Disciplina **SCC0606 - Estrutura de Dados II**
Prof. Robson Leonardo Ferreira Cordeiro

Relatório - Primeiro Projeto

Felipe Andrade Garcia Tommaselli (11800910)
Gianluca Capezzuto Sardinha (11876933)
São Carlos, Brasil

1. Introdução

Primeiramente, é importante ressaltar a importância da análise de algoritmos, resolvendo problemas para ordenação de números. Os algoritmos de ordenação que foram examinados serviram para o desenvolvimento da identificação de diferentes estilos/métodos que podem ser implementados com um mesmo intuito, garantindo que seja tarefa do programador decidir qual é o melhor caso para o seu uso desejado. Nesse sentido, há a necessidade de uma prova tanto teórica quanto empírica, implicando na necessidade de análises assintóticas e práticas, provando a complementaridade entre elas. Portanto, bastou que fosse feita a implementação da forma mais curta e simples para a verificação dos resultados, possibilitando uma discussão sobre os resultados que foram obtidos tanto em aula quanto ao longo do trabalho.

2. Implementação

A divisão do código, em primeira análise, foi feita por meio de um arquivo “main.c”, um "source.c" e um “header.h”. O “header.h” é um arquivo TAD, no qual foram feitas as declarações e atribuições elementares ao código todo. Essa escolha é muito vantajosa quando sabemos que há valores que precisam ser trocados globalmente no código de forma constante. O arquivo “source.c”, ele sim contém algoritmos e lógicas de ordenação em si. Nele, além de todos os algoritmos de ordenação, estão as funções bases de manipulação da lista principal a ser ordenada (como criação, destruição, etc.). Por fim, o arquivo “main.c” reúne toda a lógica teórica do projeto de forma prática. Nele estão implementadas todas as chamadas para as funções do “source.c” e as mensurações de tempo dos algoritmos de ordenação. Essa decisão de organização do projeto foi inspirada em arquivos trabalhados durante as aulas, de forma que facilitasse o entendimento de terceiros sobre as implementações escolhidas.

Outro aspecto da implementação é a fidelidade com o pseudocódigo nos algoritmos de ordenação. Tentou-se ao máximo manter os algoritmos implementados coerentes com o pseudocódigo fornecido, a fim de evitar erros ou divergências desnecessárias.

Além disso, outras pequenas escolhas de implementação foram feitas, a fim de melhorar a utilização do código. Dentre elas, uma estrutura condicional que garante que para pequenas entradas o vetor original e o ordenado são impressos na tela, uma vez que é possível averiguar que o algoritmo de fato está realizando a ordenação sem poluir muito a saída; ou ainda, outras estruturas de escolha que pedem ao usuário qual algoritmo para ser executado, tamanho da entrada e como a entrada estará ordenada.

Por se tratar de um projeto com uma extensão e complexidade considerável é natural que surjam diversos problemas e erros. Contudo, é também natural que algumas dificuldades se

sobressaiam, instigando a solução de problemáticas mais complexas, essas merecem maior atenção. Uma das dificuldades mais instigantes que surgiram durante o trabalho foi na análise teórica do "Bubblesort" e do "Bubblesort aprimorado". Esta dificuldade citada foi dada devido a dúvidas em relação ao sinal negativo da função quadrática que foi encontrado no decorrer da análise assintótica, porém para evitar problemas com a notação do algoritmo foi considerado o módulo da mesma.

3. Análise Teórica

Analisar e estimar a complexidade de um código antes mesmo de executá-lo é uma técnica indispensável para qualquer programador, esta análise pode prevenir erros e incertezas ao longo de um projeto. Nesse momento, será analisada a complexidade de tempo teórica dos algoritmos de ordenação implementados. Além disso, vale ressaltar que toda análise foi baseada nas análises feitas em aula e nos pseudocódigos disponibilizados.

3.1. Bubblesort

Pela análise da Figura 1 é possível estimar a complexidade do algoritmo de ordenação Bubblesort. A figura detalha passo a passo a análise da complexidade em unidades de tempo, para cada operação e processo realizado pelo pseudocódigo. Vale enfatizar que essa análise também é congruente com o algoritmo implementado, uma vez que eles devem apenas diferir por constantes (atribuições e pequenos processos), ou seja, o Big-O do pseudocódigo disponibilizado é o mesmo do código implementado.

