Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital
Bacharelado em Tecnologia da Informação
DIM0119: Estruturas de Dados Básicas I

Análise Empírica de Algoritmos de Busca

Discentes: Moisés Sousa Araújo
Bruna Hellen De Castro D. Barbosa
Docente: Selan Rodrigues dos Santos

Natal-RN
2019

Contents

1 Introdução

A busca de métodos eficientes que usem poucos recursos com resultados rápidos e efetivos é uma demanda constante. Devido a estas necessidades, é necessário reproduzir, comparar e simular as possíveis soluções dos problemas enfrentados, para adiantar-se no limiar teórico da construção lógica da solução à sua aplicação em um cenário real.

Hoje, os sistemas operam com uma quantidade massiva de dados e qualquer otimização de código representa diferença para o usuário final, a comparação de algortimos, chamada de análise empírica, permite a visualização do comportamento em um cenário estressante. Um algoritmo pode ser muito bom para um conjuntos de dados pequenos, entretanto à medida que aumentamos esse valor o tempo de execução também cresce e pode crescer vertigitosamente.

Nas próximas seções do relatório, serão discutidos o processo e os resultados obtidos da análise dos seguintes algoritmos de busca: busca linear, busca binária iterativa e recursiva, busca fibonacci, jump search, busca ternária iterativa e recusiva. Serão detalhadas suas performance individuais e comparadas a outras funções em um cenário que envolve uma grande quantidade de dados.

2 Metodologia

2.1 Materiais e Ferramentas

A implementação dos algoritmos foi feita na linguagem de programação C++ versão 11, o compilador usado foi usandoGCC(GNU Compiler Collection) v6.3.0 em conjunto com o CMake v3.7.2, GNU Make v4.1 em um notebook Lenovo Ideapad320 com as seguintes configurações:

2.1.1 Caracterização técnica da máquina usada

1. Processador
   - Intel® Core^{TM} i3 6^{a} geração i3-6006U
2. Memória RAM
   - 4GB

2.2 Métrica

Optou-se pela biblioteca std::chrono do C++11 para fazer a medição de tempo e a unidade escolhida foi microsegundos. A função utilizada do chrono foi a std::chrono::high$_{r}esolution_{c}lock :: now$.

2.3 Documentação

A documentação foi feita com o uso da ferramenta Doxygen, disponível para consultas por parte do usuário.

2.4 Cenário Predefinido

Devido a natureza da tarefa pedida, o cenário escolhido foi o pior caso para todos os algoritmos, ocorrendo quando a chave buscada não se encontra no arranjo submetido.

O dataset utilizado possui um total de $3x10^8$ elementos e por padrão foram feitos cerca de 20 amostras por busca.

2.5 A implementação

Ao compilar e iniciar o programa, é solicitada ao usuário a quantidade de intervalos desejada e quais buscas serão executadas, os algoritmos selecionados são executados após a alocação do vetor de buscas com a quantidade de elementos definida pelo usuário (ou uma quantidade padrão caso não seja passado nenhuma) menor ou igual ao total da máquina utilizada para os testes, durante seu tempo de execução a biblioteca std::chronos faz a contagem do intervalo de tempo até a chegada ao fim do vetor (tempo final - tempo inicial) através da função chronos::clock.

O resultado final é armazenado em um container e em seguida todo o conteúdo gerado é passado para um arquivo com o label da busca ou "$analyze_output.plt$". Os tempos de execução são resultantes da média de 100 execuções no intervalo definido pelo tamanho máximo do vetor de busca. Após a geração do arquivo, os dados são plotados com a ajuda da biblioteca gnuplot em um gráfico que relaciona o tempo (y) com o quantidade de itens (x).

As buscas são realizadas no pior caso possível, ou seja, o elemento não está no vetor de busca. Foram executadas 20 amostras de teste para cada busca e o vetor alvo das funções do tipo *long int* aloca um total de $3x10^8$ elementos.

2.6 Algoritmos

2.6.1 Busca Linear Iterativa

A busca linear iterativa usa uma estratégia simples: dado um arranjo de tamanho k percorre-se e compara-se todos os elementos do arranjo com o elemento pesquisado, se algum elemento do arranjo for igual, a posição do elemento igual no vetor é retornada, se não a busca continua. Caso o elemento não exista no arranjo retorna-se um valor inválido como -1. Na implementação dessa busca e de todas as outras é retornada um iterador para o último elemento do $std :: vector < longint > dataset$. Salientando que os iteradores $std :: vector < longint >:: iterator$ usam itr como seu namespace.

- Complexidade
  - Melhor caso: $O(1)$
  - Pior caso: $O(n)$

Código em C++ usado para análise:

```
//Busca Linear

 // a funcao recebe o valor a ser procurado e 2 iteradores para o nicio
     e fim do vetor alvo

itr linear(int key, itr l, itr r)
{
    //percorre todo o vetor
   while(l != r){
       //se o valor pesquisado for encontrado  retornado um iterador da
           sua posicao no vetor

      if(key == *l) return l;

      ++l;
   }
    //se nao  h  o valor procurado o retorno  um iterador do fim do vetor
   return r;
}
```

2.6.2 Busca Binária Iterativa

A busca binária utiliza o paradigma de divisão e conquista, dividindo o arranjo dado em duas partes, sendo l o lado mais esquerdo do arranjo, r o lado mais direito do arranjo e mid o meio do nosso arranjo faremos $mid = r - l/2$ e compara-se se a chave n é igual a mid, se sim retorna-se mid, se não são feitas as comparações $n < mid$ ou $n > mid$ e a busca é performada no lado em que a expressão é verdadeira, alterando nossos delimitadores até o econtro de n ou a chegada ao fim do arranjo. Para seu uso há a necessidade de que o arranjo esteja ordenado.

- Complexidade
  - Melhor caso: $O(1)$
  - Pior caso: $O(\log_2 n)$)

Código em C++ usado para análise:

```
\\Busca Binaria
itr binary(int key, itr l, itr r)
{

   int distance;
   itr middle;

   //o vetor   percorrido at chegar na  partio  de tamanho 1
   while(distance != 0)
```

```
  {
      //calculo do tamanho da particao do vetor
    distance = std::distance(l, r)/2;
    middle = l + distance;

    //retorno ou escolha da direo de pesquisa caso no encontre o valor

    if(*middle == key)
    {
      return middle;
    }
    else if(*middle > key)
    {
      r = --middle;
    }
    else if(*middle < key)
    {
      l = ++middle;
    }

  }

  //retorna iterador com o final do vetor

  return r;
}
```

2.6.3 Busca Ternária Iterativa

A busca ternária é um algoritmo de busca para ser usado em arranjos ordernados, ela o divide em 3 partes iguais delimitadas pelas posições $mid1$ e $mid3$, onde a chave n será buscada. Seguindo uma estratégia parecida com a busca binária com um número de partições maior.

- Complexidade
  - Melhor caso: O(1)
  - Pior caso: $O(\log_3 n)$

Código em C++ usado para análise:

```
\\Busca Ternaria Iterativa
itr ternary(int key, itr l, itr r)
{
   //divisao do vetor em 3 blocos
  auto partition = std::distance(l, r)/3;

  //posicoes da particao central
```

```
  auto t1 = l + partition;
  auto t2 = t1 + partition;

   //o vetor   percorrido at a particao ter somente 1 elemento ou at o
       valor ser encontrado
  while(partition != 0)
  {

     partition = std::distance(l, r)/3;
     t1 = l + partition;
      t2 = t1 + partition;

     //checagem do resultado ou de qual dreo deve ser tomada
     if(*t1 == key)
     {
        return t1;
     }
     else if(*t2 == key)
     {
        return t2;
     }
     else if(*t1 < key)
     {
        l = ++t1;
     }
     else if(*t2 > key)
     {
        r = --t2;
     }
     else
     {
        l = ++t2;
        r = --t1;
     }
  }

  //iterador para ultima posicao do vetor
  return r;
}
```

