Benchmarking

Igor Souza Tavares
Universidade Cruzeiro do Sul
São Paulo-SP, Brasil
E-mail: igorsouza96@live.com

***Resumo*— Neste artigo é proposto a implementação de algoritmos de estrutura de dados, execução e análise de complexidade. Os tipos de algoritmos são de ordenação, divisão e conquista, backtraking e algoritmo guloso. Serão apresentados os resultados de diferentes execuções, metodologia, gráficos de crescimento e os códigos dos algoritmos implementados.**

Palavras-Chaves—Análise de Algoritmo, ordenação, divisão e conquista, complexidade de algoritmo, Python, benchmarking;

I. INTRODUÇÃO

Apresentarei a implementação de 4 tipos de algoritmos de estrutura de dados, analisando a complexidade de cada um dos algoritmos, com múltiplos testes de benchmarking utilizando de dados de diferentes tamanhos e executados numa mesma máquina, apresentando todas informação e dados comparativos, e inclusive o tempo de execução.

O conteúdo deste artigo é estruturado como a seguir:
No capítulo II serão apresentados os conceitos referentes aos algoritmos escolhidos como os nomes de cada um e sua respectiva complexidade assintótica. No capítulo III será apresentada a metodologia e descrição dos algoritmos escolhidos, explicando como as estruturas dos dados são executadas, a linguagem utilizada e as informações do computador onde foram executados os algoritmos. No capítulo IV será apresentado o resultado dos experimentos e analises dos algoritmos executados, utilizando de gráficos de crescimento para comparação e formulação da conclusão desta análise.

II. ALGORITMOS ESCOLHIDOS

Para ordenação a escolha foi o algoritmo Insertion Sort que tem complexidade $O(n^2)$, para divisão e conquista foi o Merge Sort com complexidade O(n log n), já para backtraking a escolha foi o Sudoku, que em uma escala de 3x3 tem complexidade $O(n^{n\ x\ n})$, já para algoritmos gulosos a escolha foi o troco mínimo que tem complexidade O(n).

III. METODOLOGIA

Cada algoritmo escolhido tem sua função, metodologia e complexidade distintas uns dos outros.

O algoritmo Insertion Sort tem a função de ordenar uma lista de dados inserindo item por item no local adequado para ordenação correta desta lista O método utilizado no algoritmo é o de analisar a lista item por item identificando os itens que estão desordenados e inserindo-os no lugar correto, ordenando um item de cada vez. O algoritmo Merge Sort também tem a função de ordenar uma lista de dados, porém utilizando outro método mais eficiente, com isso o algoritmo ordena a lista de uma maneira melhor e mais rápida. Com o método de divisão e conquista, dividindo recursivamente a lista de dados até que restem pequenas quantidades de dados para ordenação, tornando mais fácil e rápido esta função. O algoritmo do Sudoku preenche uma matriz de dados, completando os campos vazios de forma organizada e calculada matematicamente, sem repetir os valores na mesma linha, coluna ou regiões, solucionando o problema apresentado pelo jogo Sudoku. Utilizando o método de tentativa e erro para preencher corretamente todos os campos da matriz. Já o algoritmo de Troco Mínimo analisa um valor e calcula a menor quantidade de cédulas ou moedas necessárias para chegar ao valor apresentado. O algoritmo tem uma solução "gulosa", que sempre parte da função de pegar o maior valor apresentado como moeda de troco, para conseguir a menor quantidade de moedas possíveis para o troco.

A linguagem utilizada é Python que é de alto nível e orientada a objetos, que tem uma fácil compreensão e de código reduzido, facilitando o processo de desenvolvimento dos algoritmos.

O computador utilizado foi um notebook pessoal, com o processador Intel Core i7 (4º geração) de 2.60 GHz, memória RAM de 16GB DDR3, SSD de 240GB e sistema operacional Windows 10 de 64 bits.

IV. RESULTADOS

Após a implementação adequada dos algoritmos escolhidos na linguagem selecionada, os algoritmos foram executados com determinados valores para análise e comparação de tempo, utilizando de gráficos apontando a quantidade de dados da lista no eixo "X" e o tempo em segundos no eixo "Y".

Os valores para o tamanho das listas nos algoritmos de ordenação e de divisão e conquista foram de mil, dez mil, cem mil, um milhão, dez milhões e cem milhões. Para o algoritmo de Sudoku serão utilizadas matrizes de 3x3, 4x4, 5x5, 8x8 e 16x16. Já para o algoritmo de Troco Mínimo serão utilizados valores de moedas de 1, 2, 5, 10, 20, 50, 100, 500 e 1000, para valores de troco que serão descritos junto aos gráficos que serão apresentados.

Insertion Sort

De início foi implementado o algoritmo Insertion Sort em linguagem Python, de uma maneira mais simples para analisarmos o tempo de execução com valores pré-definidos.

Para o melhor caso foram utilizadas listas com valores já ordenados para podermos analisar o comportamento do algoritmo em realizar a ordenação sem necessitar de alteração nas listas.

O algoritmo que tem complexidade O(n) no melhor caso se provou que realmente respeita sua complexidade assintótica em tempo de execução. Podemos verificar na figura do gráfico a baixo que o crescimento do tempo de execução acompanha o aumento do valor inserido para o cálculo.

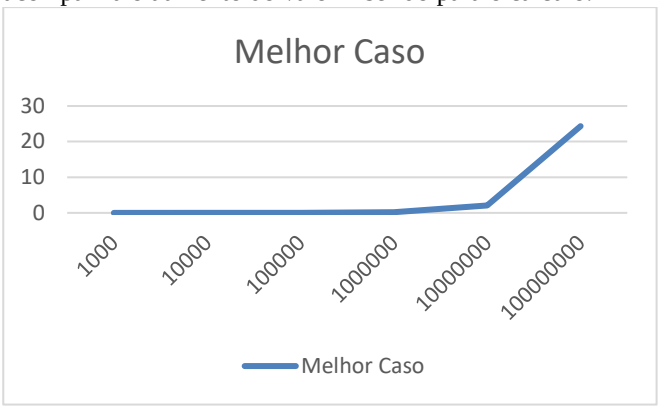

Figura 1 - Gráfico do melhor caso do Insertion Sort

Porém o tempo é muito inferior no melhor se comparado com os mesmos valores no pior caso, conforme iremos mostrar. Para o pior caso foram utilizadas as mesmas listas com os mesmos valores, porém com os dados ordenados inversamente, ou seja, as listas estavam totalmente desordenadas, fazendo com que o algoritmo execute todas as trocas possíveis, utilizando muito mais tempo para execução deste mesmo código. No pior caso o algoritmo Insertion Sort tem complexidade $O(n^2)$. Devido o tempo necessário para execução do algoritmo com valores acima de 1 milhão levar mais de 1 hora, para estes valores mais altos foram inseridos no gráfico os valores de mais de 3600 segundos, que representa que o algoritmo precisou de mais de 1 hora para o termino da execução no pior caso.

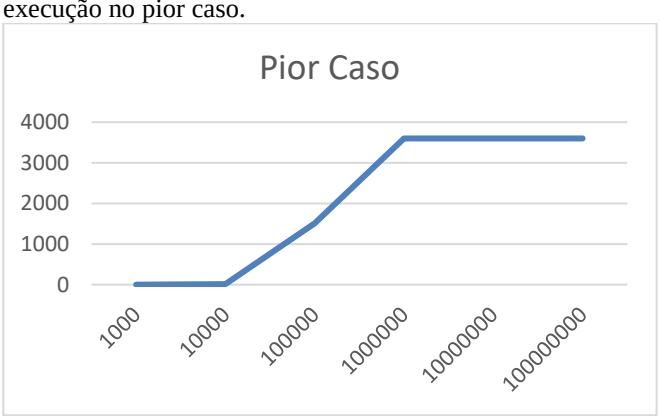

Figura 2 - Gráfico do pior caso do Insertion Sort

Podemos ver em a comparação dos dois casos e entender que o pior caso tem um crescimento exponencial, algo que em grande escala requer muito mais tempo de execução.

Merge Sort

Para o algoritmo Merge Sort também foram utilizadas as mesmas listas e utilizando dos mesmos métodos, para fins de comparação. Porém com este algoritmo o método de execução se diferencia, pois este utiliza o método de divisão e conquista que consiste em dividir o problema em pequenos problemas para soluciona-los mais facilmente, tornando a solução final mais rápida e menos complexa.

No melhor caso com listas de valores ordenados o algoritmo tem complexidade O(n log n), que é melhor que o algoritmo anterior, porém verificamos que em casos de valores baixos o algoritmo se comporta de maneira pior como mostra o gráfico abaixo.

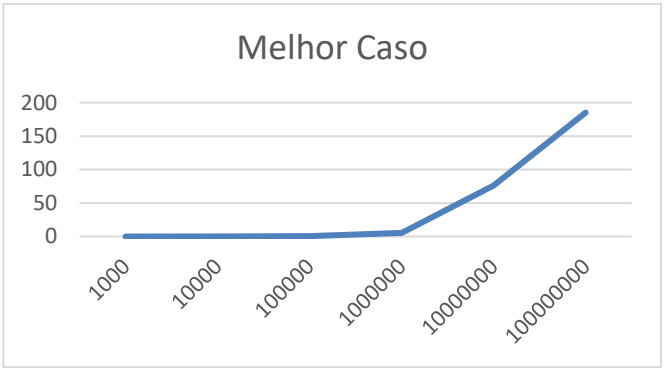

Figura 3 - Gráfico do melhor caso do Merde Sort

Pelo fato do algoritmo executar a divisão da lista para sua verificação de ordenação, ele acaba levando mais tempo que o algoritmo Insertion Sort, que apenas compara os itens que altera de posição os que estão fora de ordem. Com isso em pequenas quantidades de dados no melhor caso ele tem crescimento assintótico menor que o Insertion Sort.

Já no pior caso o cenário muda, pois o método utilizado pelo Merge Sort é mais eficiente, tornando sua execução com listas de grandes valores muito mais rápido, como podemos ver no gráfico abaixo.

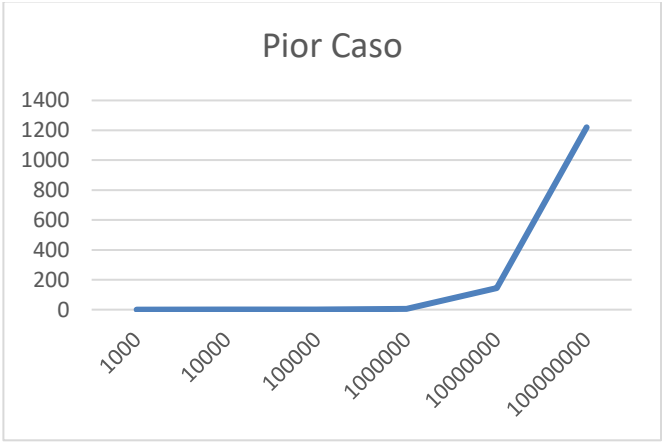

Figura 4 - Gráfico do melhor caso do Merge Sort

Comparando com os mesmos valores utilizados no algoritmo anterior, o Merge Sort se mostrou muito mais eficiente, executando o código com listas de valores acima de 1 milhão em bem menos de 1 hora, podendo assim ser inclusos os tempos até da lista com 100 milhões que levou em média cerca de 1220 segundos para execução completa.

Sudoku

O algoritmo de solução do jogo Sudoku é um pouco mais complexo sua execução e compreensão, devido a quantidade de informação necessária para comparação, e também por se tratar de um algoritmo de tentativa e erro, que pode executar os códigos mais de uma vez.

Para execução foram consideradas diversos tamanhos de matrizes e cada matriz foi preenchida 70% dos campos previamente, fazendo com que o algoritmo preencha os outros 30% dos campos vazios.

Podemos verificar que o tempo de execução do algoritmo é bem pequeno, e não há muita diferença para diferença entre os valores analisados, mantendo o tempo de execução a baixo de 1 segundo, conforme o gráfico aponta abaixo.

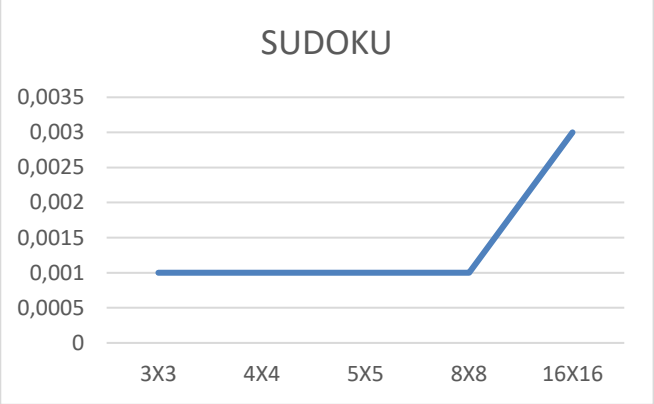

Figura 5 - Gráfico do Algoritmo Sudoku

Com essa implementação podemos verificar que ficou um algoritmo limpo e bastante rápido, solucionando o problema do jogo Sudoku muito rápido.

Troco Mínimo

O algoritmo de Troco Mínimo recebe um valor e uma lista de moedas predefinidas para o cálculo do valor para o troco, e quanto menor a quantidade de moedas é melhor para o resultado final. É conhecido por algoritmo guloso pois neste caso ele sempre pega a maior moeda possível para o troco e com isso obtém uma menor quantidade de moedas possíveis.

Após a implementação, execução e análise do algoritmo, foram informados os seguintes valores para troco:

Troco
70,00
20,00
60,00
68,00
85,00
104,00
217,00
99,00
201,00
369,00

E as moedas disponíveis foram:

Moedas
1
2
5
10
20
50
100
500
1000

Com isso o algoritmo foi executado para cada um dos valores de troco, porém o tempo de execução foi o mesmo que ficou a baixo de 1 segundo, não importando o valor indicado ao algoritmo, em todos os testes o tempo de execução não sofreu alterações, como podemos observar no gráfico abaixo.

Portanto se trata de um algoritmo muito eficiente e rápido, exigindo um tempo praticamente insignificante para sua execução, independentemente do valor apresentado.

V. CONCLUSÃO

Ao final deste experimento concluímos que o algoritmo Insertion Sort é o algoritmo que tem o maior tempo de execução e a maior complexidade assintótica também, vimos que o Merge Sort é uma melhor opção de algoritmo de ordenação que utiliza a metodologia de divisão e conquista, conseguindo um melhor desempenho em tempo de execução. Já o algoritmo de Sudoku percebemos que tem uma ótima execução, ficando com um tempo muito baixo se comparado com os algoritmos anteriores, e o de Troco Mínimo também, mantando um tempo a baixo de 1 segundo, sendo considerado o algoritmo mais rápido do que os outros analisados neste artigo.

Analisando os tempos de execução considerados na teoria, para os algoritmos Insertion Sort, Merge Sort e Troco Mínimo coincidem com os testes na prática, porém para o código implementado do algoritmo de Sudoku, que tem complexidade O($n^{n\,x\,n}$), na pratica esteve abaixo deste nível de complexidade, ficando com complexidade O(n) nos testes.

VI. REFERÊNCIAS

[1] Algoritmo Insertion Sort em Python. Disponível em: <https://www.geeksforgeeks.org/python-program-for-insertion-sort/>. Acessado em: 27/10/2020.

[2] Stackoverflow em Português. Disponível em: <https://pt.stackoverflow.com/>. Acessado em: 03/11/2020.

[3] Algoritmo Merge Sort em Python. Disponível em: <https://panda.ime.usp.br/pythonds/static/pythonds_pt/05-OrdenacaoBusca/OMergeSort.html>. Acessado em: 03/11/2020.

[4] Sudoku Backtraking-7. Disponível em: <https://www.geeksforgeeks.org/sudoku-backtracking-7/>. Acessado em: 08/11/2020.

[5] Algoritmos Gulosos – Troco Mínimo. Disponível em: <https://pt.slideshare.net/jackocap/anlise-de-algoritmos-algoritmos-gulosos-troco-mnimo>. Acessado em 08/11/2020.

APENDICE A – CÓDIGO ALGORITMO INSERTION SORT

```
import time

def insertionSort(lista):
	for i in range(1, len(lista)):
		chave = lista[i]
		j = i - 1
		while j >= 0 and chave < lista[j]:
			lista[j+1] = lista[j]
			j -= 1
		lista[j+1] = chave


#EXECUTANDO
lista = list(range(0, 10000))
lista.reverse()
#print("Lista: ", lista)

inicio = time.time()
insertionSort(lista)
fim = time.time()
#print("\nLista organizada: ", lista)
print("\nTempo: ",fim - inicio)
```

APENDICE B – CÓDIGO ALGORITMO MERGE SORT

```
import time

def mergeSort(lista):
		if len(lista)>1:
			meio = len(lista)//2
			left = lista[:meio]
			right = lista[meio:]

			mergeSort(left)
			mergeSort(right)

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

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

			while j < len(right):
				lista[k]=right[j]
				j=j+1
				k=k+1


#EXECUTANDO
lista = list(range(0, 10000))
lista.reverse()
#print("Lista: ",lista)

inicio = time.time()
mergeSort(lista)
fim = time.time()
#print("\nLista organizada: ", lista)
print("\nTempo: ", fim - inicio)
```

APENDICE C – CÓDIGO ALGORITMO SUDOKU

```
import time

inicio = time.time()
def print_grid(arr):
	for i in range(16):
		if i in [4, 8, 12]:
			print("------------+-------------+-------------+-------------")
		for j in range(16):
			if(arr[i][j] <10):
				print(arr[i][j],end='  ')
			else:
				print(arr[i][j],end=' ')
			if j in [3, 7, 11]: print("|",end=' ')

		print()

def find_empty_location(arr, l):
	for row in range(16):
		for col in range(16):
			if(arr[row][col] == 0):
				l[0] = row
				l[1] = col
				return True
	return False


def used_in_row(arr, row, num):
	for i in range(16):
		if(arr[row][i] == num):
			return True
	return False


def used_in_col(arr, col, num):
	for i in range(16):
		if(arr[i][col] == num):
			return True
	return False


def used_in_box(arr, row, col, num):
	for i in range(4):
		for j in range(4):
			if(arr[i + row][j + col] == num):
				return True
	return False


def check_location_is_safe(arr, row, col, num):
	return not used_in_row(arr, row, num) and not used_in_col(arr, col, num) and not used_in_box(arr, row - row % 4, col - col % 4,
num)


def solve_sudoku(arr):
    l = [0, 0]
    if(not find_empty_location(arr, l)):
        return True
    row = l[0]
    col = l[1]
    for num in range(1, 17):
        if(check_location_is_safe(arr, row, col, num)):
```

```
            arr[row][col] = num
            if(solve_sudoku(arr)):
                return True
            arr[row][col] = 0
    return False

if __name__=="__main__":

            grid =[[0 for x in range(16)]for y in range(16)]

            grid =[[1, 2, 0, 4, 5, 0, 7, 8, 0, 10, 11, 0, 13, 14, 0, 16],
                        [0, 6, 7, 0, 1, 2, 0, 4, 13, 0, 15, 16, 0, 10, 11, 12],
                        [9, 0, 11, 12, 0, 14, 15, 0, 1, 2, 0, 4, 5, 0, 7, 8],
                        [13, 14, 0, 16, 9, 0, 11, 12, 0, 6, 7, 0, 1, 2, 0, 4],
                        [0, 1, 0, 3, 0, 5, 0, 7, 0, 9, 12, 11, 14, 13, 16, 15],
                        [6, 5, 8, 7, 2, 1, 0, 3, 0, 13, 0, 15, 0, 9, 0, 11],
                        [10, 0, 12, 0, 14, 0, 16, 0, 2, 0, 4, 3, 6, 5, 8, 7],
                        [14, 13, 0, 15, 0, 9, 0, 11, 0, 5, 0, 7, 2, 1, 4, 3],
                        [3, 0, 0, 2, 0, 0, 5, 6, 11, 0, 9, 10, 15, 16, 13, 14],
                        [7, 8, 5, 6, 3, 4, 1, 0, 0, 16, 0, 0, 11, 0, 9, 10],
                        [0, 0, 9, 0, 0, 16, 13, 0, 3, 4, 1, 2, 7, 8, 5, 6],
                        [15, 0, 13, 14, 0, 12, 9, 10, 0, 8, 0, 6, 0, 4, 1, 2],
                        [4, 3, 2, 1, 8, 0, 6, 0, 12, 0, 10, 9, 0, 0, 14, 13],
                        [8, 7, 6, 5, 4, 0, 2, 1, 0, 0, 14, 13, 0, 11, 0, 9],
                        [12, 11, 10, 9, 16, 15, 14, 0, 4, 0, 2, 0, 8, 0, 0, 5],
                        [16, 15, 0, 13, 12, 0, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]]
            if(solve_sudoku(grid)):
                        print_grid(grid)
            else:
                        print("Solução não existe!")

fim = time.time()
print("\nTempo: ",fim - inicio)
```

APENDICE D – CÓDIGO ALGORITMO TROCO MÍNIMO

```
import time


moedas = [1000, 500, 100, 50, 20 , 10, 5, 2, 1]
def troco_min(valor, total=0):
	print("Troco = R$ ", valor)
	print("Cédulas:")
	for i in range(len(moedas)):
		troco = valor // moedas[i]
		valor -= troco * moedas[i]
		total += troco
		if(troco != 0):	print(troco, " de R$ ",moedas[i])
	print("Total de Moedas: ",total)

inicio = time.time()
troco_min(369)
fim = time.time()
print("\nTempo: ",fim - inicio)
```