Figura 1- Análise de complexidade do Bubblesort

```
## Bubble Sort

1: procedimento Main; por fim, tem-se que -10n^2-14n+2 unidades de tempo rege esse algoritmo
2:      n ← quantidade de inteiros a serem ordenados; 1 unidade para inicialização de n
3:      Inicializar A[n] . Array de inteiros de comprimento n; 1 unidade para inicialização do vetor
4:      Bubble_Sort(A, n) . Chamada do algoritmo de ordenação
5: fim procedimento
...
1: procedimento Bubble_Sort(A, n); Custo total: somando tudo, tem-se -10n^2-14n unidades de tempo, ou seja,
                                    a função tem Big-O O(n^2)
2:      para i ← 0 até n-1 faça; 1 unidade para inicialização de i, n unidades para testar se i < n-1 e n-1
                                 unidades para incrementar, ou seja, i = 2n
3:          para j ← 0 até n-i-2; 1 unidade para inicialização de j, n-i-1 unidades para testar se j <= n-i-2
                                  e n-i-2 unidades para incrementar, ou seja, j = -2n-2
4:              se A[j] > A[j + 1] então; caso A[j] seja maior que a[j+1] há um tempo de execução constante
                                          igual a 1, que é a comparação entre os valores
5:                  troque(A[j], A[j + 1]); 3 unidades vindas das atribuições feitas para a troca de variáveis,
                                              sendo executada n-i-2 vezes (pelo comando "para"), ou seja, -3n-6
6:              fim se
7:              j ← j + 1
8:          fim para
9:          i ← i + 1
10:     fim para
11: fim procedimento
```

Como observado na imagem, o resultado estimado foi "unidades de tempo", ou seja, desconsiderando os procedimentos pouco custosos, chega que o Big-O do Bubblesort com base no pseudocódigos é $O(n^2)$; ou seja, ele é um algoritmo de custo quadrático para melhor, pior e médio caso. Resultado esse já esperado, uma vez que é necessário duas estruturas de repetição aninhadas percorrendo todo o vetor independente do caso a caso.

3.2. Bubblesort aprimorado

Seguindo a mesma análise do caso anterior, visto que o "Bubblesort aprimorado" é apenas uma melhoria de melhor caso do "Bubblesort", é possível analisar passo a passo o custo em unidades de tempo pela Figura 2.

Figura 2- Análise de complexidade do Bubblesort aprimorado

```
## Bubble Sort Aprimorado

1: procedimento Main; por fim, tem-se que -10n^2-17n-2 unidades de tempo rege esse algoritmo
2:      n ← quantidade de inteiros a serem ordenados; 1 unidade para inicialização de n
3:      Inicializar A[n] . Array de inteiros de comprimento n; 1 unidade para inicialização do vetor
4:      Optimized_Bubble_Sort(A, n) . Chamada do algoritmo de ordenação;
5: fim procedimento
...
1: procedimento Bubble_Sort(A, n); Custo total: somando tudo, tem-se -10n^2-17n-4 unidades de tempo, ou
                                   seja, a função tem Big-O O(n^2)
2:      ordenado ← true; 2 unidades, uma para inicialização de ordenado e outra pela atribuição de true
3:      para i ← 0 até n-1 faça; 1 unidade para inicialização de i, n unidades para testar se i < n-1 e n-1
                                 unidades para incrementar, ou seja, i = 2n
4:          para j ← 0 até n-i-2; 1 unidade para inicialização de j, n-i-1 unidades para testar se j <= n-i-2
                                  e n-i-2 unidades para incrementar, ou seja, j = -2n-2
5:              se A[j] > A[j + 1] então; caso A[j] seja maior que a[j+1] há um tempo de execução constante
                                          igual a 1, que é a comparação entre os valores
6:                  ordenado ← false; 1 unidade pela atribuição de false
7:                  troque(A[j], A[j + 1]); 3 unidades vindas das atribuições feitas para a troca de variáveis,
                                            sendo executada n-i-2 vezes (pelo comando "para"), ou seja, -3n-6
8:              fim se
9:              j ← j + 1
10:         fim para
11:         se ordenado = true então; 1 unidade para comparar se ordenado é igual a true
12:             sair
13:         fim se
14:         i ← i + 1
15:     fim para
16: fim procedimento
```

É possível estimar pela imagem que a complexidade de tempo é algo em torno de "( ) unidades de tempo". Apenas há variações de resultados comparado com o "Bubblesort" no melhor caso, nessa situação o algoritmo aprimorado acaba possuindo um custo em unidades de tempo menor, porém apenas diferendo em constantes, uma vez que a complexidade ainda é $O(n^2)$ para todos os casos. Mais uma vez é possível comprovar que esse resultado é coerente, visto que novamente há duas estruturas de repetição aninhadas, o que garante a complexidade quadrática ao algoritmo.

3.3. Quicksort

Para o estudo feito em relação ao Quicksort é possível observar pela Figura 3 a complexidade presente neste algoritmo, detalhando passo a passo o que foi apresentado no pseudocódigo é possível verificar qual o custo de cada função. Dessa forma, será possível comparar com os resultados obtidos pela análise empírica, vide seção 4.3.

