Análise sobre Algoritmos de Ordenação Interna

Rafael Campos Nunes

8 de abril de 2018

Resumo

Este trabalho tem como intento apresentar o significado de algoritmo utilizado em dispositivos computacionais e colocar em destaque o estudo de algoritmos de ordenação interna. Esse estudo será feito utilizando dois parâmetros: sendo o primeiro tamanho da entrada e o segundo o quão desordenada está dessa entrada. Por fim, este estudo apresenta uma análise global sobre os algoritmos de ordenação evidenciando os melhores e os piores algoritmos em cada ponto de análise.

Conteúdo

Parte I

Introdução

Atualmente se emprega a utilização da palavra *algoritmos* para denotar um conjunto de passos para se realizar uma tarefa. Embora tal definição esteja correta, essa definição não é suficientemente rígida para os campos de estudo exatos.

Um desses campos, a ciência da computação, utiliza-se do conceito de algoritmo para definir instruções computacionais, isto é, instruções realizadas em um dispositivo capaz de realizar cálculos tal qual é feito à mão. Pela natureza dos computadores, esses algoritmos não podem ser descritos da mesma maneira como são descritas receitas de bolo, a exemplo.

Os algoritmos computacionais são, portanto, nada mais que um conjunto de instruções para um fim, de forma que tais instruções são descritas a um nível de precisão necessário para um computar executá-las. A partir dessa definição pode-se observar outro aspecto de algoritmos: a recorrência com que estão presentes no século XXI.

É notória a utilização em grande escala de computadores, tablets e smartphones pelo sociedade atual, contudo, os algoritmos que realizam as diversas tarefas desses dispositivos são invisíveis aos olhos do utilizador. Tais algoritmos são objeto de estudo nesse relatório, mais especificamente, os que tangem a ordenção de dados.

Os algoritmos de ordenação tem como finalidade a ordenação de dados, isto é, dada uma entrada de dados, chamada doravante de *instância*, ele reorganiza os dados de maneira que obedeçam uma característica definida *a priori* e retorna uma instância com os respectivos dados ordenados.

Parte II

Referencial teórico

Informalmente um algoritmo é qualquer procedimento computacional bem definido que tomando como entrada um conjunto de dados A transforma tal conjunto em um conjunto de dados B, sendo este considerado o resultado ou saída de um algoritmo. Também pode-se considerar um algoritmo como uma ferramenta para resolver um problema computacional bem especificado[1].

Além de um algoritmo ser uma ferramenta com propósito excertado no parágrafo anterior, ele também pode ser definido como tecnologia. Isso pode ser explicado colocando os diversos tipos de algoritmos diferentes para resolver uma mesma tarefa, a partir disso pode se observar a diferença de performance, tanto de tempo de execução e utilização de recursos como a memória do dispositivo computacional. Um algoritmo eficiente utilizará esses recursos, que são finitos, eficientemente, ou seja, fazendo o melhor aproveitamento tanto da unidade de processamento central como da memória primária ou de armazenamento em massa[2].

Considerando os problemas de ordenação de dados têm-se os algoritmos de ordenação cujo propósito é reorganizar os elementos de uma instância sob uma restrição definida pelo programador. Esses, por sua vez, podem ser categorizados em dois grupos: os algoritmos de ordenação interna e os algoritmos de ordenação externa. O primeiro tange os algoritmos que se utilizam exclusivamente da memória primária do computador a memória de acesso randômico (RAM) e serão objetos de análise neste relatório. A segunda categoria abarca os algoritmos que utilizam, além da memória primária, a memória de armazenamento em massa. Este fato decorre da limitação física de armazenar todos os dados a serem ordenados na RAM.

Tomando os dois grupos de algoritmos de ordenação tem-se, ainda, outras características que são inerentes a esses como a estabilidade ou não estabilidade do algoritmo e o método utilizado para a ordenação. A propriedade de estabilidade de um algoritmo confere a este a não troca de posição de chaves cujos valores são iguais, ou seja, a posição relativa dessas chaves é mantida. O método de ordenação descreve como o algoritmo rearranja os elementos,

[1] CORMEN et al. Algoritmos p.3
[2] CORMEN et al. Algoritmos p.7-9

seja por partição, mesclagem, seleção ou troca.

Por fim, ao lado dos dois grupos expostos anteriormente, é válido ressaltar algumas estratégias que são extensivamente utilizadas para resolver, da melhor maneira, problemas de ordenação, mas não limitado a eles. Elas compreendem a forma de como os algoritmos resolvem o problema, e são: estratégia gulosa, programação dinâmica e divisão e conquista. Segundo Toscani e Veloso, a primeira estratégia é útil para resolução de problemas de otimização combinatória cujas soluções podem ser alcançadas por meio de sequências de decisões. A segunda é aplicada a problemas que não é fácil se chegar a uma sequência ótima de decisões sem testar as várias sequências possíveis e a terceira estratégia é a decomposição de um problema em várias partes para então resolver as várias partes e, por fim, recombinar as partes obtendo a solução do problema[3].

[3]TOSCANI, VELOSO. Complexidade de Algoritmos p.188-190

Capítulo 1

Algoritmos quadráticos

Algoritmos quadráticos são, a rigor, algoritmos onde a grandeza assintótica tem crescimento proporcional a n^2, dessa forma pode-se observar o melhor, médio e pior caso desses algoritmos com pelo menos um deles contendo essa grandeza.

É importante salientar que, embora a grandeza assintótica desses algoritmos sejam iguais, eles não necessariamente tem a mesma performance quando colocados em execução sobre a mesma quantidade de dados, ou seja, sobre uma instância de mesmo tamanho. Isto é consequência do fato de que a notação assintótica despreza termos absolutos em seu cálculo.

Portanto, para saber exatamente qual de dois algoritmos com mesma ordem de grandeza assintótica é melhor, é necessário realizar o cálculo em termos absolutos com relação a trocas ou deslocamentos de elementos feitos por estes.

1.1 Bubble Sort

O Bubblesort é um algoritmo de ordenação estável e comparativo que realiza a trocas de chaves adjacentes à posição de ordenação de acordo com o valor dessas.

Esse algoritmo é considerado o mais lento em termos de execução pois sua execução é afetada por dois motivos: a quantidade excessiva de compa-

rações e de trocas necessárias. Tais motivos conferem o título de algorimo de ordenação mais lento dentre os destacados nesse trabalho.

A seguir, pode-se visualizar o código em C++ de sua implementação. O algoritmo itera sobre a instância de tamanho n por, no mínimo, n^2 vezes.

```
1  void bubble(std::vector<int> instancia) {
2      bool swapped;
3
4      do {
5          swapped = false;
6
7          for (std::size_t j = 1; j < instance.size(); j++) {
8              if (instance[j] < instance[j-1]) {
9                  std::swap(instance[j], instance[j-1]);
10                 swapped = true;
11             }
12         }
13     } while (swapped != false);
14
15     return instance;
16 }
```

Código 1: Bubble Sort em C++11

O cálculo da complexidade é de fácil apreensão. Basta observar que os dois laços tem uma quantidade de operações parecidas sendo que, o primeiro opera n vezes e o segundo opera, para cada, iteração do primeiro, $n-1$ vezes. Como consequência, o cálculo de $T(n)$ é, pelo princípio multiplicativo:

$$T(n) = n * (n-1) = n^2 - n$$

Ademais, é sabido que, para o *Bubble Sort* o médio e pior caso é uma ordem de grandeza n^2, sendo o melhor caso com uma ordem de grandeza linear.

Para uma instância de tamanho 5 e contendo os números *<5, 4, 3, 2, 1>* e dispondo esses elementos em uma tabela, o algoritmo se comporta da seguinte maneira:

Tabela 1.1: Teste de mesa do algoritmo Bubble Sort

Execução	Instância
0	**5 4** 3 2 1
1	4 **5 3** 2 1
2	4 3 **5 2** 1
3	4 3 2 **5 1**
4	**4 3** 2 1 5
5	3 **4 2** 1 5
.	...
.	...
.	...
7	3 2 1 4 5
.	...
.	...
.	...
14	1 2 3 4 5

Esse processo de troca entre elementos adjacentes se repete até que a instância esteja ordenada.

