Análise Teórica e Experimental de Algoritmos para o Problema de Ordenação

Adalberto Andrade

adalberto.andrade@eic.cefet-rj.br

Flavio Matias D. Carvalho

flavio.carvalho@eic.cefet-rj.br

Francimary Garcia de Oliveira

francimary.oliveira@eic.cefet-rj.br

Abstract: *The present work aims to analyze in a scientific and experimental way, the problem of sorting and some algorithms known to solve this problem. The theoretical analysis was presented as well as the comparison of the processing times between the algorithms Bubble Sort, Insertion Sort, Merge Sort and Quick Sort.*

Resumo: *O presente trabalho tem por objetivo analisar de forma científica e experimental o problema de ordenação e alguns algoritmos conhecidos que se prestam a solucionar tal problema. Foi apresentada a análise teórica das complexidades e comparados os tempos de processamento entre os algoritmos Bubble Sort, Insertion Sort, Merge Sort e Quick Sort.*

1. Introdução

Este trabalho visa consolidar os conceitos apresentados na disciplina de Análise e Projeto de Algoritmos, que compõe a grade de disciplinas do Mestrado em Ciência da Computação. Com isso, foi exercitado o método científico na análise por meio da comparação teórica e experimental de algoritmos selecionados.

O objetivo do trabalho é permitir que se comece a tomar contato com o método científico na área de análise de algoritmos, exercitando, ao mesmo tempo, abordagens teóricas e experimentais para a caracterização dos parâmetros de eficiência de um ou mais algoritmos. Dessa forma, optou-se por ambientes de programação e desenvolvimento de maior familiaridade e problemas computacionais e algoritmos que despertem interesse.

O trabalho busca responder e demonstrar o tempo de execução teórico de cada algoritmo no pior caso, no caso médio e no melhor caso, e em seguida, evidenciar as análises teóricas através do comportamento experimental. Também será mostrado o resultado de implementações de algoritmos em diferentes ambientes como forma de

evidenciar o comportamento assimptótico semelhante. Por fim, se propõe a verificar de que formas se comportam diferentes algoritmos para o mesmo problema.

Escolheu-se para esta finalidade o problema de ordenação, que pode ser resolvido de diversas maneiras por diferentes algoritmos. Foram selecionados os algoritmos denominados como: **Bubble Sort, Insertion Sort, Merge Sort e Quick Sort**.

2. O Problema de Ordenação

Ordenação é um dos problemas mais estudados em ciência da computação. Além de ser a base para muitos algoritmos, consome um tempo de processamento considerável para muitas aplicações típicas. Existe uma grande variação de problemas e algoritmos de ordenação [Manber 1989].

Este trabalho avaliará alguns dos algoritmos que se prestam a resolver o problema de ordenação, que basicamente consiste em:

> ***Dado n números distintos*** $x_1, x_2, \ldots, x_n$, ***organizá-los em ordem incremental. Ou seja, encontrar uma sequencia de índices distintos*** $1 \leq i_1, i_2, \ldots, i_n \leq n$, ***tal que***
>
> $x_{i1} \leq x_{i2} \leq \ldots x_{in}$.

2.1 Bubble Sort

O algoritmo conhecido como bubble sort também é conhecido como ordenação por flutuação ou "por bolha". É um algoritmo de ordenação bem simples que percorre a lista diversas vezes, a cada passagem fazendo o maior elemento da sequência “flutuar” para o topo, como bolhas em um tanque de água que vão para seu próprio nível, e disso vem o nome do algoritmo.

Este algoritmo percorre a lista de itens ordenáveis do início ao fim, verificando a ordem dos elementos dois a dois, e trocando-os de lugar se necessário. Percorre-se a lista até que nenhum elemento tenha sido trocado de lugar na passagem anterior.

No melhor caso, o algoritmo executa n operações relevantes, onde n representa o número de elementos da lista. No pior caso, são feitas n^2 operações, ou seja, a complexidade desse algoritmo é de Ordem quadrática $O(n^2)$ [Cormen et al. 1990].

2.2 Insertion Sort

O algoritmo de ordenação Insertion Sort, ou ordenação por inserção, é um algoritmo que, dado uma estrutura, como uma lista, constrói uma matriz final com um elemento de cada vez, uma inserção por vez.

