UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
INSTITUTO METRÓPOLE DIGITAL

Estrutura de Dados Básica I
Análise Empírica

Gustavo Araújo Carvalho

Yuri Alessandro Dantas Tonheca Martins

INTRODUÇÃO

Dada a problemática computacional sobre busca de um elemento em um arranjo unidimensional, faz-se necessário um estudo de caso para compreender qual dos algoritmos pretendidos por um suposto cliente é o mais eficiente.

Existem diversas funções capazes de buscar um elemento qualquer em um vetor. Entretanto, o que se torna importante estudar é: todas esses algoritmos funcionam da mesma maneira, nas mesma situações?

Fazendo-se uso dos conhecimentos de análise empírica obtidos no decorrer da disciplina de Estrutura de Dados Básica, este relatório faz uma análise, como atividade dessa disciplina do curso de Bacharelado em Tecnologia da Informação, de como se comportam alguns algoritmos de buscas em diferentes cenários, de modo que se possa obter uma conclusão rápida de qual(is) deles são mais recomendados para um determinado cenário.

METÓDO

Os testes de algoritmo foram realizados sobre um vetor de inteiros de tamanho 2^{28}. Além disso, 25 amostras de comprimentos diferentes foram utilizadas para fins de análise, iniciando de 2^4 até o tamanho máximo.

Durante o procedimento, a biblioteca *chrono*, da linguagem de programação C++11, foi utilizada para medir o tempo de execução de cada algoritmo. Esse tempo era armazenado, junto com o tamanho da amostra do vetor no atual momento, em um par ordenado usado para gerar os gráficos (presentes em *Resultados*) de análise.

Uma máquina com um processador *Intel Core i5*, 8GB de memória RAM e sistema operacional Ubuntu 14.04 foi utilizada para realizar o procedimento de testes. O compilador g++ na versão 4.8.4 foi utilizado para a compilação do programa. Todos os testes foram rodados sob as mesmas condições de uso da máquina - dedicada somente a essa tarefa durante o processo.

Para conclusões concretas, todos os algoritmos foram submetidos a três cenários de complexidades temporais distintas:

- Procurar por um elemento *k* que não se encontra no vetor, o que representa o pior caso de busca;
- O elemento *k* procurado encontra-se a ¾ do local de início da busca;
- Buscar a terceira ocorrência de *k* no vetor.

Em todos os casos, o vetor foi populado com números aleatórios gerados a partir da *seed* "*123451234512345*" e com o *range* de 0 até 2^{29}, sendo também ordenado, caso necessário.

Para o primeiro caso, foi pedido aos algoritmos para procurarem um número fora do *range* dos elementos contidos no vetor ($2^{29} + 1$), gerando sempre o caso em que o valor não se encontra no vetor. No segundo, foi passado como chave o número que se encontra na posição ¾ do vetor. Já no terceiro, foi gerado um número aleatório que passa a ocupar as duas primeiras posições do vetor e uma terceira posição também gerada aleatoriamente. Esse número é a chave que os algoritmos deverão encontrar.

Foram analisados oito algoritmos diferentes, implementados todos na linguagem de programação C++, considerando esses três cenários distintos. Os códigos foram:

1. Busca sequencial iterativa;
2. Busca sequencial recursiva;
3. Busca binária iterativa;
4. Busca binária recursiva;
5. Busca ternária iterativa;
6. Busca ternária recursiva.

Além dos seis citados, mais dois casos, próprios do sistema, foram utilizados para fins de testes:

1. Busca sequencial padrão (std::search);
2. Busca binária padrão (std::bsearch).

Em termos de comparação dos algoritmos, a análise foi feita com base no tempo decorrido para *n* elementos.

A busca sequencial percorre todo o vetor de inteiros longo e, caso o elemento seja encontrado no decorrer disso, a função retorna a posição dele. Caso contrário, ela percorrerá todo o laço e, ao final, retornará "-1", significando que o elemento não foi encontrado.

