PROJETO E ANÁLISE DE ALGORITMOS

Ronan José Lopes

Implementação e análise dos algoritmos de ordenação InsertionSort, MergeSort e TimSort

Universidade Federal de São João del-Rei

2021

Sumário

1 Introdução

Uma das tarefas mais comuns na implementação de algoritmos, com um grande número de aplicações possíveis, a ordenação de elementos em um conjunto se torna também objeto constante de estudo em termos de otimização. As muitas soluções disponíveis, apesar de produzirem o resultado esperado, podem se mostrar indesejadas em termos de desempenho, dependendo do cenário em que são utilizadas.

A fim de se demonstrar alguns desses cenários, bem como poder demonstrar como é feita a implementação, utilização e análise de algoritmos dessa classe, este trabalho tem como proposta um estudo de caso envolvendo os algoritmos de ordenação conhecidos como InsertionSort e MergeSort, bem como o algoritmo híbrido que utiliza ambos, denominado TimSort. O algoritmo que leva o nome de seu idealizador (Tim Peters) foi criado em 2002 para ser usado na linguagem de programação Python, e tem sido o algoritmo de ordenação padrão de Python desde a versão 2.3. Ele atualmente é usado também para ordenar arrays em Java SE 7.

Nas seções seguintes, detalha-se o problema proposto, bem como detalhes dos algoritmos mencionados para solução do problema de ordenação. Posteriormente, para análise, as rotinas são comparadas em um cenário teórico em termos de complexidade e, para conferência em uma aplicação real, os tempos de execução são medidos e analisados para diferentes formatos de entrada.

2 Detalhamento do Problema Proposto

A proposta do trabalho aqui desenvolvido consiste em implementar os algoritmos em foco (InsertionSort, MergeSorte e, a partir da utilização das funções implementadas nestes, o algoritmo híbrido TimSort). Para testar o funcionamento das rotinas implementadas, algumas entradas a serem processadas devem ser geradas. A fim de explorar diferentes conjunturas, de forma que um algoritmo em particular não se beneficie do formato do conjunto a ser ordenado, os seguintes cenários são determinados para geração das entradas:

- Vetor gerado com elementos em posição completamente aleatória
- Vetor gerado com elementos em ordem crescente
- Vetor gerado com elementos em ordem decrescente

Para avaliar também como os algoritmos se comportam à medida que o tamanho da entrada cresce, os vetores gerados possuem tamanhos 32, 64, 1.024, 10.000, 100.000 e 1.000.000. Esse desempenho será medido em termos de tempo de processamento efetivo e também em número de operações relevantes dentro do algoritmo (como trocas de posição e comparação de maior/menor). Como a medida do tempo em um sistema operacional oscila devido à concorrência dos processos pela CPU, para cada um desses contextos serão feitas 10 execuções, com o intuito de se obter uma média mais confiável. Isso gera um total de 180 execuções (3 configurações de entrada x 6 tamanhos de entrada x 10 execuções) para cada algoritmo.

A partir dos resultados obtidos, será verificado o comportamento assintótico na prática condiz com a análise de complexidade teórica destes algoritmos. Esta tem por finalidade descrever o comportamento de determinada rotina à medida que o tamanho da entrada a ser processada cresce. Além disso, fornece uma análise dos casos de pior, melhor e médio desempenho de cada algoritmo, de forma a auxiliar a escolha de acordo com a característica dos conjuntos de dados que serão efetivamente ordenados na aplicação.

3 O Funcionamento dos Algoritmos

3.1 Insertion Sort

O Insertion Sort é um algoritmo de classificação baseado em comparação local. Nele, um subconjunto do vetor é mantido como ordenado, o qual vai crescendo à medida que o vetor é percorrido e cada elemento é devidamente posicionado. Nesse posicionamento o novo item em foco tem que encontrar seu lugar apropriado e então ser inserido. Dessa característica origina o seu nome Insertion Sort, ou ordenação por inserção.

Em um passo-a-passo, o algoritmo segue as etapas abaixo ao percorrer o vetor:

- Se for o primeiro elemento, ele já está classificado.
- Escolha o próximo elemento
- Compare com todos os elementos na sub-lista classificada
- Desloque todos os elementos na sub-lista classificada que são maiores que o valor a ser ordenado
- Insira o valor
- Repita até que a sub-lista classificada seja toda a lista

Um exemplo para ilustrar:

Parte-se de um vetor não ordenado

O Insertion Sort compara os dois primeiros elementos

Os números 14 e 33 já estão ordenados de forma crescente. Neste momento, o elemento 14 é a sub-lista ordenada.

33 e 27 são comparados

Neste caso, 33 está na posição incorreta

33 e 27 trocam de lugar. Ele também verifica todos os elementos da sub-lista ordenada. Aqui, vemos que a sub-lista classificada tem apenas um elemento 14, e 27 é maior que 14. Portanto, a sub-lista classificada permanece classificada após a troca.

Agora 14 e 27 são a sub-lista ordenada. 33 e 10 são comparados

Eles não estão na ordem crescente correta

Trocam de lugar

Entretanto, a sub-lista ainda não está ordenada corretamente, pois 10 ainda está após o 27

Então segue a troca

3.2 Merge Sort

O Merge Sort é um dos algoritmos de classificação mais eficientes. Ele tem como princípio de funcionamento a divisão e conquista. O algoritmo divide repetidamente uma

14 e 10 estão desordenados, então trocam de logar

Ao fim da terceira iteração, temos uma sub-lista ordenada de 4 itens. O processo segue iterativamente.

lista em várias sublistas até que cada sublista consista em um único elemento, e ao retornar mesclando essas sublistas ordenadamente, obtém-se a lista original de forma ordenada. Esse processo pode ser descrito pelos passos a seguir:

- Se houver apenas 1 elemnto na lista, está ordenado, retorne
- Divida a lista recursivamente ao meio até que não possa ser mais dividida
- Mescle as listas menores em uma nova lista ordenada

Para ilustrar o funcionamento, toma-se como exemplo de partida o vetor abaixo.

