TRABALHO PRÁTICO FPAA

Nomes: Mateus Pedreira Oliveira, Igor Rogério Soares Vasconcelos, Enzo Miranda Terenzi

Exercício 1:

Para a execução do primeiro exercício construímos dois programas em python para realizar cada um dos itens desejados, ordenação por ordem alfabética e por repetição. Dentro dos dois códigos utilizamos o método de divisão e conquista para atingir o resultado final.

ORDENAÇÃO POR ORDEM ALFABÉTICA

Código da divisão e conquista:

```
def merge_sort(palavras):
    if len(palavras) <= 1:
        return palavras

    mid = len(palavras) // 2
    left = merge_sort(palavras[:mid])
    right = merge_sort(palavras[mid:])

    merged = merge(left, right)
    return merged


def merge(left, right):
    merged = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i].lower() < right[j].lower():
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1

    while i < len(left):
        merged.append(left[i])
        i += 1

    while j < len(right):
        merged.append(right[j])
        j += 1

    return merged
```

Dentro desse código operamos um merge sort que tem como objetivo dividir nosso arquivo de palavras e realizar a comparação de tamanho entre as palavras. O tamanho das palavras é mensurado baseada na ordem alfabética que as palavras se encontram.

Um exemplo a ser observado dentro do arquivo de saída é que após a execução do programa temos também identificado as palavras que possuem apenas uma letra e não são excluídas palavras com repetição.

Para uma melhor visualização do arquivo de saída temos um link para o arquivo presente no github do projeto.

Arquivo de saída ordem alfabética.

ORDENAÇÃO POR FREQUENCIA

Código divisão e conquista

```
def merge_sort(palavras):
    if len(palavras) <= 1:
        return palavras


    mid = len(palavras) // 2
    left = merge_sort(palavras[:mid])
    right = merge_sort(palavras[mid:])

    merged = merge(left, right)
    return merged


def merge(left, right):
    merged = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i][1] < right[j][1] or (left[i][1] == right[j][1] and
left[i][0] <= right[j][0]):
            merged.append(left[i])
            i += 1
        else:
            merged.append(right[j])
            j += 1

    while i < len(left):
        merged.append(left[i])
        i += 1
```

```
    while j < len(right):
        merged.append(right[j])
        j += 1

    return merged

with open('final_Freq.txt', 'w') as output_file:
    for word, count in frequencia_palavras:
        for _ in range(count):
            print(word, file=output_file)
```

Assim como no código anterior temos um merge sort para fazer a divisão do arquivo e comparar o conteúdo. No entanto temos a adição de um contador que irá fazer o cálculo de repetição das palavras e assim no arquivo final teremos as palavras em ordem de repetição daquela que tem menos até a que tem mais repetição.

Para uma melhor visualização do arquivo de saída temos um link para o arquivo presente no github do projeto.

Arquivo de saída ordem de frequência.

ANÁLISE DE COMPLEXIDADE DOS CÓDIGOS

1. Divisão: A função `merge_sort` divide a lista de palavras pela metade até que cada divisão contenha apenas um elemento. Esse processo de divisão ocorre no máximo log2(n) vezes, onde "n" é o número de palavras. A cada divisão, a lista é dividida ao meio, reduzindo seu tamanho pela metade a cada etapa.
2. Combinação: Depois de dividir a lista em partes menores, o algoritmo começa a mesclar as partes ordenadas. Ele compara os elementos das duas metades ordenadas e os insere em ordem na lista final. A quantidade de comparações necessárias depende do tamanho das duas metades. Como cada metade tem aproximadamente a metade do tamanho da lista original, a complexidade é linear em relação ao tamanho da lista, ou seja, O(n).

Portanto, considerando tanto a divisão quanto a combinação, o tempo de execução total do Merge Sort é dado pela multiplicação do tempo de execução para cada etapa:

Tempo de execução do Merge Sort = Tempo de divisão * Tempo de combinação

O tempo de divisão é O(log n) devido ao número de divisões (log2(n)), enquanto o tempo de combinação é O(n) porque leva tempo linear para mesclar as metades. Assim, o tempo de execução total do Merge Sort é O(n log n).

MÉTRICAS DAS EXECUÇÕES

As execuções de cada programa foram realizadas 3 vezes em sequência utilizando um processador AMD Ryzen 7 5700X e foram obtidos os seguintes resultados:

Ordenação por ordem alfabética:

