USP – UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E COMPUTAÇÃO (ICMC)
DISCIPLINA: LABORATÓRIO DE INTRODUÇÃO A CIÊNCIA DA COMPUTAÇÃO II

RELATÓRIO 01 - EFICIÊNCIA DE ALGORITMOS DE ORDENAÇÃO

Carlos Filipe de Castro Lemos
nUSP: 12542630

1 Introdução

Durante as aulas da disciplina de Laboratório de Ciência da Computação II da USP de São Carlos, os alunos foram estimulados a se questionarem sobre a eficiência de mecanismos de ordenação que eram objeto de estudo. Nesse contexto, aprenderam os elementos de constituição dos algoritmos, as funções de eficiência, métodos de análise de eficiência e variáveis que interferiam nas suas performances.

Posteriormente, os alunos foram estimulados a desenvolverem um software de ordenação e, por meio dele, deveriam realizar a medição do tempo que o programa demoraria para organizar vetores de tamanhos diferentes (com 25, 100, 1000 e 10000 elementos). Vale acrescentar que tais experimentos deveriam englobar os três mecanismos de ordenação estudados, quais sejam: Bubble Sort, Insertion Sort e Merge Sort.

Além disso, eles deveriam também consolidar os dados e comparar os resultados obtidos. Dessa forma, também, os alunos poderiam solidificar os ensinamentos adquiridos e poderiam comprovar a veracidade das informações ministradas em sala de aula.

2 Metodologia e Desenvolvimento

2.1 Metodologia

A metodologia utilizada no presente relatório está diretamente relacionada com a medição de eficiência dos algoritmos de ordenação (Bubble Sort, Insertion Sort e Merge Sort). Nesse contexto, é preciso esclarecer que o termo "eficiência", embora apresente variados significados, está empregado no sentido de *rapidez* ou *velocidade*. Além disso, é conveniente explicitar que foram tomados alguns cuidados no cálculo do tempo, na obtenção dos

resultados e na sua posterior comparação.

No *cálculo do tempo*, observou-se as seguintes medidas mitigadoras de erros:

1. **Casos de Teste:** os casos testes consideraram a quantidade de elementos (25, 100, 1000 e 10000) e a qualidade do vetor (pior e melhor caso). Para tanto, considerou-se vetores com números inteiros gerados aleatoriamente na faixa de 0 a 100. Como melhor caso, considerou-se os vetores devidamente ordenados e para o pior caso aqueles que foram preenchidos de forma decrescente.

2. **Geração Vetores:** valendo-se da linguagem C, os casos de testes foram gerados de forma aleatória, por meio da biblioteca *<time.h>* (funções *rand()* e *srand()* - alimentadas por um *seed* baseado no horário em que foi feito o teste). Vale dizer que tal vetor foi copiado para outros, de modo que cada medição de tempo, relativa aos vários mecanismos de ordenação, tiveram os mesmos casos testes. Essa medida teve a finalidade de assegurar a solidez dos dados para comparação de resultados.

3. **Medição do Tempo:** a medição do tempo foi realizada por meio da biblioteca *<time.h>*, que disponibiliza a função *clock()*. Essa função permitiu a demarcarção do tempo antes e depois da execução e, mediante uma simples subtração, obter o tempo de execução do código.

Na *consolidação dos resultados*, também adotou-se um método de diluição de erros. Isto é, com a finalidade de evitar variações de resultados, em razão de elementos impossíveis de serem controlados, realizou-se 10 medições para cada caso teste, sendo certo que, posteriormente, calculou-se a média aritmética simples do conjunto.

Por fim, foram realizadas *comparações* entre os resultados médios obtidos nos passos anteriores e entre os mecanismos de ordenação entre si (melhores e piores casos). Nesse contexto, vale acrescentar que a comparação dos resultados médios foi realizada com base no tempo em que o programa demorou para calcular uma única operação, sendo certo que, para obter esse número, dividiu-se o tempo médio pela quantidade de elementos do caso teste (25, 100, 1000 ou 10000 elementos). Isso posibilitou a verificação da velocidade com que o algoritmo de ordenação operou individualmente. Porém, nos casos de comparação entre métodos diferentes, confrontou-se os dados de tempo médio dos mecanismos.

2.2 Bubble Sort e suas Características

2.2.1 Conceito

O método de ordenação do Bubble Sort traz como ideia principal o fato de que se deve percorrer várias vezes o mesmo vetor e, a cada passagem, deverá levar o elemento *chave* para o final do vetor. Para tanto, anda pelo vetor, posição por posição, realizando trocas entre elementos adjacentes.

2.2.2 Código-Fonte

O código fonte do Bubble Sort traz a vantagem de ser compacto e bastante simples. Destaca-se o uso de um laço externo e de um laço interno, sendo certo que, no seu âmago, encontra-se uma estrutura de comparação para realizar a troca de posições.

```
int* bubbleSort(int* vetor, int tamanho){
    int i, j;
    for(i = 0; i < tamanho-1; i++){
        for(j = 0; j < tamanho-1-i; j++){
            if (vetor[j] > vetor[j+1]){
                int aux = vetor[j];
                vetor[j] = vetor[j+1];
                vetor[j+1] = aux;
            }
        }
    }
    return vetor;
}
```

2.2.3 Função de Eficiência e Casos Específicos

De acordo com a literatura especializada, o método de ordenação Bubble Sort, embora muito simples e compacto, descompensa suas benesses promovendo um alto custo de execução. Em qualquer situação, sua complexidade no tempo é do tipo **quadrática** ($\Theta(cn^2)$), pois irá percorrer todos os casos, independente de estar completamente ordenado ou desordenado. Vale acrescentar que especialistas enaltecem a possibilidade de se modificar o código-fonte para encerrar os laços de repetição caso o vetor esteja totalmente ordenado, porém tal cuidado não foi adotado nos experimentos desse relatório.

