UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

INSTITUTO METRÓPOLE DIGITAL

Leonandro Valério Barbosa Gurgel

Camila Duarte de Souza

Análise empírica sobre complexidade de algoritmos de busca

NATAL/RN

2019

Sumário:

1 Introdução

Este trabalho visa realizar uma análise empírica sobre os diferentes tipos de algoritmos de busca, além de determinar quais configurações de entrada afetam o desempenho desses algoritmos. O problema computacional utilizado para implementação das funções foi a busca em um arranjo sequencial. A utilização da biblioteca C++ “chronos” foi um ponto crucial para que se conseguisse determinar o tempo de compilação de cada função dos algoritmos de busca solicitados; e para uma ajuda da visualização dessa comparação, também foi utilizada a criação de gráficos.

2 Métodos e Materiais

Para a análise, cada algoritmo de busca foi submetido a 26 entradas de dados tais que a entrada posterior é o dobro da anterior, as entradas começam com tamanho 32 e terminam com tamanho 536.870.912, e, para cada entrada, o algoritmo de busca, que estava sendo analisado, foi executado 100 vezes e depois foi tirada uma média desses tempos, conforme foto abaixo do código.

2.1 Hardware

Para realizar esta análise, foi utilizado o notebook Aspire A515-51G da fabricante Acer, que conta com 8 gigas de memória ram de tipo DDR4 e um processador Intel core i5 7200 com clock de 2.51Ghz a 2.71Ghz.

2.2 Software

Os algoritmos foram implementados na linguagem C++ em sua versão C11, foi usado o compilador g++ da família GCC(GNU Compiler Collection) na versão 8.2. Todos os códigos foram executados no sistema operacional Ubuntu versão 18.04 LTS. Os gráficos referentes às simulações foram gerados com o software Gnuplot em sua versão 5.2.

2.2.1 Busca linear

O algoritmo de busca linear foi implementada em sua forma iterativa de acordo com o pseudo-código abaixo.

```
function busca_linear (*inicio, *fim, alvo):
    while (inicio != fim):
        if(*(inicio) == alvo):
```

```
        return inicio
    inicio++
return unsuccessful
```

2.2.2 Busca Binária Iterativa

O algoritmo de busca binária iterativa foi implementado de acordo com o pseudo-código abaixo:

```
function binary_iterative(*inicio, *fim, alvo):
    while (inicio <= fim):
        adicionar_indice := (inicio-fim)/2
        if (*(inicio+adicionar_indice) == alvo):
            return (inicio+adicionar_indice)
        else:
            if (*(inicio+adicionar_indice) > alvo:
                fim = (inicio+(adicionar_indice-1))
            else:
                inicio = (inicio+(adicionar_indice+1))
    return unsuccessful
```

2.2.3 Busca Binária Recursiva

O algoritmo de busca binária recursiva foi implementado de acordo com o pseudo-código abaixo:

```
function binaria_recursiva(*inicio, *fim, alvo):
    adicionar_indice := (fim - inicio)/2
    if(inicio <= fim):
        if(*(inicio + adicionar_indice)  ==  alvo):
            return (inicio + adicionar_indice)
        else:
            if(*(inicio + adicionar_indice) > alvo):
                fim = *(inicio + (adicionar_indice - 1))
                binaria_recursiva(inicio, fim, alvo)
```

```
        else:
            inicio += adicionar_indice + 1
            binaria_recursiva(inicio, fim, alvo)
    else:
        return unsuccessful
```

