UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
INSTITUTO METRÓPOLE DIGITAL

RELATÓRIO DE ANÁLISE EMPÍRICA DE ALGORITMOS DE ORDENAÇÃO

NATAL/RN
2020

UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
INSTITUTO METRÓPOLE DIGITAL

DIEGO FILGUEIRAS BALDERRAMA
JOSE VICTOR FERREIRA DA FONSECA
PAULO VANZOLINI MOURA DA SILVA

RELATÓRIO DE ANÁLISE EMPÍRICA DE ALGORITMOS DE ORDENAÇÃO

Trabalho referente à nota parcial da Unidade I da disciplina DIM0119 - Estrutura de Dados Básica I - T02, orientado pelo Prof. Mr. Guilherme Fernandes de Araújo.

NATAL/RN
2020

SUMÁRIO

INTRODUÇÃO	**4**
DESENVOLVIMENTO	**7**
RESULTADOS E CONSIDERAÇÕES FINAIS	**10**

1. INTRODUÇÃO

Ordenar um conjunto de elementos significa colocar eles em uma determinada ordem, que pode ser crescente ou decrescente. Quando se fala em ordem crescente, significa dizer que os elementos estão organizados de modo que um elemento, exceto o primeiro do conjunto, tem alguma característica que o classifica como maior ou superior aos anteriores a ele. Ou seja, cada elemento é maior que todos seus anteriores.

A ordenação de forma decrescente é exatamente o contrário. Todos os elementos selecionados ao acaso, salvo o primeiro, tem sempre a característica de ser classificado como menor ou inferior aos seus anteriores.

Na programação, por muitas vezes, é mais vantajoso trabalhar com elementos ordenados, seja para economia de tempo ou maior facilidade durante o desenvolvimento do código. Um exemplo disso é a busca binária, que só pode ser aplicada em elementos ordenados e tem sua complexidade muito inferior se comparada a outros algoritmos de busca.

São muitos os algoritmos de ordenação: Selection Sort, Insertion Sort, Bubble Sort, Merge Sort, Shell Sort, Quick Sort, dentre outros. No presente relatório, faremos a análise empírica do Bubble Sort e Merge Sort.

1.1 - Bubble Sort:

O Bubble Sort, também conhecido como ordenação bolha, percorre todo o conjunto de elementos analisando os adjacentes par a par. Então, analisa o primeiro e segundo elementos, depois o segundo e terceiro e assim sucessivamente, até analisar o penúltimo e último elementos.

Essa análise é feita através de uma comparação para saber se o par está ordenado, de forma crescente ou decrescente, de acordo com nosso interesse.

Caso o par esteja ordenado, segue para o próximo passo. Senão, realiza uma troca de posição entre o par analisado.

Essa operação é repetida várias vezes até que nenhuma troca possa ser feita em todo o conjunto.

Para exemplificar, será usado um conjunto pequeno de dados inteiros desordenados e através do método bubble sort ele será totalmente ordenado de forma crescente.

Observe:

5 3 2

3 **5 2**

3 2 5

3 2 5

2 **3 5**

2 3 5

Perceba que ao final de cada iteração o maior elemento irá para o final do conjunto. Então é garantido que se o maior elemento não está na última posição, após a primeira iteração ele será 'jogado' para lá, pois a análise está sendo feita par a par e ele será sempre maior do que qualquer outro elemento do conjunto.

A partir disso, observamos que não precisa mais percorrer todo o conjunto a cada iteração, pois o último elemento estará sempre na sua posição correta a cada 'rodada' do laço.

Para finalizar, devemos considerar também que esse método modifica o conjunto original. Então caso o usuário deseje obter um conjunto ordenado sem perder o conjunto disposto da forma inicial, deve usar outro método.

1.2 - Merge Sort:

O Merge Sort, também conhecido como ordenação por mistura, é um método que segue a ideia de dividir para conquistar. Basicamente, divide de forma recursiva um conjunto de elementos de forma que se obtenha vários subconjuntos com apenas um elemento cada.

