Universidade Federal de Minas Gerais
Departamento de Ciência da Computação

Trabalho Prático 2 : Biblioteca Digital de Arendelle

Vinicius Julião Ramos
viniciusjuliao@dcc.ufmg.br

12 de Junho de 2019

Resumo

O objetivo deste trabalho é a realização da comparação das variações do algoritmo de ordenação quicksort, tomando como premissa a necessidade de implementar uma biblioteca digital para o reino de Arendelle. Para isso, o trabalho consistirá na abordagem comparativa, dos métodos desenvolvidos para a ordenação dos documentos do acervo do reino, em que levará-se em conta, mais adiante, os conceitos e detalhes que beneficiam ou depreciam cada método.

1 Introdução

O sistema da biblioteca digital de Arendelle possuirá um grande acervo de documentos, disponíveis para leitura pelos moradores do reino, tal que, a fim de melhorar a experiência dos usuários no acesso à biblioteca, seja necessário manter uma organização visual para melhor compreender as informações trazidas por esse dados. Nesse sentido, umas das partes mais importantes do desenvolvimento da biblioteca digital é o trabalho de ordenar as listas de livros, pergaminhos e escritos do sistema. Com isso trará-se melhorias e facilidades na inserção dos arendellenses à literatura, história e ciência para alcançar o objetivo de potencializar o desenvolvimento econômico através do acesso à informação.

Para executar a tarefa de ordenação desse acervo, é necessário implementar um algoritmo que seja mais rápido e eficaz do que os demais métodos, na maioria dos casos. Observando tais fatores, a ordenação via *quicksort* se destaca por suas características de uso e desenvolvimento, das quais vale ressaltar: o baixo custo computacional, o baixo custo de memória e a possibilidade de melhoramento do desempenho de execução [2], que está associada à análise da organização dos elementos, seguida das variações de implementação na forma de seleção do pivô e nas formas de otimização do código. Nesse sentido, o trabalho consistirá em analisar os custos de execução do *quicksort* aplicando-se pequenas mudanças no método original, para identificar os benefícios e prejuízos de cada variante.

A forma padrão de execução do método de ordenação tratada neste trabalho, ensinada em sala de aula, é implementada através da recursão que consiste em reordenar dois sub vetores a partir do vetor inicial, particionando-os através da seleção do pivô como o elemento central do vetor dado inicialmente. Partindo do método original, mudar-se-á a forma como o método ordena os elementos da seguinte maneira: alteração da forma recursiva pela forma iterativa; alteração na seleção do pivô, selecionando o elemento que representa a mediana entre o elemento central e os elementos da extremidade do vetor; seleção do pivô como o primeiro elemento do vetor; alteração que faz com que a partir de uma certa quantidade de elementos do sub vetor, o método de ordenação *insertion*[3] seja utilizado para finalizar o trabalho.

Inicialmente, o programa incumbe-se de ordenar apenas um vetor de números inteiros, sem quaisquer implementações de uma estrutura de dados composta por uma chave e seus dados, pois o trabalho consiste na análise da performance das variantes do *quicksort* segundo um vetor de números inteiros, como tratado no enunciado do problema. Após o estudo e o levantamento da implementação que apresentou melhor desempenho, será possível desenvolver as estruturas de dados (objetos e structs) necessárias para o desenvolvimento da biblioteca digital de Arendelle, juntamente com a variante de ordenação mais adequada para o sistema.

2 Implementação

No desenvolvimento deste trabalho, fez-se uso de algumas ferramentas que viabilizassem o desenvolvimento e aumentassem a produtividade de elaboração dos códigos, além da aplicação de conceitos de programação que tornassem o código mais legível e organizado. Para cargo de produtividade, utilizou-se o editor de texto Visual Studio Code [5], versão 1.34, que possui corretores com modo *autocomplete* diminuindo erros de digitação e sintaxe. Quanto ao quesito de organização e facilidade de implementação, a linguagem escolhida foi C++ [7] que fornece suporte a orientação a objetos, necessária no desenvolvimento das classes do *quicksort* que careciam do conceito de herança ao implementar as variações desse algoritmo de ordenação.

2.1 Organização do código e decisões de implementação

Para dar maior legibilidade e administrar apenas os aspectos necessários ao desenvolver as variantes do *quicksort* usou-se o conceito de *Orientação a Objetos*. Tendo em vista que parte-se do princípio da existência de um método padrão para o algoritmo de ordenação, deste exercício, implementou-se uma classe pai e as variantes estão estruturadas em subclasses para sobrescrever apenas os métodos que correspondem à mudança proposta pelo exercício. Além disso, houve a necessidade de implementar algumas classes auxiliares para contribuir na montagem das estruturas de dados. Vale ressaltar, ainda, que o arquivo principal de execução do exercício, conta com uma lógica de manipulação dos parâmetros passados no comando que executa o programa. Sendo assim, pode-se detalhar a relação das classes segundo o diagrama:

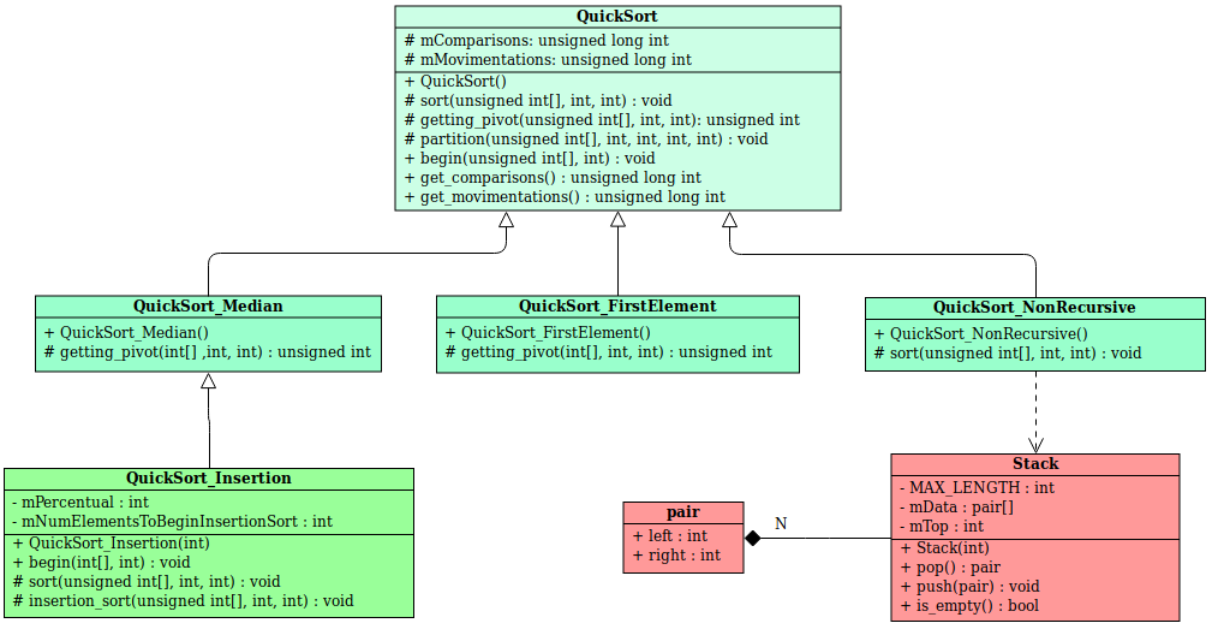

Figura 1: Diagrama de classes UML.

A partir do diagrama da Figura 1 observa-se a relação entre variantes do *quicksort* com o algoritmo padrão e a dependência criada com outras estruturas de dados. Tal estruturação deu maior legibilidade e facilidade de desenvolvimento do problema dado, uma vez que é necessário tratar alguns parâmetros na execução do programa. Assim, basta uma explicação *top-down* desse diagrama para que se entenda a dinâmica de execução de cada uma das classes criadas, uma vez que a propriedade de herança garante que o comportamento de uma sub-classe seja igual à super-classe para os métodos não sobrescritos[8].