Para se entender o funcionamento da ordenação por Insertion Sort, pode ser feito um paralelo com a forma de como algumas pessoas organizam um baralho num jogo de cartas. Imagine que você está jogando cartas. Você está com as cartas na mão e elas estão ordenadas. Você recebe uma nova carta e deve colocá-la na posição correta da sua mão de cartas, de forma que as cartas obedeçam a ordenação.

A cada nova carta adicionada a sua mão de cartas, a nova carta pode ser menor que algumas das cartas que você já tem na mão ou maior, e assim, você começa a comparar a nova carta com todas as cartas na sua mão até encontrar sua posição correta. Você insere a nova carta na posição correta, e, novamente, sua mão é composta de cartas totalmente ordenadas. Então, você recebe outra carta e repete o mesmo procedimento. Então outra carta, e outra, e assim por diante, até você não receber mais cartas [Cormen et al. 1990].

Desta forma, resumidamente, a ideia por trás da ordenação por inserção consiste em percorrer as posições da sequencia, começando com o índice 1 (um), e inserir a nova carta no lugar correto na sequencia ordenada à esquerda daquela posição.

2.3 Merge Sort

O algoritmo Merge Sort utiliza o conceito de Divisão e Conquista. Isto é, dado um problema a resolver em n entradas, nessa estratégia quebramos as entradas em k subconjuntos distintos $1 < k \leq n$, obtendo k subproblemas. Esses subproblemas devem ser resolvidos e então um método deve ser encontrado para combinar as soluções em uma solução do todo. Enquanto os subconjuntos ainda se mantem relativamente grandes, a estratégia de divisão e conquista pode ser reaplicada, até que um subproblema fique tão pequeno, que para ser resolvido não necessite ser quebrado.

No algoritmo Merge Sort, temos uma sequencia de n elementos a[1], a[2],...,a[n], onde a ideia geral é quebra-los em dois conjuntos a[1],...,a[n/2] e a[n/2 + 1],...,a[n]. Cada conjunto é individualmente ordenado e as sequencias resultantes ordenadas, são combinadas para produzir uma única sequencia ordenada de n elementos [Manber 1989].

2.4 Quick Sort

A abordagem de divisão e conquista é novamente utilizada para se chegar a um método de ordenação eficiente e diferente do Merge Sort. No Merge Sort o conjunto a[1:n] é dividido ao meio em dois subconjuntos que são independentemente ordenados e posteriormente combinados.

No algoritmo Quick Sort, a divisão em dois subconjuntos é realizada de tal forma que os subconjuntos ordenados não necessitem ser combinados posteriormente. Isto é conseguido pelo rearranjo dos elementos em a[1:n], tal que $a[i] \leq a[j]$ para todo i entre 1 e m e para todo j entre m+1 e n e algum m, $1 \leq m \leq n$. Portanto os elementos em a[1:m] e a[m+1,n] podem ser independentemente ordenados e sem necessidade de combinação. O rearranjo dos elementos é realizado através da escolha de algum elemento de a[ ], onde t = a[s] e então reordenando os demais elementos, tal que todos os elementos que aparecerem antes de e em a[1:n] são menores ou iguais a t e todos os elementos que aparecerem depois de t são maiores ou igual a e. Este rearranjo é denominado como particionamento [Manber 1989].

3. Análise Teórica dos Algoritmos

3.1. Bublle Sort

O algoritmo Bubble Sort é considerado uma das abordagens mais simples para realização da operação de ordenação. Trabalha com a comparação entre dois elementos por vez e a realização de uma troca caso estejam em ordem contrária, de forma que na primeira execução das comparações a saída será o maior elemento ordenado no final da sequencia. Na segunda execução, o segundo maior elemento será posicionado corretamente na sequencia e assim sucessivamente, até que nenhuma troca seja realizada, indicando assim que toda a lista está ordenada [Rocha 2013].

Melhor caso: Quando a sequencia já se encontra ordenada. Onde nenhuma troca será realizada e serão necessárias **n** comparações.

