Algortimos de Ordenação

Elysson A. Lacerda[1], **Gustavo A. Monteiro**[2]

[1]Universidade Federal do Ceará (UFC)
Quixadá - Ceará - Brazil

Abstract. *This document consists of describing the implementation of sorting algorithms formed by doubly linked lists. Work developed in C++ and uses Object Orientation concepts.*

Resumo. *Este documento consiste em descrever a implementação de algortimos de ordenação formada por listas duplamente encadeadas. Trabalho desenvolvido em* ***C++*** *e utiliza conceitos de Orientação a Objeto.*

1. Apresentação

Os algoritmos de ordenação são amplamente utilizados em **Ciência da Computação** para organizar conjuntos de dados de forma crescente ou descrescente.Entender dados e saber como ordená-los de forma eficiente é fundamental em muitas áreas como por exemplo **Bando de Dados**, **Algoritmos de Busca**, **Processamento de imagens**, **Análise de dados** e muitas outras, escolher um algoritmo de ordenação será baseado nas carcterísticas dos dados e nos requisitos de desempenho.

Algoritmos de Ordenação como o Buble Sort, Insertion Sort e Selection Sort são geralmente adequados para conjuntos de dados menores ou parcialmente ordenados, devido a sua eficiência ser limitada em casos médio e piores, podendo ser úteis para situações em que a simplicidade do código é mais importante do que sua eficiência.

Por outro lado. algoritmos como Merge Sort e o Quick Sort são mais eficientes em lidar com grandes conjuntos de dados desordenados, Merge sorte apresenta uma complexidade chamativa para ordenação geral enquanto o Quick Sort tem um boa efiência média e frequentemente o mais rápido, embora perca desempenho em casos extremos.

1.1. Lista Duplamente Encadeada

Uma **Lista Duplamente Encadeada** é uma estrutura de dados no qual os elementos são organizados em nós que possuem referências tanta para o próximo nó quanto para o nó anterior.Essa estrutura permite acesso de ambas as direções aos elementos, podendo ser útil em muitas situações.Cada nó de uma lista desse tipo contém um valor e dois ponteiros : um para o nó anterior e outro para o próximo nó, sendo o primeiro da lista chamado de ”cabeça” e o último de ”cauda”, os ponteiros anteriores e o próximo do nó cabeça geralmente são nulos ou apontam para sentinelas especiais.

Uma das principais vantagens dessas listas é a capacidades de percorrer a lista nos dois sentidos, facilitando operações como a remoção de um elemento específico, uma vez que é possível atualizar os pontiros anterios e o próximo dos nós adjacentes de maneira aficiente.Além disso, a inserção e remoção podem ter uma complexidade boa, desde que o nó no qual a operação ocorra, seja conhecido, mas, o uso requer mais espaço de memória

em comparação com outros tipos de listas devido a cada nó armazenar duas referências e sua manipulação requer atenção para não quebrar a lista.

Exemplos de aplicação seria em um sistema de gerenciamento de memória, processamento de texto, implementação de cache e muitas outras, o acesso de ambas as direções é valioso em certos contextos dependendo de cada problema.

1.2. Incentivo

Para criar uma implementação que mistura ambos os conceitos, é necessário cria um **List.h** que representerá a lista duplamente encadeada, incluindo a estrutura de nó que contém os elementos da lista e as referências para o próximo e o nó anterior e os métodos públicos da classe. Para implementação dos métodos declarados, é criado a **List.cpp** que seriam as operações da lista e funções auxiliares.Na **Main.cpp** é a porta de entrada da implementação, instânciar uma lista e preenchê-la de elementos e chamar os métodos de ordenação para testar o funcionamento do algortimos.

2. Classe List

Nessa classe está o esqueleto da aplicação.

2.1. List.h

Aqui foram definidos os algoritmos e as funções que serviram para ajudar a operar com a lista duplamente encadeada, sendo suas implementações realizadas dentro da **List.cpp**, uma ressalva sobre a criação da lista, seu esqueleto se encontra na **List.h** e seu uso para com os algoritmos de ordenação na **List.cpp**.

3. Main.cpp

Na **Main.cpp** temos os testes que foram realizados, a lógica dos teste foi que para cada algoritmo, ele ordenaria um vetor de tamanho variados, no caso, entre 1000 e 99000, exemplo : BubbleSort ordenaria um vetor com 1000 elementos, depois com 2000, e assim por diante. Para cálcular a média cada execução seria feita 5 vezes e calculada a média do tempo em microssegundo de cada algoritmo, gerados os dados e guardados em um **.txt** que através da ferramenta **GNU Plot** seriam gerados os gráficos correspondentes a cada algoritmo de ordenação e visualizar seu comportamento.

4. Algoritmos Estudados e suas Filosofias

Nesta seção apresentaremos os algoritmos de ordenação escolhidos e suas respectivas filosofias, cada um tem sua particularidade e forma de ordenar inteiros, podendo em alguns casos um ser melhor ou muito melhor que outro, dependendo bastante de cada situação.

4.1. Bubble Sort

Conhecido como algoritmo de ordenação por bolha ou por flutuação, sua ideia base seria realizar comparações em elementos que estão em posições consecutivas, se o elemento que está na posição mais a frente for menor que o elemento que está na posição anterior, é feita uma troca, se não, não é feito nada, os números que são menores estão flutuando uma posição para trás e os maiores estão flutuando uma posição para frente.

Imagine uma lista de elementos [4,7,2,5,4,0], o algoritmo começa comparando o primeiro elemento, que é o 4, com o próximo elemento, o 7, como o 4 é menor que o 7, não há necessidade de troca. Em seguida, o algoritmo compara o 7 com o próximo elemento, o 2, e identifica que o 7 é maior que o 2, nesse caso, ocorre uma troca entre esses elementos, o processo continua, comparando o 7 com o próximo elemento, o 5, mais uma vez, ocorre uma troca, pois o 7 é maior que o 5 em seguida, o algoritmo compara o 7 com o próximo elemento, que é o 4, e novamente identifica que o 7 é maior que o 4, portanto, ocorre mais uma troca após percorrer a lista uma vez, o maior elemento, o 7, encontra-se na última posição, agora, o algoritmo recomeça o processo a partir do início, ignorando o último elemento, que já está em sua posição correta. Dessa forma, a lista [4, 7, 2, 5, 4, 0] se torna [4, 2, 5, 4, 0, 7]. O algoritmo continua percorrendo a lista, comparando e trocando elementos adjacentes conforme necessário, após a segunda iteração, a lista se torna [4, 2, 4, 0, 5, 7]. Nas iterações subsequentes, os elementos vão gradualmente se movendo para suas posições corretas,no caso do elemento 0, ele precisará passar por várias trocas até chegar à sua posição correta, pois é o menor elemento da lista. Assim, o algoritmo percorre a lista repetidamente até que não haja mais trocas necessárias, resultando na lista completamente ordenada: [0, 2, 4, 4, 5, 7].