Execution time 100k: 0.34 seconds

Execution time 100k: 0.34 seconds

Execution time 100k: 0.34 seconds

Média = 0.34

Desvio Padrão = 0

Execution time 200k: 0.75 seconds

Execution time 200k: 0.74 seconds

Execution time 200k: 0.73 seconds

Média = 0.74

Desvio Padrão = 0.01000000000

Execution time 300k: 1.13 seconds

Execution time 300k: 1.14 seconds

Execution time 300k: 1.17 seconds

Média = 1.4666

Desvio Padrão = 0.02081666000

Execution time 400k: 1.63 seconds

Execution time 400k: 1.56 seconds

Execution time 400k: 1.59 seconds

Média = 1.59333

Desvio Padrão = 0.03511884585

Execution time 500k: 2.01 seconds

Execution time 500k: 1.97 seconds

Execution time 500k: 2.00 seconds

Média = 1.99333

Desvio Padrão = 0.02081666000

Execution time 600k: 2.45 seconds

Execution time 600k: 2.44 seconds

Execution time 600k: 2.44 seconds

Média = 2.44333

Desvio Padrão = 0.005773502692

Execution time 700k: 2.89 seconds

Execution time 700k: 2.91 seconds

Execution time 700k: 2.91 seconds

Média = 2.90333

Desvio Padrão = 0.01154700539

Ordenação por frequência:

Execution time 100k: 0.07 seconds

Execution time 100k: 0.07 seconds

Execution time 100k: 0.07 seconds

Média = 0.7

Desvio Padrão = 0

Execution time 200k: 0.13 seconds

Execution time 200k: 0.13 seconds

Execution time 200k: 0.13 seconds

Média = 0.13

Desvio Padrão = 0

Execution time 300k: 0.22 seconds

Execution time 300k: 0.20 seconds

Execution time 300k: 0.20 seconds

Média = 0.20666

Desvio Padrão = 0.01154700539

Execution time 400k: 0.28 seconds

Execution time 400k: 0.27 seconds

Execution time 400k: 0.26 seconds

Média = 0.27

Desvio Padrão = 0.01000000000

Execution time 500k: 0.35 seconds

Execution time 500k: 0.33 seconds

Execution time 500k: 0.33 seconds

Média = 0.33666

Desvio Padrão = 0.33666

Execution time 600k: 0.42 seconds

Execution time 600k: 0.41 seconds

Execution time 600k: 0.41 seconds

Média = 0.41333

Desvio Padrão = 0.005773502692

Execution time 700k: 0.49 seconds

Execution time 700k: 0.47 seconds

Execution time 700k: 0.47 seconds

Média = 0.47666

Desvio Padrão = 0.01154700539

Exercício 02

Para a execução do segundo exercício construímos um programa em python para realizar a execução da atividade. Esse programa realizará um merge sort para separar as palavras do arquivo *palavras_100k.txt* e separar aquelas que possuem maior frequência.

Após achar as palavras mais frequentes executamos uma função que tem como objetivo resolver o problema da mochila. Executados as duas funções principais temos o output dos pesos e valores máximos junto com um gráfico que é feito a partir do código utilizando a biblioteca *Matplotlib.*

FUNÇÃO DE MERGE SORT

```
def merge_sort(palavras):
    if len(palavras) <= 1:
        return palavras


    meio = len(palavras) // 2
    esquerda = merge_sort(palavras[:meio])
    direita = merge_sort(palavras[meio:])


    mesclado = merge(esquerda, direita)
    return mesclado


def merge(esquerda, direita):
    mesclado = []
    i = j = 0


    while i < len(esquerda) and j < len(direita):
        if esquerda[i][1] < direita[j][1] or (esquerda[i][1] == direita[j][1]
and esquerda[i][0] <= direita[j][0]):
            mesclado.append(esquerda[i])
            i += 1
        else:
            mesclado.append(direita[j])
            j += 1


    while i < len(esquerda):
        mesclado.append(esquerda[i])
        i += 1

    while j < len(direita):
        mesclado.append(direita[j])
        j += 1

    return mesclado
```

FUNÇÃO PARA CHAMAR O MERGE SORT E EXECUTAR O PROBLEMA DA MOCHILA

```
def palavras_mais_frequentes(caminho_arquivo, k=50):

    with open(caminho_arquivo, 'r') as arquivo:
        conteudo = arquivo.read()
        palavras = conteudo.split()
```

```
    contador = collections.Counter(palavras)
    frequencia_palavras = [(palavra, frequencia) for palavra, frequencia in
contador.items()]


    palavras_ordenadas = merge_sort(frequencia_palavras)


    palavras_mais_frequentes = palavras_ordenadas[:k]
    return palavras_mais_frequentes


def resolver_problema_mochila(palavras, pesos, valores, peso_maximo):
    n = len(palavras)


    tabela = [[0] * (peso_maximo + 1) for _ in range(n + 1)]

    for i in range(1, n + 1):
        for j in range(1, peso_maximo + 1):
            if pesos[i - 1] > j:

                tabela[i][j] = tabela[i - 1][j]
            else:

                tabela[i][j] = max(tabela[i - 1][j], valores[i - 1] + tabela[i
- 1][j - pesos[i - 1]])


    return tabela[n][peso_maximo]
```