2.2.4 Busca Ternária Iterativa

A busca ternária iterativa foi implementada de acordo com o pseudo-código abaixo.

```
function ternária_iterativa(*inicio, *fim, alvo):
    while(inicio <= fim):
        adicionar_indice := (fim - inicio)/3
        if(*(inicio + adicionar_indice + adicionar_indice) ==  alvo):
            return(inicio + adicionar_indice + adicionar_indice)
        else:
            if(*(inicio + adicionar_indice + adicionar_indice) < alvo):
                inicio += ( adicionar_indice + adicionar_indice + 1)
            else:
                if(*(inicio + adicionar_indice) == alvo):
                    return (inicio + adicionar_indice)
                else:
                    if(*(inicio + adicionar_indice) < alvo):
                        inicio := adicionar_indice + 1
                        fim := (inicio + (adicionar_indice + adicionar_indice-1))
                    else:
                        fim := (inicio +( adicionar_indice - 1))
    return unsuccessful
```

2.2.5 Busca Ternária Recursiva

O algoritmo de busca ternária recursiva foi implementado de acordo com o pseudo-código abaixo:

```
function ternaria_recursiva(*inicio, *fim, alvo):
    adicionar_indice := (fim - inicio)/3
    if(inicio <= fim):
        if(*(inicio + adicionar_indice + adicionar_indice) == alvo):
            return (inicio + adicionar_indice + adicionar_indice)
        else:
            if(*(inicio + adicionar_indice + adicionar_indice) < alvo):
                inicio += adicionar_indice + adicionar_indice +1
                ternaria_recursiva(inicio, fim, valor)
            else:
                if(*(inicio + adicionar_indice) == alvo):
                    return (inicio + adicionar_indice)
                else:
                    if(*(inicio + adicionar_indice) < alvo):
                        inicio += adicionar_indice + 1
                        fim := (inicio + (adicionar_indice + adicionar_indice -1))
                        ternaria_recursiva(inicio, fim, valor)
                    else:
                        fim := (inicio + (adicionar_indice - 1))
                        ternaria_recursiva(inicio, fim, valor)
    return  unsuccessful
```

2.2.6 Busca de Fibonacci

O algoritmo de busca de Fibonacci foi implementado de acordo com o pseudo-código abaixo:

```
function fibonacci_search(*inicio, *fim, *alvo):
    while (inicio != fim):
        receberValores := fibonacci_calculator(fim-inicio)
        adicionar_indice1 := receberValores[0]
        adicionar_indice2 := receberValores[1]
```

```
    if( *(inicio+adicionar_indice1) == alvo):
        return ((inicio+adicionar_indice1) - clone_inicio)
    else if (inicio+adicionar_indice2 - clone_inicio == alvo):
      return ((inicio+adicionar_indice2) - clone_inicio)
    else:
      if (*(inicio+adicionar_indice1) > alvo):
          fim = inicio+adicionar_indice1
      else if (*(inicio+adicionar_indice1) < alvo)
          inicio = inicio+(adicionar_indice1+1)
      else if (*(inicio+adicionar_indice2) > alvo):
          fim = inicio+adicionar_indice2
      else if (*(inicio+adicionar_indice2) < alvo):
          inicio = inicio+(adicionar_indice2+1)
return unsuccessful
```

2.2.7 Jump Search

O algoritmo de busca Jump Search foi implementado de acordo com o pseudo-código abaixo:

```
function jump_search(*inicio, *fim, alvo):
  pulo := piso(raiz(fim-inicio))
  if(*(inicio)==alvo):
    return first
  else:
    while (*(inicio) <= alvo e inicio <= fim):
      inicio := inicio + pulo
     *aux := inicio - pulo
    while (aux <= inicio):
      if(*(aux) == alvo):
        return aux
      else:
        aux := aux + 1
```

return unsuccessful

2.2.8 List calculator

Função pensada para gerar um vetor de testes com ordenação crescente e com elementos distintos. Implementada de acordo com o pseudo-código abaixo.

