**UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE**
**INSTITUTO METRÓPOLE DIGITAL**
**CURSO DE BACHARELADO EM TECNOLOGIA DA INFORMAÇÃO**

**Carlos Vinicius Fernandes Rodrigues**
**João Victor Bezerra Barboza**

# RELATÓRIO - ANÁLISE EMPÍRICA

**NATAL / RN**
**201**

CARLOS VINICIUS FERNANDES RODRIGUES
JOÃO VICTOR BEZERRA BARBOZA

# RELATÓRIO - ANÁLISE EMPÍRICA

Relatório apresentado à disciplina Estrutura de Dados Básicas I, ministrada por Dr. Selan Rodrigues dos Santos da Universidade Federal do Rio Grande do Norte, como requisito parcial para obtenção de notas.

**NATAL / RN**
**2016**

**SUMÁRIO**

## 1. INTRODUÇÃO

O relatório a seguir possui o intuito de analisar e escolher um algoritmo ideal para realizar buscas em um arranjo através de análise empírica. Há também o intuito de analisar e constatar a veracidade das complexidades dos algoritmos vistos em teoria anteriormente, vendo-os na prática.

Durante este relatório, serão vistos determinados cenários para um vetor, em que será utilizado para algumas opções de algoritmos. Tais algoritmos são: busca sequencial iterativa, busca sequencial recursiva, busca binária iterativa, busca binária recursiva, busca ternária iterativa, busca ternária recursiva, busca sequencial padrão (*std::search*) e busca binária padrão (*std::bsearch*). Os cenários são: um elemento *k* a ser procurado não está em um conjunto de valores; um elemento *k* está a ¾ de distância do início de um conjunto de valores; e a terceira ocorrência de um elemento *k* deverá ser encontrada no conjunto de valores.

Ao fim, será explanado como poderá ser selecionada a melhor forma de busca para cada caso a partir dos dados recolhidos, sempre visando o entendimento e a replicação do que foi feito para a realização deste trabalho.

## 2. MATERIAIS E MÉTODOS

### 2.1. O problema computacional

O problema computacional utilizado para criação dos algoritmos pode ser descrito da seguinte forma:

> *Dado um conjunto de valores previamente armazenados em um arranjo A, nas posições A[l], A[l+1],..., A[r], sendo 0 <= l <= r E N º, verificar se um valor chave k está entre este conjunto de valores. Em caso positivo, indicar qual o índice da localização de k em A. Caso contrário, retornar -1.*

Para resolver o problema, foi escolhido oito algoritmos, com complexidades temporais variadas, e analisar os resultados em diferentes cenários, são eles: o valor chave *k* não está no conjunto de valores, ou seja, o algoritmo retornará -1; o valor chave *k* está a ¾ de distância do início do conjunto de valores; e o algoritmo deve retornar, se existir, a posição da terceira ocorrência da chave *k* no conjunto de valores.

### 2.2. Algoritmos

A linguagem de programação adotada para escrita dos algoritmos foi a C++. Os algoritmos escolhidos para fazer a análise foram: busca sequencial iterativa, busca sequencial recursiva, busca binária iterativa, busca binária recursiva, busca ternária iterativa, busca ternária recursiva, busca sequencial padrão (std::search) e busca binária padrão (std::bsearch). Para alocar o conjunto de valores do problema, foi escolhido o *std::vector*, pela praticidade em relação às outras práticas conhecidas anteriormente.

#### 2.2.1. Busca Sequencial

A busca sequencial (também conhecida como busca linear) é um algoritmo que busca uma chave *k* em um conjunto de valores sequencialmente, isto é, procurando pelos elementos, um a um, do início até o fim.

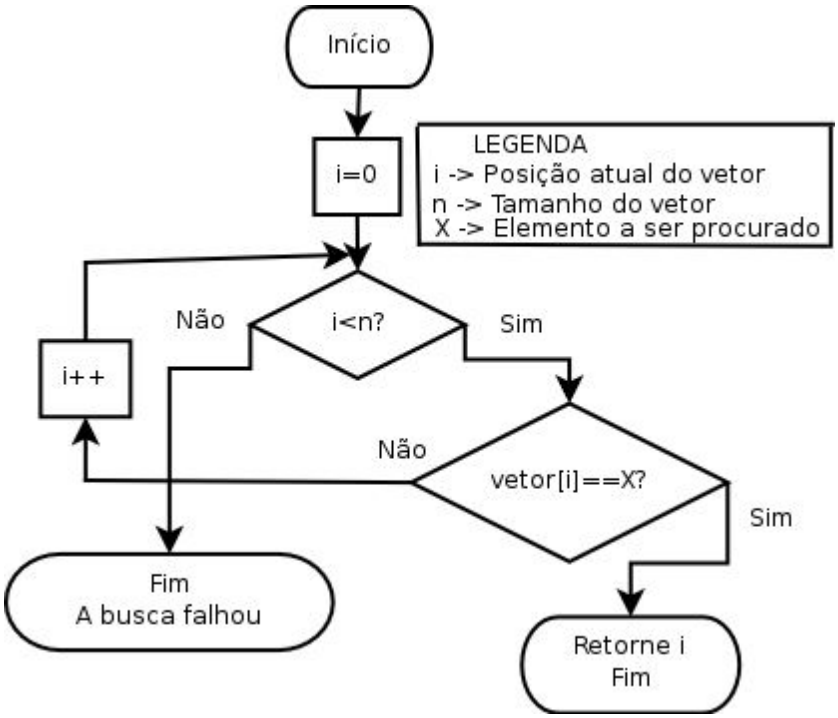



Figura 1 - Fluxograma da busca sequencial. Fonte: Wikipédia[1].

