Relatorio

Karine Piacentini Coelho da Costa[1]

March 21, 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 no capítulo 3.

2.1 Características técnicas

Os algorítmos 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 operaciona: (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. */`

```
Função buscaLin(V: arranjo de inteiros; l: inteiro; r: inteiro; k:
 inteiro): inteiro
    var i: inteiro
    para i ← l até r faça
        se V[i] == k então
            retorna i
        fim
    fim
    retorna −1
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 descrito anteriormente 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.          */
Função buscaBin_it(V: arranjo de inteiros; l: inteiro; r: inteiro; k: inteiro): inteiro
    var m: inteiro /* último valor da primeira metade do arranjo  */

    enquanto r ≥ l faça
        m ← (l + r)/2
        se k == V[m] então
            retorna m
        senão se k < V[m] então
            r ← m − 1
        senão
            l ← m + 1
        fim
    fim
    retorna −1
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 ≤ r; l,r ≥ 0; V em ordem crescente. */
Função buscaBin_rec(V: arranjo de inteiros; l: inteiro; r: inteiro; k:
 inteiro): inteiro
    var m: inteiro /* último valor da primeira metade do arranjo */

    se r < l então
        retorna −1
    senão
        m ← (l + r)/2
        se k == V[m] então
            retorna m
        senão se k < V[m] então
            retorna buscaBin_rec(V,l,m − 1,k)
        senão
            retorna buscaBin_rec(V,m + 1,r,k)
        fim
    fim
fim
```

2.2.3 Busca ternária

A busca ternário 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 ≤ r;  l, r ≥ 0;  V em ordem crescente.  */
Função buscaTer_it(V: arranjo de inteiros; l: inteiro; r: inteiro; k: inteiro): inteiro
    var t1: inteiro /* último valor do primeiro terço do arranjo */
    var t2: inteiro /* último valor do segundo terço do arranjo */

    enquanto r ≥ l faça
        t1 ← l + (r − l)/3
        t2 ← r − (r − l)/3

        se k == V[t1] então
            retorna t1
        senão se k == V[t2] então
            retorna t2
        senão se k < V[t1] então
            r ← t1 − 1
        senão se k < V[t2] então
            l ← t1 + 1
            r ← t2 − 1
        senão
            l ← t2 + 1
        fim
    fim
    retorna −1
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 ≤ r;  l, r ≥ 0;  V em ordem crescente.                    */
Função buscaTer_rec(V: arranjo de inteiros; l: inteiro; r: inteiro; k:
  inteiro): inteiro
    var t1: inteiro /* último valor do primeiro terço do arranjo   */
    var t2: inteiro /* último valor do segundo terço do arranjo    */

    se r < l então
        retorna −1
    senão
        t1 ← l + (r − l)/3
        t2 ← r − (r − l)/3

        se k == V[t1] então
            retorna t1
        senão se k == V[t2] então
            retorna t2
        senão se k < V[t1] então
            retorna buscaTer_rec(V,l,t1 − 1,k)
        senão se k < V[t2] então
            retorna buscaTer_rec(V,t1 + 1,t2 − 1,k)
        senão
            retorna buscaTer_rec(V,t2 + 1,r,k)
        fim
    fim
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. */`

Função `buscaJump`(V: **arranjo de inteiros**; l: **inteiro**; r: **inteiro**; k: **inteiro**): **inteiro**
  var m: **inteiro**
  var p: **inteiro** `/* tamanho do salto */`

  $p \leftarrow \sqrt{r - l + 1}$
  $m \leftarrow l + p$
  enquanto $m \leq r$ **faça**
    se $k\ ==\ \mathsf{V}[m]$ **então**
      retorna m
    senão se $k\ <\ \mathsf{V}[m]$ **então**
      retorna `buscaLin`$(\mathsf{V}, m - p, m - 1, k)$
    fim
    $m \leftarrow m + p$
  fim
  se $m > r$ **e** $\mathsf{V}[r] > k$ **então**
    retorna `buscaLin`$(\mathsf{V}, m - p, r, k)$
  fim
  retorna -1
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 ≤ r;  l, r ≥ 0;  V em ordem crescente.  */
Função buscaBin_it(V: arranjo de inteiros; l: inteiro; r: inteiro; k: inteiro): inteiro
    var size: inteiro
    var i: inteiro
    var i_fib1: inteiro
    var fib1: inteiro
    var Fib: arranjo de inteiros
    size ← r − l + 1
    /* Calcula a serie de Fibonacci até o i-ésimo termo,
    onde F(i) <= size  */
    Fib[0] ← 0
    Fib[1] ← 1
    i ← 1
    enquanto Fib[i] < size faça
        i ← i + 1
        Fib[i] ← Fib[i − 1] + Fib[i − 2]
    fim

