Henrique Soares Assumpção e Silva
Matrícula: 2020006620
henriquesoares@dcc.ufmg.br

Trabalho Prático II
Estruturas de Dados
Universidade Federal de Minas Gerais

2021-08-11

1 Introdução

A empresa Reallocator CO. fez o contrato para o desenvolvimento de métodos de ordenação para os dados obtidos na rede mega-net, bem como a análise do tempo de execução desses métodos. A empresa deseja testar duas hipóteses relativas à aplicação destes métodos nos dados da rede, sendo a primeira delas relativa à estabilidade dos métodos na ordenação dos campos apresentados pelos dados, e a segunda relativa ao potencial impacto desses métodos no desempenho dos servidores de envio.
Essa documentação tem como objetivo especificar a implementação desses métodos de ordenação de forma detalhada, explicitando os aspectos mais relevantes da implementação, bem como concluindo que os métodos de fato seguem as especificações requisitadas e apresentando os resultados experimentais de modo a testar as hipóteses levantadas pela empresa.

2 Método

O programa foi desenvolvido na linguagem C++, compilada pelo compilador G++ da GNU Compiler Collection, no sistema operacional Windows 10. O programa foi inicialmente implementado e compilado em uma máquina com 16GB de RAM em um processador Intel Core I5-9400F.

2.1 Estrutura de dados

Para auxiliar no desenvolvimento do programa, criou-se a estrutura `Consc` no arquivo `consc.h`, desenvolvida em uma STRUCT, que representa um determinado indivíduo dentro da mega-net. Essa estrutura possui três atributos, sendo eles:

- BINARY_DATA: atributo de tipo `std::string` que armazena o campo DADOS das informações fornecidas. É utilizado para imprimir os valores no formato 'stdout'
- DATA: atributo de tipo `int` que armazena o campo DADOS das informações fornecidas. O valor armazenado em BINARY_DATA é convertido para `int`, para que as operações de ordenação possam ser realizadas
- NAME: atributo de tipo `string` que armazena o campo NOME das informações fornecidas

Essa estrutura apresenta apenas um método além dos construtores e destrutor, chamado `void print()`, que imprime os valores armazenados por uma instância de `Consc` no formato requisitado pelo enunciado.

2.2 Especificação de funções e métodos utilizados

Foram criadas implementações de quatro métodos de ordenação para a realização deste projeto, de acordo com as implementações vistas em sala de aula [1] e de acordo com referências bibliográficas externas [2]. Para a realização da ordenação dos campos NOME exigida pelo enunciado, implementou-se uma função de comparação por ordem alfabética, que será descrita a seguir:

- `bool compare_strings_alphabet(const string &s1, const string &s2, int compare_mode)`: Função que retorna o resultado da comparação alfabética entre duas `strings`, de acordo com um modo de comparação definido pela variável `compare_mode`. A função excuta um comando `switch` sobre essa mesma variável, que permite com que o usuário escolha entre as seguintes operações lógicas: $\{<,>,=,\leq\}$.

Após a definição desta métrica de comparação, implementou-se as funções que realizam os algoritmos de ordenação:

- **Merge sort**: Este algoritmo de ordenação é um exemplo de um algoritmo por divisão e conquista. Seu funcionamento consiste em realizar sucessivas divisões de um determinado array com respeito ao seu ponto médio, e então ordenar e agrupar os vários sub-arrays em um único array ordenado. Devido às especificações do enunciado, esse algoritmo ordena o array de acordo com o atributo NAME de cada

objeto do tipo `Consc` na ordem alfabética, no entanto, por motivos que serão explicados futuramente nesta documentação, também é possível ordenar o array de acordo com o atributo DATA.
Para realizar sua implementação, foram criadas duas funções:

- `void merge(Consc *vec, int left, int middle, int right, bool order_by_data = false)`:
  Função que realiza a fusão(merge) ordenada entre dois sub-arrays. Recebe como entrada um array de objetos do tipo `Consc`, três valores inteiros que indicam o início, o meio e o fim do array em questão, e um valor booleano indicando qual campo deverá ser considerado para a ordenação, que por padrão é o atributo NAME. Para realizar a ordenação, essa função chama a função `compare_strings` para realizar a operação $\leq$ de acordo com a ordem alfabética.
- `void merge_sort(Consc *vec, int begin, int end, bool order_by_data = false)`:
  Função principal do algoritmo, que recebe um array de objetos do tipo `Consc`, dois valores inteiros que indicam o início e o fim da seção do array que se deseja ordenar, e um valor booleano indicando qual campo deverá ser considerado para a ordenação, que por padrão é o atributo NAME. Essa função encontra o ponto médio do array de entrada e realiza chamadas recursivas para os respectivos sub-arrays. Por fim, chama a função `merge` e encerra.

- **Heapsort**: Este algoritmo de ordenação ordena um determinado conjunto de elementos por meio da criação de um HEAP [2] - estrutura de dados baseada em árvores que garantem uma ordenação entre as chaves dos nós - e da realização de sucessivas checagens da validade de sua estrutura, de modo a garantir com que o conjunto se encontre ordenado após estas diversas operações. Devido às especificações do enunciado, esse algoritmo ordena o array de acordo com o atributo DATA de cada objeto do tipo `Consc`.
  Para realizar sua implementação, foram criadas três funções:

  - `void remake(int left, int right, Consc *vec)`:
    Função que auxilia na criação de um HEAP válido. Recebe como entrada um array de objetos do tipo `Consc` e dois valores inteiros que indicam o início e o fim do sub-array a ser transformado, onde o início será considerado como o nó pai do HEAP. A função realiza a ordenação do sub-array de modo a garantir que este será um HEAP válido.
  - `void create_heap(Consc *vec, int n)`:
    Função que transforma um determinado array em um HEAP válido. Recebe como entrada um array de objetos do tipo `Consc` e um valor inteiro indicando o tamanho do array. Realiza chamadas sucessivas à função `remake` até que o array esteja ordenado de acordo com a lógica de um HEAP.
  - `void heap_sort(Consc *vec, int *n)`:
    Função principal do algoritmo, que recebe como entrada um array de objetos do tipo `Consc` e um valor inteiro indicando o tamanho do array. Essa função chama inicialmente a função `create_heap` e então realiza sucessivas chamadas à função `remake`, até que o array de entrada esteja ordenado.