Pior caso: Quando a sequencia se encontra invertida. As operações de comparação e troca de posição dos elementos são executadas da seguinte forma:

- Primeiro passo: n - 1 trocas
- Segundo passo: n – 2 trocas e assim sucessivamente

Desta forma o número de trocas será igual a: $(n – 1) + (n – 2) +(n – 3) + ... + 2 + 1 = [n(n – 1)]/2$, o que resultará em um número de trocas na ordem de $\mathbf{n^2}$.

Portanto o tempo gasto na execução do algoritmo Bubble Sort varia em ordem quadrática em relação ao número de elementos a serem ordenados.

- $T = O (n^2)$
- Atividades mais custosas:
    - Comparações e
    - Trocas de posição

Trata-se de um algoritmo lento para entradas grandes e, portanto pouco recomendado para tais cenários.

Abaixo o código (Python) utilizado neste trabalho para implementar a ordenação com o método Bubble Sort:

```
[1]def bubbleSort(listab):
    for bolhinha in range(len(listab)-1,0,-1):
        for i in range(bolhinha):
            if listab[i]>listab[i+1]:
                temp = listab[i]
                listab[i] = listab[i+1]
                listab[i+1] = temp
```

3.2. Insertion Sort

- É eficiente na ordenação de um número pequeno de elementos.
- Funciona de forma análoga a maneira que ordenamos cartas de baralho na mão.
- Para o perfeito entendimento deste algoritmo, importante provar três coisas:
    - Inicialização: É verdadeira antes da primeira iteração.
    - Manutenção: Se é verdadeiro no começo da iteração se manterá verdadeiro também na seguinte.
    - Término: Quando o ciclo for concluído o vetor estará ordenado de forma correta.

Estes três passos podem ser comparados às propriedades do método da indução matemática como abaixo apresentado:

Inicialização: Base de indução
Manutenção: Passo indutivo
Término: A indução é concluída quando o ciclo termina.

- **Melhor caso:** Quando os elementos já estão ordenados. Serão realizadas 1 comparação e 1 troca por etapa, totalizando (n – 1) comparações e (n – 1) trocas.
- **Pior caso:** Quando os elementos da sequencia estão na ordem inversa. O n-ésimo número é comparado a todos os n-1 números anteriores a ele. O número total de comparações para ordenação pode ser tão alto quanto 1

[1] O código para execução deste algoritmo encontra-se disponível em: goo.gl/QwvrKX

+ 2 + ... + n-1 = ½ (n-1)(n-2), resultando em uma grandeza de ordem quadrática $O(n^2)$. Além disso, inserir o n-ésimo número na posição correta envolve movimentar outros elementos. No pior caso, n – 1 elementos serão movidos, portanto o número de movimentação de elementos também estará na $O(n^2)$.

Em relação ao tempo gasto na execução do algoritmo Insertion Sort, temos:
$T = O(n^2)$, portanto na ordem quadrática.

Trata-se de um algoritmo lento para entradas grandes e, portanto pouco recomendado para tais cenários. Mas pode se mostrar como um método bom para adicionar um pequeno conjunto de dados a uma sequencia já ordenada, originando outra sequencia ordenada, pois neste caso o custo pode ser considerado linear.

Abaixo o código (Python) utilizado neste trabalho para implementar a ordenação com o método Insertion Sort:

[2]
```
def insertionSort(lista1):
    for index in range(1,len(lista1)):
        valor_atual= lista1[index]
        posicao = index
        while posicao > 0 and lista1[posicao-1] > valor_atual:
            lista1[posicao] = lista1[posicao-1]
            posicao = posicao-1
        lista1[posicao] = valor_atual
    return lista1
```

3.3. Merge Sort

O algoritmo Merge Sort implementa o método de Divisão e Conquista, que prevê ações de Dividir, Conquistar e Combinar, da seguinte forma:

- Dividir: Dividir os n elementos da sequencia a ser ordenada, em duas sequencias com n/2 elementos cada.
- Conquistar: Ordenar as duas subsequências recursivamente, utilizando o merge sort.
- Combinar: Combinar as duas subsequências para gerar uma solução única ordenada.

