



Universidade Federal de Viçosa - *Campus* Florestal
Bacharelado em Ciência da Computação
CCF 492 - Tópicos Especiais II
Prof. Daniel Mendes Barbosa

Trabalho Prático 01

Busca e Ordenação

Samuel Jhonata S. Tavares	2282
Wandella Maia de Oliveira	2292

Florestal - MG
2018

Sumário

1	INTRODUÇÃO	3
2	DESENVOLVIMENTO	4
2.1	Considerações Gerais	4
2.2	Implementação	4
2.2.1	Bubble	4
2.2.2	Quicksort	5
2.2.3	Logica	6
2.2.4	Main	10
2.3	Análise dos Resultados	11
3	CONCLUSÃO	17
	REFERÊNCIAS	18

1 Introdução

Este trabalho tem por objetivo apresentar a implementação do exercício 9 da primeira lista de exercícios da disciplina de Projeto e Análise de Algoritmo da Universidade Federal de Viçosa - *Campus* Florestal, utilizando algoritmos de ordenação (*Quicksort* e *Bubblesort*) e busca (Sequencial e Binária) para identificar se, dado um vetor, é possível encontrar nele dois números que somados deem como resultado um número X arbitrário.

No capítulo 2, é apresentado todo o desenvolvimento do trabalho, com suas considerações gerais na seção 2.1, na sequência, é apresentada toda a implementação na seção 2.2, sendo mostrados todos os seus detalhes e, na seção 2.3 é apresentada a análise dos resultados. Já no capítulo 3, é apresentada uma breve conclusão do trabalho.

2 Desenvolvimento

2.1 Considerações Gerais

Foi utilizada como ferramenta a *IDE Netbeans* para a implementação, numa máquina com *Windows 10*, 8 GB de RAM, processador *Intel Core i5*. Para geração de gráficos, foi utilizado o *software Microsoft Excel*.

Como algoritmos de ordenação, foram escolhidos o (*Quicksort* e o *Bubblesort*), com complexidades diferentes ($n \log n$ e n^2) e para busca, os algoritmos de busca sequencial e busca binária, também com complexidades diferentes (n e $\log n$).

O algoritmo de busca binária foi retirado da internet (WIKIPEDIA, 2018) e sofreu algumas alterações para a contagem das operações de comparação. O algoritmo *QuickSort* foi retirado da internet (UFMG, 2010) e também sofreu algumas modificações.

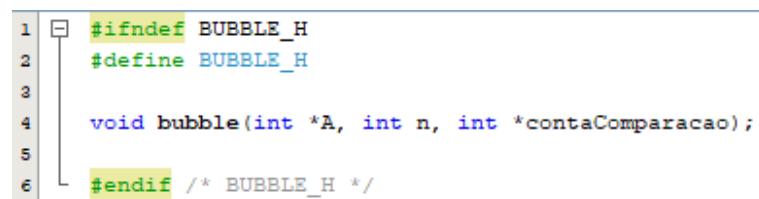
2.2 Implementação

A implementação foi feita de forma modular, onde cada módulo foi separado em arquivos: *Bubble*, *Quicksort*, *Logica* e *Main*.

2.2.1 Bubble

Um dos algoritmos de ordenação escolhido foi o *Bubblesort*, sendo considerado um dos piores algoritmos para este fim, na Figura 1 é mostrado o arquivo de cabeçalho *bubble.h*.

Figura 1 – Arquivo *bubble.h*



```

1  #ifndef BUBBLE_H
2  #define BUBBLE_H
3
4  void bubble(int *A, int n, int *contaComparacao);
5
6  #endif /* BUBBLE_H */

```

Na Figura 2 o arquivo *bubble.c*, com a implementação do algoritmo, que consiste basicamente em percorrer todas as posições do vetor, comparando elemento a elemento com cada um dos outros elementos.

Figura 2 – Arquivo *bubble.c*

```
1  #include <stdio.h>
2
3  #include "bubble.h"
4
5  //ordena de forma crescente
6  void bubble(int *A, int n, int *contaComparacao) {
7      int i, j, aux;
8      for (i = 0; i < n; i++) {
9          for (j = 0; j < n - 1; j++) {
10             (*contaComparacao)++; //comparação do for
11
12             (*contaComparacao)++; //comparação do if
13             if (A[j] > A[j + 1]) {
14                 aux = A[j];
15                 A[j] = A[j + 1];
16                 A[j + 1] = aux;
17             }
18         }
19     }
20 }
```

2.2.2 Quicksort

O *Quicksort*, foi o segundo algoritmo a ser escolhido para realizar testes na ordenação (UFMG, 2010). Este algoritmo, ele tem como paradigma divisão e conquista, por isso ele é um método rápido e eficiente, na Figura 3 é mostrado o arquivo de cabeçalho *quick_sort.h*.

Figura 3 – Arquivo *quick_sort.h*

```
1  #ifndef QUICK_SORT_H
2  #define QUICK_SORT_H
3
4
5  void QuickSort(int *A, int tam, int *contaComparacao);
6
7  #endif /* QUICK_SORT_H */
```

Na Figura 4 o arquivo *quick_sort.c*, mostra a implementação do algoritmo, que consiste em organizar os dados corrente em uma árvore implícita, fazendo chamadas recursivas a si mesmo.