    i_fib1 ← i − 2
    enquanto l < r faça
        fib1 ← l + F[i_fib1]
        se k == V[fib1] então
            retorna fib1
        senão se k < V[fib1] então
            /* Novos tamanhos das partições à esquerda  */
            r ← fib1 − 1
            i_fib1 = i_fib1 − 2
            se i_fib1 < 0 então
                i_fib1 ← 0
            fim
        senão
            /* Procure os novos tamanhos das partições à direita  */
            l ← fib1 + 1
            i ← i_fib1 + 1
            enquanto Fib[i] < r − l + 1 faça
                i ← i − 1
            fim
            i_fib1 = i − 1
        fim
    fim
    retorna −1
fim
```

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 análisar 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 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.

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.

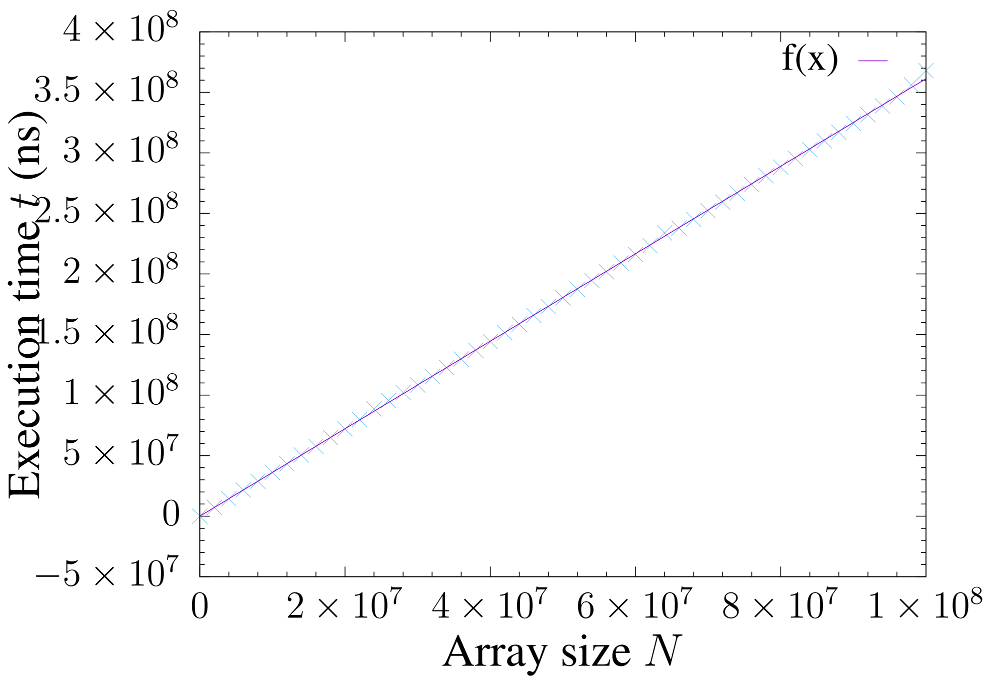

Figure 3.1: Tempo vs...

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.

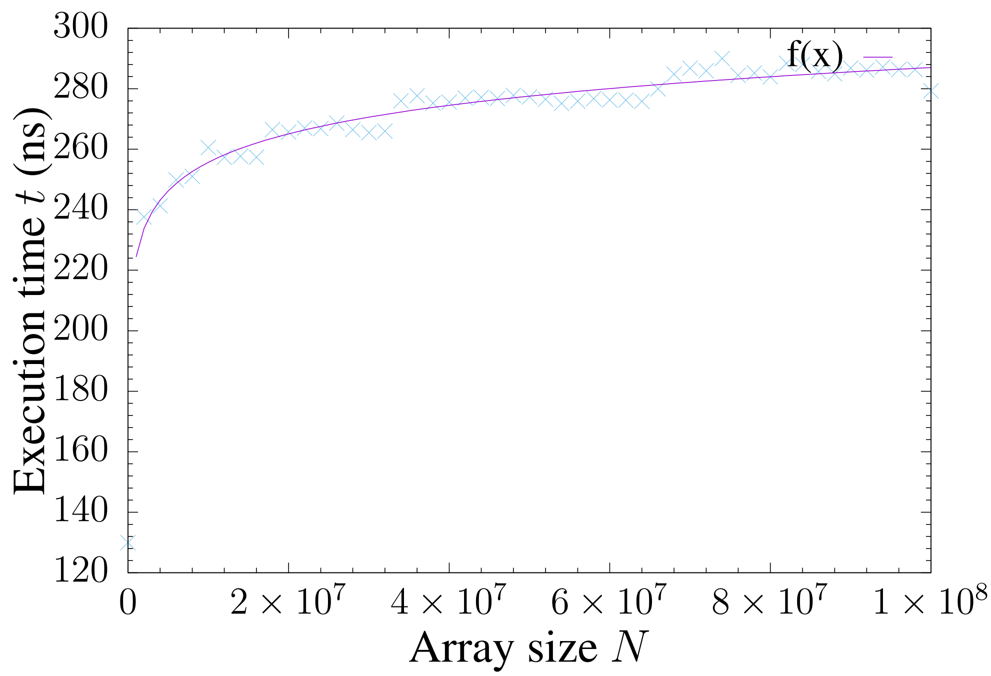

Figure 3.2: Tempo vs...

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