1.2 Selection Sort

O algoritmo *Selection Sort* é um algoritmo de seleção não estável, *in-place* e, apesar de compartilhar a mesma complexidade quadrática do *Bubble Sort*, esse por sua vez faz menos comparações em termos absolutos. Sendo assim, o *Selection Sort* é um algoritmo mais eficiente que o *Bubble Sort*.

Abaixo, segue o código do algoritmo em C++:

```cpp
void selection(std::vector<int> instancia) {
    for (int i = 0; i < instancia.size(); i++) {
        int min = i;
        int aux = instancia[i];

        for (int j = 1; j < instancia.size(); j++) {
            if (instancia[j] < instancia[min]) {
                min = j;
            }
        }

        std::swap(instancia[min], instancia[i]);
    }
}
```

Código 2: Selection Sort em C++11

O algoritmo sempre seleciona o índice do menor elemento do vetor e o troca com o elemento na i-ésima posição ao final da iteração. Esse processo se repete até o fim da execução do algoritmo, onde a instância estará toda ordenada.

A complexidade do algoritmo pode ser calculada tomando a quantidade de vezes que os dois laços são executados. Para o pior caso, isto é, quando você tem a instância de entrada ordenada de maneira decrescente, os laços farão o número máximo de iterações. Sendo assim, o primeiro laço fará n iterações e o segundo fará $n-1$ iterações para cada iteração do primeiro laço.

Portanto, a complexidade do algoritmo em seu pior caso pode ser calculada utilizando-se, novamente, do princípio multiplicativo:

$$T(n) = n * (n-1) = n^2 - n$$

Analisando assintoticamente, ou seja, para n muito grandes remove-se as constantes e termos de menor ordem[1], resultando em $\mathcal{O}(n^2)$.

Analisando a quantidade de iterações do algoritmo em uma instância de tamanho 5 e contendo os números *<5, 4, 3, 2, 1>*, dispondo esses elementos

[1] TOSCANI, VELOSO. Complexidade de Algoritmos p.24-27

em uma tabela, o algoritmo se comporta da seguinte maneira:

Tabela 1.2: Teste de mesa do algoritmo Bubble Sort

Execução	Instância
0	**5** 4 3 2 **1**
1	1 **4** 3 **2** 5
2	1 2 **3** 4 5
3	1 2 3 4 5

Observa-se que, pela tabela acima, o fato desse algoritmo não fazer demasiadas comparações e inserir a característica seletiva reduz bastante a quantidade de iterações necessárias para se ordenar o mesmo vetor.

1.3 Insertion Sort

O algoritmo *Insertion Sort*, também conhecido como algoritmo de ordenação de cartas, é estável, *in-place* e o método utilizado para ordenar é a inserção. Esse algoritmo é mais eficiente que o *Bubble Sort* e mais eficiente que o *Selection Sort* em seu melhor caso.

O código em C++ abaixo descreve a implementação do *Insertion Sort*:

```
1  void insertion(std::vector<int> instancia) {
2      for (int i = 1; i < instancia.size(); i++) {
3          int aux = instancia[i];
4          int j = i-1;
5
6          while (j >= 0 && aux > instancia[j]) {
7               instancia[j+1] = instancia[j];
8               j--;
9          }
10
11         instancia[j+1] = aux;
12     }
13 }
```

Código 3: Insertion Sort em C++11

O algoritmo primeiro salva o elemento que está localizado na *i-ésima* posição e após isso faz o deslocamento de todos os elementos que forem menores que o localizado na *i-ésima* posição. Ao final de cada iteração, insere o elemento na posição adjacente a direita da *j-ésima* posição.

Atribuindo valor de v à instância e o valor salvo na *i-ésima* posição de aux pode se descrever o seguinte conjunto de passos para esse algoritmo:

1. Guardar o valor na i-ésima posição;
2. Deslocar os elementos da j-ésima posição à direita enquanto $aux \leq v[j]$ e $j \geq 0$;
3. Inserir aux na j-ésima posição adjacente a direita;
4. Volte ao passo 1.

A complexidade do algoritmo nesse caso é calculada observando a quantidade de vezes que os dois laços são executados. Considerando o pior caso, já explicado nos algoritmos anteriores, é possível observar duas coisas:

1. O primeiro laço é executado n vezes
2. O segundo laço é executado $1, 2, ..., n-1$ vezes

O segundo fato é compreendido ao atentar-se que o valor de j é sempre $i-1$ e como ele tem uma restrição no segundo laço ele é executado $1, 2, ..., n-1$ vezes. Deste fato se apreende que o segundo laço é na verdade uma progressão aritmética onde a soma de seus termos pode ser expressa como:

$$S = 1, 2, 3, ..., n-1 \rightarrow S = \sum_{i=1}^{n-1} i$$

A equação que descreve a soma destes termos é:

$$S = \frac{n * (1 + n - 1)}{2} \tag{1.1}$$

Por fim, adicionando ao cálculo final a complexidade dos dois laços de iteração resulta em:

$$T(n) = n + \frac{n * (1 + n - 1)}{2} \rightarrow n + \frac{n^2}{2}$$

$$T(n) = \frac{2n + n^2}{2} \tag{1.2}$$

Dado o cálculo de complexidade em tempo do *Insertion Sort* da equação 1.2 faz-se a análise assintótica nesse, obtendo o pior caso assintótico de $\mathcal{O}(n^2)$ após descartar constantes e variáveis de menor ordem.

1.4 Shell Sort

O algoritmo *Shell Sort* é um algoritmo conhecido por ser uma versão do *Insertion Sort* com a adição de *gaps*, conferindo ao algoritmo a posição de mais eficiente dentre os algoritmos de complexidade quadrática. A ordenação utiliza o método da inserção - por ser derivado do *Insertion Sort* - e é feita *in-place*.

Essa ordenação compara pares de valores distantes, sendo essa distância denominada por *gap* contendo uma amplitude de pelo menos 2. Ao passo da execução diminui-se gradativamente essa distância por um fator do *gap* a fim de completar a ordenação quando o $gap \leq 1$.

O *Shell Sort* é um algoritmo com pequeno tamanho de código, não utiliza a pilha de chamadas de função, ou seja, não é recursivo e é razoavelmente rápido onde a memória em certos dispositivos, principalmente embarcados, é um item de luxo. Abaixo pode ser visto a implementação do algoritmo como descrita no parágrafo anterior.

```
1  void shell(std::vector<int> &v) {
2
3      int gap = 1;
4      int i, j;
5
6      while (gap < v.size()) {
7          gap = gap*3+1;
8      }
9
10     while (gap > 1) {
11         gap /= 3;
12
13         for (i = gap; i < v.size(); i++) {
14             int aux = v[i];
15             j = i;
16
17             while (j >= gap && aux < v[j-gap]) {
18                 v[j] = v[j-gap];
19                 j -= gap;
20             }
21
22             v[j] = aux;
23         }
24     }
25 }
```

Código 4: Shell Sort em C++

O código primeiro calula um intervalo com a expressão denotada na linha 7 até que este seja maior que *v.size()*. Após isto, entra no loop com a condição de que o gap tem de ser maior que um. A ordenação ocorre com a comparação e inserção dos elementos na posição correta utilizando os intervalos de *gap*.

Tomando uma instância de cinco elementos, sendo eles $< 5, 4, 3, 2, 1 >$ a ordenação ocorre da seguinte maneira:

1. Calcula se o *gap* inicial, sendo este correspondente a 13;
2. Entrando no loop de ordenação o *gap* é diminuído por um fator de 3 resultando em 4

Após o cálculo final do intervalo inicial começa-se a ordenar os números. Abaixo pode-se visto uma tabela com a instância e os respectivos números pertencentes aos intervalos destacados em negrito

Tabela 1.3: Ordenação por Shell Sort

Passo da execução	Instância	Intervalo
0	5, 4, 3, 2, **1**	4
1	5, 4, 3, 2, 1	1

Após o intervalo ser igual a um o algoritmo funciona exatamente da mesma maneira que o *Insertion Sort*.

Atendando-se para complexidade do algoritmo é necessário observar uma propriedade muito importante deste: o cálculo do *gap*. Esse cálculo influencia na complexidade do pior caso do algoritmo. A título de exemplo, encontra-se abaixo o cálculo de *gaps* e suas respectivas complexidades de pior caso.

Tabela 1.4: Complexidades de pior caso relativas ao cálculo do *gap*

Cálculo do gap	Complexidade em pior caso
$\frac{n}{2^k}$	$\Theta(n^2)$
$2^k + 1$	$\Theta(n^{3/2})$
$2 \cdot [\frac{n}{2^{k+1}}] + 1$	$\Theta(n^{3/2})$

Contudo, existem cálculos que demonstram os melhores limites superiores e inferiores existentes. O melhor já encontrado, pois este não depende do intervalo calculado, é proporcional a $n \cdot log(n)$ e o pior caso, limite inferior, encontrado foi o $\mathcal{O}(n^{\frac{4}{3}})$ [2].

[2]SEDGEWICK. Journal of Algorithms p.7

Capítulo 2

Algoritmos logaritmicos

Algoritmos logaritmicos ou linearmente logaritmicos são denotados por ordens de grandeza assintótica logaritmica, isto é, seu crescimento é proporcional ao crescimento de $log(n)$ ou $n \cdot log(n)$, sendo este se estivermos nos referindo ao crescimento dos linearmente logaritmicos.

2.1 Quick Sort

Considerado um algoritmo de ordenação eficiente que, embora o pior caso seja $\mathcal{O}(n^2)$, quando o pivô é bem escolhido pode ser de duas a três vezes mais rápido do que o *Heap Sort* ou o *Merge Sort*.

O algoritmo particiona a instância com relação ao pivô escolhido classificando suas partes e depois concatenando as partes classificadas. É interessante ressaltar que o melhor pivô é aquele que divide a partição de tal forma que as duas partes em tamanho sejam iguais ou que sua diferença seja no máximo de um. A complexidade do algoritmo não é influenciada diretamente pelo cálculo do pivô como no caso do *Shell Sort*. Contudo, a escolha pode fazer o algoritmo se enquadrar no pior caso.

O funcionamento do *Quick Sort* pode ser definido pelos seguintes passos:

1. Selecione qualquer elemento da instância como pivô

2. Divida todos os outros elementos exceto o pivô (deve atender a duas

restrições) Todos os elementos à esquerda do pivô devem ser menor que este Todos os elementos à direita do pivô devem ser maior que este

3. Usar recursão para ordenar as duas partições
4. Mesclar a primeira partição ordenada, o pivô, e a segunda partição ordenada

O terceiro e quarto passo são transparentes e tão triviais que não são explicitamente citados mesmo o algoritmo sendo de divisão e conquista[1]. O excerto abaixo concerne o algoritmo de divisão recursiva utilizado pelo *Quick Sort*.

```
void quick(std::vector<int> &v, int l, int r) {

    std::pair<int, int> p = partition(v, l, r);

    if (l < p.second) {
        quick(v, l, p.second);
    }

    if (p.first < r) {
        quick(v, p.first, r);
    }
}
```

Código 5: Função divide e seleciona as partições para ordenação

O algoritmo abaixo denota como é feita a ordenação dado uma instância, uma posição inicial (l) e uma posição final (r).

[1]Rosetta Code on Quick sort.

```cpp
std::pair<int, int> partition(std::vector<int> &v, int l,
                                            int r) {

    std::pair<int, int> pair;

    pair.first = l;
    pair.second = r;

    int pivot = v[(pair.first+pair.second) >> 1];

    do {
        while (v[pair.first] < pivot) pair.first++;
        while (v[pair.second] > pivot) pair.second--;

        if (pair.first <= pair.second) {
            std::swap(v[pair.first], v[pair.second]);
            pair.first++;
            pair.second--;
        }

    } while (pair.first <= pair.second);

    return pair;
}
```

Código 6: Função que ordena a partição dada entre l e r

A complexidade do algoritmo como já foi citada é linear logaritmica no melhor e médio caso, sendo limitada por uma cota assintótica quadrática em seu pior caso ($\mathcal{O}(n^2)$).

2.2 Heap Sort

O *Heap Sort* é um algoritmo de ordenação não estável, de comparação que utiliza o método de seleção para ordenar os elementos de um vetor, convém descrever que tal ordenação é feita *in-place* como outros algoritmos já expostos. Esse, tem vantagem em relação ao *Quick Sort* por ter um pior

caso limitado por $\mathcal{O}(n \cdot log(n))$, contudo, ainda é mais lento que este na maioria dos casos.

O algoritmo utiliza-se da estrutura de dados *heap* para organizar seus elementos minimizando o tempo de acesso e remoção ao máximo/mínimo elemento, tal fator pode ser pensado como o algoritmo de seleção com a estrutura de dados certa[2].

O *heap* é uma estrutura de dados que pode ser visto como uma árvore binária quase completa que satisfaz uma propriedade[3]. A propriedade de que o pai na i-ésima posição do vetor, este indexado por 0, é sempre maior se se somente se o *heap* for um *heap* de máximo no qual os elementos da posição $2 \cdot i + 1$ e $2 \cdot i + 2$ são menores que o pai.

Dada a propriedade de *heap* máximo no parágrafo anterior, observa-se que tais operações são realizadas em tempo constante no computador, ainda mais rápidas se utilizadas as instruções de deslocamento de bits, onde fazem tal cálculo em uma única instrução assembly[4]. Este fato pode ser visto ao tomar um nó na posição i e uma raíz indexada por 1, onde decorrem as seguintes operações para encontrar o *pai* o *filho à esquerda* e o *filho à direita* do nó especificado:

1. $pai = i >> 1$
2. $filho_{esq} = i << 1$
3. $filho_{dir} = (i << 1) + 1$

Ao construir o *heap* atenta-se às propriedades expostas e pode ser feita pelo seguinte código, denominado de *heapify*:

[2]SKIENA. Sorting and Searching p.109
[3]CORMEN et al. Algoritmos p.110
[4]CORMEN et al. Algoritmos p.111

```cpp
void heapify(std::vector<int> &v, int n, int i) {

    int largest = i;

    // position of the sons of the node at v[i]
    int l = 2*i+1;
    int r = 2*(i+1);

    if (l < n && v[l] > v[largest]) {
        largest = l;
    }

    if (r < n && v[r] > v[largest]) {
        largest = r;
    }

    if (largest != i) {
        std::swap(v[i], v[largest]);
        heapify(v, n, largest);
    }
}
```

Código 7: Código da construção de uma heap em C++

O código acima constrói uma estrutura *heap* de máximo, isto é, que obedeça a propriedade já exposta. Observa-se que há uma verificação com os elementos $filho_{esq}$ e $filho_{dir}$ presentes nos índices l e r, após essa verificação a última asserta que se foi encontrado um elemento maior que o presente no antigo índice i e se for esse o caso, troca-o com a posição do maior elemento, chamando a mesma função pois a propriedade do *heap* foi alterada.

Contudo, para a construção efetiva do *heap* a função *heapify* deve ser chamada da seguinte maneira:

```cpp
for (int i = n/2-1; i >= 0; i--) {
    heapify(v, n, i);
}
```

Código 8: Código que executa a construção do *heap*

O *heap* é a sequência construída do elemento $n/2-1$ da instância, pois é garantido o elemento no índice em questão tenha uma das operações $2*i+1$ ou $2*i+2$ satisfeitas.

Após a asserção das condições e o *heap* construído basta ordenar, trocando a posição do elemento na primeira posição com o da última posição da instância, chamando logo em seguida a função *heapify* para atestar a propriedade de *heap* se for necessário.

```
for (int i = n - 1; i >= 0; i--) {
    std::swap(v[0], v[i]);
    heapify(v, i, 0);
}
```

Código 9: Código que ordena um *heap*

A complexidade do algoritmo desta seção é, em seu pior caso $\mathcal{O}(n \cdot log(n))$ pois a construção do *heap* implica uma complexidade proporcional a $log(n)$ e para a ordenação é observado que o algoritmo estará limitado superiormente por um $n + n \cdot log(n)$ dado que, na recorrência de ordenação presente no último código, para cada iteração do laço chama-se *heapify* que por sua vez é $log(n)$.

2.3 Merge Sort

O *Merge Sort* é um algoritmo estável, de natureza comparativa e não realiza a ordenação *in-place*, devendo tomar um espaço auxiliar proporcional a n ao fazê-la.

O algoritmo realiza a ordenção utilizando-se de uma estratégia parecida ao *Quick Sort*, contudo, a escolha da partição é dada sempre ao meio da instância. Além disso, o algoritmo tem natureza recursiva e faz divisões subsequentes até sobrar o próprio elemento como partição para que, ao mesclar, os valores serem comparados e colocados em suas devidas posições.

O código abaixo ilustra o funcionmento do particionamento da instância e as chamadas recursivas.

```
void merge(std::vector<int> &v, std::size_t l, std::size_t r) {

    if (l < r) {
        std::size_t p = (l+r) >> 1;

        merge(v, l, p);
        merge(v, p+1, r);
        merge_partition(v, l, p, r);
    }
}
```

Código 10: Função recursiva que divide a instância para sucessiva ordenação

A função *merge_partition* mescla as instâncias divididas até que a instância esteja toda ordenada e ao fazê-la o algoritmo aloca um espaço equivalente a $r - l + 1$.

Após a definição do tamanho da instância, o algoritmo compara os valores e ordena estes em uma instância de tamanho já definido que será utilizada para copiar os valores ao vetor original, chamada doravante de instância auxiliar. Entretanto, no meio desse processo alguns números do vetor original ficam faltando, sendo necessário a inclusão destes por meio de dois outros laços, denotados por copiar os valores das posições *lbegin* até p e de *rbegin* até r. Finalmente após esse preenchimento com os valores da instância original insere-se nesta os elementos que estão na instância auxiliar completando a ordenação.

O código abaixo excerta todas as proposições, de execução do algoritmo, do parágrafo anterior.

```cpp
void merge_partition(std::vector<int> &v, std::size_t l, std::size_t p,
                     std::size_t r) {

    std::size_t lbegin = l;
    std::size_t rbegin = p+1;

    std::size_t size = r-l+1;

    std::vector<int> aux;
    aux.resize(size);
    std::size_t aux_i = 0;

    while (lbegin <= p && rbegin <= r) {
        if (v[lbegin] < v[rbegin]) {
            aux[aux_i] = v[lbegin];
            lbegin++;
        } else {
            aux[aux_i] = v[rbegin];
            rbegin++;
        }

        aux_i++;
    }

    while (lbegin <= p) {
        aux[aux_i] = v[lbegin];
        lbegin++;
        aux_i++;
    }

    while (rbegin <= r) {
        aux[aux_i] = v[rbegin];
        rbegin++;
        aux_i++;
    }

    for (aux_i = l; aux_i <= r; aux_i++) {
        v[aux_i] = aux[aux_i-l];
    }
}
```

Código 11: Função que ordena e mescla os elementos da instância

A complexidade do *Merge Sort* é eficaz pois é proporcional a $n \cdot log(n)$ em todos os casos estudados. Contudo, o algoritmo deixa a desejar quanto a utilização de memória dado que ele é um algoritmo que não faz ordenação *in-place*.

Parte III

Metodologia

O presente trabalho utiliza-se da linguagem de programação C++ (padrão *c++11*) para desenvolver os algoritmos de ordenação e calcular o tempo de execução desses. O compilador utilizado é *g++* (GNU C++ Compiler), na versão 5.4.0, que será acionado por um *Makefile* para compilar todos os algoritmos em um executável para executar os testes. Tais testes são aplicados a cada algoritmo compilado, com instâncias de tamanho e ordem especificado pelo professor, onde cada algoritmo gerará um arquivo *comma separated values* (CSV) contendo o tempo de execução do algoritmo para instância de tamanho n. Tais informações são utilizadas para a geração posterior dos gráficos indicando, empiricamente, a ordem de grandeza do algoritmo testado.

Após ter os dados do tempo de execução de cada algoritmo em formato CSV, utiliza-se a ferramenta *Mathematica* para importar, gerar os gráficos e exportar as imagens respectivas a cada CSV.

Os algoritmos de ordenação serão executados em uma máquina com as seguintes especificações:

- Processador: Intel i3-2350M (2.30GHz)
- Processador de vídeo: NVIDIA GTX630M
- Memória primária: 12GiB

Para gerar uma imagem a partir de dados CSV na plataforma *Mathematica* o seguinte código é utilizado:

```
1 bubble = Import["bubble-decrescent.csv", "Data"]
2 Image[ListLinePlot[bubble, PlotRange -> All,
3                    AxesLabel -> {"tamanho da entrada(n)", "tempo(ms)"}],
4       ImageSize -> Large]
```

Código 12: Código em Mathematica importando um arquivo e plotando seu gráfico

Para algoritmos de ordem não quadrática utilizou-se uma outra função ao fazer o plot do gráfico, tal função pode ser vista abaixo.

```
ListLogPlot[heap, PlotRange -> All,
             AxesLabel -> {"tamanho da entrada(n)", "tempo(ms)"}]
```

Código 13: Código em Mathematica desenhando um gráfico linear logaritmico

Parte IV

Resultados e conclusão

Após a realização dos testes constatou-se que o binário gerado pelo código é de pequeno impacto na performance do computador, sendo que, mesmo ao carregar inicialmente três vetores, cada um com cem mil posições de inteiros, a memória utilizada por este foi de somente $2.3MiB$ em seu auge de execução. É válido ressaltar que em um ambiente *Linux 4.13.0-38-generic*, sob o hardware já especificado, foi possível notar que o processo do programa utilizava uma parcela entre 24% e 26% do processador.

Após isso, verificou-se empiricamente que os algoritmos obedecem à fundamentação teórica que excerta a complexidade desses algoritmos em seus variados casos. A tabela abaixo mostra a complexidade de tais algoritmos.

Tabela 2.1: Complexidade de tempo dos algoritmos estudados no relatório

Algoritmo	Melhor caso Ω	Médio caso Θ	Pior caso $\mathcal{O}$
Bubble	n	n^2	n^2
Selection	n^2	n^2	n^2
Insertion	n	n^2	n^2
Shell	Depende do *gap*	Depende do *gap*	$n \cdot log(n)$
Quick	$n \cdot log(n)$	$n \cdot log(n)$	n^2
Heap	$n \cdot log(n)$	$n \cdot log(n)$	$n \cdot log(n)$
Merge	$n \cdot log(n)$	$n \cdot log(n)$	$n \cdot log(n)$

Para uma melhor compreensão dos efeitos dessas grandezas dispõe se tais complexidades na figura 2.1 .

Pode-se verificar, colocando todos os algoritmos em análise, que os algoritmos de complexidade quadrática ganham destaque no gráfico. Sendo esse um motivo que até ofusca a visualização dos outros. Tal afirmação é observada no gráfico abaixo, montado sobre a coleta de tempo de execução dos algoritmos em função do tamanho da entrada.

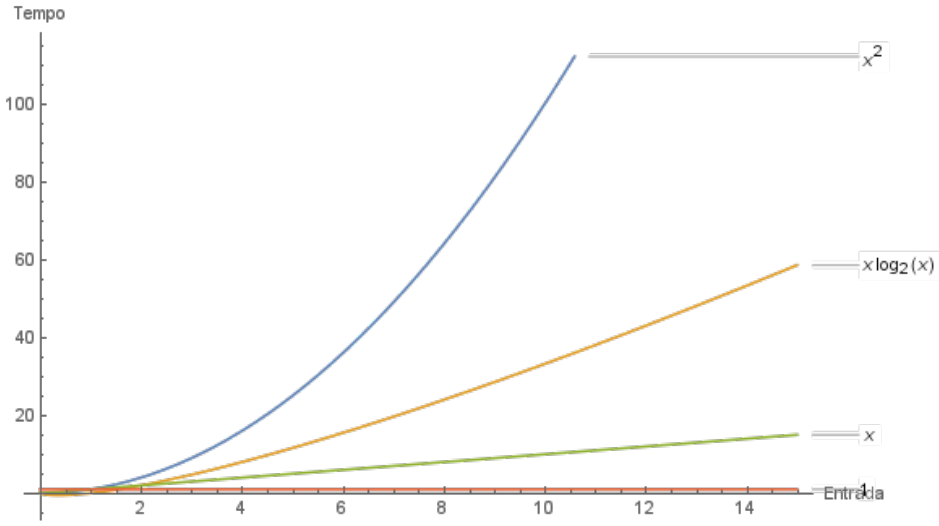

Figura 2.1: Gráfico mostrando ordens de grandeza mais comuns de algoritmos

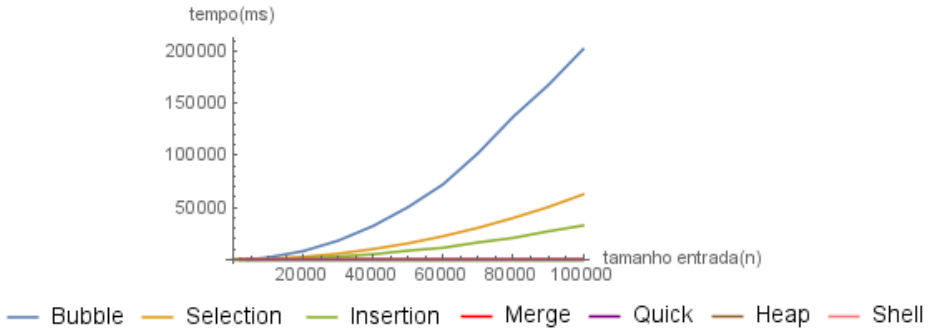

Figura 2.2: Performance de todos os algoritmos em dados aleatórios

Capítulo 3

Análise dos algoritmos quadráticos

Os algoritmos de complexidade quadrática se mostraram ineficientes como consequência os mais lentos, como esperado dada a teoria existente. Abaixo seguem-se as análises contemplando todos os algoritmos em suas diversas formas de ordenação.

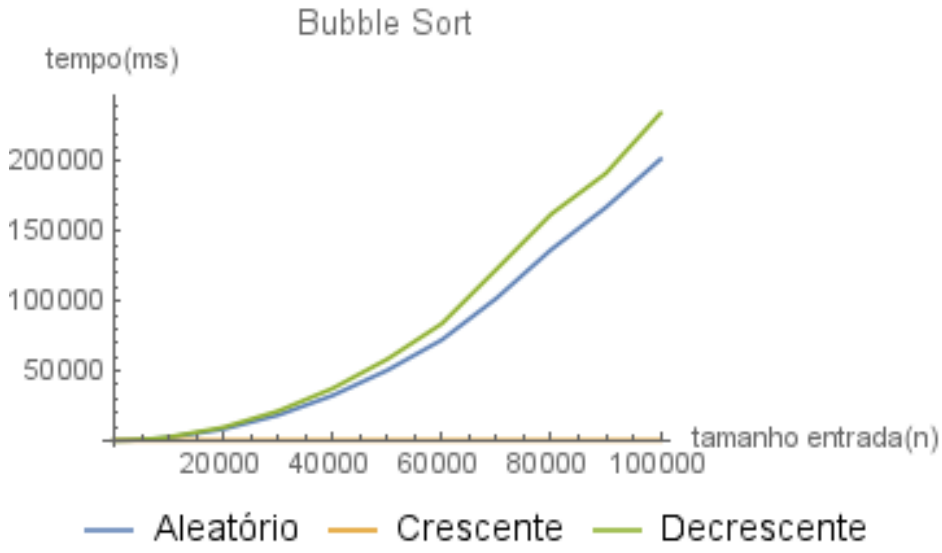

Figura 3.1: Performance do Bubble Sort

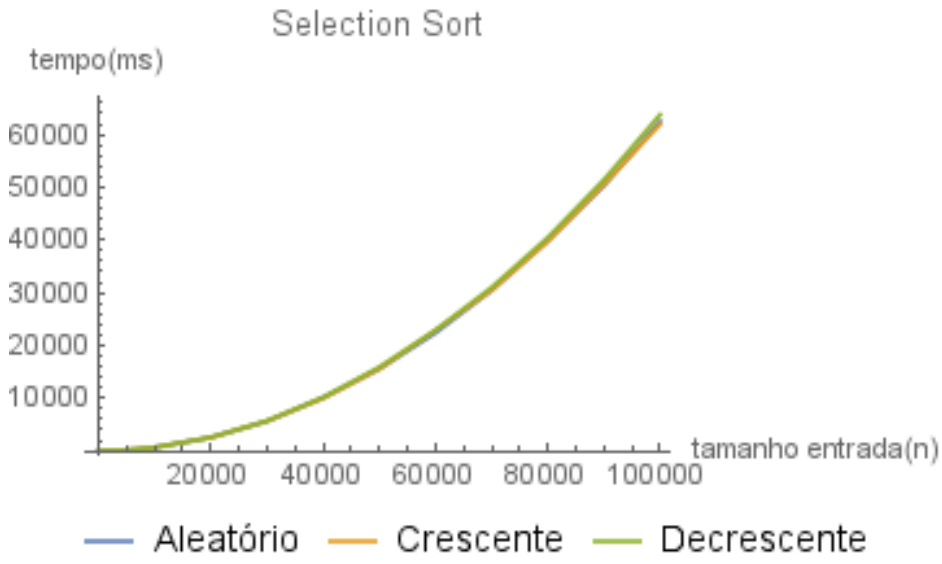

Figura 3.2: Performance do Selection Sort

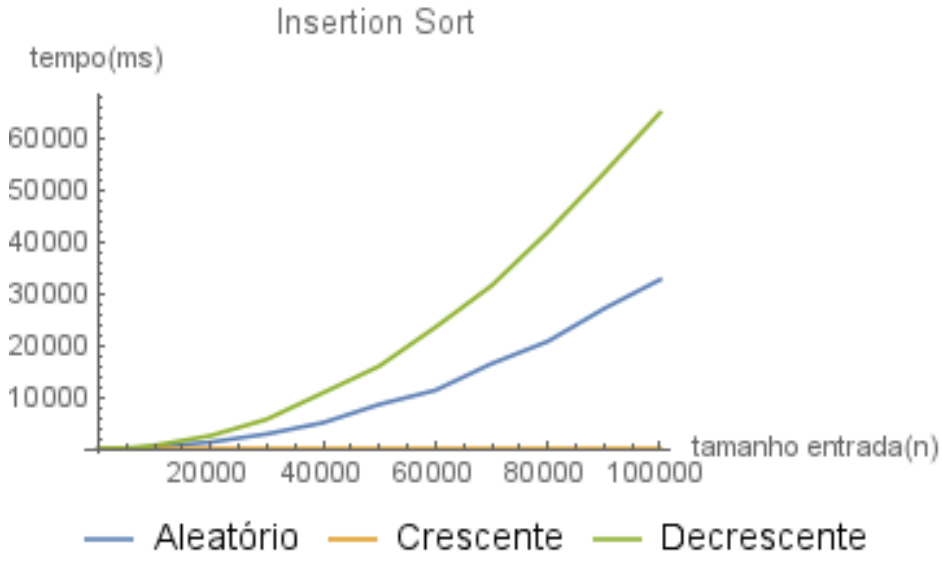

Figura 3.3: Performance do Insertion Sort

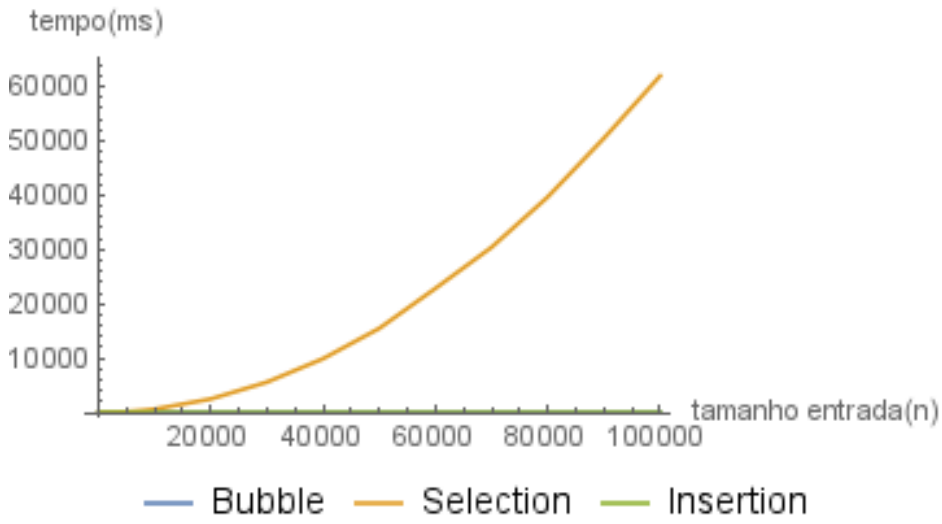

Figura 3.4: Performance dos algoritmos quadráticos em dados crescentes

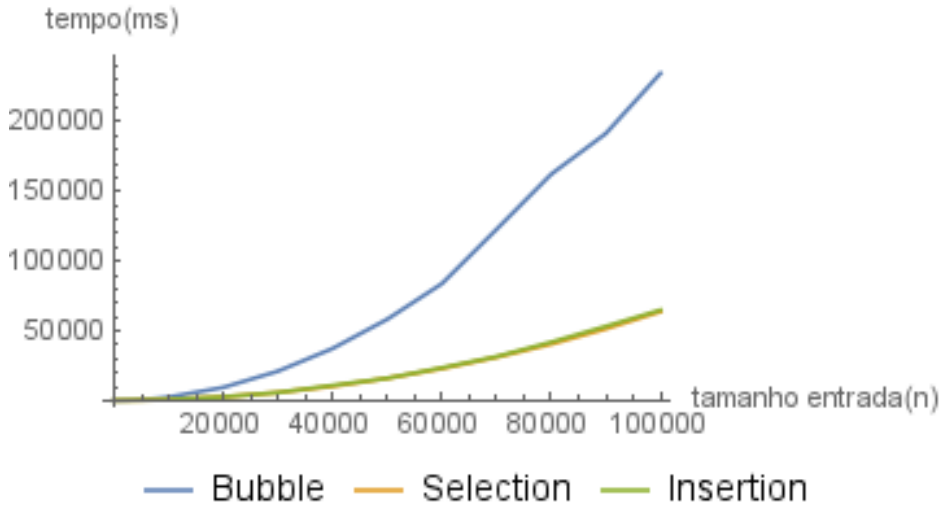

Figura 3.5: Performance dos algoritmos quadráticos em dados decrescentes

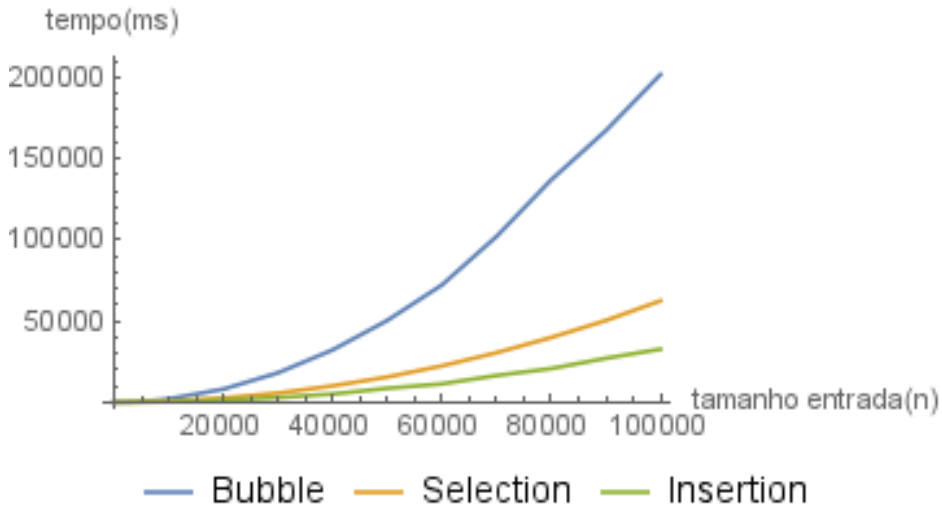

Figura 3.6: Performance dos algoritmos quadráticos em dados aleatórios

Capítulo 4

Análise dos algoritmos linear logaritmicos

Inicialmente, para mostrar com fidelidade o crescimento assintótico do gráfico algumas mudanças foram feitas. Os algoritmos linear logaritmicos não mostram o comportamento de crescimento esperado utilizando a função *ListLinePlot*. Utilizando-se dessa função, ao plotar os dados do *Merge Sort*, observou-se que o gráfico demonstrava um crescimento linear ao olhar unicamente para os pontos plotados.

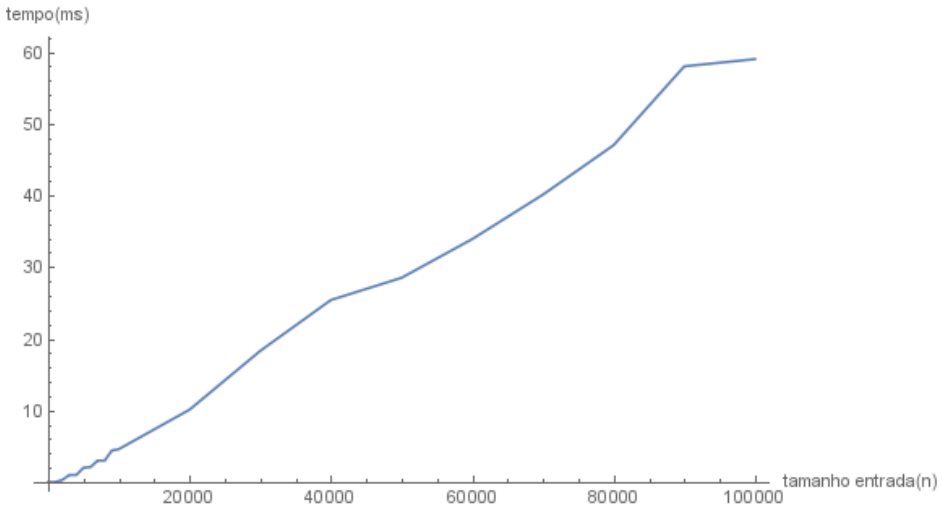

Figura 4.1: Gráfico do *Merge sort* utilizando *ListLinePlot[...]*

Ao utilizar a função *ListLogPlot* pode-se obter um resultado fiel ao desejado, ou seja, observa-se o crescimento logaritmico esperado do algoritmo testado.

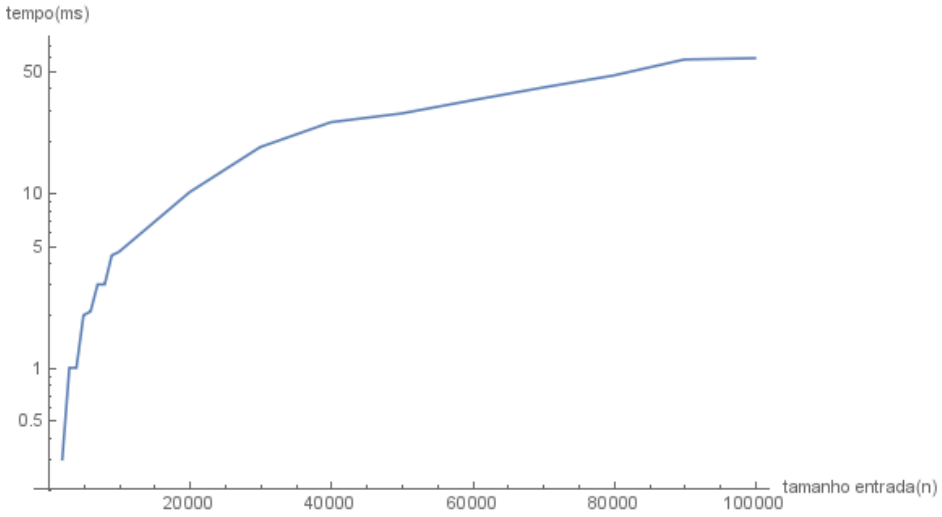

Figura 4.2: Gráfico do *Merge Sort* utilizando *ListLogPlot[...]*

Ademais, um algoritmo que, a princípio, parecia ter comportamento quadrático foi, empiricamente, enquadrado em outra categoria: a linear logaritmica. O *Shell Sort* mostrou-se eficiente na ordenação com um *gap* calculado como $gap = 3 \cdot gap + 1$ e, ao gerar o gráfico, mostrou ser um algoritmo pertencente a categoria desta seção. Portanto, conclui-se que, para esse cálculo de intervalo o *Shell Sort* se encaixa como algoritmo linear logaritmico.

Por fim, conclui-se que os algoritmos linear logaritmicos de forma geral são mais eficazes que os algoritmos de complexidade quadrática. Isso foi demonstrado empiricamente, fato que pode ser observado nos gráficos posteriores onde é visível que, dos algoritmos mais lentos utilizou-se menos de 100 milisegundos para ordenar uma instância com cem mil elementos.

A comparação final dos quatro algoritmos quando pode ser vista na figura 4.17 para os dados aleatórios.

É visível a diferença de tempo de execução dos quatro algoritmos, com exceção ao *Shell* e ao *Heap* que tem tempos parecidos para entradas próximas a 20000. Com isso é possível classificar, quanto ao tempo utilizado para

ordenar, esses algoritmos. Do mais rápido para o mais lento obtém-se a seguinte lista:

1. Quick
2. Merge
3. Shell
4. Heap

Portanto, pode-se observar que o *Heap* é o algoritmo mais lento entre os linear logaritmicos, sendo menos eficiente que o *Shell Sort* em todas as situações testadas. Além disso, é notório que o *Quick Sort* se manteve como algoritmo mais eficiente dentre os demais utilizando um método de escolha de pivô trivial.

Abaixo pode se observar o resultado dos testes empíricos sobre os algoritmos linear logaritmicos em diferentes tipos de dados.

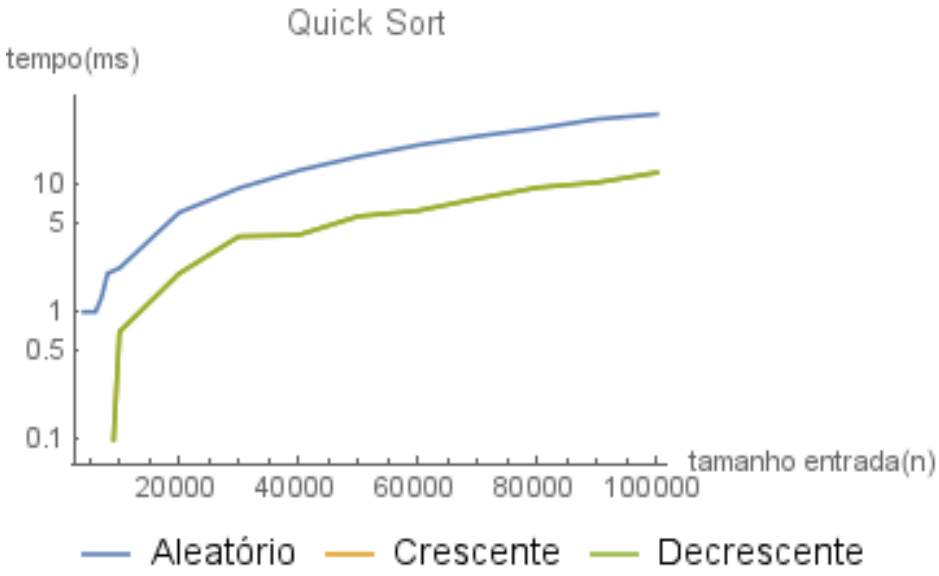

Figura 4.3: Performance do Quick sort

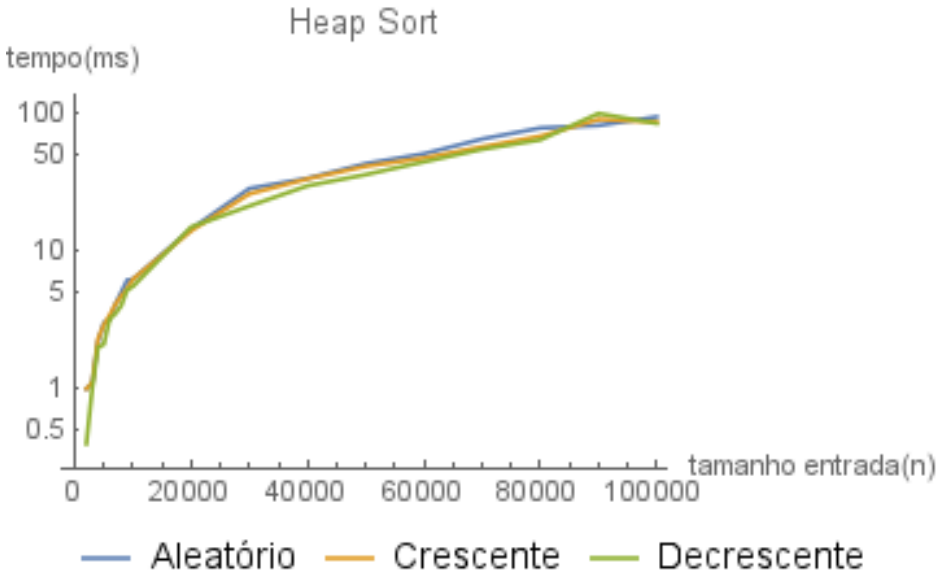

Figura 4.4: Performance do Heap sort

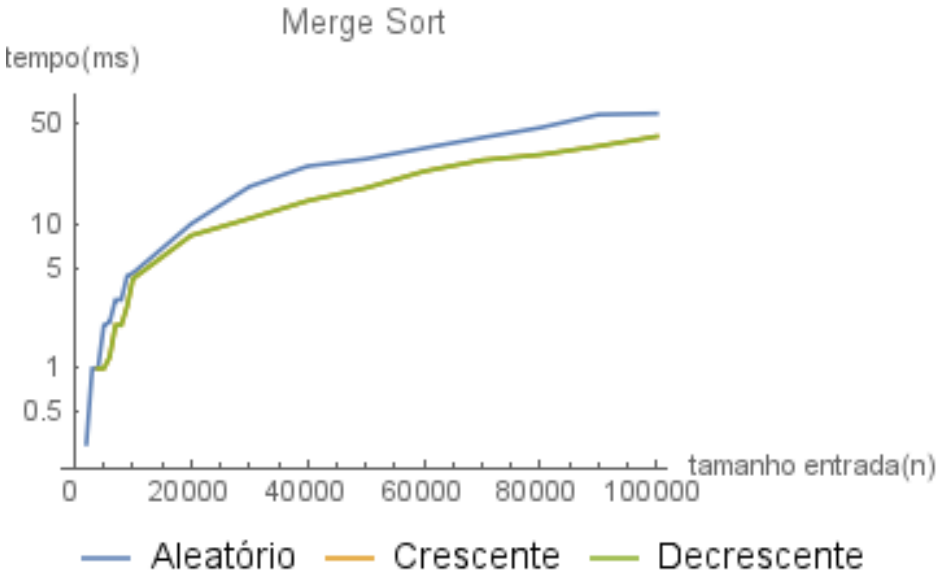

Figura 4.5: Performance do Merge sort

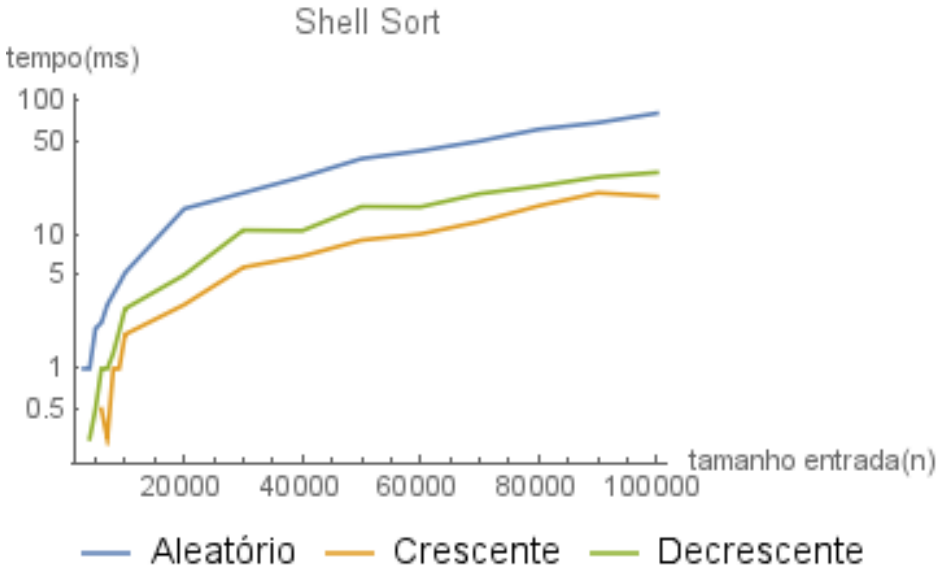

Figura 4.6: Performance do Shell sort

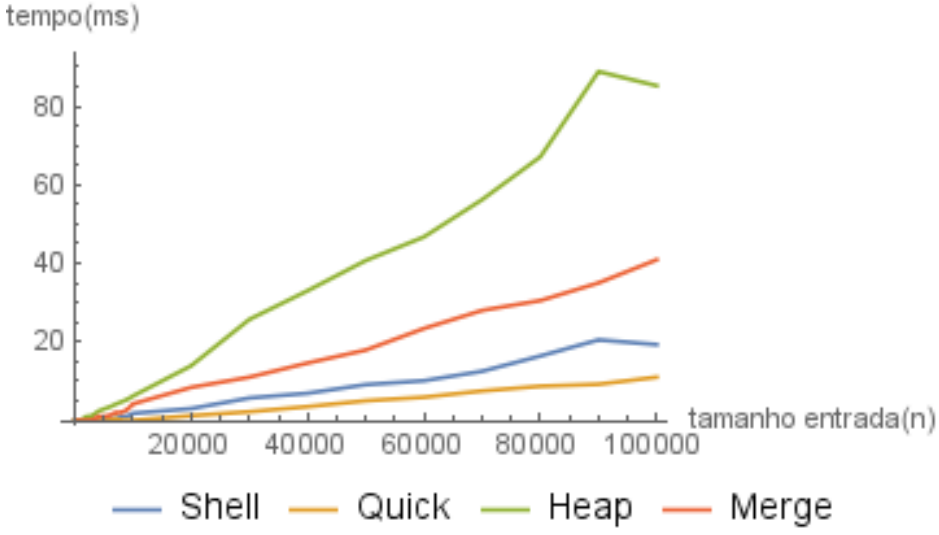

Figura 4.7: Performance dos linear logaritmicos em dados crescentes

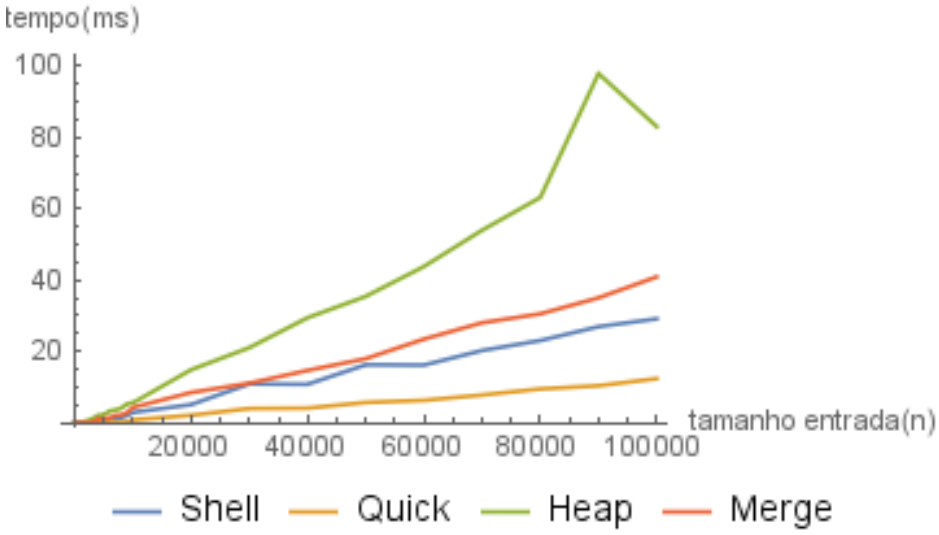

Figura 4.8: Performance dos linear logaritmicos em dados decrescentes

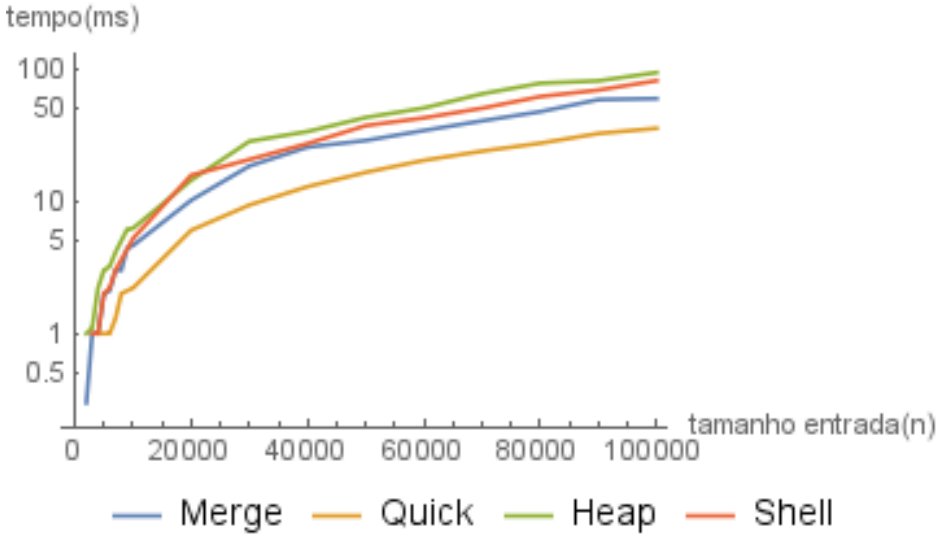

Figura 4.9: Performance dos linear logaritmicos em dados aleatórios

Parte V

Anexos

O trabalho acompanha uma pasta intitulada *algorithms* onde contém todos os algoritmos de ordenação e seus respectivos testes.

Se necessário, a implementação dos algoritmos de ordenação pode ser também encontrada em: `https://github.com/rafaelcn/algorithms`.

Parte VI

Referências

CORMEN, Thomas et al. **Algoritmos**. 3. ed. Elsevier, 2012.

TOSCANI, Laira, VELOSO, Paulo. **Complexidade de Algoritmos : análise, projeto e métodos**. 3. ed. Bookman, 2012.

SEDGEWICK, Robert. **A New Upper Bound for Shellsort**. Journal of Algorithms, 1986

SKIENA, Steven. **Sorting and Searching**. Springer, 2008.

Wolfram Development Platform. Acesso disponível em: <`https://develop.wolframcloud.com/app/`>

Rosetta Code. **Quick Sort**. Acesso disponível em: <`https://rosettacode.org/wiki/Sorting_algorithms/Quicksort`>