[2] O código para execução deste algoritmo encontra-se disponível em: goo.gl/QwvrKX

Em relação ao tempo de processamento para o Merge Sort, temos que o tempo para operação de combinar é proporcional ao n. Então o tempo de processamento para o Merge Sort é descrito pela seguinte relação de recorrência [Horowitz 1997]:

$$T(n) = \begin{cases} a & n = 1, \text{ onde a é uma constante} \\ 2T(n/2) + cn & n > 1, \text{ onde c é uma constante} \end{cases}$$

Quando n está em potência de 2, podemos resolver esta equação por sucessivas substituições:

$$\begin{aligned} T(n) &= 2(2T(n/4) + cn/2) + cn \\ &= 4T(n/4) + 2cn \\ &= 4(2T(n/8) + cn/4) + 2cn \\ &. \\ &. \\ &= 2^K T(1) + Kcn \\ &= an + cn \text{ Log } n \end{aligned}$$

Notamos que, se $2^K < n \leq 2^{K+1}$ então $T(n) \leq T(2^{K+1})$. Portanto:

$$T(n) = O(n \log n)$$

Melhor caso: Onde nunca é necessário trocar após comparações. A complexidade é O(n log n).
Pior caso: Onde sempre é necessário trocar após comparações. A complexidade é O(n log n).

Para o Merge Sort não há tanta importância se a sequencia está no melhor, médio ou pior caso, porque para qualquer um dos cenários a complexidade estará na ordem n log n.
Isso se dá pelo fato de que este algoritmo independentemente da situação em que se encontra a sequencia, sempre irá dividir e intercalar.

Trata-se de um algoritmo eficiente para grandes entradas, no entanto possui algumas desvantagens que precisam ser avaliadas:

- Utiliza funções recursivas;
- Gasto extra de memória. O algoritmo cria uma cópia do vetor para cada nível da chamada recursiva, totalizando um uso adicional de memória igual a (n log n).

Abaixo o código (Python) utilizado neste trabalho para implementar a ordenação com o método Merge Sort:

[3]
```
def mergeSort(lista):
#    print("Divisao ",lista)
    if len(lista)>1:
        meio = len(lista)//2
        ladoesquerdo = lista[:meio]
        ladodireito = lista[meio:]

        mergeSort(ladoesquerdo)
        mergeSort(ladodireito)

        i=0
        j=0
        k=0
        while i < len(ladoesquerdo) and j < len(ladodireito):
            if ladoesquerdo[i] < ladodireito[j]:
                lista[k]=ladoesquerdo[i]
                i=i+1
            else:
                lista[k]=ladodireito[j]
                j=j+1
            k=k+1

        while i < len(ladoesquerdo):
            lista[k]=ladoesquerdo[i]
            i=i+1
            k=k+1

        while j < len(ladodireito):
            lista[k]=ladodireito[j]
            j=j+1
            k=k+1
#    print("Juntando ",lista)
    return lista
```

3.4. Quick Sort

O algoritmo Quick Sort, em relação aos demais avaliados neste trabalho, se mostra como o mais eficiente, devido à sua eficiência média. Também tendo a vantagem de realizar a ordenação de forma já organizada. A abordagem de Divisão e Conquista, é refletida no Quick Sort de modo recursivo da seguinte maneira:

---

[3] O código para execução deste algoritmo encontra-se disponível em: goo.gl/QwvrKX

- Dividir: A sequencia é particionada em subsequências.
- Conquistar: A ordenação é realizada nas subsequências, através de chamadas recursivas ao Quick Sort.
- Combinar: Devido às subsequências já estarem ordenadas, nenhum trabalho é necessário para combiná-las.

O tempo de execução do Quick Sort depende se o particionamento está balanceado ou não, que por sua vez depende de quais elementos são usados para a partição (pivô).
Melhor caso: Com particionamento balanceado, quando a partição produz duas subsequências com não mais de n/2 elementos, o algoritmo tem tempo similar ao merge sort, ou seja, na ordem de **O(n log n).**
Pior caso: Com particionamento desbalanceado, por exemplo, uma subsequência com n-1 elementos e outra com 0 elementos (pivô é o maior ou o menor elemento da sequencia), seu tempo é similar ao Insertion Sort, ou seja, **$O(n^2)$.**

Trata-se de um algoritmo indicado para grandes entradas, tendo a vantagem em relação ao Merge Sort, de não realizar gasto extra de memória e, portanto sendo o menos custoso em relação aos demais algoritmos avaliados.