2.3 Insertion Sort e suas Características

2.3.1 Conceito

O Insertion Sort percorre todos os elementos de um vetor, e, em cada posição que se encontra, copia o elemento (*chave*) e faz o retorno até o momento de fazer sua inserção na posição ordenada. Embora utilize laços de repetição, e passe por todos os elementos do vetor sequencialmente, o fato é que seu mecanismo é mais eficiente que aquele do Bubble Sort, pois os retornos do Insertion Sort são mais velozes que as voltas completas do outro. Além disso, há muito menos comparações para serem realizadas.

2.3.2 Código-Fonte

O código fonte do Insertion Sort traz a vantagem de ser bem compacto e utilizar poucas linhas de instruções. Destaca-se por utilizar um laço externo (tipo *for()* que controla a movimentação sobre os elementos do vetor) e um laço interno (tipo *while()*, que promove o retorno e a inserção do elemento chave na posição ordenada).

```
int* insertionSort(int* vetor, int tamanho){
    int j;
    for(j = 1; j < tamanho; j++){
        int chave = vetor[j];
        int i = j - 1
        while (i >= 0 && vetor[i] > chave){
            vetor[i+1] = vetor[i];
            i--;
        }
        vetor[i+1] = chave;
    }
    return vetor;
}
```

2.3.3 Função de Eficiência e Casos Específicos

De acordo com a literatura especializada, a ordem de eficiência do Insertion Sort é do tipo **quadrática** para quase todos casos ($\Theta(cn^2)$). No pior caso, pode ser expresso como $f(n) = an^2 + bn + c$ e, nos melhores casos, teremos baixa complexidade O(n).

2.4 Merge Sort e suas Características

2.4.1 Conceito

O Merge Sort é um mecanismo de ordenação que adota a técnica de dividir para conquistar. Isto é, tomando um conjunto de n-elementos, o Merge Sort irá dividí-los virtualmente em subconjuntos cada vez menores (pela metade), de modo que, ao final dessa primeira etapa (conquista), terá apenas conjuntos unitários. Na sequência, por meio do método da intercalação, irá reunir e ordenar, dois a dois, os elementos, ou seja, irá juntar elementos em subconjuntos de duas unidades. Na sequência, haverá a repetição do método de intercalação, porém, agora, não serão com elementos unitários, mas, sim, com os elementos dos subconjuntos de dois elementos, formando, dessa forma, subconjuntos de 4

elementos. Esse padrão irá se repetir, gerando subconjuntos de 8 elementos, e assim sucessivamente até que, ao final dessa última etapa (conquista), a sequência de n-elementos esteja completamente ordenada.

2.4.2 Código-Fonte

O código fonte do Merge Sort é bastante extenso. Inicialmente, trabalha com metodologia recursiva para dividir o vetor em subconjuntos unitários. Depois, utiliza a função acessória de intercalação para agregar e ordenar os seus elementos.

```
void intercala(int* vetor, int inicio, int centro, int fim){
    int* vetorAux = (int*)malloc(sizeof(int) * (fim-inicio)+1);
    int i = inicio;
    int j = centro+1;
    int k = 0;
    while(i <= centro && j <= fim){
        if (vetor[i] <= vetor[j]){
            vetorAux[k] = vetor[i];
            i++;
        }
        else{
            vetorAux[k] = vetor[j];
            j++;
        }
        k++;
    }
    while(i <= centro){
        vetorAux[k] = vetor[i];
        i++;
        k++;
    }
    while(j <= fim){
        vetorAux[k] = vetor[j];
        j++;
        k++;
    }
    for(i = inicio, k = 0; i <= fim; i++,k++)
        vetor[i] = vetorAux[k];
    free(vetorAux);
}
```

```
void mergeSort(int* vetor, int inicio, int fim){
    if (fim <= inicio) return;
    int centro = (int)((inicio+fim)/2.0);
    mergeSort(vetor, inicio, centro);
    mergeSort(vetor, centro+1, fim);
    intercala(vetor, inicio, centro, fim);
}
```

2.4.3 Função de Eficiência e Casos Específicos

Em primeiro lugar, é preciso anotar que, como o Merge Sort usa recursividade e vetores acessórios, faz grande uso de memória, diferindo-o dos outros dois mecanismos referente ao espaço de complexidade. Porém, sua complexidade no tempo de execução é do tipo **logarítmica** $\Theta(n \log_2 n)$ em qualquer caso teste (melhor, pior ou intermediários). Isso faz dele um poderoso mecanismo de ordenação.

3 Resultados

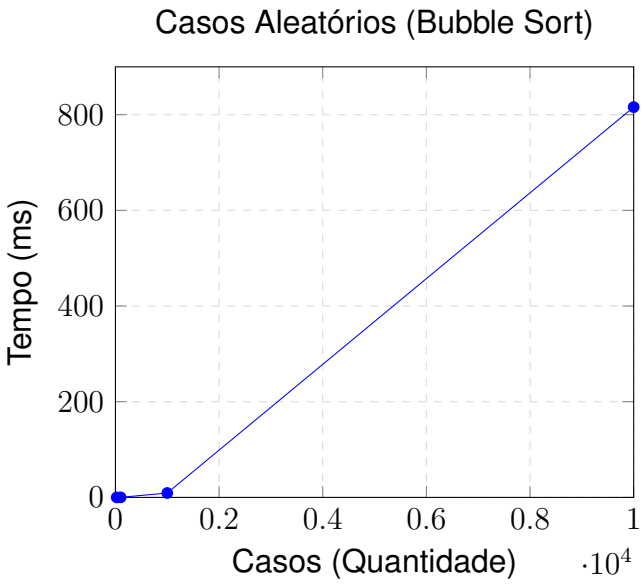
3.1 Analise Assintótica do Bubble Sort

3.1.1 Casos Aleatórios (25, 100, 1000 e 10000 elementos)

Os dados brutos consolidados, conforme metodologia, apresentaram os seguintes valores:

CASOS ALEATÓRIOS (BUBBLE SORT)

Teste	25	100	1000	10000
1	0,007	0,058	8,519	809,518
2	0,006	0,055	9,722	818,790
3	0,006	0,054	8,838	839,789
4	0,006	0,054	9,471	831,437
5	0,006	0,237	8,297	814,834
6	0,006	0,122	8,883	814,460
7	0,006	0,055	8,305	817,670
8	0,006	0,055	9,166	807,763
9	0,010	0,053	11,457	815,952
10	0,006	0,117	9,114	787,976
Média	**0,007**	**0,086**	**9,177**	**815,819**



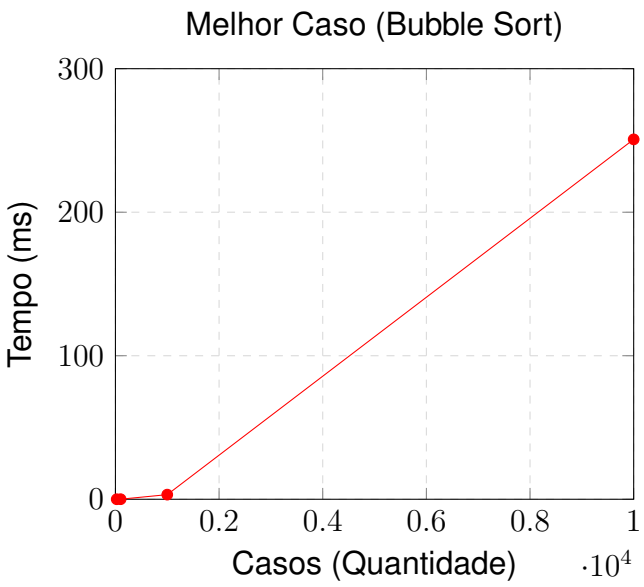
Nesse contexto, considerando os casos testes da tabela e o gráfico, percebemos que o programa demorou 0,00028 ms para calcular uma unidade no caso de 25 elementos. Da mesma forma, demorou 0,00086 ms no caso de 100 elementos, 0,009177 ms para 1000 elementos e 0,0815819 ms para 10000 elementos. Assim, houve um aumento de 3 vezes quando se passou do caso de 25 para o de 100 elementos e, da mesma forma, 32 vezes do de 25 para o de 1000 elementos e 291 vezes de 25 para o de 10000 elementos.

3.1.2 Melhor Caso (25, 100, 1000 e 10000 elementos)

Os dados brutos consolidados, conforme metodologia, apresentaram os seguintes valores:

MELHOR CASO (BUBBLE SORT)

Teste	25	100	1000	10000
1	0,003	0,030	3,296	242,827
2	0,003	0,096	3,425	245,953
3	0,004	0,028	2,888	273,424
4	0,004	0,029	3,104	241,547
5	0,003	0,024	3,552	249,626
6	0,069	0,025	3,316	248,341
7	0,004	0,024	3,784	250,622
8	0,003	0,096	2,436	256,855
9	0,004	0,031	3,438	245,159
10	0,004	0,030	2,915	253,009
Média	**0,010**	**0,041**	**3,215**	**250,736**



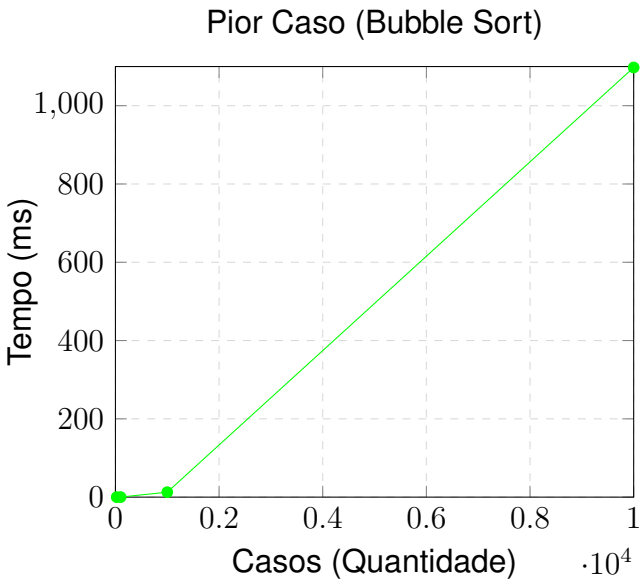
Nesse contexto, considerando os casos testes da tabela e o gráfico, percebemos que o programa demorou 0,004 ms para calcular uma unidade no caso de 25 elementos. Da mesma forma, demorou 0,0041 ms para calcular uma unidade no caso de 100 elementos, 0,003215 ms para 1000 elementos e 0,0250736 ms para o de 10000 elementos. Assim, quase não houve variação de tempo para o caso de 100 elementos em relação ao de 25; mas, diminuiu um pouco em relaçao ao de 1000 elementos e aumentou em 6 vezes em relaçao ao de 10000 elementos.

3.1.3 Pior Caso (25, 100, 1000 e 10000 elementos)

Os dados brutos consolidados, conforme metodologia, apresentaram os seguintes valores:

PIOR CASO (BUBBLE SORT)

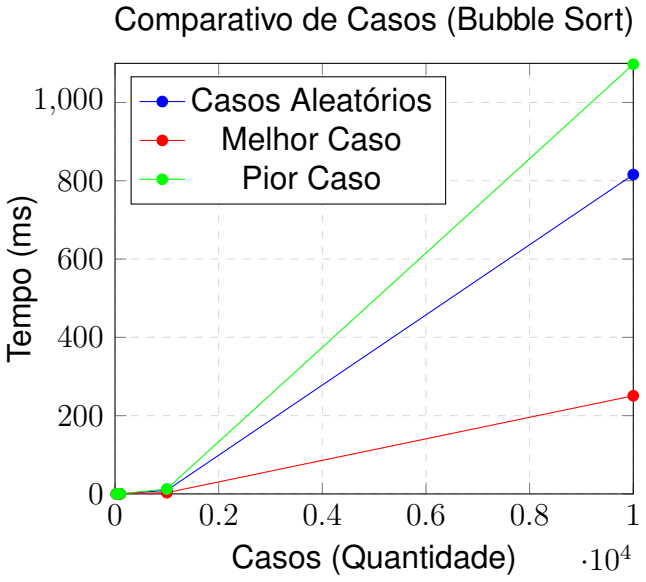
Teste	25	100	1000	10000
1	0,006	0,132	10,673	1133,875
2	0,006	0,157	11,587	1082,065
3	0,006	0,134	12,035	1076,824
4	0,006	0,138	16,433	1066,483
5	0,006	0,227	12,035	1095,325
6	0,006	0,067	13,084	1111,312
7	0,006	0,066	17,249	1134,148
8	0,006	0,067	11,048	1105,498
9	0,007	0,132	11,365	1077,777
10	0,006	0,074	11,046	1093,280
Média	**0,006**	**0,119**	**12,656**	**1097,659**



Nesse contexto, considerando os casos testes da tabela e o gráfico, percebemos que o programa demorou 0,00024 ms para calcular uma unidade no caso de 25 elementos. Da mesma forma, demorou 0,00119 ms para 100 elementos, 0,012656 ms para 1000 elementos e 0,1097659 ms para 10000 elementos. Assim, houve um aumento de 5 vezes do caso de 100 elementos em relação ao de 25; e, da mesma forma, 52 vezes em relaçao ao de 1000 elementos e 457 vezes em relaçao ao de 10000 elementos.

3.1.4 Comparativo de Casos

Comparando-se os casos do acima, por meio do gráfico abaixo, contatamos que o Bubble Sort é um mecanismo de ordenação bastante eficiente para pequenas ordenações, principalmente quando elas já estão ordenadas. Porém, para casos com números elevados de elementos, especialmente os piores casos, verificamos que ele não é uma boa opção.



Além disso, os dados obtidos nos itens anteriores revelam uma compatibilidade com as informações da literatura especializada e também com as aulas ministradas em laboratório, uma vez que denotam aumento exponencial do tempo (O(n^2)).

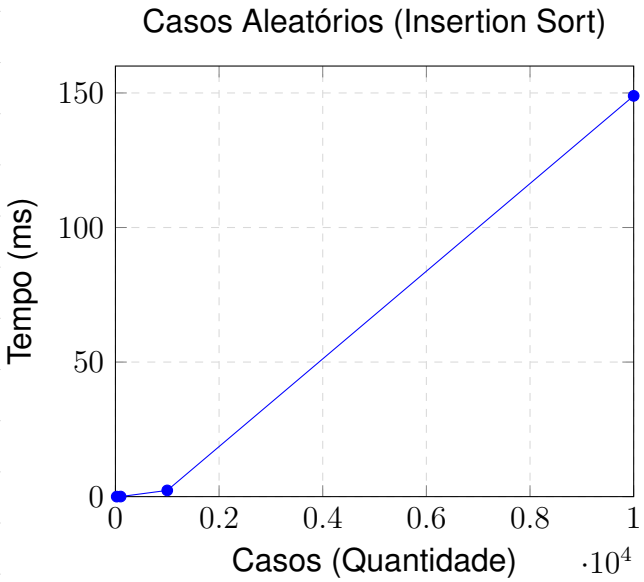
3.2 Analise Assintótica do Insertion Sort

3.2.1 Casos Aleatórios (25, 100, 1000 e 10000 elementos)

Os dados brutos consolidados, conforme metodologia, apresentaram os seguintes valores:

CASOS ALEATÓRIOS (INSERTION SORT)

Teste	25	100	1000	10000
1	0,002	0,018	1,958	148,199
2	0,002	0,018	2,625	146,019
3	0,002	0,016	2,111	149,08
4	0,002	0,015	2,554	143,08
5	0,003	0,015	2,996	159,293
6	0,002	0,017	2,24	146,961
7	0,002	0,015	1,998	147,524
8	0,002	0,016	2,783	145,265
9	0,002	0,086	1,847	145,774
10	0,001	0,016	1,958	158,049
Média	**0,002**	**0,023**	**2,307**	**148,924**

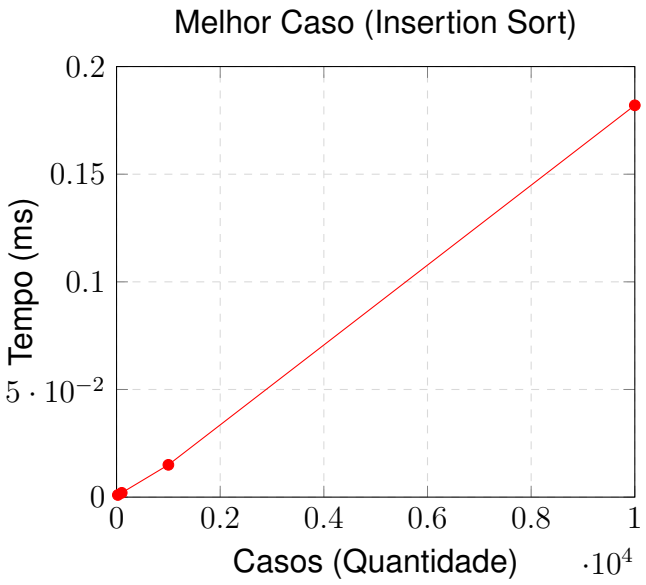


Nesse contexto, considerando os casos testes da tabela e o gráfico, percebemos que o programa demorou 0,00008 ms para calcular uma unidade no caso de 25 elementos. Da mesma forma, demorou 0,00023 ms para 100 elementos, 0,0002307 ms para 1000 elementos e 0,0148924 ms para 10000. Assim, houve um aumento de 3 vezes do caso de 100 elementos em relação ao de 25; e, da mesma forma, 28 vezes em relaçao ao de 1000 elementos e 186 vezes em relaçao ao de 10000 elementos.

3.2.2 Melhor Caso (25, 100, 1000 e 10000 elementos)

Os dados brutos consolidados, conforme metodologia, apresentaram os seguintes valores:

MELHOR CASO (INSERTION SORT)				
Teste	**25**	**100**	**1000**	**10000**
1	0,001	0,001	0,008	0,089
2	0,001	0,001	0,009	0,157
3	0,001	0,002	0,073	0,96
4	0,001	0,001	0,008	0,075
5	0,001	0,001	0,009	0,076
6	0,001	0,002	0,009	0,075
7	0,001	0,002	0,009	0,075
8	0,001	0,002	0,008	0,166
9	0,001	0,002	0,008	0,075
10	0,001	0,002	0,009	0,075
Média	**0,001**	**0,002**	**0,015**	**0,182**

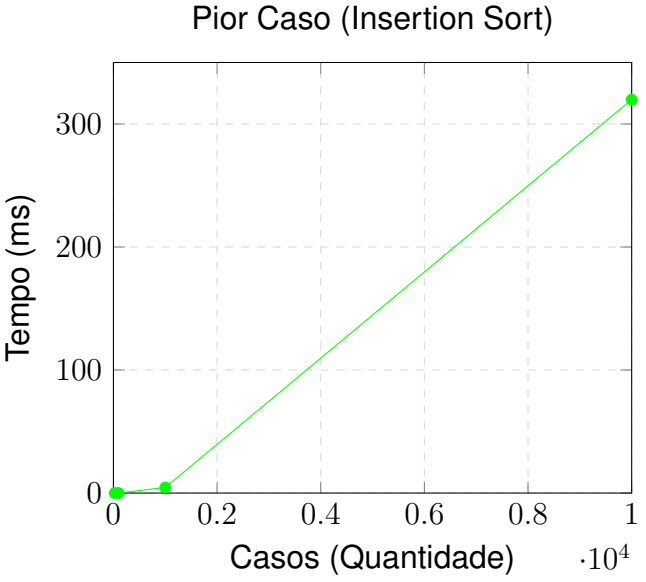


Nesse contexto, considerando os casos testes da tabela e o gráfico, percebemos que o programa demorou 0,00004 ms para calcular uma unidade no caso de 25 elementos. Da mesma forma, demorou 0,00008 ms para 100 elementos, 0,00015 ms para 1000 elementos e 0,0000182 ms para 10000. Assim, houve um aumento de 2 vezes do caso de 100 elementos em relação ao de 25; e, da mesma forma, 375 vezes em relaçao ao de 1000 elementos e 4 vezes em relaçao ao de 10000 elementos.

3.2.3 Pior Caso (25, 100, 1000 e 10000 elementos)

Os dados brutos consolidados, conforme metodologia, apresentaram os seguintes valores:

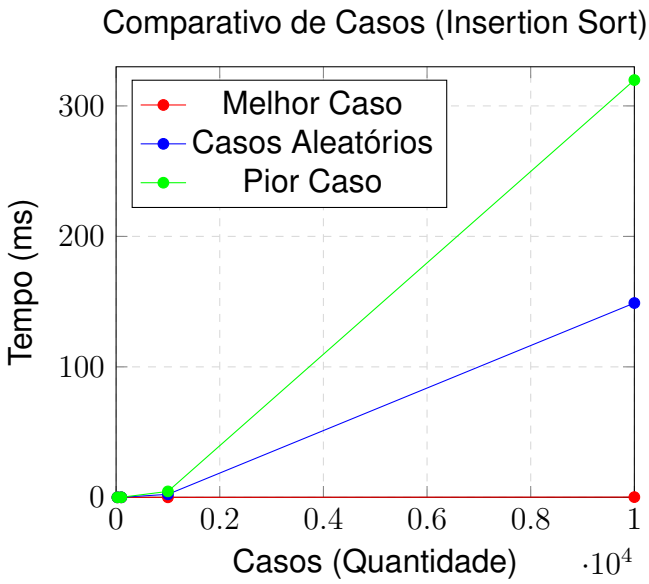
PIOR CASO (INSERTION SORT)				
Teste	**25**	**100**	**1000**	**10000**
1	0,003	0,03	7,695	332,807
2	0,003	0,029	8,005	307,232
3	0,003	0,029	3,553	327,656
4	0,002	0,03	4,389	323,154
5	0,003	0,029	3,644	319,03
6	0,003	0,03	3,719	313,047
7	0,002	0,031	3,924	328,469
8	0,004	0,03	3,38	305,974
9	0,068	0,03	4,017	311,309
10	0,003	0,096	3,158	329,468
Média	**0,009**	**0,036**	**4,548**	**319,815**



Nesse contexto, considerando os casos testes da tabela e o gráfico, percebemos que o programa demorou 0,00036 ms para calcular uma unidade no caso de 25 elementos. Da mesma forma, demorou 0,00036 ms para 100 elementos, 0,004548 ms para 1000 elementos e 0,0319815 ms para 10000. Assim, não houve variação do caso de 100 elementos em relação ao de 25; mas, houve um aumento de 12 vezes em relaçao ao de 1000 elementos e 88 vezes em relaçao ao de 10000 elementos.

3.2.4 Comparativo de Casos

Comparando-se os casos do acima, por meio do gráfico abaixo, constatamos que o Insertion Sort é mais rápido que o Bubble Sort em todos os casos. Nesse sentido, é um mecanismo de ordenação bastante eficiente para pequenas e médias ordenações. Porém, para casos com números elevados de elementos, especialmente os piores casos, verificamos que ainda não é uma boa opção, haja vista que o tempo de execução aumenta de forma significativa.



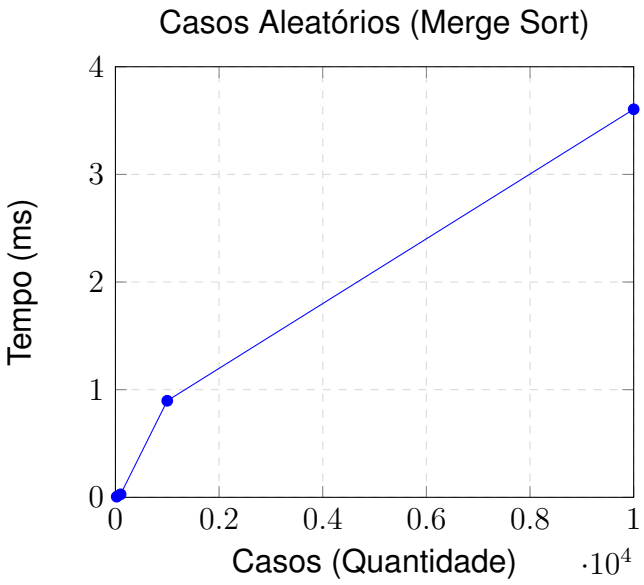
Além disso, os dados obtidos nos itens anteriores revelam uma compatibilidade com as informações da literatura especializada e também com as aulas ministradas em laboratório, uma vez que denotam aumento exponencial do tempo (O(n^2)).

3.3 Analise Assintótica do Merge Sort

3.3.1 Casos Aleatórios (25, 100, 1000 e 10000 elementos)

Os dados brutos consolidados, conforme metodologia, apresentaram os seguintes valores:

CASOS ALEATÓRIOS (MERGE SORT)				
Teste	**25**	**100**	**1000**	**10000**
1	0,006	0,087	0,871	4,119
2	0,007	0,021	0,428	4,002
3	0,006	0,021	3,014	3,477
4	0,007	0,02	0,297	3,076
5	0,007	0,022	2,448	2,9
6	0,006	0,021	0,731	3,16
7	0,006	0,02	0,381	4,131
8	0,006	0,021	0,234	3,909
9	0,007	0,023	0,334	3,849
10	0,006	0,022	0,231	3,425
Média	**0,006**	**0,028**	**0,897**	**3,605**

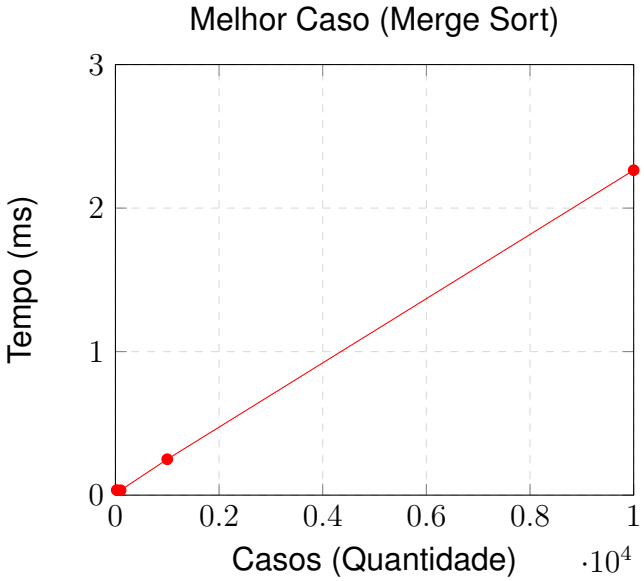


Nesse contexto, considerando os casos testes da tabela e o gráfico, percebemos que o programa demorou 0,00024 ms para calcular uma unidade no caso de 25 elementos. Da mesma forma, demorou 0,00028 ms para 100 elementos, 0,000897 ms para 1000 elementos e 0,0003605 ms para 10000 elementos. Assim, não houve variação de tempo, por unidade processada, maior que 0,5 vezes comparando-se todos os casos.

