Tópicos Especiais em Computação II

Trabalho Esquenta

Francilândio Lima Serafim (472644)

UNIVERSIDADE
FEDERAL DO CEARÁ

04 de Setembro de 2023

Conteúdo

1 Introdução

O presente relatório descreve o procedimento feito para analisar de forma prática alguns algoritmos de busca de diferentes complexidades dentre os quais estão:

a) Algoritmo 1: busca linear v1 O(n)

b) Algoritmo 1: busca linear v2 O(n)

c) Algoritmo 2: busca binária O($\log n$)

d) Algoritmo 3: busca quadrática com contagem de repetição de elementos O(n^2)

e) Algoritmo 4: busca ternária O($\log n$)

f) Algoritmo 5: busca cúbica - tripla checagem O(n^3)

Sendo que a proposta é passar como entrada diferentes conjuntos de dados, que são compostos por números inteiros ordenados ou não ordenados variando em tamanho de modo que cada classe de dados possui arquivos com diferentes quantidades de números, correspondendo às quantidades: 100, 200, 1000, 2000, 5000, 10000, 50000, 100000, 500000, 1000000, 5000000, 10000000 e 100000000. E a partir das execuções desses algoritmos, medir o tempo e consumo de memória gastos.

No presente trabalho, devido alguns contratempos causados por fatores como a dificuldade em escolher uma linguagem que possibilitasse a fácil mensuração de tempo de execução e consumo de memória de cada algoritmo e a grande demanda de tempo para execução de alguns algoritmos com bases maiores de dados que ameaçava o deadline do prazo de entrega, nem todos os requisitos foram atendidos como por exemplo a falta de execução dos algoritmos de busca cúbica e quadrática para as bases ordenadas a partir de 5000 números e todas as bases não ordenadas. Também não foi possível treinar os algoritmos exceto os supracitados com bases com quantidade de números a partir de 5000000 tanto ordenados como não ordenados.

2 Metodologia

Dada a dificuldade de medir o consumo de memória das execuções dos algoritmos na linguagem Java, tendo feitas tentativas de executar localmente e falhando por falta de memória mesmo para os casos mais simples, partindo para tentar executar no codespace do github e também encontrando dificuldades foi decidido converter os algoritmos passados da linguagem Java para Python rodando os códigos no Google Colaboratory.

O código consiste em funções que automatizam as execuções de cada algoritmo sobre as bases através de laços de repetição, após cada execução os dados de consumo de memória, usando a biblioteca *tracemalloc*, e tempo, usando a biblioteca *time*, foram calculados e colocados em um dicionário para posteriormente serem gerados gráficos visando a facilitação de visualizar e entender todo o processo.

Um exemplo de código é mostrado na Figura 1, nele estão as funções de pesquisa binária e a responsável pela automatização das iterações usando as diferentes bases de dados.

```python
def PesquisaBinaria(x, v, e, d):
    meio = (e + d) // 2
    if v[meio] == x:
        return meio
    if e >= d:
        return -1
    elif v[meio] < x:
        return PesquisaBinaria(x, v, meio + 1, d)
    else:
        return PesquisaBinaria(x, v, e, meio - 1)

def buscaBinariaAuto(ordenados):
  tracemalloc.start()
  buscabinariaordenado = {"memoria":{}, "tempo":{}}

  for arquivo in (ordenados):
    arrayo = []
    with open(arquivo, 'r') as file:
        readero = csv.reader(file)
        for line in readero:
            for token in line:
                arrayo.append(int(token))

    startMemoryo, _ = tracemalloc.get_traced_memory()
    startTimeo = time.time()
    PesquisaBinaria(arrayo[len(arrayo) - 1], arrayo, 0, len(arrayo))
    endTimeo = time.time()
    endMemoryo, _ = tracemalloc.get_traced_memory()
    totalTimeo = endTimeo - startTimeo
    totalMemoryo = (endMemoryo - startMemoryo)
    buscabinariaordenado['memoria'][arquivo] = totalMemoryo
    buscabinariaordenado['tempo'][arquivo] = totalTimeo

  return buscabinariaordenado
dicbuscabinaria = buscaBinariaAuto(ordenados)
with open("buscaBinaria.pkl", "wb") as f:
    # Usa a função dump para salvar o dicionário no arquivo
    pickle.dump(dicbuscabinaria, f)
```

Figura 1: Funções em Python para pesquisa binária

3 Resultados

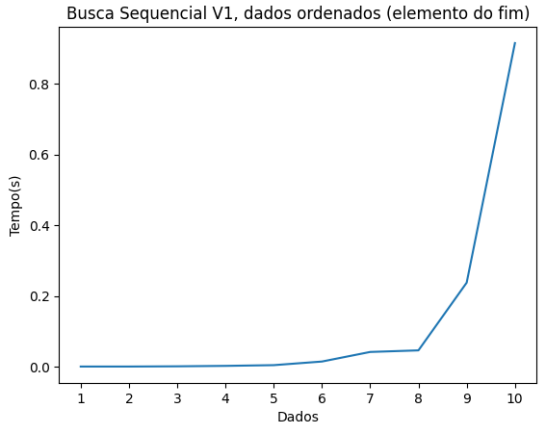
Por convenção, os tamanhos dos dados foram enumerados da seguinte forma:

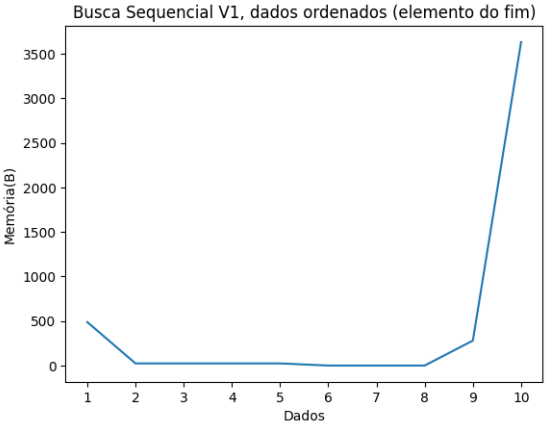
1. 100
2. 200
3. 1000
4. 2000
5. 5000
6. 10000
7. 50000
8. 100000
9. 500000
10. 1000000
11. 5000000
12. 10000000
13. 100000000

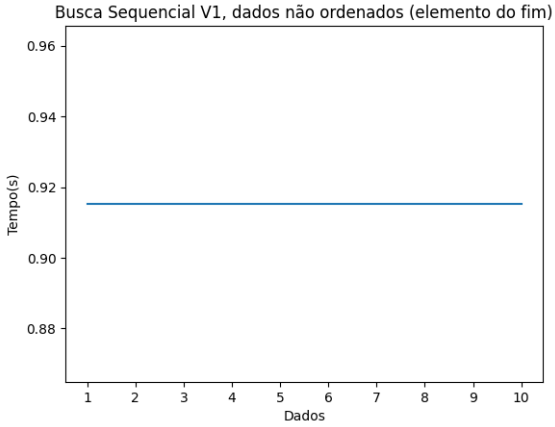
A convenção foi aderida para enxugar os gráficos utilizados para ilustrar as medições, sendo que neles os números correspondentes às quantidades de números das bases são colocados no eixo x.

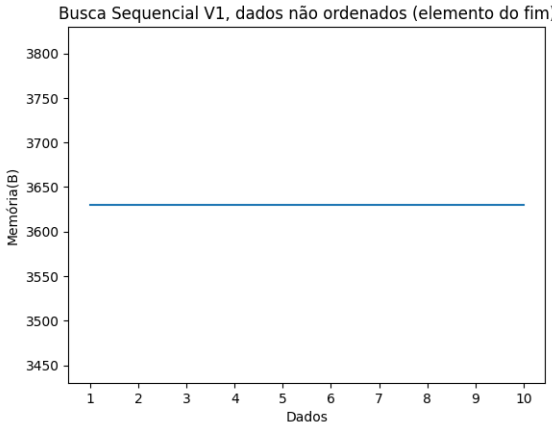
Em todas as execuções com exceção de uma relacionada ao algoritmo de pesquisa binária foram buscados elementos na última posição dos arrays povoados com os números das bases de dados, de outra forma o elemento buscado era o do meio.

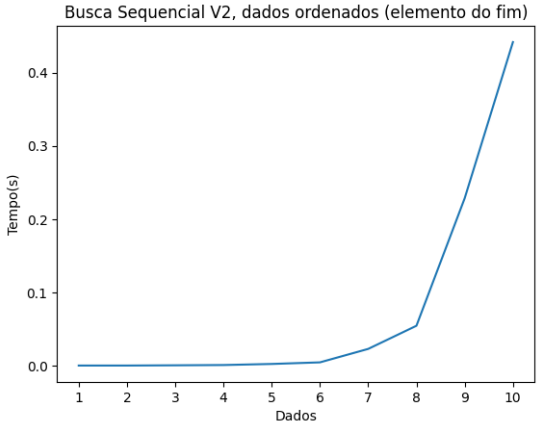
Devido a falta de testes com números em diferentes posições no array para cada execução dos algoritmos, ficou inviável debater sobre em que cenários cada algoritmo funciona melhor, porém através dos gráficos mostrados a seguir é possível fazer comparações e deduzir a superioridade de um em relação ao outro em termos de gasto de memória e tempo de execução, comprovando o que se espera dadas as conhecidas complexidades de cada um. Porém em alguns casos os resultados parecem incondizentes com o que se esperava como é mostrado nos gráficos de busca sequencial V1 e V2 usando dados não ordenados tanto na memória como no tempo, sendo apresentados retas constantes para todas as 10 primeiras bases usadas de dados não ordenados.

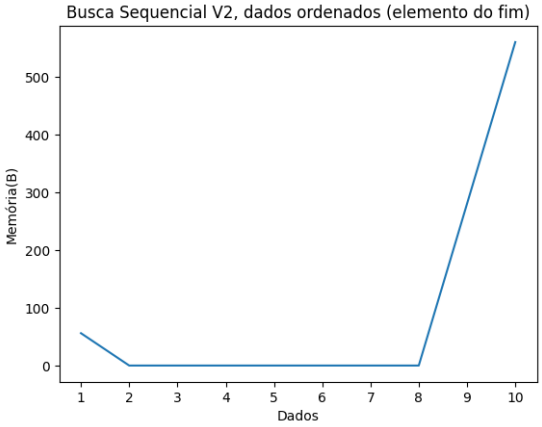

Busca Sequencial V1, dados ordenados (elemento do fim)
Tempo(s)
0.0
0.2
0.4
0.6
0.8
1
2
3
4
5
6
7
8
9
10
Dados

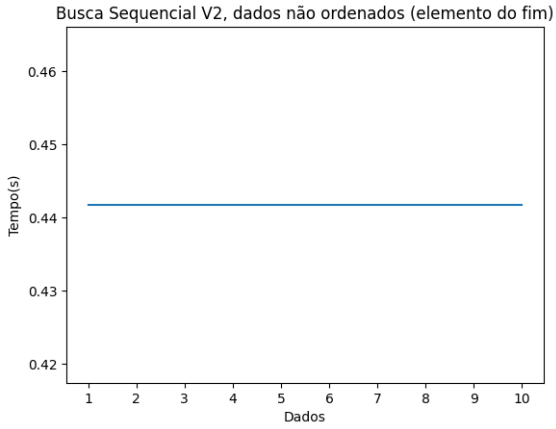

Busca Sequencial V1, dados ordenados (elemento do fim)
Memória(B)
0
500
1000
1500
2000
2500
3000
3500
1
2
3
4
5
6
7
8
9
10
Dados

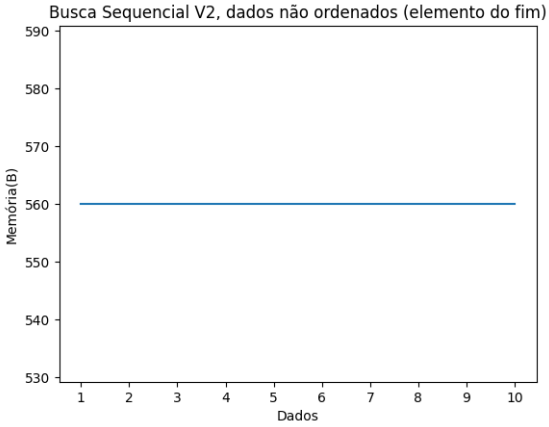

Busca Sequencial V1, dados não ordenados (elemento do fim)
Tempo(s)
0.88
0.90
0.92
0.94
0.96
1
2
3
4
5
6
7
8
9
10
Dados

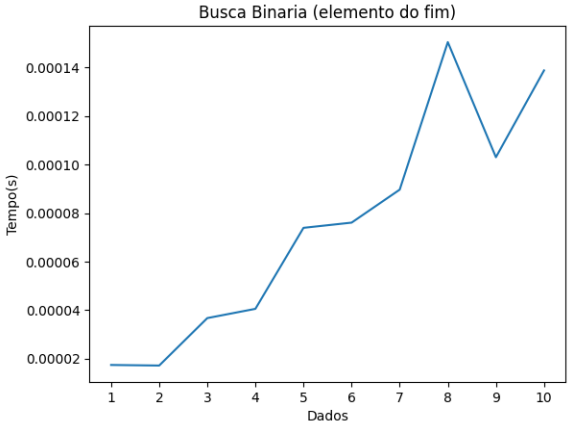

Busca Sequencial V1, dados não ordenados (elemento do fim)
Memória(B)
3800
3750
3700
3650
3600
3550
3500
3450
1
2
3
4
5
6
7
8
9
10
Dados

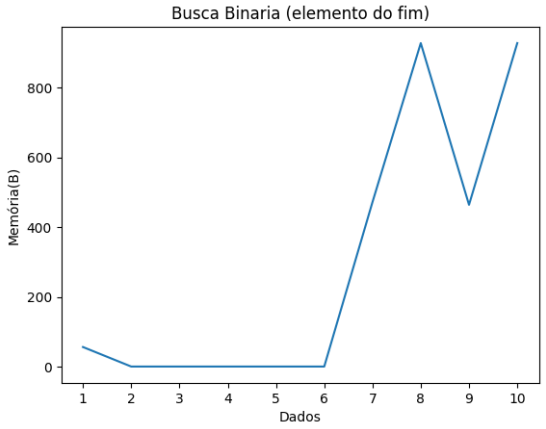

Busca Sequencial V2, dados ordenados (elemento do fim)
Tempo(s)
0.4
0.3
0.2
0.1
0.0
1
2
3
4
5
6
7
8
9
10
Dados

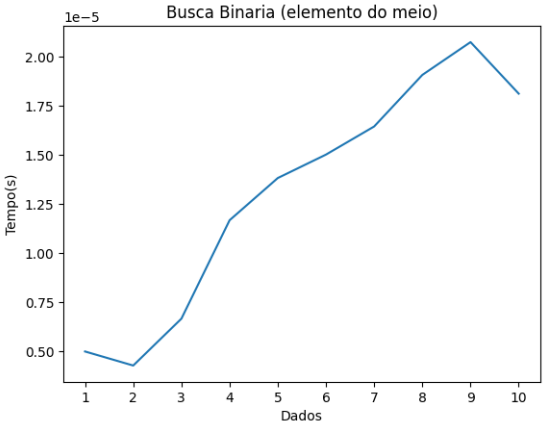

Busca Sequencial V2, dados ordenados (elemento do fim)
Memória(B)
500
400
300
200
100
0
1
2
3
4
5
6
7
8
9
10
Dados

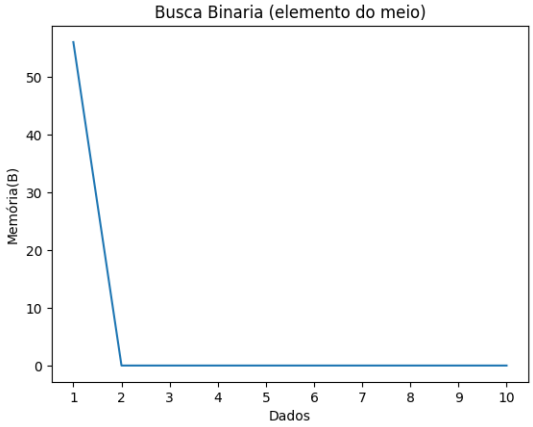

Busca Sequencial V2, dados não ordenados (elemento do fim)
Tempo(s)
0.46
0.45
0.44
0.43
0.42
1
2
3
4
5
6
7
8
9
10
Dados

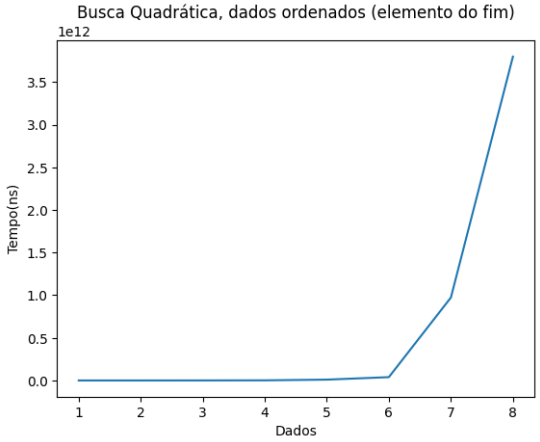

Busca Sequencial V2, dados não ordenados (elemento do fim)
Memória(B)
590
580
570
560
550
540
530
1
2
3
4
5
6
7
8
9
10
Dados

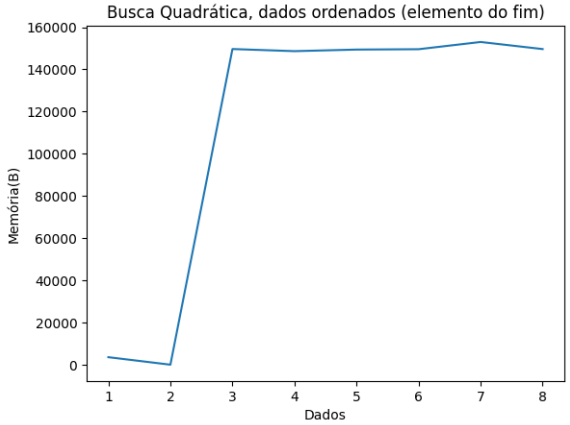

Busca Binaria (elemento do fim)
Tempo(s)
0.00014
0.00012
0.00010
0.00008
0.00006
0.00004
0.00002
1
2
3
4
5
6
7
8
9
10
Dados

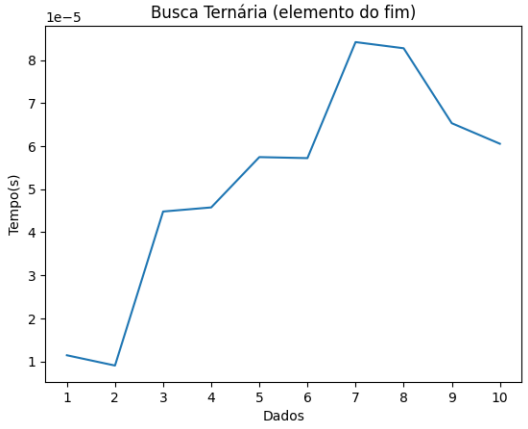

Busca Binaria (elemento do fim)
Memória(B)
0
200
400
600
800
1 2 3 4 5 6 7 8 9 10
Dados

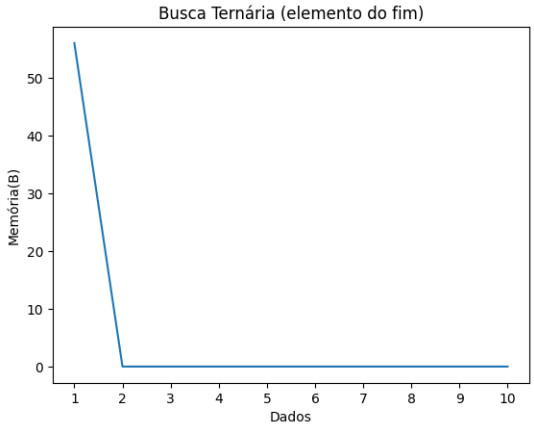

Busca Binaria (elemento do meio)
1e−5
Tempo(s)
0.50
0.75
1.00
1.25
1.50
1.75
2.00
1 2 3 4 5 6 7 8 9 10
Dados

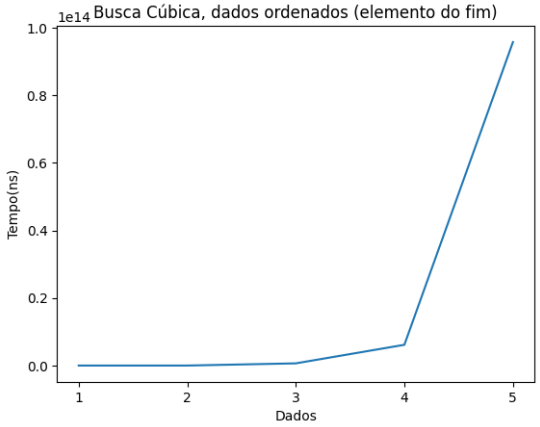

Busca Binaria (elemento do meio)
Memória(B)
0
10
20
30
40
50
1 2 3 4 5 6 7 8 9 10
Dados

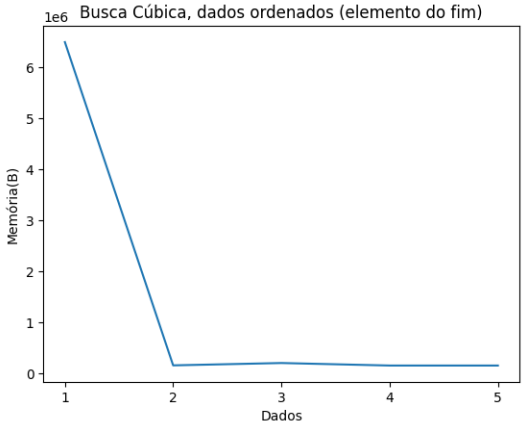

Busca Quadrática, dados ordenados (elemento do fim)
1e12
Tempo(ns)
Dados


Busca Quadrática, dados ordenados (elemento do fim)
Memória(B)
Dados


Busca Ternária (elemento do fim)
1e−5
Tempo(s)
Dados


Busca Ternária (elemento do fim)
Memória(B)
Dados
0
10
20
30
40
50
1
2
3
4
5
6
7
8
9
10


Busca Cúbica, dados ordenados (elemento do fim)
1e14
Tempo(ns)
Dados
0.0
0.2
0.4
0.6
0.8
1.0
1
2
3
4
5


Busca Cúbica, dados ordenados (elemento do fim)
1e6
Memória(B)
Dados
0
1
2
3
4
5
6
1
2
3
4
5