Abaixo o código (Python) utilizado neste trabalho para implementar a ordenação com o método Quick Sort:

[4]
```
def quickSort(lista):
  ajudaquickSort(lista,0,len(lista)-1)

def ajudaquickSort(lista,primeiro,ultimo):
  if primeiro<ultimo:

     pontodivisao = particao(lista,primeiro,ultimo)

     ajudaquickSort(lista,primeiro,pontodivisao-1)
     ajudaquickSort(lista,pontodivisao+1,ultimo)

#O Quick Sort primeiro seleciona um valor, chamado de valorpivo
#Selecionamos o primeiro valor como pivo
def particao(lista,primeiro,ultimo):
  valorpivo = lista[primeiro]

#Seleciona-se os marcadores para depois dividir
  marcaesquerda = primeiro+1
  marcadireita = ultimo

#Vai deslocando ate achar um valor maior que o pivo
```

[4] O código para execução deste algoritmo encontra-se disponível em: goo.gl/QwvrKX

```
    feito = False
    while not feito:

        while marcaesquerda <= marcadireita and lista[marcaesquerda] <=
valorpivo:
            marcaesquerda = marcaesquerda + 1

    # Rearranja marcas (chaves) de modo que as "menores" precedam "maiores"
    # Depois ordena as duas sublistas de chaves menores e maiores recursivamente
    # ate que a lista completa se encontre ordenada.
        while lista[marcadireita] >= valorpivo and marcadireita >= marcaesquerda:
            marcadireita = marcadireita -1

        if marcadireita < marcaesquerda:
            feito = True
        else:
            temp = lista[marcaesquerda]
            lista[marcaesquerda] = lista[marcadireita]
            lista[marcadireita] = temp

    temp = lista[primeiro]
    lista[primeiro] = lista[marcadireita]
    lista[marcadireita] = temp

return marcadireita
```

4. Comparação teórica dos algoritmos

Na Tabela 1 abaixo, n é o número de itens a serem classificados. As colunas “Melhor”, "Médio" e "Pior" dão a complexidade de tempo em cada caso, sob o pressuposto de que o comprimento de cada chave é constante, e que, portanto, todas as comparações, trocas e outras operações necessárias acontecem em tempo constante. Os tempos de execução listados abaixo devem ser entendidos como sendo dentro da grande notação O, daí a base dos logaritmos não importa.

Tabela 1. Comparação de Complexidade

	Tempo		
Algoritmo	**Melhor**	**Médio**	**Pior**
Bubble Sort	O(n)	O(n^2)	O(n^2)
Insertion Sort	O(n)	O(n^2)	O(n^2)
Merge Sort	O(n log n)	O(n log n)	O(n log n)
Quick Sort	O(n log n)	O(n log n)	O(n^2)

5. Resultados experimentais

Os códigos testados foram escritos na linguagem Python 2.7.13. Os algoritmos foram implementados em diferentes ambientes (computadores distintos, com sistemas operacionais distintos) de forma a ser possível evidenciar o comportamento assimptótico semelhante destes.

A ordenação foi executada para diferentes quantidades de itens, gerados aleatoriamente. No caso, foram testados sequencias de 100, 400, 1.000, 4.000 e 10.000 itens.

A Figura 1 abaixo mostra os tempos de execução dos diferentes algoritmos anteriormente indicados, em um ambiente com processador Intel Core i5-3210M 2.50 GHz, memória RAM de 6,00 GB e Sistema Operacional Windows 10.

Na Figura 2, utilizou-se uma plataforma computacional mais antiga, com menor poder de processamento e menos memória.

Na Figura 3, testou-se em uma plataforma com mais poder de processamento e mais memória.

Os valores que deveriam ser indicados no gráfico, onde o programa registrou que o tempo de execução foi zero, não aparecem. Isso porque uma escala logarítmica não pode ser usada para valores negativos ou zero. Assim, verificamos os tempos apresentados após execução em ambientes computacionais distintos.

