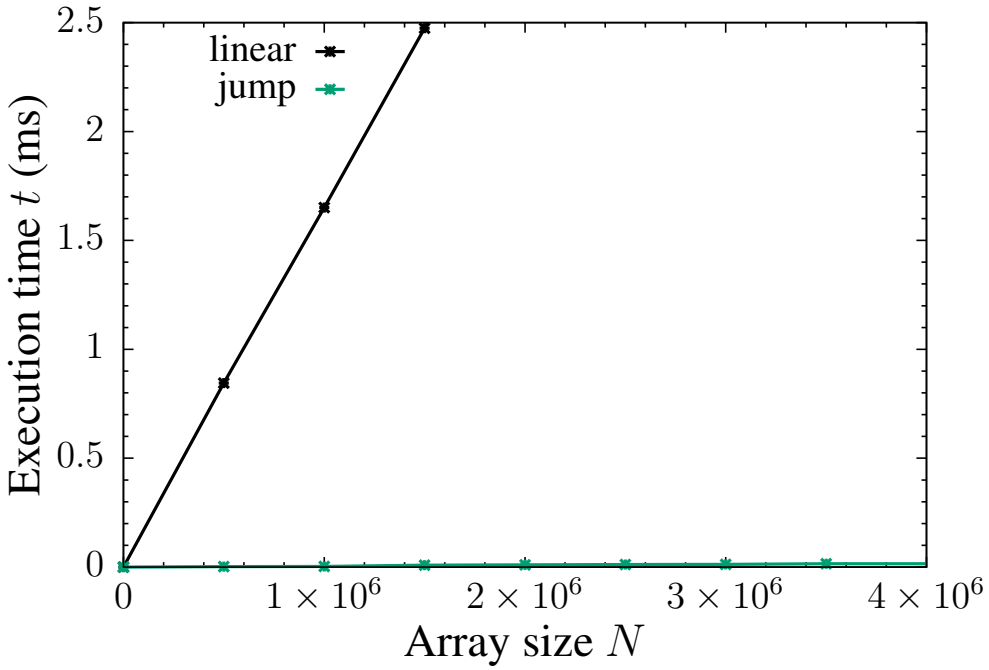

Figure 4.2: Ampliação da figura 4.1.

4.2 Implementação mais eficiente: recursiva ou iterativa?

O segundo objetivo é determinar qual implementação é mais eficiente, se a recursiva ou a iterativa. Analisando as curvas mostradas nas figuras 4.3 e 4.4 verifica-se que a versão iterativa e recursiva se diferenciam muito pouco, tanto na versão binária ou ternária do algoritmo. Para alguns tamanhos a curva da implementação iterativa parece ligeiramente mais eficiente, no entanto a medida de tempo de execução está sujeita a flutuações e não há uma medida do erro da obtenção dos pontos por isso fica difícil concluir qual implementação é mais eficiente com esses dados.

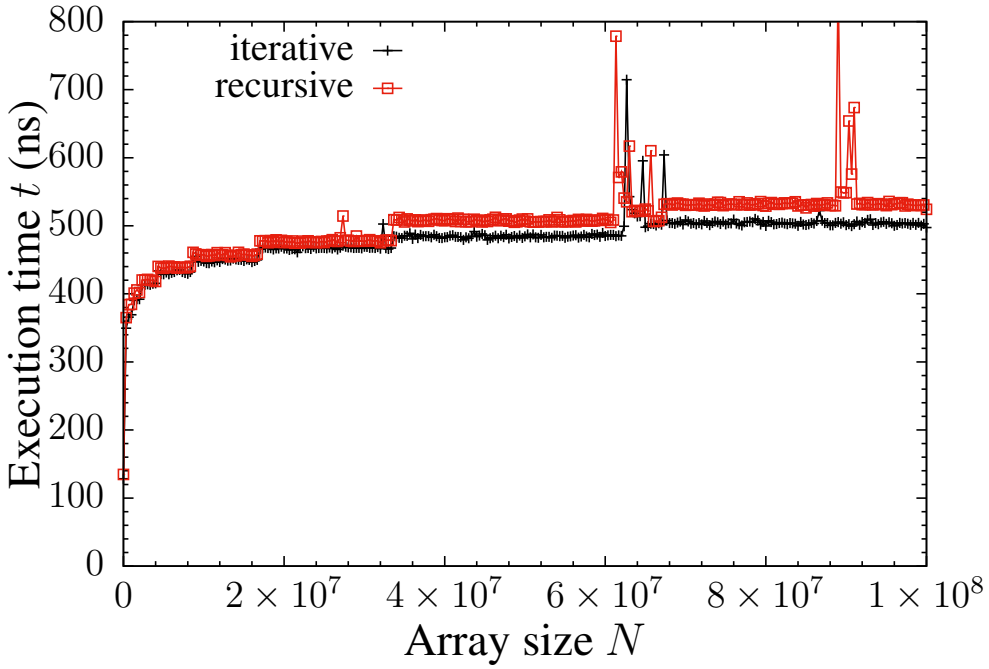

Figure 4.3: Comparação do tempo de execução médio das simuluções da implementação iterativa (preto) e recursiva (vermelho) do algoritmo de busca binária.

