UNIVERSIDADE FEDERAL DO CEARÁ – CAMPUS SOBRAL
CURSO DE ENGENHARIA DE COMPUTAÇÃO

DISCIPLINA: TÓPICOS ESPECIAIS EM COMPUTAÇÃO II

PROFESSORA: FISCHER JONATAS FERREIRA

TRABALHO ESQUENTA 02 - Algoritmos para encontrar o maior valor

ALUNO	MATRÍCULA
Francisco Cassiano de Vasconcelos Souza	413067

Sobral – CE

2023

SUMÁRIO

1. INTRODUÇÃO

A tarefa de encontrar o maior valor em um conjunto de dados é uma operação fundamental em ciência da computação e análise de dados. Em diversos contextos, desde ordenação e análise estatística até processamento de informações, a necessidade de identificar o valor máximo de um conjunto de elementos é comum e essencial. Neste relatório, abordaremos dois dos algoritmos para encontrar o maior valor de um vetor, cada algoritmo oferece uma abordagem distinta para solucionar o mesmo problema, e entender as características é fundamental para selecionar a melhor solução.

2. MÉTODO

Para a execução dos algoritmos foi escolhida a linguagem de programação Python pela facilidade de uso e por uma gama de bibliotecas disponíveis adequadas para medição de desempenho.

A máquina utilizada para a análise de desempenho dos algoritmos de busca portava um processador **Intel(R) Core(TM) i5-1145G7**, memória RAM **8,00 GB**.

Os arquivos utilizados como fonte de dados para o experimento e análise foram disponibilizados na atividade anterior, porém podem ser gerados novos arquivos com o arquivo gerador **geraArquivoDesordenado.py**. Para a análise dos algoritmos foram utilizadas apenas as instâncias não ordenadas.

A análise de desempenho do algoritmo é realizada através da função disponível no arquivo **./module/valoresMetricas.py**.

```
        total_tempo = 0

        total_memoria = 0

        tracemalloc.start()

        for _ in range(iteracoes):

            inicio_tempo = time.time()
```

```
            codigo()  # Executa o trecho de código fornecido
            fim_tempo = time.time()
            tempo_gasto = fim_tempo - inicio_tempo
            total_tempo += tempo_gasto
            snapshot = tracemalloc.take_snapshot()
            memoria_usada = sum(stat.size for stat in
snapshot.statistics("lineno"))
            total_memoria += memoria_usada
        media_tempo = total_tempo / iteracoes
        media_memoria = total_memoria / iteracoes
```

O código inicia um cronômetro utilizando **time.time()** e um rastreador de alocação de memória usando **tracemalloc.start()**, executa o trecho de código pela função **codigo()** as vezes determinadas pela variável **iteracoes**, medindo o tempo gasto e uso de memória em cada execução.

3. DESENVOLVIMENTO

Será realizada uma análise de desempenho dos seguintes algoritmos que encontram o maior valor em um vetor:

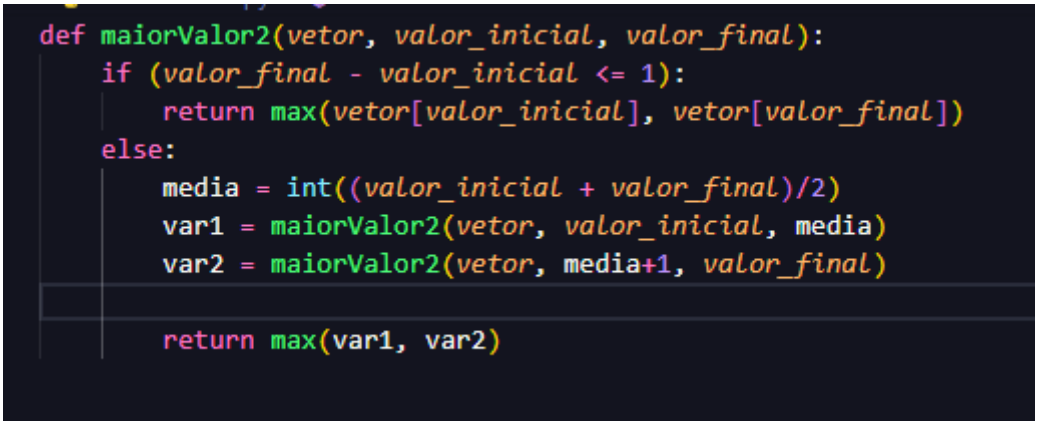
Figura 01- Algoritmo maiorValor1

```
def maiorValor1(vetor, instance):
    maior = vetor[0]
    for i in range(1, instance):
        if vetor[i] > maior:
            maior = vetor[i]
    return maior
```

Fonte: Próprio autor

No geral, essa função encontra e retorna o maior valor dentro de uma parte específica do vetor, limitada pelo valor de **instance.**

Figura 02- Algoritmo maiorValor2



```
def maiorValor2(vetor, valor_inicial, valor_final):
    if (valor_final - valor_inicial <= 1):
        return max(vetor[valor_inicial], vetor[valor_final])
    else:
        media = int((valor_inicial + valor_final)/2)
        var1 = maiorValor2(vetor, valor_inicial, media)
        var2 = maiorValor2(vetor, media+1, valor_final)

        return max(var1, var2)
```

Fonte: Próprio autor

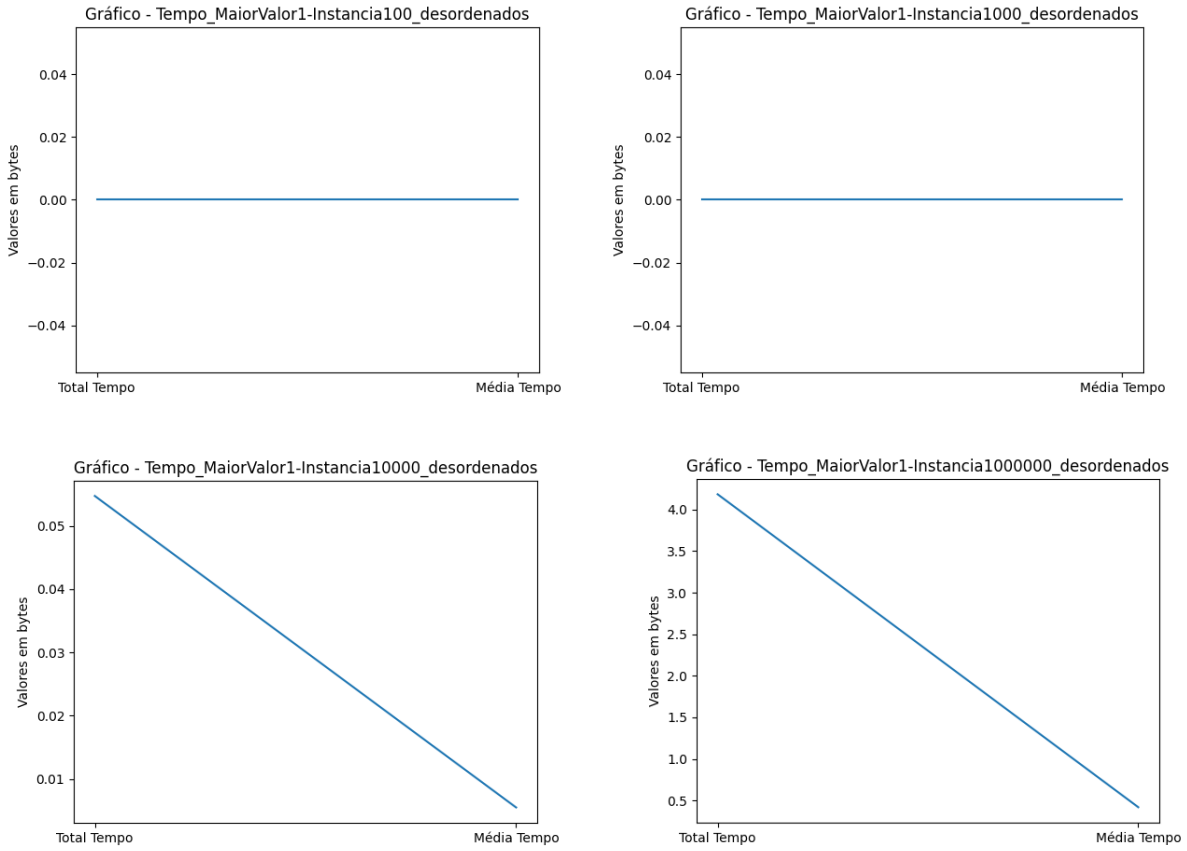
Utilizou-se 13 instâncias de dados com tamanhos variados que foram gerados pelo algoritmo anteriormente citado, **geraArquivoDesordenado.py**. Foram utilizadas apenas instâncias **não ordenadas.**