O algoritmo da busca sequencial, na versão iterativa, então, foi escrito da seguinte forma:

```
long int BuscaSequencialIterativa( std::vector<long int> & A, long int k, long int left, long int right, int ocorrencia ){
	for ( auto i = left ; i < right ; i ++ ){
		if ( k == A[i]){
		if ( ocorrencia==1 || ocorrencia == 0 ) return i;
		ocorrencia--;
		}
	}
	return -1;
}
```

É possível perceber algumas mudanças no algoritmo devido a algumas exigências dos cenários de implementação, como no caso em que deve-se retornar a terceira ocorrência da chave *k*. Todos os códigos contam com essas mudanças.

O algoritmo da busca sequencial, na versão recursiva, foi escrito da seguinte forma:

```
long int BuscaSequencialRecursiva( std::vector<long int> & A, long int k, long int left, long int right, int ocorrencia ){
	if ( left > right ) return -1;
	else{
		if( k == A[left] && (ocorrencia == 1 || ocorrencia == 0) ) return left;
```

[1] Disponível em <https://pt.wikipedia.org/wiki/Busca_linear>. Acesso em mar. 2016.

```
                else if ( k==A[left] ) return BuscaSequencialRecursiva( A, k, left+1, right,
ocorrencia-1 );
                else return BuscaSequencialRecursiva( A, k, left+1, right, ocorrencia );
        }
}
```

Há também o algoritmo de busca binária padrão (std::search). O algoritmo utilizado pode ser visto a seguir:

```
long int SearchWrapper( std::vector <long int> &A, long int key, long int left, long int
right, int ocorrencia ){
        long int needle[1] = { key };
        std::vector< long int >::iterator it;
        it = std::search ( A.begin(), A.begin()+right, needle, needle+1 );
        if ( it!=A.begin()+right && ocorrencia < 2 )
                return (it-A.begin());
        else if ( it!=A.begin()+right && ocorrencia > 1 ){
                while( ocorrencia > 1 ){
                ocorrencia--;
                it = std::search( it-left+1, A.begin()+right, needle, needle+1 );
        }
        if ( it != A.begin()+right ) return ( it-A.begin() );
        else return -1;
        }
        else return -1;
}
```

### 2.2.2. Busca Binária

A busca binária é um algoritmo para busca de uma chave *k* em um conjunto de valores que utiliza o paradigma da divisão e conquista. Tendo a certeza de que o conjunto de valores utilizado seja ordenado, ele é dividido em duas partes, comparando a chave com o elemento do meio conjunto. Se o elemento do meio for igual à chave, é retornada a posição do elemento. Se for maior, é feita uma nova busca com os elementos à sua esquerda. Se for menor, a nova busca é feita com os elementos à direita. Assim é feito sucessivamente, até que k seja achado ou que não haja mais divisões possíveis.

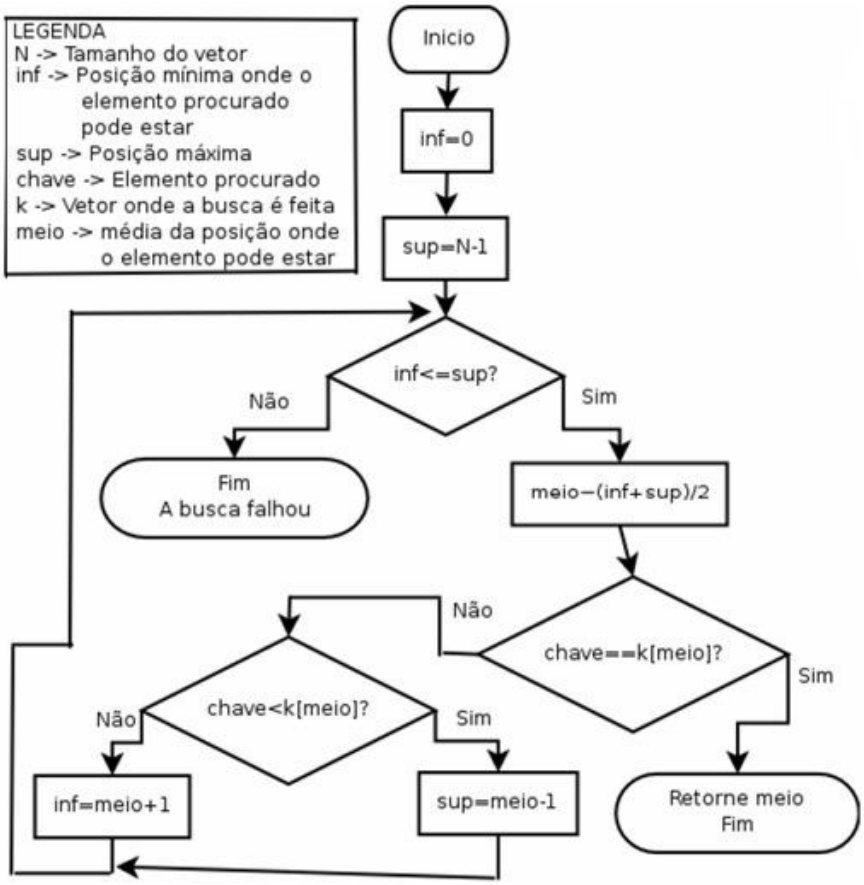



Figura 2 - Fluxograma da busca binária. Fonte: Joseane Alves Freire[2].