function gera_vetor(tamanho):

vetor := ***new*** int[tamanho]

for(i=0; i<tamanho; i++):

vetor[i] := i+1

return vetor

2.2.9 Main

Função principal onde são chamadas as funções e realizados os testes, e, por fim, são gravados no arquivo os dados referentes ao algoritmo e seus números. É bom notar que no pseudo-código abaixo, eu uso a função: busca_linear. Porém esse é somente um exemplo, ficando ao critério do usuário a escolha da função. Recomenda-se que o arquivo que for guardar os dados, tenha o mesmo nome que o algoritmo de busca.

function main():

stream dados

dados.***open***(“busca_linear.dat”, ofstream::app)

tamanho := 32

for (j = 1; j <= 25; j++):

alvo := tamanho+2

vetor_base := ***gera_vetor***(tamanho)

for (i = 1; i <= 100; i++):

contador_inicial := ***steady_clock::now***()

busca_linear(***inicio***(vetor_base), ***final***(vetor_base), alvo)

contador_final := ***steady_clock::now***()

```
        tempos += (contador_final - contador_inicial)
    dados << tamanho << “  “ << tempos/100 << endl
    tamanho := tamanho*2
dados.close()
```

3 Resultados

Como supracitado, os algoritmos foram submetidos ao seu pior caso, e todos sob as mesmas condições. Com uma entrada de dados variando entre 32 e 536.870.912. Cada algoritmo foi testado 100 vezes para cada entrada de dados e daí foi tirado seu tempo médio de cada caso e foram gravados em arquivo a entrada de dados e o tempo médio que ele levou para ser executado. A partir desses arquivos, foram gerados diversos gráficos com o software Gnuplot.

3.1 Busca Linear x Jump Search

Foram testados os algoritmos: busca linear e jump search. E os resultados estão descritos no gráfico