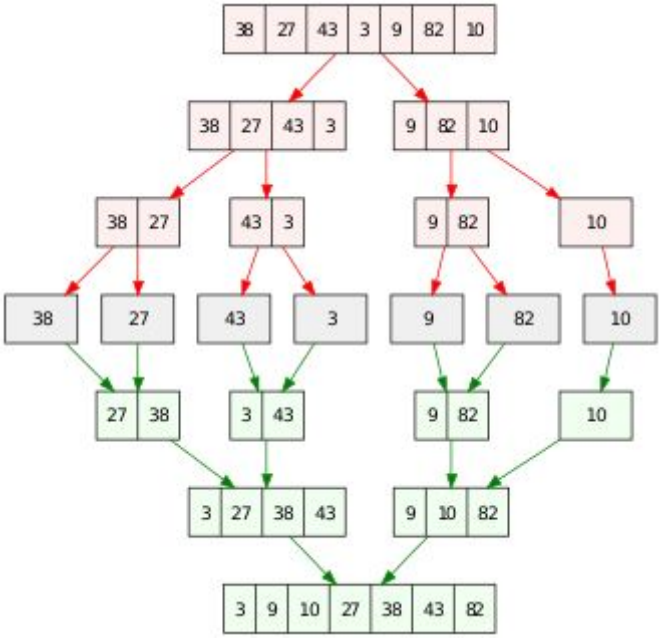
Após fazer todas as divisões, combina dois subconjuntos de forma a obter um conjunto maior e ordenado — *Para fins de exemplificação trataremos os exemplos usados como ordenação de forma crescente* —, repetindo esse processo até se ter apenas 1 conjunto.

Quando é obtido apenas um elemento, então é fácil dizer que ele sempre será o menor ou maior do conjunto, pois só existe ele. Logo após, combina de dois em dois subconjuntos e são formados vários conjuntos de 1 ou 2 elementos,

comparando entre esses dois qual é o menor para colocar na primeira posição e o maior será o seguinte. Após esse passo é como se existissem várias pilhas ordenadas com um ou dois elementos cada, de modo que sempre serão comparados os elementos que estão em cima da pilha. O elemento que for menor será retirado e colocado na primeira posição, depois compara novamente e segue o mesmo passo, até se obter duas pilhas com quatro elementos cada. Esse processo irá ser repetido até chegar novamente a um conjunto de tamanho N novamente, sendo N o tamanho do conjunto original.

Para exemplificar, usaremos um pequeno conjunto de dados inteiros não ordenados e faremos a ordenação pelo merge sort para obter um arranjo crescente de dados.

Observe:



Fonte: Wikipedia - Merge Algorithm

Esse método possui a vantagem de ter uma boa performance, porque tanto no melhor, quanto no pior caso ele tem uma complexidade O(N log N). Além disso, é um algoritmo estável, pois não faz trocas desnecessárias alterando a ordem de dados iguais.

Entretanto, possui a desvantagem de ter um gasto maior por ser recursivo e utiliza mais memória devido o uso de vetor auxiliar durante a ordenação.

2. DESENVOLVIMENTO

Como já foi dito, realizamos a análise empírica do Bubble Sort e do Merge Sort neste trabalho. Os algoritmos foram implementados em C++, utilizando o compilador g++ (9.3.0) e a IDE Microsoft Visual Studio Code (1.50.1). Além disso, foram utilizadas as ferramentas Git e GitHub para versionamento de código e o Gnuplot versão 5.2 para plotar os gráficos a partir dos dados obtidos. O repositório no GitHub pode ser acessado para consulta por meio do link a seguir: https://github.com/paulovanzo/analise_empirica_sort.

As amostras dos testes foram geradas automaticamente de forma randômica por um programa à parte, que gerou um arquivo com os dados a serem lidos pelo programa principal posteriormente.

```
5   void generate(){
6
7     std::ofstream arquivo;
8
9     arquivo.open(("../data/vectors.txt"));
10
11      srand(time(NULL));
12
13      for (int tam = 10; tam < 500; tam += 10){
14        for (int k = 0; k < 50; ++k){
15          for (int i = 0; i < tam; ++i){
16            arquivo << 1+rand()%tam << " ";
17          }
18          arquivo << "\n";
19        }
20      }
21
22    arquivo.close();
23
24  }
```

Função para gerar amostras.

Para os dois algoritmos, o tamanho da amostra variou de 10 a 500, aumentando de 10 em 10, e para cada tamanho foram feitos 50 testes com diferentes amostras geradas de forma aleatória pela função exibida anteriormente, nos quais foram aplicados uma média aritmética para eliminar os picos de processamento paralelo e gerar resultados mais consistentes e homogêneos.