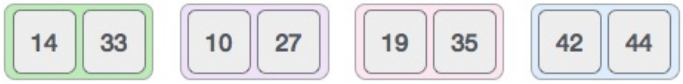
Sabemos que o Merge Sort primeiro divide todo o vetor iterativamente em metades iguais, até que os elementos singulares sejam alcançados. Vemos aqui que um vetor de 8 itens é dividida em dois vetores de tamanho 4.

Isso não altera a seqüência dos itens. Agora dividimos esses dois vetores em metades.

Nós dividimos ainda mais esses vetores e alcançamos o valor único que não pode mais ser dividido.

Agora, nós os combinamos exatamente da mesma maneira como foram divididos. Observe as cores dadas a essas listas.

Primeiro comparamos o elemento de cada lista e, em seguida, os combinamos em outra lista de maneira ordenada. Vemos que 14 e 33 estão em posições ordenadas. Comparamos 27 e 10 e na lista de destino de 2 valores colocamos 10 primeiro, seguido por 27. Alteramos a ordem de 19 e 35, enquanto 42 e 44 são colocados sequencialmente.



Na próxima iteração da fase de mesclagem, dois índices iteram sobre os conjuntos, de forma que se compara o menor elemento corrente de um conjunto com o menor elemento corrente do outro. Dessa forma, ao retirar os elementos um a um, de forma ordenada, obtém-se uma nova sub-lista ordenada.

Após a última mesclagem, a lista original encontra-se ordenada

3.3 Tim Sort

O Tim Sort foi implementado pela primeira vez em 2002 por Tim Peters para uso em Python. Supostamente, veio do entendimento de que a maioria dos algoritmos de classificação nascem em salas de aula e não são projetados para uso prático em dados do mundo real. Tim Sort tira proveito de padrões comuns em dados e utiliza uma combinação de Merge Sort e Insertion Sort junto com alguma lógica interna para otimizar a manipulação de dados em grande escala.

A chave para entender a implementação de Tim Sort é entender o conceito de *runs*. O Tim Sort aproveita os dados pré-classificados que ocorrem naturalmente a seu favor. Por pré-ordenado, simplesmente queremos dizer que os elementos sequenciais estão todos aumentando ou diminuindo (não nos importamos com quais). Em um passo a passo, o algoritmo pode ser descrito da seguinte maneira:

- Estabeleça um tamanho de minrun que seja uma potência de 2 (geralmente 32, nunca mais que 64 ou seu Insertion Sort perderá eficiência)

- Encontrar uma run, uma sub-lista que se estenda enquanto os elementos estiverem ordenados, mas que tenha, pelo menos a quantidade minrun de elementos.
- Se a run não tiver a quantidade mínima do minrun, usar o Insertion Sort para pegar os itens subsequentes ou anteriores e inseri-los na run até que tenha o tamanho mínimo correto.
- Repita até que todo o vetor seja dividido em subseções classificadas.
- Usar o merge do Merge Sort para unir os vetores ordenados.

3.4 Análise de Complexidade

A análise de complexidade do algoritmo é projetada para comparar dois algoritmos em um nível teórico - ignorando detalhes de baixo nível, como a linguagem de programação, o hardware em que o algoritmo é executado ou o conjunto de instruções de uma determinada CPU. O objetivo é comparar algoritmos em termos das operações relevantes que são executadas na rotina. Um algoritmo ruim escrito em uma linguagem de programação de baixo nível, como assembly, pode executar muito mais rápido do que um bom algoritmo escrito em uma linguagem de programação de alto nível, como Python ou Ruby. Portanto, a análise de complexidade provê um parâmetro do que realmente é um "algoritmo melhor".

Essa análise é feita em 3 cenários: melhor caso, pior caso e caso médio. A ocorrência de um cenário particular depende do formato da entrada a ser processada. No caso do Insertion Sort, por exemplo, o pior caso se dá quando a entrada está em ordem decrescente. Esse cenário faz com que o algoritmo percorra, no *loop* interno, todas as posições possíveis para fazer a inserção do elemento. Como esse laço de repetição está dentro de uma iteração que percorre os N elementos do vetor, é dito que nesse pior caso a complexidade é $O(n^2)$. A tabela abaixo exibe a complexidade dos algoritmos analisados e de antemão provê uma referência do seu comportamento à medida que o tamanho da entrada crescer durante a execução.

	Melhor Caso	Caso Médio	Pior Caso
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)
Mergesort	Ω(n log(n))	θ(n log(n))	O(n log(n))
Timsort	Ω(n)	θ(n log(n))	O(n log(n))

Análise de complexidade dos algoritmos em estudo

4 Implementação dos Algoritmos

Para implementar as rotinas descritas na seção anterior, optou-se pela linguagem python, com características de maior flexibilidade e menor verbosidade. Na própria linguagem foi implementado também algumas funções para auxiliarem na geração dos elementos nos formatos requisitados, no arquivo *gerar_entradas.py*.

A função para criação de listas para entrada recebe como parâmetro, além do número de elementos a ordenação. O parâmetro "ASC"gera elementos ordenados crescentemente, "DESC"decrescentemente e na ausência do parâmetro, os elementos são gerados aleatoriamente. No script de execução, os mesmos elementos gerados são passados para execução e medição pelos 3 algoritmos, de forma que o arquivo de entrada é salvo para conferência apenas.

```
1  # -*- coding: utf-8 -*-
2  import random
3  import sys
4
5
6  def criarListas(num_elementos, ordenacao):
7      elementos = []
8
9      for x in range(num_elementos):
10         tmp = random.randint(0, num_elementos)
11         elementos.append(tmp)
12
13     if ordenacao=="ASC":
14         elementos.sort()
15     elif ordenacao=="DESC":
16         elementos.sort(reverse=True)
17
18     salvarArquivo(elementos, num_elementos, ordenacao)
19
20     return elementos
21
22
23 def salvarArquivo(arr, size, ordenacao = ''):
24     filename = "n"+str(size) + ordenacao
25     with open("inputs/"+filename, 'w') as f:
26         for item in arr:
27             f.write("%s\n" % item)
```

Funções para geração dos elementos e armazenamento em arquivo

No arquivo *sort.py*, a função principal recebe dois parâmetros da linha de comando: o número de elementos a serem testados e a ordem (que segue os mesmos critérios descritos na geração). Para exemplificar, para fazer uma execução para 10 mil elementos ordenados crescentemente, a execução seria: *python sort.py 10000 ASC*. A mesma entrada é processada pelos 3 algoritmos, que têm seu tempo de execução e número de operações individualmente calculados. Os dados são entregues ao arquivo de saída com nome padronizado. Os dados da execução do Tim Sort, por exemplo, são adicionados ao arquivo *tim n10000ASC*.

O primeiro algoritmo a ser implementado foi o Insertion Sort, de funcionamento mais intuitivo e que requer poucas linhas de código. A função recebe, além do vetor de

```
def analiseAlgoritmo(elementos, algorithm):

    global operacoes
    operacoes = 0
    inicio = time.time()


    if algorithm=="insertion":
        ordenacao = insertionSort(elementos,0,len(elementos)-1)
    elif algorithm=="merge":
        ordenacao = mergeSort(elementos,0,len(elementos)-1)
    elif algorithm=="tim":
        ordenacao = timSort(elementos)

    tempo_segundos = time.time() - inicio

    print algorithm
    print "tempo: "+str(tempo_segundos)+"s | "+str(operacoes)+" comparacões"
    print "-----------------------------"

    filename="n"+''.join(sys.argv[1:])

    with open("outputs/"+algorithm+" "+filename, "a") as f:
        f.write("%ss\t%sc\n" % (tempo_segundos, operacoes))


try:

    try:
        ordenacao = sys.argv[2]
    except IndexError:
        ordenacao = ""

    elementos = gerar_entradas.criarListas(int(sys.argv[1]), ordenacao)
    analiseAlgoritmo(elementos[:], "insertion")
    analiseAlgoritmo(elementos[:], "merge")
    analiseAlgoritmo(elementos[:], "tim")

except:
    print "Parâmetros inválidos ou arquivo de amostras não encontrado."
```

Função principal para execução e análise dos algoritmos

elementos a ser ordenado, um índice de início e outro de fim, que serão utilizados posteriormente nas sub-ordenações do Tim Sort. Em uma execução comum, esses índices são 0, e tamanho do vetor - 1, respectivamente. O laço mais externo percorre o vetor apontando o elemento corrente a ser posicionado, enquanto o laço mais interno vai retornando pela sub-lista ordenada, até identificar o posicionamento do elemento, de forma que a sub-lista se mantenha ordenada.

```
def insertionSort(arr, left, right):
    for i in range(left + 1, right + 1):
        j = i
        #O indice j volta pela sub-lista ordenada,
        #buscando a posição onde o elemento deve ser
        #inserido para que a sub-lista mantenha-se ordenada
        while j > left and arr[j] < arr[j - 1]:
            operacoes+=1
            arr[j], arr[j - 1] = arr[j - 1], arr[j]
            j -= 1

    return arr
```

Função com a implementação do Insertion Sort

A implementação do Merge Sort é subdividida em 2 funções. A função principal, *mergeSort*, é responsável por subdividir recursivamente o vetor até que se obtenha um

elemento único, que por definição já está ordenado. A função merge, faz o processo de volta, onde os elementos são mesclados em sub-listas ordenadas.

O processo de ordenação em si se dá na iteração em que o índice i itera sobre a lista da esquerda e j itera sobre a lista da direita. Partindo do princípio de que cada sub-lista, partindo do elemento único, está ordenada, os índices iteram de forma que se posicionam no menor elemento corrente de cada sub-lista, de forma a compará-los e remover sempre o menor elemento entre eles, mantendo a lista mesclada em ordem.

O Tim Sort, por sua vez, combina os algoritmos implementados anteriormente em uma estratégia que utiliza a vantagem do Insertion Sort para elementos que eventualmente estejam ordenados em sub-conjuntos do vetor principal, sem cair no seu pior caso. O vetor principal é dividido nos subconjuntos denominados *runs*, que serão ordenados pelo Insertion Sort.

