Inácio Lima de Souza Filho – 509153

Para o trabalho foram usados 5 algoritmos de busca e para 7 casos diferentes:

1. Algoritmo 1: busca linear v1 O(n)
2. Algoritmo 1: busca linear v2 O(n)
3. Algoritmo 2: busca binária O(log n) (número procurado primeiro ou último)
4. Algoritmo 2: busca binária O(log n) (número procurado o elemento do meio)
5. Algoritmo 3: busca quadrática com contagem de repetição de elementos O(n^2)
6. Algoritmo 4: busca ternária O(log n)
7. Algoritmo 5: busca cúbica - tripla checagem O(n^3)

O código fornecido foi feito em Java, mas houve a conversão para python, foi usado a estrutura de array do Numpy no lugar dos vetores do Java. Para medir a memória foi utilizada a biblioteca do tracemalloc e como está sendo trabalhado com threads foi salvo em uma variável global, memoria.

```python
def thread_function(funcao):
    tracemalloc.start()  # Inicia o rastreamento
    data = funcao()
    current, peak = tracemalloc.get_traced_memory()
    print(f"Uso de memória: {peak / 10**3} KB")
    global memoria
    memoria = peak / 10**6
    tracemalloc.stop()  # Para o rastreamento
```

Para fazer a busca foi usado um método que faz a busca para qualquer algoritmo recebido por parâmetro. Nesse algoritmo são usados vários tamanhos de vetores (os tamanhos foram mudados para melhor visualização) e feito a busca para cada tamanho, também é gerado o vetor e passado a função para executar em uma thread separada. No final se calcula o tempo total gasto e imprime na tela.

```python
tamanhos = [100, 200, 400, 800, 1600, 3200, 6400, 12800, 25600, 51200, 102400, 204800, 409600, 819200, 1638400,
            3276800, 6553600, 13107200, 26214400, 52428800, 104857600]
memoria = 0
def buscar(funcao, posicaoElementoPorcentagem, quantidade_buscas=21):
    resultadosMem = []
    resultadosTempo = []
    for i in range(quantidade_buscas):
        tamanho = tamanhos[i]
        vetor = gerarArrayOrdenado(tamanho)
        tempoInicial = time.time()
        posicaoElemento = int((vetor.shape[0])*posicaoElementoPorcentagem/100)
        if posicaoElemento==vetor.shape[0]:
            posicaoElemento -= 1
        thread = threading.Thread(target=thread_function,
                                  args=(lambda : funcao(vetor[posicaoElemento], vetor.copy(), 0, vetor.shape[0]),))
        thread.start()
        thread.join()
        tempoFinal = time.time()
        tempoDecorrido = tempoFinal - tempoInicial
        print(f"Tempo decorrido: {tempoDecorrido} segundos para o tamanho {tamanho}")
        resultadosTempo.append(tempoDecorrido)
        resultadosMem.append(memoria)
    return (resultadosTempo, resultadosMem)
```

Para cada algoritmo foi desenvolvida uma função e essa função foi passada como parâmetro para ser feito a busca.

Para busca Linear V1

```
def buscaLinearV1(x, v, *args):
    indice = -1
    for i in range(v.shape[0]):
        if v[i] == x:
            indice = i
    return indice
```

```
resultado = buscar(buscaLinearV1, 0)
resultadoTotTempo.append(resultado[0])
resultadoTotMem.append(resultado[1])
```

Nesse caso, o segundo parâmetro da função é a posição a ser buscada no vetor, se 0 a primeira posição, se 100 a última. Para a busca quadrática e cúbica também foi necessário limitar o tamanho do vetor, porque o tempo de processamento ficou muito grande, a limitação é o terceiro parâmetro da função.

```
resultado = buscar(quadratica, 100,8)
resultadoTotTempo.append(resultado[0])
resultadoTotMem.append(resultado[1])
```

O último passo foi passar esses resultados para tabelas:

```
pd.DataFrame(resultadoTotMem, columns=tamanhos).to_excel("resultadoMemoria.xlsx")
pd.DataFrame(resultadoTotTempo, columns=tamanhos).to_excel("resultadoTempo.xlsx")
```

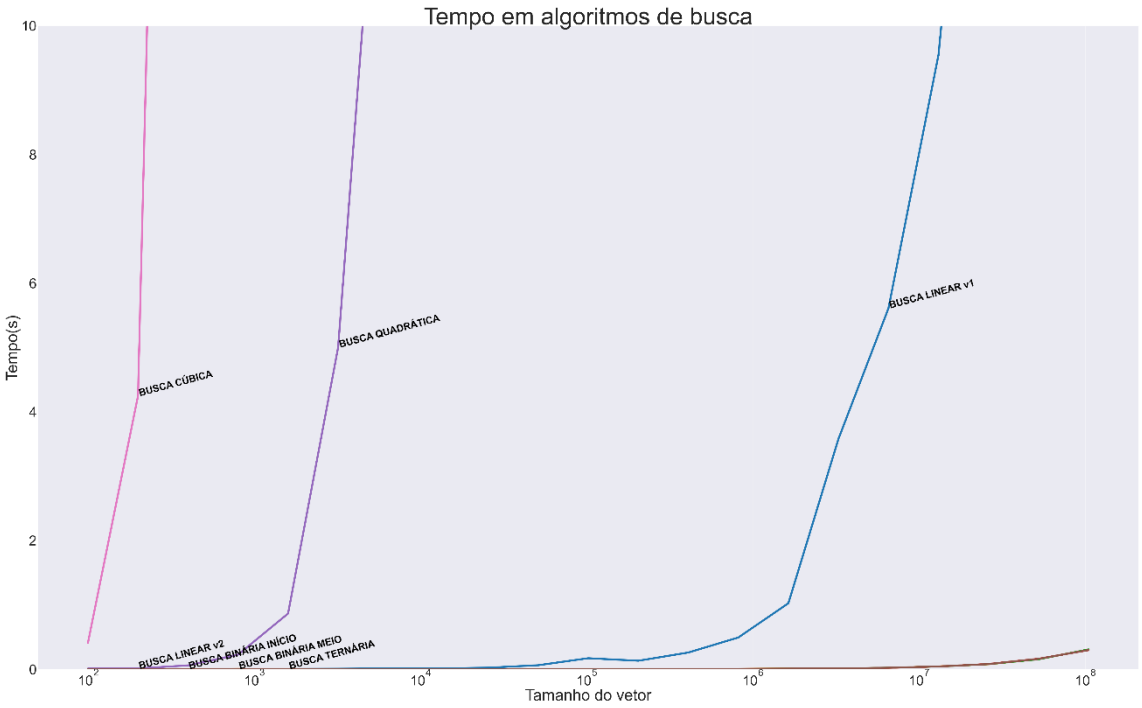
Segue o resultado:

	Tamanho	0	1	2	3	4	5	6
0	100	0.001997	0.005018	0.003001	0.004395	0.007998	0.002990	0.711479
1	200	0.003002	0.005091	0.005019	0.007800	0.013992	0.001998	4.271470
2	400	0.002993	0.004659	0.004051	0.004348	0.066532	0.002418	37.529347
3	800	0.002999	0.001800	0.003002	0.004507	0.296452	0.001997	336.927718
4	1600	0.003999	0.003559	0.004171	0.004555	1.075625	0.003000	NaN
5	3200	0.005008	0.003852	0.004545	0.004988	4.124570	0.003001	NaN
6	6400	0.004002	0.003543	0.004111	0.004226	16.403162	0.001010	NaN
7	12800	0.011005	0.004931	0.005024	0.003001	69.594408	0.003004	NaN
8	25600	0.013018	0.002999	0.006036	0.007071	NaN	0.003479	NaN
9	51200	0.028006	0.005056	0.005346	0.005049	NaN	0.003310	NaN
10	102400	0.064616	0.005149	0.005425	0.004047	NaN	0.004015	NaN
11	204800	0.124736	0.021885	0.007985	0.006409	NaN	0.006166	NaN
12	409600	0.185955	0.009651	0.003840	0.005366	NaN	0.006821	NaN
13	819200	0.379121	0.008786	0.009595	0.004997	NaN	0.008264	NaN
14	1638400	0.659957	0.009881	0.008370	0.006576	NaN	0.008775	NaN
15	3276800	2.053887	0.009080	0.016723	0.007010	NaN	0.011096	NaN
16	6553600	3.032741	0.010538	0.017529	0.009999	NaN	0.012789	NaN
17	13107200	5.852978	0.018999	0.036863	0.032097	NaN	0.021198	NaN
18	26214400	13.257146	0.051391	0.054834	0.048928	NaN	0.029740	NaN
19	52428800	30.119372	0.088158	0.085934	0.064995	NaN	0.058006	NaN
20	104857600	51.489315	0.237523	0.257322	0.147228	NaN	0.156119	NaN