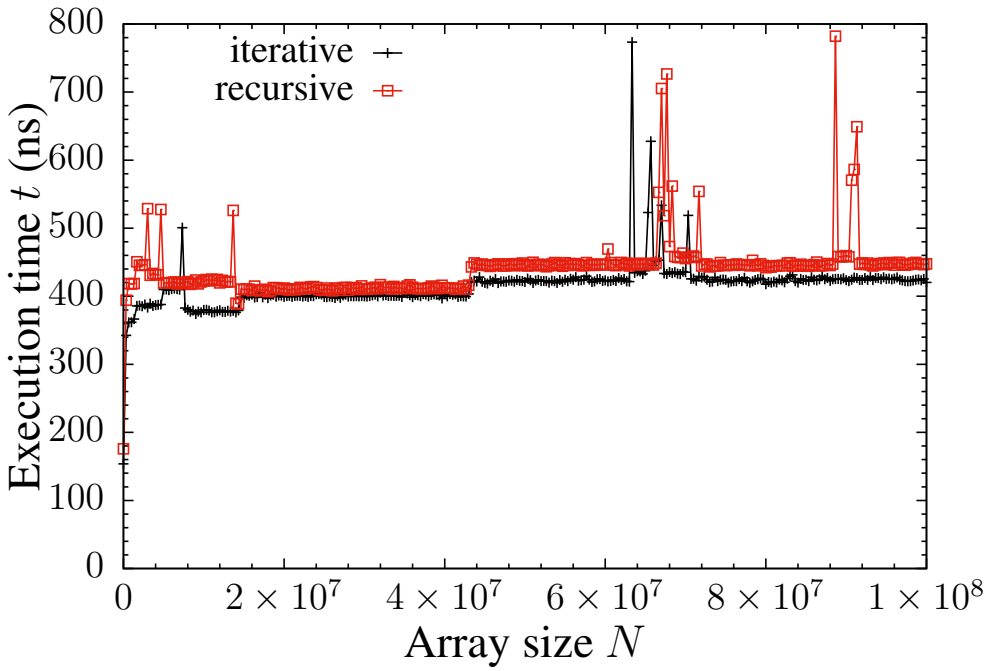

Figure 4.4: Comparação do tempo de execução médio das simuluções da implementação iterativa (preto) e recursiva (vermelho) do algoritmo de busca ternária.

4.3 Influência do tamanho da partição nos algoritmos de busca não lineares

Outra comparação interessante pode ser feita com os algoritmo de busca binária, ternária e Fibonacci (figura 4.5). Assim consegue-se verificar a influência do tamanho da partição nas buscas não lineares. Dos dados obtidos, pode-se concluir que a busca ternária é ligeiramente mais rápida que a binária e a de Fibonacci é muito mais lenta que ambas.