3.1 maiorValor1 O(n)

O código usa um único loop for que itera através dos elementos do vetor de 1 até instance - 1, que trata-se da quantidade de elementos no vetor. Portanto, o tempo de execução é diretamente proporcional ao tamanho do vetor, o que resulta em uma complexidade de tempo O(n), onde n é o número de elementos no vetor. À medida que o tamanho do vetor aumenta, o tempo de execução aumentará linearmente. Para vetores pequenos, o tempo de execução será relativamente curto, mas para vetores grandes, o tempo de execução pode tornar-se significativo.

Para a análise foi utilizadas as 13 instâncias não ordenadas, para cada instância realizou-se 10 iterações. As figuras a seguir mostram o consumo de tempo nas instâncias 100, 1000, 10000 e 1000000.

Figura 03- Consumo de tempo algoritmo maiorValor1



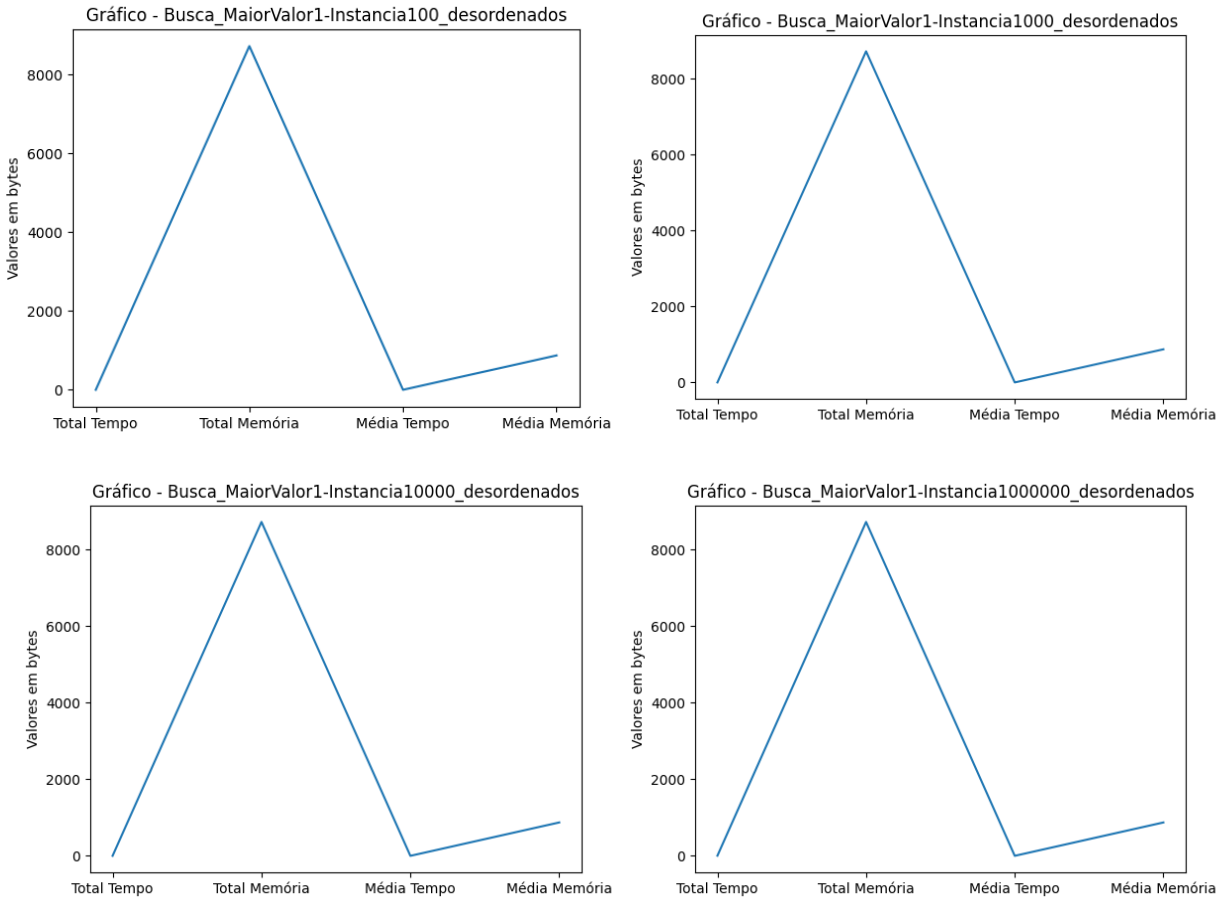
Fonte: Próprio autor

Nota-se que como foi mencionado acima, à medida que o vetor de entrada cresce o consumo de tempo do algoritmo cresce linearmente e para vetores muito grandes o tempo pode ser considerável.

A função maiorValor1 não cria estruturas de dados adicionais com base no tamanho do vetor, exceto para uma variável (**maior**) para armazenar o maior valor encontrado. Portanto, o consumo de memória é constante e não depende do tamanho do vetor. Independentemente de o vetor ter 100 elementos ou 100 milhões de elementos, o consumo de memória da função maiorValor1 será praticamente o mesmo.

As figuras a seguir mostram o consumo de memória nas instâncias 100, 1000, 10000 e 1000000.

Figura 04- Consumo de memória algoritmo maiorValor1



Fonte: Próprio autor

Nos gráficos acima, percebe-se que o consumo de memória de fato é praticamente constante independente do tamanho da entrada do vetor, não importando a instância.

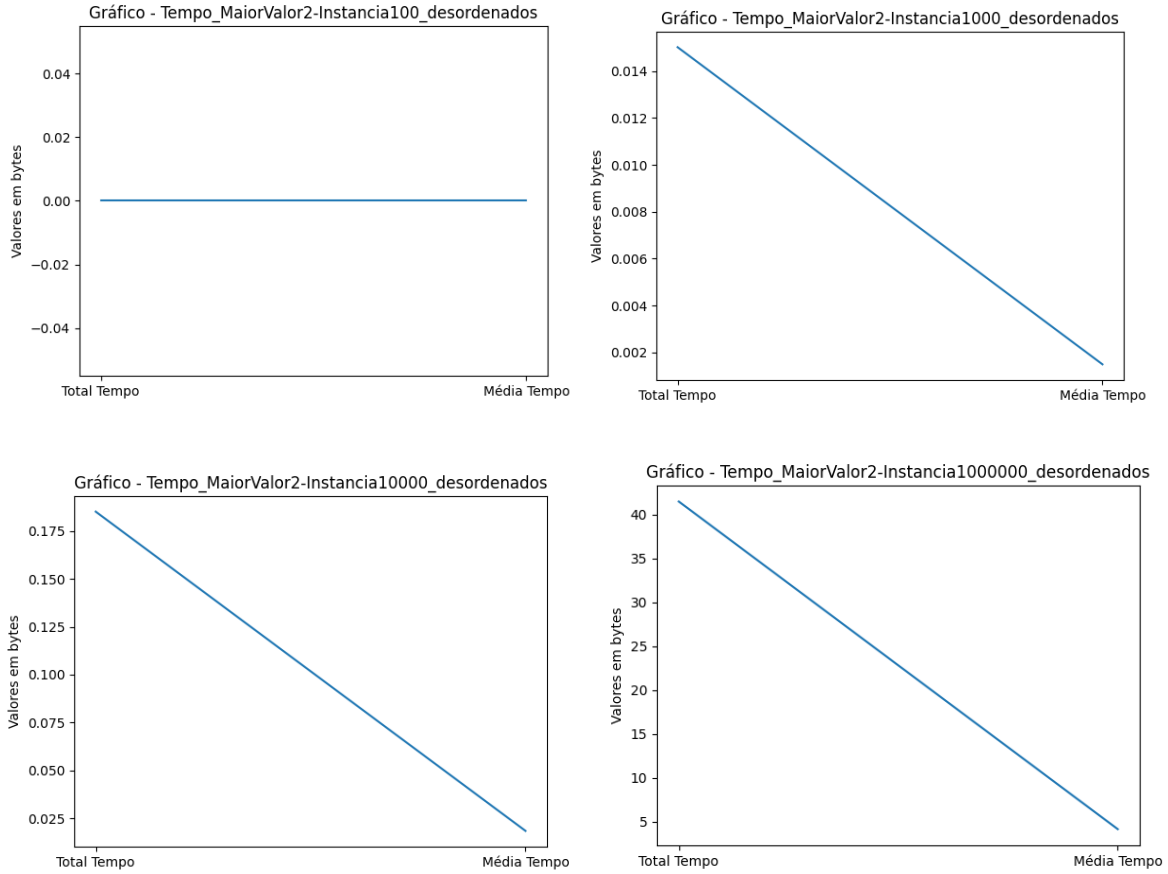
3.2 maiorValor2 O(n log n)

O código maiorValor2 implementa um algoritmo de busca recursiva para encontrar o maior valor em um vetor.

O algoritmo maiorValor2 divide o vetor pela metade em cada chamada recursiva. Portanto, a recursão ocorre aproximadamente log2(n) vezes, onde "n" é o tamanho do vetor. Em cada nível da recursão, o algoritmo realiza uma operação de comparação para encontrar o máximo valor entre os elementos da metade do vetor. Como resultado, a complexidade de tempo é O(n log n), onde "n" é o tamanho do vetor.

As figuras a seguir mostram o consumo de tempo nas instâncias 100, 1000, 10000 e 1000000.

Figura 05- Consumo de tempo algoritmo maiorValor2



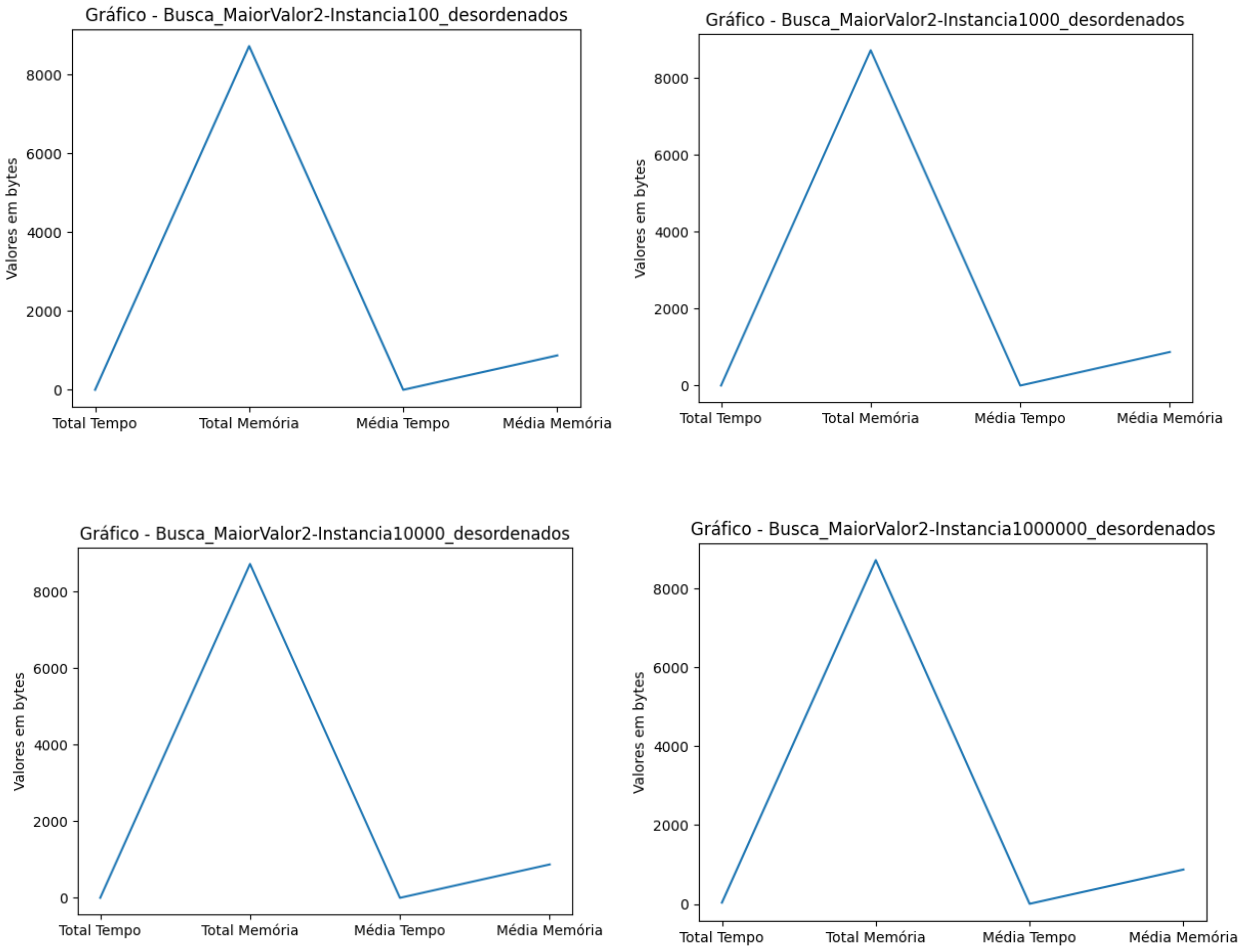
Fonte: Próprio autor

O algoritmo maiorValor2 possui uma complexidade de tempo O(n log n) onde “n” é o tamanho do vetor, o que o torna menos eficiente quando tratando-se de entradas não ordenadas. Pode-se notar que à medida que a instância cresce o consumo de tempo cresce linearmente e com maior velocidade.

O consumo de memória do algoritmo é logarítmico em relação ao tamanho do vetor (O(log n)), independentemente da ordenação dos valores. Ele usa memória adicional para as chamadas recursivas, mas o consumo de memória é controlado.

As figuras a seguir mostram o consumo de memória nas instâncias 100, 1000, 10000 e 1000000.

Figura 06- Consumo de memória algoritmo maiorValor2



Fonte: Próprio autor

Observa-se que o consumo de memória é praticamente constante, não importando o tamanho da instância do vetor

4. CONCLUSÃO

Para vetores não ordenados, o algoritmo **maiorValor1**(Busca Linear) é a escolha mais eficiente em termos de tempo. É fácil de implementar e eficiente para vetores pequenos a moderados, independentemente de sua ordenação.

O algoritmo **maiorValor2**(Divisão e Conquista) não é a escolha mais eficaz para vetores não ordenados, pois sua complexidade de tempo é maior.

Portanto, ao lidar com vetores não ordenados, o algoritmo **maiorValor1** é a opção mais adequada em termos de eficiência de tempo. A escolha entre os algoritmos irá sempre depender das necessidades específicas do problema a qual deseja-se resolver, mas nesse caso o algoritmo **maiorValor1** é o mais adequado.

5. REFERÊNCIA BIBLIOGRÁFICA

[1] PUC, Dep. Informática, **Programação II Busca em Vetor.** Disponível em: https://www-di.inf.puc-rio.br/~bfeijo/prog2/ProgII_Busca_Vetor.pdf . Acesso em: 11 de setembro de 2023.