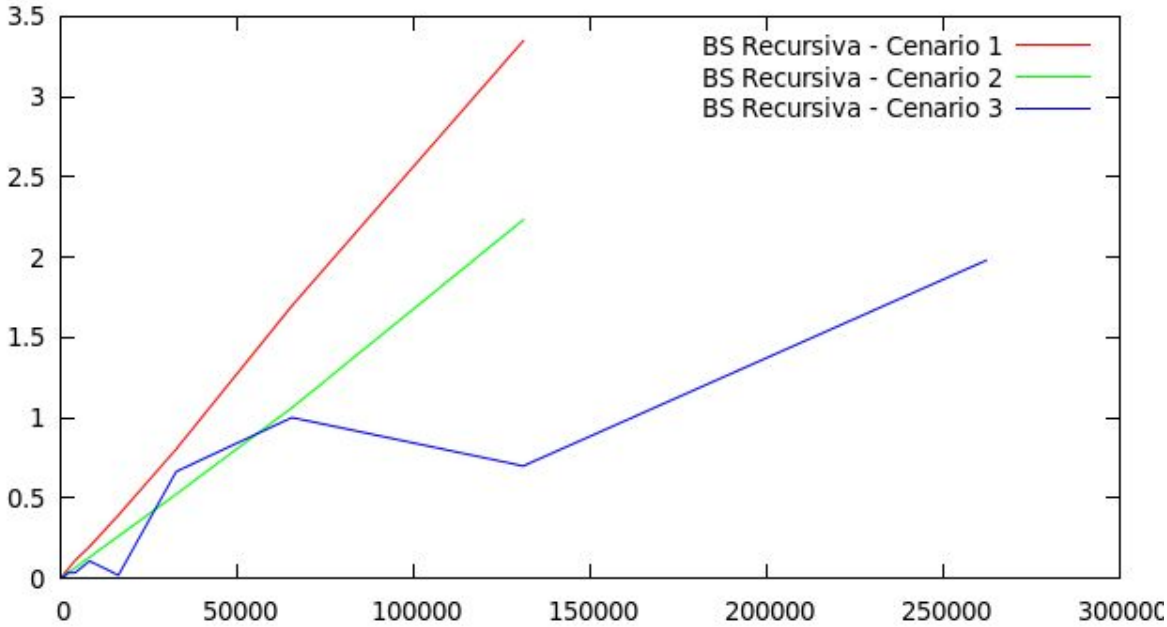
A busca binária, por sua vez, começa comparando o valor desejado com o elemento central do arranjo, transportando a busca para um "sub-vetor" de elementos a direita do central, caso o elemento encontrado seja menor que o procurado, ou à esquerda, caso esse seja maior. A busca se repete até encontrar tal elemento - retornando sua posição - ou não - retornando "-1". Esse processo, no entanto, é dependente do vetor estar sempre ordenado (devido as comparações feitas com o elemento do meio). Essa é uma pré condição para a busca binária.

Não distante disso, a busca ternária segue a mesma pré condição. Entretanto, ao invés de dividir o vetor em duas partes (maior e menor que o elemento central), a busca ternária divide o vetor em três partes. O cálculo dessa divisão é feito da seguinte maneira: "*([2x tamanho do vetor] + valor inicial do vetor) / 2*" e "*(tamanho do vetor + [2x valor inicial do vetor]) / 2*". Isso permite que o vetor vá sendo dividido em partes cada vez menores, até que o elemento seja encontrado, retornando sua posição, ou não, retornando -1.

Para as duas funções próprias do sistema, pouco se altera. Ambas funcionam a partir de *templates*, o que permite que qualquer tipo de dado seja processado. A função binária (std::bsearch) funciona desde que o programador seja capaz de lhe fornecer uma função *compara( )*, que deve retornar se um valor é maior que outro. Já a função padrão (std::search) funciona normalmente para os tipos básicos do sistema, mas, caso necessário, também é capaz de receber um predicado de comparação de dois termos.

RESULTADOS

Imagem 1 - Buscas Sequenciais Recursivas



**Devido às limitações das funções recursivas, a busca sequencial recursiva tem um limite muito baixo*

Imagem 2 - Buscas Sequenciais Iterativas - Cenario 1

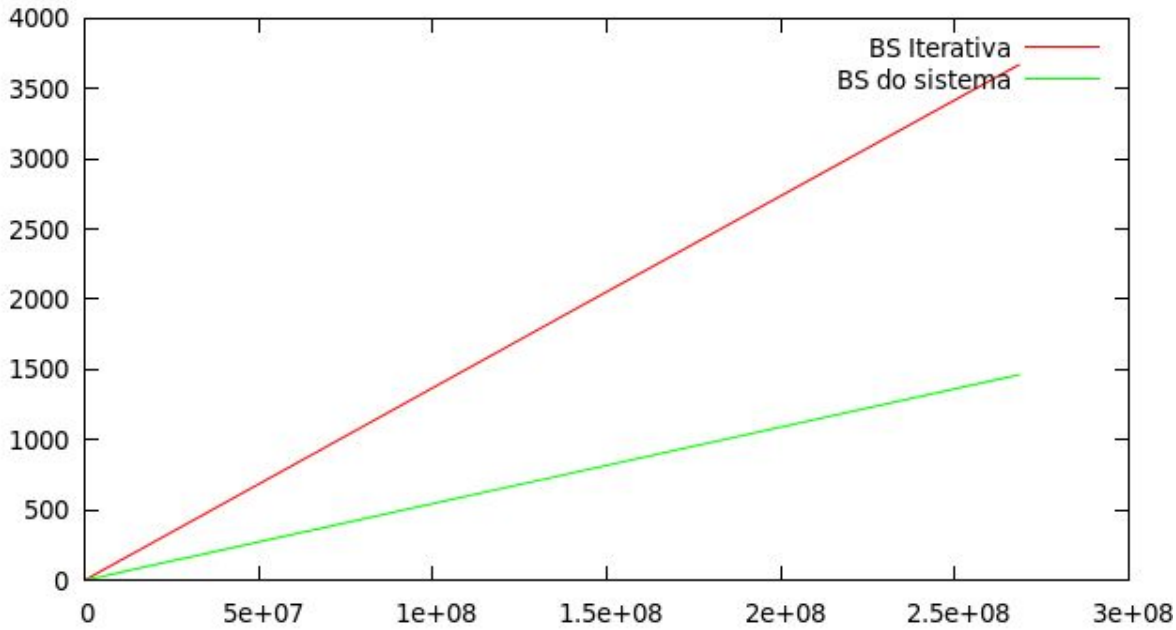


Imagem 3 - Buscas Binarias - Cenario 1

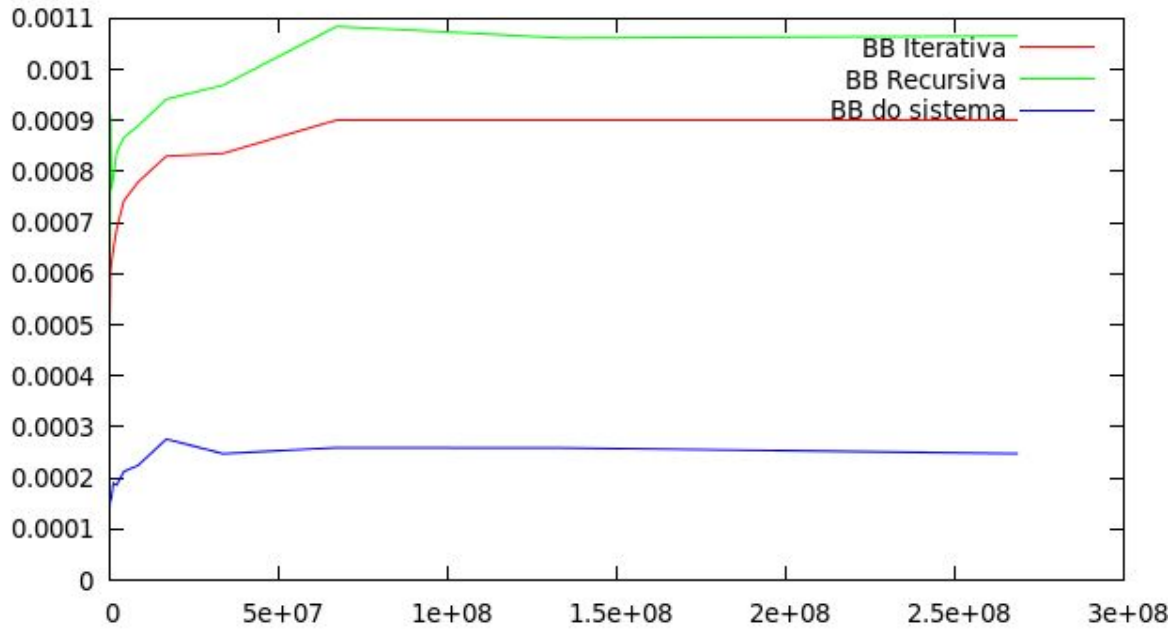


Imagem 4 - Buscas Ternarias - Cenario 1

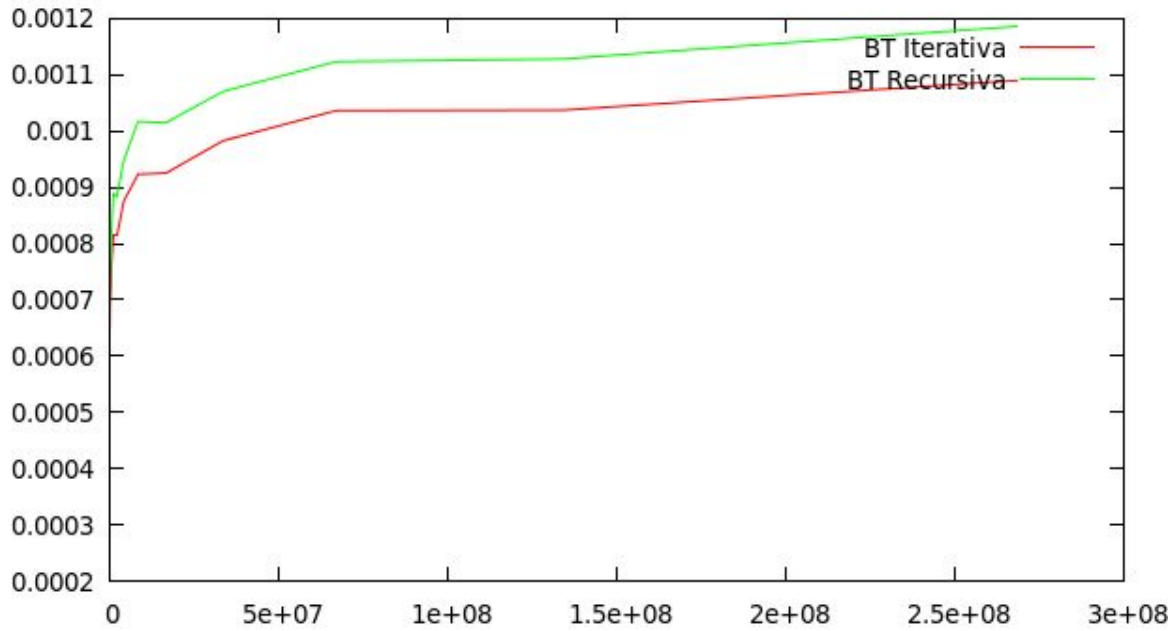


Imagem 5 - Buscas Sequenciais Iterativas - Cenario 2

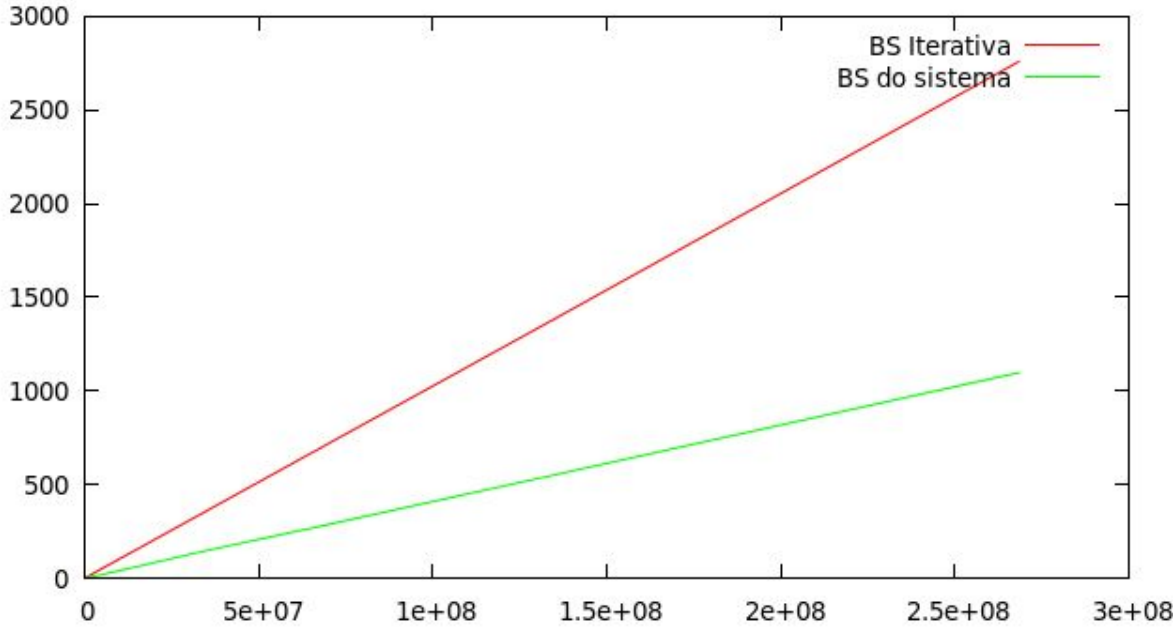


Imagem 6 - Buscas Binarias - Cenario 2

