

**Algoritmos e Estruturas de Dados   
2ª Série**

**(Problema)**

|  |  |
| --- | --- |
|  |  |

**Ordenar Ficheiros de dimensão arbitrária**

|  |  |
| --- | --- |
| Nº 52977 | Guilherme Dias |
| Nº 52778 | Tomás Cheng |

Nº52766 Álvaro Reis

Licenciatura em Engenharia Informática e de Computadores  
Semestre de Verão 2024/2025

16/05/2025

**Índice**

[**Introdução 3**](#_vejbrp2lkp95)

[**Operações entre coleções de pontos no plano 4**](#_bykdeynkhr2j)

[**Análise do problema 5**](#_t4rhrwd6z48p)

[Implementações 6](#_yoni7ha4fzsw)

[Implementação 1 6](#_pjwbq29epszi)

[Representação de Dados 7](#_i9gxbnl8iptp)

[Implementação 2 8](#_asn5vj7r5kwp)

[Representação de Dados 9](#_4rtyp0u67qvc)

[Estruturas de Dados 10](#_u6gj8il84tr4)

[Algoritmos e análise da complexidade 13](#_jo982twjsz1u)

[**Avaliação Experimental 17**](#_1cfhylcj5rcg)

[Implementação 1 18](#_e0n7ua2wk0ea)

[Implementação 2 19](#_8uhp8hy87p30)

[Resultados 20](#_jy5brj8zk4cy)

[**Conclusões 21**](#_m16tzoqeksob)

[**Referências 22**](#_6p9ewsdpimjx)

# 

# **Introdução**

Este trabalho foi desenvolvido no âmbito da unidade curricular de Algoritmos e Estruturas de Dados (AED), do curso de Licenciatura em Engenharia Informática e de Computadores do Instituto Superior de Engenharia de Lisboa, durante o semestre de verão de 2024/2025.  
Este trabalho aborda o desenvolvimento de uma aplicação para processamento de coleções de pontos no plano, implementando operações fundamentais de conjuntos como união, interseção e diferença. O problema foi resolvido através de duas abordagens distintas:

Na implementação 1 utilizámos as estruturas de dados fornecidas pela biblioteca padrão da linguagem *Kotlin* enquanto que na implementação 2 utilizamos uma tabela de dispersão (*hash map*) desenvolvida especificamente para este projeto

O sistema processa ficheiros de texto contendo coordenadas de pontos no plano (formato .co) e permite executar operações entre dois conjuntos de pontos carregados.

# **Operações entre coleções de pontos no plano**

O problema da série dois tem como objetivo fazer operações entre coleções de pontos no plano. Para conseguirmos chegar ao objetivo da resolução deste problema temos de ter em conta que cada colecção de pontos está descrita em um ficheiro de texto, assim cada ponto no ficheiro é descrito por um identificador de coordenadas, sendo elas x e y. Para fazer a aplicação *ProcessPointsCollections* deve ter as funcionalidades de:

- Receber como parâmetro dois ficheiros de texto, com uma extensão .co;

- Permitir a produção de um ficheiro, com uma extensão .co, que tem os pontos, sem repetições, que repetem pelo menos um dos ficheiros de entrada, sendo essa a operação de união (*union*);

- Produzir outro ficheiro, com uma extensão .co, que contém os pontos comuns entre os ficheiros de entrada, a operação interseção (*intersection*);

- Permite produzir outro novo ficheiro, com uma extensão .co, que contém os pontos únicos que estejam presentes apenas em um dos ficheiros *input*, a operação diferença (*difference*)

Os parâmetros de execução da aplicação *ProcessPointsCollections* devem processar os comandos de:

* *load* dos ficheiros *document1.co* e *document2.co*, que carrega os pontos dos dois ficheiros num *hash map* ou tabela de dispersão, que é estruturado de forma a permitir a consulta eficiente para responder a todas as operações subsequentes.
* *union* que como já explicado acima faz um ficheiro *output.co* que contém os pontos presentes em pelo menos um dos ficheiros de entrada, sem repetições.
* *intersection* que também como explicado acima, fabrica um ficheiro *output.co* que contém os pontos que ocorrem em ambos os ficheiros de entrada, sem repetições.
* *difference*, que faz o ficheiro *output.co* que tem os pontos que estão no ficheiro *document1.co* mas que não estão no ficheiro *document2.co*, sem repetições.
* por fim *exit*, que termina a aplicação

O formato dos ficheiros de extensão .co apresentam as coordenadas com caráter ‘v’, de seguida o id do ponto, e as coordenadas x e y.

# Análise do problema

O problema central consiste em manipular e comparar conjuntos de pontos no plano, representados por identificadores e coordenadas (X, Y), contidos em dois ficheiros de texto com extensão .co.   
O sistema a desenvolver deve carregar os dados de ambos os ficheiros e permitir realizar operações de união, interseção e diferença entre os conjuntos de pontos, exportando os resultados para novos ficheiros, também com extensão .co.  
Para uma execução eficiente dessas operações, é essencial utilizar uma estrutura de dados que permita acesso rápido e inserção eficiente.  
Assim, o problema exige o uso de uma tabela de dispersão (*hash map*) personalizada, implementada do zero, sem recorrer às bibliotecas padrão da linguagem *Kotlin* na segunda fase do trabalho.

## Implementações

### Implementação 1

A Implementação 1 do projeto tem como objetivo resolver o problema proposto de manipulação de coleções de pontos no plano, utilizando exclusivamente as estruturas de dados fornecidas pela biblioteca do *Kotlin*.  
Esta implementação corresponde à primeira fase do trabalho, onde é permitido recorrer a estas estruturas prontas da linguagem.  
Esta implementação permite ao utilizador executar operações sobre coleções de pontos no plano, lidos a partir de dois ficheiros de entrada.  
É realizado as seguintes operações:

* *load*: carrega os dois ficheiros de pontos;
* *union*: cria um novo ficheiro com a união dos pontos dos dois ficheiros;
* *intersection*: cria um novo ficheiro com os pontos comuns;
* *difference*: cria um novo ficheiro com os pontos únicos do primeiro ficheiro;
* *exit*: termina o programa.

Com esta implementação é possível:

* Carregar dois ficheiros de pontos do tipo .co;
* Efetuar operações entre os pontos de ambos os ficheiros: união, interseção e diferença;
* Guardar os resultados num novo ficheiro de saída;
* Terminar a aplicação com o comando exit.

A função principal da classe é a *run*(), que implementa um ciclo de interação com o utilizador. Existem cinco comandos principais sendo eles:

* *load* (ficheiro 1) (ficheiro 2): carrega dois ficheiros com pontos;
* *union* resultado.co: guarda num ficheiro todos os pontos de ambos os ficheiros (sem repetições);
* *intersection* resultado.co: guarda os pontos que existem simultaneamente nos dois ficheiros;
* *difference* resultado.co: guarda os pontos que existem apenas no primeiro ficheiro;
* *exit*: termina a aplicação.

#### Representação de Dados

Os pontos são representados por uma classe chamada Ponto, definida como uma *data class*. Esta classe tem três atributos:

* id: identificador do ponto
* x: coordenada X (inteiro);
* y: coordenada Y (inteiro).

Ao utilizar conjuntos, assegura-se que não existem pontos duplicados, e que a verificação de existência é eficiente (complexidade próxima de O(1)).

### Implementação 2

A Implementação 2 tal como a implementação 1 tem como objetivo resolver o problema proposto de manipulação de coleções de pontos no plano, mas utilizando exclusivamente as estruturas de dados feitas no ponto 1.4.

Esta implementação permite ao utilizador executar operações sobre coleções de pontos no plano, lidos a partir de dois ficheiros de entrada.

É realizado as operações *load, union, intersection, difference e exit*.

Com esta implementação é possível:

* Carregar dois ficheiros de pontos do tipo .co;
* Efetuar operações entre os pontos de ambos os ficheiros: união, interseção e diferença;
* Guardar os resultados num novo ficheiro de saída;
* Terminar a aplicação com o comando *exit*.

A função principal da classe é a *run*(), que implementa um ciclo de interação com o utilizador. Existem cinco comandos principais sendo eles:

* *load* (ficheiro 1) (ficheiro 2): carrega dois ficheiros com pontos;
* *union* resultado.co: guarda num ficheiro todos os pontos de ambos os ficheiros (sem repetições);
* *intersection* resultado.co: guarda os pontos que existem simultaneamente nos dois ficheiros;
* *difference* resultado.co: guarda os pontos que existem apenas no primeiro ficheiro;
* *exit*: termina a aplicação.

#### 

#### Representação de Dados

Os pontos são representados por uma classe chamada Ponto, definida como uma data class. Esta classe tem três atributos:

* id: identificador do ponto
* x: coordenada X (inteiro);
* y: coordenada Y (inteiro).

Ao utilizar conjuntos, assegura-se que não existem pontos duplicados, e que a verificação de existência é eficiente (complexidade próxima de O(1)).

## Estruturas de Dados

A principal estrutura de dados utilizada para a resolução do problema foram os *Hash Maps*. *Hash Maps* armazena dados em pares chave-valor. Para conseguirmos armazenar esses valores utilizamos as tabelas de dispersão que são baseadas na distribuição de chaves num array unidimensional T[0..M-1], a distribuição é feita através de cálculos de cada uma das chaves, o valor de uma função com predefinição h, ou seja de uma função de dispersão, por sua vez essa função entrega a cada chave, um inteiro entre 0 e M-1, designado endereço de dispersão.

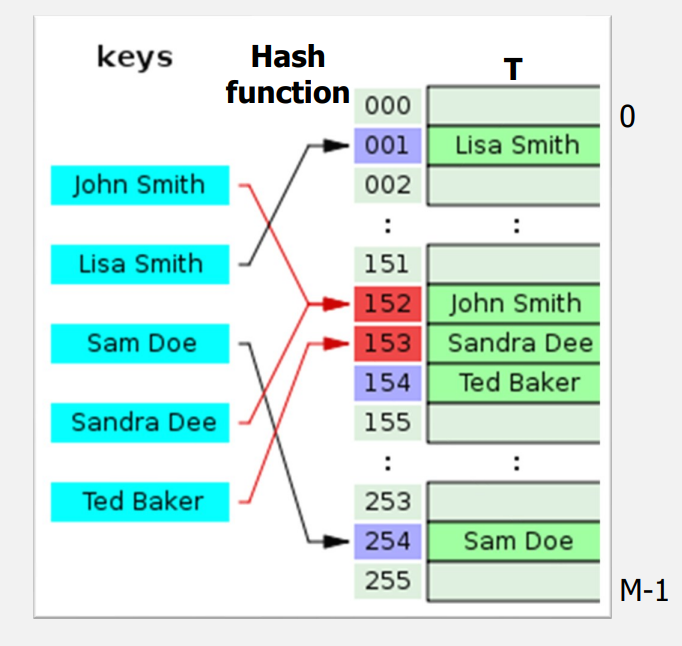


Imagem 1- Exemplo de uma tabela de dispersão presente na

parte 12 disponibilizado no *Moodle* na UC de AED

Em caso gerais a função de dispersão transforma a chave num inteiro de 0 a M-1 sendo M a dimensão da tabela de dispersão, distribuindo o melhor possível as chaves pelas posições livres da tabela de dispersão, sendo esse M de preferência ímpar, tendo cálculo simples. Idealmente as funções de dispersão devem ter como probabilidade de ocorrer o mesmo valor para duas chaves distintas de 1/M.

Para adicionar elementos à *Hash table* (tabela de dispersão), colocamos cada uma das chaves numa posição calculada em função do seu valor, de forma independente da ordem de chegada dos elementos.

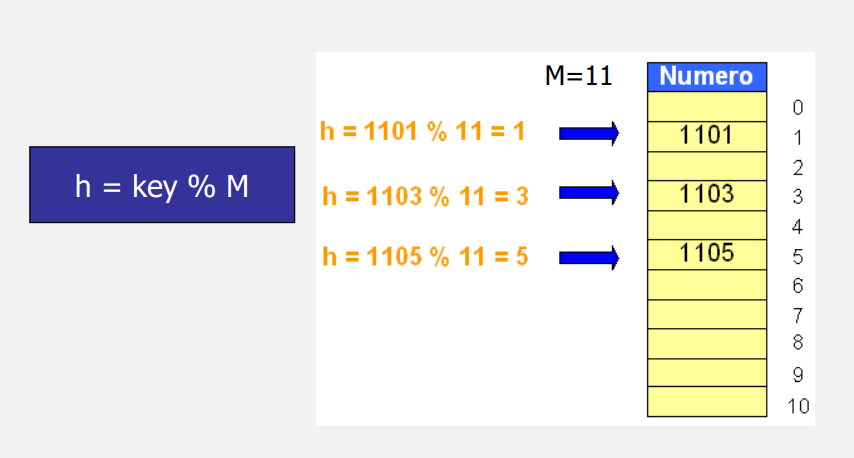


Imagem 2- Exemplo de como adicionar elementos a uma *Hash Table*

presente na parte 12 disponibilizado no *Moodle* na UC de AED

As funções de dispersão podem retornar o mesmo valor para duas ou mais chaves distintas, assim resultando em colisões, podendo por vezes nos piores casos, colidir no mesmo endereço da tabela de dispersão, no entanto ao fazer uma *hash table* com dimensão adequada e uma boa função normalmente não ocorrem colisões.

Existem duas formas de resolução das colisões:

* *Separate Chaining*: todos os elementos ficam com o mesmo endereço na *hash table*, e são colocados na mesma lista ligada, a posição com endereço na tabela fica apontada à cabeça da lista;
* *Open Addressing*: todos os elementos são colocados na *hashtable* sendo umas posições válidos e outros *null*, sendo possível usar três técnicas, *Linear Probing*, *Quadratic Probing*, *Double Hashing*.

Começando por falar das técnicas de *Open Addressing*, *Linear Probing*, a posição dada pela função de dispersão (h’) está ocupada, a chave é arrumada na primeira posição livre adjacente.

A técnica *Quadratic Probing*, assim que a posição está ocupada pela função de dispersão, a chave é colocada numa posição entregue por uma função quadrática.

Por fim *Double hashing*, a posição dada pela primeira função de dispersão está ocupada, calcula-se e aplica-se um deslocamento através da segunda função de dispersão, até encontrar uma posição livre.

Fora do *Open Addressing* conseguimos utilizar o *Separate Chaining* como todos colidem, são colocados numa lista ligada, assim ficando a cabeça da lista na posição que é entregue pela função de dispersão.

## 

## Algoritmos e análise da complexidade

Para resolver o problema utilizamos os algoritmos que foram citados anteriormente sendo o *load, union, intersection, difference*. Eles são algoritmos porque como explicado nos tópicos acima *load* irá ler ficheiros linha a linha, cria objetos e armazena, *union* que junta duas coleções e eliminar os duplicados, o *intersection* encontra elementos comuns entre duas coleções, o *difference* faz uma subtração assim verificando os que são diferentes nos dois

Cada um tem uma função diferente nas duas implementações sendo na:

Implementação 1:

* *load*: ler os ficheiros como Set<Ponto>
* *union*: set1 union set2
* *intersection:* set1 intersect set2
* *difference:* set1 subtract set2

Implementação 2:

* *load*: Guarda lista de (Ponto2, Boolean)
* *union*: Itera e adiciona sem duplicados
* *intersection:* Verificar quais pontos estão nos dois
* *difference:* Filtra pontos do 1º que não estão no segundo

Em relação à complexidade dos algoritmos, cada um tem uma específica sendo no algoritmo *load* O(n), devido a sua leitura e *parsing* e inserção na lista, e juntando os dois obtemos esse valor tanto no tempo quanto no espaço.

No algoritmo union é de O(n), devido à iteração nos pontos, verificação e inserção em vistos, e no total resulta nessa complexidade quanto a tempo e espaço.

Em *intersection*, a complexidade também de O(n), pois nas etapas de Filtros e conversão em conjuntos e interseção com in, no total equivale ao tempo e ao espaço essa complexidade

Por fim o *difference,* cria um mapa com pontos do segundo ficheiro e verifica cada ponto do primeiro está fora do mapa e do resultado no total tendo O(n) como complexidade nesses dois pontos e no tempo e espaço.

# **Avaliação Experimental**

A avaliação foi feita a partir no *intellij*, os testes foram feitos com os conjuntos de ficheiros que pode ser observado na primeira linha da ***Tabela 1***.

Os testes foram feitos num computador *HP* com as seguintes características:

* *CPU - Ryzen 7 5700U with Radeon Graphics*
* *GPU - AMD Radeon(TM) Graphics*
* *RAM - 16GB, 3200 MT/s*

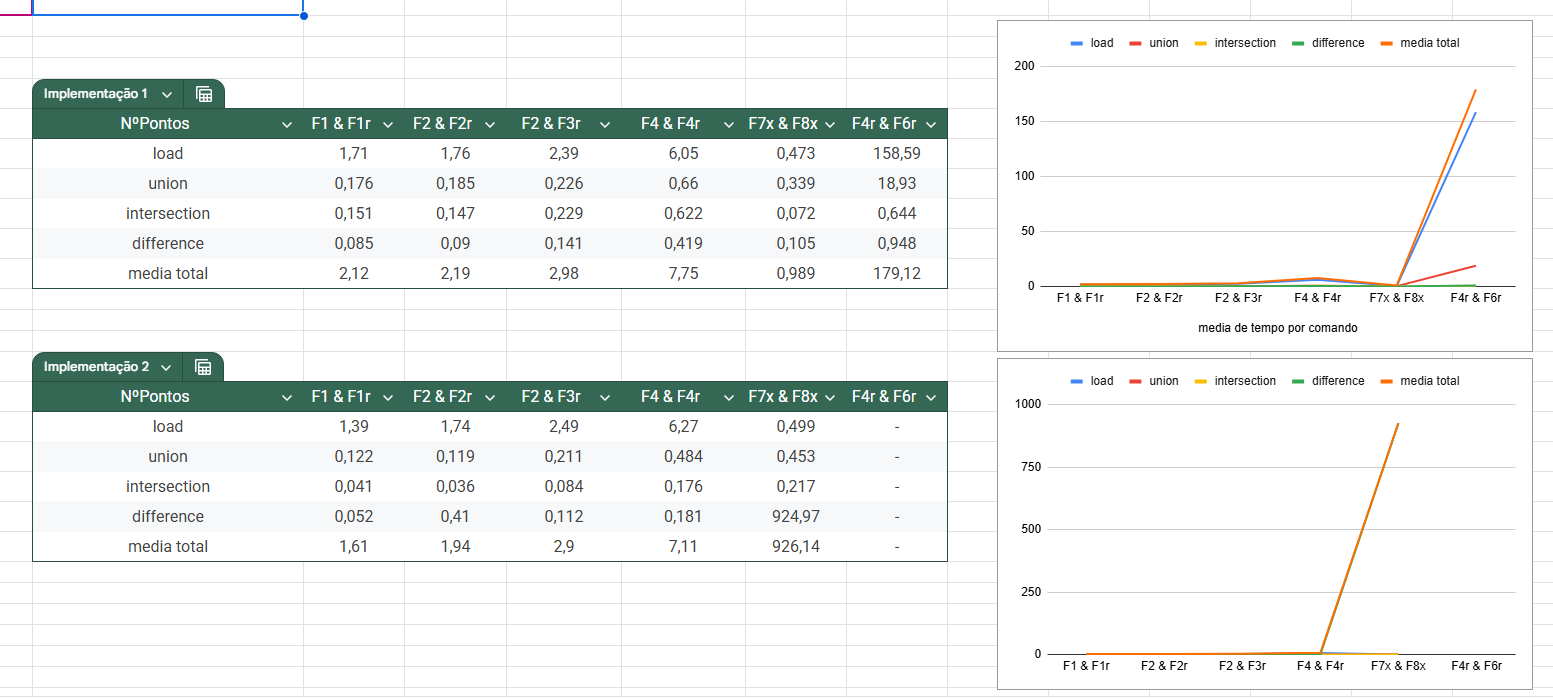
Nos resultados a seguir, os primeiros 5 conjuntos de ficheiros foram feitos repartindo a pasta *Files 1*, que foi nos entregue, em conjuntos de 2 ficheiros, já o 6º conjunto foi feito com o intuito de ver o quanto o código demorava para cada comando usando os 2 maiores ficheiros.