O tamanho desses subconjuntos, segundo princípios e análises do autor, performam bem variando entre 32 a 65 elementos. Para tirar vantagem do fato de que o Merge Sort funciona melhor com conjuntos que têm aproximadamente o mesmo tamanho e sejam base de 2, o tamanho foi eleito em 32 elementos. Após ter os sub-conjuntos ordenados pelo Insertion Sort, o Tim Sort mescla os elementos dessas sub-listas utilizando a função *merge* previamente implementada, até obter a lista original completamente ordenada.

```python
def merge(lista, i_esquerda, i_meio, i_direita):

    global operacoes

    #indices para subdividir a lista
    index1 = i_meio - i_esquerda + 1
    index2 = i_direita - i_meio

    #inicializando arrays temporários
    temp_esquerda = [0] * (index1)
    temp_direita = [0] * (index2)

    #copiando os dados da lista pros arrays
    for i in range(0 , index1):
        operacoes+=1
        temp_esquerda[i] = lista[i_esquerda + i]
    for j in range(0 , index2):
        operacoes+=1
        temp_direita[j] = lista[i_meio + 1 + j]


    #os indices i e j iteram pelos arrays subdivididos, enquanto k itera pela lista
    i, j, k = 0, 0, i_esquerda

    #merge
    while i < index1 and j < index2 :
        operacoes+=1
        if temp_esquerda[i] <= temp_direita[j]:
            lista[k] = temp_esquerda[i]
            i += 1
        else:
            lista[k] = temp_direita[j]
            j += 1
        k += 1


    #quando a comparação entre as duas sublistas acabar, copiar elementos restantes
    while i < index1:
        lista[k] = temp_esquerda[i]
        i += 1
        k += 1
    while j < index2:
        lista[k] = temp_direita[j]
        j += 1
        k += 1


def mergeSort(lista,i_esquerda,i_direita):

    global operacoes

    if i_esquerda < i_direita:
        operacoes+=1
        meio = (i_esquerda+i_direita)/2
        mergeSort(lista, i_esquerda, meio)
        mergeSort(lista, meio+1, i_direita)
        merge(lista, i_esquerda, meio, i_direita)
```

Funções implementadas para o Merge Sort

```python
def timSort(arr):
    global operacoes
    n = len(arr)
    minRun = 32 #Tamanho mínimo do run, segundo recomendações do autor

    #Ordenando os subarrays individuais de tamanho minRun
    for inicio in range(0, n, minRun):
        fim = min(inicio + minRun - 1, n - 1)
        insertionSort(arr, inicio, fim)

    # Começando a mesclagem a partir do tamanho mínimo da run (32)
    tamanho = minRun
    while tamanho < n:

        # escolhendo o ponto de partida do array à esquerda
        # as sub-listas que serão mescladas são arr[esuerda..esquerda+tamanho-1]
        # e arr[esquerda+tamanho, esquerda+2*tamanho-1]
        # depois de mesclar, fazemos o salto em 2*tamanho
        for esquerda in range(0, n, 2 * tamanho):

            # Encontrando o ponto de divisão entre as sub-listas
            meio = min(n - 1, esquerda + tamanho - 1)
            direita = min((esquerda + 2 * tamanho - 1), (n - 1))

            # Mesclando as sublistas arr[esquerda..meio] e arr[meio+1..direita]
            merge(arr, esquerda, meio, direita)

        tamanho = 2 * tamanho
    return arr
```

Implementação do Tim Sort

5 Apresentação e análise dos Resultados

Para medição dos tempos e comparações das implementações demonstradas na seção anterior, foi utilizado um computador pessoal de processador não-dedicado, modelo Intel® Core i7-7500U (3M Cache, 2.7 GHz até 3.5 GHz com Max Turbo). Os algoritmos foram executados no sistema operacional Ubuntu 18.04 e os resultados adicionados aos final dos respectivos arquivos na pasta outputs.

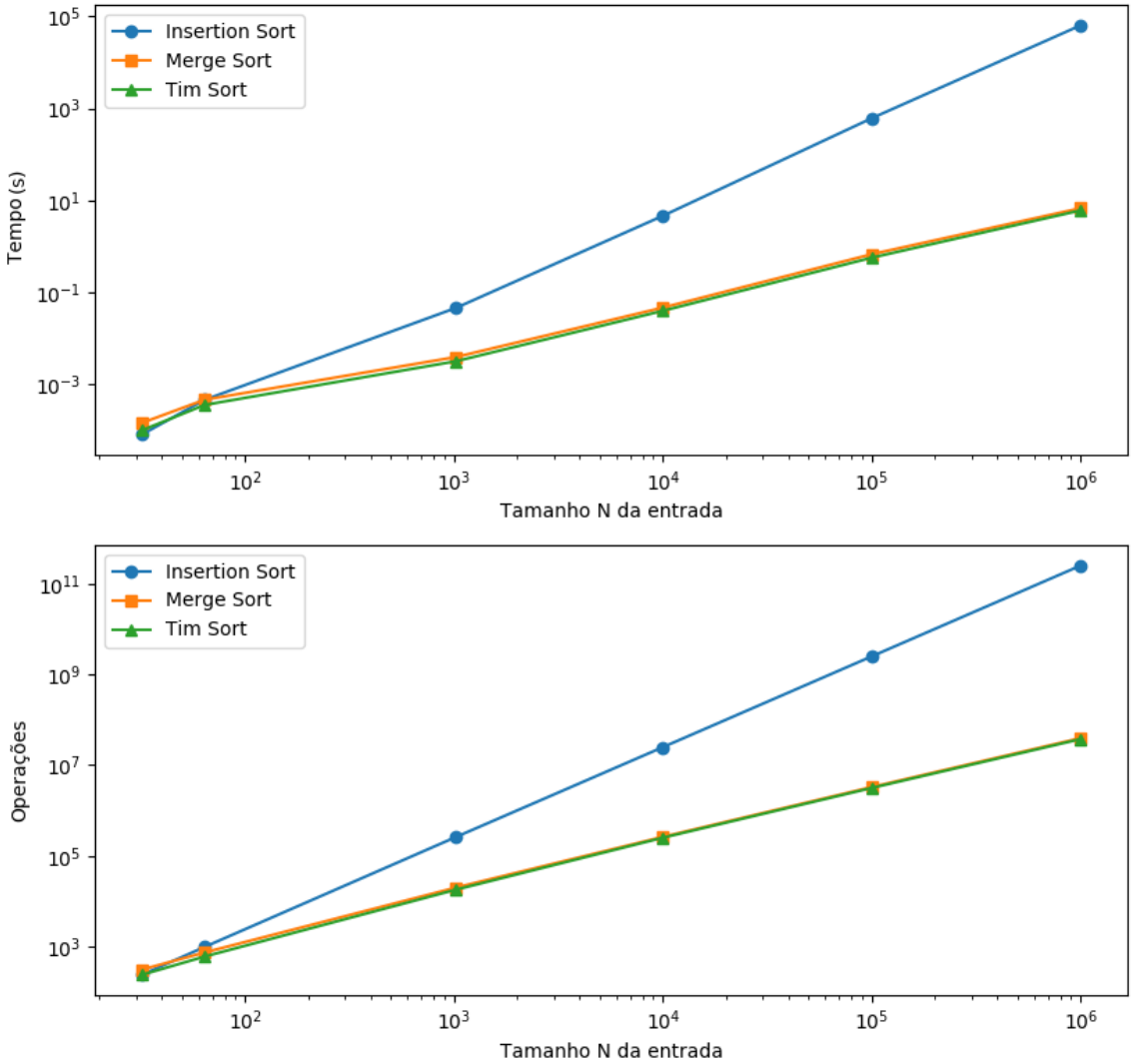
Os tempos foram medidos tomando como referência somente a função de ordenação, de forma a evitar influência de possíveis *overheads* de geração, entrada ou saída de dados. Alguns cenários, como Insertion Sort, para 1 milhão de entradas, não foram executados em 10 repetições por questões de tempo hábil, já que esses cenários demandavam muitas horas de execução.

