

Relatorio

Karine Piacentini Coelho da Costa¹

March 21, 2019

¹karinepcdc@ufrn.br

Contents

1	Introdução	2
2	Metodologia	3
2.1	Características técnicas	3
2.2	Algoritmos	3
2.2.1	Busca linear	3
2.2.2	Busca binária	4
2.2.3	Busca ternária	6
2.2.4	<i>Jump search</i>	8
2.2.5	Busca de Fibonacci	9
2.3	Cenários das simulações	11
2.4	metodologia	11
2.4.1	Simulações de tempo de execução	11
2.4.2	Simulações do número de passos da operação dominante	11
3	Resultados	12
3.1	Busca linear	12
3.2	Busca binária	13
3.3	Busca ternária	14
3.4	<i>Jump search</i>	16
3.5	Busca de Fibonacci	17

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 obtenção dos resultados apresentados no capítulo 3.

2.1 Características técnicas

Os algoritmos de buscas foram implementados na linguagem C++ e o compilador utilizado foi o g++ (tipo e versão????). 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
- Placa mãe: ????
- Sistema operacional: (tipo e versão????)

2.2 Algoritmos

Dado um arranjo sequencial V , cujos elementos estão ordenado em *ordem crescente* (sem repetição) e um valor chave k , os algoritmos aqui descritos tem como objetivo retornar o índice da localização de k em V , caso este valor não esteja presente no arranjo, devem retornar -1 . Os códigos utilizados no estudo baseados nesses algoritmos encontram-se no apêndice 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, senão, 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  $i \leftarrow l$  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 retornando 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 descrito anteriormente até que o elemento seja encontrado ou, caso a partição analisada 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: $l \leq r$; $l, r \geq 0$; 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  $r \geq l$  faça  