Abaixo está uma representação do código implementado da forma convencional :

```
1  // ordena um vetor A[l..r] com indice inicial l
2  // e indice final r, contendo r-l+1 elementos
3  void bubblesort(int A[], int l, int r) {
4      for(int i = l; i < r; i++) {
5          for(int j = r; j > i; j--) {
6              if(A[j] < A[j-1]) {
7                  swap(A[j],A[j-1]);
8              }
9          }
10     }
11 }
```

4.2. Insertion Sort

A ideia dele é parecido quando uma pessoa vai ordenar cartas de baralho em uma mão durante um jogo, imagine que temos uma lista de elementos [4,7,2,5,4,0] a ideia para esse algoritmo seria que começando pelo segundo elemento que seria o 7 e ignorar os elementos que vierem depois, após isso, comparar o 7 com o 4, se o 7 fosse menor que o 4, haveria uma troca, se não, n faz nada. A ideia desse algoritmo, também chamado de algoritmo de ordenação por inserção, é partir de uma lista pequena e ir inserindo números de forma ordenada, de forma que ela cresca e sempre inserindo um número na sua posição ordenada.

Continuando a ordenação do exemplo, haverá uma troca entra o 7 e o 4,em seguida passará para ao próximo elemento que é o 2, para isso é necessário considerar a pequena lista já ordenada que é a [4,7], comparando primeiramente o 2 com o 7, haverá uma troca pois o 7 é maior que o 2, em seguida comparar o 2 com o 4, trocamos novamente a posição, após isso não terá mais elementos para comparar com a pequana lista inicial, a lista nova será o resultado das comparações, sendo agora [2,4,7] de maneira ordenada. Agora para

comparar o 5 com a lista formada [2,4,7], 5 é menor que 7, incrementa o tamanho da lista e o 7 é colocado na frente do 5, para a comparação entre o 5 e o 4, o 5 não será trocado de lugar, visto que ele não é menor que 4, assim 5 chegou na sua posição, tornando a lista ordenada. os mesmos procedimentos são feitos para o resto dos elementos, lembrando que para o 0 será feita várias trocas até chegar na sua posição correta.

Abaixo está uma representação do código implementado da forma convencional :

```
// ordena um vetor A[l..r] com indice inicial l
// e indice final r, contendo r-l+1 elementos
void insertionsort(int A[], int l, int r) {
    for(int j = l+1; j <= r; j++) {
        int key = A[j];
        int i = j-1;
        while(i >= l && A[i] > key) {
            A[i+1] = A[i];
            i--;
        }
        A[i+1] = key;
    }
}
```

4.3. Selection Sort

Conhecido como Ordenação por Seleção, a ideia dele é olhar para uma lista de dados, e procurar sempre o elemento menor ou maior da lista, no caso crescente, o menor de todos, encontrando ele e colocando na primeira posição, procura-se o segundo menor, e assim por diante.

Imagine a seguinte lista de inteiro [7,5,1,8,3] no caso ao percorrer a lista, o menor elemento é o 1, logo, ele é trocado pelo 7, a sua posição, e assim sucessivamente, mas com a particularidade que a lista, após a troca é um pouco menor, sendo o primeiro elemento o 1 que já está na sua posição correta até agora, [1,5,7,8,3] é a lista a ser analisada, mas a análise será feita a partir do 5, visto que o 1 é o menor dos menores , então identificando o segundo menor sendo o 3, ele será trocado com o 5, indo para a segunda posição na lista, [1,3,7,8,5] a nova lista, ao final do processo, as trocas serão feitas de forma que se encontre o menor dos menores, o resultado final é a lista ordenada de maneira crescente [1,3,5,7,8].

Abaixo está uma representação do código implementado da forma convencional :

```
// ordena um vetor A[l..r] com indice inicial l
// e indice final r, contendo r-l+1 elementos
void selectionSort(int A[], int l, int r) {
    for (int i = l; i < r; i++) {
        int indexMin = i;
        for (int j = i + 1; j <= r; j++) {
            if (A[j] < A[indexMin]) {
                indexMin = j;
            }
        }
```

```
11          int aux = A[i];
12          A[i] = A[indexMin];
13          A[indexMin] = aux;
14      }
15  }
```

4.4. Merge Sort

Esse algoritmo é baseado em uma técnica de implementação de algortimo que seria **Dividir para Conquistar**, a ordenação ocorre quando pega-se a lista de dados e separa em listas menores e depois que todos estiverem separados, unir novamente, ordenando eles.

Imagine uma lista de inteiros, [4,7,2,6,4,1,8,3], será feita uma divisão da lista em duas listas, pois o merge prefere resolver problemas menores, após dividir, ficaremos com [4,7,2,6] e [4,1,8,3], mesmo com essa divisão, ainda é difícil para o merge resolver, é feita outra divisão da lista, a lista é quebrada novamente, ficando com [4,7], [2,6], [4,1] e [8,3], mesmo após isso, ainda é difícil para ele resolver, portanto é feita mais uma quebra das listas, [4], [7], [2], [6], [4], [1], [8], [3], percebe-se que a função é chamada por ela mesma até que resulte em um cenário favorável, onde temos várias listas com apenas um elemento e conseguentemente ordenada.

Após as quebras necessárias, é hora de juntar e conquistar a ordenação, juntando o elemento 4 com o 7, resulta em [4,7] que já está ordenada, em seguida juntando 2 e 6 temos a lista [2,6] que também está ordenada, próxima lista seria as com os elementos 4 e 1, mas veja que é necessário trocar os elementos de lugar resultanto em [1,4] e por fim os elemento 8 e 3 também vão trocar de lugar ficando com a lista [3,8].