Os gráficos para visualização dos dados gerados foram utilizados com o auxílio da biblioteca *matplotlib* do python. O script para leitura e plotagem dos dados se encontra disponível junto ao código fonte. O mesmo pode ser executado no formato *python charts.py |ASC|DESC||*, apenas com o parâmetro da ordenação a ser considerado, seguindo o mesmo padrão anterior. O código faz a leitura dos arquivos emitidos na execução, fazendo a média dos dados nas linhas presentes.

O primeiro resultado apresentado para os 3 algoritmos é dado pelas entradas em que os elementos estavam arranjados aleatoriamente. Neste caso, conforme mostra o gráfico, o algoritmo do Insertion Sort tende a fazer um elevado número de trocas e consequentemente se mostra mais lento que o Merge Sort e Tim Sort, que na escala visual se mostram bem próximos principalmente devido à diferença de ordem de seus resultados com o Insertion Sort. Mesmo nos piores cenários, os dois últimos não chegaram a demandar 10 segundos para execução, portanto, serão analisados de forma isolada posteriormente.

No segundo cenário, todavia, é o caso em que o Insertion Sort tem seu melhor caso (O(n)) e melhor performance dentre os 3 algoritmos. Como todos os elementos estão em ordem, na prática, o Insertion Sort faz apenas uma varredura no vetor verificando e nenhuma troca é efetuada. O Tim Sort por sua vez tende a performar melhor do que o Merge Sort, já que utiliza o Insertion Sort para ordenação das sub-listas e, portanto, esse cenário também o favorece.

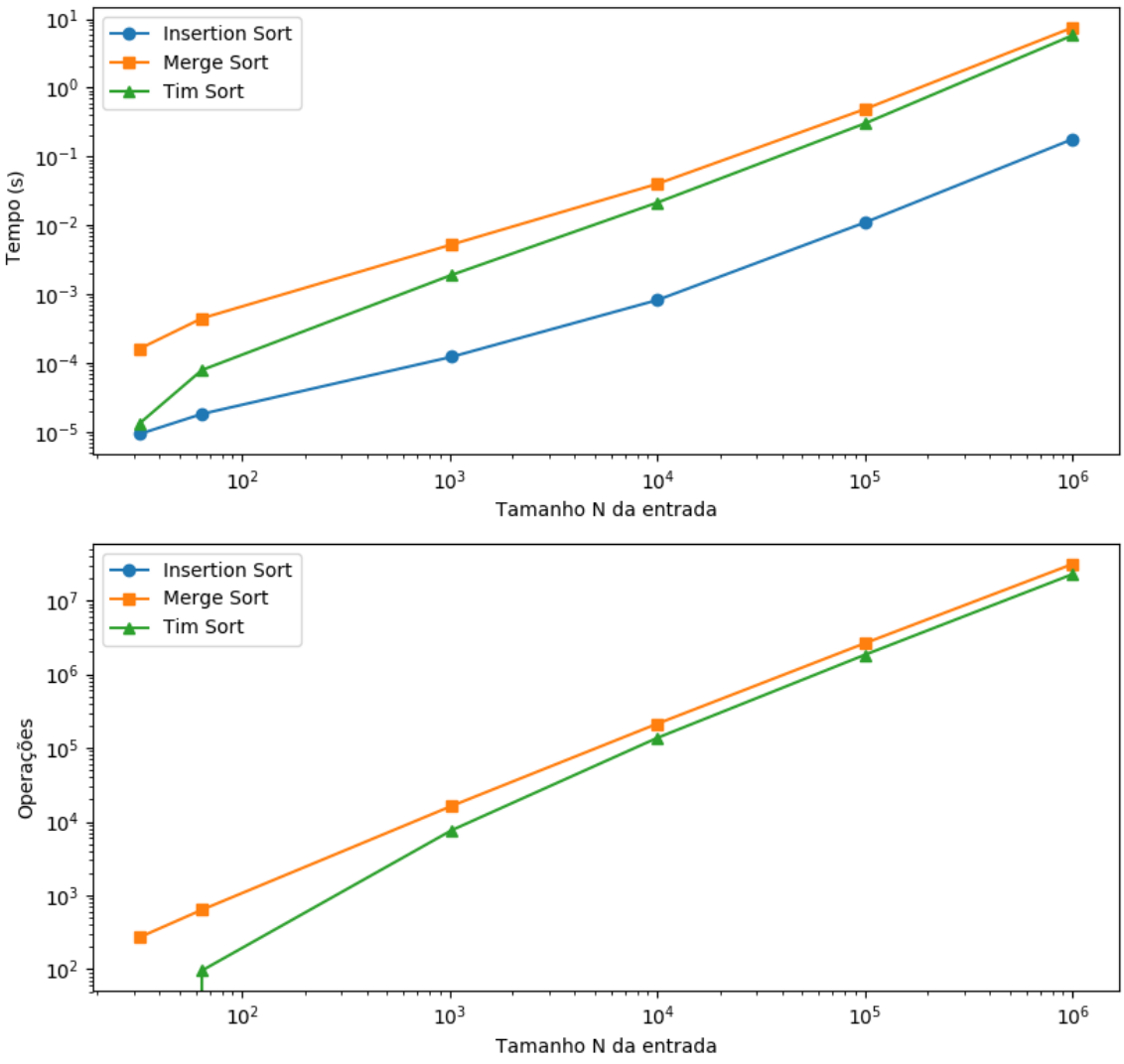
Na última configuração de entrada, onde os elementos estão ordenados decrescentemente, o Insertion Sort encontra seu pior caso, $O(n^2)$. Um detalhe importante a ser observado sobre o Tim Sort é que, apesar de utilizar o Insertion Sort, devido à sua estratégia de sub-divisão, o algoritmo ainda tem uma boa performance, bem próximo ao Merge Sort. Outra observação é que, apesar do tempo cerca de 2 vezes maior em relação à



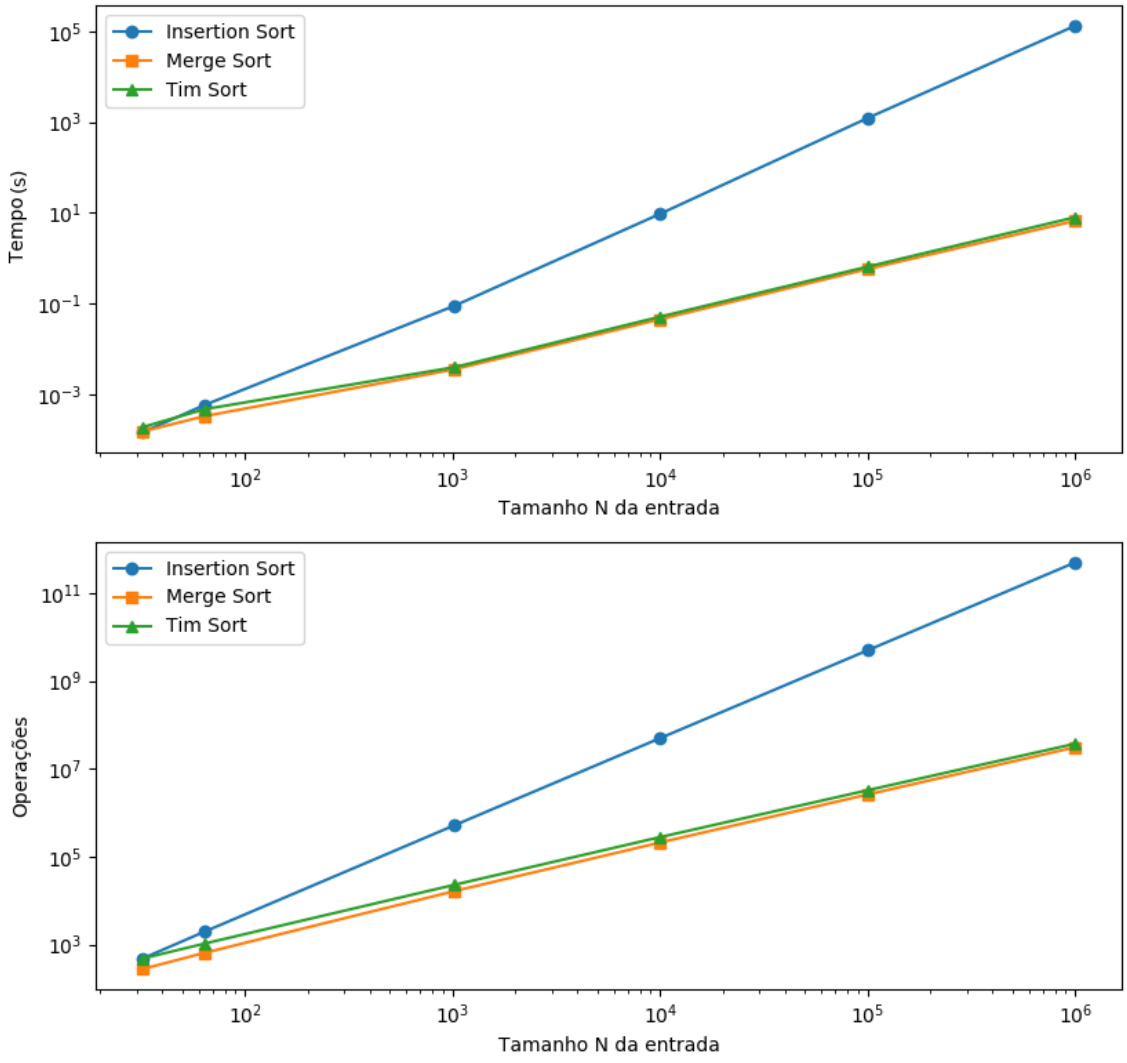
Resultados de execução para entrada aleatória

entrada aleatória, o traçado dos gráficos são bem semelhantes devido à escala logarítimica que se mostrou uma melhor escolha em razão do crescimento muito rápido da curva do Insertion Sort.