3.3.2 Melhor Caso (25, 100, 1000 e 10000 elementos)

Os dados brutos consolidados, conforme metodologia, apresentaram os seguintes valores:

MELHOR CASO (MERGE SORT)				
Teste	**25**	**100**	**1000**	**10000**
1	0,006	0,017	0,234	2,132
2	0,006	0,181	0,31	2,401
3	0,005	0,017	0,257	2,149
4	0,006	0,018	0,008	2,129
5	0,006	0,018	0,237	2,424
6	0,234	0,017	0,363	2,538
7	0,006	0,017	0,245	2,181
8	0,006	0,017	0,16	2,339
9	0,008	0,018	0,172	2,107
10	0,071	0,018	0,517	2,239
Média	**0,035**	**0,034**	**0,250**	**2,264**



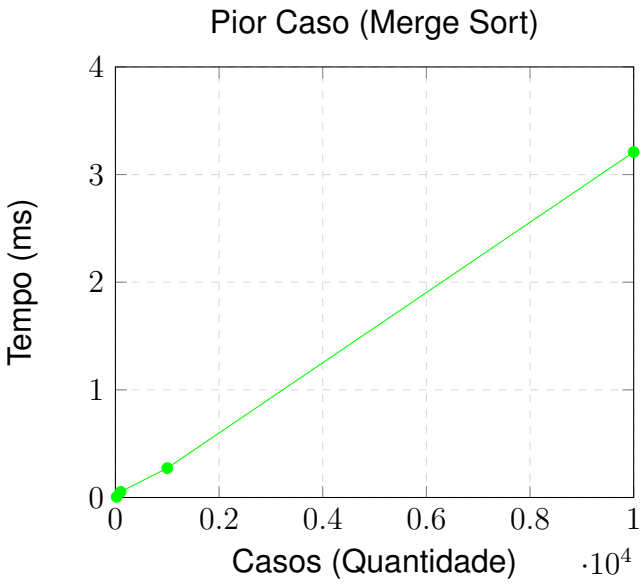
Nesse contexto, considerando os casos testes da tabela e o gráfico, percebemos que o programa demorou 0,0014 ms para calcular uma unidade no caso de 25 elementos. Da mesma forma, demorou 0,00034 ms para 100 elementos, 0,00025 ms para 1000 elementos e 0,0002264 ms para 10000 elementos. Assim, verifica-se que o caso com menor quantidade de elementos (25 elementos) foi aquele que apresentou maior tempo de execução, enquanto os outros casos apresentaram valores inferiores, porém estáveis entre si (com variações insignificantes).

3.3.3 Pior Caso (25, 100, 1000 e 10000 elementos)

Os dados brutos consolidados, conforme metodologia, apresentaram os seguintes valores:

PIOR CASO (MERGE SORT)

Teste	25	100	1000	10000
1	0,005	0,017	0,736	3,267
2	0,006	0,017	0,444	3,13
3	0,006	0,083	0,159	2,064
4	0,006	0,179	0,246	2,073
5	0,006	0,082	0,162	6,909
6	0,005	0,017	0,164	2,957
7	0,005	0,016	0,245	3,275
8	0,006	0,018	0,242	2,316
9	0,008	0,083	0,163	3,633
10	0,006	0,018	0,165	2,463
Média	**0,006**	**0,053**	**0,273**	**3,209**

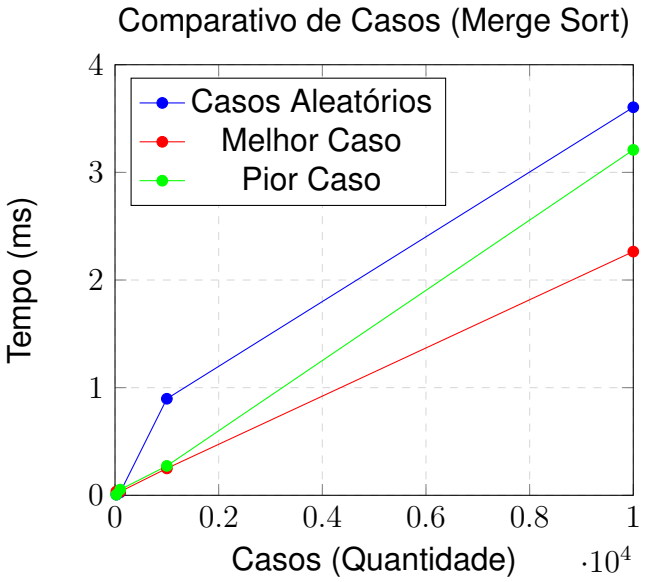


Nesse contexto, considerando os casos testes da tabela e o gráfico, percebemos que o programa demorou 0,00024 ms para calcular uma unidade no caso de 25 elementos. Da mesma forma, demorou 0,00053 ms para 100 elementos, 0,000273 ms para 1000 elementos e 0,0003209 ms para 10000 elementos. Assim, verifica-se que todos os casos de execução apresentaram valores estáveis entre si (com variações insignificantes).

3.3.4 Comparativo de Casos

Comparando-se os casos do acima, por meio do gráfico abaixo, constatamos que o Merge Sort apresenta características peculiares. Primeiro porque, diferente do Bubble Sort e do Insertion Sort, ele é um mecanismo que demora a iniciar, mas, uma vez em funcionamento, realiza operações de forma muito veloz. Tanto é assim que, diante dos casos de testes (aleatórios, melhor e pior caso), os resultados foram muito rápidos, apresentando

pouca variação de tempo, especialmente em vetores com elevados números de elementos.