Figura 4 – Arquivo *quick.c*

```

1  #include "quick_sort.h"
2
3  //http://www2.dcc.ufmg.br/livros/algoritmos-edicao2/cap4/codigo/c/4.1a4.7e4.14-ordenacao.c
4  //ordena de forma crescente
5  void Particao(int Esq, int Dir, int *i, int *j, int *A, int *contaComparacao) {
6      int pivo, aux;
7      *i = Esq;
8      *j = Dir;
9      pivo = A[( *i + *j ) / 2]; /* obtem o pivo x */
10     do {
11         //comparação do while
12         while (pivo > A[*i]) {
13             (*contaComparacao)++;
14             (*i)++;
15         }
16
17         //comparação do while
18         while (pivo < A[*j]) {
19             (*contaComparacao)++;
20             (*j)--;
21         }
22
23         (*contaComparacao)++; //comparação do if
24         if (*i <= *j) {
25             aux = A[*i];
26             A[*i] = A[*j];
27             A[*j] = aux;
28             (*i)++;
29             (*j)--;
30         }
31
32         (*contaComparacao)++; //comparação do do_while
33     } while (*i <= *j);
34 }
35
36 void Ordena(int Esq, int Dir, int *A, int *contaComparacao) {
37     int i, j;
38     Particao(Esq, Dir, &i, &j, A, contaComparacao);
39
40     (*contaComparacao)++; //comparação do if
41     if (Esq < j) Ordena(Esq, j, A, contaComparacao);
42
43     //comparação do if
44     (*contaComparacao)++;
45     if (i < Dir) Ordena(i, Dir, A, contaComparacao);
46 }
47
48 void QuickSort(int *A, int n, int *contaComparacao) {
49     Ordena(0, n - 1, A, contaComparacao);
50 }

```

2.2.3 Logica

O módulo de lógica é responsável por implementar a verificação dos pares de valores cuja soma seja o valor informado, além dos métodos auxiliares. Na Figura 5, é mostrado o arquivo de cabeçalho *logica.h*. Foi criada uma estrutura, chamada "Par", para salvar os pares de valores que são calculados na função `valoresSoma()`.

Figura 5 – Arquivo *logica.h*

```

1  #ifndef LOGICA_H
2  #define LOGICA_H
3  #include "quick_sort.h"
4  #include "bubble.h"
5
6  #define BUBBLE 1
7  #define QUICK 2
8
9  #define SEQUENCIAL 1
10 #define BINARIA 2
11
12 typedef struct Par {
13     int p1;
14     int p2;
15 } Par;
16
17 int* geraNumero(int max, int tam);
18 int* copiaVetor(int *a, int tam);
19 void mostraVetor(int *a, int tam);
20 int PesquisaBinaria(int vet[], int chave, int Tam, int *contaComparacao);
21 int pesquisaSequencial(int vet[], int chave, int tam, int *contaComparacao);
22 Par* valoresSoma(int x);
23 int verifica(int *a, int x, int tam, int metOrden, int metPesq, int *contaComparacao);
24
25 #endif /* LOGICA_H */
26

```

Na Figura 6 é mostrada a implementação das funções *geraNumero()* (USP, 2018), *copiaVetor()*, e *mostraVetor()*.

Figura 6 – Arquivo *logica.c*

```

1  #include <stdio.h>
2
3  #include "logica.h"
4
5  int* geraNumero(int max, int tam) { // gera 'tam' números aleatório entre 0 e 'max'
6      int *retorno;
7      int i;
8      retorno = malloc(sizeof (int) * tam);
9
10     for (i = 0; i < tam; i++) {
11         retorno[i] = rand() % (max);
12     }
13
14     return retorno;
15 }
16
17 int* copiaVetor(int *a, int tam) { //cria um novo vetor com os mesmos valores do vetor 'a'
18     int *retorno;
19     int i;
20     retorno = malloc(sizeof (int) * tam);
21
22     for (i = 0; i < tam; i++) {
23         retorno[i] = a[i];
24     }
25
26     return retorno;
27 }
28
29 void mostraVetor(int *a, int tam) { //imprime o vetor na tela
30     int i;
31     for (i = 0; i < tam; i++) {
32         printf("%d ", a[i]);
33     }
34 }

```

A função de *PesquisaBinaria()*, mostrada na Figura 7, é utilizada para identificar a posição que o número procurado está no vetor. Caso não tenha o número no vetor, é retornado -1. Basicamente, a cada iteração, o "subvetor" é dividido em duas partes, e o número do meio é comparado com o valor a ser buscado. Caso ele não seja o valor procurado, será analisado o "subvetor" inferior (com valores menores que o meio) ou o "subvetor" superior (com valores maiores), sendo feito isso até encontrar ou esgotar as possibilidades.

Figura 7 – Função *PesquisaBinaria()*

```

37 //https://pt.wikipedia.org/wiki/Pesquisa_binária
38 //retorna a posição em que o número foi encontrado (ou -1 caso contrário)
39 int PesquisaBinaria(int vet[], int chave, int Tam, int *contaComparacao) {
40     int inf = 0; // limite inferior (o primeiro índice de vetor em C é zero)
41     int sup = Tam - 1; // limite superior (termina em um número a menos. 0 a 9 são 10 números)
42     int meio;
43
44     //percorre o vetor 'metade a metade'
45     while (inf <= sup) {
46         //comparação do while
47         (*contaComparacao)++;
48
49         meio = (inf + sup) / 2;
50
51         //comparação do if
52         (*contaComparacao)++;
53         if (chave == vet[meio]) {
54             return meio;
55         }
56
57         //comparação do if
58         (*contaComparacao)++;
59         if (chave < vet[meio])
60             sup = meio - 1;
61         else
62             inf = meio + 1;
63     }
64     return -1; // não encontrado
65 }

```

A função de *pesquisaSequencial()*, mostrada na Figura 8, também é utilizada para identificar a posição que o número procurado está no vetor. Ela basicamente faz uma pesquisa sequencial até encontrar o valor pesquisado, ou terminar as posições do vetor.

Figura 8 – Função *pesquisaSequencial()*

```

67 //retorna a posição em que o número foi encontrado (ou -1 caso contrário)
68 int pesquisaSequencial(int vet[], int chave, int tam, int *contaComparacao) {
69     int i;
70     for (i = 0; i < tam; i++) { //percorre posição a posição, até encontrar a chave
71         //comparação do for
72         (*contaComparacao)++;
73
74         //comparação do if
75         (*contaComparacao)++;
76         if (chave == vet[i]) {
77             return i;
78         }
79     }
80     return -1; //não encontrado
81 }

```


A função *valoresSoma()*, mostrada na Figura 9, é responsável por verificar os pares de números cuja soma seja igual ao valor indicado. As possibilidades foram calculadas após observar que, para valores pares, a quantidade de possibilidades é sempre a metade menos um e, para valores ímpares, é sempre o piso da divisão por 2.

Figura 9 – Função *valoresSoma()*

```
83 //retorna um par de valores cuja a soma é 'x'
84 Par* valoresSoma(int x) {
85     Par *retorno;
86     int qtd;
87
88     //calcula a qtd de possibilidades diferentes
89     if (x % 2 == 0) { //caso 'x' seja par
90         qtd = x / 2 - 1;
91     } else { //caso 'x' seja impar
92         qtd = x / 2;
93     }
94
95     retorno = malloc(sizeof (Par) * qtd);
96     int i, j, cont = 0;
97     //calcula os pares de valores
98     for (i = 1; i <= qtd; i++) {
99         for (j = 2; j < x; j++) {
100             if (i + j == x) {
101                 retorno[cont].p1 = i;
102                 retorno[cont].p2 = j;
103                 cont++;
104             }
105         }
106     }
107     return retorno;
108 }
```

A função *verifica()*, mostrada na Figura 10, é responsável por verificar se no vetor existem dois valores cuja soma é o valor indicado. Ela utiliza as funções de ordenação e de busca, de acordo com a entrada indicada na sua chamada. Também utiliza a função *valoresSoma()* para calcular os pares a serem pesquisados. Quando é invocada, o vetor é copiado para um outro, para que, ao fim da sua execução, ele não tenha sido alterado pelas funções de ordenação. Ao terminar sua execução, esse vetor é desalocado.

Figura 10 – Função *verifica()*

```

110 //verifica se existem os pares de valores em 'a' cuja soma é 'x'
111 int verifica(int *a, int x, int tam, int metOrden, int metPesq, int *contaComparacao) {
112     Par *pares;
113     int *S;
114     S = copiaVetor(a, tam); //copia os valores do vetor 'a' em 'S'
115     *contaComparacao = 0;
116     int i;
117     //ordena o vetor
118     if (metOrden == QUICK) {
119         QuickSort(S, tam, contaComparacao);
120     } else if (metOrden == BUBBLE) {
121         bubble(S, tam, contaComparacao);
122     } else {
123         free(S);
124         return -1;
125     }
126     //verifica os pares de valores possiveis
127     pares = valoresSoma(x);
128
129     //verifica se existem os pares de valores
130     if (metPesq == SEQUENCIAL) {
131         for (i = 0; i < x / 2; i++) {
132             if (pesquisaSequencial(S, pares[i].p1, tam, contaComparacao) != -1 && pesquisaSequencial(S, pares[i].p2, tam, contaComparacao) != -1) {
133                 free(S);
134                 return 1;
135             }
136         }
137     } else if (metPesq == BINARIA) {
138         for (i = 0; i < x / 2; i++) {
139             if (PesquisaBinaria(S, pares[i].p1, tam, contaComparacao) != -1 && PesquisaBinaria(S, pares[i].p2, tam, contaComparacao) != -1) {
140                 free(S);
141                 return 1;
142             }
143         }
144     } else {
145         free(S);
146         return -1;
147     }
148 }

```

2.2.4 Main

Na função principal(*Main*) mostrada na Figura 11, chamamos os métodos dos arquivos *bubble*, *quick.h* e *logica* para realizar a execução do programa. Sendo assim, ao realizar os testes optamos por definir um vetor de sete elementos com valores para testar os algoritmos de ordenação, e para fins de comparação, utilizamos ponteiros para contar a complexidade temporal do algoritmo. Vale ressaltar que, no caso de contar a complexidade de tempo, nosso objetivo foi contar o número de comparações. Por fim, após fazer tais comparações, verificamos se foi possível fazer a soma do número x, no caso ele retornará 1 se for possível, 0 se não for e -1 se der algum erro.

Figura 11 – Função *main()*

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include "logica.h"
5
6  int main() {
7      int tam, x, result1, result2, result3, result4;
8      int *a, *b;
9      int tamanhos[] = {10, 100, 1000, 10000, 50000, 100000, 200000};
10     unsigned long int contaComparacao1;
11     unsigned long int contaComparacao2;
12     unsigned long int contaComparacao3;
13     unsigned long int contaComparacao4;
14     srand(time(NULL)); //semente de números aleatórios setada para o horário atual
15
16     int i;
17     for (i = 0; i < 7; i++) {
18
19         tam = tamanhos[i];
20         x = rand() % (tam * 2);
21         a = geraNumero(tam * 2, tam);
22
23         printf("\nTamanho vetor: %d", tam);
24         printf("\nPesquisar por x=%d", x);
25         result1 = verifica(a, x, tam, BUBBLE, SEQUENCIAL, &contaComparacao1);
26         printf("\nComparacoes (BUBBLE, SEQUENCIAL): %lu", contaComparacao1);
27         result2 = verifica(a, x, tam, BUBBLE, BINARIA, &contaComparacao2);
28         printf("\nComparacoes (BUBBLE, BINARIA): %lu", contaComparacao2);
29         result3 = verifica(a, x, tam, QUICK, SEQUENCIAL, &contaComparacao3);
30         printf("\nComparacoes (QUICK, SEQUENCIAL): %lu", contaComparacao3);
31         result4 = verifica(a, x, tam, QUICK, BINARIA, &contaComparacao4);
32         printf("\nComparacoes (QUICK, BINARIA): %lu", contaComparacao4);
33         printf("\n Resposta: ");
34         if (result1 == 1 && result2 == 1 && result3 == 1 && result4 == 1) {
35             printf("É possível");
36         } else if (result1 == 0 && result2 == 0 && result3 == 0 && result4 == 0) {
37             printf("Não é possível");
38         } else {
39             printf("DIVERGENCIA!!! (%d,%d,%d,%d)", result1, result2, result3, result4);
40         }
41         printf("\n-----\n");
42     }
43     return 0;
44 }

```

2.3 Análise dos Resultados

Após a implementação, foi executado por duas vezes o programa para se fazer uma análise dos resultados obtidos. Cada uma delas tem como valores de entrada 10, 100, 1.000, 10.000, 50.000, 100.000 e 200.000, cada uma delas para as quatro possibilidades:

- *Bubblesort* + Busca Sequencial,
- *Bubblesort* + Busca Binária,
- *Quicksort* + Busca Sequencial e
- *Quicksort* + Busca Binária.

Para fazer a análise de tempo, optou-se por analisar uma operação relevante nos algoritmos: número de comparações. Assim, é possível contar quantas comparações foram feitas em cada versão de uso dos algoritmos de ordenação e busca, sendo possível compará-los com esse parâmetro, sem utilizar tempo de relógio, uma vez que isso poderia ser afetado pelas configurações do computador, ou mesmo por outras aplicações.

Na Figura 12 é mostrada a saída da primeira execução do programa.

Figura 12 – Saída Execução 1

```
Tamanho vetor: 10
Pesquisar por x=14
Comparacoes (BUBBLE, SEQUENCIAL): 354
Comparacoes (BUBBLE, BINARIA): 277
Comparacoes (QUICK, SEQUENCIAL): 243
Comparacoes (QUICK, BINARIA): 166
Resposta: Não é possível
-----

Tamanho vetor: 100
Pesquisar por x=179
Comparacoes (BUBBLE, SEQUENCIAL): 21810
Comparacoes (BUBBLE, BINARIA): 20088
Comparacoes (QUICK, SEQUENCIAL): 2986
Comparacoes (QUICK, BINARIA): 1264
Resposta: É possível
-----

Tamanho vetor: 1000
Pesquisar por x=1955
Comparacoes (BUBBLE, SEQUENCIAL): 1999948
Comparacoes (BUBBLE, BINARIA): 1998046
Comparacoes (QUICK, SEQUENCIAL): 18315
Comparacoes (QUICK, BINARIA): 16413
Resposta: É possível
-----

Tamanho vetor: 10000
Pesquisar por x=17189
Comparacoes (BUBBLE, SEQUENCIAL): 200118210
Comparacoes (BUBBLE, BINARIA): 199980310
Comparacoes (QUICK, SEQUENCIAL): 328418
Comparacoes (QUICK, BINARIA): 190518
Resposta: É possível
-----

Tamanho vetor: 50000
Pesquisar por x=11242
Comparacoes (BUBBLE, SEQUENCIAL): 705267088
Comparacoes (BUBBLE, BINARIA): 704932977
Comparacoes (QUICK, SEQUENCIAL): 1452234
Comparacoes (QUICK, BINARIA): 1118123
Resposta: É possível
-----

Tamanho vetor: 100000
Pesquisar por x=24675
Comparacoes (BUBBLE, SEQUENCIAL): 2820081518
Comparacoes (BUBBLE, BINARIA): 2819930898
Comparacoes (QUICK, SEQUENCIAL): 2586719
Comparacoes (QUICK, BINARIA): 2436099
Resposta: É possível
-----

Tamanho vetor: 200000
Pesquisar por x=1010
Comparacoes (BUBBLE, SEQUENCIAL): 2690201022
Comparacoes (BUBBLE, BINARIA): 2690188763
Comparacoes (QUICK, SEQUENCIAL): 5039617
Comparacoes (QUICK, BINARIA): 5027358
Resposta: É possível
-----

EXECUTAR SUCCESSFUL (tempo total: 11m 32s)
```

É fácil perceber que as combinações com o algoritmo *Quicksort* são muito mais eficientes do que as combinações com *Bubblesort*. Isso se deve pelo fato de que a complexidade assintótica do primeiro é melhor que a do segundo, tendo um grande impacto no resultado final.

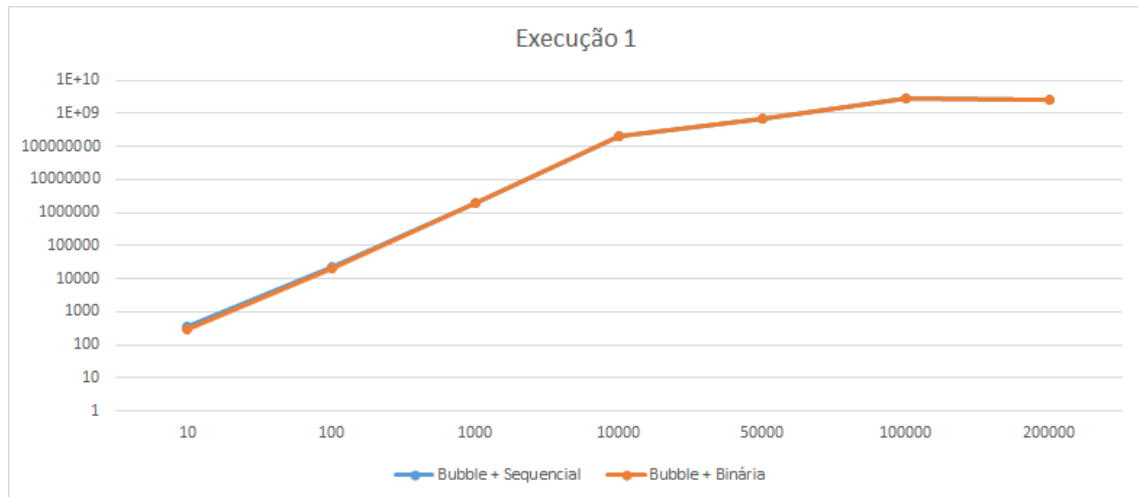
Também é possível perceber uma pequena melhora na utilização da Busca binária, sendo mais eficiente que a busca sequencial.

Para a primeira execução, escolheu-se a representação dos gráficos em escala logarítmica.

No gráfico da Figura 13 são mostrados os valores das execuções do *Bubblesort* com pesquisa sequencial e com pesquisa binária. Foi escolhido mostrar os resultados desses dois algoritmos separadamente dos outros dois que utilizam *Quicksort*, devido ao fato dos valores, quando as entradas são grandes, serem muito discrepantes (na casa de 10^9 , enquanto *Quicksort* na casa de 10^6). Assim, é possível perceber que o tempo de execução

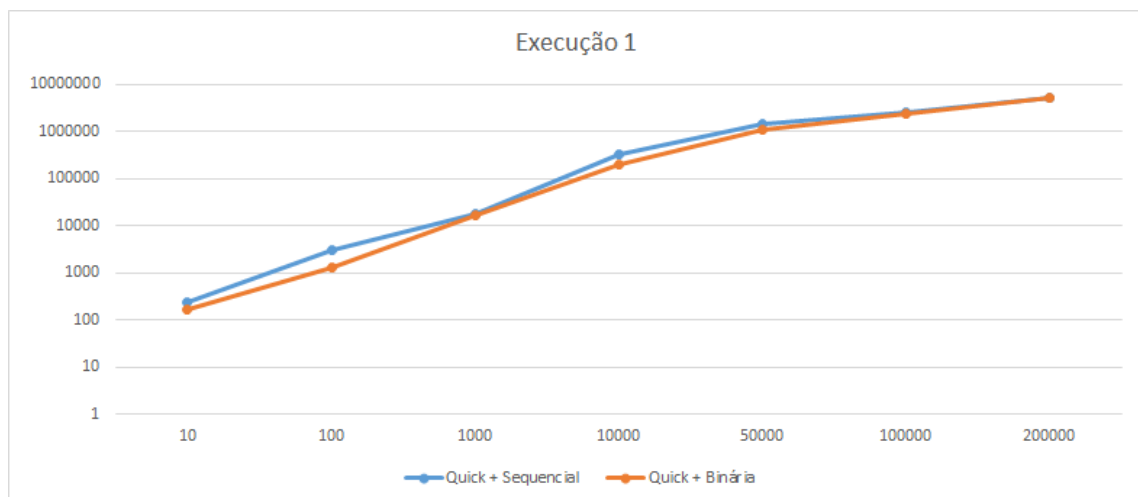
crece de forma acelerada quando a entrada aumenta.

Figura 13 – Gráfico primeira execução para os algoritmos *Bubblesort* + Busca Sequencial e *Bubblesort* + Busca Binária



Já no gráfico da Figura 14 são mostrados os valores das execuções para os algoritmos *Quicksort* com pesquisa sequencial e com pesquisa binária. Em comparação com os algoritmos utilizando *Bubblesort*, o resultado é bem melhor, crescendo mais lentamente de acordo com o valor de entrada. Mais uma vez, a busca binária se mostrou mais eficiente.

Figura 14 – Gráfico primeira execução para os algoritmos *Quicksort* + Busca Sequencial e *Quicksort* + Busca Binária



Para a segunda execução, escolheu-se a representação dos gráficos em escala linear.

Para se ter uma melhor análise, foi feita uma segunda execução, como é mostrada na Figura 15.

Figura 15 – Saída Execução 2