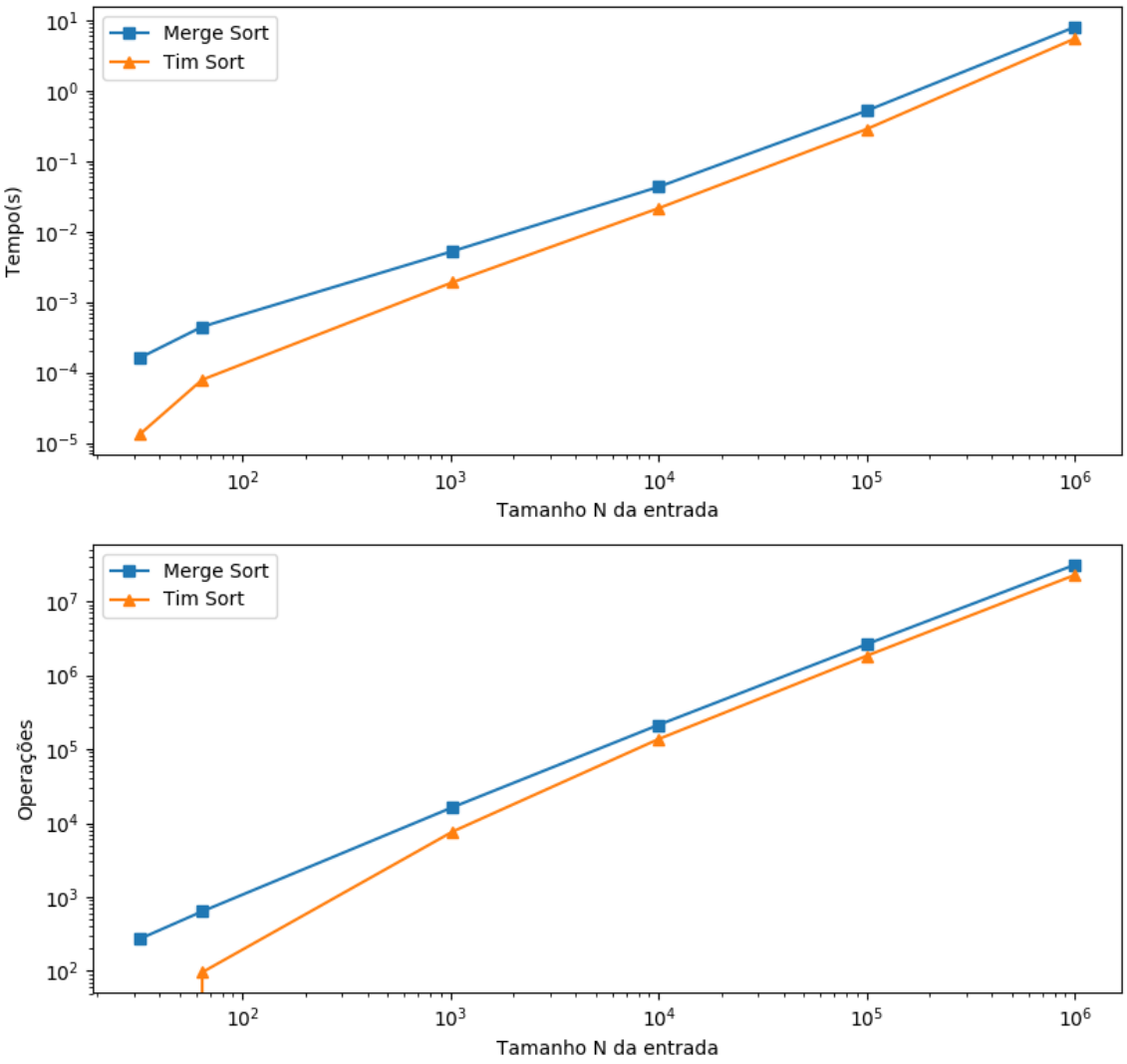
Por fim, para análise separada dos algoritmos Tim e Merge Sort, que demonstraram uma significativa melhor performance, são apresentados abaixo os gráficos para entradas crescentes e decrescentes, respectivamente. O baixo tempo de execução de ambos mesmo para entradas com muitos elementos, tornam a diferença significativamente pequena. Conforme observado anteriormente, o Tim Sort leva vantagem da configuração pré-ordenada crescentemente do vetor, mas, mesmo no cenário inverso, tem performance bem próxima do Merge Sort.