A implementação do Bubble Sort utilizada pode ser vista a seguir:

```
1  void bubble (vector <int> &list){
2
3      // Evaluates the length of the list
4      size_t tam = list.size();
5
6      // Runs across the array
7      for (size_t i = 0; i < tam-1; i++){
8          for (size_t j = 0; j < tam-1; j++){
9
10             // If the next element is greater then
11             // the current element they switch positions
12             if (list.at(j) > list.at(j+1)){
13                 int aux = list.at(j);
14                 list.at(j) = list.at(j+1);
15                 list.at(j+1) = aux;
16             }
17         }
18     }
19 }
```

Função do Bubble Sort.

Já o Merge Sort foi implementado usando duas funções. A primeira função divide recursivamente a lista recebida e passa para a segunda, que é onde de fato de fato ocorre a ordenação dos elementos. As duas funções implementadas podem ser vistas a seguir:

```
1  void merge_sort(vector <int> &list, int begin, int end){
2
3      //  Checks if there's still elements to be sorted
4      if (begin < end){
5
6          //  Sets the middle element index
7          size_t mid = (size_t)begin + (end - begin) / 2;
8
9          // Sorts the left side of the list
10         merge_sort(list, begin, mid);
11
12         // Sorts the right side of the list
13         merge_sort(list, mid+1, end);
14
15         // Sorts both sides together
16         merge(list, (size_t)begin, mid, (size_t)end);
17
18     }
19 }
```

Primeira função do Merge Sort.

```cpp
void merge(vector <int> &list, size_t begin, size_t mid, size_t end){

    // Copies the orginal list into the auxiliar list
    vector <int> aux = list;

    //  Indexes that will be used to access
    // elements in both original and auxiliar lists
    size_t i = begin, j = (mid + 1), k = begin;

    // Sorts the original list according to the auxiliar one
    while(i <= mid && j <= end){
        if(aux.at(i) <= aux.at(j)){
            list.at(k) = aux.at(i);
            i++;
        }else{
            list.at(k) = aux.at(j);
            j++;
        }
        k++;
    }

    //  Copies the rest of the left side of
    // the auxiliar list into the original one
    while(i <= mid){
        list.at(k) = aux.at(i);
        k++;
        i++;
    }
}
```

Segunda função do Merge Sort.

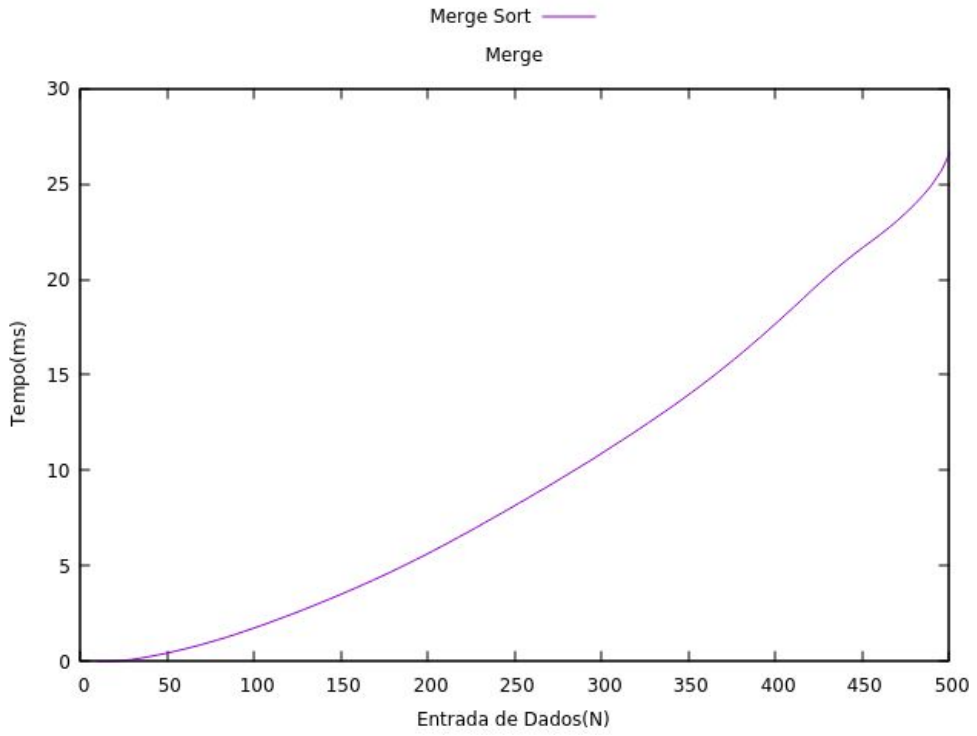
Após a execução dos testes, o tempo médio para cada tamanho das amostras foi salvo em um arquivo externo para que, em seguida, fosse possível a geração dos gráficos por meio do Gnuplot.