O algoritmo da busca binária, na versão iterativa, foi escrito da seguinte forma:

```
long int BuscaBinarialterativa( std::vector<long int> & A, long int k, long int left, long int right, int ocorrencia ){
        long int middle;
        while ( right >= left ){
        middle = ( left + right ) / 2;
        if ( k == A[middle] ){
                if ( ocorrencia >= 1 ){
                        auto i(1);
                        if( A[middle-1] == k ){
                                while ( A[middle-i] == k ){
                                        i++;
                                }
                                if( A[middle+ocorrencia-i] == k )
                                        return middle-i+ocorrencia;
                                else return -1;
                        }
                        else return middle;
                }
                return middle;
        }
        else if ( k < A[middle] ) right = middle-1;
```

[2] Disponível em <http://slideplayer.com.br/slide/89041/>. Acesso em mar. 2016.

```
        else left = middle + 1;
        }
        return -1;
}
```

O algoritmo da busca binária, na versão recursiva, pode ser visto a seguir:

```
long int BuscaBinariaRecursiva ( std::vector<long int> & A, long int k, long int left, long int right, int ocorrencia ){
        long int middle = 0;
        if ( left > right ) return -1;
        else{
        middle = ( left + right ) / 2;
        if ( k == A[middle] ){
                if ( ocorrencia >= 1 ){
                        auto i(1);
                        if( A[middle-1] == k ){
                                while ( A[middle-i] == k ){
                                        i++;
                                }
                                if( A[middle+ocorrencia-i] == k )
                                        return middle-i+ocorrencia;
                                else return -1;
                        }
                        else return middle;
                }
                return middle;
        }
        else if ( k < A[middle] ) return BuscaBinariaRecursiva( A, k, left, middle-1, ocorrencia );
        else return BuscaBinariaRecursiva( A, k, middle+1, right, ocorrencia );
        }
}
```

E o algoritmo da busca binária padrão (std::bsearch) foi escrito da seguinte forma:

```
long int BSearchWrapper( std::vector <long int> &A, long int key, long int left, long int right, int ocorrencia ){
   long int *p1 = ( long int * ) std::bsearch( &key, A.data(), right-left, sizeof(A[0]), compare );
   if( p1 == NULL )
      return -1;
   else if ( ocorrencia==3 ){
        auto i(1);
        if( A[(p1 - &A[0])] == key ){
           while ( A[(p1 - &A[i])] == key ){
              i++;
           }
        }
        if( A[(p1 - &A[0])+ocorrencia-i] == key )
           return ( p1 - &A[0] )-i+ocorrencia;
        else return -1;
     }
   else return ( p1 - &A[0] );
}
```

### 2.2.3. Busca ternária

A busca ternária é um algoritmo para busca de uma chave *k* em um conjunto de valores que, assim como a busca binária, utiliza o paradigma da divisão e conquista. Ele também parte da certeza de que esse conjunto é ordenado, mas conta com uma diferença: o conjunto é dividido em três partes. Logo, haverão três partes e dois "meios". Se a chave *k* for menor que o primeiro "meio", a busca parte para a divisão mais à esquerda. Se for maior que o segundo "meio", parte para a divisão mais à direita. Se for maior que o primeiro e menor que o segundo "meio", parte para a divisão central do conjunto. O processo é feito sucessivamente para o novo conjunto até encontrar a chave *k* ou não haver mais novos conjuntos para dividir.

O algoritmo da busca ternária, na versão iterativa, foi escrito da seguinte forma:

```
long int BuscaTernarialterativa( std::vector<long int> & A, long int k, long int left, long int right, int ocorrencia ){
	long int middle1, middle2;
	while ( left <= right ){
	middle1 = ( 2*left + right ) / 3;
	middle2 =  ( left + 2*right ) / 3;
	if ( k == A[middle1] ){
		if ( ocorrencia >= 1 ){
			auto i(1);
			if( A[middle1-1] == k ){
				while ( A[middle1-i] == k ){
					i++;
				}
				if( A[middle1+ocorrencia-i] == k )
					return middle1-i+ocorrencia;
				else return -1;
			}
			else return middle1;
		}
		return middle1;
	}
	if ( k == A[middle2] ){
		if ( ocorrencia >= 1 ){
			auto i(1);
			if( A[middle2-1] == k ){
				while ( A[middle2-i] == k ){
					i++;
				}
				if( A[middle2+ocorrencia-i] == k )
					return middle2-i+ocorrencia;
				else return -1;
			}
			else return middle2;
		}
```

```
            return middle2;
        }
        if ( k < A[middle1] ) right = middle1 - 1;
        if ( k > A[middle1] && k < A[middle2] ) left = middle1 + 1, right = middle2 - 1;
        if ( k > A[middle2] ) left = middle2 + 1;
        }
        return -1;
}
```

E o algoritmo da busca ternária, na versão recursiva:

```
long int BuscaTernariaRecursiva( std::vector<long int> & A, long int k, long int left, long int right, int ocorrencia ){
        long int middle1, middle2;
        if ( left > right ) return -1;
        else{
        middle1 = (2*left + right)/3;
        middle2 = (left + 2*right)/3;
        if( k == A[middle1] ){
                if ( ocorrencia >= 1 ){
                        auto i(1);
                        if( A[middle1-1] == k ){
                                while ( A[middle1-i] == k ){
                                        i++;
                                }
                                if( A[middle1+ocorrencia-i] == k )
                                        return middle1-i+ocorrencia;
                                else return -1;
                        }
                        else return middle1;
                }
                return middle1;
        }
        else if( k == A[middle2] ){
                if (ocorrencia >= 1){
                        auto i(1);
                        if( A[middle2-1] == k ){
                                while ( A[middle2-i] == k ){
                                        i++;
                                }
                                if( A[middle2+ocorrencia-i] == k )
                                        return middle2-i+ocorrencia;
                                else return -1;
                        }
                        else return middle2;
                }
                return middle2;
        }
        else if ( k < A[middle1] ) return BuscaTernariaRecursiva( A, k, left, middle1 - 1, ocorrencia );
        else if ( k > A[middle1] && k < A[middle2] ) return BuscaTernariaRecursiva( A, k, middle1 + 1, middle2 - 1, ocorrencia );
        else return BuscaTernariaRecursiva( A, k, middle2 + 1, right, ocorrencia );
        }
}
```

### 2.2.4. O código *main*

Para tornar os testes mais diretos e evitar que se fique alterando os dados no código, foi criada uma forma de fazer os testes de acordo com requisitos pedidos no terminal durante a execução do programa. Um exemplo disso é a possibilidade de escolha da situação a ser testada, que pode ser visto no fragmento de código abaixo:

```
while ( situacao < 0 || situacao > 2 )
{
   std::cout << "TABELA:\n";
   std::cout << "0 : K NÃO ESTÁ NO VETOR\n";
   std::cout << "1 : K ESTÁ A 3/4 DO INÍCIO DO VETOR\n";
   std::cout << "2 : O TERCEIRO K, SE EXISTIR, É PROCURADO\n\n";
   std::cout << "Digite a situação do teste: ";
   std::cin >> situacao;
}
std::cout << "\n";
```

O mesmo método foi utilizado para escolher o algoritmo a ser testado.

O código *main* também faz a medição do tempo e varia o tamanho do vetor de $2^2$ a $2^{27}$. O trecho do código que realiza essa operação ser visto abaixo.

```
for ( auto i( 4u ); i <= arrSz ; i *= 2){
        [...]
        while (testes < 100){
                [Os 100 testes são realizados aqui.]
        }
        [...]
}
```

### 2.2.5. Outras funções utilizadas

Também foram utilizadas nos experimentos algoritmos auxiliares. Uma função para preencher o vetor, uma para comparar valores e outra para fazer a média dos valores. São elas:

```
int compare (const void * a, const void * b)
{
  return ( *(int*)a - *(int*)b );
}

float media_instantanea( std::vector <float> valores, int m)
{
   int k = 1;
   float media = 0;
   for ( auto i(0) ; m != k ; i++ ){
      media = media + (valores[i]-media)/k;
      k++;
   }
   return media;
```

```
}

void randomFill( std::vector<long int> &V, const long int lower, const long int upper,
const unsigned int seed) {
   std::default_random_engine eng(seed);
   std::uniform_real_distribution<long double> distr(lower, upper);
   for( auto &elem : V){
      elem = distr(eng);
   }
}
```

### 2.3. Dados relacionados aos experimentos

Para a realização dos testes, foi utilizado um computador com as seguintes configurações: i5-3337U 1.8GHz (processador), 3922 MB (RAM), VAIO (marca), R0230DA (versão da BIOS), Ubuntu 14.04.4 LTS (sistema operacional), Sublime build 3103 (editor de códigos), GNU C 4.8.4 (compilador), GNOME 3.6.2 (terminal), GNUplot 4.6 (gerador de gráfico).

O maior valor de elementos que o computador pôde alocar em um vetor (std::vector) de inteiros longos (long int) foi $2^{27}$, ou seja, 134217728 elementos. Logo, faixa de valores do tamanho do vetor escolhida para testar é de $2^x$ elementos, com 2 <= x <= 27. Para preencher o vetor, foram utilizados valores pseudoaleatórios de -134217728 a 134217728. A *seed* da função *randomFill* escolhida foi 1. Durante os experimentos, apenas o terminal, o Sublime e o gerenciador de arquivos estavam abertos.

### 2.4. Geração dos gráficos

Para gerar os gráficos, primeiro foi gerado, através do código *main*, um arquivo com os dados no formato abaixo.

```
4 4.17172e-05
8 0.000202333
16 0.000149677
32 0.00035102
[...]
16777216 0.0011654
33554432 0.000206444
67108864 0.000212545
134217728 0.0012911
```

O programa utilizado para a geração dos gráficos foi o GNUplot.

### 2.5. Compilação e execução

Ao abrir a pasta do projeto, deve-se digitar, para compilar:

```
$ g++ -Wall -std=c++11 src/buscas_drive.cpp -I include/ -o bin/buscas_drive
```

E, para executar:

```
$ ./bin/buscas_drive
```

## 3. Gráficos

Legenda para os gráficos: -o = vetor ordenado.

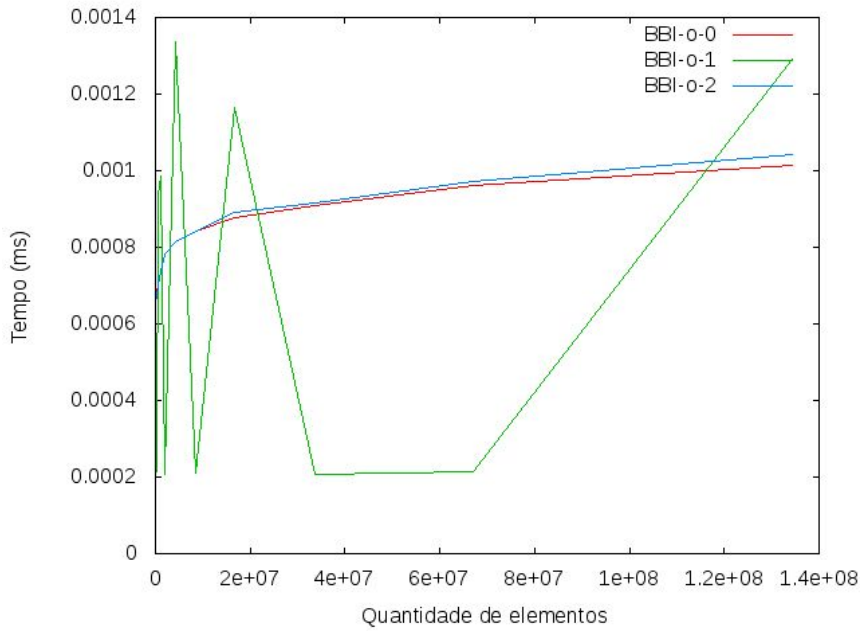
-d = vetor desordenado.

-0 = valor da chave *k* não está no conjunto de valores.

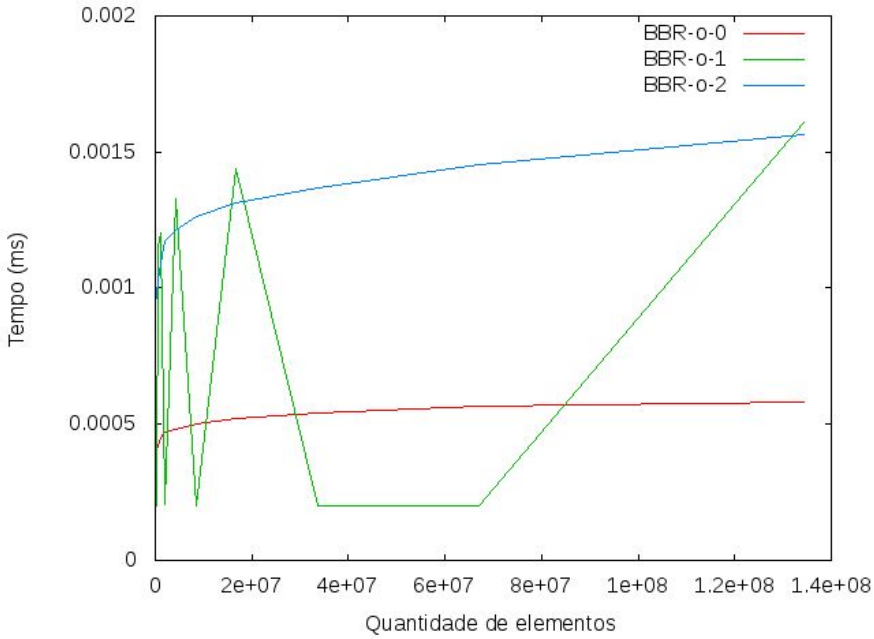
-1 = valor da chave *k* está a ¾ de distância do início do conjunto de valores.

-2 = o algoritmo deve retornar, se existir, a posição da terceira ocorrência da chave *k* no conjunto de valores.

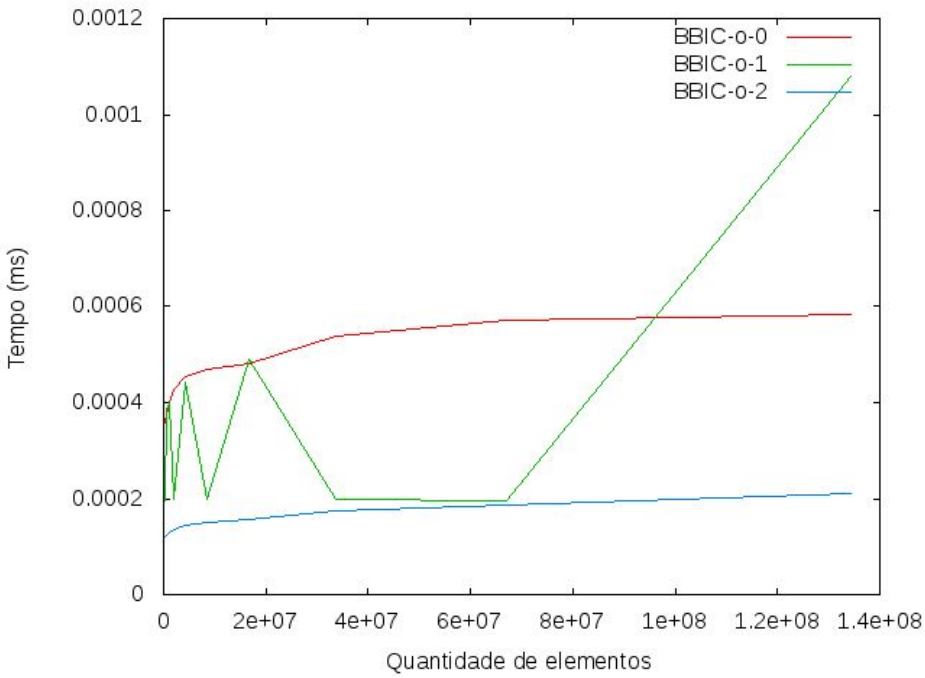
Busca Binária Iterativa:



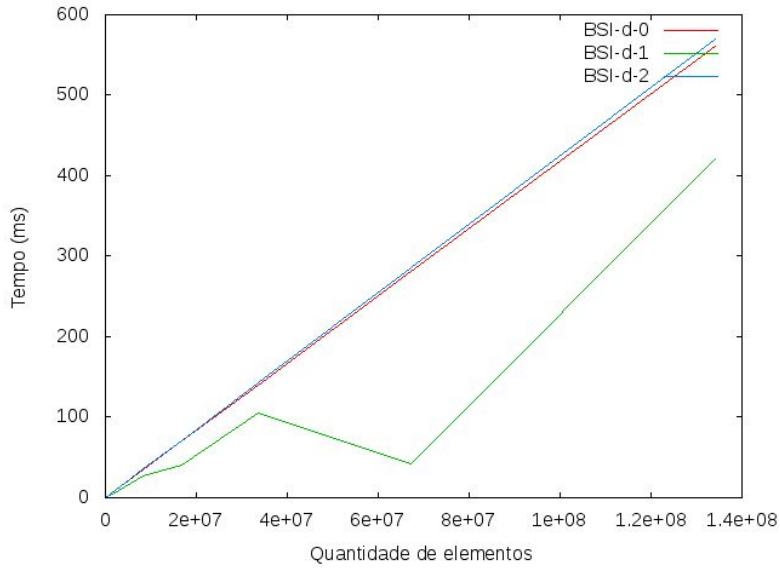
Busca Binária Recursiva:

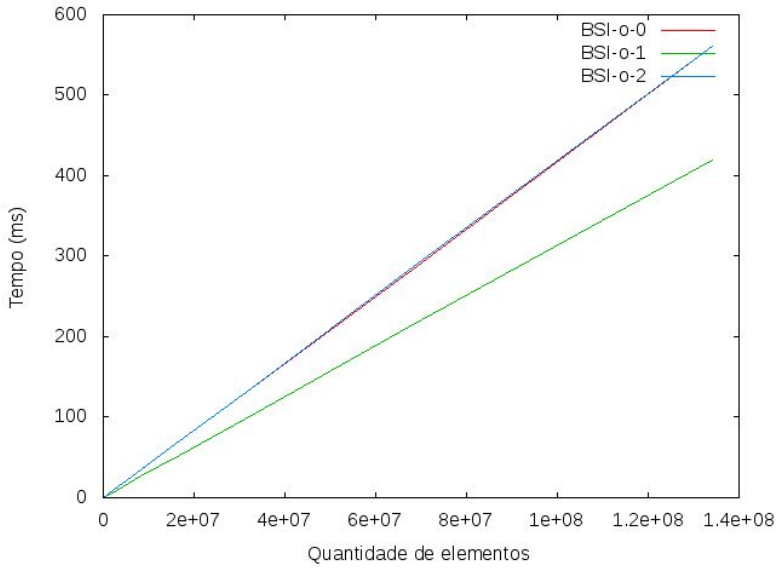



Busca Binária Padrão:

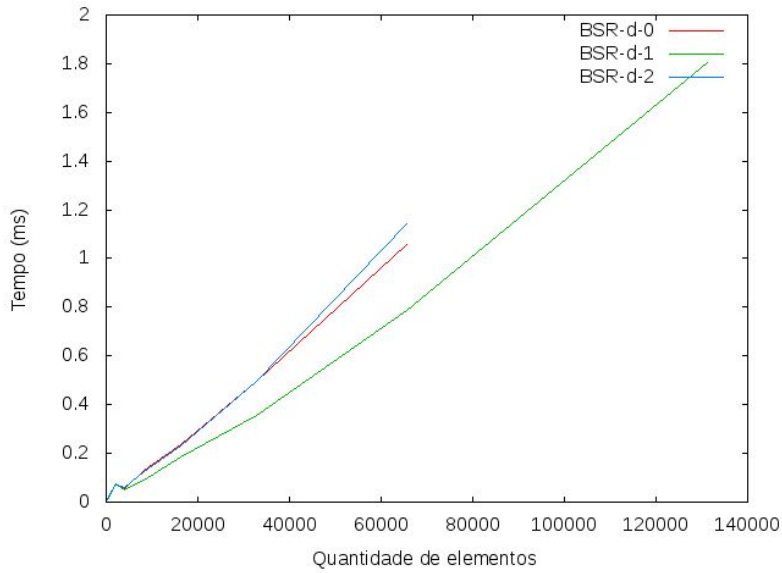


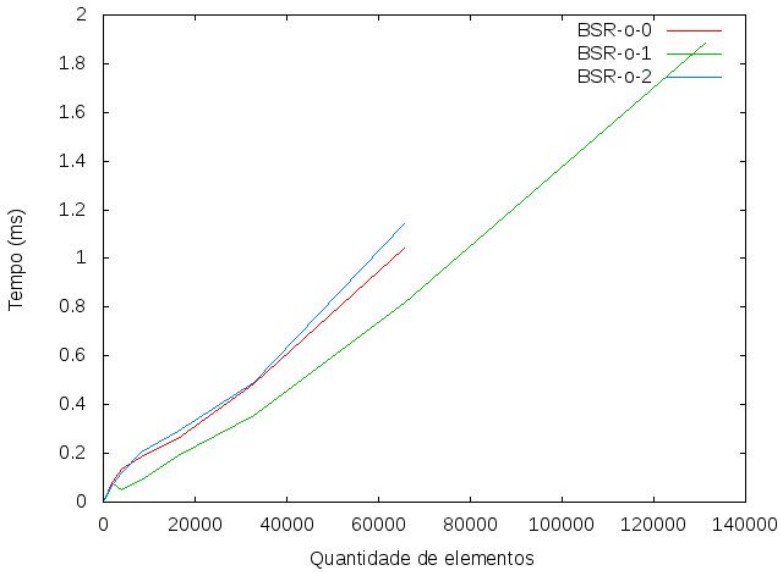
Busca Sequencial Iterativa (vetor desordenado / ordenado):



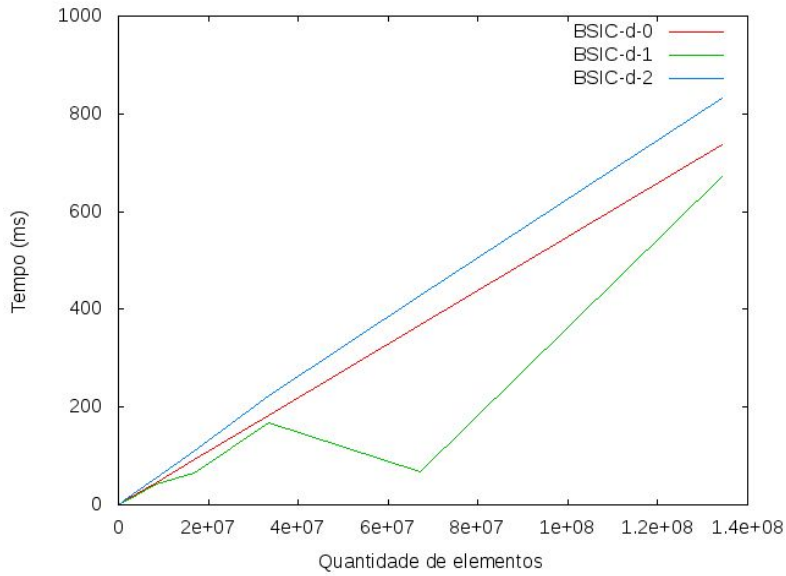



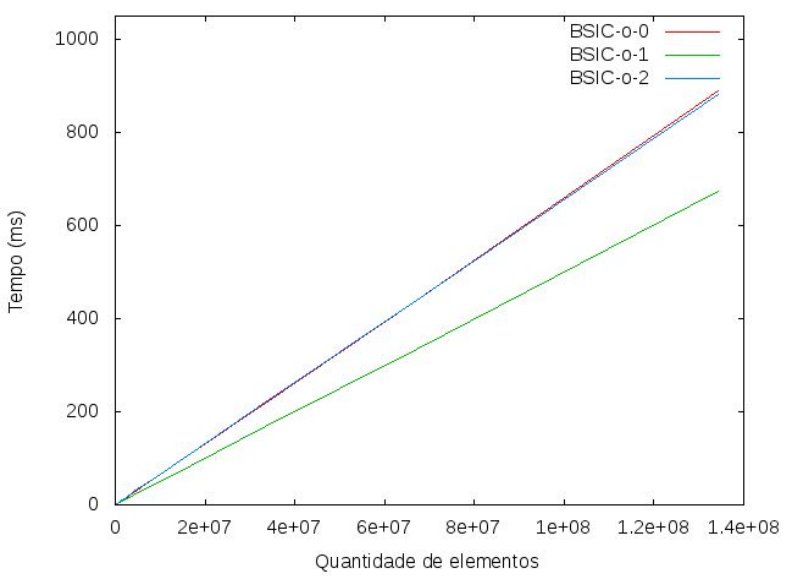

Busca Sequencial Recursiva (vetor desordenado / ordenado):



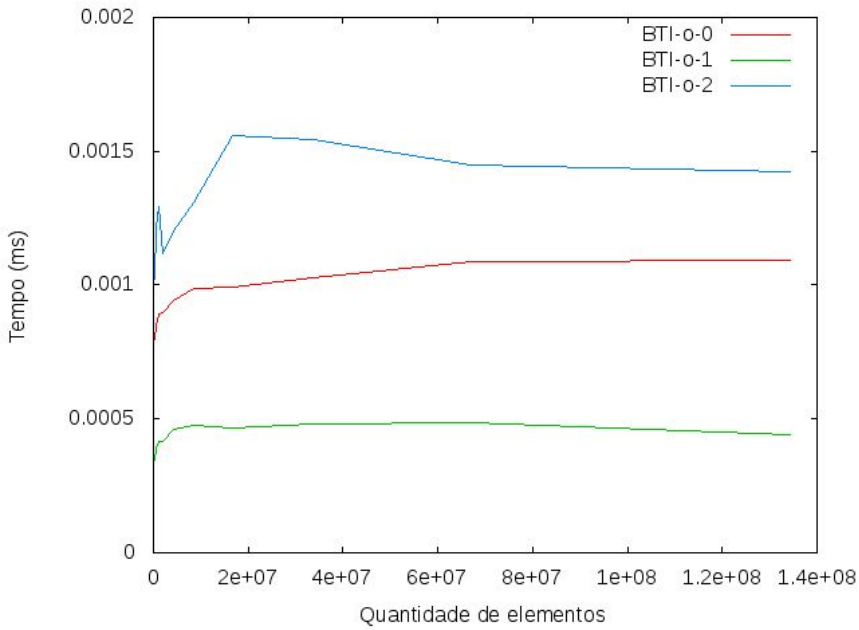



Busca Sequencial Padrão (vetor desordenado / ordenado):

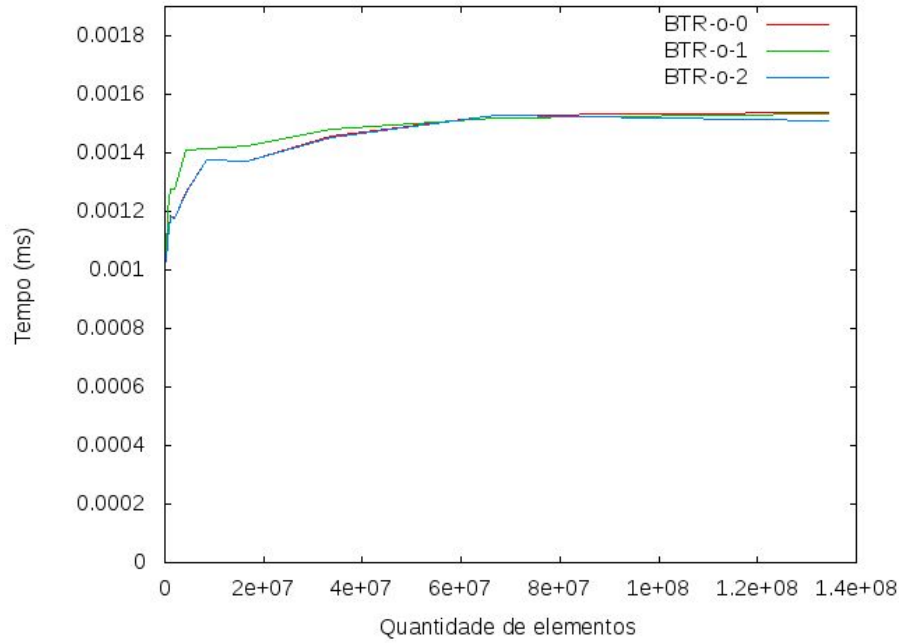



Busca Ternária Iterativa:



Busca Ternária Recursiva:



## 4. LEVANTAMENTO E ANÁLISE DE DADOS

De maneira geral, foi descoberto que não há algoritmos melhores que o outros. Há algoritmos que, em determinadas situações, são mais eficientes.

Para um cenário em que há um vetor desordenado e não muito grande, a melhor opção é a busca sequencial. Em um cenário onde o vetor não é muito grande, as buscas recursivas também podem ser uma opção. Entre a busca ternária e a binária, a melhor opção é a binária, pois a diferença entre as duas não é tão grande e há mais dificuldade na implementação da ternária.

