Trabalho 1

UC: Estruturas de Dados e Algoritmos
Abril de 2023

INSTITUTO UNIVERSITÁRIO DE LISBOA

Licenciatura em Ciência de Dados
CDA2

Gonçalo Girão 111515
João Magarça 111640

ÍNDICE

Introdução

O trabalho em questão foi elaborado seguindo as orientações da unidade curricular de Estruturas de Dados e Algoritmos, da Licenciatura em Ciência de Dados do ISCTE.

Este primeiro trabalho tem como objetivo encontrar o valor máximo e mínimo de uma lista de valores com 100 000 observações. A resolução deste problema deve ter em vista a complexidade temporal de algoritmos, de modo a resolver o problema da forma mais eficiente possível tendo em conta a visão lecionada na UC "divide to conquer".

Bibliotecas

Em primeiro lugar decidimos importar algumas packages que considerámos necessárias para a resolução dos problemas:

```
import pandas as pd
import time
import matplotlib.pyplot as plt
```

Partitions

No desenvolvimento da primeira função partitions são pedidos 2 argumentos, a base de dados (data) e o número de observações por partição (n). Dividindo o tamanho do ficheiro por n, descobrimos em quantas partes é dividido o ficheiro. Depois, através de um "ciclo for" lemos o ficheiro e selecionando a primeira coluna através do comando iloc(a única com dados em cada linha), adicionamos os valores a uma lista. Depois o método yield devolve-nos um iterador sobre as várias partições do ficheiro.

```
def partitions(data, n):
    part = int(len(pd.read_csv(data))) // n
    for k in range(0, part):
        bd = pd.read_csv(data, skiprows=(k)*n,  nrows=n)
        bd = bd.iloc[:, 0].tolist()
        yield bd
```

Sorting

Ao desenvolver a função sorting (list, ascending) que recebe uma lista de valores não ordenados e devolve a lista ordenada, o número de iterações que o algoritmo levou a concluir a ordenação, o mínimo e o máximo da lista.

Primeiramente procuramos o algoritmo mais eficiente usado em aula para ordenar uma lista, verificámos o Merge seria o mais eficiente.

Algoritmos de ordenação – complexidade

Sorting Big-O Cheat Sheet			
Algoritmo	Worst Case	Best Case	Average Case
Insertion	$O(n^2)$	$O(n)$	$O(n^2)$
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quicksort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$

A função sorting começa por verificar se o tamanho da lista "data" é menor ou igual a 1. Se for, a lista já está ordenada e a função retorna a própria lista, número de iterações, valor mínimo e máximo da lista.

Caso a lista tenha mais de um elemento, a função divide a lista em duas partes iguais e chama a função "sorting" recursivamente para cada uma dessas partes. A chamada recursiva retorna a lista ordenada, o número de iterações realizadas, o valor mínimo e o valor máximo da lista.

Em seguida, a função combina as duas partes ordenadas usando a função "merge_ascending" se "ascending" for True, ou "merge_decreasing" se "ascending" for False. A função "merge_ascending" combina as duas partes ordenadas em ordem crescente. Criando uma lista vazia e, em seguida, comparando os elementos das duas partes ordenadas. Os elementos são adicionados à lista em ordem crescente até que todas as listas tenham sido percorridas. A lista já ordenada é então retornada, juntamente com o número de iterações realizadas, o valor mínimo e o valor máximo da lista.

```python
def sorting(data, ascending, iteracoes = 0):
    ini = 0
    fim = len(data)
    if len(data) <= 1:
        return data, iteracoes, data[0], data[0]
    meio = (ini+fim) // 2
    esquerda, iteracoes_esquerda, min_esquerda, max_esquerda = sorting(data[:meio], ascending)
    direita, iteracoes_direita, min_direita, max_direita = sorting(data[meio:], ascending)
    iteracoes += iteracoes_esquerda + iteracoes_direita
    minimo = min(min_esquerda, min_direita)
    maximo = max(max_esquerda, max_direita)

    if ascending == True:
        return merge_ascending(esquerda, direita, iteracoes, minimo, maximo)
    if ascending == False:
        return merge_decreasing(esquerda, direita, iteracoes, minimo, maximo)
```

O parâmetro ascending permite ordenar a lista por ordem crescente ou decrescente. Para isso desenvolvemos duas funções distintas para ordenar a lista de forma crescente(merge_ascending) e de forma decrescente(merge_decreasing).

```python
def merge_ascending(esquerda, direita, iteracoes, minimo, maximo):
    lista_ordenada = []
    i = 0
    j = 0
    while i < len(esquerda) and j < len(direita):
        if esquerda[i] < direita[j]:
            lista_ordenada.append(esquerda[i])
            i += 1
        else:
            lista_ordenada.append(direita[j])
            j += 1
        iteracoes += 1
    lista_ordenada += esquerda[i:]
    lista_ordenada += direita[j:]
    minimo = min(minimo, lista_ordenada[0])
    maximo = max(maximo, lista_ordenada[-1])
    return lista_ordenada, iteracoes, minimo, maximo
```

```python
def merge_decreasing(esquerda, direita, iteracoes, minimo, maximo):
    lista_ordenada = []
    i = 0
    j = 0
    while i < len(esquerda) and j < len(direita):
        if esquerda[i] > direita[j]:
            lista_ordenada.append(esquerda[i])
            i += 1
        else:
            lista_ordenada.append(direita[j])
            j += 1
        iteracoes += 1
    lista_ordenada += esquerda[i:]
    lista_ordenada += direita[j:]
    minimo = min(minimo, lista_ordenada[-1])
    maximo = max(maximo, lista_ordenada[0])
    return lista_ordenada, iteracoes, minimo, maximo
```

Execute

Na alínea c) foi desenvolvida a função execute. Esta que para cada partição (através do ciclo for)calcula, com o auxílio das funções partitions e sorting:
O tempo de execução da iteração (através do package time);
O máximo;
O mínimo;
O número de iterações.
Esta função recebe a base de dados, o critério
de ordenação e o número de observações por partição. Esta função itera sobre as várias partições do ficheiro, sendo que, para cada uma são ordenados os seus valores numéricos. O output da função é uma tabela(pd.DataFrame) com o máximo, o mínimo e o número de iterações que o sorting executa de cada partição do ficheiro.
Começa por criar as listas vazias de máximos, mínimos, tempos e iterações.
Depois com um iterador nas partições, utilizamos a função sorting(iterador, ascending) para definir o número de iterações, o mínimo e o máximo para cada partição que são adicionados às respetivas listas.

No fim essas são dispostas nas colunas do data frame com os respetivos nomes

```python
def execute(data, n, ascending=True):
    maximos = []
    minimos = []
    tempos = []
    iteracoes = []
    for p in partitions(data, n):
        ini = time.time()
        ord, itr, minimo, maximo = sorting(p, ascending)
        fim = time.time()
        tempo = fim - ini
        maximos.append(maximo)
        minimos.append(minimo)
        tempos.append(tempo)
        iteracoes.append(itr)
    resultados = pd.DataFrame({'Partições':[i for i in range(1, len(maximos)+1)],
                  'Máximo':maximos,
                  'Minimo':minimos,
                  'Tempo':tempos,
                  'Iterações': iteracoes})
    return resultados
```

Assim é retornada a tabela por exemplo para 1000 observações por partição:

```
Python Console
>>> execute("data.csv",1000,True)
    Partições   Máximo   Minimo     Tempo  Iterações
0           1  9993167 -9998182  0.003006       8700
1           2  9952319 -9987211  0.002527       8686
2           3  9983445 -9958315  0.003518       8677
3           4  9982899 -9961840  0.002084       8690
4           5  9997938 -9984509  0.003996       8703
..        ...      ...      ...       ...        ...
95         96  9988582 -9925964  0.005054       8696
96         97  9994785 -9987263  0.007018       8700
97         98  9993301 -9977318  0.005742       8698
98         99  9954747 -9957233  0.005643       8702
99        100  9999991 -9991902  0.003625       8711

[100 rows x 5 columns]
```

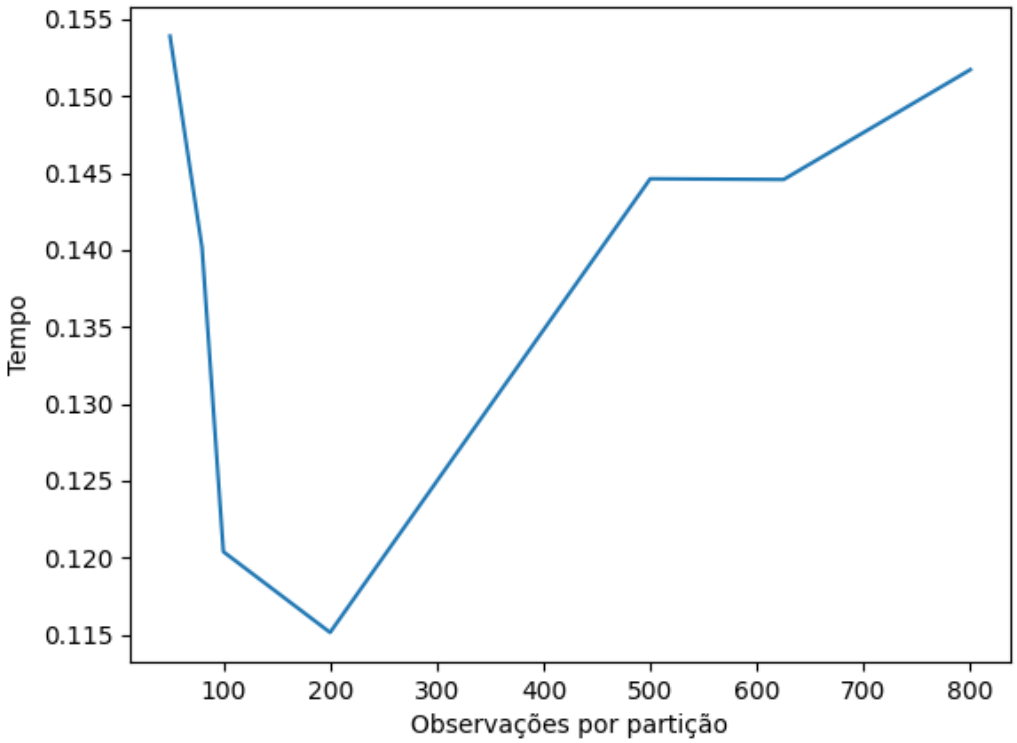
Experiências/Conclusão

Com a criação desta função teste, procurámos experimentar essencialmente a relação entre o número de observações por partição (n) e o tempo médio de execução da função sorting em cada partição (de n observações).

O tempo de execução para cada n é a soma dos tempos das listas com n observações por partição na função “sorting”, que são extraídos da 4ª coluna do data frame resultante da função execute(resultados) onde são posteriormente armazenados numa lista

```python
def teste(data):
    valores = [100, 200, 500, 625, 800]
    tempos = []
    for n in valores:
        resultados = execute(data, n)
        media_tempos = sum(resultados.iloc[:, 3].tolist())
        tempos.append(media_tempos)

    plt.plot(valores, tempos)
    plt.xlabel('Observações por partição')
    plt.ylabel('Tempo')
    plt.show()
```



Verifica-se então que, da lista de valores observados, no intervalo de observações 50-200 o intervalo de tempo diminui à medida que o número de observações por partição aumenta, sendo o n=200 o valor ideal de observações para ordenar cada partição.
Por outro lado, o valor 50 parece o menos eficiente para solucionar o problema.
Quanto ao segundo intervalo(>200), à medida que as observações por partição aumentam, os intervalos de tempo também aumentam, sendo cada vez menos eficientes.