2.6.4 Busca Fibonacci

A busca Fibonacci também usa a estratégia de divisão e conquista e uso de arranjos ordenados como nas anteriores, porém a divisão é definida pela Sequência de Fibonacci. É procurado um número de fibonaci maior ou igual ao total de elementos do arranjo, o número é decomposto em seus antecessores e a primeira parte do arranjo fica com um total corresponde ao menor número e a segunda com o maior.

- Complexidade
  - Pior caso:$O\log_n$

Código em C++ usado para análise:

```
\\Busca Fibonacci
itr fibonacci(int key, itr l, itr r)
{

    //tamanho da particao
   auto distanceF1 = generateFib(l, r);

    // posio  equivalente ao numero de fibonacci na primeira posicao
   itr itrFib1;

   //o vetor  percorrindo at a  partio  ter tamanho 1
   while(distanceF1 != 0){

       //movimento para particoes menores com base na seq. de fibonacci
      distanceF1 = generateFib(l, r);
      itrFib1 = l + distanceF1;

      //tomada de deciso da direcao da busca ou retorno do resultado
      if(*itrFib1 == key)
      {
         return itrFib1;
      }
      else if(*itrFib1 > key)
      {
         r = --itrFib1;
      }
      else if(*itrFib1 < key)
      {
         l = ++itrFib1;
      }
   }

    // ltima  posio  do vetor retornada ao nao encontrar o valor
   return r;
}
```

2.6.5 Busca Jump Search

A jump search necessita da ordenação do vetor. Os passos para sua execução são feitos em saltos de tamanho iguais definidos por uma constante. No código abaixo foi definido como padrão o salto de 3 itens a cada iteração. Nos bloco do salto é checado se o primeiro valor é corresponde ao item procurado, caso a condição seja verdadeira o índice do elemento é retornado, caso contrário é

checado se o valor da busca é menor, se sim é realizada uma busca sublinear no pedaço do vetor, caso contrário é realizado outro salto.

- Complexidade
  - Melhor caso: O(1)
  - Pior caso: $O(\sqrt{n})$

Código em C++ usado para análise:

```
\\Jump Search

itr jump(int key, itr l, itr r)
{
    //o vetor   percorrido em intervalos de 3 elementos at o seu fim ou
        at a
    //  posio  do elemento pesquisado
   while(l != r)
   {
      //checagem do valor ou retorno da posicao com o valor requerido

      if(*l == key)
      {
         return l;
      }
      else if(key < *l)
      {
          //realizacao da busca linear no intervalo
         return linear(key, l - 1, l+3);
      }
      else if(key > *l)
      {
         l = l+3;
      }
   }

   //retorno do final do vetor em caso de erro
   return r;
}
```

2.6.6 Busca Binária Recursiva

A busca binária recursiva possui a mesma lógica da sua versão iterativa, mas a direção da busca é definida através de chamadas para a própria função.

- Complexidade
  - Melhor caso: O(1)

- Pior caso: $O(\log_2 n)$

Código em C++ usado para análise:

```
\\Busca Binaria Recursiva
itr binary_r(int key, itr l, itr r)
{
    //calculo do elementro central da particao
   int m_distance = std::distance(l, r)/2;
   itr m_val = l + m_distance;

   //checagem da posicao dos iteradores do vetor

   if(l <= r)
   {
      //retorno da  posio  que contem o valor buscado ou chamada para a
          prpria funcao com a direcao a ser tomada pela busca

      if(key == *m_val)
      {
         return m_val;
      }
      else if(key < *m_val)
      {
         return binary_r(key, l, --m_val);
      }
      else if(key > *m_val)
      {
         return binary_r(key, ++m_val, r);
      }
   }

   //retorno do final do vetor caso a no exista o valor da busca no vetor

   return r;
}
```

2.6.7 Busca Ternária Recursiva

A busca ternária recursiva difere da sua versão iterativa devido as chamadas internas de si para para percorrer o vetor.

- Complexidade
  - Melhor caso: $O(1)$
  - Pior caso: $O(log_3 n)$

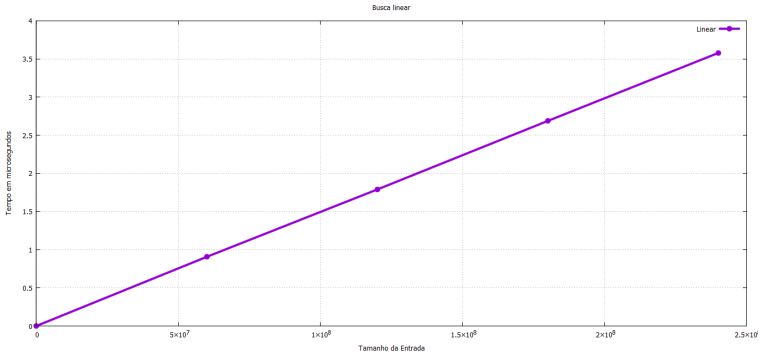
Código em C++ usado para análise:

```
\\Busca Ternaria Recursiva

itr ternary_r(int key, itr l, itr r){

    //segmentacao de 3 partes no vetor
   auto partition = std::distance(l, r)/3;

   auto t1 = l + partition;
   auto t2 = t1 + partition;

    //checaagem da posicao dos iteradores
   if(l <= r)
   {
       //retorno do resultado ou chamada da propria funcao com a direcao
           a ser percorrida
      if(*t1 == key)
      {
         return t1;
      }
      else if(*t2 == key)
      {
         return t2;
      }
      else if(*t1 < key)
      {
         return ternary_r(key, ++t1, r);
      }
      else if(*t2 > key)
      {
         return ternary_r(key, l, --t2);
      }
      else
      {
         return ternary_r(key, ++t2, --t1);
      }
   }

     //retorna iterador para o fim do vetor caso o valor no seja
         encontrado
   return r;
}
```

3 Resultados

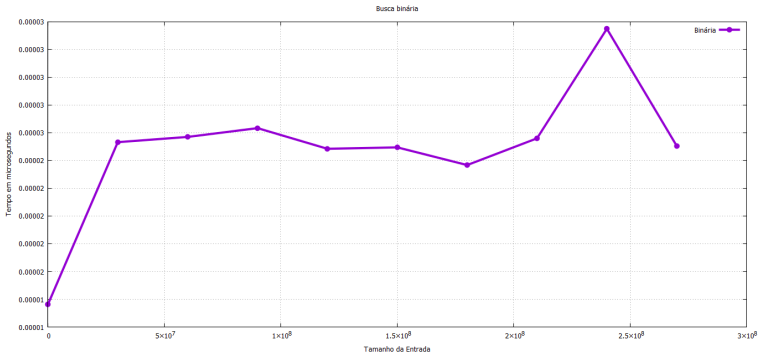
3.1 Performance Individual

3.1.1 Busca Linear



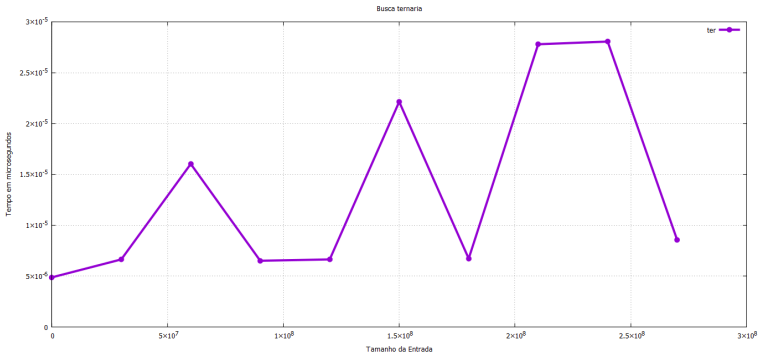
O gráfico correspondente a busca linear explicita a complexidade da função $O(n)$ em uma reta crescente. A reta permite a visualização da ineficiência com relação a grandes entradas de dados. Na realização dos testes foi a função com maior latência.