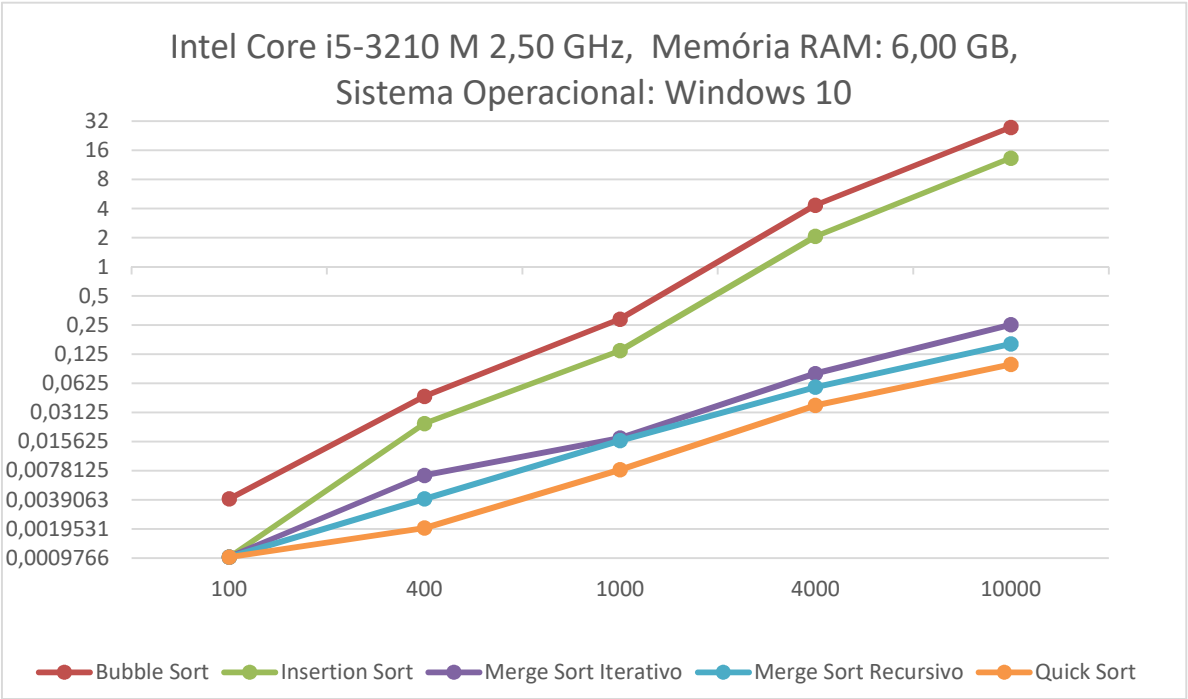

Figura 1. Tempos de execução dos diferentes algoritmos para diferentes quantidades de itens de entrada, máquina padrão.

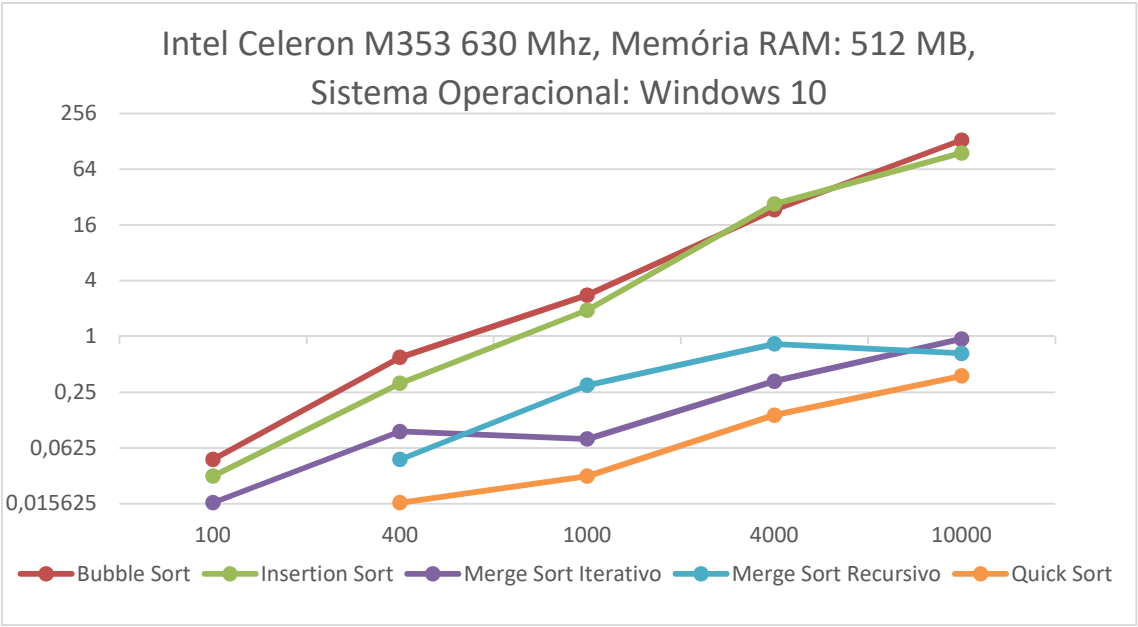

Figura 2. Tempos de execução dos diferentes algoritmos para diferentes quantidades de itens de entrada, máquina lenta.

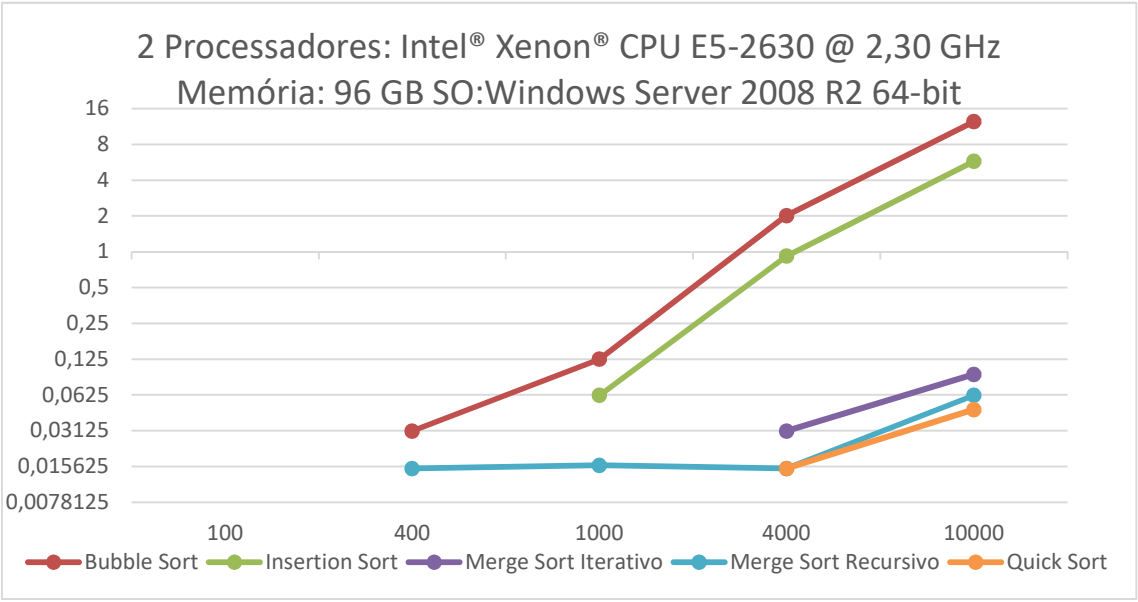

Figura 3. Tempos de execução dos diferentes algoritmos para diferentes quantidades de itens de entrada, máquina de alto desempenho.

6. Conclusão

Ficou constatado, observando os valores obtidos, que o comportamento experimental evidenciou as análises teóricas realizadas. Ou seja, no caso médio, onde temos uma lista com elementos aleatoriamente organizados, as implementações do Bubble Sort e do

Insertion Sort, executam a ordenação em um tempo maior que as implementações do Merge Sort e do Quick Sort.

Também foi possível evidenciar que ao implementarmos os algoritmos em diferentes ambientes, no caso, computadores distintos com sistemas operacionais distintos, o comportamento assimptótico foi semelhante.

Referências:

Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2009) [1990]. Algoritmos (3rd ed.). MIT Press and McGraw-Hill.

Manber, UDI; [1989] Introduction to Algorithms A Creative Approach. Adison-Weley.

Horowitz, Ellis; Sartaj, Sahni; Rajasekaran, Sanguthevar; [1997]. Computer Algorithms. Computer Science Press

Rocha, Julio Cesar Ferreira; [2013]. Complexidade de Algoritmos Insertion, Selection e Bubble Sort, Centro Universitário do Sul de Minas