```

Tamanho vetor: 10
Pesquisar por x=9
Comparacoes (BUBBLE, SEQUENCIAL): 270
Comparacoes (BUBBLE, BINARIA): 235
Comparacoes (QUICK, SEQUENCIAL): 147
Comparacoes (QUICK, BINARIA): 112
Resposta: Não é possível
-----

Tamanho vetor: 100
Pesquisar por x=113
Comparacoes (BUBBLE, SEQUENCIAL): 20534
Comparacoes (BUBBLE, BINARIA): 19925
Comparacoes (QUICK, SEQUENCIAL): 1770
Comparacoes (QUICK, BINARIA): 1161
Resposta: É possível
-----

Tamanho vetor: 1000
Pesquisar por x=803
Comparacoes (BUBBLE, SEQUENCIAL): 2000830
Comparacoes (BUBBLE, BINARIA): 1998073
Comparacoes (QUICK, SEQUENCIAL): 18117
Comparacoes (QUICK, BINARIA): 15360
Resposta: É possível
-----

Tamanho vetor: 10000
Pesquisar por x=156
Comparacoes (BUBBLE, SEQUENCIAL): 200360202
Comparacoes (BUBBLE, BINARIA): 199980960
Comparacoes (QUICK, SEQUENCIAL): 583666
Comparacoes (QUICK, BINARIA): 204424
Resposta: É possível
-----

Tamanho vetor: 50000
Pesquisar por x=985
Comparacoes (BUBBLE, SEQUENCIAL): 705035800
Comparacoes (BUBBLE, BINARIA): 704932834
Comparacoes (QUICK, SEQUENCIAL): 1232545
Comparacoes (QUICK, BINARIA): 1129579
Resposta: É possível
-----

Tamanho vetor: 100000
Pesquisar por x=8909
Comparacoes (BUBBLE, SEQUENCIAL): 2819985136
Comparacoes (BUBBLE, BINARIA): 2819930895
Comparacoes (QUICK, SEQUENCIAL): 2309993
Comparacoes (QUICK, BINARIA): 2255752
Resposta: É possível
-----

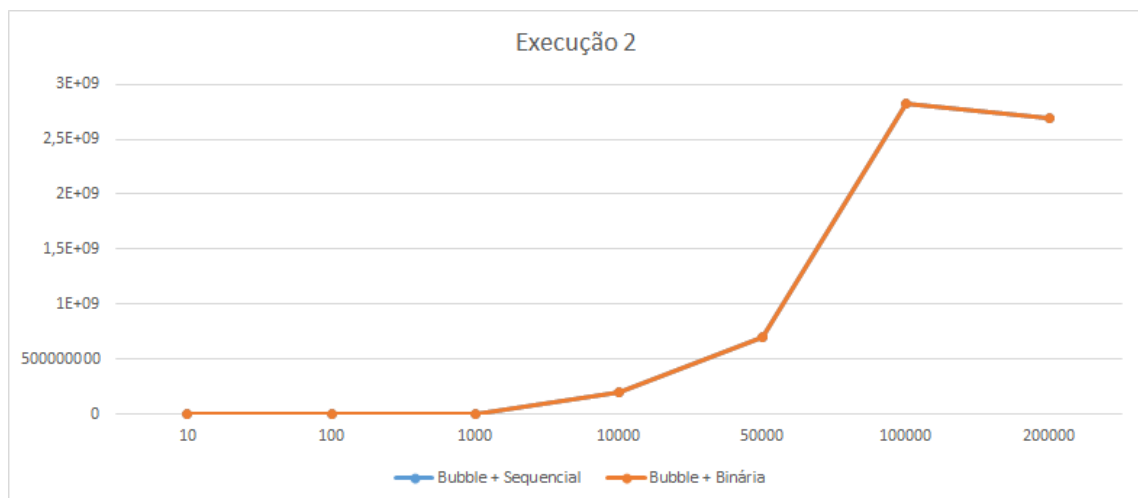
Tamanho vetor: 200000
Pesquisar por x=28941
Comparacoes (BUBBLE, SEQUENCIAL): 2690541832
Comparacoes (BUBBLE, BINARIA): 2690188754
Comparacoes (QUICK, SEQUENCIAL): 5074900
Comparacoes (QUICK, BINARIA): 4721822
Resposta: É possível
-----

EXECUTAR SUCCESSFUL (tempo total: 11m 30s)

```

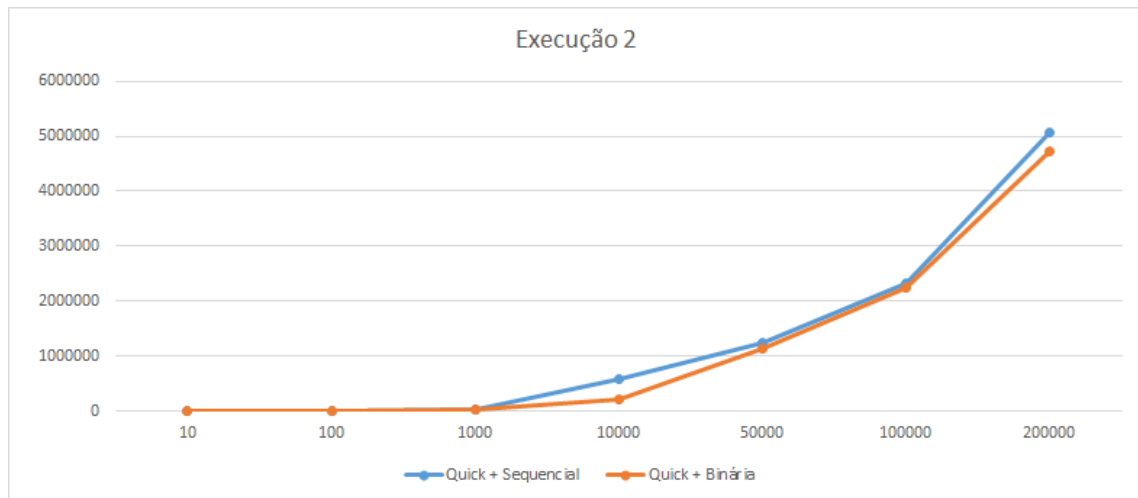
O gráfico da Figura 16 mostra um resultado muito parecido com a primeira execução para valores dos algoritmos com *Bubblesort*.

Percebe-se que o tempo de execução de 100 mil para 200 mil teve uma leve queda, provavelmente pelo fato da configuração de entrada.

Figura 16 – Gráfico segunda execução para os algoritmos *Bubblesort* + Busca Sequencial e *Bubblesort* + Busca Binária

O gráfico da Figura 17 mostra um resultado muito parecido com a primeira execução para valores dos algoritmos com *Quicksort*.

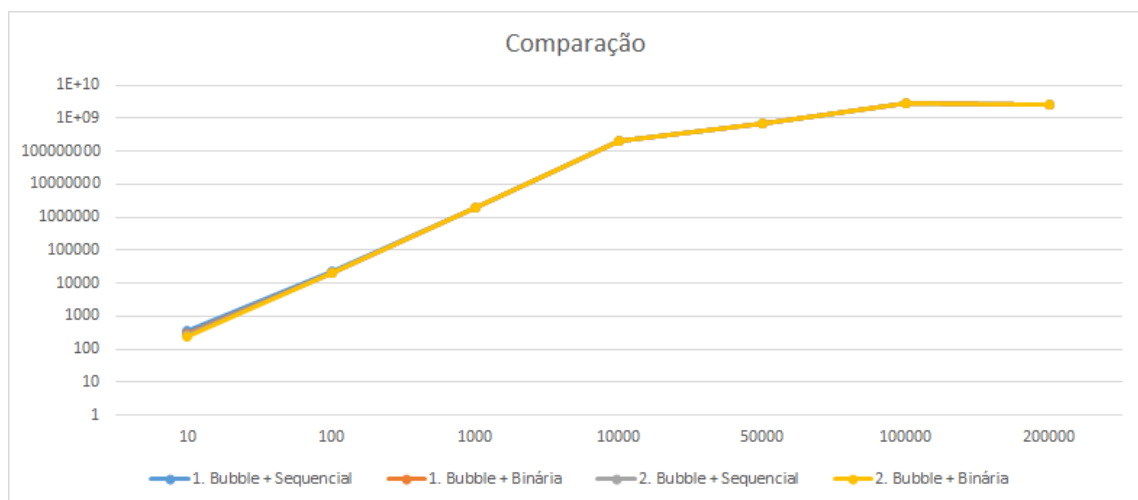
Figura 17 – Gráfico segunda execução para os algoritmos *Quicksort* + Busca Sequencial e *Quicksort* + Busca Binária



Para a comparação entre as execuções, escolheu-se a representação dos gráficos em escala logarítmica.

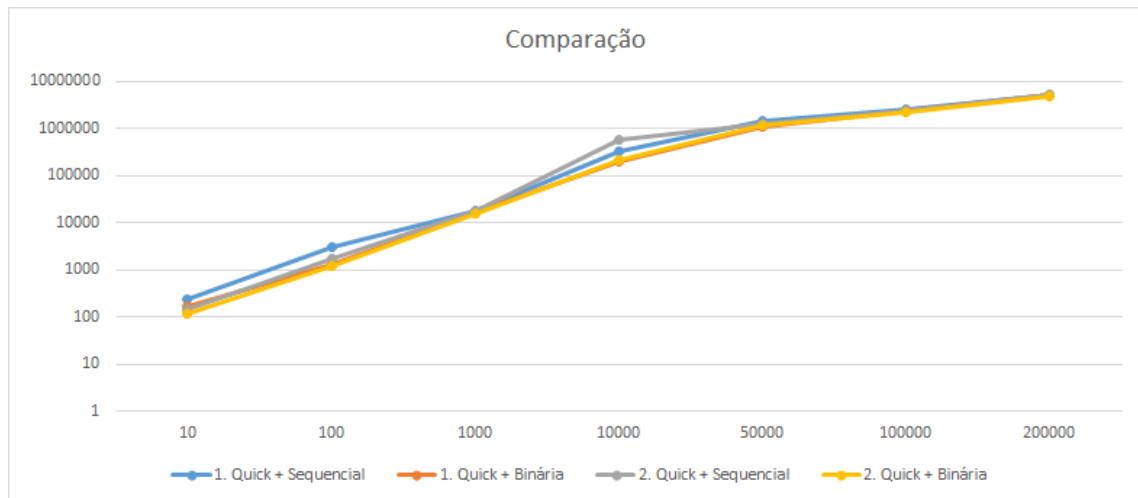
No gráfico da Figura 18 é feita uma comparação entre as duas execuções para os algoritmos com *Bubblesort*, onde é possível notar o mesmo comportamento, diferenciando em números absolutos apenas pelas configurações iniciais.

Figura 18 – Comparação *Bubblesort* + Busca Sequencial e *Bubblesort* + Busca Binária em ambas as execuções



No gráfico da Figura 19 é feita uma comparação entre as duas execuções para os algoritmos com *Quicksort*, onde é possível notar o mesmo que aconteceu com os algoritmos com *Bubblesort*.

Figura 19 – Comparação *Quicksort* + Busca Sequencial e *Quicksort* + Busca Binária em ambas as execuções



Assim, é possível perceber que os valores de ambas as execuções se assemelharam muito, podendo concluir, na prática, que a melhor combinação é usar os algoritmos *Quicksort* + Busca Binária.

Já fazendo uma análise assintótica, foi possível concluir que os algoritmos *Quicksort* + Busca Sequencial e *Quicksort* + Busca Binária são $O(n \log n)$, uma vez que *Quicksort* é $O(n \log n)$, Busca Binária é $O(\log n)$ e Busca sequencial é $O(n)$. Portanto, assintoticamente, não há diferença no uso de qualquer um desses dois, sendo as melhores escolhas.

Analisando assintoticamente Os algoritmos *Bubblesort* + Busca Sequencial e *Bubblesort* + Busca Binária são $O(n^2)$, uma vez que o algoritmo *Bubblesort* é $O(n^2)$, Busca Binária é $O(\log n)$ e Busca sequencial é $O(n)$. Assintoticamente também não faz diferença escolher qualquer um desses.

3 Conclusão

Esse trabalho possibilitou um melhor entendimento do funcionamento dos algoritmos *Quicksort*, *Bubblesort*, Busca Binária e Busca Sequencial, avaliando-os na prática e também assintoticamente. Além disso, foi possível estudar o comportamento desses algoritmos ao unir ordenação e busca. Sendo assim, chegamos a conclusão que, por pouca diferença o *Quicksort* + Busca Binária obteve o melhor resultado em termos de tempo de execução.

Referências

- UFMG. *Ordenação.c*. 2010. Disponível em: <<http://www2.dcc.ufmg.br/livros/algoritmos-edicao2/cap4/codigo/c/4.1a4.7e4.14-ordenacao.c>>. Acesso em: 02 set. 2018.
- USP. *Números aleatórios*. 2018. Disponível em: <<https://www.ime.usp.br/~pf/algoritmos/aulas/random.html>>. Acesso em: 02 set. 2018.
- WIKIPEDIA. *Pesquisa binária*. 2018. Disponível em: <https://pt.wikipedia.org/wiki/Pesquisa_binária>. Acesso em: 02 set. 2018.