Durante as implementações, alguns valores inesperados ocorreram, todas elas durante os testes das buscas quando a chave *k* está a ¾ de distância do início do vetor. Uma possível explicação para isso é que, como o tamanho do vetor é muito maior que a faixa de valores que vai preencher os elementos, há muitas repetições do mesmo valor, e pode ter ocorrido de um desses valores terem sido encontrados antes de chegar a ¾, finalizando a busca antes do tempo real necessário para chegar à posição.

As funções matemáticas que descrevem as funções geradas são: T(n) = n - sequencial; T(n) = log2(n) - binária; T(n) = log3(n) - ternária. Logo, é possível estimar o tempo de execução para qualquer valor (desde que esteja dentro do domínio da função).

Ao final das análises, é possível concluir que a análise empírica é compatível com a análise matemática da complexidade temporal.

## 5. ANEXOS

### Anexo I - Código main completo

```
#include <iostream>
#include <algorithm>
#include <fstream>
#include <sstream>
#include <random>
#include <vector>
#include <chrono>
#include <cstdlib>

#include "other_functions.cpp"
#include "buscas.cpp"

#define N 134217728

int main( int argc, char * argv[] )
{
   auto teste(-1), ocorrencia(0), situacao(-1), key(0), ordenado(0), keyposition(0), fim(1);
   std::string tst="valores.txt";
   std::ofstream outfile;
   std::vector< float > resultados_buscas( 100 );

   long int (*pfunc[])( std::vector<long int> &, long int, long int, long int, int) =
   {
      BuscaSequencialIterativa, BuscaSequencialRecursiva, BuscaBinariaIterativa,
      BuscaBinariaRecursiva, BuscaTernariaIterativa, BuscaTernariaRecursiva,
      SearchWrapper, BSearchWrapper
   };

   auto arrSz( 0ul );

   if ( argc > 1 )
   {
      std::stringstream( argv[1] ) >> arrSz;
   }
   else
   {
      arrSz = N;
   }

   std::cout << ">>> Required array size is: " << arrSz << std::endl;

   std::cout << ">>> Alocando vetor.\n";
   std::vector< long int > V( arrSz ); // 2^27 = 134217728
   std::cout << ">>> Vetor alocado.\n";

   std::random_device r;

   std::cout << ">>> Preenchendo vetor...\n";
   randomFill( V, -134217728, 134217728, 1 );

   std::cout << "\nVetor ordenado? (1/0) ";
   std::cin >> ordenado;
```

```
if (ordenado==1)
{
   std::cout << ">>> Ordenando vetor...\n";
   std::sort (V.begin(), V.end());
   std::cout << ">>> Vetor ordenado.\n";
}

while(fim == 1){
   while ( teste < 0 || teste > 7 )
   {
      std::cout << "\nTABELA:\n";
      std::cout << "0 : SEQUENCIAL  ITERATIVA\n";
      std::cout << "1 : SEQUENCIAL  RECURSIVA\n";
      std::cout << "2 : BINÁRIA     ITERATIVA\n";
      std::cout << "3 : BINÁRIA     RECURSIVA\n";
      std::cout << "4 : TERNÁRIA    ITERATIVA\n";
      std::cout << "5 : TERNÁRIA    RECURSIVA\n";
      std::cout << "6 : SEQUENCIAL (STD::SEARCH)\n";
      std::cout << "7 : BINÁRIA   (STD::BSEARCH)\n\n";
      std::cout << "Digite o codigo a testar: ";
      std::cin >> teste;
   }
   std::cout << "\n";

   switch(teste)
   {
      case 0:
         tst="valores_bsi.txt";
         break;
      case 1:
         tst="valores_bsr.txt";
         break;
      case 2:
         tst="valores_bbi.txt";
         break;
      case 3:
         tst="valores_bbr.txt";
         break;
      case 4:
         tst="valores_bti.txt";
         break;
      case 5:
         tst="valores_btr.txt";
         break;
      case 6:
         tst="valores_bsic.txt";
         break;
      case 7:
         tst="valores_bbic.txt";
         break;
   }

   outfile.open(tst);

   while ( situacao < 0 || situacao > 2 )
   {
      std::cout << "TABELA:\n";
      std::cout << "0 : K NÃO ESTÁ NO VETOR\n";
```

```
            std::cout << "1 : K ESTÁ A 3/4 DO INÍCIO DO VETOR\n";
            std::cout << "2 : O TERCEIRO K, SE EXISTIR, É PROCURADO\n\n";
            std::cout << "Digite a situação do teste: ";
            std::cin >> situacao;
        }
        std::cout << "\n";

        std::cout << ">>> Iniciando testes...\n";
        for ( auto i(4u); i <= arrSz ; i *= 2){
            auto testes(0);
            switch (situacao)
            {
                case 0:
                    key = 134217729;
                    ocorrencia = 0;
                    break;
                case 1:
                    keyposition = (3 * (i-1))/4;
                    key = V[keyposition];
                    ocorrencia = 0;
                    break;
                case 2:
                    key = 134200700;
                    ocorrencia=3;
                    break;
            }
            while (testes < 100)
            {
                auto start = std::chrono::steady_clock::now();
                //=====================================================//
                        pfunc[teste](V,key,0,i,ocorrencia);
                //=====================================================//
                auto end = std::chrono::steady_clock::now();

                auto diff = end - start;

                resultados_buscas[testes] =  std::chrono::duration <double, std::milli> (diff).count();
                testes++;
            }
            std::cout << ">>> Testes finalizados.\n";
            std::cout << ">>> Calculando média e gravando em arquivo...\n";
            auto media = media_instantanea(resultados_buscas, 100);

            if (outfile.is_open() && outfile.good())
            {
                outfile << i << " " << media << std::endl;
            }
            std::cout << ">>> Média das execuções para " << i << " elementos: " << media << ".\n";
        }
        std::cout << ">>> Saindo...\n";
        outfile.close();
        std::cout << "Deseja refazer teste com outra configuração? (1/0)";
        std::cin >> fim;
        teste=-1;
        situacao=-1;
    }
    return EXIT_SUCCESS;
}
```