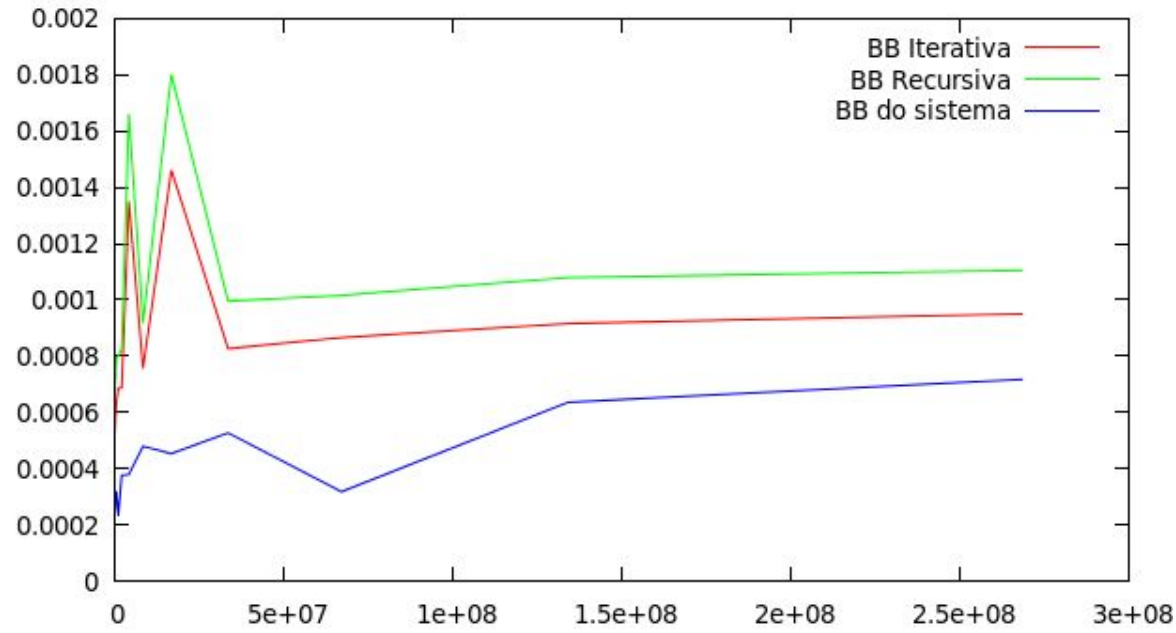


Imagem 7 - Buscas Ternarias - Cenario 2

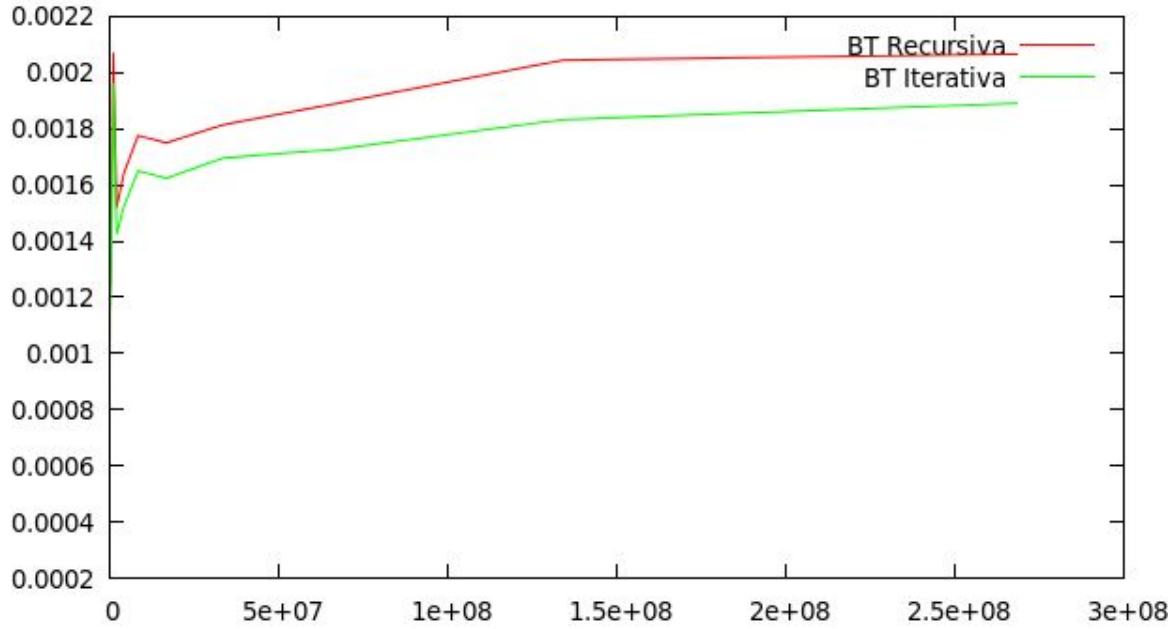


Imagem 8 - Buscas Sequenciais Iterativas - Cenario 3