Agora é necessário montar sublistas de tamanho 4, a princípio devemos comparar a lista com [4,7] e a lista [2,6], para resolver esse problema de juntar as duas, pensemos como duas pilhas de cartas de baralho, e vai olhando para as duas pilhas, identificar o menor elemento no topo de cada pilha, o caso de exemplo temos uma pilha com 4 e 7 e outra com 2 e 6, checando seus valores, vemos que no começo da lista que seria o topo da pilha, o menor valor dentre as duas pilhas é o 2, logo ele é colocando como primeiro elemento da nova lista, em seguida o segundo menor que seria o 4, ele é colocado após o 2, e assim sucessivamente, resultado em um lista com 4 elementos [2,4,6,7].O mesmo procedimento é feito para juntar as duas listas [1,4] e [3,8].

No passo final, para ordenar a lista, pegamos as duas listas resultantes das comparações anteriores, que seriam [2,4,6,7] e [1,3,4,8], executando a mesma ideia da pilha de cartas, verificando que está no topo nas duas pilhas e incrementando em ordem crescente na lista final ordenada : [1,2,3,4,4,6,7,8].

Abaixo está uma representação do código implementado da forma convencional :

```
1  // ordena um vetor A de tamanho n = r-p+1 com limites:
2  // o vetor comeca na posicao A[p]
3  // o vetor termina na posicao A[r]]
4  // Divide-se o vetor em dois subvetores de tamanho n/2
5  void merge(int A[], int p, int q, int r) {
6      int n1 = q - p + 1; // tamanho do subvetor esquerdo
7      int n2 = r - q;     // tamanho do subvetor direito
```

```
8
9       int L[n1], R[n2]; // vetores auxiliares
10
11      // Copia os elementos para os vetores auxiliares
12      for (int i = 0; i < n1; i++) {
13          L[i] = A[p + i];
14      }
15      for (int j = 0; j < n2; j++) {
16          R[j] = A[q + 1 + j];
17      }
18
19      // Combina os elementos em ordem crescente
20      int i = 0; // indice do subvetor esquerdo
21      int j = 0; // indice do subvetor direito
22      int k = p; // indice do vetor original
23
24      while (i < n1 && j < n2) {
25          if (L[i] <= R[j]) {
26              A[k] = L[i];
27              i++;
28          } else {
29              A[k] = R[j];
30              j++;
31          }
32          k++;
33      }
34
35      // Copia os elementos restantes do subvetor esquerdo, se
            houver
36      while (i < n1) {
37          A[k] = L[i];
38          i++;
39          k++;
40      }
41
42      // Copia os elementos restantes do subvetor direito, se
            houver
43      while (j < n2) {
44          A[k] = R[j];
45          j++;
46          k++;
47      }
48  }
49
50  void mergesort(int A[], int p, int r){
51      if(p<r){
52          int q = (p+q)/2; // Dividir
53          // Conquistar
54          mergesort(A,p,q);
```

```
55          mergesort(A,q+1,r);
56          // Combinar
57          merge(A,p,q,r);
58      }
59  }
```

4.5. Quick Sort

A ideia desse algoritmo seria que, se pegarmos uma lista já ordenada e se escolhermos qualquer número, obrigatoriamente a direita desse número estarão aqueles que são menores que ele e a esquerda os maiores, nesse caso é escolhido um pivô, onde a partir dele será realizada essa propriedade, isso feito inúmeras vezes até que ao pegarmos qualquer número a propriedade será preservada.

Imagine uma lista de inteiro [4,7,2,6,4,1,8,3], o primeiro passo é escolher um pivô, onde teremos que colocar números menores que ele na sua esquerda e o maior na sua direita, no exemplo, escolhendo o 3 como pivô, a esquerda teremos os elementos menores que 3 ou iguais, e a direita o elementos maiores que 3. A ideia começa quando definimos as limitações necessárias, é mais eficiente pensar que no começo temos duas divisórias, uma delas marca o espaço onde estão os elementos menores que 3 e a outra parte marca o espaço onde estão os elementos maiores que 3, dessa forma é mais eficiente de se pensar pois assim o 3 será movido apenas no final das divisórias, ao invés de ficar trocando sua posição várias vezes se for necessário, fazendo a troca no final das divisórias completas, e no final coloca-se o 3 no meio de todos.

As divisórias que delimitam as propriedades do Quick Sort chamarei de Maiores o marcador aqueles que delimitam os elementos maiores que 3 e Menores aqueles os elementos menores que 3, no começo da lista temos o número 4, ainda não mexemos as divisórias, 4 é maior que o pivô 3, logo a divisória Maiores incrementa uma posição, dizendo que na sua esquerda estão elementos maiores que 3, lembrando que a divisória dos menores continua intacta, passando para o elemento 7, 7 é maior que o pivô, a divisória Maiores incrementa uma posição e a divisória Menores continua inalterada, passando para o elemento 2, 2 é menor que o pivô, logo é feito um processedimento para casos como esse, a divisória do Maiores é incrementada uma posição, inglobando 4,7 e 2, em seguida é incrementada a divisória do Menores em uma posição, após isso é feita uma troca entre o 2 e o 4, no final desse procedimento, do lado esquerdo da divisória dos Menores estará os elementos menores que o pivô e a esquerda da divisória do Maiores estará os maiores mas agora com uma delimitação visível, englobando apenas 7 e 4 ficando parecido com essa representação simples [2 **Menores** 7,4 **Maiores** 6,4,1,8,3].

No final de vários procedimentos parecidos com o anterior a lista ficará assim : [2, 1 **Menores** 4,6,4,7,8 **Maiores**, 3] na esquerda da divisória dos menores ficará os elementos menores que o pivô e do mesmo jeito, a esquerda da divisória Maiores ficará os elementos maiores que 3, que tem um delimitação até a divisória do Menores, agora resta colocar o pivô na sua posição, incrementa a divisória dos Maiores em uma posição, o mesmo para a divisória do Menores e troca-se o elemento pivô pelo 4, logo temos : [2,1,3 **Menores** 6,4,7,8,4 **Maiores**], na esquerda da divisórias dos Menores teremos os elementos menores ou iguais ao pivô e na esquerda da divisória Maiores até a divisória Menores, os elementos maiores que o pivô.

Agora, como resultado das comparações é necessário chamar a função novamente, recursivamente, agora com duas sublistas que são : [2,1] e [6,4,7,8,4] sem mover o 3 de posição pois ele já está na sua posição correta, a medida que as chamadas recursivas são executadas a lista vai sendo ordenada.

Abaixo está uma representação do código implementado da forma convencional :

```
// Recebe um vetor A[l...r] com l <= r.
// Rearranja os elementos do vetor e devolve
// j com l...r tal que A[l..j-1] <= A[j] < A[j+1...r]
int partition(int A[], int l, int j){
    int pivo = A[r];
    int j = 1;
    for(int k = l; k < r; k++){
        if(A[K] <= pivo){
            int aux = A[k];
            A[k] = A[j];
            A[j] = aux;
            j++
        }
    }
    A[r] = A[j];
    A[j] = pivo;
    return j;
}
// Rerranja o vetor A[p...r],
// com p <= r+1, de modo que ele fique em
// ordem crescente.
 void quicksort(int A[], int l, int r){
    if(l<r){
        int j = partition(A,l,r);
        quicksort(A,l,j-1);
        quicksort(a,j+1,r);
    }
 }
```

4.6. Shaker Sort

Também conhecido como Cocktail Sort e Bubble Sort bidirecional, é uma variação do Bubble Sort que é tanto um algoritmo de ordenação estável quanto uma ordenação por comparação. A sua ideia se baseia em comparar um valor com outro ao lado, ordenando os maiores valores para as últimas posições, e retorna trazendo os menores valores.

Imagine a lista de números [5,1,3,4,6,2], com base na ideia do algoritmo, o 5 é comparado com o valor ao lado, no caso o 1, assim trocando de posição com o 1, ficando com [1,5,3,4,6,2], em seguida é feita outra comparação com o 3, visto que o 5 é maior que o 3, eles trocam de posição, [1,3,5,4,6,2], depois é feita outra troca, pois o 5 é maior que o valor ao seu lado, [1,3,4,5,6,2], nesse momento o valor 5 não é maior que 6, assim a comparação irá para o número seguinte, no caso 6, é feita a comparação com o próximo número, é feita uma troca, [1,3,4,5,2,6], como o valor 6 chegou ao final, a comparação é

feita de forma contrária, portanto o pivô de comparação será o 2, ele trocará de posição com os elementos na sua esquerda, até que o elemento ao lado seja menor que ele, por fim, levando em consideração que a comparação ao contrário só é feita quando o maior elemento está no final, temos a lista por fim ordenada : [1,2,3,4,5,6].

Abaixo está uma representação do código implementado da forma convencional :

```
void ShakerSort(int arr[], int n){
    // verificador de trocas
    bool swapped = true;
    // indicar o comeco do array
    int start = 0;
    // indicar o final do array
    int end = n - 1;

    while (swapped) {
        swapped = false;

        // Percorre o array da esquerda para a direita
        for (int i = start; i < end; ++i) {
            if (arr[i] > arr[i + 1]) {
                swap(arr[i], arr[i + 1]);
                swapped = true;
            }
        }

        if (!swapped)
            break;

        swapped = false;
        --end;
        // Percorre o array da direita para a esquerda
        for (int i = end - 1; i >= start; --i) {
            if (arr[i] > arr[i + 1]) {
                swap(arr[i], arr[i + 1]);
                swapped = true;
            }
        }
        ++start;
    }
}
```

5. Estudo isolado e em conjunto

Nesta seção temos um estudo isolado de cada algoritmo, utilizando teste com listas duplamente encadeadas de inteiros, escolhemos para as listas uma quantidade de inteiros absurdamente grandes no intuito de visualizar o comportamente de cada algortimo, demostrado nos gráficos em cada subseção tem o gráfico correspondente e um comentário acerca dos resultados baseados na média de tempo em microssegundos e quantidade de inteiros envolvidos em cada nível do gráfico.Lembrando que todos foram executados com os

mesmos vetores, com o objetivo de ter uma melhor visão dos acontecimentos, os gráficos estão no fim do documento, pois estavam ficando mal posicionados no latex.

5.1. Bubble Sort

No gráfico sobre o algoritmo Bubble Sort é possível notar que ele cresce de forma estável, mesmo com algumas variações no tempo, mas percebe-se que quando o tamanho do vetor passa de 50000, as variações se tornam mais bruscas e sua ocorrências aumentam a medida que o tamanho do vetor aumenta de 10 mil em 10 mil.Entretanto entre 90 mil e 99 mil é a região onde o algoritmo tem mais dificuldade de ordenar devido ao tamanho do vetor.

5.2. Insertion Sort

No gráfido sobre o algoritmo Insertion ele é mais estável com tamanho pequenos se mantém estável até 50 mil, após esse tamanho de vetor as variações no tempo de execução se tornam mais frequentes, entretanto após 70 mil, ele tem um aumento no tempo de execução demorando mais para ordenar os vetores.

5.3. Selection Sort

No gráfico sobre o algoritmo Selection Sort ele é estável até 50 mil mas nesse tamanho é possível notar uma bruca variação no tempo de execução e em seguida, até 70 mil as variações se tornam constantes, e após 70 mil de tamanho ele tem mais dificuldade de ordenar e no 90 mil ele tem uma variação mais brusca que nas anteriores.

5.4. Merge Sort

Nesse gráfico sobre o Merge Sort ele começa estável com poucos pontos de variação mas curiosos, percebe-se que eles em entre intervalor de 20 mil, em 30 mil ele tem um variação de tempo de execução, no 50 mil ele tem outro ponto de variação, o mesmo fenômeno se repete no 70 mil e por fim entre 90 mil e 99 mil ele tem uma variação maior no tempo de execução.

5.5. Quick Sort

Sem dúvidas o Quick Sort é melhor de todos os algoritmos, ele é bastante estável e seu comportamento é mais parecido com uma reta crescente $O(n^2)$ e com poucos pontos de variação no tempo de execução se tornando, de novo, o algoritmo que se saiu como melhor nos testes realizados.

5.6. Shaker Sort

O algoritmo escolhido tem o comportamento mais estável e com poucas variações no tempo de execução entretanto demora mais para executar em comparação com outros algoritmos.