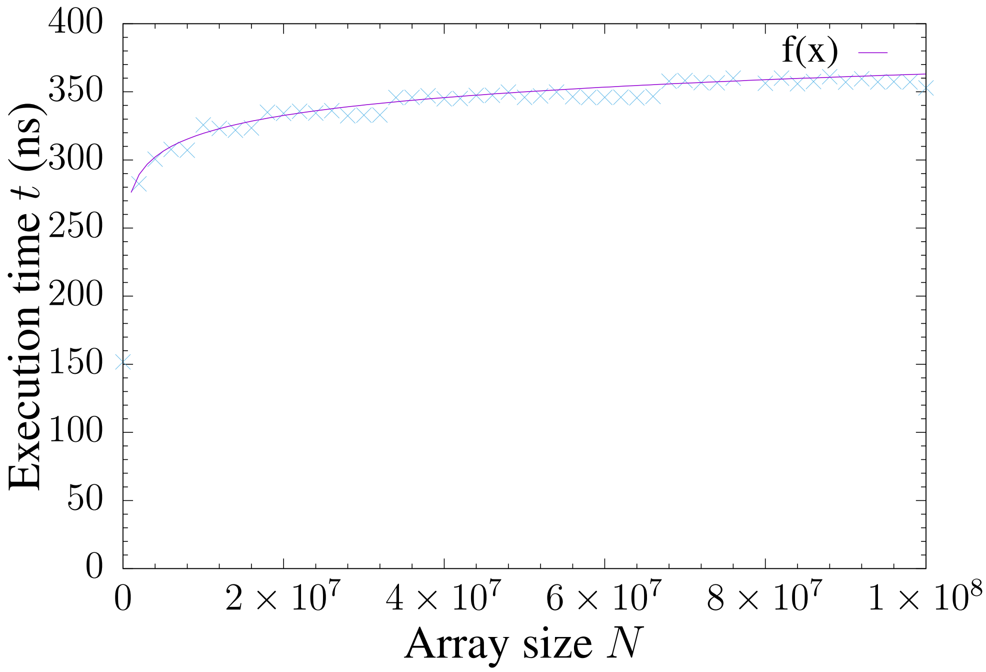

Figure 3.3: Tempo vs...

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.

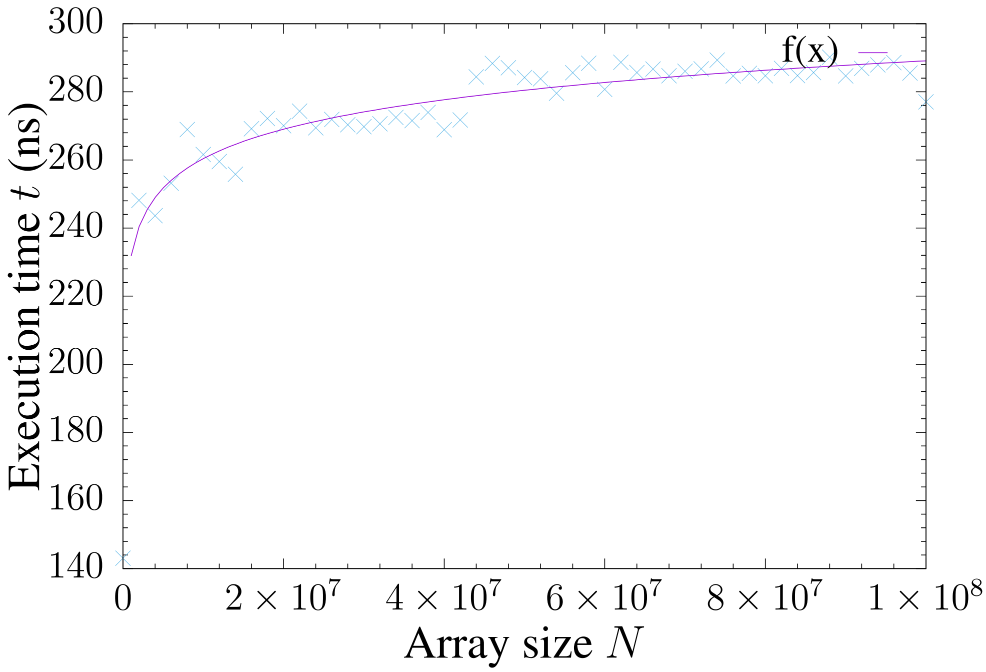

Figure 3.4: Tempo vs...

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

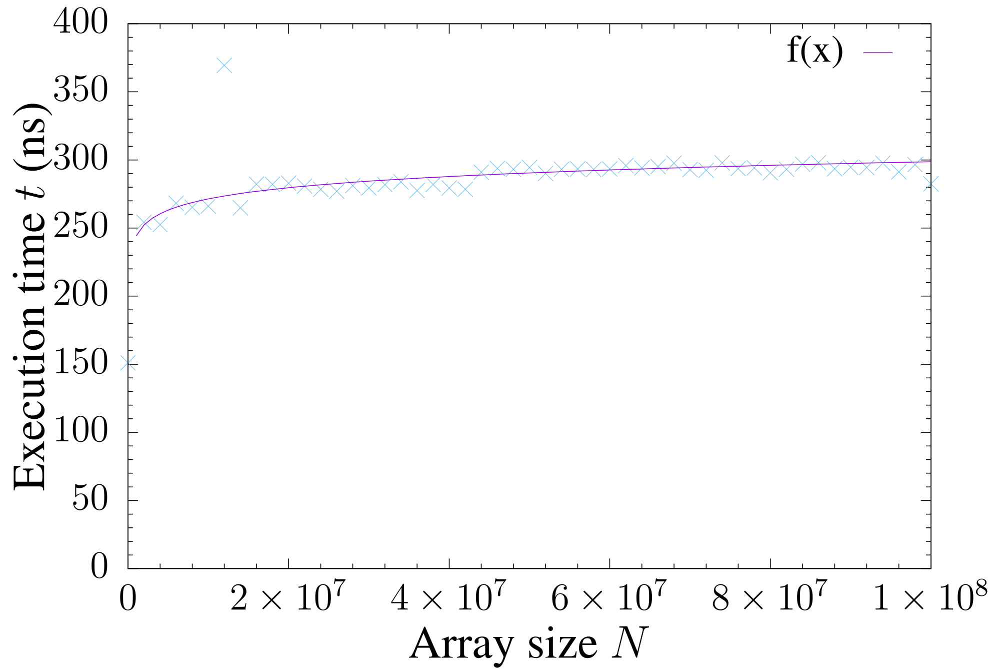

Figure 3.5: Tempo vs...

3.4 *Jump search*

Na figura abaixo temos o resultado de .... Fazendo um ajuste dos pontos da curva conseguimos determinar que o comportamento se aproxima de um comportamento quadrático.

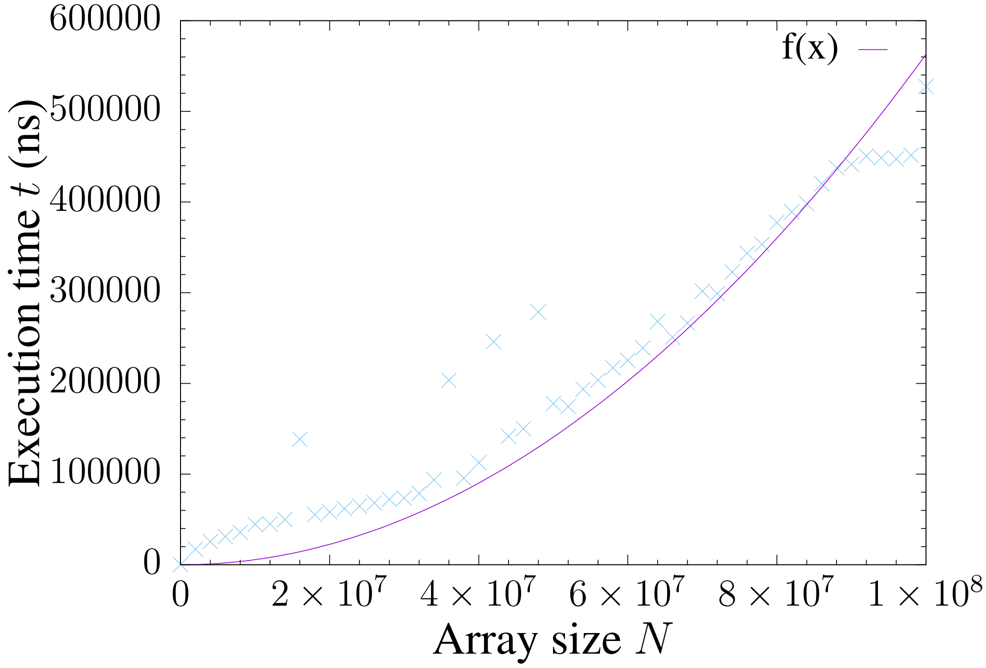

Figure 3.6: Tempo vs...

3.5 Busca de Fibonacci