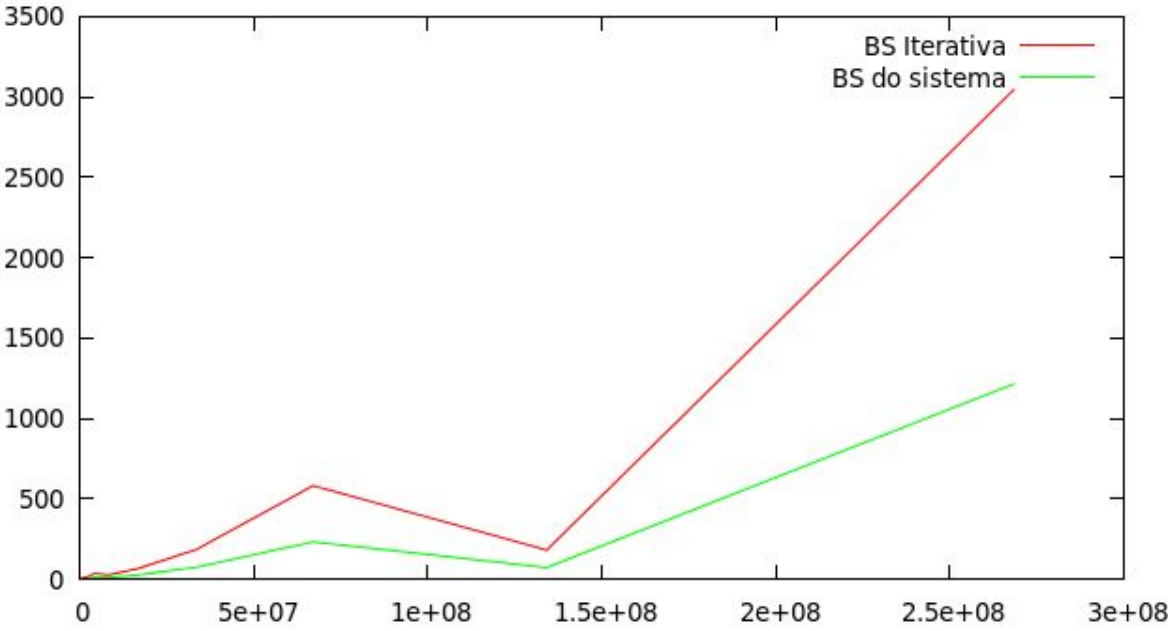


Imagem 9 - Buscas Binarias - Cenario 3

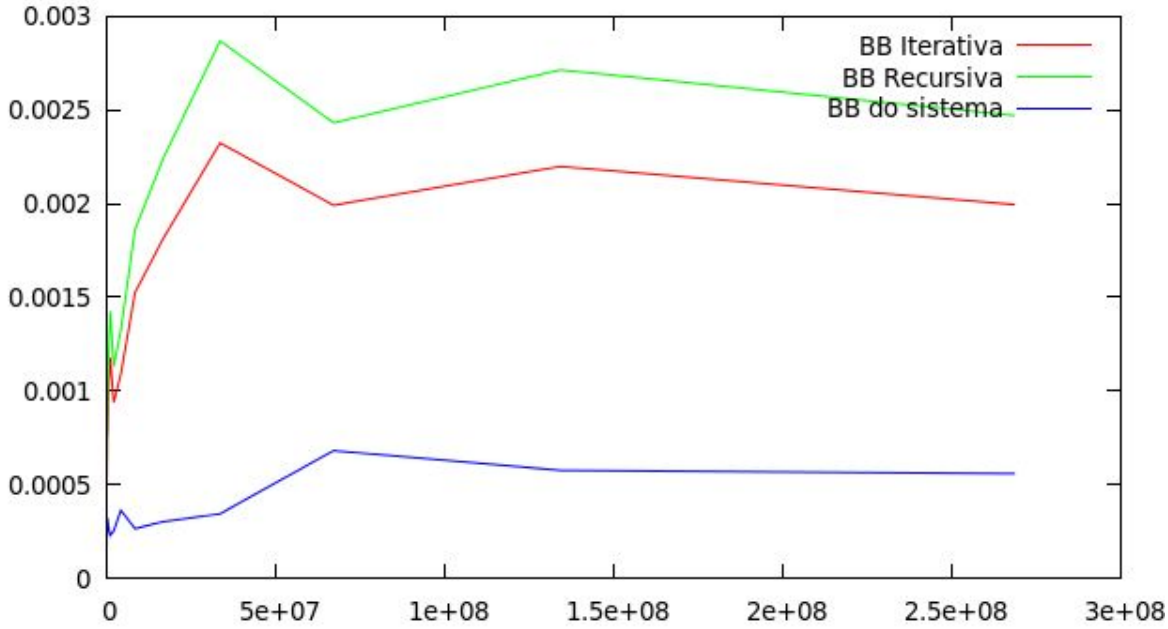
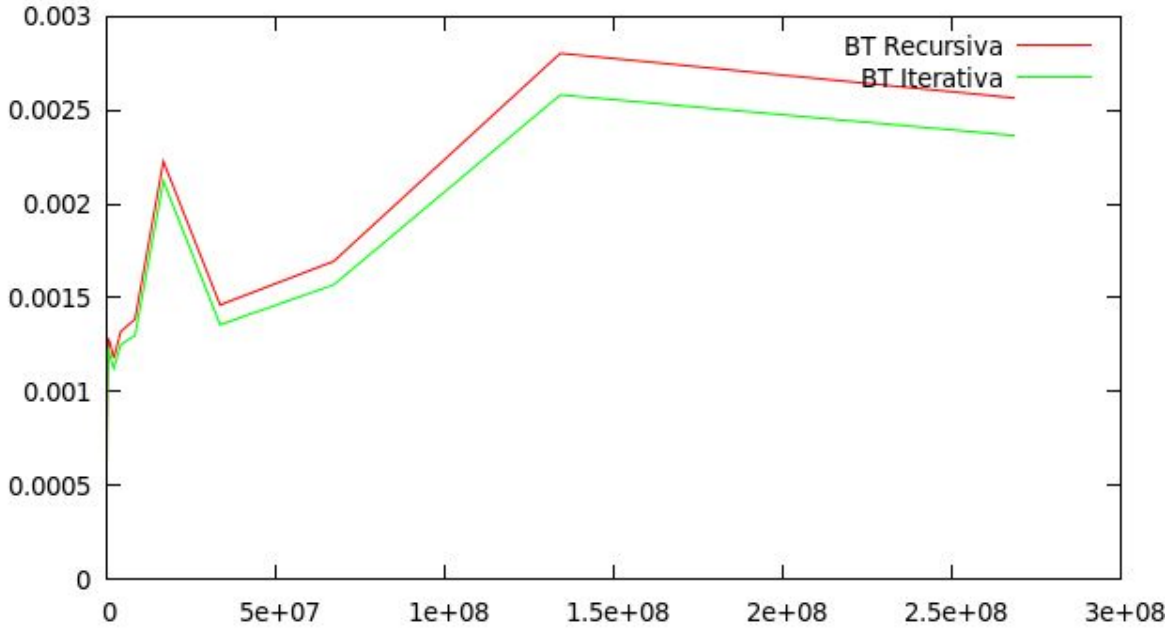


Imagem 10 - Buscas Ternarias - Cenario 3



CONCLUSÃO

A partir da análise empírica realizada sobre os algoritmos, percebe-se que a Busca Binária Iterativa é a melhor função para busca em um vetor de elementos, considerando os três cenários propostos. Naturalmente, a Busca Binária padrão do sistema (std::bsearch) responde melhor do que a busca implementada pelos programadores.

Enquanto isso, percebe-se que a busca ternária, embora boa, é ligeiramente mais lenta do que a busca binária. Isso acontece porque as condições para que a função prossiga elevam muito o número de comparações necessárias.

Já a busca sequencial é bastante lenta, comparada as demais. Mesmo a do sistema (std::search) - mais rápida que as outras sequenciais - ainda é mais lenta que a ternária e a binária. Ainda assim, percebe-se que a função de busca sequencial, embora mais lenta que seus concorrentes em um vetor ordenado, pode-se demonstrar útil quando um vetor está em ordem não crescente, tendo em vista que o tempo para ordenar um vetor (pré condição para a binária e ternária) teria uma complexidade temporal ainda maior do que a busca sequencial.

Tendo em vista que a complexidade de um algoritmo de busca binária é $\Theta(\log_2 n)$, é possível notar que as funções desse tipo, testadas e analisadas neste relatório, respeitam essa complexidade (vide imagem 3). Naturalmente, é possível observar picos e vales nos gráficos de busca binária dos cenários 2 e 3 (imagens 6 e 9), essas flutuações acontecem porque, durante a execução do programa, a busca correspondeu a um caso ruim ou até ao pior caso.

Já as funções ternárias, que possuem complexidade próxima ao log3(n), também são compatíveis com a análise. Tal como na busca binária, os vales e os picos que aparecem em todos os cenários correspondem aos casos ruins da busca de um elemento.

Neste ponto já é possível chegar a conclusão de que a busca binária e a busca ternária se comportam de maneira bastante semelhante. Porém, como já exposto, a busca binária é mais rápida. Isso é ponto chave para responder: “Se dividir o vetor em duas partes é bom para o tempo, porque não dividir ele em mais partes?”. Dada essa comparação entre as duas funções, é notável que não adianta dividi-lo cada

vez mais para achar um número, pois a quantidade de comparações necessárias irá sempre ser maior, aumentando a complexidade temporal da função tal como acontece com a busca ternária versus busca binária.

As funções de busca sequencial, com complexidade linear O(n), também respeitam essa análise (vide imagem 2). É curioso notar que o algoritmos de busca sequencial recursiva não é capaz de buscar elementos em vetores muito grandes devido ao modo como as funções recursivas funcionam: Ao serem chamadas, elas são instanciadas numa pilha e logo começam a invocar a si mesmas repetidamente. Isso acontece para que, quando alcançarem o passo base, serem desempilhadas e voltem a sua instância normal. Porém, se o número de funções nessa pilha ficar grande o suficiente, ocorre um erro de overflow que encerra o programa imediatamente. Esse caso é justamente o que ocorre na busca sequencial recursiva previamente mencionada.

Enfim, no caso em que seja necessário encontrar um elemento num vetor desordenado, a busca sequencial ainda é a recomendada, pois enquanto essa busca é apenas O(n), o custo de qualquer algoritmo de ordenação é no mínimo O(nlog(n)). Conclui-se, portanto, que as funções binárias são as mais rápidas nos cenários de arranjo ordenado e que as buscas sequenciais ainda são a melhor opção para os casos em que o arranjo está desordenado ou com ordenação desconhecida.