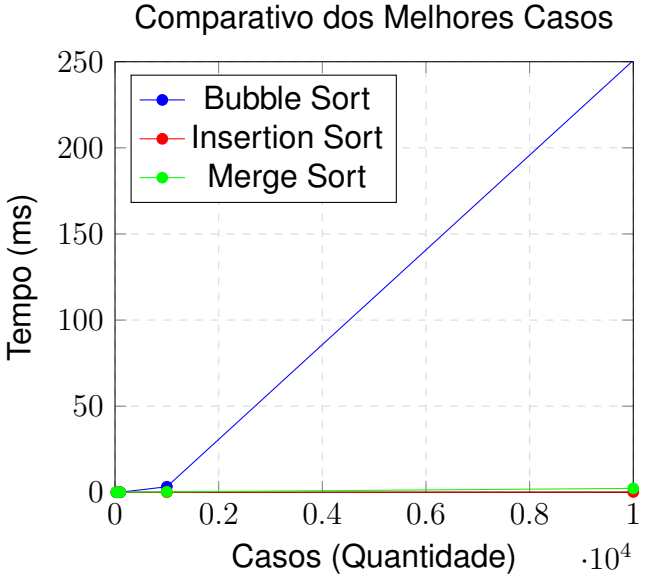


Além disso, os dados obtidos nos itens anteriores revelam uma compatibilidade com as informações da literatura especializada e também com as aulas ministradas em laboratório, uma vez que denotam velocidade típica de funções logarítimas (O($n \log n$)).

3.4 Comparativo do Bubble Sort, Insertion Sort e Merge Sort

3.4.1 Comparativo dos Melhores Casos

Os casos testes dos melhores casos do Bubble Sort, Insertion Sort e Merge Sort foram plotados conjuntamente no gráfico abaixo.

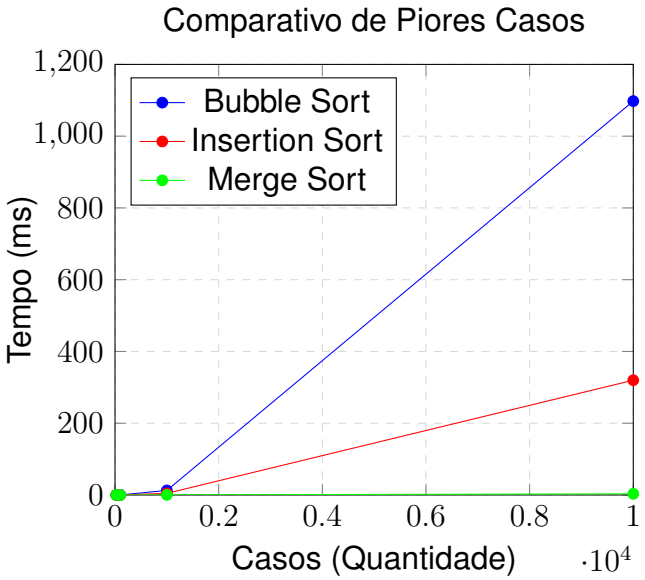


Tendo em vista essas informações, a comparação revela que, para o melhor caso, não há diferença significativa entre a eficiência do Insertion Sort e do Merge Sort (tanto que as linhas estão sobrepostas no gráfico). Porém, em situação diversa, temos o Bubble Sort, que apresenta elevado tempo de execução após casos acima de 1000 elementos.

Nesse contexto, os dados do gráfico mostram compatibilidade com as informações da literatura especializada e também com as aulas ministradas em laboratório.

3.4.2 Comparativo dos Piores Casos

Os casos testes dos piores casos do Bubble Sort, Insertion Sort e Merge Sort foram plotados conjuntamente no gráfico abaixo.



Tendo em vista essas informações, a comparação revela que, para o pior caso, há diferenças entre os três mecanismos de ordenação, sendo certo que podemos discriminar os mecanismos de ordenação na seguinte forma: Bubble Sort (menos eficiente), Insertion Sort (intermediário) e Merge Sort (mais eficiente). Tanto é assim que, considerando os casos de maior quantidade de elementos (10000), o Insertion Sort gastou 99 vezes mais tempo que o Merge Sort, enquanto o Bubble Sort, 342 vezes.

Nesse contexto, os dados do gráfico mostram compatibilidade com as informações da literatura especializada e também com as aulas ministradas em laboratório.

4 Conclusão

Embora tenha havido pequenas variações de performance em casos específicos, concluimos o relatório confirmando as expectativas reunidas inicialmente. Nesse sentido, comprovou-se, por meio de dados empíricos, as informações fornecidas pela literatura especializada

e pelas aulas de laboratório. Isto é, mostrou-se, por meio da análise assintótica dos algoritmos de ordenação, que os mecanismos Bubble Sort e Insertion Sort possuem eficiência O(n^2) e o Merge Sort é o mais eficiente deles (possuindo ordem logarítmica O($n.\log n$)).

5 Referências Bibliográficas

CORMEN, Thomas et alii. Algoritmos: Teoria e Prática. Rio de Janeiro: Editora Elsevier, 2002.

MERGE SORT, in Wikipedia: a Enciclopedia Livre. Acesso: <https://pt.wikipedia.org/wiki/Merge_sort#Complexidade>. Data de acesso: 05/10/2021.

SOUZA, José Francisco. Método Insertion Sort - Estrutura de Dados II. Universidade Federal de Juiz de Fora. Acesso: <https://www.ufjf.br/jairo_souza/files/2009/12/2-Ordenação-InsertionSort.pdf>. Data de acesso: 05/10/2021.