3. RESULTADOS E CONSIDERAÇÕES FINAIS

Para rodar o programa e gerar os gráficos foi utilizado um computador com as seguintes especificações e ferramentas:

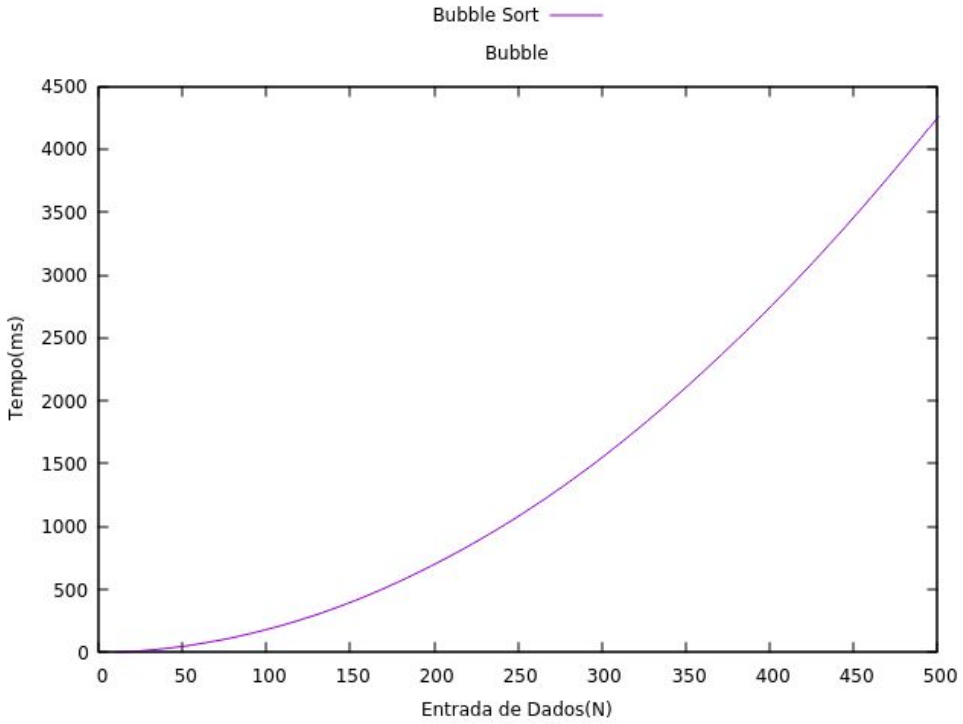
- Sistema Operacional: Ubuntu 20.04.
- Processador: AMD Ryzen 5 2600 6C/12T 3.4 GHZ.
- Memória RAM: 8GB 2400Mhz.
- Arquitetura: x86_64.
- Linguagem: C++11.
- Compilador: G++ 9.3.0.
- Gnuplot: 5.2.8

A curva de crescimento do Merge Sort teve variação um pouco maior ou igual a 1ms/N, a média com o vetor de tamanho 50 foi de 0.44 milissegundos, enquanto que a maior média de tempo foi de 26.6 milissegundos com um vetor de tamanho 500, como era esperado, já que o tempo tem um crescimento tímido conforme a entrada aumenta, como pode-se observar no gráfico a seguir:



Já a curva de crescimento do Bubble Sort teve uma variação bem maior, a média com o vetor de 50 elementos foi de 42.06 milissegundos, já a média com o vetor de 500 elementos foi de 4256.82 milissegundos, com uma variação média

aproximadamente de 100 ms/N, sendo de 10 a variação média de N, assim é possível perceber a taxa de crescimento quadrática conforme o aumento de N como era esperado, o gráfico abaixo remonta as médias do Bubble Sort obtidas:



A diferença de eficiência em termos de tempo é exorbitante quando a entrada (N) passa de 250 elementos, enquanto o Bubble Sort leva mais de um segundo para executar, o Merge Sort precisa por volta de 8.5 milissegundos, visto que o Bubble Sort tem complexidade, no pior caso, quadrática e o Merge Sort tem complexidade N log N. A diferença entre a eficiência dos algoritmos pode ser observada no gráfico a seguir: