Análise de Algoritmos - Atividade 1

Daniel Tatsch
Mestrado em Computação Aplicada
Univali - Universidade do Vale do Itajaí
Agosto de 2021

1 Introdução

Este trabalho tem como objetivo analisar experimentalmente a complexidade dos algoritmos de ordenamento (*array sorting algorithms*) Insertion Sort e Merge Sort. Para tal, considerou-se a média de tempo de 50 execuções de cada algoritmo, para *arrays* de números inteiros gerados de forma aleatória e uniformemente distribuídos entre 0 e 999.

Os algoritmos e os testes foram desenvolvidos utilizando a linguagem de programação Python e podem ser acessados através da plataforma Github, juntamente com os resultados obtidos nos experimentos

2 Exercício 1

2.1 Insertion Sort

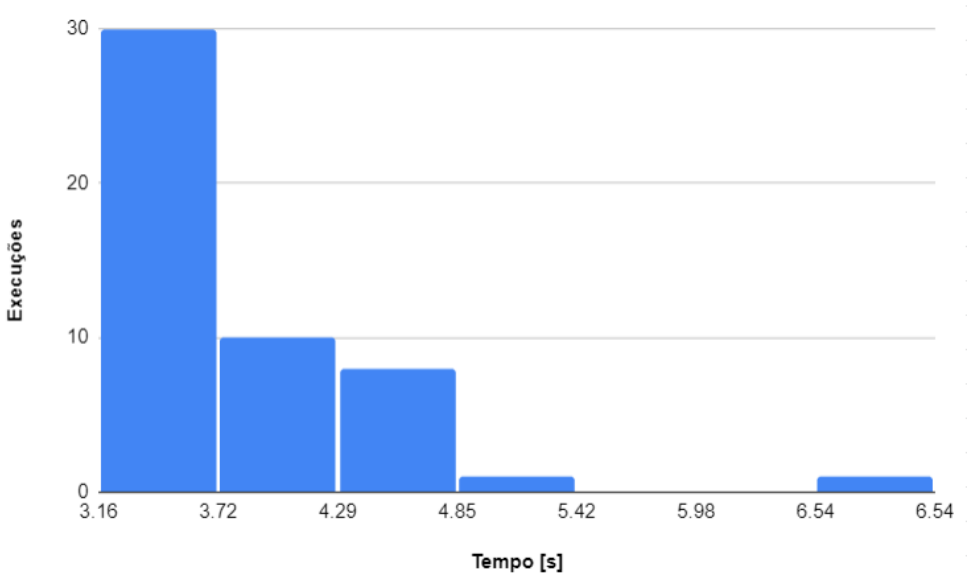
A execução do algoritmo Insertion Sort funciona com base na inserção ordenada de um *array*. A lógica de ordenamento consiste em comparar iterativamente todos os itens do *array* com os dados das posições anteriores, a partir do segundo elemento.

O tempo de execução do algoritmo foi analisado com base em um *array* com 10000 números inteiros aleatórios distribuídos de forma uniforme entre 0 e 999. O tempo médio em segundos das 50 execuções, bem como o valor máximo e mínimo obtidos no experimentos estão destacados na Tabela 1. O histograma do tempo de execução do algoritmo pode ser observado na Figura 1.

Tabela 1 - Tempos de execução do algoritmo Insertion Sort (segundos)

Média	3,4744
Mínimo	3,1605
Máximo	6,5425

Figura 1 - Histograma do algoritmo Insertion Sort



Fonte: Desenvolvida pelo autor.

Também foi realizada uma análise para verificar o aumento do tempo de execução do algoritmo de acordo com a variação do tamanho dos dados de entrada. A Figura 2 apresenta o gráfico dessa relação. É possível observar que apesar da baixa resolução, a curva do tempo de execução apresenta um comportamento quadrático de acordo com o tamanho do *array* inserido no algoritmo.

Figura 2 - Tempo de execução do Insertion Sort de acordo com o tamanho da entrada

Fonte: Desenvolvida pelo autor.

2.2 Merge Sort

O algoritmo Merge Sort funciona de forma recursiva. Ele divide o *array* de entrada pela metade e ordena as subcadeias geradas chamando a própria função recursivamente. Quando as subcadeias de entrada da recursão forem de tamanho 1, um processo de intercalação é realizado. Nele, os subconjuntos de dados são ordenados e reagrupados até que o *array* de tamanho original seja reconstruído.

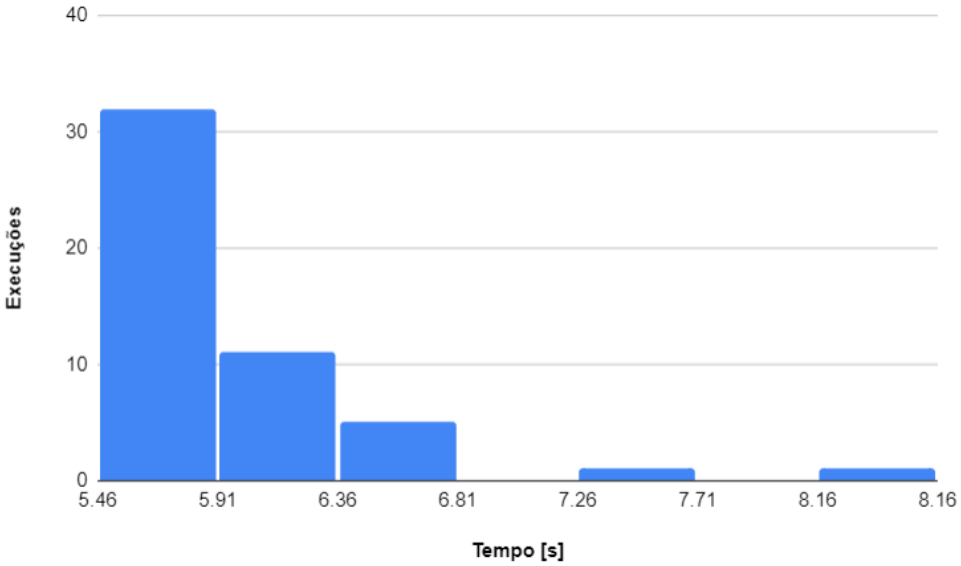
A primeira análise do Merge Sort foi realizada considerando *arrays* de mesmo tamanho dos utilizados na subseção 2.1. Ele apresentou um tempo de execução médio de 0,0384 segundos, sendo aproximadamente 100 vezes mais rápido que o algoritmo Insertion Sort.

A segunda análise considerou um aumento dos parâmetros de entrada do Merge Sort. O tamanho dos 50 *arrays* foi aumentado para 1000000, com dados aleatórios variando de 0 até 4999. O tempo de execução médio, mínimo e máximo pode ser observado na Tabela 2 e o histograma do algoritmo pode ser observado na Figura 3.

Tabela 2 - Tempos de execução do algoritmo Merge Sort (segundos)

Média	5,6843
Mínimo	5,4554
Máximo	8,1601

Figura 3 - Histograma do algoritmo Insertion Sort



Fonte: Desenvolvida pelo autor.

Assim como na Subseção 2.1, o aumento do tempo de execução do algoritmo Merge Sort também foi observado de acordo com a variação do tamanho do *array* aplicado. Os dados são apresentados na Figura 4, em que possível observar um crescimento similar ao logaritmo, conforme destacado nas informações disponíveis no site https://www.bigocheatsheet.com/ ($\Theta(n \log(n))$).

Figura 4 - Tempo de execução do Merge Sort de acordo com o tamanho da entrada

Fonte: Desenvolvida pelo autor.

3 Exercício 2

A segunda etapa da atividade consistiu em desenvolver um algoritmo híbrido de Insertion Sort e Merge Sort. Ele recebe como parâmetro a profundidade k da divisão do *array* de entrada, seguindo a lógica do Merge Sort e aplica o Insertion Sort nos subconjuntos criados. O nível de profundidade k varia de acordo com o tamanho N dos dados de entrada e pode ser definido de acordo com a Equação 1:

$$k = \lfloor \log_2(N) \rfloor \tag{1}$$

Os experimentos consistiram em realizar um laço de execução do algoritmo Merge Sort variando a profundidade em que o Insertion Sort seria inserido. Para cada nível foi extraída a média do tempo de execução de 50 operações de ordenação, considerando *arrays* de 10000 amostras ($k = 13$) e de valores aleatórios variando uniformemente de 0 a 999. A Tabela 3 apresenta a média dos tempos de execução do algoritmo híbrido de acordo com a variação do subnível em que o Insertion Sort foi aplicado.

Tabela 3 - Tempos de execução do algoritmo híbrido

k	Tempo [s]
1	1,5963
2	0,005577
3	0,007677
4	0,009418
5	0,0112
6	0,01282
7	0,01491
8	0,01655
9	0,01936
10	0,0213
11	0,02375
12	0,02786
13	0,031887

É possível observar através da Tabela 3, que o algoritmo híbrido foi em média, mais rápido com a aplicação do algoritmo de ordenação Insertion Sort em $k = 2$, ou seja, com 4 subcadeias do *array* original (cada uma com 2500 dados). Ainda é possível observar a similaridade da média dos tempos obtidos em $k = 13$ (execução completa por Merge Sort) com o valor apresentado na Subseção 2.2.

4 Conclusão

Este trabalho teve como objetivo analisar a complexidade de diferentes algoritmos de ordenamento de *arrays*. O tempo de execução dos algoritmos Insertion Sort e Merge Sort foi analisado de acordo com a variação do tamanho dos dados de entrada, gerados de forma aleatória.

Observou-se que o algoritmo Merge Sort se demonstrou mais eficiente que o Insertion Sort, de acordo com o aumento do *array* de entrada. Os experimentos práticos demonstraram que para 10000 dados aleatórios, o Merge Sort conseguiu sem média ser 100 vezes mais rápido no ordenamento.

O segundo exercício proposto possibilitou mesclar a implementação dos dois algoritmos, aplicando o Insertion Sort em diferentes subcadeias geradas pela lógica recursiva utilizada no Merge Sort. Foi possível observar que os fatores constantes do Insertion Sort permitiram que o algoritmo híbrido fosse mais rápido que a aplicação total do Merge Sort (até alcançar subcadeias de tamanho unitário), aplicando a inserção ordenada em partes do *array* original.