MÉTRICA DE EXECUÇÃO

Peso máximo: 50 Valor máximo: 50

Peso máximo: 51 Valor máximo: 51

Peso máximo: 52 Valor máximo: 51

Peso máximo: 53 Valor máximo: 53

Peso máximo: 54 Valor máximo: 53

Peso máximo: 55 Valor máximo: 53

Peso máximo: 56 Valor máximo: 55

Peso máximo: 57 Valor máximo: 56

Peso máximo: 58 Valor máximo: 56

Peso máximo: 59 Valor máximo: 57

Peso máximo: 60 Valor máximo: 58

Peso máximo: 61 Valor máximo: 58

Peso máximo: 62 Valor máximo: 60

Peso máximo: 63 Valor máximo: 60

Peso máximo: 64 Valor máximo: 61

Peso máximo: 65 Valor máximo: 62

Peso máximo: 66 Valor máximo: 63

Peso máximo: 67 Valor máximo: 63

Peso máximo: 68 Valor máximo: 64

Peso máximo: 69 Valor máximo: 65

Peso máximo: 70 Valor máximo: 65

Peso máximo: 71 Valor máximo: 66

Peso máximo: 72 Valor máximo: 67

Peso máximo: 73 Valor máximo: 68

Peso máximo: 74 Valor máximo: 69

Peso máximo: 75 Valor máximo: 69

Peso máximo: 76 Valor máximo: 70

Peso máximo: 77 Valor máximo: 71

Peso máximo: 78 Valor máximo: 72

Peso máximo: 79 Valor máximo: 72

Peso máximo: 80 Valor máximo: 73

ANÁLISE DE COMPLEXIDADE DO CÓDIGO

merge_sort(palavras): O algoritmo de ordenação utilizado é o merge sort. A complexidade do merge sort é O(n log n), onde n é o número de palavras.

palavras_mais_frequentes(caminho_arquivo, k): A leitura do arquivo e a contagem das palavras possuem complexidade O(n), onde n é o número total de palavras no arquivo. A ordenação das palavras utilizando o merge sort possui complexidade O(n

log n). Portanto, a complexidade total dessa função é O(n log n), onde n é o número total de palavras.

resolver_problema_mochila(palavras, pesos, valores, peso_maximo): O algoritmo utilizado para resolver o Problema da Mochila Sem Repetições é baseado em programação dinâmica. A complexidade desse algoritmo é O(nW), onde n é o número de palavras e W é o peso máximo.

Loop for peso_maximo in peso_maximo_lista:

Criação da tabela tabela com dimensões (n+1) x (peso_maximo+1), o que tem complexidade O(nW), onde n é o número de palavras e W é o peso máximo.

Preenchimento da tabela com complexidade O(nW), onde n é o número de palavras e W é o peso máximo.

Portanto, a complexidade total do código se dá pela complexidade do passo 2 (palavras_mais_frequentes) e do passo 4 (resolver_problema_mochila), que são ambos O(n log n) e O(nW). Onde n é o número total de palavras e W é o peso máximo.

GRÁFICO DE EXECUÇÃO

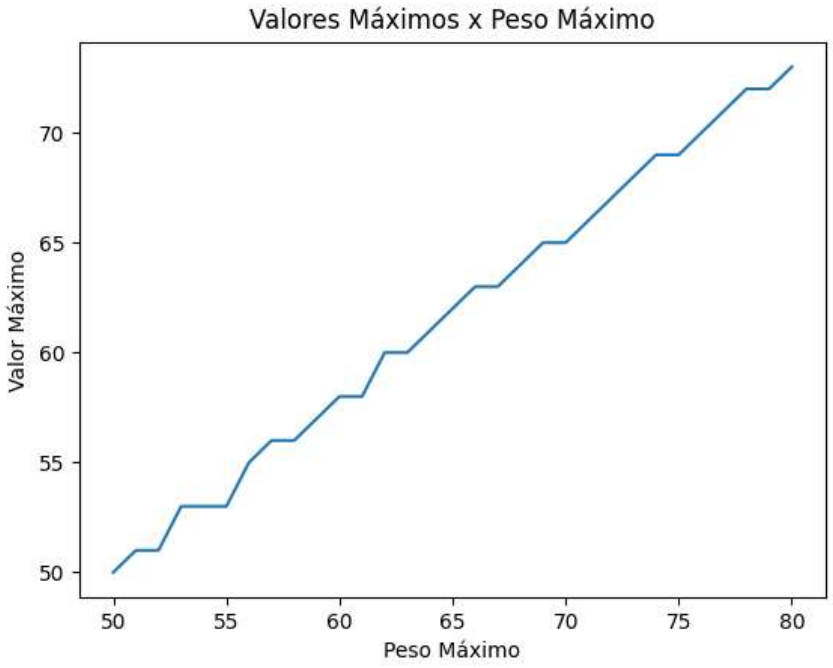

Gráfico gerado pelo próprio código através da biblioteca Matplotlib.