2.1.1 QuickSort

Classe responsável pela forma padrão da ordenação, realizando chamadas recursivas e pivotamento com o elemento central do sub vetor. Uma importante observação, quanto à quantidade acumulada de comparações e movimentações realizadas durante a ordenação, se refere à necessidade de implementar uma estrutura capaz de armazenar números muito grandes, uma vez que no pior caso de teste deste exercício, o vetor a ser ordenado possui quinhentos mil elementos. Para isso, fez uso de variáveis do tipo **unsigned long int** que possibilitam salvar um número de 0 a 18446744073709551615 [6]. Por fim, designa-se este objeto como a classe pai, que servirá de base para as demais implementações do *quicksort*, sendo composta pelos seguintes métodos:

unsigned int getting_pivot(unsigned int vector[], int i, int j)
Recebe um vetor de elementos para extração do pivô, índices de início e fim da parte do vetor a ser considerada.
Retorna o elemento central do vetor parametrizado.

void sort(unsigned int vector[], int &left, int &right)
Recebe um vetor de elementos para ordenar, índices de início e fim da parte do vetor a ser considerada.
Executa chamadas recursivas do prórpio método.

void partition(unsigned int vector[], int left, int right, int &i, int &j)
Recebe um vetor de elementos para ordenar, índices de início e fim da parte do vetor a ser considerada e referência das variáveis que indicarão as novas partições do vetor.
Executa a partição do vetor recebido em dois outros vetores, tomando um pivô como referência e separando os elementos maiores que o pivô dos elementos menores que esta referência.

void begin(unsigned int vector[], int num_elements)
Recebe um vetor de elementos para ordenar e a quantidade de elementos nesse vetor.
Executa a chamada aos demais métodos responsáveis pela ordenação via *quicksort*.

unsigned long int get_comparisons()
Retorna a quantidade de comparações de elementos do vetor executadas desde o disparo da função **begin()** até o fim da ordenação.

unsigned long int get_movimentations()
Retorna a quantidade de movimentações de elementos do vetor executadas desde o disparo da função **begin()** até o fim da ordenação.

As demais classes que pertencem ao diagrama da Figura 1 serão descritas apenas pelos métodos que sobrescrevem as funções desta super classe, uma vez que, a relação de herança entre os objetos tratará de executar os métodos não sobrescritos de acordo com a classe herdada.

2.1.2 QuickSort_FirstElement

Esta classe estende de **QuickSort** fazendo uso da forma padrão de execução, porém altera a maneira com a qual o pivô é selecionado. Neste caso, o referência sempre será o primeiro elemento do sub vetor da recursão, e para tanto, basta realizar uma sobrescrição do método que define a referência de ordenação.

unsigned int getting_pivot(unsigned int vector[], int i, int j)
Recebe um vetor de elementos para extração do pivô, índices de início e fim da parte do vetor a ser considerada.
Executa a sobrescição do método **QuickSort::getting_pivot**
Retorna o primeiro elemento do vetor parametrizado

2.1.3 QuickSort_Median

Esta classe também herda de **QuickSort** fazendo uso da forma padrão de execução e altera a maneira com a qual o pivô é selecionado. Para isso, executa-se uma rotina de mediana entre três números para marcar qual deverá ser o a nova referência de pivotamento.

unsigned int getting_pivot(unsigned int vector[], int i, int j)
Recebe um vetor de elementos para extração do pivô, índices de início e fim da parte do vetor a ser considerada.
Executa a sobrescição do método **QuickSort::getting_pivot**

Retorna o elemento que é a mediana entre o elemento central e os elementos das "bordas"do vetor parametrizado.

2.1.4 QuickSort_Insertion

Nesta implementação, a partir de um certo valor, recebido como parâmetro da classe, que corresponde ao percentual de elementos que interromperão a execução da ordenação via *quicksort*, o programa passa a ordenar o sub vetor usando o método *insertion sort*. Além disso, a forma de seleção do pivô é através da mediana de três elementos, assim como na classe **QuickSort_Median**, logo, esta é a super classe dessa variação do algoritmo. Para isso, basta sobrescrever o método **QuickSort_Median::QuickSort::sort** para que haja uma validação sobre quantos elementos o sub vetor, recebido pelo método, possui e assim redirecionar a execução para o método de ordenação por inserção.

void insertion_sort(unsigned int vector[], int left, int right)

Recebe Um vetor de elementos para se ordenar, juntamente com os índices de início e fim deste vetor.

Executa a ordenação dos elementos do vetor via algoritmo *insertion sort*.

void begin(unsigned int vector[], int num_elements)

Recebe Um vetor de elementos para se ordenar e a quantidade de elementos que este vetor possui.

Executa a sobrescição do método da superclasse **QuickSort::begin** calculando a quantidade de elementos para o qual o programa executará ordenação via *insertion sort*.

void sort(unsigned int vector[], int &left, int &right)

Recebe Um vetor de elementos para se ordenar, juntamente com os índices de início e fim deste vetor.

Executa a sobrescição do método da superclasse **QuickSort::sort**, redirecionando a execução para o método **insertion_sort()** desde que o sub vetor contenha menos elementos que o percentual estipulado nos parâmetros de instanciação.

2.1.5 QuickSort_NonRecursive

Esta classe implementa uma forma iterativa de executar a ordenação via *quicksort* fazendo uso da estrutura de dados pilha [4], para salvar o percurso que o algoritmo tomará para ordenar o supra vetor ao terminar a ordenação de um sub vetor. Para esta tarefa, sobrescreveu-se o método **QuickSort::sort** que outrora fora responsável pela recursão que se deseja impedir.

void sort(unsigned int vector[], int &left, int &right)

Recebe um vetor de elementos para se ordenar, juntamente com os índices de início e fim deste vetor

Executa a sobrescição do método **QuickSort::sort**, ordenando de forma iterativa ao invés da forma recursiva da maneira padrão do algoritmo.

2.1.6 pair

Estrutura desenvolvida para mapear os índices do vetor ao implementar o algoritmo não recursivo do *quicksort*. Tal estrutura salva o índice final e inicial que identificam o sub vetor com o qual o programa "trabalhará"em determinada iteração.

2.1.7 Stack

Esta classe implementa a estrutura de dados conhecida como pilha [4]. A implementação desse objeto foi dada de forma simples, sem elaboração de templates ou usos genéricos, uma vez que necessita-se apenas guardar, especificamente, os valores da estrutura **pair** para a execução do método não recursivo da ordenação. Para isso, guardou-se os elementos **pair** em um vetor que possui tamanho máximo igual ao número de elementos que deseja-se ordenar, pois, no pior caso do *quicksort*, é necessário armazenar os índices início e fim do sub vetor, onde cada sub vetor possui apenas um elemento, dessa forma necessita-se de uma pilha com tamanho máximo igual ao tamanho do vetor do problema.

pair pop()

Executa o desempilhamento de um elemento da estrutura.

Retorna o elemento desempilhado.

void push(pair data)

Recebe um elemento para empilhar

Executa o empilhamento do elemento parametrizado.

bool is_empty()

Executa a validação se há algum elemento na pilha.

Retorna a resposta da validação.

2.1.8 Utils

Namespace criado com o objetivo de separar as funcionalidades principais do programa das funcionalidades secundárias. Neste caso, as funcionalidades dadas como secundárias são referentes à manipulação dos vetores que passarão pelos algoritmos de ordenação ao longo do programa. Para isso, as seguintes funções foram implementadas:

void create_aleatory_vector(unsigned int vector[], unsigned int length)

Recebe um vetor qualquer e a quantidade de elementos que este vetor possui.

Executa o preenchimento do vetor com números aleatórios, em que o intervalo dos elementos do vetor vai de 1 até o parâmetro **length**.

Vale destacar um detalhe de implementação na geração dos números aleatórios. É necessário realizar a troca da raiz geradora a fim de se evitar a vícios de aleatoriedade dos números, porém usando apenas o *timestamp*, contido em <**ctime**>**::time()**, como *seed* do algoritmo randomico, não é possível precaver tal problema. No intervalo de um segundo, o programa executa todas as chamadas à este procedimento, por isso o *seed* nunca é alterado na geração dos vetores. Assim, fez-se uso da biblioteca <**chrono**> que permite calculos em unidades de tempo na casa dos microssegundos, e, graças a isso, é possível produzir uma raiz randômica que seja diferente à cada nova chamada de **create_aleatory_vector**.

void create_decreasing_vector(unsigned int vector[], unsigned int length)

Recebe um vetor qualquer e a quantidade de elementos que este vetor possui.

Executa o preenchimento do vetor com números sequenciais decrescentes, em que o intervalo dos elementos do vetor vai de 1 até o parâmetro **length**

void create_increasing_vector(unsigned int vector[], unsigned int length)

Recebe um vetor qualquer e a quantidade de elementos que este vetor possui.

Executa o preenchimento do vetor com números sequenciais crescentes, em que o intervalo dos elementos do vetor vai de 1 até o parâmetro **length**.

void copy_vector(unsigned int cp_vector[], unsigned int pst_vector[], unsigned int length)

Recebe o vetor a ser copiado, o vetor a receber as copias do valor e o tamanho de ambos os vetores

Executa a passagem de valores do vetor **cp_vector** para o vetor **pst_vector**

2.1.9 main

Para critério de legibilidade, dividiu-se o procedimento inicial do programa em duas partes. A primeira interpreta os parâmetros passados na chamada de execução, a outra executa automaticamente a chamada a todas as variantes do *quicksort*, como especificado no enunciado deste trabalho. Além disso, o programa testa se os parâmetros foram passados de forma incorreta. Para desempenhar tais tarefas os procedimentos desenvolvidos são:

int main(int argc, const char *argv[])

Recebe os parâmetros dados na chamada de execução do programa.

Executa a interpretação dos parâmetros e faz a chamada para a ordenação dos vetores de acordo com os parâmetros passados.

void execute_whith_params(std::string algorithm_variation, std::string vector_type,unsigned int items_quantity, bool print_vector)

Recebe os parâmetros que definem a variação do algoritmo que será executada, juntamente com a organização do vetor e a quantidade de elementos que este terá. Além disso, o booleano recebido é referente à necessidade de imprimir na saída padrão os vetores utilizados para a ordenação.

Executa a variação do algoritmo especificada nos parâmetros. Porém tal execução é realizada vinte e uma vezes a fim de se obter a média de comparações e movimentações executadas, e, também a mediana do tempo que o programa gastou para a execução destas repetições dada a mesma quantidade de elementos. Ao final das execuções, este procedimento imprime os dados obtidos na saída padrão.

Alguns detalhes de implementação que merecem destaque nesse ponto é o uso de polimorfismo e a instanciação de um apontador de função. O polimorfismo em conjunto com a implementação correta de métodos virtuais na herança dos objetos, no nosso caso, o **QuickSort** e suas derivações, permite que a definição da variante de ordenação seja feita apenas uma vez antes do início das vinte e uma repetições, pois a *Orientação a Objetos* possui a propriedade polimórfica que trata uma subclasse usando a referência da classe superior [9]. Quanto ao uso de apontador para funções, o motivo da aplicação é o mesmo do polimorfismo: evitar o aninhamento entre comparações (*if*) para cada uma das vinte uma iterações, a fim de determinar o método que organizará o vetor tratado na ordenação.

void execute_work_tests()

Executa chamadas à função **execute_with_params** em que cada chamada contém um tipo de variação do *quicksort*, um tipo de organização do vetor e um valor que corresponda à quantidade de elementos a serem ordenados.

2.2 Ferramentas, Sistema e Dados

A partir da elaboração do diagrama de classes e de um plano de projeto para o desenvolvimento deste trabalho, iniciou-se a parte de implementação e criação dos códigos. Com isso, necessitou-se a tomada de decisões que envolviam as definição de parâmetros de desenvolvimento e a seleção de ferramentas que viabilizassem a conclusão do projeto. Para descrever melhor quais os aspectos interferiram no progresso deste exercício, consideremos os seguintes pontos:

2.2.1 Sistema

Segundo a descrição do trabalho prático, o programa deverá funcionar em um computador com Linux, porém, o desenvolvedor possui apenas um computador com MacOS instalado. Visto que ambos os sistemas operacionais provém de um kernel Unix, a integração do desenvolvimento se deu de forma prática, bastando escolher as bibliotecas e a aplicação de compilação correta, desde que houvesse o cuidado com alguns pequenos aspectos que diferenciam um sistema Linux, como o Debian, Arch, CentOS por exemplo, de um MacOS. Houve, ainda, um empecilho de execução que é um padrão dos sistemas de kernel Linux para impedir que programas expandam uma árvore de recursão muito grande. Tal característica tenta coibir prejuízos às funções do sistema operacional. Entretanto, dado o pior caso do *quicksort* a árvore de execução criada poderá ultrapassar o limite de oito *mega bytes* predefinido pelo SO, logo faz-se necessário estender o limite de recursão de, 8MB para 128MB, através do comando:

```
$ ulimit -s 131072
```

2.2.2 Ferramentas

Neste ponto, a tomada de decisões tinha como função principal o tempo previsto para o desenvolvimento dos códigos. Dessa maneira, selecionou-se as ferramentas que atendessem a dois quesitos: familiaridade do desenvolvedor e funcionamento em sistemas Linux. Sendo assim, faz-se uma breve descrição justificativa sobre a seleção das aplicações de desenvolvimento de software:

C++11: Dada a necessidade do uso de *Orientação a Objetos* e da restrição sobre a linguagem que deveria ser utilizada para o desenvolvimento deste trabalho, *C++* foi a única linguagem que se enquadra nos dois aspectos. Entretanto, havendo a necessidade de manipulações com medidores de tempo na casa dos microssegundos surge o motivo pela escolha da biblioteca *<chrono>*[10] que está disponível apenas a partir do *C++11*.

Makefile e gcc: Visto que o desenvolvimento do software se daria para um sistema Linux e dada a restrição de compilação pela descrição do trabalho, a solução encontrada para tornar os arquivos de código

em um arquivo executável foi a integração entre o compilador **gcc** e a ferramenta **Makefile**. O **gcc** compila códigos para **C++** através do comando *g++* e usado em conjunto com o **Makefile** cria-se a possibilidade de navegar entre os diretórios e módulos da estrutura de arquivos do programa, compilando passo a passo cada uma das etapas e dependências que o programa carece.

Visual Studio Code & Debugger Plugin: Por experiências em desenvolvimentos de outros softwares, o desenvolvedor possui maior familiaridade com o editor de texto **Visual Studio Code**[5], que mostra-se uma boa ferramenta no desenvolvimento de aplicações **C++**, uma vez que disponibiliza uma gama de *plugins* e extensões. Uma das extensões que integraram este projeto foi o **Debugger Plugin** para **C++** que utiliza o debugger *GDB*[11], contanto traz a facilidade de uma boa interface gráfica que facilita a leitura e a localização de erros no código.

Valgrind: A fim aumentar a qualidade do trabalho desenvolvido, é preciso impedir que o programa produza *leaks* de memória, visto que o *quicksort* é um algoritmo recursivo na maioria dos casos. Dada uma chamada recursiva, a reserva de memória realizada por aquele trecho de código deve ser desalocada em algum momento, pois no teste geral do programa, executa-se duzentos e dez chamadas diferentes ao *quicksort* e tais *leks* podem causar o mal funcionamento do software, à medida que se avança entre as fases desse teste geral. Logo, requisitou-se o uso da ferramenta **Valgrind**[12], que realiza um relatório sobre trechos de memória reservados na execução do programa e não desalocados até o fim do processamento.

