Társila Samille Santos da Silveira

Análise empírica de algoritmos

Brasil
2020, v-1.0

Társila Samille Santos da Silveira

Análise empírica de algoritmos

Relatório técnico apresentado à disciplina de Estrutura de Dados Básicas I, como requisito parcial para obtenção de nota referente à unidade I.

Universidade Federal do Rio Grande do Norte - UFRN
Instituto Metrópole Digital - IMD
Bacharelado em Tecnologia da Informação

Brasil
2020, v-1.0

Conteúdo

1 Introdução

Este relatório objetiva realizar análises empíricas e comparações de algoritmos a busca um linear e a busca binária esta ultima em sua forma recursiva e sua forma iterativa, eles foram executadas em um mesmo computador, no sistema operacional Ubutu. Todos os valores foram registrados em tabelas e plotados em gráficos, permitindo fácil comparação.

Nas seções seguintes, é apresentado o método seguido, abrangendo os materiais e ferramentas utilizadas; depois, são mostrados os resultados obtidos e, por fim, as discussões geradas a partir deles. O único apêndice traz a implementação em C++ dos algoritmos escolhidos.

2 Metodologia

2.1 Materiais utilizados

2.1.1 Computador

- Computador: Acer Aspire-A315-42G
- Especificações:

  Processador IAMD® Ryzen 5 3500u gfx × 8

  2 x 8GB DDR4

  HD de 1T GB

  Radeon 540X Series (POLARIS12, DRM 3.35.0, 5.4.0-47-generic, LLVM 10.0.0) / AMD® Raven

- Sistema Operacional e suas Especificações:

  GNU/Linux

  Ubuntu 20.04.1 LTS(x86-64) Kernel 3.36.3

2.1.2 Ferramentas de programação

Os quatro algoritmos escolhidos foram implementados na linguagem C++, padrão ISO/IEC 14882:2011, ou simplesmente C++11.

Os códigos foram compilados pelo CMake, uma família de ferramentas de plataforma cruzada de código aberto projetada para construir, testar e empacotar software. CMake é usado para controlar o processo de compilação

do software usando uma plataforma simples e arquivos de configuração independentes do compilador, e gerar makefiles e espaços de trabalho nativos que podem ser usados no ambiente de copilação de sua escolha. Foi utilizado:

> cmake -S . -Bbuild

> cd build

> make

A biblioteca chrono3 foi responsável pelas medições do tempo de execução.

2.2 Método de comparações

Os algoritmos foram comparados segundo o critério de tempo de execução em relação ao tamanho do array.

Para gerar o gráfico de 50 pontos, o vetor inicial começou com 100 milhões e foi incrementado de 18 milhões em 18 milhões até chegar em 10^9. O vetor foi preenchido com números positivos em sequência e depois foi pedido para a função procurar por -1 (pior caso).

3 Resultados

3.1 Tabela

A Tabela 1 apresenta os resultados dos testes realizados com os dois algoritmos.

Tabela 1:

Tamanho do Array	**Linear Search em ms**	**Binary Search em ms**
100000000	586,748100000000	0,000511
118000000	628,896118000000	0,000781
136000000	761,192136000000	0,000481
154000000	824,62154000000	0,000381
172000000	929,966172000000	0,000491
190000000	872,267190000000	0,000461
208000000	1131,92208000000	0,000621
226000000	1209,25226000000	0,000441
244000000	1316,67244000000	0,000471

Tabela 1:

Tamanho do Array	**Linear Search em ms**	**Binary Search em ms**
262000000	1401,46262000000	0,000611
280000000	1495,68280000000	0,000491
298000000	1619,82298000000	0,000671
316000000	1682,14316000000	0,000741
334000000	1792,25334000000	0,00044
352000000	1976,74352000000	0,000551
370000000	2077,95370000000	0,000481
388000000	2166,33388000000	0,000501
406000000	2264,15406000000	0,000521
424000000	2344,89424000000	0,000511
442000000	2465,36442000000	0,001222
460000000	2547,33460000000	0,000511
478000000	2642,62478000000	0,000682
496000000	2863,67496000000	0,000501
514000000	2782,04514000000	0,000751
532000000	2970,1532000000	0,000611
550000000	2842,12550000000	0,000641
568000000	3169,72568000000	0,000461
586000000	3349,32586000000	0,000541
604000000	3409,14604000000	0,000461
622000000	3524,61622000000	0,000732
640000000	3620,21640000000	0,000441
658000000	3657,81658000000	0,000811
676000000	3553,61676000000	0,000481
694000000	3757,16694000000	0,001022
712000000	3830,01712000000	0,000551

Tabela 1:

Tamanho do Array	**Linear Search em ms**	**Binary Search em ms**
730000000	3835,25730000000	0,000651
748000000	4090,35748000000	0,000511
766000000	4230,16766000000	0,000531
784000000	4216,25784000000	0,000591
802000000	4613,77802000000	0,000571
820000000	4615,04820000000	0,000531
838000000	4668,63838000000	0,000521
856000000	4814,77856000000	0,000591
874000000	4431,15874000000	0,00047
892000000	5016,09892000000	0,000501
910000000	4647,62910000000	0,000551
928000000	4880,19928000000	0,000872
946000000	5292,58946000000	0,000762
964000000	5012,69964000000	0,000632
982000000	5230,51982000000	0,000491
1000000000	5398,071000000000	0,000681

3.2 Gráficos

Gráfico 1 - Análise de tempo para o Algoritmo de Busca Linear

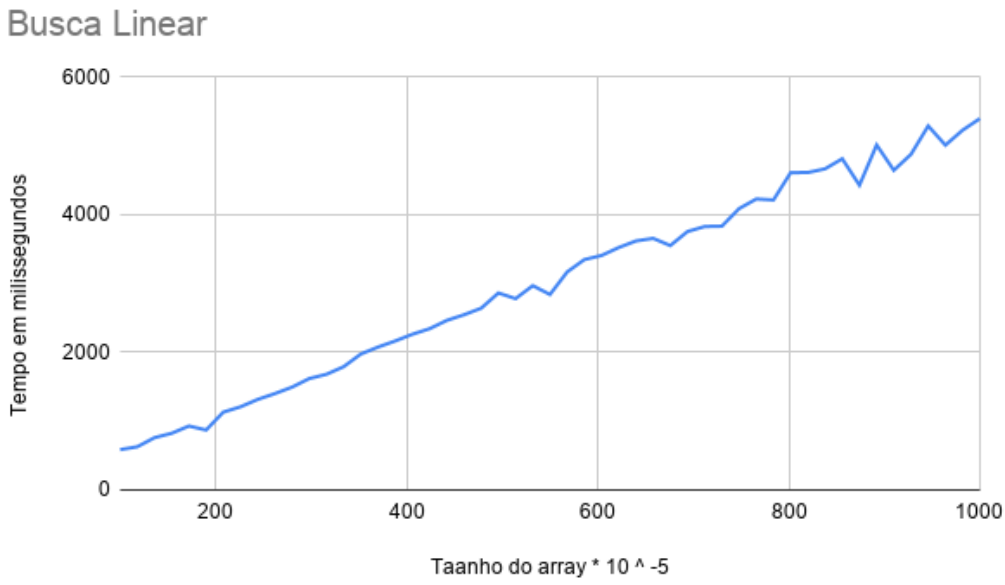

Gráfico 2 - Análise de tempo para o Algoritmo de Busca Binaria

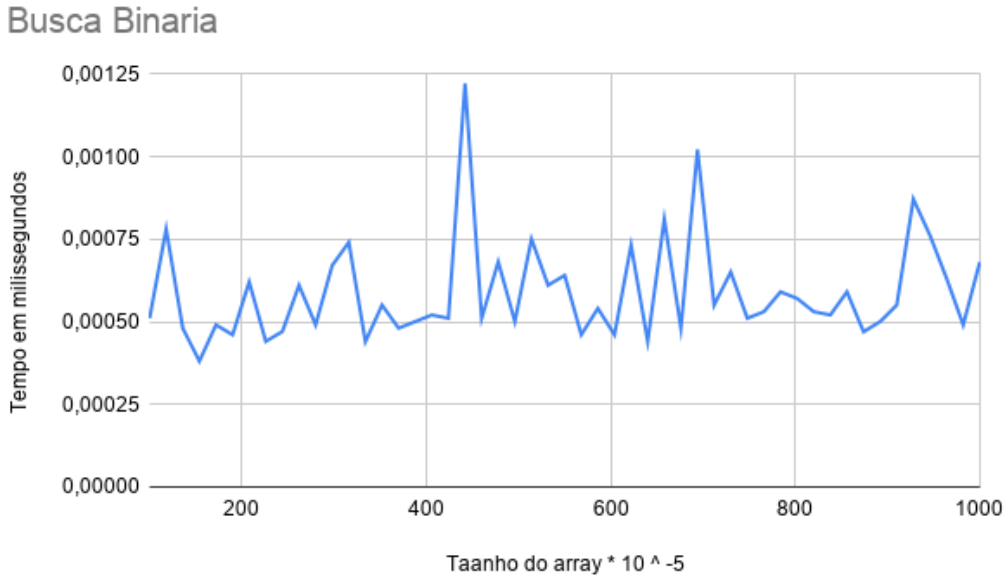

Gráfico 3 - Análise de tempo para o Algoritmo de Busca Binaria e Linear

4 Discussão

Tendo em vista o gráfico 3 ,que compara o tempo de execução do algoritmo de busca Linear e da busca Binaria, e a diferença por exemplo na escala do tempo em milissegundos dos gráficos 1 e 2 é evidente que a solução mais eficiente é a binaria.

Bibliografia

https://cmake.org/

Apêndice - Implementação dos algoritmos em CPP

4.1 Busca Linear

```
value_type * lsearch(value_type *first, value_type *last,
        value_type value){
        while(first != last && *first != value)
                first++;
        return first;
}
```

4.2 Busca Binaria

```
value_type * bsearch(value_type *first, value_type *last,
        value_type value){
        if(*(first) > value)
        return last;
        if (*(last - 1) >= *(first)) {
                value_type index = (last - first)/2;
                value_type * mid = (first + index) ;

                if (*(mid) == value)
                return mid;

                if (*(mid) > value && mid != first){
                        value_type * result =
                                bsearch(first, mid - 1, value);
                        return (*(result) != value
                        && result < last
                        && first <= result) ?
                        last : result;
                }
                if (mid != last)
                return bsearch(mid + 1, last, value);
        }
        return last;
}
```