- **Quicksort**: Este algoritmo de ordenação é outro exemplo de um algoritmo por divisão e conquista. Seu funcionamento consiste em realizar sucessivas partições do array original, e a cada passo ordenar parcialmente o array, até que no final se obtenha um array totalmente ordenado. Devido às especificações do enunciado, esse algoritmo por padrão ordena o array de acordo com o atributo NAME de cada objeto do tipo `Consc` na ordem alfabética.
  Para realizar a sua implementação, foram criadas três funções:

  - `void partition(int left, int right, int *i, int *j, Consc *vec)`:
    Função que faz a partição do array em dois sub-arrays, de modo que um contendo somente elementos menores ou iguais ao outro. Recebe como entrada dois valores inteiros indicando o início e o fim do sub-array que será analisado, dois ponteiros para inteiros que atuam como índices das partições, um array de objetos do tipo `Consc`. A função ordena os objetos de acordo com o atributo NAME, chamando a função `compare_strings_alphabet` para executar as operações $<$ e $>$.
  - `void sort(int left, int right, Consc vec)`:
    Função que realiza a ordenação dos dados de forma recursiva. Recebe como entrada um array de objetos do tipo `Consc`, dois valores inteiros, indicando o início e o fim da seção do array a ser ordenada. Realiza uma partição inicial e então faz uma chamada recursiva, primeiramente sobre o sub-array da esquerda e após isso sobre o sub-array da direita.
  - `void quick_sort(Consc vec, int n)`:
    Função que realiza o algoritmo sobre todo o array. Recebe como entrada um array de objetos do tipo `Consc`, um valor inteiro indicando o tamanho do array. Apenas chama a função `sort` com os parâmetros `left` $= 0$, `right` $= n - 1$.

- **Radix Exchange sort**: Este algoritmo de ordenação faz parte dos métodos de ordenação sem comparação, isto é, ele não realiza a comparação entre números completos e sim entre seus dígitos, de modo a ordená-los começando do dígito menos significativo até o mais significativo. Este algoritmo equivale a realizar um **Counting sort** [2] para cada dígito de um determinado conjunto de números. Para realizar sua implementação, foram criadas três funções:
  - `void max_value(Consc *vec, int n`:
    Função auxiliar que recebe como entrada um array de objetos do tipo `Consc` e um valor inteiro indicando o tamanho do array. Retorna o maior valor contido no array de acordo com o campo DATA.
  - `void count_sort(Consc *vec, int n, int exp)`:
    Função que realiza o **Counting sort** para um determinado dígito do conjunto de valores. Recebe como entrada um array de objetos do tipo `Consc` e dois valores inteiros, um indicando o tamanho do array e outro indicando qual dígito deverá ser considerado para a ordenação. Essa função inicialmente conta o número de ocorrências de cada valor do dígito `exp` dos objetos contidos em `vec`, e então modifica o array que contém as contagens, de modo a garantir que cada valor seja mapeado a um novo array ordenado.
  - `void radix_exchange_sort(Consc *vec, int n)`:
    Função principal do algoritmo, que recebe como entrada um array de objetos do tipo `Consc` e um valor inteiro indicando o tamanho do array. Encontra o maior elemento do array por meio da função `max_value`, e então chama a função `count_sort` um número de vezes igual ao número de dígitos do maior elemento, fazendo com que o array esteja ordenado ao fim do programa.

Por fim, por motivos que serão esclarescidos futuramente nesta documentação, criou-se uma função que, após realizada a ordenação do array principal de acordo com o campo DATA e NAME, procura sequências de valores objetos `Consc` com valores NAME iguais e os ordena de acordo com seus atributos DATA. Isso garante que grupos de indivíduos com nomes iguais estejam ordenados em ordem crescente de acordo com os valores armazenados em DATA. A função criada é descrita a seguir:

- `void sort_sub_groups(Consc *vec, int n)`:
  Função que recebe como entrada um array de objetos do tipo `Consc` e um valor inteiro indicando o tamanho do array. Percorre esse array procurando por grupos de objetos com atributos NAME iguais, e então chama a função `merge_sort` sobre esse sub-array, passando o valor `order_by_data = true`, ou seja, é realizado um **Merge sort** sobre esse sub-array, de modo a garantir que ele esteja ordenado de acordo com o campo DATA.

2.3 Estrutura do Programa principal

O programa principal, contido no arquivo `main.cpp`, recebe três parâmetros de entrada do usuário através da linha de comando. Assim como descrito no enunciado, o usuário deve fornecer o caminho para o arquivo contendo as informações, qual configuração deve ser utilizada para ordenar os dados e quantas linhas devem ser lidas do arquivo.
Após processar estes argumentos e checar sua validade, e.g. checar que a configuração para ordenação existe, o programa lê o número indicado de linhas do arquivo contendo os dados e então inicia um comando `switch` sobre a configuração indicada pelo usuário. Essa configuração é armazenada em uma variável de tipo `unsigned int` que pode assumir os valores 1,2,3,4, que indicam o número da configuração a ser utilizada. Para cada valor possível, os métodos de ordenação respectivos são chamados, primeiro com respeito ao campo DADOS e após isso com respeito ao campo NOME. Por fim, os elementos do array são imprimidos e o array de objetos de tipo `Consc` tem seus valores deletados.

3 Análise de Complexidade

Por sua simplicidade, optou-se por omitir a análise de complexidade da estrutura `Consc` descrita na seção 2.1. Essa estrutura possui somente um método, que percorre o array e imprime seus valores no padrão ‘‘`stdout`’’ exigido pelo enunciado, possuindo portanto ordem de complexidade de tempo de $\mathcal{O}(n)$ e ordem de complexidade de espaço constante.
Define-se n como o tamanho do array contendo objetos do tipo `Consc`.

3.1 Funções

3.1.1 Análise de Tempo

Primeiramente, irá se analisar a complexidade de tempo das funções auxiliares:

- `bool compare_strings_alphabet(const string &s1, const string &s2, int compare_mode)`:
  Seja δ o tamanho da menor string dada como entrada da função. A função irá realizar δ comparações de ordem $\mathcal{O}(1)$, portanto sua complexidade será $\mathcal{O}(\delta)$.

- `void sort_sub_groups(Consc *vec, int n)`:
  Esta função percorre o array de objetos do tipo `Consc`, encontrando sub-arrays de atributo NAME igual e então chamando a função `merge_sort` sobre esse sub-array. Assim sendo, no pior caso essa função receberá um array que possui todos os seus elementos com atributos NAME iguais, e então a função deverá percorrê-lo, algo que é da ordem de $\mathcal{O}(n)$, e então realizará um **Merge sort** sobre esses valores, que é da ordem de $\mathcal{O}(n \cdot \log(n))$. Portanto, a complexidade final da função no pior caso será de:

$$\mathcal{O}(n) + \mathcal{O}(n \cdot \log(n)) = \mathcal{O}(n \cdot \log(n)) \tag{1}$$

Agora, irá se analisar a complexidade de tempo das funções de ordenação:

- **Merge sort**: Sabe-se que este algoritmo tem ordem de complexidade de tempo de $\mathcal{O}(n \cdot \log(n))$ [2], e a implementação feita também possui essa ordem de complexidade.
  - `void merge(Consc *vec, int left, int middle, int right)`:
    A operação de merge realiza n operações de ordem $\mathcal{O}(1)$, ou seja, possui ordem de complexidade de tempo linear $\mathcal{O}(n)$.
  - `void merge_sort(Consc *vec, int begin, int end)`:
    Como esta é a função principal do algoritmo, sabemos que ela possui equação recursiva da forma $T(n) = 2T(\frac{n}{2}) + n$, que possui solução igual a $\mathcal{O}(n \cdot \log(n))$.

- **Heapsort**: Sabe-se que este algoritmo tem ordem de complexidade de tempo de $\mathcal{O}(n \cdot \log(n))$ [2], e a implementação feita também possui essa ordem de complexidade.
  - `void remake(int left, int right, Consc *vec)`:
    No pior caso desta função, ela percorre todo o galho de uma árvore binária, ou seja, executa $\mathcal{O}(log(n))$ operações.
  - `void create_heap(Consc *vec, int n)`:
    Essa função chama a função `remake` para os nós internos da árvore, ou seja, faz $\frac{n}{2} \cdot \log(n)$ operações, sendo portanto da ordem de $\mathcal{O}(n \cdot \log(n))$.
  - `void heap_sort(Consc *vec, int *n)`:
    No pior caso, essa função chama a função `remake` $n - 1$ vezes e a função `create_heap` uma vez, ou seja, faz $(n-1) \cdot \mathcal{O}(n) + \mathcal{O}(n \cdot \log(n))$, portanto sua ordem de complexidade de tempo é de $\mathcal{O}(n \cdot \log(n))$.

- **Quicksort**: Sabe-se que este algoritmo tem ordem de complexidade de tempo de $\mathcal{O}(n \cdot \log(n))$ [2] no seu melhor caso e complexidade de $\mathcal{O}(n^2)$ [2] no pior caso. A implementação feita também possui essas características.
  - `void partition(int left, int right, int *i, int *j, Consc *vec, bool order_by_data = false)`:
    A operação é sempre de ordem linear, pois sempre faz um número de operações que é uma fração de n, ou seja, possui ordem de complexidade de tempo de $\mathcal{O}(n)$.
  - `void sort(int left, int right, Consc vec,bool order_by_data = false)`:
    Essa função faz uma implementação recursiva do algoritmo em questão. No pior caso, o pivo é sistematicamente escolhido como um dos extremos do array, fazendo com que a função tenha ordem de complexidade de $\mathcal{O}(n^2)$. O melhor caso ocorre quando o array é consistentemente dividido em dois, fazendo com que a função tenha ordem de complexidade de $\mathcal{O}(n \cdot \log(n))$.
  - `void quick_sort(Consc vec, int n,bool order_by_data = false)`:
    Como essa função apenas chama a função `sort`, ela também tem ordem de complexidade de tempo de $\mathcal{O}(n \cdot \log(n))$ no seu melhor caso e complexidade de $\mathcal{O}(n^2)$ no pior caso.

- **Radix Exchange sort**: Sabe-se que este algoritmo tem ordem de complexidade de tempo de $\mathcal{O}(n \cdot \log(n))$ [2], e a implementação feita também possui essa ordem de complexidade. Define-se k como sendo o valor do maior elemento, obtido pela função `max_value`.

  - `void max_value(Consc *vec, int n`:
    Essa função percorre o array em busca do seu maior valor com respeito ao atributo DATA, ou seja, faz n comparações de ordem $\mathcal{O}(1)$, portanto sua ordem de complexidade é de $\mathcal{O}(n)$.
  - `void count_sort(Consc *vec, int n, int exp)`:
    Sabe-se que o algoritmo **Counting sort** possui ordem de complexidade de tempo dada por $\mathcal{O}(n+k)$, pois ele realiza uma passagem pelo array de tamanho n e uma passagem pelo array auxliar de tamanho k.
  - `void radix_exchange_sort(Consc *vec, int n)`:
    Essa função realiza uma chamada à função `max_value` e k chamadas à função `count_sort`, ou seja, realiza k operações de ordem $\mathcal{O}(n + k)$ e uma operação de ordem $\mathcal{O}(k)$, se considerarmos $k = \log(n)$, conclui-se que a complexidade de tempo final do algoritmo é da ordem de $\mathcal{O}(n \cdot \log(n))$.

3.1.2 Análise de Espaço

Dentre todos as funções descritas na seção 2.2, apenas as funções relacionadas aos algoritmos **Radix Exchange sort** e **Merge sort** requisitam de estruturas auxiliares para sua execução, ou seja, todas as funções que não estão relacionadas à esses algoritmos possuem ordem de complexidade de espaço constante $\mathcal{O}(1)$. Agora, será feita uma análise da complexidade de espaço dos algoritmos que possuem complexidade não-linear:

- **Merge sort**: Sabe-se que este algoritmo tem ordem de complexidade de espaço de $\mathcal{O}(n)$ [2], e a implementação feita também possui essa ordem de complexidade.

  - `void merge(Consc *vec, int left, int middle, int right)`:
    A operação de merge cria dois arrays auxiliares, que possuem um tamanho que depende de n, ou seja, a complexidade de espaço desta função será da ordem de $\mathcal{O}(n)$.
  - `void merge_sort(Consc *vec, int begin, int end)`:
    Esta é a função principal do algoritmo e divide o array principal em diversos sub-arrays de forma recursiva. Ela requer uma memória adicional da ordem de $\mathcal{O}(n)$ para realizar a ordenação do array principal.

- **Radix Exchange sort**: Sabe-se que este algoritmo tem ordem de complexidade de espaço de $\mathcal{O}(n \cdot \log(n))$ [2], e a implementação feita também possui essa ordem de complexidade. Define-se k como sendo o valor do maior elemento, obtido pela função `max_value`.

  - `void max_value(Consc *vec, int n`:
    Essa função não requer memória adicional para buscar o maior elemento com respeito ao atributo DATA do array, ou seja, possui ordem de complexidade de espaço constante.
  - `void count_sort(Consc *vec, int n, int exp)`:
    Esta função cria um array auxiliar de tamanho k para realizar a ordenação de um determinado dígito, ou seja, possui ordem de complexidade de espaço de $\mathcal{O}(k)$.
  - `void radix_exchange_sort(Consc *vec, int n)`:
    Essa função realiza uma chamada à função `max_value` e k chamadas à função `count_sort`, ou seja, realiza k operações de ordem $\mathcal{O}(n)$ e uma operação de ordem $\mathcal{O}(1)$, se considerarmos $k = \log(n)$, conclui-se que a complexidade de tempo final do algoritmo é da ordem de $\mathcal{O}(n \cdot \log(n))$.

Além disso, sabe-se que a função `sort_sub_groups` executa um **Merge sort** sobre os dados, ou seja, ela também possui ordem de complexidade de espaço da ordem de $\mathcal{O}(n)$.

3.2 Programa Principal

Define-se N como sendo o valor dado pelo usuário para o número de linhas a serem lidas do arquivo contendo as informações da Reallocator CO.

3.2.1 Análise de Tempo

Primeiramente, são lidas as primeiras N linhas do arquivo contendo as informações, algo que é da ordem de $\mathcal{O}(N)$. Além disso, a impressão dos valores do array - que também é chamada em todas as configurações - é da ordem de $\mathcal{O}(N)$. Após isso, existem quatro possibilidades de execução de acordo com a configuração escolhida:

1. **Configurações 1 e 2**: Ambas as configurações possuem algoritmos com a mesma ordem de complexidade de tempo. No melhor caso, ambos os algoritmos possuem complexidade de $\mathcal{O}(N \cdot \log(N))$, portanto serão feitas duas operações de ordem $\mathcal{O}(N \cdot \log(N))$ e duas operações de ordem $\mathcal{O}(N)$, ou seja, a ordem de complexidade de tempo desta configuração é de $\mathcal{O}(N \cdot \log(N))$. No pior caso, o **Quicksort** possui complexidade de $\mathcal{O}(N^2)$ e segundo algoritmo da configuração possui complexidade de $\mathcal{O}(N \cdot \log(N))$. Portanto, serão feitas duas operações de ordem $\mathcal{O}(N \cdot \log(N))$, uma operação de ordem $\mathcal{O}(N^2)$ e duas operações de ordem $\mathcal{O}(N)$, ou seja, ordem de complexidade de tempo desta configuração é de $\mathcal{O}(N^2)$.

2. **Configuração 3**: Ambos algoritmos dessa configuração possuem ordem de complexidade de tempo de $\mathcal{O}(N \cdot \log(N))$. Portanto, serão feitas duas operações de ordem $\mathcal{O}(N \cdot \log(N))$ e duas operações de ordem $\mathcal{O}(N)$, ou seja, a ordem de complexidade de tempo desta configuração é de $\mathcal{O}(N \cdot \log(N))$.

3. **Configuração 4**: Ambos algoritmos dessa configuração possuem ordem de complexidade de tempo de $\mathcal{O}(N \cdot \log(N))$. Portanto, serão feitas duas operações de ordem $\mathcal{O}(N \cdot \log(N))$ e duas operações de ordem $\mathcal{O}(N)$, ou seja, a ordem de complexidade de tempo desta configuração é de $\mathcal{O}(N \cdot \log(N))$.

3.2.2 Análise de Espaço

O programa aloca um array de objetos de tipo `Consc`, com um tamanho que depende de N, ou seja, independentemente da configuração escolhida o programa irá realizar uma operação de ordem de complexidade de espaço de $\mathcal{O}(N)$. A ordem de complexidade de espaço final do programa depende diretamente da configuração escolhida pelo usuário:

- **Configuração 1**: Como ambos os métodos desta configuração possuem ordem de complexidade de espaço constante, nota-se que a ordem final do programa será de $\mathcal{O}(N)$.

- **Configuração 2**: Dentre os métodos desta configuração, apenas o **Radix Exchange sort** possui ordem de complexidade de espaço não constante. Portanto, a ordem final do programa será:

$$\mathcal{O}(N) + \mathcal{O}(N \cdot \log(N)) = \mathcal{O}(N \cdot \log(N)) \tag{2}$$

- **Configuração 3**: Dentre os métodos desta configuração, apenas o **Merge sort** possui ordem de complexidade de espaço não constante. Portanto, a ordem final do programa será:

$$\mathcal{O}(N) + \mathcal{O}(N) = \mathcal{O}(N) \tag{3}$$

- **Configuração 4**: Ambos os métodos desta configuração possuem ordem de complexidade de espaço não constante. Portanto, a ordem final do programa será:

$$\mathcal{O}(N) + \mathcal{O}(N \cdot \log(N)) = \mathcal{O}(N \cdot \log(N)) \tag{4}$$

4 Configuração Experimental

O enunciado deste projeto fornece quatro configurações possíveis que devem ser implementadas para a ordenação dos valores apresentados, sendo elas:

1. **Quicksort** + **Heapsort**: Essa configuração consiste em aplicar o algoritmo **Quicksort** sobre o campo NOME das informações disponibilizadas, e então aplicar o algoritmo **Heapsort** sobre o campo DADOS destas informações.

2. **Quicksort** + **Radix Exchange sort**: Essa configuração consiste em aplicar o algoritmo **Quicksort** sobre o campo NOME das informações disponibilizadas, e então aplicar o algoritmo **Radix Exchange sort** sobre o campo DADOS destas informações.

3. **Merge sort** + **Heapsort**: Essa configuração consiste em aplicar o algoritmo **Merge sort** sobre o campo NOME das informações disponibilizadas, e então aplicar o algoritmo **Heapsort** sobre o campo DADOS destas informações.

4. **Merge sort** + **Radix Exchange sort**: Essa configuração consiste em aplicar o algoritmo **Merge sort** sobre o campo NOME das informações disponibilizadas, e então aplicar o algoritmo **Radix Exchange sort** sobre o campo DADOS destas informações.

Para avaliar o tempo de execução e eficiência das configurações possíveis, realizou-se o experimento destacado na seção 2.1 do enunciado, onde deve-se testar cada configuração possível sobre tamanhos variados de entrada contida no arquivo de homologação. Sendo assim, testou-se cada uma das quatro configurações possíveis para os valores de entrada $\{1000,10.000,50.000,100.000,200.000\}$, que representam o número de linhas do arquivo a serem lidas pelo programa principal. Para medir o tempo de execução de cada configuração, utilizou-se a biblioteca `chrono`[1] da linguagem C++, que permite a medição do tempo de execução de funções e procedimentos com grande acurácia.
Além disso, o enunciado do projeto afirma que "A ordenação[dos dados] deve ser feita em duas etapas. A primeira etapa ordena as informações considerando o campo DADOS. Posteriormente, com os dados já ordenados, deve-se ordená-los novamente considerando o campo NOME. Ao fim da ordenação, nomes iguais devem estar agrupados e ordenados. Além disso, os binários de cada agrupamento também deverão estar ordenados". Percebe-se que, no caso onde a configuração possua um algoritmo instável que realiza a ordenação dos nomes - mais especificamente nas configurações 1 e 2 - , não é possível atender à supracitada demanda. Por esse motivo, sugere-se a criação e uso da função `sort_sub_groups`, descrita na seção 2.2, que ordena os sub-arrays de mesmo nome de acordo com seus dados binários, utilizando o algoritmo **Merge sort**, de modo a garantir que essas configurações atendam os requisitos da empresa. Experimentos relativos à eficiência dessa função foram realizados e terão seus resultados apresentados na seção seguinte. Como espera-se que o programa produza uma saída de acordo com o previsto para todas as configurações, a função `sort_sub_groups` não é executada no programa principal, no entanto, sua implementação está disponível e esta documentação também procura argumentar pelos benefícios de seu uso.

5 Resultados

5.1 Estabilidade dos métodos

Além do experimento relativo ao tempo de execução, a estabilidade dos métodos utilizados também é de suma importância para o funcionamento do sistema proposto pela Reallocator CO. Segue uma análise da estabilidade de cada configuração possível:

1. **Quicksort** + **Heapsort**: Ambos os algoritmos dessa configuração são instáveis [2], ou seja, ambos os métodos irão inverter elementos duplicados quando comparados. Portanto, essa configuração é instável tanto com respeito ao campo DADOS quanto ao campo NOME. Portanto, esse método não garante que a ordenação final atenda as demandas enunciadas.

2. **Quicksort** + **Radix Exchange sort**: O algoritmo **Quicksort** é instável, ou seja, o método irá inverter duplicatas do campo NOME quando comparados. O algoritmo **Radix Exchange sort** é estável [2], ou seja, ele não inverte a ordem de elementos iguais quando comparados. Portanto, essa configuração é estável com respeito ao campo DADOS e instável com respeito ao campo NOME, e como a ordenação dos nomes é feita por último, esse método não garante que a ordenação final atenda as demandas enunciadas.

3. **Merge sort** + **Heapsort**: O algoritmo **Heapsort** é instável, ou seja, o método irá inverter duplicatas do campo DADOS quando comparados. O algoritmo **Merge sort** é estável [2], ou seja, ele não inverte a ordem de elementos iguais quando comparados. Portanto, essa configuração é estável tanto com respeito ao campo NOME e instável com respeito ao campo DADOS. e como a ordenação dos nomes é feita por último, esse método garante que a ordenação final atenda as demandas enunciadas, pois como o **Merge sort** é estável, a ordem relativa dos dados binários será mantida.

4. **Merge sort** + **Radix Exchange sort**: Ambos os algoritmos dessa configuração são estáveis , ou seja, ambos os métodos não irão inverter elementos duplicados quando comparados. Portanto, essa configuração é estável tanto com respeito ao campo DADOS quanto ao campo NOME. Portanto, esse método garante que a ordenação final atenda as demandas enunciadas.

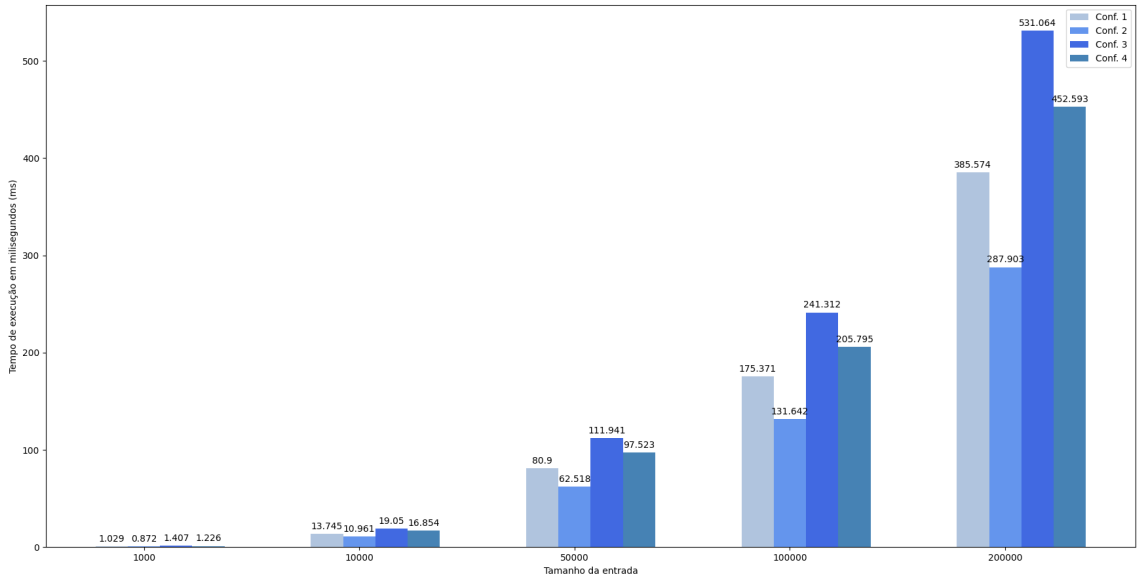
Portanto, percebe-se que as configurações 1, 2 e 3 são parcialmente ou totalmente instáveis, porém somente as configurações 1 e 2 não produzem saídas que atendem os requisitos enunciados.

5.2 Tempo de execução

A figura 1 mostra o resultado do experimento descrito na seção 2.1 do enunciado, sem levar em consideração a execução da função `sort_sub_groups`, ou seja, essa figura mostra o tempo de execução das configurações sem executar a operação de ordenar os dados de agrupamentos com nomes iguais. Observa-se que:

[1] https://www.cplusplus.com/reference/chrono/

Figure 1: Gráfico de barras dos resultados obtidos ao se testar as 4 configurações possíveis para diferentes tamanhos da entrada N, sem a execução da função `sort_sub_groups`. Está é a versão final do programa, e é possível observar que, a medida que N cresce, a configuração 2 (**Quicksort + Radix Exchange sort**) se mostra mais eficiente dentre as configurações possíveis.

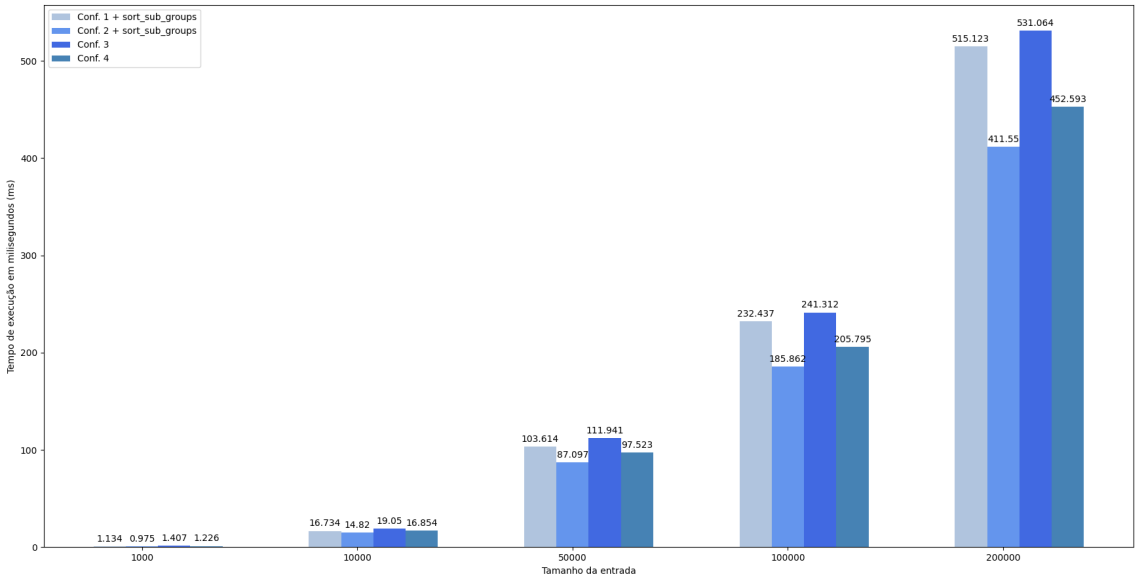


- Para $N = 1000$, apesar de todas as configurações apresentarem um tempo de execução próximo, a configuração 2 foi a mais eficiente, necessitando de 0.872ms para ordenar a entrada. O tempo médio de execução foi de 1.13ms e a configuração 3 tomou o tempo máximo de 1.407ms para ordenar a entrada.
- Para $N = 10.000$, a configuração 2 foi a mais eficiente, necessitando de 10.961ms para ordenar a entrada. O tempo médio de execução foi de 15.15ms e a configuração 3 tomou o tempo máximo de 19.05ms para ordenar a entrada.
- Para $N = 50.000$, a configuração 2 foi a mais eficiente, necessitando de 62.518ms para ordenar a entrada. O tempo médio de execução foi de 88.22ms e a configuração 3 tomou o tempo máximo de 111.941ms para ordenar a entrada.
- Para $N = 100.000$, a configuração 2 foi a mais eficiente, necessitando de 131.642ms para ordenar a entrada. O tempo médio de execução foi de 188.53ms e a configuração 3 tomou o tempo máximo de 241.312ms para ordenar a entrada.
- Para $N = 200.000$, a configuração 2 foi a mais eficiente, necessitando de 284.903ms para ordenar a entrada. O tempo médio de execução foi de 414.28ms e a configuração 1 tomou o tempo máximo de 531.064ms para ordenar a entrada.

A figura 2 mostra o tempo de execução de todas as configurações possíveis considerando a execução da função `sort_sub_groups` para as configurações 1 e 2 ,que garante que agrupamentos de nomes iguais tenham seus dados ordenados. Apesar de obviamente aumentar o tempo de execução geral das configurações, observa-se que a configuração 2 ainda se mostra a mais eficiente para todos os tamanhos de entrada, ou seja, essa função não afeta a eficiência relativa das configurações. Observa-se que:

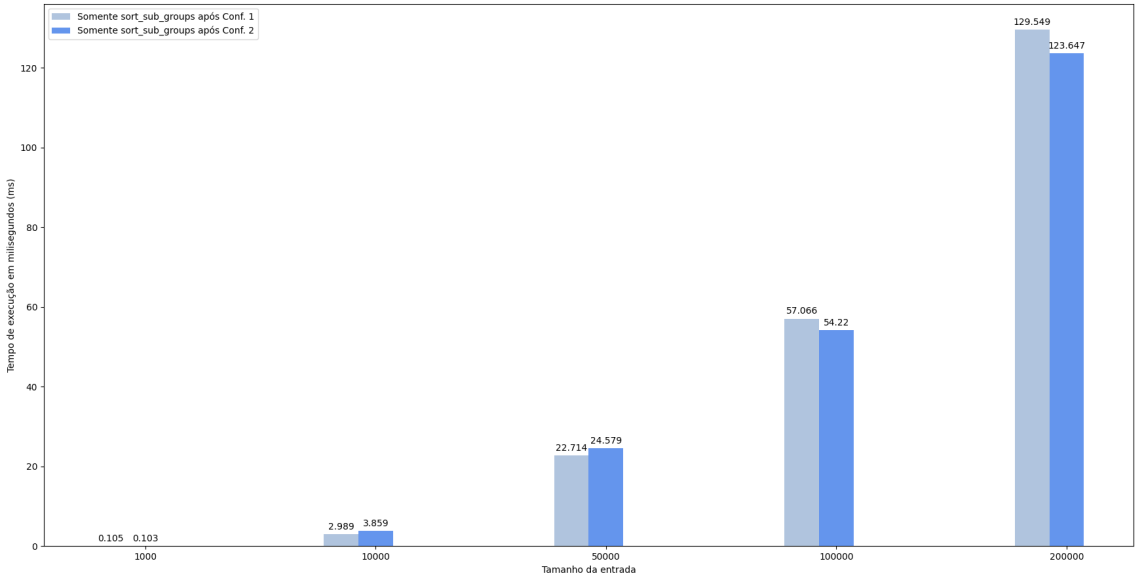
- Para $N = 1000$, apesar de todas as configurações apresentarem um tempo de execução próximo, a configuração 2 foi a mais eficiente, necessitando de 0.975ms para ordenar a entrada. O tempo médio de execução foi de 1.18ms e a configuração 3 tomou o tempo máximo de 1.407ms para ordenar a entrada.
- Para $N = 10.000$, a configuração 2 foi a mais eficiente, necessitando de 14.82ms para ordenar a entrada. O tempo médio de execução foi de 16.86ms e a configuração 3 tomou o tempo máximo de 19.05ms para ordenar a entrada.
- Para $N = 50.000$, a configuração 2 foi a mais eficiente, necessitando de 87.097ms para ordenar a entrada. O tempo médio de execução foi de 100.04ms e a configuração 3 tomou o tempo máximo de 111.941ms para ordenar a entrada.

Figure 2: Gráfico de barras dos resultados obtidos ao se testar as 4 configurações possíveis para diferentes tamanhos da entrada N, com a execução da função `sort_sub_groups` ao fim de todas as configurações. É possível observar que, a medida que N cresce, a configuração 2 (**Quicksort + Radix Exchange sort**) continua sendo a mais eficiente dentre as configurações possíveis.



- Para $N = 100.000$, a configuração 2 foi a mais eficiente, necessitando de 185.862ms para ordenar a entrada. O tempo médio de execução foi de 216.35ms e a configuração 3 tomou o tempo máximo de 241.312ms para ordenar a entrada.
- Para $N = 200.000$, a configuração 2 foi a mais eficiente, necessitando de 411.55ms para ordenar a entrada. O tempo médio de execução foi de 477.58ms e a configuração 3 tomou o tempo máximo de 531.064ms para ordenar a entrada.

Figure 3: Gráfico de barras das diferenças entre o tempo de execução dos resultados da figura 1 e da figura 2. Esse gráfico essencialmente mostra o tempo de execução da função `sort_sub_groups` após a execução das configurações 1 e 2, i.e. configurações que necessitam da terceira ordenação para atenderem os requisitos enunciados, a medida que o tamanho da entrada N cresce.



A figura 3 mostra a diferença nos tempos de execução das configurações dos resultados da figura 1 e da figura 2, o que representa o tempo de execução da função `sort_sub_groups`. Como essa função executa o algoritmo

Merge Sort após a ordenação dos dados de acordo com dois outros algoritmos da configuração em questão, não é possível observar um padrão claro no tempo de execução desta função à medida que o tamanho da entrada cresce.
Também é possível perceber, por meio da análise dos resultados obtidos na seção 3.2, que a execução desse terceiro método de ordenação não muda a ordem de complexidade de tempo do programa, pois todas as configurações apresentam algum algoritmo que possui ordem de complexidade de $\mathcal{O}(N \cdot \log(N))$, e como está é a ordem de complexidade da função `sort_sub_groups`, a ordem de complexidade de tempo final não é afetada. Isso significa que, à medida que N cresce, o impacto da supracitada terceira ordenação se torna cada vez menos significativo, algo que pode justificar o uso dessa função pela empresa.
Com relação à ordem de complexidade de espaço, observa-se que, como sempre é alocado um array de tamanho que depende de N, ou seja, independentemente da configuração é realizada uma operação de ordem $\mathcal{O}(N)$, a execução do terceiro método de ordenação também não afeta a ordem de complexidade de espaço final do programa, pois também realiza uma operação de ordem $\mathcal{O}(N)$.

6 Conclusões

Durante a execução deste trabalho, tomou-se por objetivo estudar os efeitos do uso de diversos algoritmos de ordenação diferentes no contexto de dados fornecidos pela empresa Reallocator CO. O projeto tinha como proposta principal analisar a estabilidade e eficiência destes algoritmos quando se é necessário ordenar uma entrada de tamanho variado.
Os experimentos realizados demonstram que a configuração 2 é claramente a mais eficiente quanto ao tempo de execução dentre todos os métodos testados, se mostrando significativamente mais rápida que as outras configurações para todos os tamanhos de entrada. No entanto, sabe-se que essa configuração é instável quanto ao campo NOME das informações fornecidas, e portanto não atende completamente aos requesitos da empresa.
Além disso, percebe-se que a utilização da função `sort_sub_groups` - que executa o algoritmo **Merge Sort** em grupos com nomes iguais, de modo a ordená-los de acordo com seus dados binários - não afeta de forma significativa o desempenho das configurações parcialmente ou totalmente instáveis, ou seja, o seu uso garante que a ordenação realizada atenda as demandas enunciadas e ao mesmo tempo não altera a ordem de complexidade final do programa, sendo portanto recomendada para o uso da empresa.
Assim sendo, percebe-se que o uso da configuração 2 seguido da utilização da função `sort_sub_groups` apresentou o menor tempo de execução, além de atender as demandas da empresa, tendo em vista que a ordenação dos grupos realizadas por essa função garante que grupos com nomes iguais estarão ordenados com respeito aos seus dados binários. Portanto, recomenda-se que a empresa utilize a configuração 2 seguida da função `sort_sub_groups`.
Apesar disso, o uso da configuração 4 também é recomendado, pois ela necessita apenas de duas ordenações para garantir que os dados estejam ordenados de acordo com os requisitos da empresa. Apesar de possuir um tempo de execução ligeiramente maior que a configuração 2, essa configuração atinge o objetivo de forma mais simples, além de possuir uma ordem de complexidade de tempo de pior caso menor que a configuração 2.
A implementação dos métodos de ordenação foi desafiadora e proporcionou um melhor entendimento sobre como estes métodos funcionam em situações práticas, além de incentivar a comparação entre tais métodos em um contexto real com um conjunto de dados relativamente grande. Em geral, a experiência trouxe grande aprendizado e sem dúvida colaborou para o maior entendimento do funcionamento destes algoritmos.

References

[1] Chaimowicz, L. and Prates, R. (2020). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via moodle. Departamento de Ciencia da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

[2] Ziviani, N., Projeto de Algoritmos com Implementações em Pascal e C, 2ª Edição, Editora Thomson, 2004.

7 Instruções para compilação e execução

Os arquivos foram estruturados de acordo com o exigido pelo enunciado, ou seja, para executar o programa principal basta seguir os seguintes passos:

- Acesse o diretório `TP/`
- Utilizando um terminal, execute o arquivo `Makefile`, através do comando `make`

- Após isso, execute os seguintes passos de acordo com o sistema operacional utilizado:
  - No terminal Linux, execute o comando:
    `./bin/run.out <TARGET_FILE> <MODE> <NUM_LINES>`
  - No prompt de comando do Windows, execute o comando:
    `.\bin\run.out <TARGET_FILE> <MODE> <NUM_LINES>`

  Onde <TARGET_FILE> indica o nome do arquivo com os comandos, <MODE> indica qual configuração deverá ser utilizada na ordenação e <NUM_LINES> indica o número de linhas a serem lidas do arquivo principal.
- Por fim, execute o comando `make clean` para remover os arquivos `.o` e `.out` gerados. Por limitações do próprio `Make`, esse comando pode não funcionar corretamente em sistemas Windows. Nesse caso, recomenda-se que se delete os arquivos compilados manualmente.

Todo o código deste projeto, bem como esta documentação, serão disponibilizados para o livre acesso e manipulação em um repositório do GitHub[2], sob a licensa THE UNLICENSE.

[2]https://github.com/HenrySilvaCS/Data-Structures-UFMG/tree/main/Second%20Project