5.7. Todos os Algoritmos

É realmente interessante ver a diferença entre todos eles, o mesmo objetivo, mas com formas, maneiras e tempos diferentes para cada um. O que mais nos impressionou foi o Quick Sort, que a linha dele mal aparece no grafico de tão rapido que ele é, outro que impressionou foi o Shaker Sort, mas de maneira negativa, inicialmente pensávamos que seria

mais rápido, não sabemos se foi por causa dos vetores testados ou uma implementação ineficiente da nossa parte, ele ficou realmente lento, muito mais até que o Bubble. Em relação ao Merge Sort, ele também foi um agrado, pois é até que bem eficiente, podemos ver também a instabilidade de alguns algoritmos, como o Bubble e do Selection. Ademais, podemos ver a diferença de velocidade absurda que existe apenas na forma de ordenar um determinado vetor, onde podemos aprender que uma pequena mudança pode fazer grande diferença na otimização do algoritmo.

6. Perguntas

1. Uma descrição breve das principais características de cada um dos 6 algoritmos de ordenação que foram programados, como por exemplo:
   (a) Qual a complexidade de pior caso ?
      i. **Bubble Sort** tem complexidade $O(n^2)$ no pior caso.
      ii. **Insertion Sort** tem complexidade $O(n^2)$ no pior caso.
      iii. **Selection Sort** tem complexidade $O(n^2)$ no pior caso.
      iv. **MergeSort** tem complexidade $O(n \times \log n)$ no pior caso.
      v. **QuickSort** tem complexidade $O(n^2)$ no pior caso.
      vi. **Shaker sort** tem complexidade $O(n^2)$ no pior caso.

   (b) são estáveis ?
   O BubbleSort,MergeSort e InsertionSort são estáveis pois eles preservem a ordem relativa dos elementos com chaves iguais, enquanto, SelectionSort, QuickSort nãe é garantido pois nas trocas dos elementos quando feitas a ordenação, elas podem alterar a ordem relativa de elementos com chaves iguais e o Shaker Sort é estável.

   (c) são algoritmos **in loco** ?
   BubbleSort,InsertionSort e SelectionSort são **in loco** devido a ordenação ser feita diretamente na lista de entrada, sem interferir no tamanho da lista, enquanto MergeSort e QuickSort não são **in loco** pois eles mudam o tamanho da lista original para auxiliar na ordenação do números e o Shaker Sort é in loco

   (d) são recursivos ou iterativos ?
   Iterativos são : BubbleSort, MergeSort, SelectionSort, Shaker Sort
   Recursivos : QuickSort e InsertionSort.

   (e) Qual a ideia principal do algoritmo? Como ele faz para ordenar o vetor?
   A respostas para essa pergunta se encontra na seção 4 sobre as filosofias, clique no número 4 e será levado para a seção correta, espero que ajude a compreender sua filosofia.
2. Gráficos + Comparações;
   A resposta está na seção 5 Estudo isolado, clique no número 5 e será levado para a seção com a visão sobre os algoritmos, e logo mais a baixo as imagens com os gráficos.

7. Divisão de trabalho

Os algoritmos foram divididos, onde o Elysson ficou com o Quick, Merge e Insertion, o Gustavo com o Selection, Shaker e o Bubble. A ideia do List.h foi desenvolvida em conjuto, pois foi necessario entender as funções que seriam usadas, a criação do Node

e sua ultilização nas funções definidas. A Main foi feita em conjunto também, onde o Elysson fez as funções relacionadas com a leitura de dados e o Gustavo com a escrita, inicialmente o Elysson fez as duas, mas por mudanças posteriores, o Gustavo recriou ela. A função de gerar dados foi feita pelo Elysson e a de calcular tempo foi feita pelo Gustavo.

8. Dificuldades

As duas principais dificuldades foram a implementação dos algoritmos e o tempo de execução. Com relação a implementação dos algoritmos, foi difícil decider qual o sexto algoritmo para o projeto, inicialmente pensamos no radix sort, mas ele se mostrou muito ineficiente, pesquisamos como otimizar, pois, quando vimos ele em um vídeo do Youtube, ele parecia extremamente rapido, mas na nossa implementação ele não se provou-se eficiente, com isso fomos para o Shaker sort, ainda dentro da implementação, fizemos cada algoritmo por vez, para não ter o perigo de acharmos que estão corretos, e depois, mais na frente se mostrarem errados e assim gerando muitos problemas para serem corrigidos, alem disso tudo, ainda tivemos que testar um por um para ver se realmente estavam ordenando corretamente. Agora, com relação ao tempo de execução, inicialmente ficamos pensando que a lentidão se daria por conta dos algoritmos em si, após pesquisarmos e conversar com colegas, supomos que a lentidão estava na Main. Inicialmente estavamos usando a função gerar dados de formar ineficiente, alem disso, estavamos fazendo cada algoritmo por vez, ou seja, primeiro rodava de 1000 até 99000 de um algoritmos, e depois ia para o próximo, inicialmente pensamos que não teria problema nenhum, mas escrevemos de tal forma que chamavam varias funções para cada execução, após percebermos essa ineficiencia, procuramos formas de reduzir a quantidade de funções executadas, achando uma forma de reduzir a quantidade de pushback, conseguimos melhorar muito a eficiência, quando tentamos rodar os algoritmos pela primeira vez, iniciamos as 22 horas, deixas o notebook ligado a madrugada e perto das 9 do outro dia ainda estava no selection e o tempo estava na cada do 10 elevado a 8, desistimos de tentar rodar buscamos uma maneira mais eficiênte, chegamos na atual, onde não sabemos o tempo exato, mas achamos que se apróxima das 3 horas variando do computador ou notebook usado, e dizemos isso porque testamos o código no notebook e num pc, e no pc demorou quase o dobro do tempo do notebook.