Overleaf: A forma de registrar todo o trabalho executado, as tomadas de decisões que envolvem o projeto, as justificativas e análises que circundam a elaboração de uma solução para o problema é através da documentação do trabalho. Dada esta necessidade, faz-se necessário, também, o uso de uma ferramenta que padronize esse tipo de documento segundo algum parâmetro e facilite tal padronização. Para tanto, a ferramenta **Overleaf**[13] provê suporte de templates que fazem o trabalho de colocar o texto segundo as regras descritas pelas bibliotecas de implementação, como é o caso deste documento, que segue os parâmetros de publicação de artigos, descritos pela *Sociedade Brasileira de Computação - SBC*[14].

Gerador de gráficos: O sistema MacOS possui uma gama de aplicativos nativos de fácil utilização e de bom aspecto visual. Portanto, visto que os dados de execução foram exportados em um tipo de tabela, em que os separadores se davam pelo caracter de espaço, tornou-se viável o uso de aplicações que traçam gráficos a partir de tabelas. Para isso, usou-se o aplicativo *Numbers*[15] para a geração de gráficos e personalização das tabelas, para que fosse possível melhorar a visualização dos dados.

2.2.3 Dados

A proposta deste trabalho prático solicitou a implementação de um programa que recebesse parâmetros na chamada de execução e imprimisse alguns dados referentes à ordenação executada. Portanto, os dados, que o programa lê e produz, são referentes apenas ao proposto nos enunciados do problema. Sendo assim, o programa está preparado para receber os seguintes parâmetros de forma respectiva:

<Variação do Quicksort> <Disposição dos elementos no vetor> <Número de elementos do vetor>

Além desses, um ultimo parâmetro pode ser passado de forma opcional, que se refere à impressão dos vetores utilizados na ordenação. Na saída de dados, também produziu-se as informações da mesma forma e disposição que o solicitado no enunciado do trabalho, enviando para a saída padrão os atributos solicitados na ordem previamente especificada.

3 Instruções de Compilação e Execução

Após a conclusão do trabalho, deve-se de instruir os demais usuários, sobre a forma como executar o programa e as especificações de execução e características que o software possui. Sendo assim vale lembrar que o programa foi desenvolvido e compilador utilizando **gcc** e **Makefile**, por isso, usando o terminal *bash* e abra o diretório raiz dos códigos (o mesmo diretório que possui o arquivo *main.cpp*), na sequencia execute o comando que compilará os códigos:

```
$ make
```

Um arquivo executável chamado *executavel* será criado e através deste arquivo, se dará a execução da solução. Porém, antes da chamada ao programa, deve-se alterar o parâmetro do sistema que faz referência ao tamanho da recursividade criada por um programa. Para isso, basta executar:

```
$ ulimit -s 131072
```

Há três possibilidades de respostas diferentes dadas pelo programa, de acordo com os parâmetros passados, pode-se retratar da seguinte maneira:

3.1 Execução parametrizada

Através dessa chamada, o programa esperará 3 parâmetros, previamente especificados no texto base deste trabalho, como entrada. E após vinte e uma execuções, o programa responderá na saída padrão do sistema os dados de custo médio da execução do programa. Vale relembrar como se dá a entrada e a saída do programa, respectivamente, tomando o exemplo:

```
$ ./executavel QNR Ale 100000
```

```
QNR Ale 100000 2213248 422254 15566
```

3.2 Execução parametrizada e exibição dos vetores utilizados

Nesta execução, o programa esperará 4 parâmetros, os mesmos três do tópico anterior e um parâmetro que identifica a necessidade de printar os vetores. Visto que o programa realiza vinte e uma repetições de execução, então vinte e um vetores são utilizados para executar o programa e após a execução da ordenação destes arranjos de números inteiros, o programa responderá na saída padrão do sistema os dados de custo médio da execução do programa e os vetores utilizados da seguinte maneira:

```
$ ./executavel QNR Ale 10 -p
```

```
QNR Ale 10 39 11 1
2 10 9 10 2 1 9 9 2 5
7 4 1 6 4 9 2 10 8 9
1 2 10 5 7 3 10 2 10 3
2 2 8 2 10 2 2 2 5 9
6 10 7 1 3 6 10 1 4 3
10 8 9 10 9 10 8 10 3 7
4 6 8 9 2 7 6 2 5 1
1 8 4 7 2 9 10 10 8 10
2 5 9 7 8 8 5 10 3 6
9 4 5 5 8 10 9 1 9 8
3 2 4 4 1 4 7 10 8 2
4 2 2 1 7 3 9 10 3 8
1 1 8 2 7 5 3 1 9 10
5 9 7 1 3 2 1 3 8 4
```

```
9  7  9  10  6  6  9  2  10  8
3  8  8  9  9  3  7  1  9  9
7  6  10  8  2  7  5  3  1  3
1  4  9  4  5  4  3  2  10  7
5  5  1  3  8  8  1  1  9  1
9  3  10  2  1  2  2  3  1  5
3  1  2  1  4  9  10  2  10  9
```

3.3 Execução total dos casos de teste

Esta é a execução responsável pela coleta de dados do programa, pois executa todas as combinações de parâmetros, desde a variação do algoritmo original até a variação da disposição inicial dos elementos no vetor a ser ordenado. Tal execução toma como regra a especificação dos casos de teste para este trabalho: o vetor deve conter, a priori, cinquenta mil elementos, aumentando de cinquenta mil em cinquenta mil até atingir quinhentos mil elementos. Para cada combinação de parâmetros, o programa executará vinte e uma vezes, obterá a média de comparações e movimentações realizadas pela respectiva variação do *quicksort*, juntamente com a mediana do tempo de execução das vinte e uma repetições, e ao final, imprimirá na saída padrão os resultados de todas as duzentas e dez combinações possíveis. Para invocar esse teste de execução, basta não passar parâmetros iniciais na chamada do programa:

```
$ ./executavel
```

A chamada a esta função levará algum tempo para terminar, então aconselha-se salvar os dados em um arquivo de texto para visualização posterior. Para isso, basta utilizar o comando > que fará com que a saída padrão do sistema, utilizada pelo programa seja redirecionada a algum arquivo ou instância que receba o tipo de informação dada pelo software. Pode-se executar um exemplo em que toda a saída do programa é redirecionada a um arquivo chamado *saida.txt*:

```
$ ./executavel > saida.txt
```

Tal comando de redirecionamento da saída do programa poderá ser executado em qualquer um dos três tipo de execução descritos nesta seção.

4 Análise Experimental

Antes de expor os detalhes das respostas obtidas na execução do estudo comparativo, é necessário abordar os aspectos e recursos computacionais sob os quais o programa foi executado. Visto que aqui realiza-se uma etapa empírica de estudo, o entendimento das condições do sistema, no instante em que a o programa desempenhava seu trabalho, agregará mais variáveis à análise de execução, com isso, têm-se mais firmeza argumentativa nas afirmações que envolvem o desempenho, o custo e a qualidade da solução encontrada.

Após essa primeira análise, pode-se realizar a comparação entre as soluções do *quicksort* aqui desenvolvidas. Neste instante, sabe-se os parâmetros do sistema que beneficiam ou depreciam o uso de alguns métodos em relação aos demais, por isso torna-se possível um estudo que considera, os casos extremos que se deve evitar ao implementar a ordenação utilizando o método estudado.

4.1 Parâmetros de Sistema

Apesar do desenvolvimento se dar a partir de um MacOS, a execução do estudo realizou-se em um computador com o sistema Linux Ubuntu 18.04LTS, visto que a recomendação do texto base do trabalho exigia o funcionamento do programa nesse tipo de SO e dada a necessidade de deixar o programa executar durante um longo tempo até que completasse todo o trabalho. Tal computador possui um processador *Intel Core2Duo*® e conta com 6GB de memória RAM, porém temos uma comparação entre o uso de memória e processamento antes e durante a execução do software:

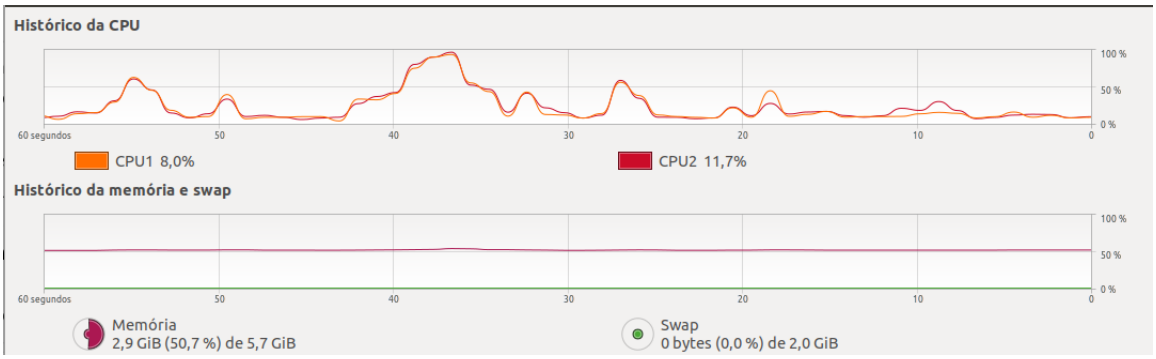

Figura 2: Disponibilidade de recursos da máquina antes da execução do programa.

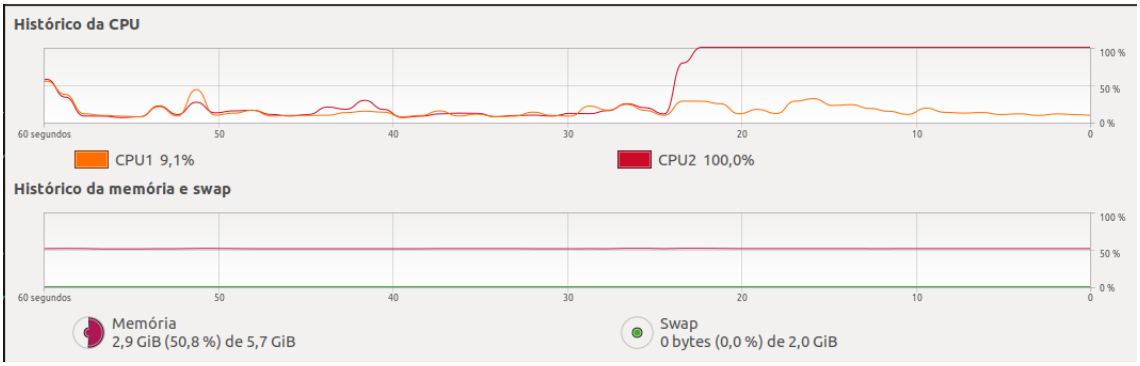

Figura 3: Disponibilidade de recursos da máquina durante a execução do programa.

Através da análise das Figuras 2 e 3 observa-se que há um pico anormal no gráfico que registra o uso da CPU2 do computador. Tal anomalia acontece após o disparo de execução do comando que gerará todos os casos de estudo para este trabalho.

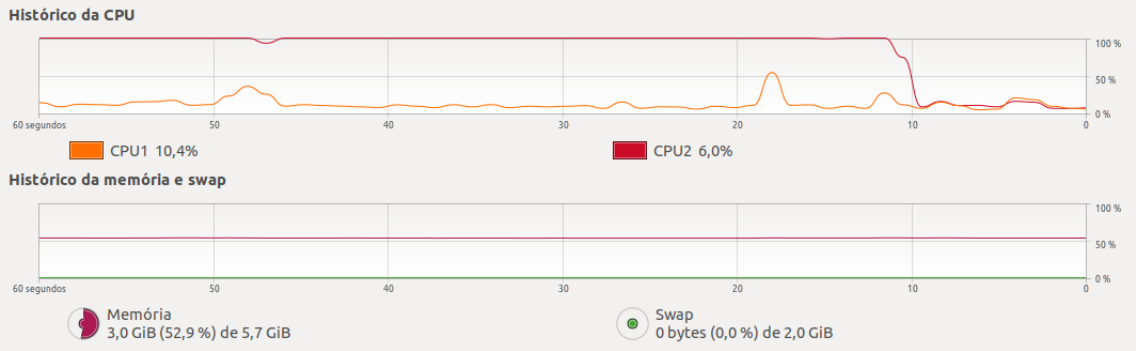

Figura 4: Retomada do estado inicial da disponibilidade de recursos.

É possível ressaltar, ainda, que o uso de um dos núcleos do processador aumentou e permaneceu alto durante toda a execução do programa, até o seu final, registrado no vale do da Figura 4. Tal fato provém da falta da implementação de *threads* que permitiriam dividir o único processo deste programa em vários outros subprocessos, logo, todo o programa ocupará a disponibilidade da CPU, elevando para 100% o uso de um dos núcleos de processamento.

4.2 Metodologia de Análise

O acompanhamento e estudo sobre a melhor forma de ordenar usando alguma das variações do *quicksort* se deu através da visualização do comportamento assintótico dos dados de saída. A execução do programa retorna o número de comparações, o número de movimentações, a quantidade de tempo gasto para resolver a ordenação de um vetor de N elementos, e a partir destas informações pode-se considerar quais delas têm maior peso de análise para que seja possível afirmar e argumentar sobre as propriedades dos métodos.

Após uma análise individual de cada alternativa do *quicksort* elaborada neste trabalho, terá-se uma conclusão sobre os pontos principais que necessitam de uma nova abordagem. Então, de posse destes pontos de destaque, fará-se uma análise comparativa entre eles.

De antemão, vale explicar a nomeclatura que fora utilizada para a montagem da saída dos dados e para a plotagem dos gráficos exibidos nas seções seguintes. As denominações explicadas, se referem à disposição dos elementos no vetor passado aos métodos de ordenação: **Ale** - vetor com elementos embaralhados de forma aleatória, **OrdC** - vetor com elementos sequencialmente ordenados de forma crescente, **OrdD** - vetor com elementos sequencialmente ordenados de forma decrescente. As demais nomeclaturas serão especificadas ao passo em que faz-se referência a estas.

4.3 QuickSort - (QC)

Visto que esta é a forma padrão da ordenação estudada, vale a pena argumentar de forma mais detalhada, para, posteriormente, realizar a parte comparativa da existência de algum método que se enquadra melhor diante da necessidade. Para tanto, alguns recursos darão maior clareza dos fatos:

O fato, facilmente visualizável, de que as propriedades tempo, comparações e movimentações são diretamente proporcionais à quantidade de elementos, não descarta a necessidade de uma análise mais profunda sobre os dados. Observa-se de início que em **OrdC** e **OrdD** a diferença entre as propriedades é muito pequena, onde o maior fator é no quesito movimentações, em que, para o vetor decrescente, registra-se o dobro de trocas de posições comparado ao vetor crescente. Porém, a discrepância maior está nos dados que vêm de **Ale**, uma vez que, tanto para o número de comparações e movimentações observadas na Tabela 1, quanto o tempo plotado na Figura 5 os números são muito superiores.

Vale ressaltar que o tempo de execução da aplicação é uma boa medida para o custo de um programa, visto que tal métrica resume o poder computacional. Ao implementar um serviço, como a biblioteca de Arendelle, deseja-se saber qual o algoritmo que atenderá da melhor forma possível levando, e isso envolve diretamente o tempo de processamento dos dados. Logo, faremos uma visualização mais baseada na métrica

Tabela 1: Números de comparações (Cmp) e movimentações (Mov) para cada tipo de vetor.

Elementos	Cmp OrdC	Cmp OrdD	Cmp Ale	Mov OrdC	Mov OrdD	MovAle
50000	750015	750028	1055213	32767	57766	199092
100000	1600016	1600030	2224950	65535	115534	422178
150000	2456803	2456820	3443971	84464	159464	653179
200000	3400017	3400032	4740449	131071	231070	890846
250000	4250017	4250032	5949097	131071	256070	1133245
300000	5213588	5213606	7364087	168928	318928	1374637
350000	6213588	6213606	8681277	218928	393928	1621346
400000	7200018	7200034	10034738	262143	462142	1872406
450000	8100018	8100034	11395118	262143	487142	2123002
500000	9000018	9000034	12737363	262143	512142	2376039

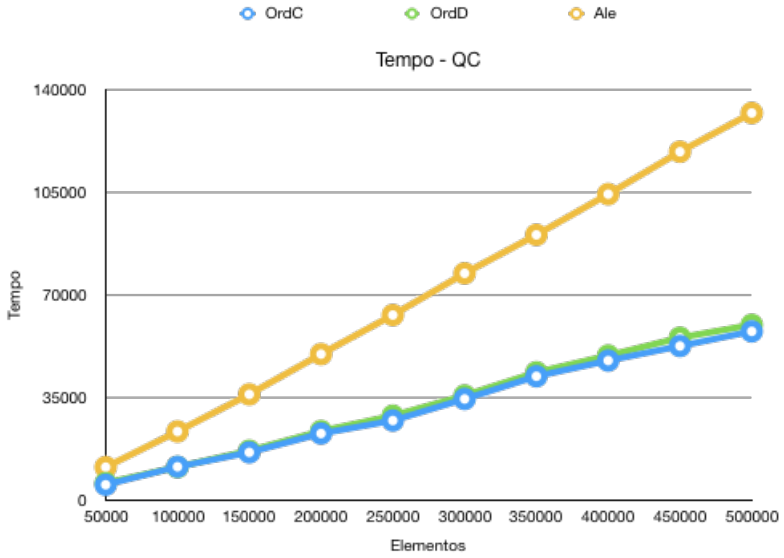

Figura 5: Elementos x Tempo de execução.

referente ao tempo, porém sem desconsiderar outros fatores, que eventualmente seja importantes. A partir disso, destaca-se o comportamento assintótico do método **QC**, considerando **Ale**, que segundo a Figura 5, possui uma característica muito próxima à uma função linear, $O(n)$, que reflete em um bom custo, se comparado aos demais métodos disponíveis.

4.4 QuickSort_NonRecursive - (QNR)

Espera-se, de início, que uma repetição por recursão seja mais demorada e custosa, se comparada à repetição dada iterativamente. Um método auto recorrente acumula uma pilha de chamadas ao sistema e ao longo que tal pilha cresce, os recursos diminuem e é cada vez mais custoso para a máquina completar uma instrução, visto que sua área de execução e registro encontra-se cheia.

No entanto, existem algoritmos que carecem da execução por vias recursivas, ou então a implementação iterativa através do uso de uma estrutura de dados **Pilha** para o registro da ultima instrução antes da chamada recursiva, que é o caso desta variação do *quicksort*. Com isso, há o trabalho em validar e preencher a **Pilha** e controlar o funcionamento das iterações através do uso de comparações, o que pode aumentar o custo do algoritmo, e depreciar o benefício da substituição da recursão pela iteração.

Como explicado anteriormente, na seção 2, a classe **QuickSort_NonRecursive** estende da classe **QuickSort** e por isso faz uso da função **QuickSort::partition**, responsável pela comparação e movimentação

entre os itens do vetor. Por isso, nestes dois parâmetros, **QNR** é exatamente idêntico à **QC**, logo a Tabela 1 também faz jus à este método. Porém, como as etapas da execução da ordenação se diferem, deve-se avaliar a o tempo de execução para o **QNR**.

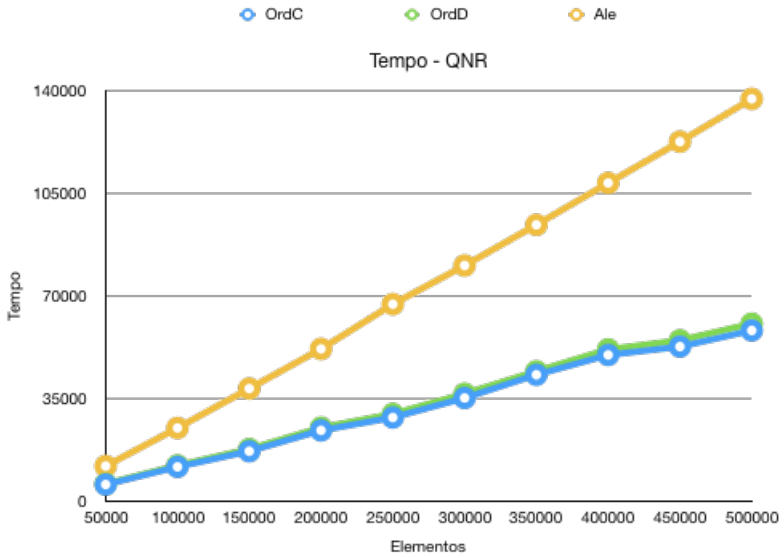

Figura 6: Elementos x Tempo de execução.

A similaridade entre o gráfico da Figura 6 e da Figura 5, quanto ao comportamento das funções em relação à quantidade de elementos, causa a indagação sobre um resultado inesperado: A execução iterativa deveria apresentar melhor desempenho do que a execução recursiva. Entretanto, é plausível dizer que ao aplicar a estrutura **Pilha** no controle da execução do *quicksort*, necessitou-se da implementação de vária validações, em forma de comparações, sobre o estado da estrutura de dados, que causou na agregação de um maior custo, que anteriormente não existira. Por isso, nem sempre pode-se afirmar que a iteração sempre será melhor perante a recursão.

Vale destacar ainda, que este método é desvantajoso, uma vez que o custo de execução de **QC** é equivalente. Porém tal método não requer a alocação de mais memória para sua execução, dada a forma com a qual o compilador resolve os métodos recursivos de forma automática e barata (no sentido de alocação de memória).

4.5 QuickSort_Median - (QM3)

Neste tipo de abordagem de seleção do pivô, espera-se que o número partições "degeneradas"seja menor. Uma vez que ao usar a mediana entre três números, garante-se que a partição com o menor número de elementos sempre tenha no mínimo 1 elemento, a partição mais degenerada nesse caso possuiria, ao menos, um elemento. O fato da diminuição de partições degeneradas traz o benefício de uma menor quantidade de recursões criadas, pois sempre haverá uma chamada recursiva para cada partição elaborada. Uma vez que o alto número de instruções na pilha de execução eleva os custos, falando de forma prática, o resultado esperado desse teste é de que seja mais rápido que a forma padrão de execução (**QC**).

Observando a Tabela 1 em comparação com a Tabela 2, nota-se o comportamento muito próximo ao examinar o tempo e a quantidade de movimentações para todos os tipos de disposição do vetor, porém há um comportamento um pouco mais custoso quando refere-se à quantidade de comparações realizadas. Pode-se explicar facilmente este fato, uma vez que na necessidade de obter a mediana, é necessário exercer pelo menos uma comparação entre dois elementos, assim, há um custo a mais para a seleção do pivô. Com isso, espera-se que haja algum aumento no custo do comparações entre **QC** e **QM3**.

O aumento da quantidade de comparações pode ser observada pelos seguintes gráficos:

Assim, nota-se que apesar de muito discreto, há um peso a mais na execução de **QM3** e que reflete

Tabela 2: QM3 - Números de comparações (Cmp) e movimentações (Mov) para cada tipo de vetor.

Elementos	Cmp OrdC	Cmp OrdD	Cmp Ale	Mov OrdC	Mov OrdD	MovAle
50000	950011	950028	1197171	32767	57766	11198
100000	2000012	2000030	2520558	65535	115534	23283
150000	3056799	3056820	3863853	84464	159464	35807
200000	4200013	4200032	5272281	131071	231070	48668
250000	5250013	5250032	6716567	131071	256070	61715
300000	6413584	413606	8128260	168928	318928	74780
350000	7613584	7613606	9586627	218928	393928	87899
400000	8800014	8800034	11062436	262143	462142	101170
450000	9900014	9900034	12499991	262143	487142	115781
500000	11000014	11000034	13970269	262143	512142	128655

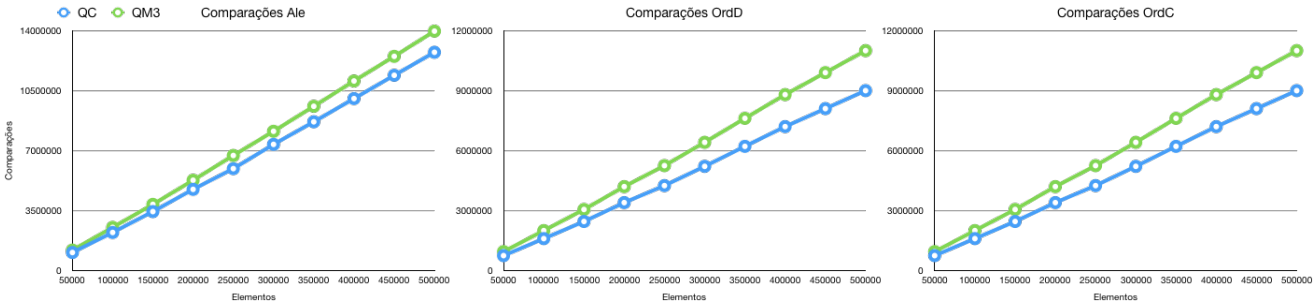

Figura 7: Comparações entre QC e QM3.

diretamente no custo temporal de execução. Portanto, a versão padrão do *quicksort* é melhor classificada em comparação à variação que seleciona o pivô através de uma mediana de três elementos.

4.6 QuickSort_Inserion - (QI1, QI5, QI10)

Para melhor detalhar a comparação entre estas três versões do algoritmo de ordenação, pode-se expor o comportamento assintótico das funções de acordo com o aumento do percentual de disparo do *insertion sort*. Tendo em vista que o algoritmo de ordenação por inserção é descrito por $O(n^2)$, à medida que se aumenta o percentual de elementos que serão ordenados por este método, o custo de $O(n \log n)$, que provém do *quicksort*, ficará cada vez mais invisível.

Outro aspecto de destaque refere-se à a forma de seleção do pivô, que nesta alternativa de ordenação é o mesmo usado em **QM3**. Por isso, pode-se observar que em (QI1) o comportamento dos gráficos é muito próximo ao comportamento de **QM3**.

Nesta variação do *quicksort* as disposições **OrdC** e **OrdD** se encaixam nos melhores casos. Isso ocorre uma vez que para **QM3**, o elemento central sempre será a mediana e não haverá partições degeneradas para esses tipos de organização do vetor. Sendo assim, as análises seguintes se baseiam apenas nos casos dos vetores com elementos dispostos aleatoriamente.

Como afirmado anteriormente, é observável a alteração do comportamento assintótico mais próximo de uma função quadrática, de acordo com o aumento no percentual de referência de acionamento do *insertion sort*. Por isso, é melhor que na necessidade de implementar um algoritmo misto entre ordenação por inserção e *quicksort* use-se percentuais mais baixos. É provável, ainda, que em uma implementação mista, o custo de execução seja menor do que na execução padrão, uma vez que para valores pequenos de n uma função quadrática se comporta de forma inferior à uma função $n \log n$.

A afirmação anterior vem propriamente da definição de comportamento assintótico, onde exista um m para o qual uma função $f(n)$ seja maior que uma função $g(n)$ para todo valor maior que m [1]. Usando $g(n)$ como o custo do *quicksort* $(n \log n)$ e $f(n)$ o custo do *insertion sort*, pode-se encontrar um valor de m para

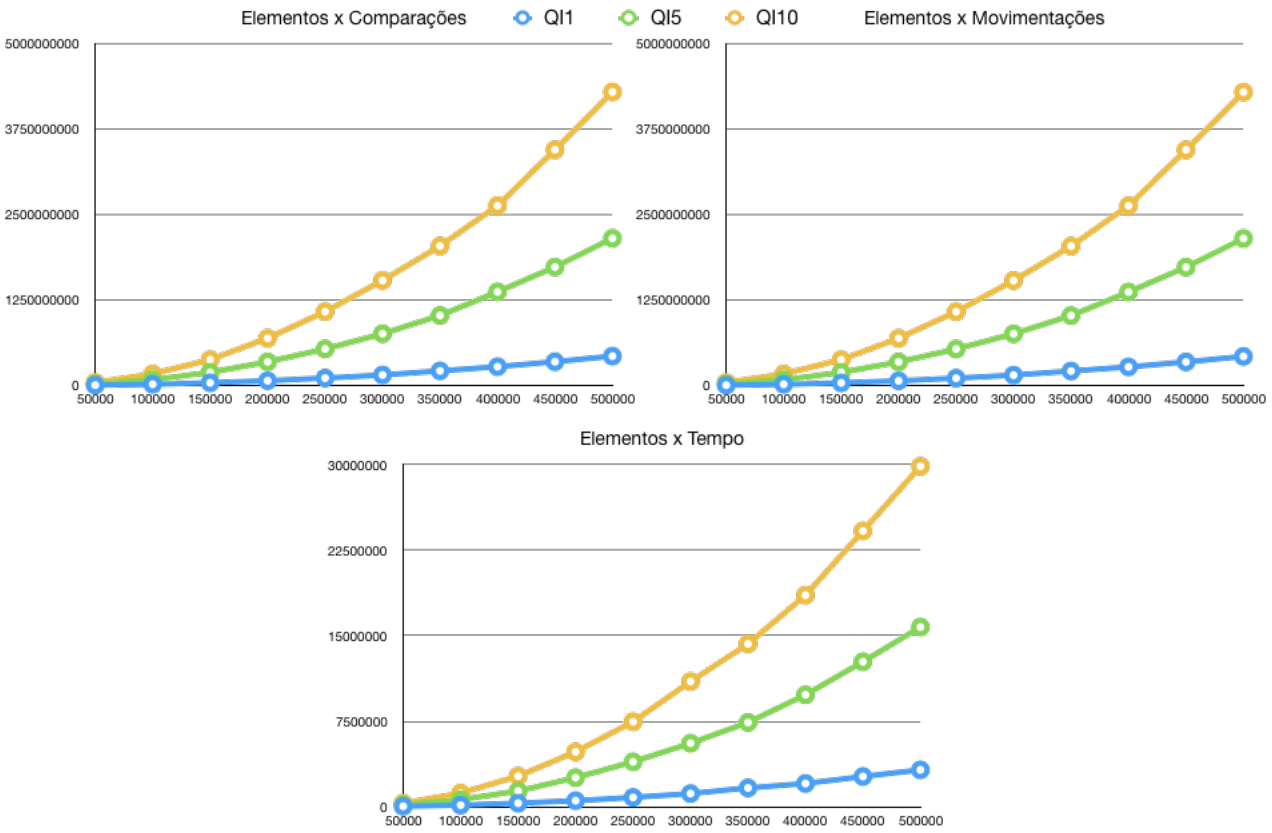

Figura 8: Comportamento dos custos.

o qual $f(m) < g(m)$. Então, na implementação mista, basta escolher o percentual de referência igual a m e terá-se uma melhora no desempenho do custo.

4.7 QuickSort_FirstElement - (QPE)

Esta implementação possui uma grande peculiaridade de funcionamento, em que uma má interpretação poderia gerar falsas afirmações. É sabido que, quando o pivô escolhido é o menor elemento do sub vetor, haverá um partição degenerada, por isso, por causa dos testes gerados por esse trabalho prático, este método aparenta ser o de pior desempenho. Isso se deve ao fato de que há a necessidade de analisar o comportamento do algoritmo para os casos **OrdC** e **OrdD**.

Para os casos em que o vetor está ordenado de forma crescente ou decrescente, nesta implementação, o algoritmo cairá em seu pior caso, que tem custo quadrático ($O(n^2)$). Para o vetor ordenado crescentemente é fácil inferir a afirmação, porém, para o caso decrescente, basta imaginar que na primeira passagem pela função **QuickSort::partition** o vetor será completamente ordenado de maneira crescente, porém particionado em dois outros sub vetores, que deste ponto em diante serão responsáveis pelo pior caso do *quicksort*.

De posse destas informações, podemos realizar algumas afirmações que serão comprovadas pelos gráficos a seguir. O comportamento da execução de **QPE** deve ser idêntico ou muito próximo ao comportamento de **QC** onde ambos são executados com um vetor **Ale** (Tabela 3). Outra afirmação a ser dada é que o gráfico do comportamento de **QPE** deve ser em forma de parábola côncava para cima para os casos de **OrdD** e **OrdC**(Figura 9).

As afirmações realizadas no parágrafo anterior se confirmam segundo os dados do gráfico e da tabela. Baseado nisso, podemos abstrair a aplicabilidade de **QPE** no sistema de biblioteca virtual de Arendelle. Visto que a biblioteca já existe em sua forma física, provavelmente os livros, pergaminhos e escritos já estejam parcialmente ordenados. Por isso, o uso da variante de ordenação, tratada nesse tópico, não seja

Tabela 3: Comparações(Cmp), Movimentações(Mov) e Tempo(Tmp) na execução das variações QC e QPE.

Elementos	**Cmp QC**	**Cmp QPE**	**Mov QC**	**Mov QPE**	**Tmp QC**	**Tmp QPE**
50000	1055213	1197171	199092	204713	11190	11198
100000	2224950	2520558	422178	432875	23345	23283
150000	3443971	3863853	653179	671617	35991	35807
200000	4740449	5272281	890846	913831	49707	48668
250000	5949097	6716567	1133245	1159349	63025	61715
300000	7364087	8128260	1374637	1411157	77249	74780
350000	8681277	9586627	1621346	1664804	90363	87899
400000	10034738	11062436	1872406	1920354	104233	101170
450000	11395118	12499991	2123002	2180037	118688	115781
500000	12737363	13970269	2376039	2441306	131901	128655

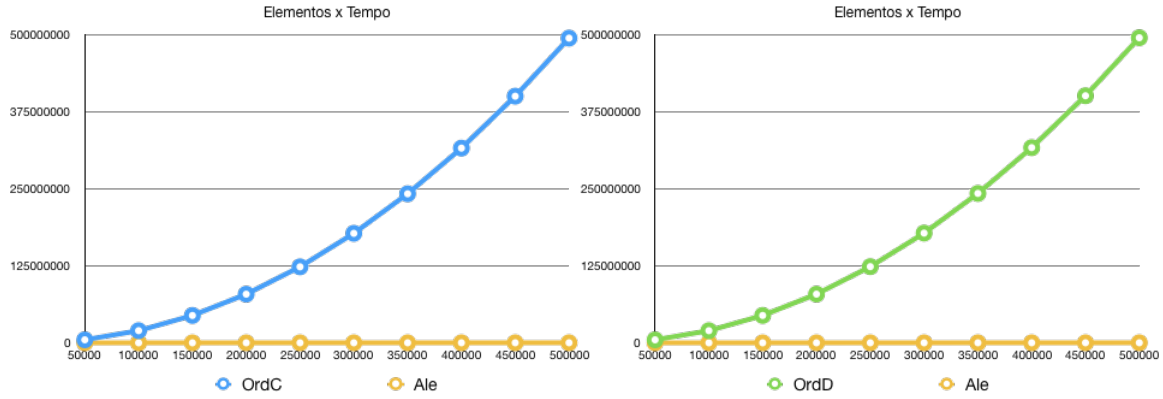

Figura 9: Comportamento assintótico de QPE para vetores OrdD e OrdC.

o ideal, dado que, quando o algoritmo encontra um sub vetor ordenado, ele cairá em seu pior caso e não executará o trabalho da maneira esperada (rápida e eficaz).

Com isso, nota-se que apesar da similaridade de **QPE** com a forma padrão, **QC**, para o caso de Arendelle, a escolha do primeiro elemento como pivô do sub vetor não é uma boa opção.

5 Conclusão

Por fim, ao analisar todo o trabalho desempenhado, conclui-se que há diversas maneiras de realizar a mesmo tarefa, porém o estudo da melhor alternativa é de grande valia para qualquer projeto. Das implementações do *quicksort* realizadas aqui, o **QuickSort (QC)** apresentou melhor desempenho segundo a análise dos casos médios e valores de mediana, e também no tratamento dos casos especiais, levando em consideração a forma inicial como os dados da bibliotaca física do reino está organizada. Sendo assim, este algoritmo será utilizado durante o elaboração da biblioteca digital de Arendelle.

Durante o desenvolvimento do trabalho, encontrou-se algumas dificuldades que foram superadas graças à ajuda dos monitores e da disponibilidade de recursos do laboratório *WINET*[16]. A primeira dificuldade encontrada relaciona-se ao limite do sistema para a pilha de recursão criada pelo programa; com isso, alguns casos de teste não conseguiam terminar sua execução pois necessitavam de mais espaço para as funções de recorrência. Após questionamento no fórum de dúvidas da matéria, os monitores foram solícitos ao dar a solução através do comando *ulimit*.

Quanto à segunda dificuldade encontrada, o problema consistia na demora do programa para o fim da execução de todos os duzentos e dez testes realizados para o caso de estudo. Por conta do grande número de variantes do *quicksort*, fez-se necessário alocar uma máquina que permanecesse ligada durante algum tempo sem interrupções; o aluno desenvolvedor deste trabalho, realiza tarefas de iniciação científica no laboratório *WINET* e solicitou neste local uma máquina que pudesse desempenhar o trabalho requisitado. Os companheiros de laboratório auxiliaram na disponibilização de uma máquina, e graças a isso concluiu-se todas as tarefas propostas para este segundo trabalho prático.

6 Referências

Referências

[1] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. 2001. *Introduction to Algorithms* (2nd Edition, p 44 - 45.). McGraw-Hill Higher Education.

[2] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. 2001. *Introduction to Algorithms* (2nd Edition, p. 145 - 152). McGraw-Hill Higher Education.

[3] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. 2001. *Introduction to Algorithms* (2nd Edition, p. 15 - 20). McGraw-Hill Higher Education.

[4] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. 2001. *Introduction to Algorithms* (2nd Edition, p. 200 - 201). McGraw-Hill Higher Education.

[5] https://code.visualstudio.com 09 jun. 2019

[6] https://www.tutorialspoint.com/cprogramming/c_data_types.htm Acessado em: 06 jun. 2019.

[7] http://www.cplusplus.com 09 jun. 2019

[8] Scott Meyers. 2005. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs* (3rd Edition, p. 170). Addison-Wesley Professional.

[9] Scott Meyers. 2005. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs* (3rd Edition, p. 156 - 161). Addison-Wesley Professional.

[10] http://www.cplusplus.com/reference/chrono Acesso em: 06 jun. 2019.

[11] https://www.gnu.org/software/gdb/ Acesso em: 06 jun. 2019.

[12] http://valgrind.org Acesso em: 06 jun. 2019.

[13] http://overleaf.com Acesso em: 06 de jun. 2019.

[14] http://www.sbc.org.br/documentos- da- sbc/summary/169- templates- para- artigos- e- capitulos- de- livros/ 878- modelosparapublicaodeartigos Acesso em: 06 jun. 2019.

[15] https://www.apple.com/br/numbers/ Acesso em: 09 jun. 2019

[16] http://www.winet.dcc.ufmg.br/doku.php Acesso em: 07 jun. 2019