Pela a análise, os métodos mais rápidos foram a busca binária no meio do vetor e a busca ternária. Não foi possível executar os algoritmos quadráticos e cúbicos porque eles

levariam muito tempo, então foi executado apenas para tamanhos menores, mas mesmo assim é possível perceber como o tempo nesses algoritmos foi muito maior.

Para plotar os gráficos foi usado o lineplot do Seaborn, como valores do cúbico e quadrático destoam muito dos outros valores foi feita uma mudança da escala. Segue o resultado:



Nesse caso é possível notar com clareza como os algoritmos de O(log n) são muito menores que os outros. Para a busca linear o gráfico ficou parecendo um exponencial, mas isso ocorre porque quando foi plotado o gráfico se usou escala logarítmica para os tamanhos.

Já para a memória também foi montado uma tabela:

	Tamanho	0	1	2	3	4	5	6
0	100	0.001904	0.001016	0.001872	0.000944	0.003434	0.000992	0.072997
1	200	0.001816	0.001816	0.002208	0.001744	0.003590	0.001792	0.094960
2	400	0.003500	0.003500	0.003836	0.003376	0.016206	0.003488	0.094536
3	800	0.006700	0.006828	0.007092	0.006604	0.061507	0.006688	0.122332
4	1600	0.013100	0.013100	0.013548	0.013004	0.080029	0.013088	NaN
5	3200	0.025900	0.025900	0.026404	0.025804	0.120520	0.026764	NaN
6	6400	0.051500	0.051500	0.052060	0.051404	0.129010	0.051516	NaN
7	12800	0.102700	0.102700	0.102856	0.102604	0.192785	0.102716	NaN
8	25600	0.205100	0.205100	0.205772	0.205004	NaN	0.205116	NaN
9	51200	0.434066	0.409900	0.410628	0.409804	NaN	0.409916	NaN
10	102400	0.823042	0.819500	0.820284	0.819404	NaN	0.819516	NaN
11	204800	1.641112	1.638700	1.639540	1.638604	NaN	1.638716	NaN
12	409600	3.282803	3.277100	3.277996	3.277004	NaN	3.277116	NaN
13	819200	6.572995	6.553900	6.554852	6.553804	NaN	6.553916	NaN
14	1638400	13.171747	13.107712	13.108508	13.107556	NaN	13.107516	NaN
15	3276800	26.275780	26.214700	26.215764	26.214604	NaN	26.214868	NaN
16	6553600	52.492204	52.429100	52.429912	52.431212	NaN	52.429116	NaN
17	13107200	104.923057	104.914818	104.861748	104.857804	NaN	104.858068	NaN
18	26214400	209.782492	209.718068	209.719722	209.715404	NaN	209.715668	NaN
19	52428800	419.505731	419.439209	419.432140	419.437156	NaN	419.433076	NaN
20	104857600	838.949373	838.870249	838.923685	838.864879	NaN	838.866327	NaN

Nesse caso é notável como a memória não teve muita variação entre um método e outro, apenas um pequeno uso maior para o método de busca binária e ternária. Segue o gráfico: