Trabalho 2 - Divisão e Conquista

O problema da "Maior Subsequência Crescente Contígua" é um problema clássico em ciência da computação e algoritmos, dessa forma, ele consiste em encontrar a subsequência contígua mais longa em uma lista de números, onde os elementos estão em ordem crescente.

Assim, a definição exata pode variar, mas em geral, uma subsequência é um conjunto de elementos em uma ordem específica, mantendo sua posição relativa na lista original. No caso da "Maior Subsequência Crescente Contígua", estamos procurando uma subsequência de números consecutivos que estejam em ordem crescente. Dessa maneira, existem várias abordagens para resolver esse problema, como a abordagem de divisão e conquista ou algoritmos baseados em programação dinâmica. Sendo a única forma de critério de desempate a eficiência desejada para lidar com diferentes tamanhos de lista.

Como uma forma de implementação e resolução do problema, temos o algoritmo de divisão e conquista que tem sua complexidade de O(n) por apenas ter um único for necessário.

divide_and_conquer.py > ...

```python
def divide_increasing_subsequences(number_list):
    subsequences = []
    current_subsequence = [number_list[0]]

    for i in range(1, len(number_list)):
        if number_list[i] >= number_list[i - 1]:
            current_subsequence.append(number_list[i])
        else:
            subsequences.append(current_subsequence)
            current_subsequence = [number_list[i]]

    if current_subsequence:
        subsequences.append(current_subsequence)

    return subsequences

def longest_subsequence(number_list):

    subsequences = divide_increasing_subsequences(number_list)

    max_subsequence = max(subsequences, key=len)
    return max_subsequence

```

O ponto de partida é a função **longest_subsequence**, a qual invoca a função **divide_increasing_subsequences**. Esta última itera pela lista de números, identificando e

formando subsequências crescentes. Durante a iteração, cada número é comparado ao seu predecessor imediato para determinar se a sequência está crescendo ou se uma nova subsequência deve ser iniciada.

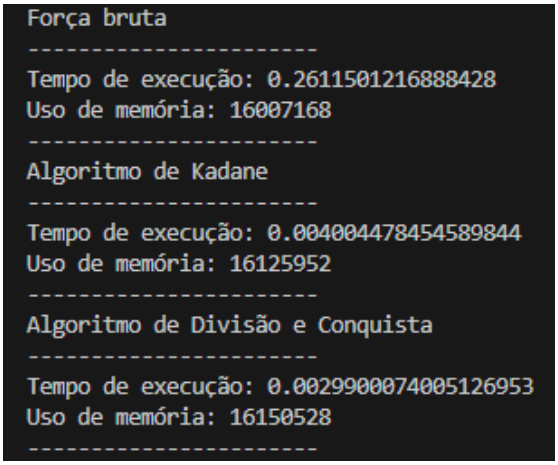
A função de segmentação, **divide_increasing_subsequences**, resulta em diversas subsequências crescentes presentes na lista original. Cada vez que um número subsequente não atende a essa condição, uma nova subsequência é iniciada, permitindo a formação de vários conjuntos crescentes.

Após a identificação de todas as subsequências, a função **longest_subsequence** seleciona a subsequência mais longa entre elas. Isso é feito comparando o comprimento de cada subsequência para determinar aquela que é a mais extensa.

Assim, é possível identificar a maior sequência de números crescentes na lista original. Ao aplicar a segmentação e seleção das subsequências crescentes, esse método oferece uma abordagem eficaz para identificar padrões crescentes em conjuntos de números, tendo aplicabilidade em análise de dados e algoritmos de otimização.

Tendo como exemplo 3 tipos de soluções para o problema podemos observar que a divisão e conquista se torna muito melhor em grande escala do que o força bruta, principalmente quando estamos de escalabilidade.

No exemplo abaixo foi utilizado um conjunto de 500 exemplos de lista de números de tamanho entre 5 e 100.

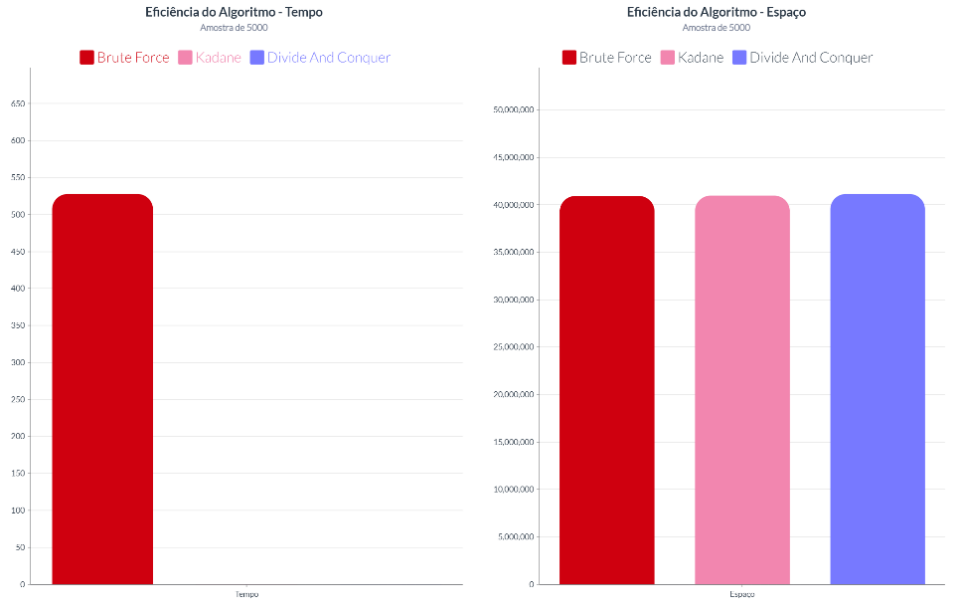


No exemplo abaixo foi utilizado um conjunto de 1000 exemplos de lista de números de tamanho entre 5 e 500.

```
Força bruta
----------------------
Tempo de execução: 20.903597116470337
Uso de memória: 19021824
----------------------
Algoritmo de Kadane
----------------------
Tempo de execução: 0.043852806091308594
Uso de memória: 19132416
----------------------
Algoritmo de Divisão e Conquista
----------------------
Tempo de execução: 0.02790999412536621
Uso de memória: 19263488
----------------------
```

No exemplo abaixo foi utilizado um conjunto de 5000 exemplos de lista de números de tamanho entre 5 e 1000.

```
Força bruta
----------------------
Tempo de execução: 528.1048517227173
Uso de memória: 40951808
----------------------
Algoritmo de Kadane
----------------------
Tempo de execução: 0.4306783676147461
Uso de memória: 40988672
----------------------
Algoritmo de Divisão e Conquista
----------------------
Tempo de execução: 0.251314640045166
Uso de memória: 41160704
----------------------
```

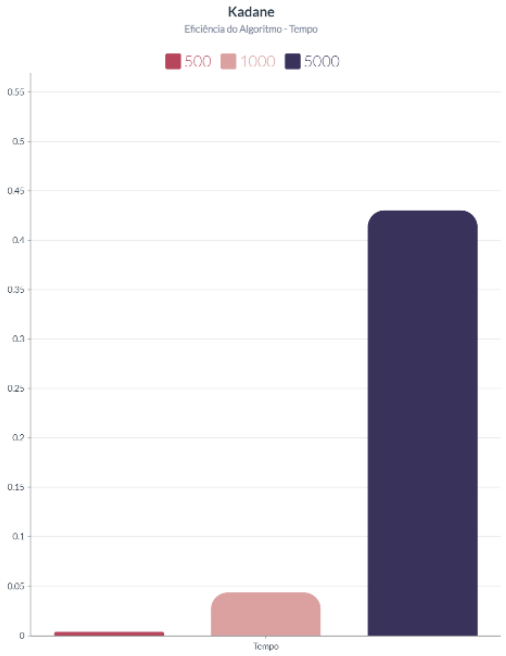


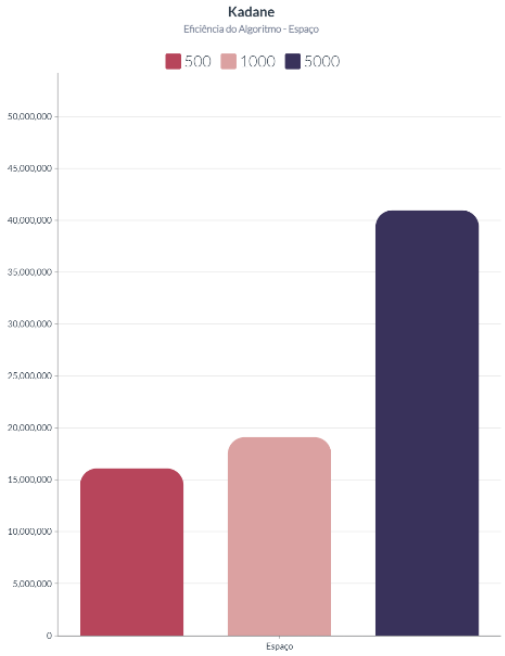
Olhamos também que com o aumento da entrada o tamanho do espaço utilizado mante um aumento “estável” enquanto o tempo de execução altera drasticamente para algoritmos como o força bruta.

Assim, observamos o quanto é gritante a diferença entre o tempo dos algoritmos.

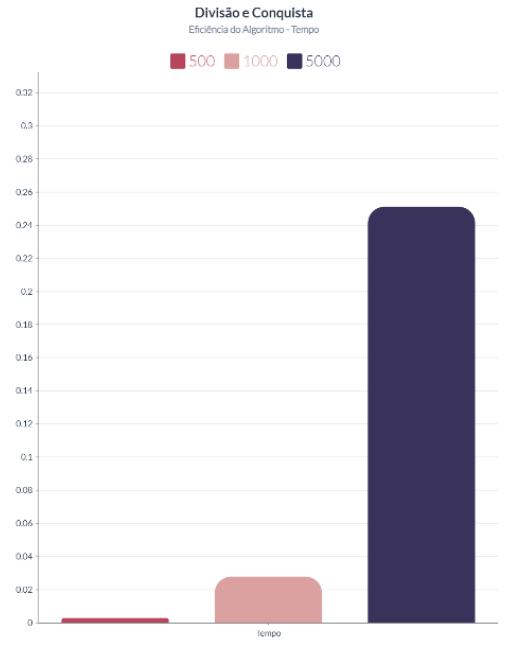
Agora vamos observá-los de forma individual

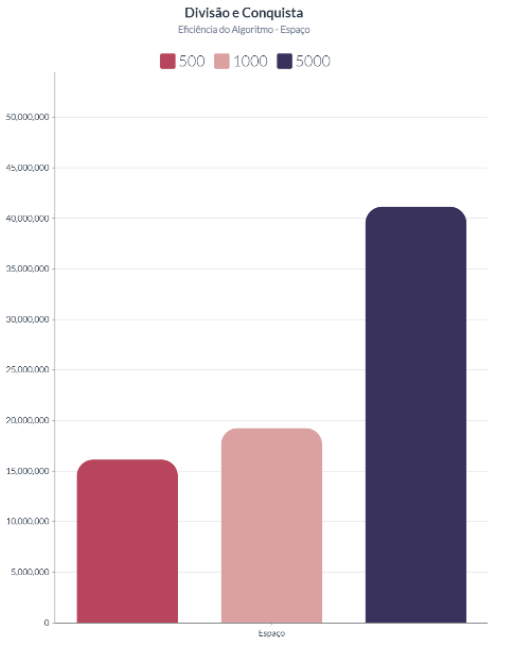
O método de Força Bruta se destacou pela sua simplicidade, porém, exibiu um desempenho consideravelmente inferior em relação ao tempo de execução em todos os testes realizados. Além disso, sua demanda de memória foi significativamente mais alta, especialmente em conjuntos de dados extensos, o que pode limitar sua aplicabilidade em contextos com restrições de recursos.





Em contrapartida, o Algoritmo de Kadane demonstrou ser uma opção mais eficiente. Apresentou consistentemente o menor tempo de execução e uso de memória na maioria dos casos de teste. Sua capacidade de lidar eficazmente com conjuntos de dados maiores e a eficiência em termos de recursos o posicionam como uma escolha preferencial em muitos contextos.





Por fim, o Algoritmo de Divisão e Conquista mostrou um desempenho intermediário entre os algoritmos avaliados. Embora tenha sido mais eficiente que a abordagem de Força Bruta, ainda

ficou aquém do desempenho do Algoritmo de Kadane em termos de tempo de execução e uso de memória.

Em suma, os resultados apontam o Algoritmo de Kadane como a escolha mais promissora para a busca da maior subsequência crescente, principalmente em cenários onde a eficiência é crucial, especialmente em conjuntos de dados extensos. No entanto, a seleção do algoritmo mais adequado pode variar de acordo com as necessidades específicas de cada aplicação e as limitações de recursos.

Observação: Logo abaixo apresento meu código de força bruta e kadane em sequência.

```python
def find_max_sequence_force_brute(sequence):
    max_length = 0
    max_subsequence = []

    for i in range(len(sequence)):
        for j in range(i, len(sequence)):
            subsequence = sequence[i:j+1]

            is_sequence = True

            for k in range(len(subsequence)-1):
                if subsequence[k] >= subsequence[k+1]:
                    is_sequence = False
                    break

            if is_sequence and len(subsequence) > max_length:
                max_length = len(subsequence)
                max_subsequence = subsequence

    return max_subsequence if max_length > 0 else sequence
```

```python
def update_max_subsequence_Kadane(current_max, max_subsequence):
    if len(current_max) > len(max_subsequence):
        return list(current_max)
    else:
        return list(max_subsequence)

def find_max_sequence_Kadane(sequence):
    current_max = max_subsequence = [sequence[0]]

    for i in range(1, len(sequence)):
        if sequence[i] > current_max[-1]:
            current_max.append(sequence[i])
        else:
            current_max = [sequence[i]]

        max_subsequence = update_max_subsequence_Kadane(current_max, max_subsequence)

    return max_subsequence
```