3.1.2 Busca Binária Iterativa



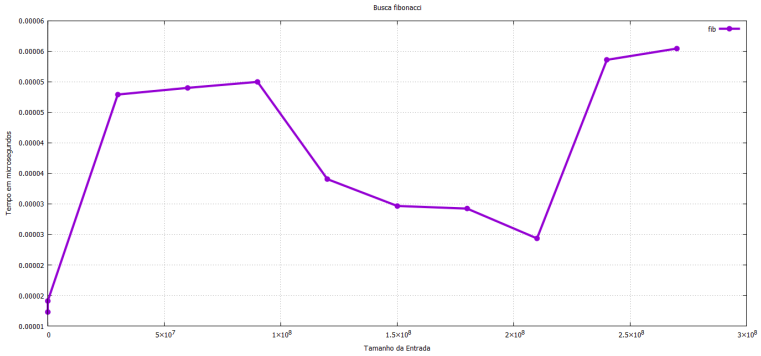
O gráfico acima tenta aproximar-se da complexidade da busca binária, ou seja, $O(logN)$. O gráfico tem uma série de variações para uma quantidade maior de amostras maiores testadas em relação ao tempo de execução, mas no caso acima sua variação está bem suave e mais consistente. Seu bom desempenho é notório em tese e na prática comparado aos outros algortimos.

3.1.3 Busca Ternária Iterativa



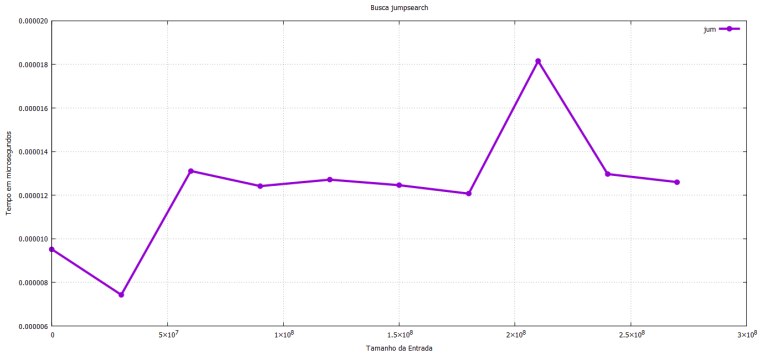
A busca ternária também é uma busca que envolve partição do arranjo, no caso em 3 segmentos. O seu gráfico possui mais oscilações em relação a outras buscas que utilizam o mesmo recurso.

3.1.4 Busca Fibonacci Iterativa



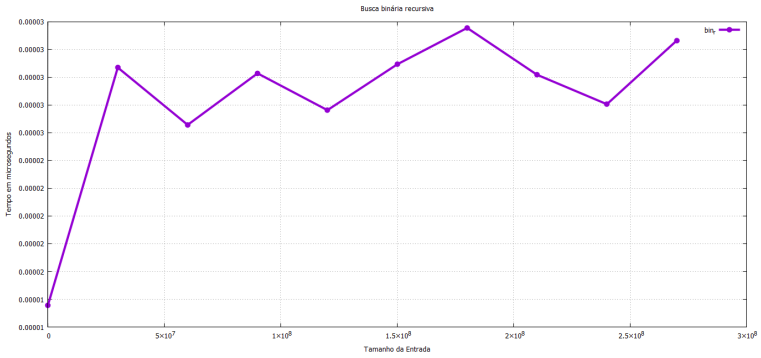
A busca Fibonnaci, por se tratar também de uma busca com segmentação (com subarranjos de tamanhos variáveis) do alvo da busca, registrou resultados que remetem um pouco aos da busca binária em termos de formato.