Figura 3 - Análise de complexidade do Quicksort

```
## Quicksort

1: procedimento Main; Custo total: somando tudo, tem-se 16nlog(n)+38log(n)+3 unidades de tempo, ou
                                seja, a função tem Big-O O(nlog(n))
2:      n ← quantidade de inteiros a serem ordenados; 1 unidade para inicialização de n
3:      Inicializar A[n] . Array de inteiros de comprimento n; 1 unidade para inicialização do vetor
4:      Quicksort(A, 0, n – 1) . Chamada do algoritmo de ordenação;
5: fim procedimento
...
1: procedimento Quicksort(A, inicio, fim)
2:      se inicio < fim então; 1 unidade para comparação
3:          pivo ← Random_Partition(A, inicio, fim); 1 unidade para atribuição e outra por chamar a função de
                                                     complexidade (n)
4:          Quicksort(A, inicio, pivo – 1); chamando recursivamente a própria função, ou seja, complexidade
                                            de log(n), porém é passado como um dos parâmetros da função o
                                            pivo que atribui uma função de complexidade n; por fim, tem-se
                                            que a complexidade fica de nlog(n)
5:          Quicksort(A, pivo + 1, fim); o mesmo serve para ordenar a outra parte
6:      fim se
7: fim procedimento
...
1: procedimento Random_Partition(A, inicio, fim)
2:      k ← número inteiro aleatório entre inicio e fim; 5 unidades, sendo 1 da atribuição e as outras das
                                                         operações matemáticas
3:      troque(A[k], A[fim]); 3 unidades vindas das atribuições feitas para a troca de variáveis
4:      devolve Partition(A, inicio, fim); chamanda a função de complexidade (n)
5: fim procedimento
...
1: função Partition(A, inicio, fim)
2:      pivo ← A[fim]; 2 unidades, sendo 1 para inicialização e outra para atribuição
3:      i ← (inicio – 1); 3 unidades, sendo 1 para inicialização, 1 para atribuição e outra para subtração
4:      para j ← inicio até fim-1 faça; 1 unidade para inicialização de j, n unidades para testar se i < n-1 e
                                        n-1 unidades para incrementar, ou seja, j = 2n
5:          se A[j] < pivo então; 1 unidade para comparação
6:              i ← i + 1; 2 unidades, sendo 1 da soma e outra da atribuição
7:              troque(A[i], A[j]); 3 unidades vindas das atribuições feitas para a troca de variáveis
8:          fim se
9:          j ← j + 1
10:     fim para
11:     troque(A[i + 1], A[fim]); 3 unidades vindas das atribuições feitas para a troca de variáveis
12:     devolve (i + 1); 2 unidades, sendo 1 da soma e outra pelo fato de devolver um valor
13: fim função
```

Portanto, a partir do observado na figura concluiu-se que o Big-O do Quicksort com base no pseudocódigo apresentado é $O(nlog(n))$. Dessa forma, ele é um algoritmo que possui um custo razoável para maioria dos casos, exceto para o pior. Pois, como visto anteriormente nas aulas foi apresentado que para casos em que é necessário o algoritmo executar todos os seus procedimentos o custo é excedido, chegando ao caso de $O(n^2)$, mas continua sendo uma ótima opção para uso prático em casos mais genéricos como o apresentado no pseudocódigo.

3.4. Radixsort

Apesar do seu funcionamento alternativo baseando-se na sequência de algarismos dos números inteiros, o Radixsort é um ótimo algoritmo para a utilização em casos que seja necessário entradas muito grandes. Nesse sentido, é importante ressaltar que ordenando cada algarismo por iteração feita ao fim de todas as iterações a lista dos elementos estará ordenada, o que é possível ser observado na Figura 4, garantindo a complexidade linear do algoritmo.

Figura 4 - Análise de complexidade do Radixsort

```
## Radix Sort

1: procedimento Main; por fim, tem-se que 22n+29 unidades de tempo rege esse algoritmo
2:      n ← quantidade de inteiros a serem ordenados; 2 unidades, sendo 1 para inicialização de n e outra para
                                                    atribuição do tamanho a n
3:      Inicializar A[n]; 1 unidade para inicialização de vetor
4:      Radix_Sort(A, n); 1 Unidade para chamar a função
5: fim procedimento
...
1: procedimento Radix_Sort(A, n); Custo total: somanto tudo, tem-se 22n+27 unidades de tempo, ou seja, a função
                                  tem Big-O O(n)
2:      maior ← maior inteiro armazenado em A; 1 unidade para inicialização de maior, n unidades para testar
                                            se maior < n-1 e n-1 unidades para incrementar, ou seja, maior =
                                            2n
3:      posicao ← 1; 2 unidades, sendo 1 para inicialização e outra para atribuição
4:      enquanto (maior/posicao) > 0 faça; 1 unidade de tempo pela comparação
5:          Counting_Sort(A, n, posicao);
6:          posicao ← posicao * 10; 2 unidades, sendo uma da multiplicação e outra da atribuição
7:      fim enquanto
8: fim procedimento
...
1: procedimento Counting_Sort(A, n, posicao)
2:      Inicializar B[10] com zeros; 1 unidade pela inicialização do vetor
3:      para i ← 0 até n-1 faça; 1 unidade para inicialização de i, n unidades para testar se i < n-1 e n-1
                                 unidades para incrementar, ou seja, i = 2n
4:          chave ← A[i]/posicao, 2 unidades, sendo 1 para atribuição e outra para a divisão
5:          chave ← chave mod 10; 2 unidades, sendo 1 para atribuição e outra para o resto da divisão por 10
6:          B[chave] ← B[chave] + 1; 2 unidades, sendo 1 para atribuição e outra para a soma
7:          i ← i + 1
8:      fim para
9:      para i ← 1 até 9 faça; 1 unidade para inicialização de i, 9 unidades para testar se i < 9 e 8 unidades
                               para incrementar, ou seja, i = 18
10:         B[i] ← B[i] + B[i - 1]; 2 unidades, sendo 1 para atribuição e outra para soma
11:         i ← i + 1
12:     fim para
13:     Inicializar C[n]; 1 unidade pela incialização do vetor
14:     para i ← n-1 até 0 faça; 1 unidade para inicialização de 1, n unidades para testar se n-1 < i e n-1
                                 unidades para decrementar, ou seja, i = 2n
15:         chave ← A[i]/posicao; 2 unidades, sendo 1 para a atribuição e outra para a divisão
16:         chave ← chave mod 10; 2 unidades, sendo 1 para atribuição e outra para o resto da divisão por 10
17:         B[chave] ← B[chave] - 1; 2 unidades, sendo 1 para atribuição e outra para a subtração
18:         C[B[chave]] ← A[i]; 1 unidade para atribuição
19:         i ← i - 1
20:     fim para
21:     para i ← 0 até n-1 faça; 1 unidade para inicialização de i, n unidades para testar se n-1 < i e n-1
                                 unidades para incrementar, ou seja, i = 2n
22:         A[i] ← C[i]; 1 unidades para atribuição
23:         i ← i + 1;
24:     fim para
25: fim procedimento
```