9. Gráficos dos estudos isolados e em conjunto

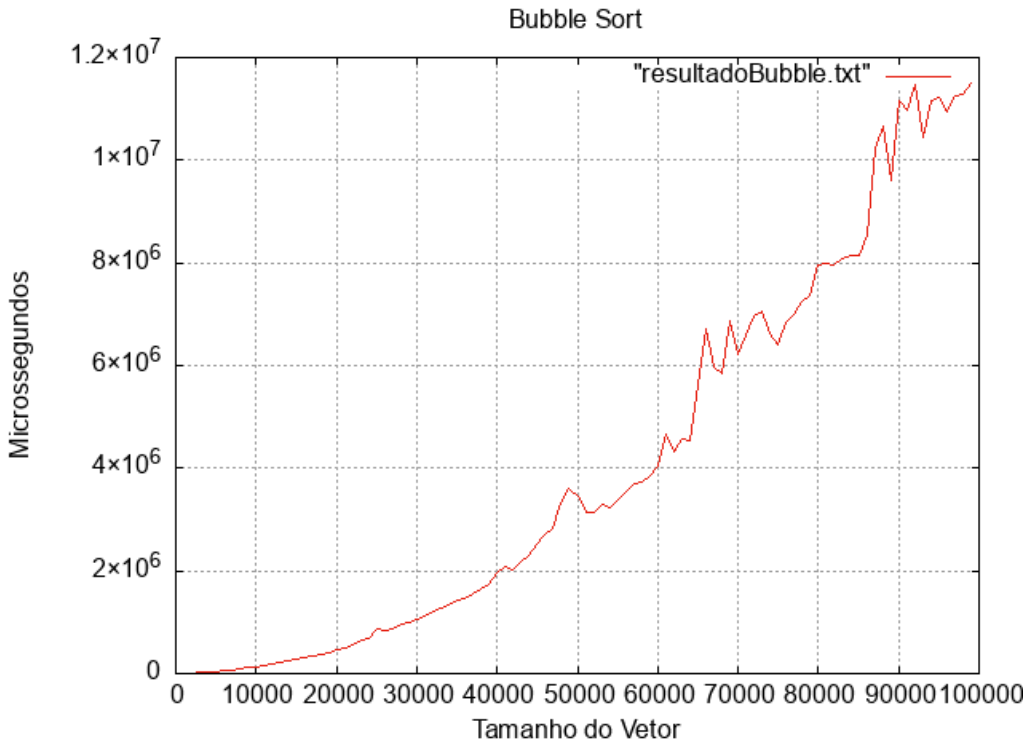

Figure 1. Gráfico do Bubble sort

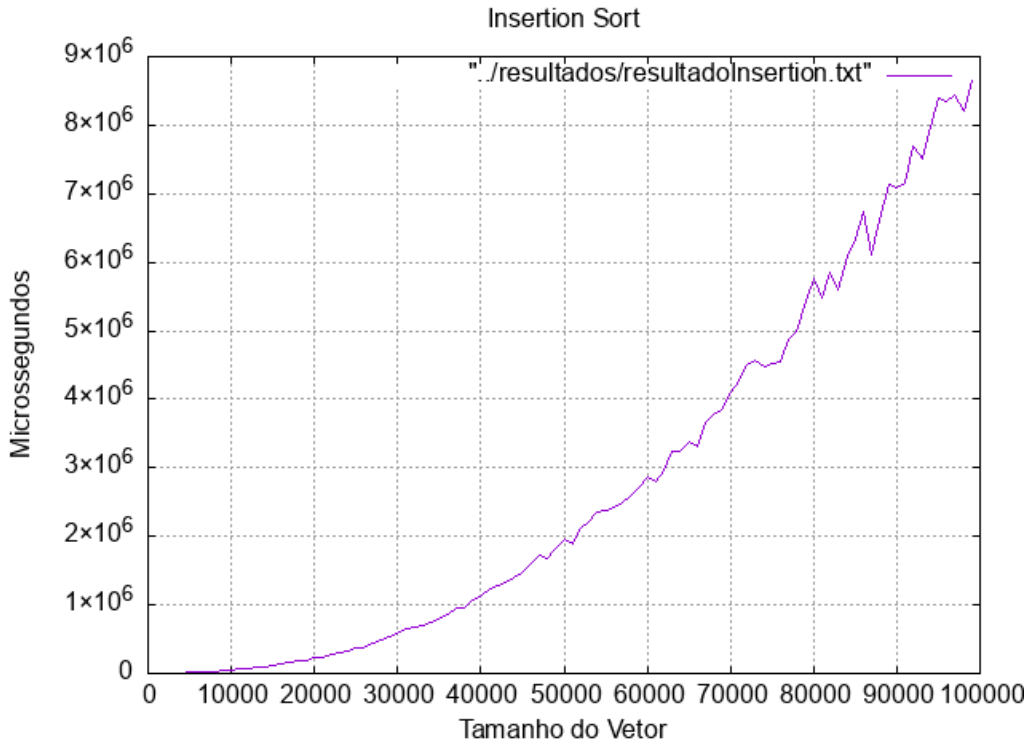

Figure 2. Gráfico do Insertion Sort

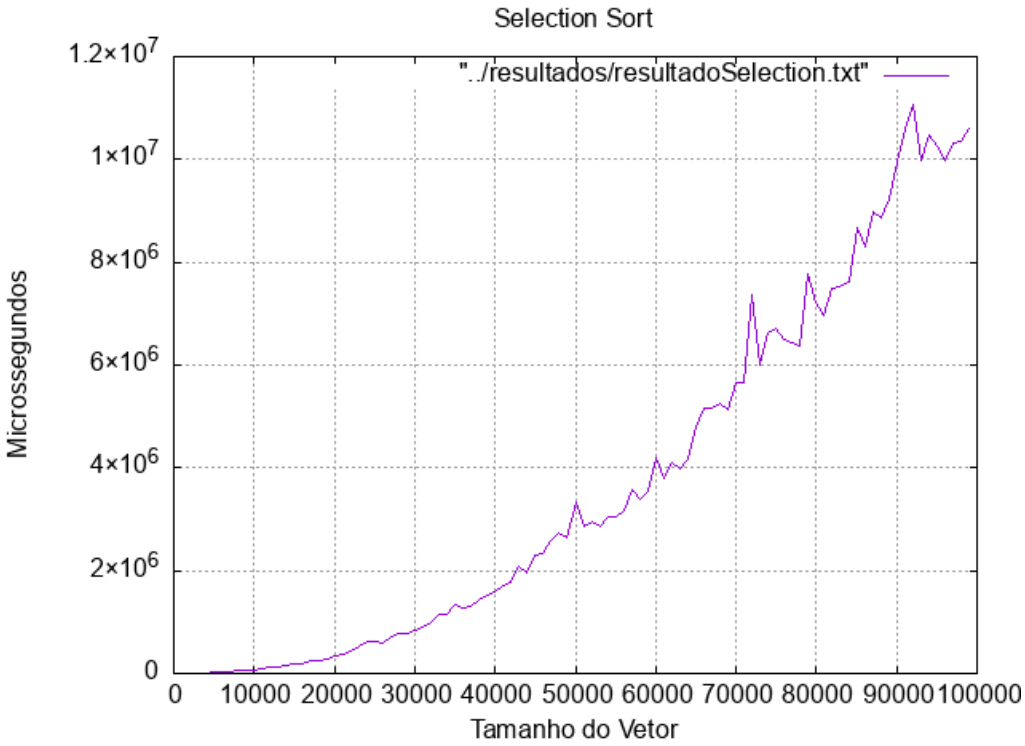

Figure 3. Gráfico do Selection Sort

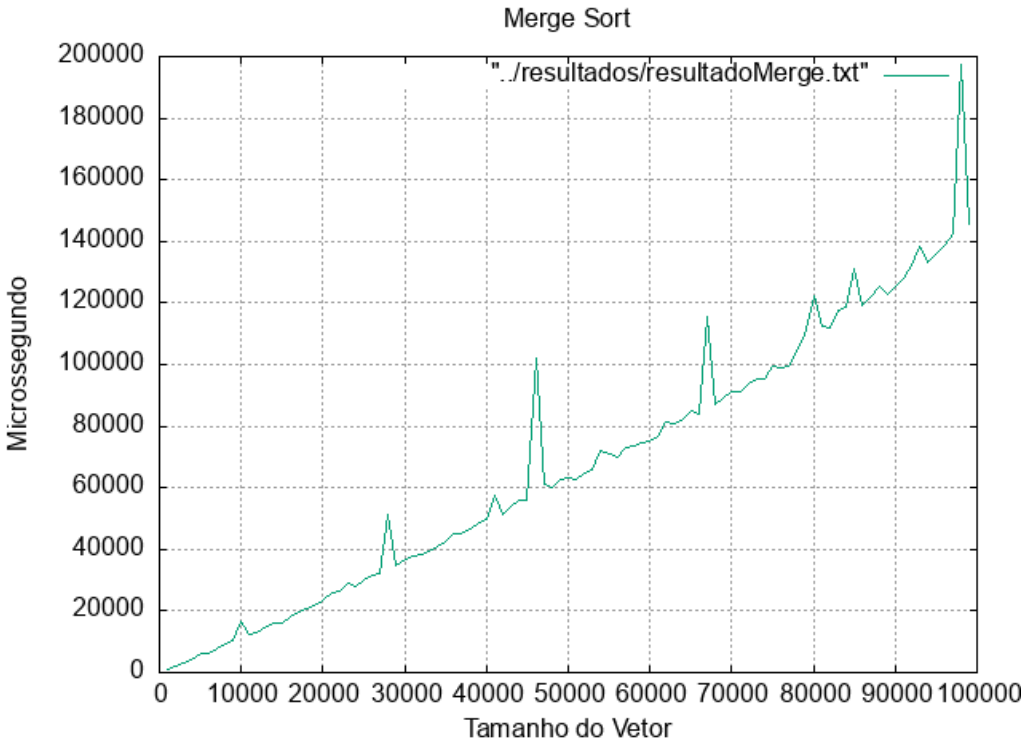

Figure 4. Gráfico do Merge Sort

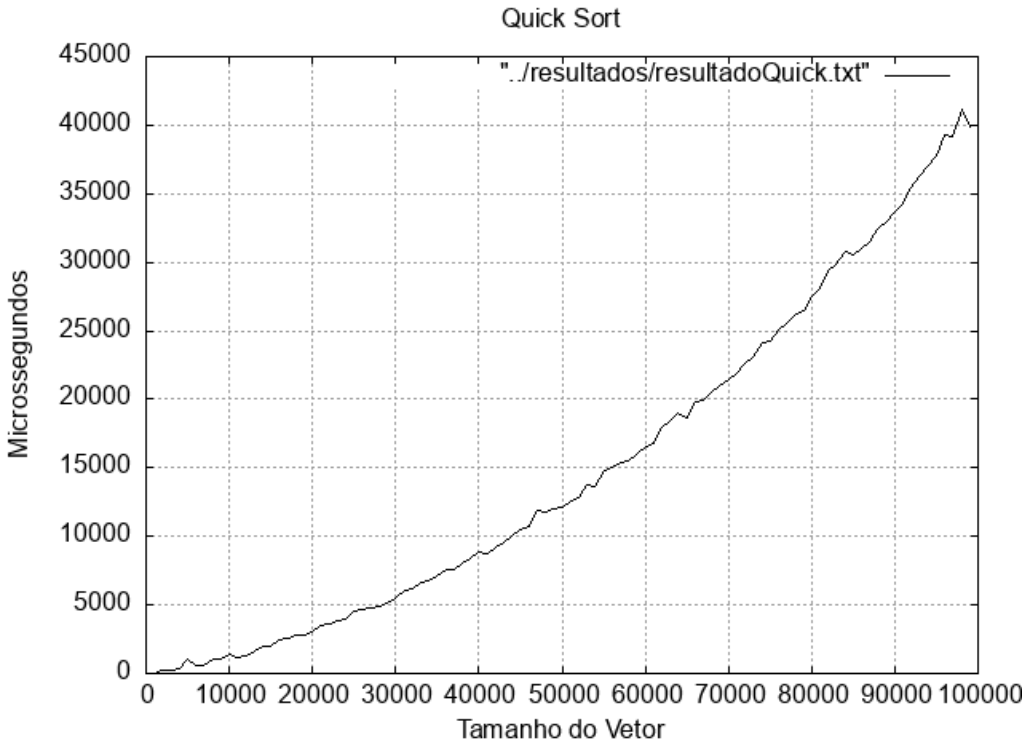

Figure 5. Gráfico do Quick Sort

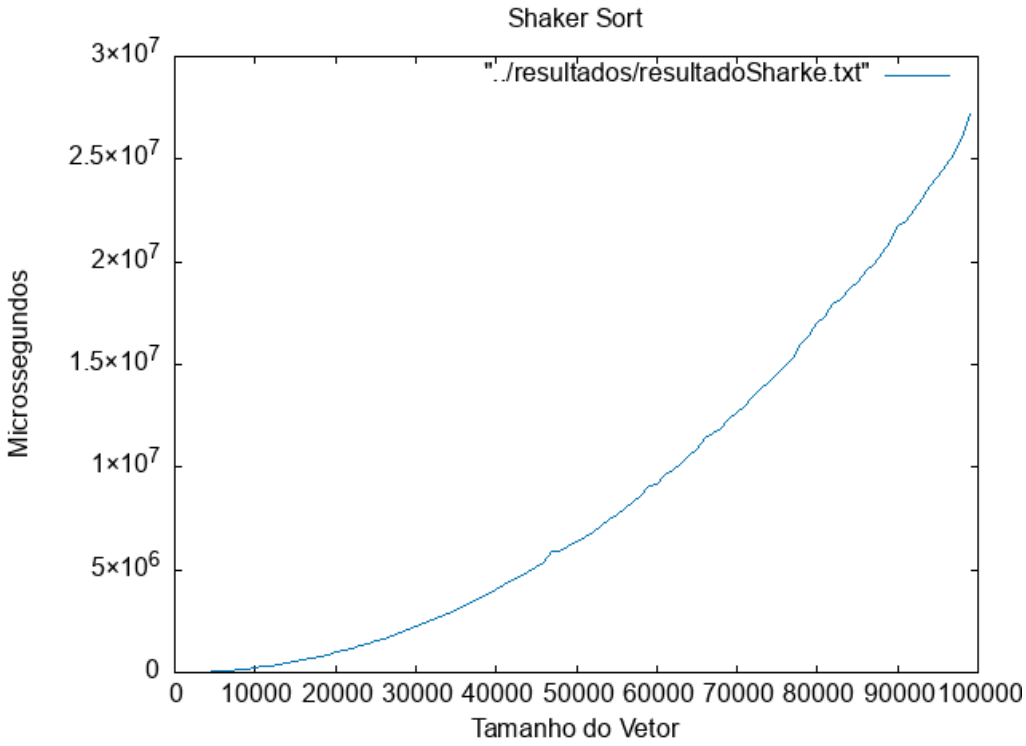

Figure 6. Gráfico do Shaker Sort

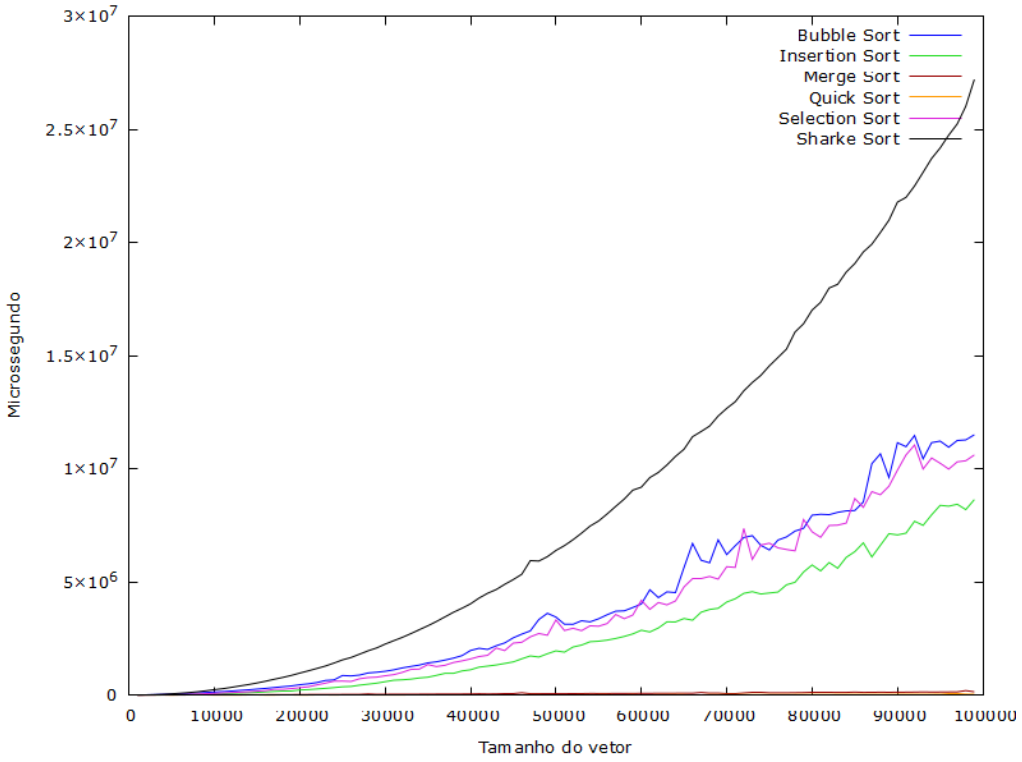

Figure 7. Gráfico de todos os algoritmos

10. Referências

Para a produção desse projeto utilizamos o livro [Calle 2017] e para complementar esse trabalho também utilizamos a playlist do canal Programação Dinâmica para entender melhor as filosofias dos algoritmos. Utilizamos de varios vídeos do Youtube falando sobre os algoritmos e como crialos, não colocaremos todos, pois, em alguns apenas vimos conceitos especificos em pequenos trechos, mas os vídeos onde realmente deram uma base boa iremos colocar na região abaixo, clique nos nomes para ser direcionado para os vídeos.
Vídeo sobre o radix: Radix sort + Pokémon
Comparando as velocidades dos algoritmos: 15 Sorting Algorithms in 6 Minutes
Gráficos no laboratório didático de Física: utilizando o gnuplot
Plotando gráficos com o Gnuplot.
Curso de C++ 50 - Operações com arquivos (ofstream) - Parte 1
Aula 23 - Arquivos Texto — Leitura e Escrita de Caracteres Palavras e Linhas — Curso de C++

References

[Calle 2017] Calle, P. D. (2017). ***Introdução à Programação Orientada para Objetos em Linguagem C++***. Universidade de São Paulo - USP, 1th edition.