3.1.5 Jump Search



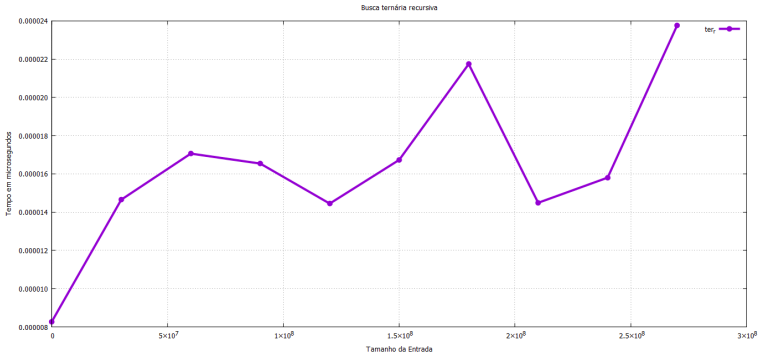
A jump search possui intervalos mais constantes devido ao percuso de busca ser definido pela constante k.

3.1.6 Busca Binária Recursiva



A busca binária recursiva apresentou resultados satisfatórios.

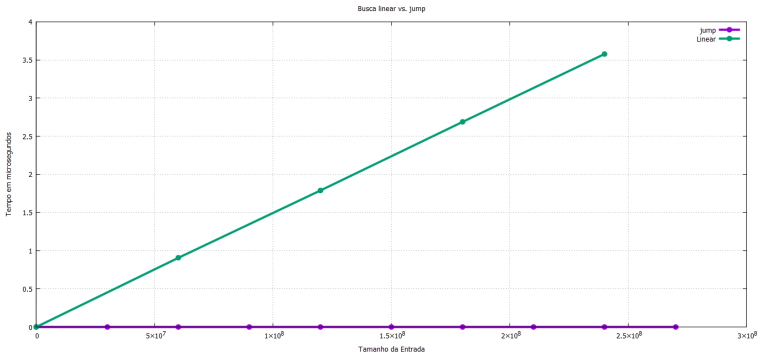
3.1.7 Busca Ternária Recursiva



O comportamento do gráfico está um pouco mais difuso em comparação a outras funções.

3.2 Algoritmos Lineares

"Considerando os dois algoritmos lineares, busca linear e jump search, qual é o mais eficiente no pior caso?"

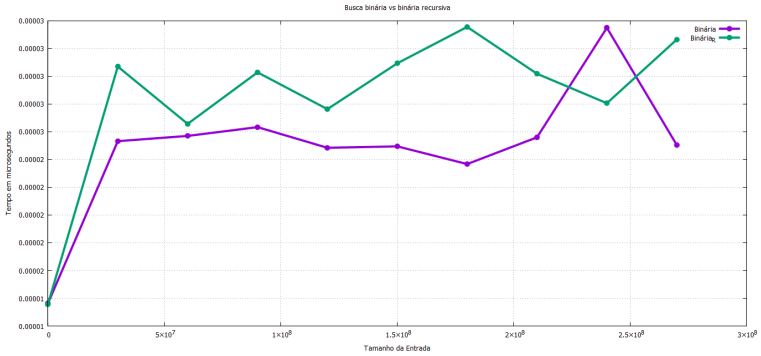


A jump search otimiza bastante o tempo de pesquisa com os saltos, no comparativo o resultado é facilmente observável.

A saída acima está semelhante ao comparativo com a busca binária, devido ao fato de que assintoticamente o valor da complexidade da jump search está entre o $O(n)$ e o $(\log_n)$.

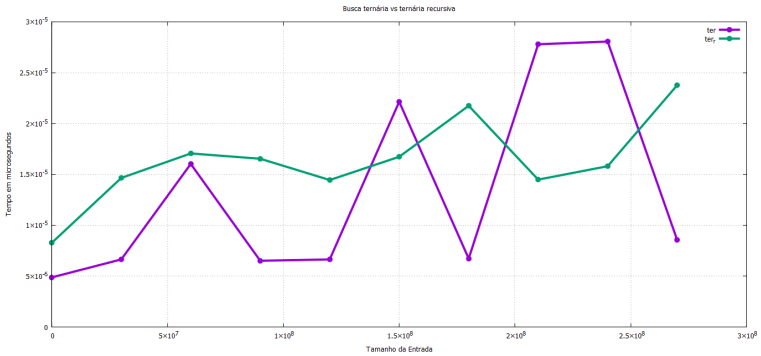
3.3 Recursão x Iteração

"Qual tipo de estratégia de implementação é mais eficiente, recursão ou iteração?"



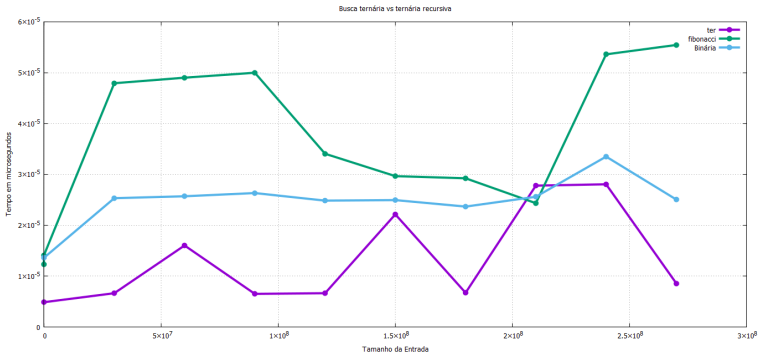
As linhas que representam cada função demostram a vantagem do uso da versão iterativa do algoritimo. A explicação para o resultado reside na quantidade de recursos alocados para uma função recursiva. Como já é conhecido uma chamada para uma função recursiva deixa os processos anteriores em aberto até que se chegue ao caso de parada.

É importante ressaltar que o uso de algoritmos recursivos são importantes para a visualização e simplificação do problema para o programador, mas do ponto de vista prático em um cenário, como desse trabalho, a performance obtida não é satisfatória.



Também foram comparadas as versões das buscas ternárias, apesar de alguns pontos de interseção e em que a versão iterativa supera a recursiva os comentário continuam válidos e podem ser comprovados com a avaliação do trecho final do gráfico.

3.4 Tamanho da Partição e sua Influência no Desempenho



Algortimos que dividem o objeto de pesquisa são claramente mais vantajosos por cortar boa parte da repetição de passos denecessários. De início, a primeira hipótese que pode ser formulada é que quanto mais subdivisões na coleção de itens utilizada maior será a vantagem. O gráfico mostra o resultado da comparação de três das buscas: ternária, Fibonacci e binária.

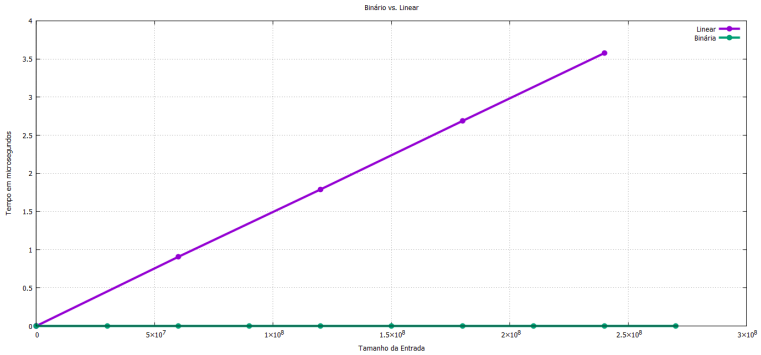
Cada função tem um comportamento particular, comparando os padrões temos uma função que separa itens em tamahos variavéis e duas que separam tamanhos predefinidos.

O resultado das respectivas amostras indicam que segmentar o vetor em mais partes é a melhor tomada de decisão para economia de tempo, já que a busca ternária marcou os menores valores de tempo de execução.

Entretanto, em uma análise mais profunda da complexidade dos algoritmos

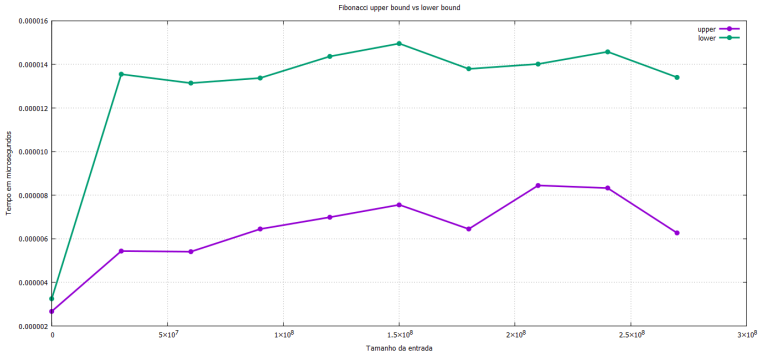
de busca binária e ternária, vemos que eles possuem a complexidade para o pior caso de $O(2 * \log_2 n)$ e $O(4 * \log_3 n)$, respectivamente. Ao comparar as duas constantes constata-se que a busca ternária é menos efeciente, pois sua complexidade pode ser decomposta em $O(2/\log_2 3 * log_2 n)$ que é maior que o valor correspondente a busca binária.

3.5 Algoritmos de Classes de Complexidade Diferentes



No gráfico temos a comparação de um algortimo $O(n)$ e um $O(\log n)$. A diferença dos algortimos é muito clara desde o início, todavia os valores só serão significativos para grandes valores de dados. Nos testes a busca binária obteve pouca variação de seus valores temporais.

3.6 O pior caso da Busca Fibonacci



Os dois cenários comparados acima representam os casos em que o valor não existe no vetor, mas existe um elemento de valor aproximado em suas extremidades (lower bound à esquerda e upper bound à direita).

O pior caso, a upper bound, pode ser explicado pela segmetação de duas partes de tamanhos distintos. A segunda parte do vetor sempre possuirá partição maior o que acarreta em mais subdivsões desse pedaço para continuação da busca até a chegar-se ao último elemento.

4 Conclusões

A análise empirica dos algoritmos supracitados permitiu a vizualização direta dos custos de operações de algoritmos relativamente simples. Pensar no pior caso de execução de código é crucial para realização de qualquer trabalho, tendo em vista as limitações físicas e temporais.

Dentre todos os algortimos comparados os que obtiveram melhores resultado possuem uma complexidade $O(\log n)$ ou algo próximo a isso. A busca binária se destacou em todos os cenários e sua análise matemática lhe garante ser a melhor escolha nos cenários testados.

É preciso esclarecer que os resultados não são tão precisos devido as limitações de hardware, durante a execuçãao dos códigos houveram oscilações nos valores registrados e a limitação do total de memória alocada é um fator que restringe a amostragem dos dados.

Uma análise empírica é um processo demorado para ser aplicado em situações cotidianas, em todos os casos a melhor saída é realizar uma análise assintótica. Ter a complexidade de um algortimo dá aos programadores o poder de previsão do comportamento do código permitindo a realização de um trabalho eficaz.

5 Referências Bibliográficas

Artigos:

"Why is Binary Search preferred over Ternary Search", disponível em: https://www.geeksforgeeks.org/binary search-preferred-ternary-search

"Divide-and-conquer algorithm" , disponível em: https://en.wikipedia.org/wiki/Divide-and-conquer$_algorith$