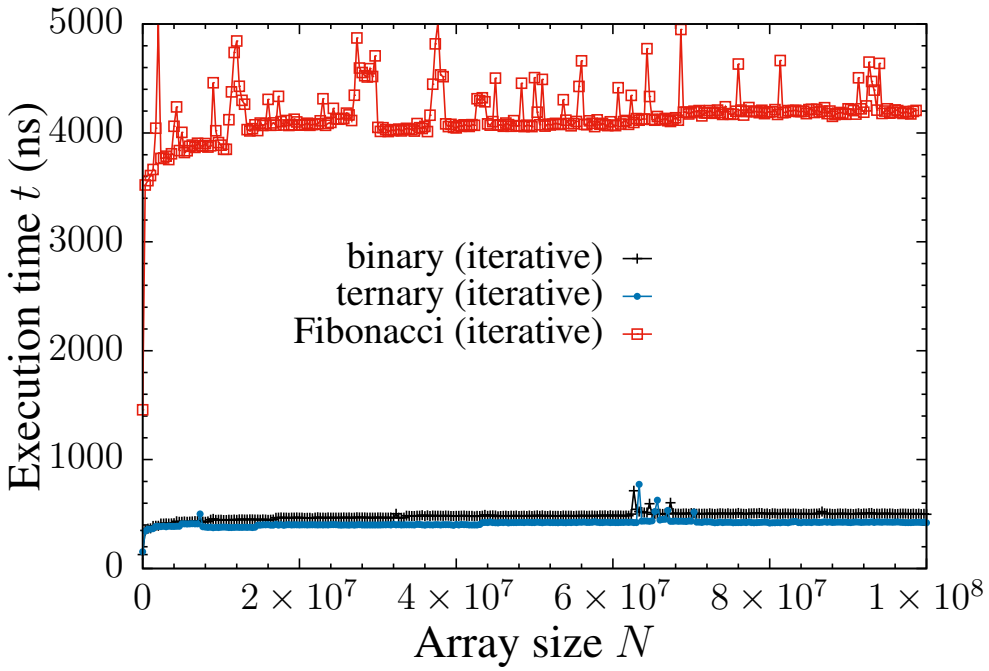

Figure 4.5: Comparação entre algoritmos (versões iterativas) com diferentes tamanhos de partições: binário (preto), ternário (azul) e Fibonacci (vermelho).

4.4 Diferenciação de algoritmos de classe de complexidade diferentes

Como pode ser verificado na seção de resultados, para tamanhos grandes do arranjo, enquanto a busca linerar demora centenas de milisegundos, a busca binária demora centenas de nanosegundos (10^6 vezes menos), sendo consideravelmente mais eficiente. Uma pergunta relevante nesse caso, é a partir de que momento algoritmos de classe de complexidade diferentes se diferenciam. Comparando a busca linear com a binária (figura 4.6), concluímos que para tamanhos superiores que 100 a busca binária já é superior.

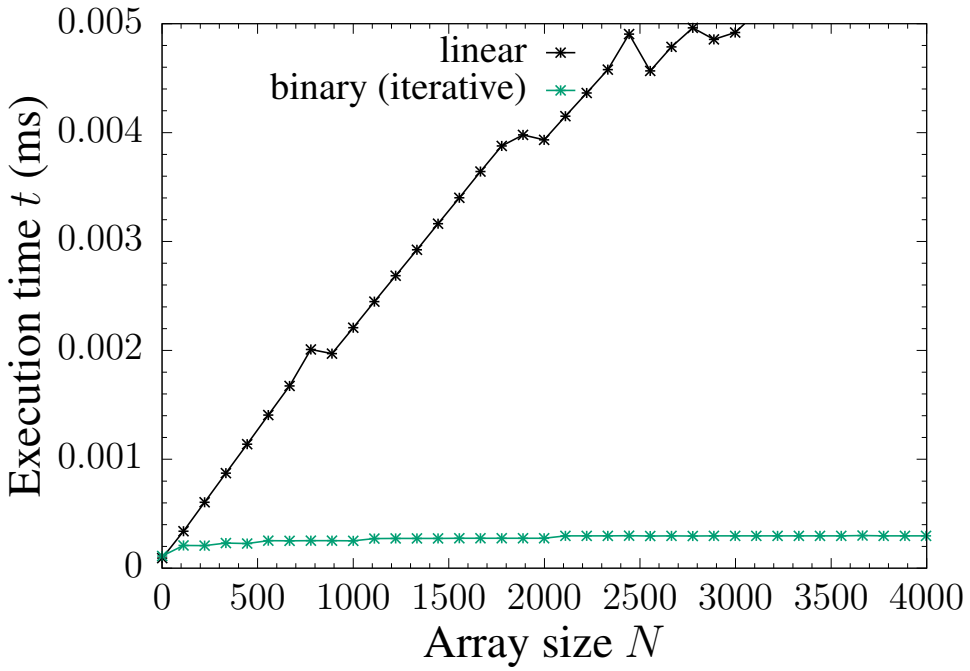

Figure 4.6: Comparação do tempo de execução médio da busca linear (preto) com a binária (verde).

4.5 Cenários do algoritmo de Fibonacci

Por fim, o quinto objetivo procura determinar se existe diferentes categorias de cenários de pior caso para o algoritmo de busca de Fibonacci. Para isso, dois cenários de pior caso foram simulados:

- Cenário 1: o elemento procurado não está no vetor mas possui um valor próximo dos elementos no começo do arranjo (gráfico em preto na figura 4.7);
- Cenário 2: o elemento procurado não está no vetor mas possui um valor próximo dos elementos no final do arranjo (gráfico em vermelho na figura 4.7);.