5      $m \leftarrow (l + r)/2$   
6     se  $k == V[m]$  então  
7       retorna  $m$   
8     senão se  $k < V[m]$  então  
9        $r \leftarrow m - 1$   
10    senão  
11       $l \leftarrow m + 1$   
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     |  $m \leftarrow (l + r)/2$ 
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 se algum desses elementos é o valor procurado k . Caso seja, retorna o índice do elemento no arranjo, caso não, realiza-se o mesmo procedimento no terço que possivelmente contém o valor procurado, ou seja, se $k < t_1$, faz-se a busca ternária no primeiro terço, se $t_1 < k < t_2$ busca-se no segundo terço, senão a busca é feita 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  $t_1$ : inteiro /* último valor do primeiro terço do arranjo */
3  |   var  $t_2$ : inteiro /* último valor do segundo terço do arranjo */
4  |
5  |   enquanto  $r \geq l$  faça
6  |       |    $t_1 \leftarrow l + (r - l)/3$ 
7  |       |    $t_2 \leftarrow r - (r - l)/3$ 
8  |       |
9  |       |   se  $k == V[t_1]$  então
10 |          |   retorna  $t_1$ 
11 |       |   senão se  $k == V[t_2]$  então
12 |          |   retorna  $t_2$ 
13 |       |   senão se  $k < V[t_1]$  então
14 |          |    $r \leftarrow t_1 - 1$ 
15 |       |   senão se  $k < V[t_2]$  então
16 |          |    $l \leftarrow t_1 + 1$ 
17 |          |    $r \leftarrow t_2 - 1$ 
18 |       |   senão
19 |          |    $l \leftarrow t_2 + 1$ 
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  $t_1$ : inteiro /* último valor do primeiro terço do arranjo */