Por conseguinte, pode concluir-se a partir da imagem que o Big-O do pseudocódigo acima apresentado é de $O(n)$, fato esse que garante a utilização dele para diversos casos nos quais as constantes que multiplicam o valor de $n$não sejam grandes o suficientes para ter uma influência pejorativa na majoração do tempo.

3.5. Heapsort

Por último há o Heapsort, que apesar do mesmo nome não tem o mesmo intuito da memória heap, sendo formado por uma árvore no qual seu funcionamento consiste na análise das subárvores

criadas a partir da criação do nó raiz. Nesse sentido, é possível observar na Figura 5 que há essa diferenciação pelos elementos da direita e da esquerda, garantindo a formação de uma árvore.

Figura 5 - Análise de complexidade do Heapsort

```
## Heapsort

1: procedimento Main
2:      n ← quantidade de inteiros a serem ordenados; 1 unidade para inicialização de n
3:      Inicializar A[n]; 1 unidade para inicialização do vetor
4:      Heapsort(A, n); 1 unidade para chamar a função
5: fim procedimento
...
1: procedimento Heapsort(A, n) Custo total: somando tudo, tem-se (3nlog(n)+57n-2log(n)-30)/2 unidades de tempo,
                                        ou seja, a função tem Big-O (nlog(n))
2:      para i ← (n/2)-1 até 0 faça; 1 unidade para inicialização de i, n/2 unidades para testar se (n/2)-1 < i
                                    e (n/2)-1 unidades para decrementar, ou seja, i = n
3:          Heapify(A, n, i); chamando a função que possui log(n) de complexidade
4:          i ← i - 1
5:      fim para
6:      para i ← n-1 até 1 faça; n-1 unidades para testar se n-1 < i e n-1 unidades para decrementar, ou seja,
                                i = 2n-2
7:          troque(A[0], A[i]); 3 unidades vindas das atribuições feitas para a troca de variáveis
8:          Heapify(A, i, 0) chamando a função que possui log(n) de complexidade
9:          i ← i - 1
10:     fim para
11: fim procedimento

1: procedimento Heapify(A, n, i)
2:      maior ← i; 1 unidade para atribuição
3:      esquerda ← (2 * i) + 1; 3 unidades, sendo uma para atribuição, uma para multiplicação e outra para soma
4:      direita ← (2 * i) + 2; 3 unidades, sendo uma para atribuição, uma para multiplicação e outra para soma
5:      se esquerda < n & A[esquerda] > A[maior] então; 2 unidades pelas comparações feitas
6:          maior ← esquerda; 1 unidades para atribuição
7:      fim se
8:      se direita < n & A[direita] > A[maior] então; 2 unidades pelas comparações feitas
9:          maior ← direita; 1 unidades para atribuição
10:     fim se
11:     se maior <> i então; 2 unidades pelas comparações feitas
12:         troque(A[i], A[maior]); 3 unidades vindas das atribuições feitas para a troca de variáveis
13:         Heapify(A, n, maior); Caso a comparação feita seja verdadeira, há a chamada recursiva da função
                                  Heapify, mais o retorno da função, ou seja, fazendo a árvore de recorrência
                                  é possível concluir que T(n)=log(n)
14:     fim se
15: fim procedimento
```

Por fim, é possível observar a partir do visto na figura que o resultado para esse caso é o mesmo encontrado no Quicksort, ou seja com complexidade de ordem $O(nlog(n))$ para todos os casos. Ademais, mesmo tendo ordem de complexidade igual ao do Quicksort é importante ressaltar que para casos de entradas de ordem 10^n com $n > 6$ é possível encontrar uma pequena diferença no algoritmo do Heapsort.

4. Resultados e Discussão

Nesse momento serão brevemente discutidos alguns dos resultados obtidos, principalmente a análise de complexidade empírica dos algoritmos de ordenação, a qual é coerente com a análise teórica já feita. Em todos os casos, foi feita uma implementação do algoritmo para diferentes entradas e foi analisado o tempo de execução e o desvio padrão obtido. A partir dos resultados de tempos foram plotados gráficos Log-Log, para analisar os coeficientes de inclinação de reta gerados, facilitando a análise de complexidade.

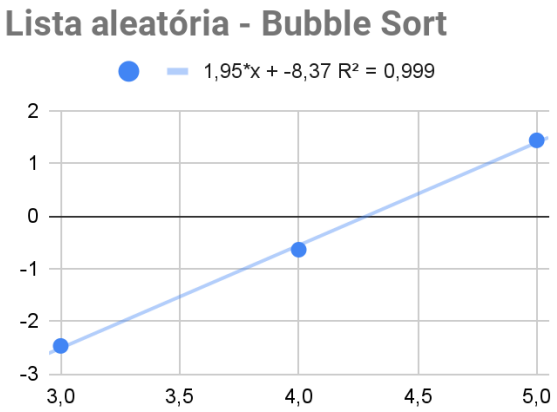
Além disso, os desvios padrões foram coletados e analisados, (conforme implementado no código), porém como pelo gráfico já é possível ter uma análise visual da dispersão dos resultados, preferiu-se não explicitar esses dados a fim de manter a simplicidade e clareza do relatório.

4.1. Bubblesort

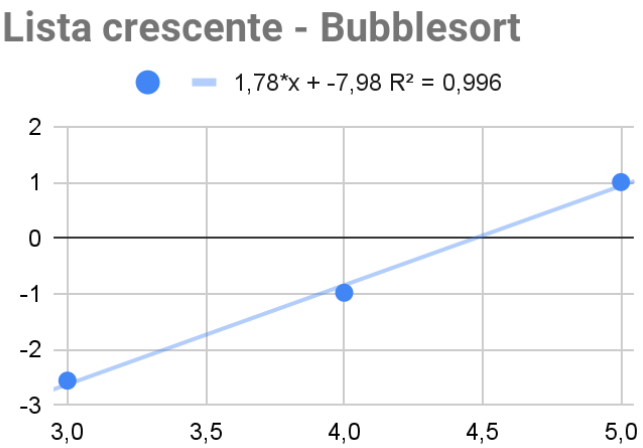
Como já descrito anteriormente o "Bubblesort" possui complexidade (n^2), ou seja, complexidade quadrática. Por isso, é esperado que ao plotar o gráfico com seus valores de tempo para diferentes entradas em uma planilha virtual utilizando um modelo log-log, seja obtida uma reta de coeficiente de inclinação perto de 2.

Conjunto tabelas e gráficos 1 - Análise empírica do Bubblesort

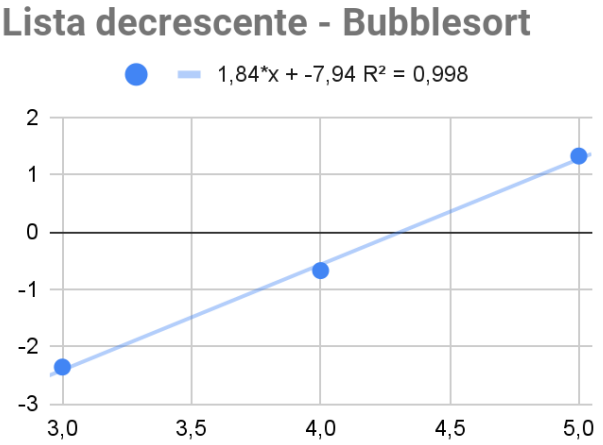
	BUBBLESORT	
	Entradas	Tempo
Lista aleatória	1000	0,0033812
	10000	0,2276885
	100000	27,2834197
	-	-
	-	-



	BUBBLESORT	
	Entradas	Tempo
Lista crescente	1000	0,0027616
	10000	0,1047713
	100000	10,1650830
	-	-
	-	-



	BUBBLESORT	
	Entradas	Tempo
Lista decres-cente	1000	0,0043575
	10000	0,2104078
	100000	21,0474406
	-	-
	-	-



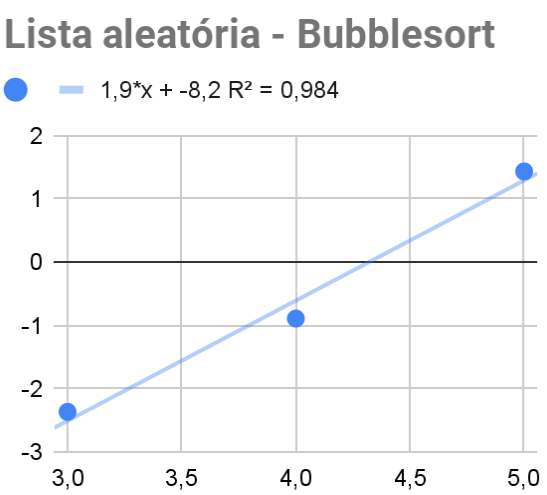
Pela análise dos gráficos obtidos, fica evidente que o comportamento assintótico do "Bubblesort" se aproxima muito do quadrático, dentro de uma estimativa de erro aceitável, resultado esse extremamente coerente com a análise teórica.

4.2. Bubblesort aprimorado

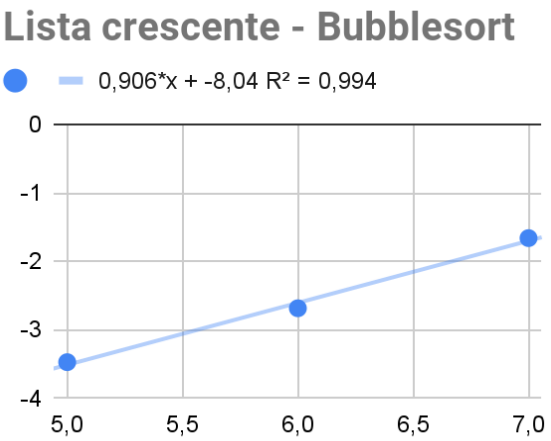
Seguindo a mesma linha de pensamento do algoritmo passado, o Bubblesort aprimorado deve apresentar o mesmo comportamento quadrático para todos os três casos, apenas com melhoras significativas para o melhor caso.

Conjunto tabelas e gráficos 2 - Análise empírica do Bubblesort aprimorado

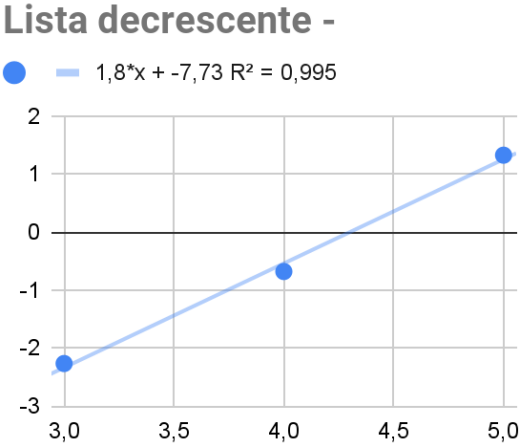
	BUBBLESORT APRIMORADO	
	Entradas	Tempo
Lista aleato-ria	1000	0,004338
	10000	0,1289112
	100000	27,1343096
	-	-
	-	-



	BUBBLESORT APRIMORADO	
	Entradas	Tempo
Lista crescente	-	-
	-	-
	100000	0,0003367
	1000000	0,0020637
	10000000	0,0218387



	BUBBLESORT APRIMORADO	
	Entradas	Tempo
Lista decres-cente	1000	0,005383
	10000	0,2078799
	100000	21,0948254
	-	-
	-	-

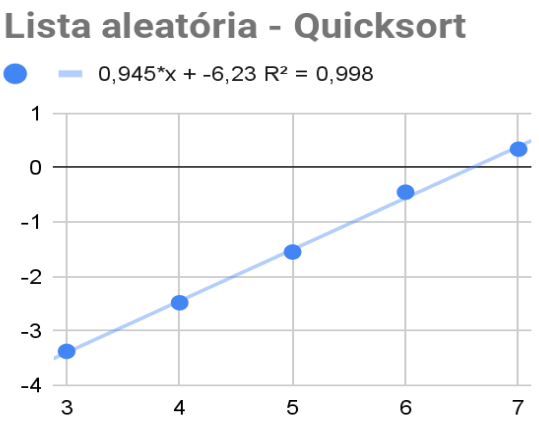


4.3. Quicksort

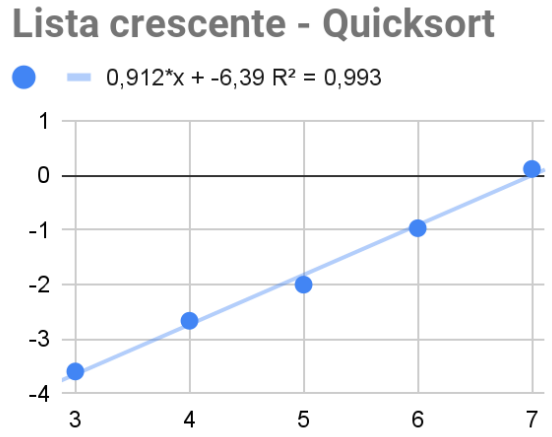
O "Quicksort" por sua vez, possui $O(n \log n)$ para melhor caso e caso médio, comportamento esse que em um gráfico log-log, resulta em uma reta com coeficiente de inclinação perto de 1, porém maior. Vale ressaltar que por diferenças externas ao rodar o código os coeficientes ficaram menores que 1, porém bastante coerente comparando com a ordem de grandeza da incerteza. Para o pior caso é esperado que o algoritmo apresente comportamento quadrático, $O(n^2)$, implicando em um custo computacional muito alto.

Conjunto tabelas e gráficos 3 - Análise empírica do Quicksort

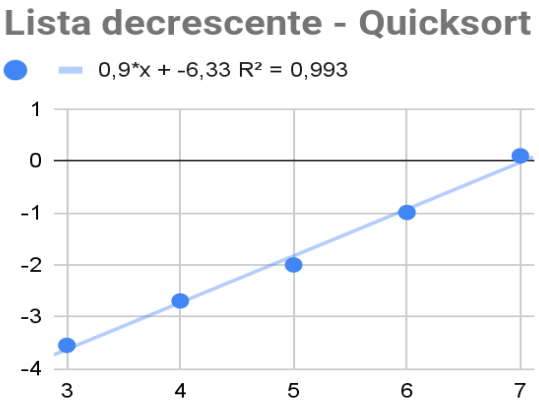
	QUICKSORT	
	Entradas	Tempo
Lista aleató-ria	1000	0,000424
	10000	0,0033077
	100000	0,0282420
	1000000	0,3549124
	10000000	2,1815803



	QUICKSORT	
	Entradas	Tempo
Lista crescente	1000	0,0002536
	10000	0,0021565
	100000	0,0099183
	1000000	0,1073732
	10000000	1,3096975



	QUICKSORT	
	Entradas	Tempo
Lista decres-cente	1000	0,0002799
	10000	0,0020034
	100000	0,0098978
	1000000	0,1013513
	10000000	1,2498290



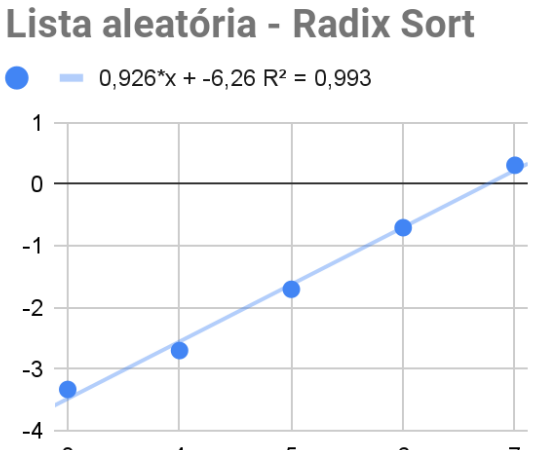
É de extrema importância ressaltar que o pior caso do quicksort não é o caso da lista decrescente, uma vez que ele não ordena o vetor de acordo com elementos "ao lado". Por isso, o comportamento do algoritmo na lista decrescente não foi quadrático.

4.4. Radixsort

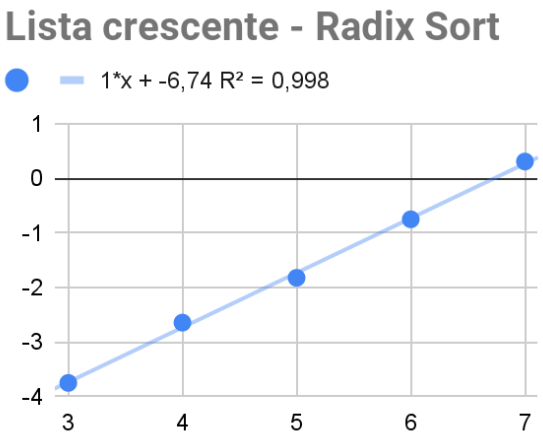
Para o "Radixsort" a análise de complexidade aponta para um $O(n)$, ou seja um comportamento linear, nesse caso as constantes que multiplicam o termo linear podem influenciar significamente o custo do código para entradas suficientemente grandes. Novamente, tem-se um coeficiente de inclinação da reta perto de 1, dentro de uma margem de erros coerente.

Conjunto tabelas e gráficos 4 - Análise empírica do Radixsort

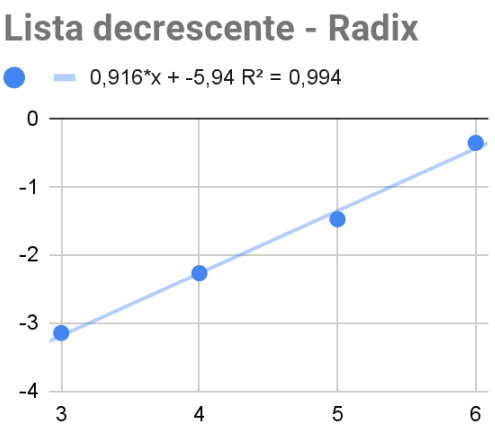
	RADIXSORT	
	Entradas	Tempo
Lista aleatória	1000	0,0004704
	10000	0,0020101
	100000	0,0197909
	1000000	0,1973914
	10000000	2,0289726



	RADIXSORT	
	Entradas	Tempo
Lista crescente	1000	0,0001775
	10000	0,0022643
	100000	0,0150114
	1000000	0,1775214
	10000000	2,0344768



	RADIXSORT	
	Entradas	Tempo
Lista decres-cente	1000	0,0007044
	10000	0,0053170
	100000	0,0328924
	1000000	0,4353556
	-	-

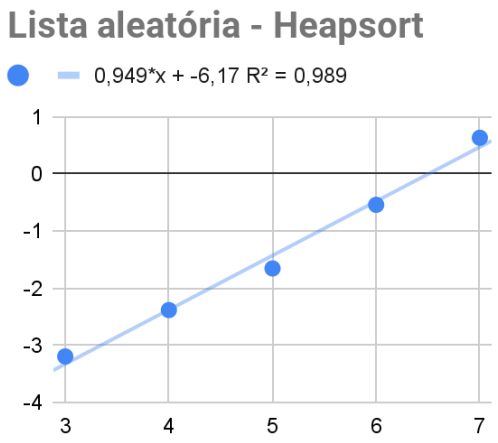


4.5. Heapsort

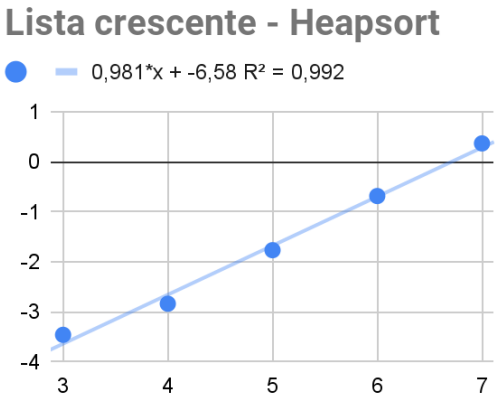
Por fim, a análise de complexidade do ”Heapsort” se aproxima significativamente do “Quicksort”, em ambos os algoritmos tem-se $O(n \log n)$. Contudo, o “Heapsort” não possui diferença para o pior caso, ao contrário do “Quicksort” que passa a ser quadrático. Além disso, outra diferença é que devido às constantes no termos de complexidade, o “Heapsort” é ligeiramente mais custoso que o “Quicksort”.

Conjunto tabelas e gráficos 5 - Análise empírica do Heapsort

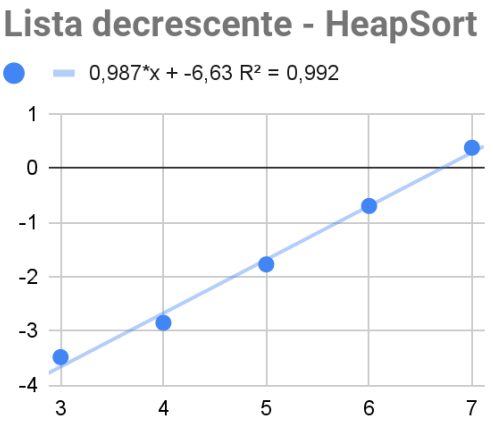
	HEAPSORT	
	Entradas	Tempo
Lista aleatória	1000	0,0006439
	10000	0,0041637
	100000	0,0222387
	1000000	0,2883054
	10000000	4,2829713



	HEAPSORT	
	Entradas	Tempo
Lista crescente	1000	0,0003462
	10000	0,0014501
	100000	0,0170607
	1000000	0,2066045
	10000000	2,3314329



	HEAPSORT	
	Entradas	Tempo
Lista decrescente	1000	0,0003263
	10000	0,0013985
	100000	0,0166951
	1000000	0,1994075
	10000000	2,3627542



5. Conclusão

A execução dos diferentes tipos de algoritmos sugeridos possibilitou construir conhecimentos práticos, em demasia nos de ordenação; tratando-se da arte de ordenar esses algarismos com diversas formas diferentes garante a existência da ciência da computação, uma vez que possibilitou a aplicação ampla desses conceitos, sob diversos pontos de vista e sob a influência de diferentes casos de ordenação.

Então, é justamente diante dessa perspectiva que se conclui acerca da importância da potencialidade do estudo de algoritmos. Por mais diversos que tenham se mostrado as análises tanto empíricas quanto teóricas, a diferença entre os estilos de ordenação sempre apresentou influência direta da ordem da notação de Big-O final.

Com isso, perante as incontáveis aplicações na vida cotidiana, a utilização de diversos algoritmos visando sempre a viabilidade para o melhor caso necessário é de extrema importância. Acerca destes princípios tais como a implementação dos códigos, a análise teórica e empírica dos mesmos, velocidade de processamento, tem sua importância fundamentada em quesitos que vão muito além do anseio de intelectualizar-se, estando intimamente ligados ao entendimento de inúmeros casos corriqueiros não só no âmbito da individualidade humana, mas também em relação ao escopo computacional.

Referências Bibliográficas

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 3rd edition, MIT Press, 2009. Disponível em: <https://mitpress.mit.edu/books/introduction-algorithms-third-edition>. Acesso: 06/06/2021

[2] BAASE, Sara (1988). Computer Algorithms. Introduction to Design and Analysis (em inglês) 2ª ed. Reading, Massachusetts: Addison-Wesley. Acesso em: 06/06/2021