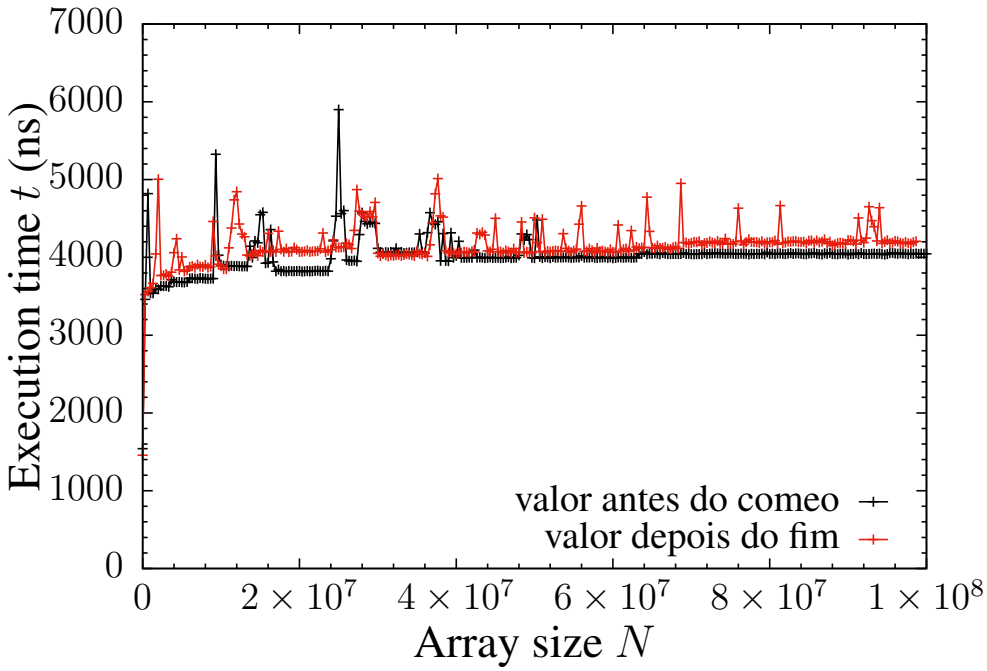

Figure 4.7: Comparação entre dois cenários de pior caso do algoritmo de busca de Fibonacci. Em um cenário, o elemento procurado está fora do arranjo, mas próximo dos elementos de seu começo (em preto). No outro, o elemento procurado está fora do arranjo, mas próximo dos elementos do fim deste (em vermelho).

A figura 4.7 mostra que há uma ligeira diferença entre os dois, sendo o cenário 1 consistentemente mais rápido que o cenário 2. Deste modo, a conclusão é que há sim dois casos de pior cenário para o algoritmo de busca de Fibonacci, mas a diferença de performace é despresível.

Chapter 5

Conclusão

Uma análise de complexidade empírica para os algoritmos de busca linear, busca binária (versões iterativas e recursiva), busca ternária (versões iterativas e recursiva), *jump search* e busca de Fibonacci foi feita. As simulações desses algoritmos consideravam apenas o pior cenário da busca em um arranjo com seus elementos ordenados em *ordem crescente*.

Com este estudo, determinou-se que entre os algoritmos de busca linear o *jump search* é o mais eficiente. Também foi possível verificar que a implementação iterativa é ligeiramente mais eficiente que a recursiva para tamanhos maiores de sistema. Comparando os algoritmos de busca binária, ternária e de Fibonacci, foi possível ver que o tamanho da partição influencia na busca e que as buscas ternária e binária são mais consideravelmente rápidas do que a busca de Fibonacci, sendo a busca ternária a mais eficiênte. Ademais, ao comparar a busca binária e linear, algoritmos de classe de complexidade diferentes, constatou-se que a diferenciação entre esse algoritmo ocorre já com tamanhos de arranjo pequenos (da ordem de 100). Além disso, verificou-se que existe duas categorias de cenários de pior caso para o algoritmo de Fibonacci, quando o valor procurado está fora do arranjo, mas mais próximo dos elementos do começo, o algoritmo tem uma performace ligeiramente melhor do que quando esse se encontra fora do arranjo, mas mais próximo dos elementos do fim.

Por fim, após essa análise foi possível concluir que os algoritmos mais eficiêntes são os de busca ternária e binária iterativos. Mas, como estes dependem do arranjo estar ordenado, em situações mais gerais, o único algoritmo disponível, dentre os estudados, é o de busca linear.