3  |   var  $t_2$ : inteiro /* último valor do segundo terço do arranjo */
4  |
5  |   se  $r < l$  então
6  |   |   retorna  $-1$ 
7  |   senão
8  |   |    $t_1 \leftarrow l + (r - l)/3$ 
9  |   |    $t_2 \leftarrow r - (r - l)/3$ 
10 |
11 |   se  $k == V[t_1]$  então
12 |   |   retorna  $t_1$ 
13 |   senão se  $k == V[t_2]$  então
14 |   |   retorna  $t_2$ 
15 |   senão se  $k < V[t_1]$  então
16 |   |   retorna buscaTer_rec( $V, l, t_1 - 1, k$ )
17 |   senão se  $k < V[t_2]$  então
18 |   |   retorna buscaTer_rec( $V, t_1 + 1, t_2 - 1, k$ )
19 |   senão
20 |   |   retorna buscaTer_rec( $V, t_2 + 1, r, k$ )
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, compara-se o m -ésimo elemento ao valor buscado, caso seja igual, retorna-se m , se for menor, 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: $l \leq r$; $l, r \geq 0$; 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  |    $p \leftarrow \sqrt{r - l + 1}$ 
6  |    $m \leftarrow l + p$ 
7  |   enquanto  $m \leq r$  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 |       |    $m \leftarrow m + p$ 
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 tamanho diferente. O tamanho da primeira partição é o menor número da série de Fibonacci $F(i)$ tal que o tamanho do arranjo n é maior ou igual a $Fib(i + 2)$. 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: $l \leq r$; $l, r \geq 0$; 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  $\leftarrow r - l + 1$ 
   /* Calcula a serie de Fibonacci até o i-ésimo termo,
   onde F(i) <= size */
8  |   Fib[0]  $\leftarrow 0$ 
9  |   Fib[1]  $\leftarrow 1$ 
10 |   i  $\leftarrow 1$ 
11 |   enquanto Fib[i] < size faça
12 |   |   i  $\leftarrow i + 1$ 
13 |   |   Fib[i]  $\leftarrow Fib[i - 1] + Fib[i - 2]$ 
14 |   fim
15 |
16 |    $i_{fib1} \leftarrow i - 2$ 
17 |   enquanto  $l < r$  faça
18 |   |   fib1  $\leftarrow l + F[i_{fib1}]$ 
19 |   |   se  $k == V[fib1]$  então
20 |   |   |   retorna fib1
21 |   |   senão se  $k < V[fib1]$  então
22 |   |   |   /* Novos tamanhos das partições à esquerda */
23 |   |   |   r  $\leftarrow i_{fib1} - 1$ 
24 |   |   |    $i_{fib1} = i_{fib1} - 2$ 
25 |   |   |   se  $i_{fib1} < 0$  então
26 |   |   |   |    $i_{fib1} \leftarrow 0$ 
27 |   |   fim
28 |   |   senão
29 |   |   |   /* Procure os novos tamanhos das partições à direita */
30 |   |   |   l  $\leftarrow i_{fib1} + 1$ 
31 |   |   |   i  $\leftarrow i_{fib1} + 1$ 
32 |   |   |   enquanto Fib[i] < r - l + 1 faça
33 |   |   |   |   i  $\leftarrow i - 1$ 
34 |   |   |   fim
35 |   |    $i_{fib1} = i - 1$ 
36 |   fim
37 fim
   retorna -1

```

2.3 Cenários das simulações

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

2.4 metodologia

Para comparar os diferentes algoritmos de busca, simulações do tempo de execução para diferentes tamanhos de arranjos de entrada foram feitas. Também para analisar a diferença entre a implementação iterativa e recursiva do algoritmo???? foram feitas simulações medindo o número de passos da operação dominante.

Um vetor de inteiros longos de tamanho 10^8 preenchido com números pares em ordem crescente foi utilizado para gerar as amostras.

50 amostras do vetor foram utilizadas com tamanhos variando de 100 até 10^8 , com crescimento linear. ????

2.4.1 Simulações de tempo de execução

Utilizou-se a biblioteca Chronos com precisão de microsegundos 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 recursiva:

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

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

2.4.2 Simulações do número de passos da operação dominante

Chapter 3

Resultados

Os resultados obtidos com as simulações estão descrito abaixo.

3.1 Busca linear

Na figura abaixo temos o resultado de Fazendo um ajuste dos pontos da curva conseguimos determinar que o comportamento é linear.

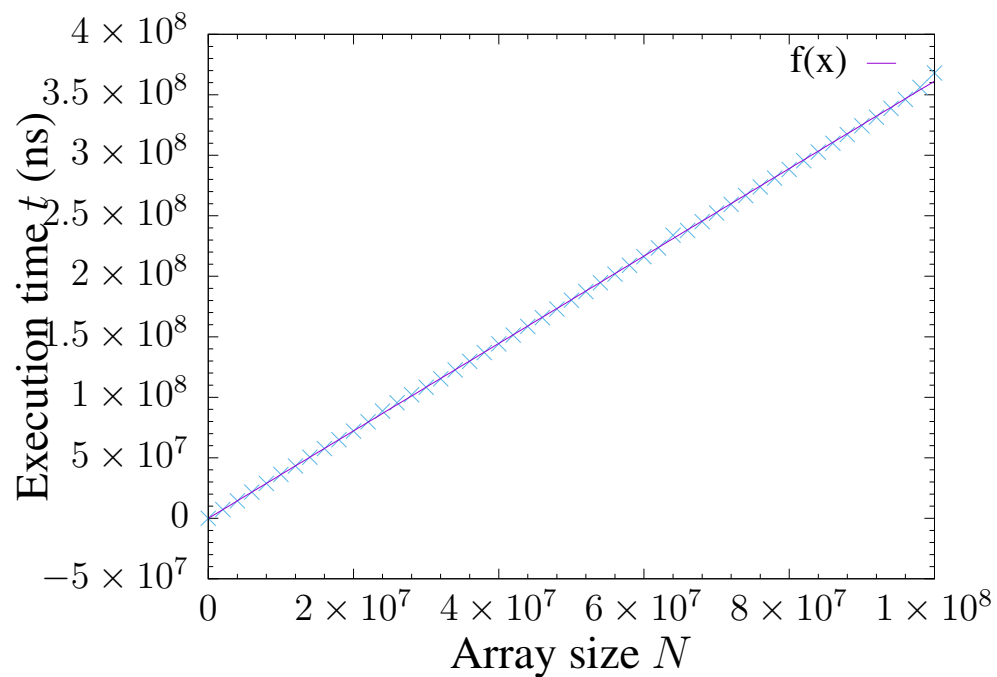


Figure 3.1: Tempo vs...

3.2 Busca binária

Na figura abaixo temos o resultado de Fazendo um ajuste dos pontos da curva conseguimos determinar que o comportamento é logarítmico.

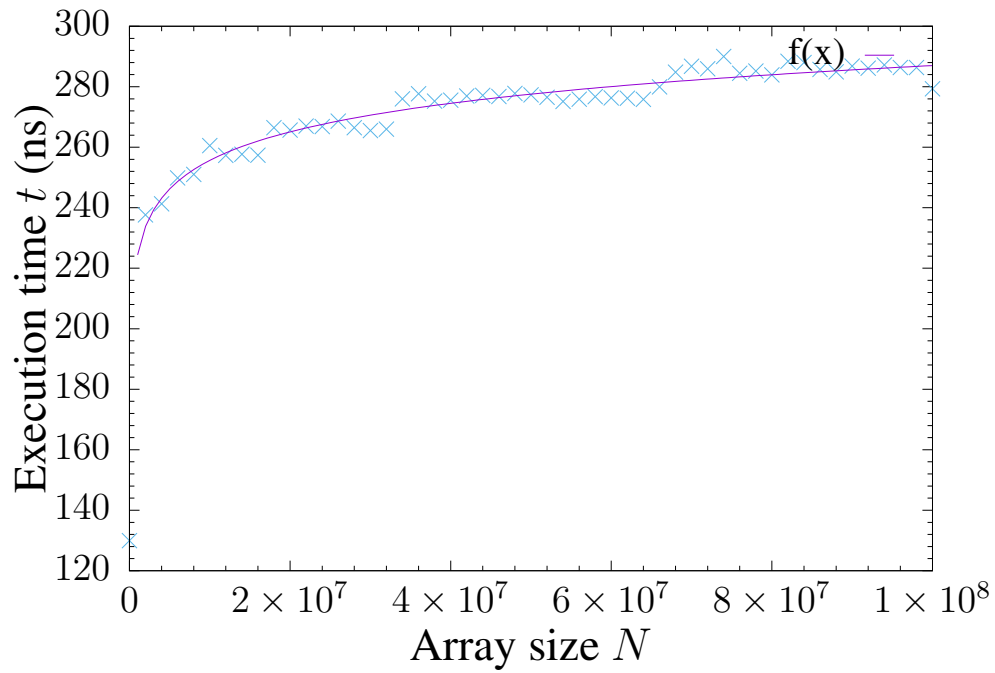


Figure 3.2: Tempo vs...

Na figura abaixo temos o resultado de Fazendo um ajuste dos pontos da curva conseguimos determinar que o comportamento é logarítmico.

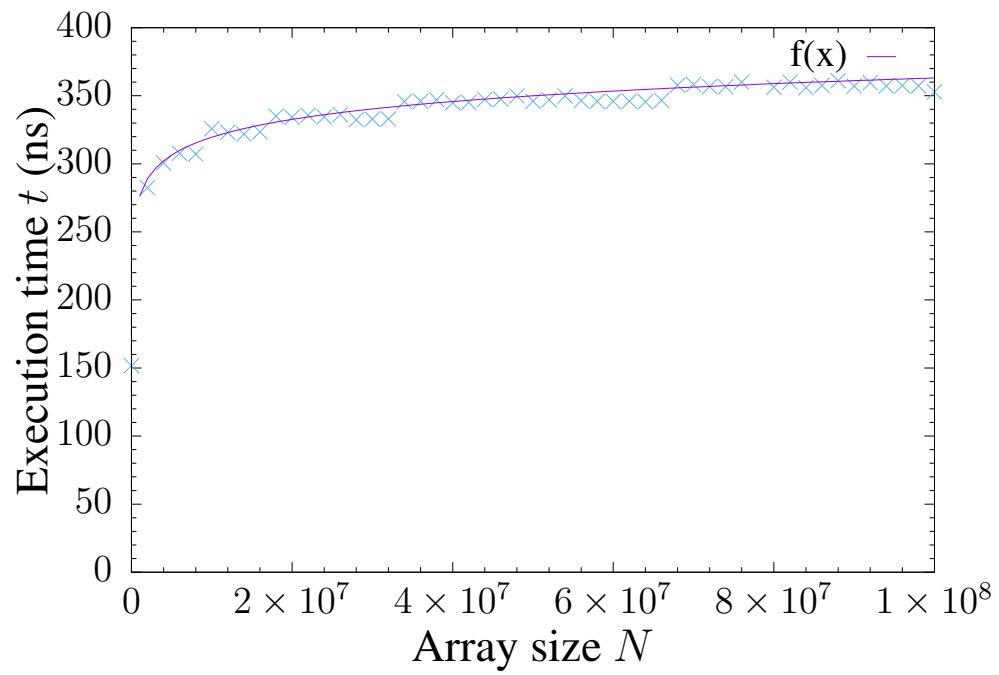


Figure 3.3: Tempo vs...

3.3 Busca ternária

Na figura abaixo temos o resultado de Fazendo um ajuste dos pontos da curva conseguimos determinar que o comportamento é logarítmico.

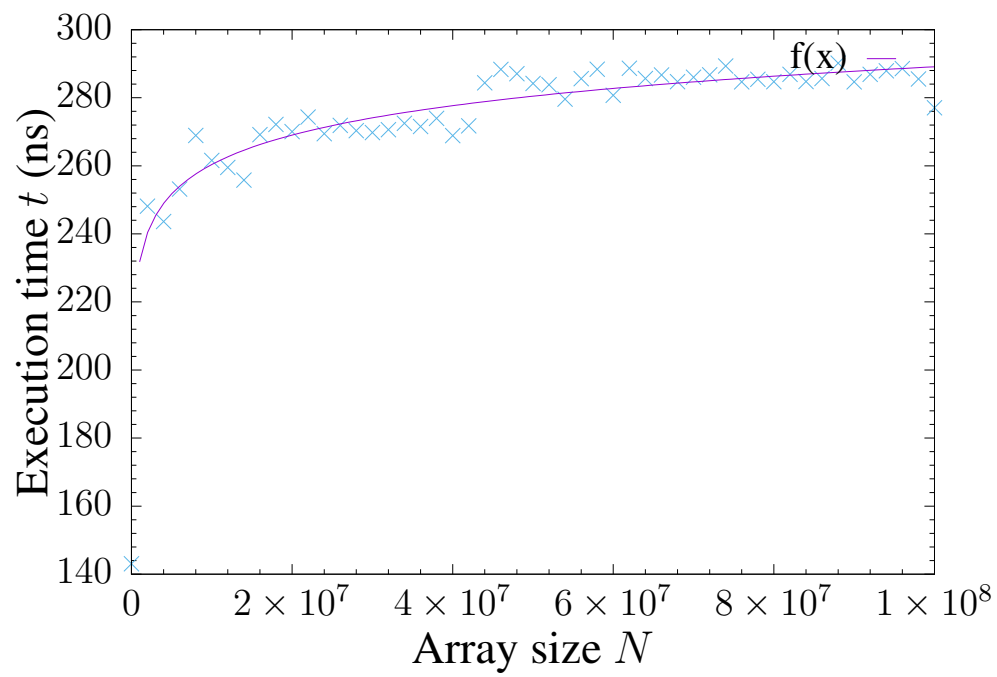


Figure 3.4: Tempo vs...

Na figura abaixo temos o resultado de Fazendo um ajuste dos pontos da curva conseguimos determinar que o comportamento é logarítmico.

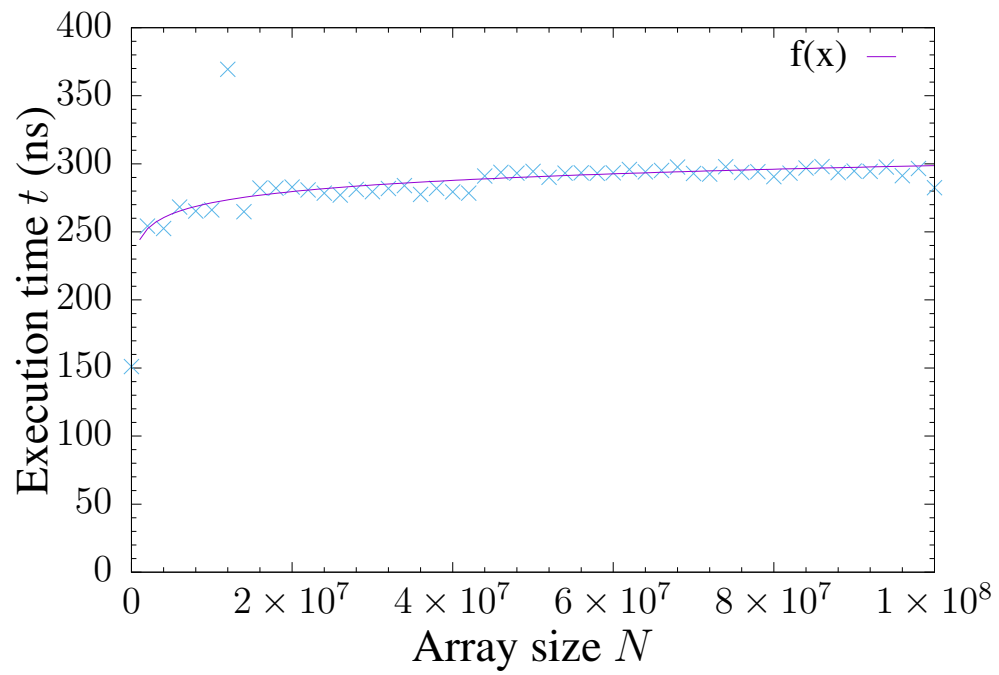


Figure 3.5: Tempo vs...

3.4 *Jump search*

Na figura abaixo temos o resultado de Fazendo um ajuste dos pontos da curva conseguimos determinar que o comportamento é logarítmico.

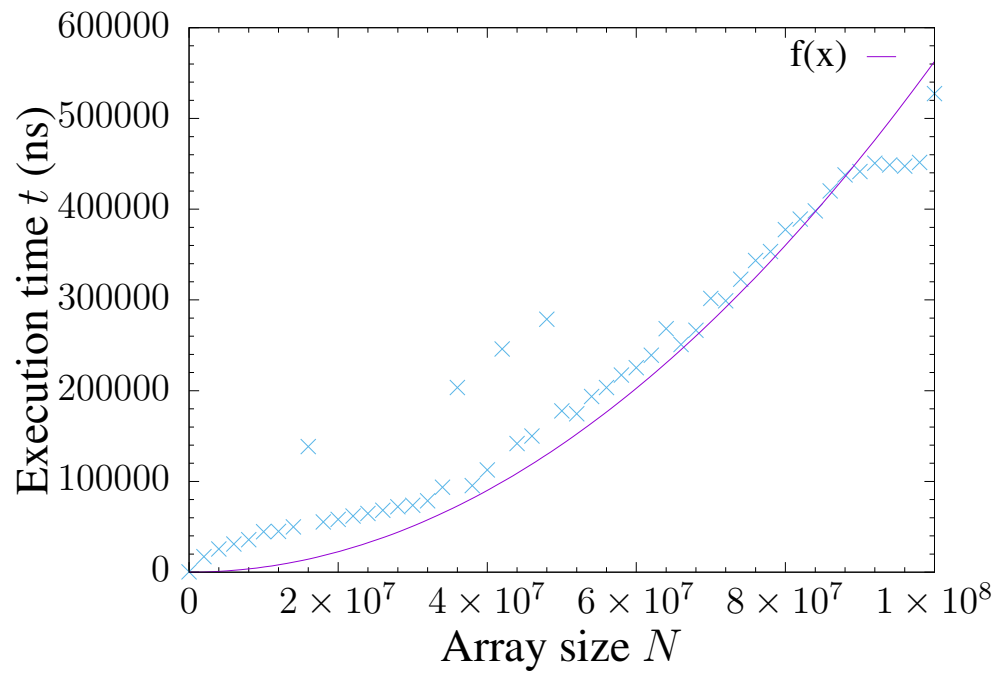


Figure 3.6: Tempo vs...

3.5 Busca de Fibonacci