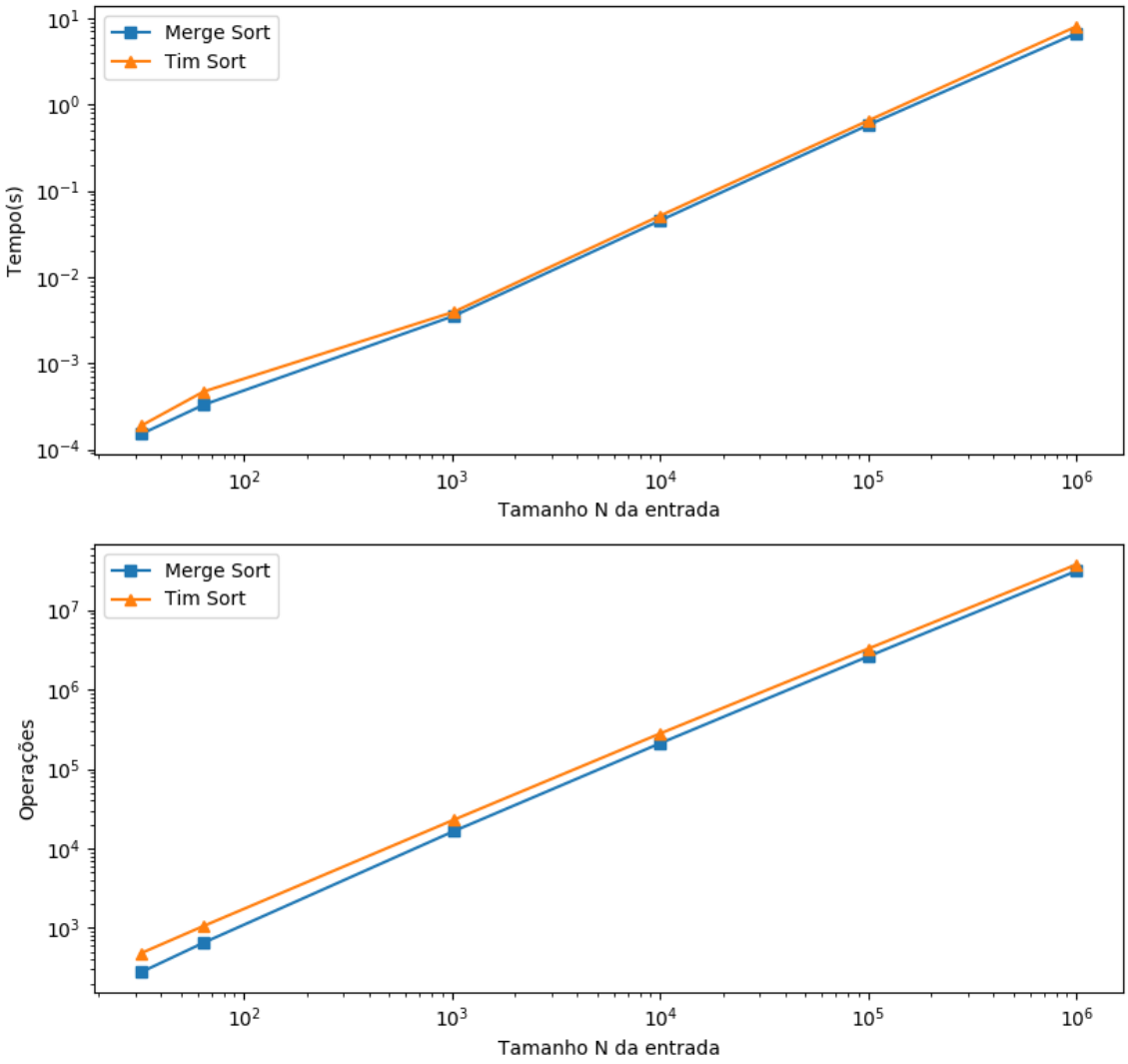
Resultados de execução para entrada ordenada crescentemente



Resultados de execução para entrada ordenada decrescentemente



Resultados de execução para entrada ordenada crescentemente



Resultados de execução para entrada ordenada decrescentemente

Conclusão

O Tim Sort, como algoritmo híbrido, tira vantagem do fato de que, em aplicações no mundo real, frequentemente são encontrados padrões nos quais sub-conjuntos de uma lista já estão ordenados. Se beneficiando dessa característica e fazendo uso das funções de ordenação do Insertion e Merge Sort para uma estratégia de divisão e conquista, o algoritmo se mostra muito eficiente aos cenários em que é submetido na prática e acabou se tornando o método padrão das linguagens Python e Java SE 7.

Para trabalhos futuros, sugerem-se algumas otimizações que visam melhorar ainda mais o desempenho do Tim Sort, como por exemplo a ordenação por inserção com busca binária e o merge com galopeamento binário. A implementação aqui demonstrada na linguagem python, ainda que menos verbosa e de mais fácil compreensão, é menos eficiente do que uma linguagem de mais baixo nível, como C. Essa transcrição também é sugerida como melhoria em termos de performance.

Por fim, o trabalho aqui proposto é também uma demonstração de análise e comparação de diferentes algoritmos para solução de um mesmo problema. A partir da análise teórica de complexidade das rotinas propostas, a verificação prática se dá para confirmar o comportamento assintótico dessas soluções em diferentes contextos à medida que são sobrecarregados com entradas de diversas escalas. Em um caso de uso real, esse processo vem apoiar a tomada de decisão de acordo com o cenário em questão.

Referências

Insertion Sort Algorithm. Disponível em: <https://www.tutorialspoint.com/data_structures_algorithms/insertion_sort_algorithm.htm>. Acesso em março de 2021.

Merge Sort Algorithm. Disponível em: <https://www.tutorialspoint.com/data_structures_algorithms/merge_sort_algorithm.htm >. Acesso em março de 2021.

Tim Sort Algorithm. Disponível em: <https://www.geeksforgeeks.org/timsort/>. Acesso em março de 2021.

An Introduction to Algorithm Complexity Analysis. Disponível em: <https://discrete.gr/complexity/>. Acesso em março de 2021.