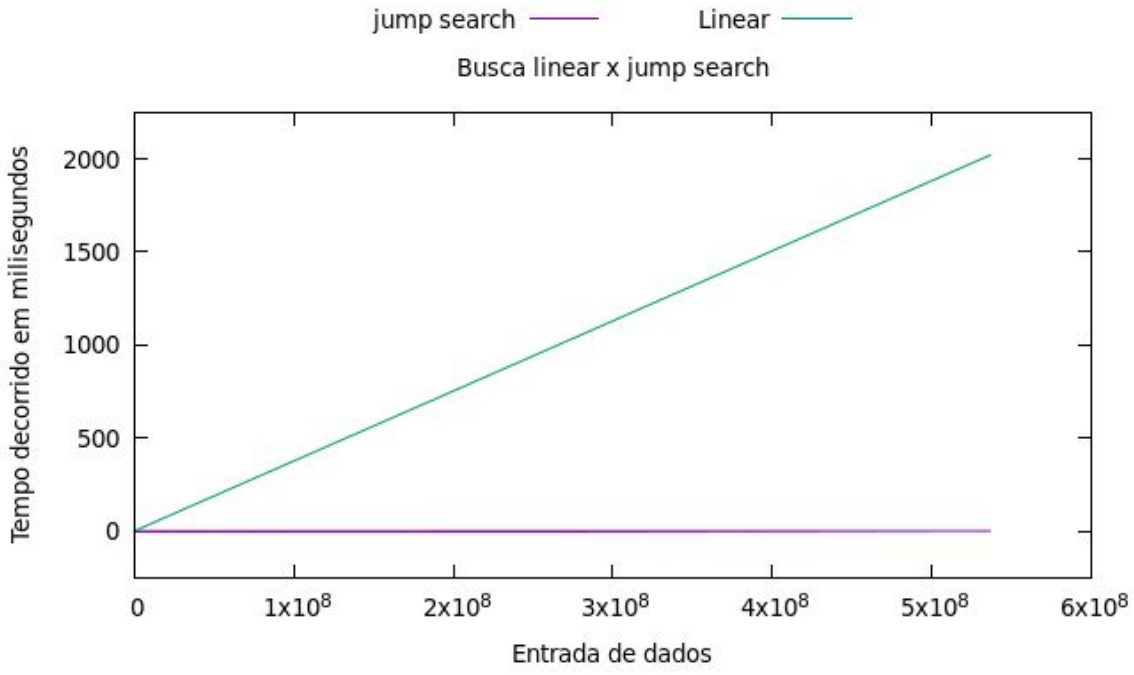

Figura 1 - Busca linear x jump search. Gnuplot ver 8.2

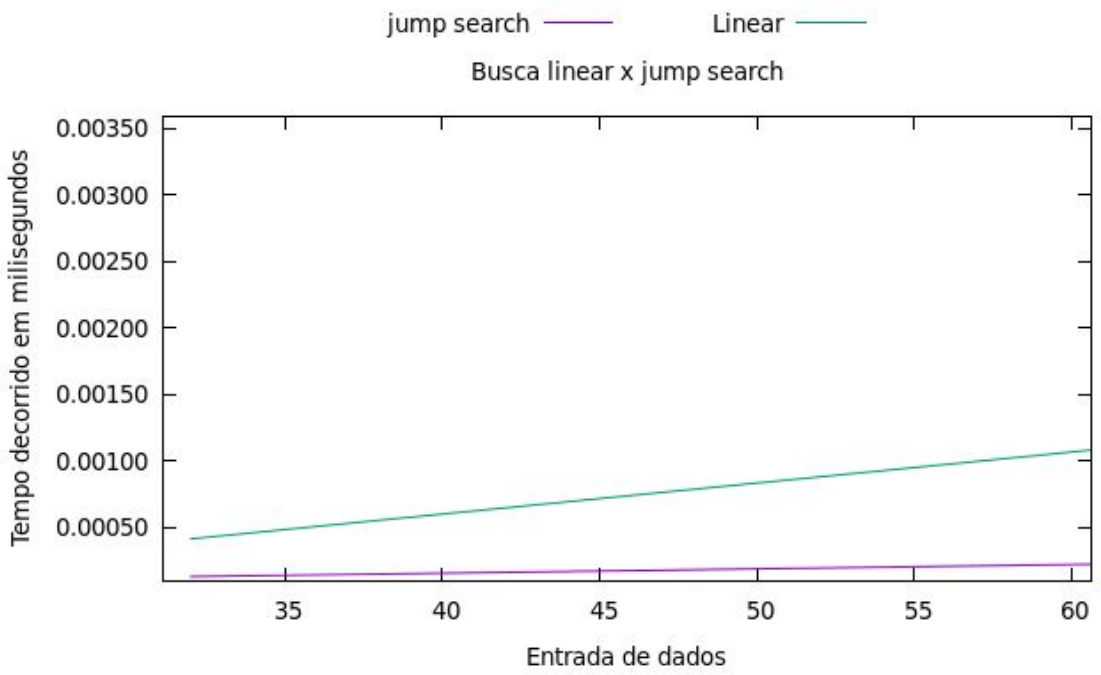

Figura 2 - Busca linear x jump search, com zoom. Gnuplot ver 8.2

3.2 Recursão x Iteração

Usando o algoritmo de busca binária, foi testada suas versões: iterativa e recursiva.

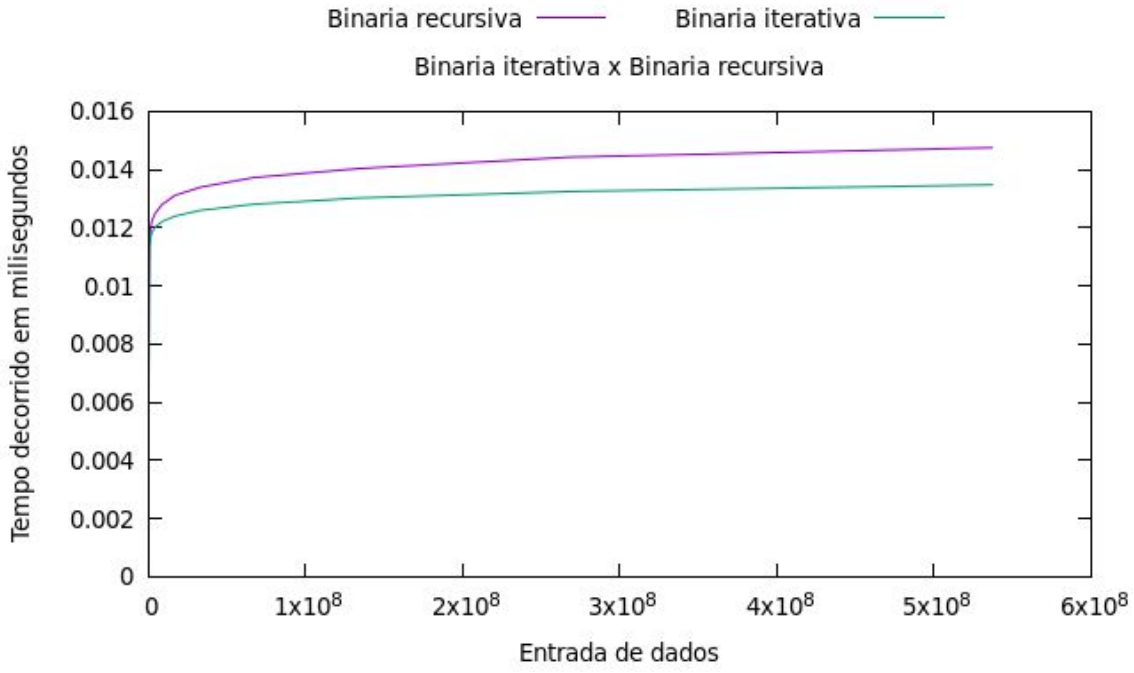

Figura 3 - Busca binária: recursiva x iterativa. Gnuplot ver 8.2

3.3 Tamanho da partição

Usando os algoritmos em suas formas iterativas, foi medida a eficiência das buscas: binária, ternária e fibonacci. Ambas com complexidade logarítmica, se diferenciando basicamente nos tamanhos de suas partições.

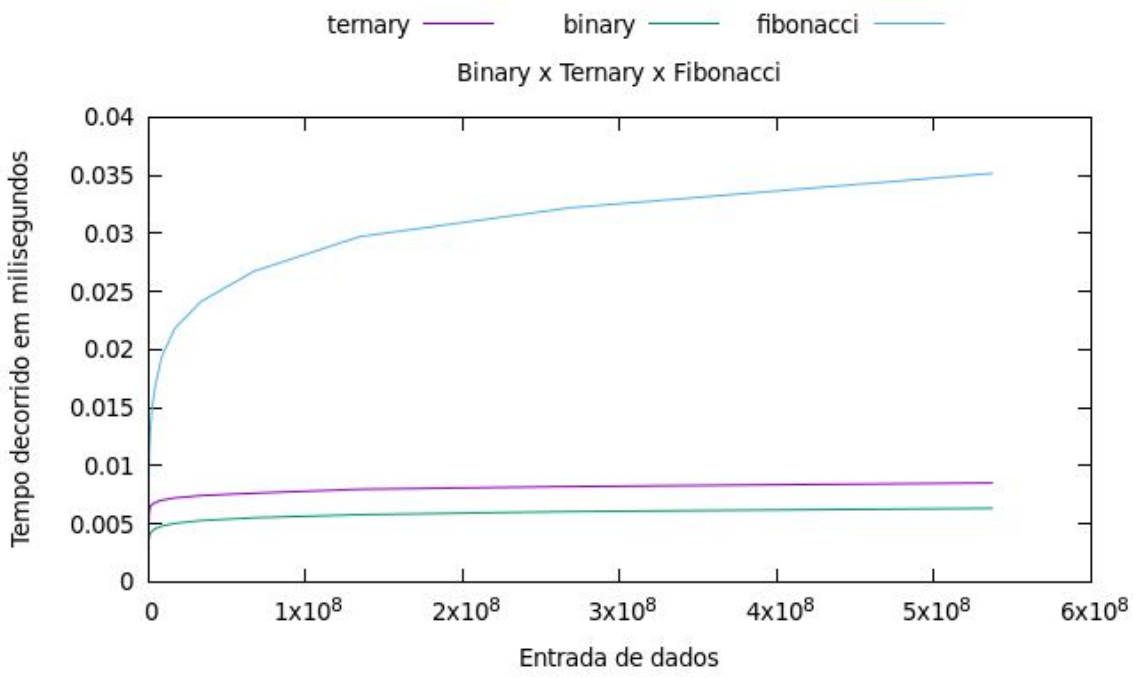

Figura 4 - Busca binária x Ternária x Fibonacci. Gnuplot ver 8.2

3.4 Complexidades diferentes

Foram testados os algoritmos: busca linear e busca binária iterativa, a primeira de complexidade linear e a segunda de complexidade logarítmica.

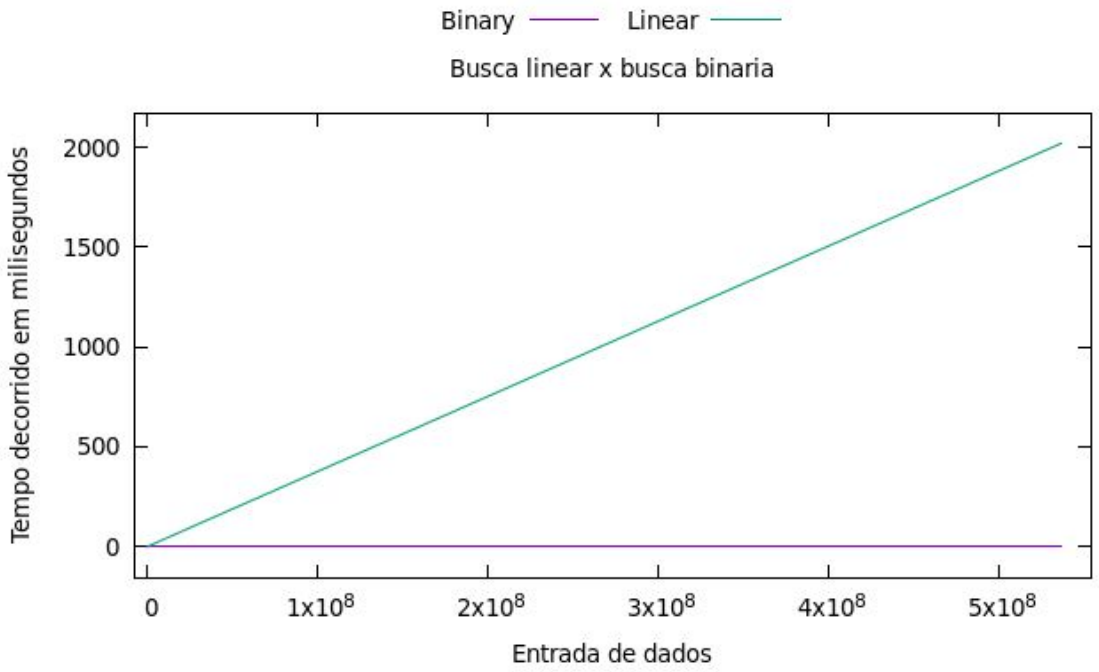

Figura 5 - Busca binária x busca linear. Gnuplot ver 8.2

Vemos aqui a imagem com zoom para os primeiros casos de testes.

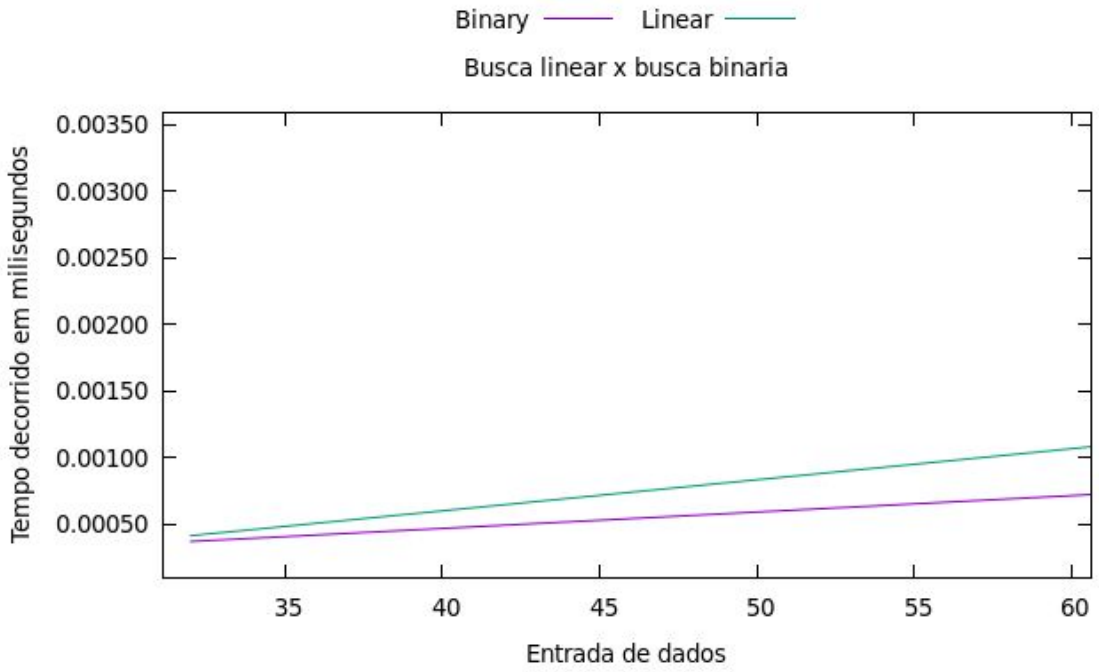

Figura 6 - Busca binária x busca linear, com zoom. Gnuplot ver 8.2

3.5 O pior caso da busca fibonacci

Foi testado o algoritmo da busca de fibonacci para dois casos: um busc-

ando um valor logo após o último elemento do vetor e no outro caso, buscando um elemento anterior ao primeiro elemento do vetor. Sendo representado no gráfico como: Fibonacci (+1) para o elemento acima do vetor e Fibonacci (-1) para o elemento abaixo do vetor.

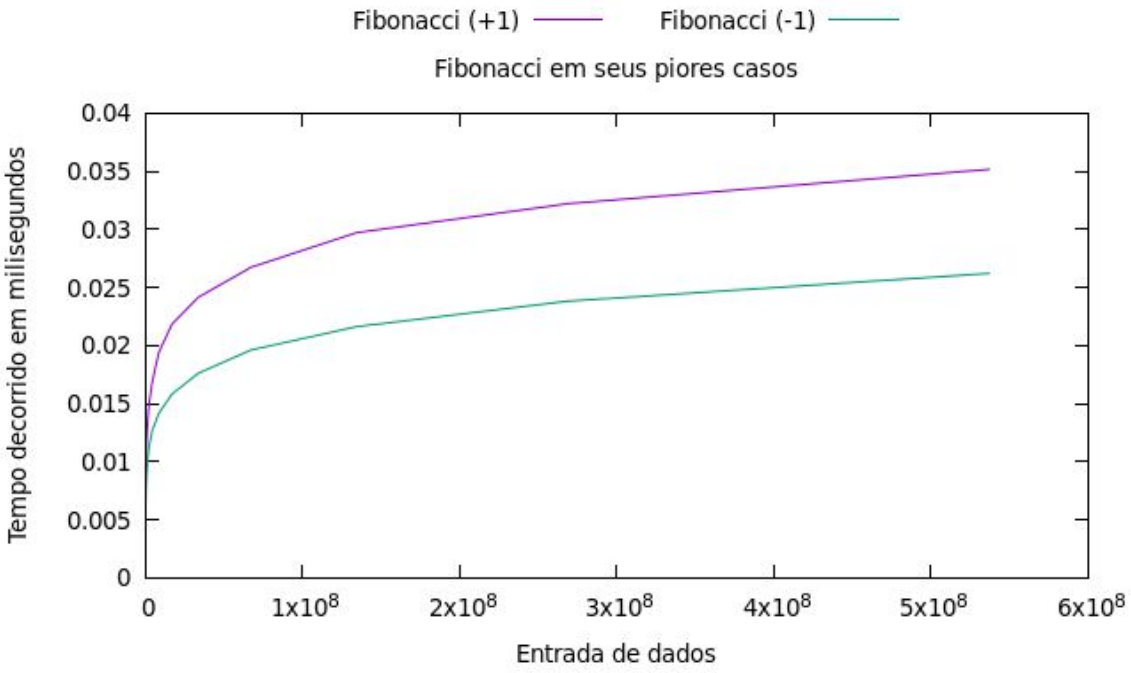

Figura 7 - Busca de fibonacci em seus piores casos. Gnuplot ver 8.2

4 Discussão

4.1 Busca linear x jump search

A jump search é claramente superior a busca linear, pois, ela sempre consegue buscar somente em um bloco específico bem menor em relação ao tamanho do vetor, enquanto a linear, em seu pior caso tem que percorrer todo o vetor, levando muito mais passos e consequentemente, tempo.

4.2 Busca binária: iterativa x recursiva

A partir da interpretação do gráfico da figura x, fica claro que as chamadas recursivas acabam sendo menos eficientes que as iterações, por consequência, a busca binária acaba sendo um pouco mais mais eficiente em sua versão iterativa.

4.3 Busca binária x busca ternária x busca de fibonacci

Pelos resultados obtidos na figura 4, o número de divisões aliado ao tamanho das partições acaba influenciando na eficiência. Como a busca binária realiza menos divisões a cada iteração, torna-se mais eficiente que a ternária e a fibonacci. Já a busca ternária acaba realizando partições menores que a fibonacci, principalmente com grandes entradas de dados, tornando-a mais eficiente que a fibonacci.

4.4 Busca binária x busca linear

Devido a diferença de complexidade entre as duas, podemos notar uma diferença muito grande de eficiência, enquanto a binária tem complexidade logarítmica e pouco se altera mesmo com quantias grandiosas de entrada de dados, a busca linear se torna muito ineficiente quando as entradas começam a ficar grandes, inclusive, a busca binária se mostra mais eficaz desde entradas muito pequenas até as maiores, testadas, como visto na figura 6.

5 Conclusão

Com esse projeto, chegamos a conclusão da importância de se observar o comportamento de diferentes algoritmos, observando-os nos piores casos para que seja feita a melhor escolha de implementação, para evitar uma demora desnecessária no tempo de execução do código, por exemplo.

Além disso, o uso de bibliotecas, como a Chrono, citada na introdução, auxiliam bastante na resolução de problemas computacionais, e a criação de gráficos são de suma importância para uma melhor observação das comparações feitas.

Com nossos resultados obtidos, pudemos perceber implementando os códigos no pior caso, que o algoritmo mais eficiente na situação dada foi o de busca binária, já que possui complexidade log(n). Além disso, pudemos observar também que o algoritmo menos eficiente nesse caso foi o de busca linear, que possui complexidade n, que foi a maior dos algoritmos implementados.