O processo foi feito 5 vezes para chegar às médias e em cada teste do conjunto foi feito de uma só vez, ou seja, os comando foram feitos todos de seguida pela ordem da tabela e esse processo foi repetido 5 vezes, isto para cada implementação, e no fim foi calculado a média de tempo desse mesmo processo.

## 

## **Implementação 1**

Tabela 1: Resultados do tempo de execução da Implementação 1 considerando várias amostras.

****

A Figura 3, ilustra em termos comparativos através de um gráfico, os tempos de execução da Implementação 1.

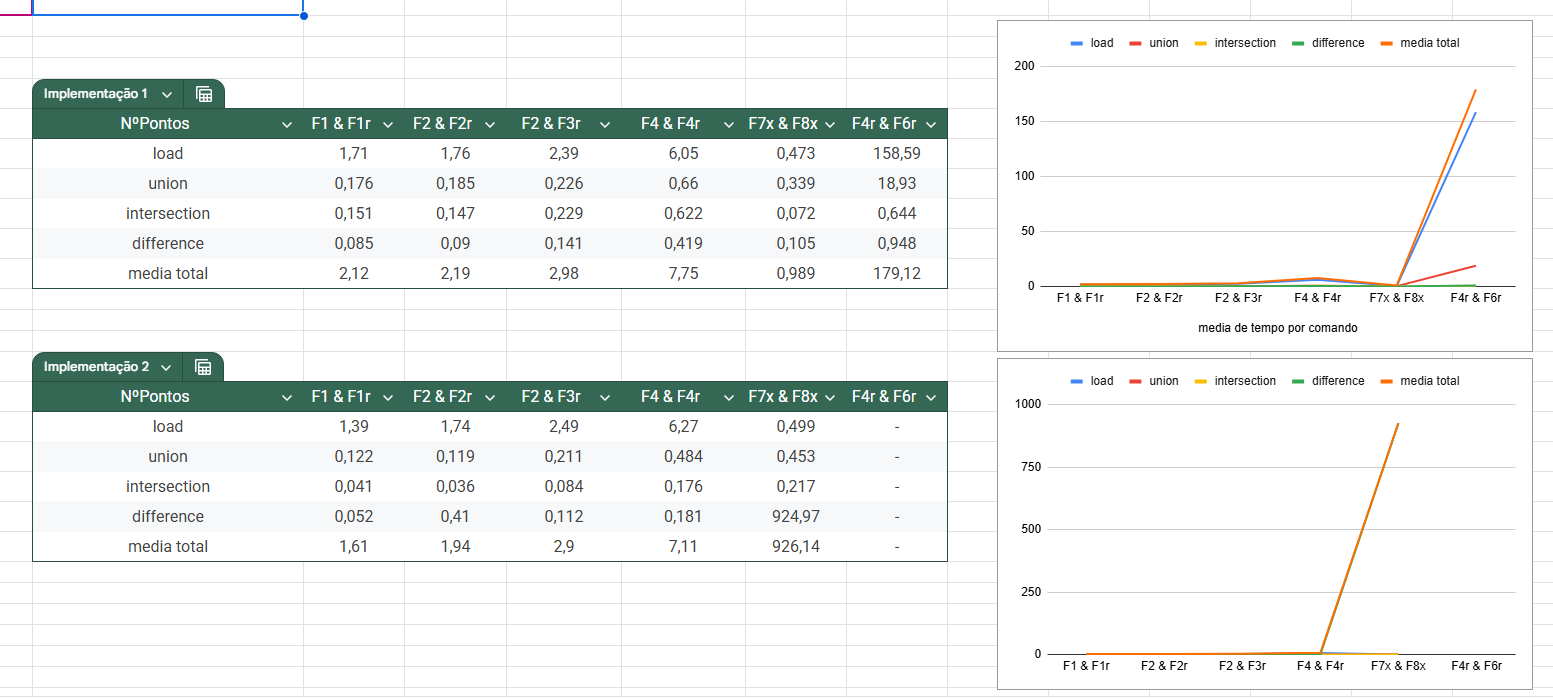
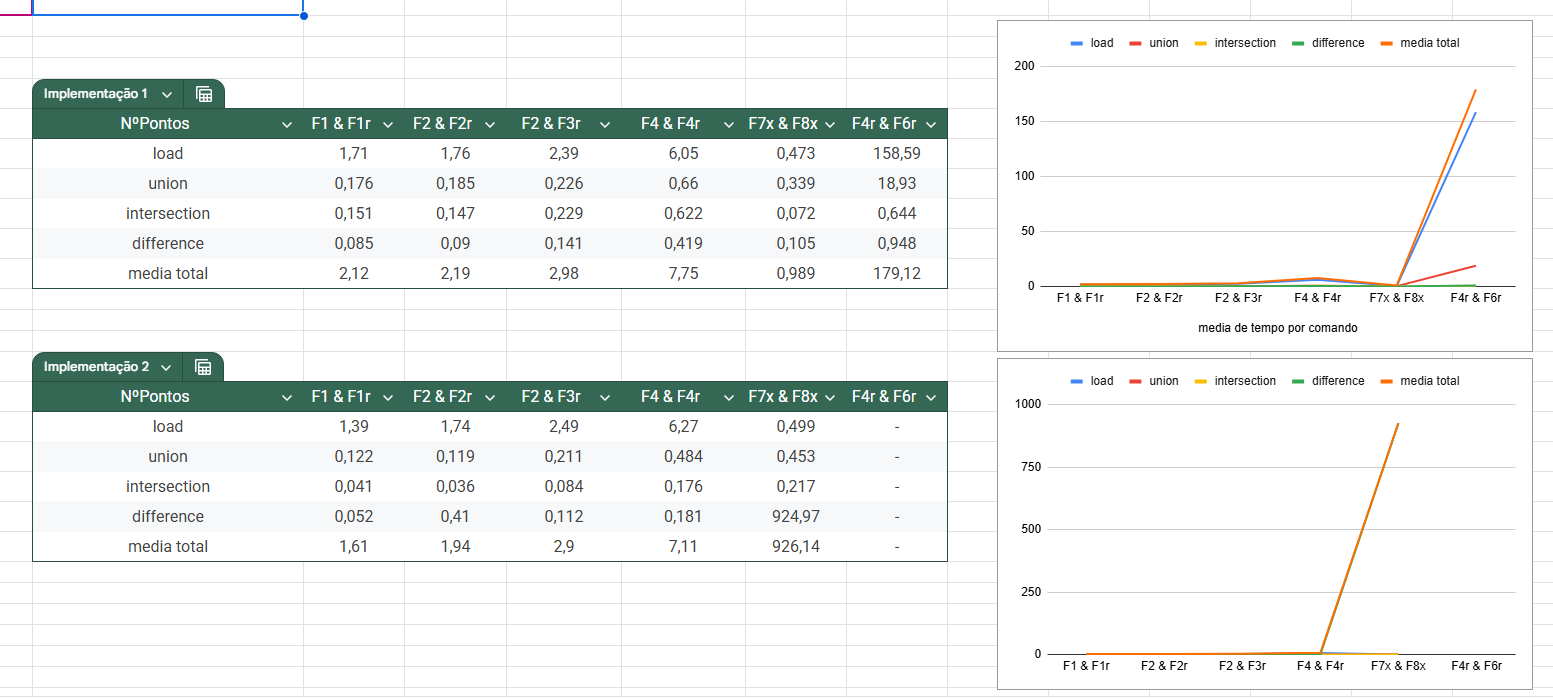


Figura 3: Comparação dos tempos de execução da Implementação 1.

## 

## **Implementação 2**

Tabela 2: Resultados do tempo de execução da Implementação 2 considerando várias amostras.



A Figura 4, ilustra em termos comparativos através de um gráfico, os tempos de execução da Implementação 2.

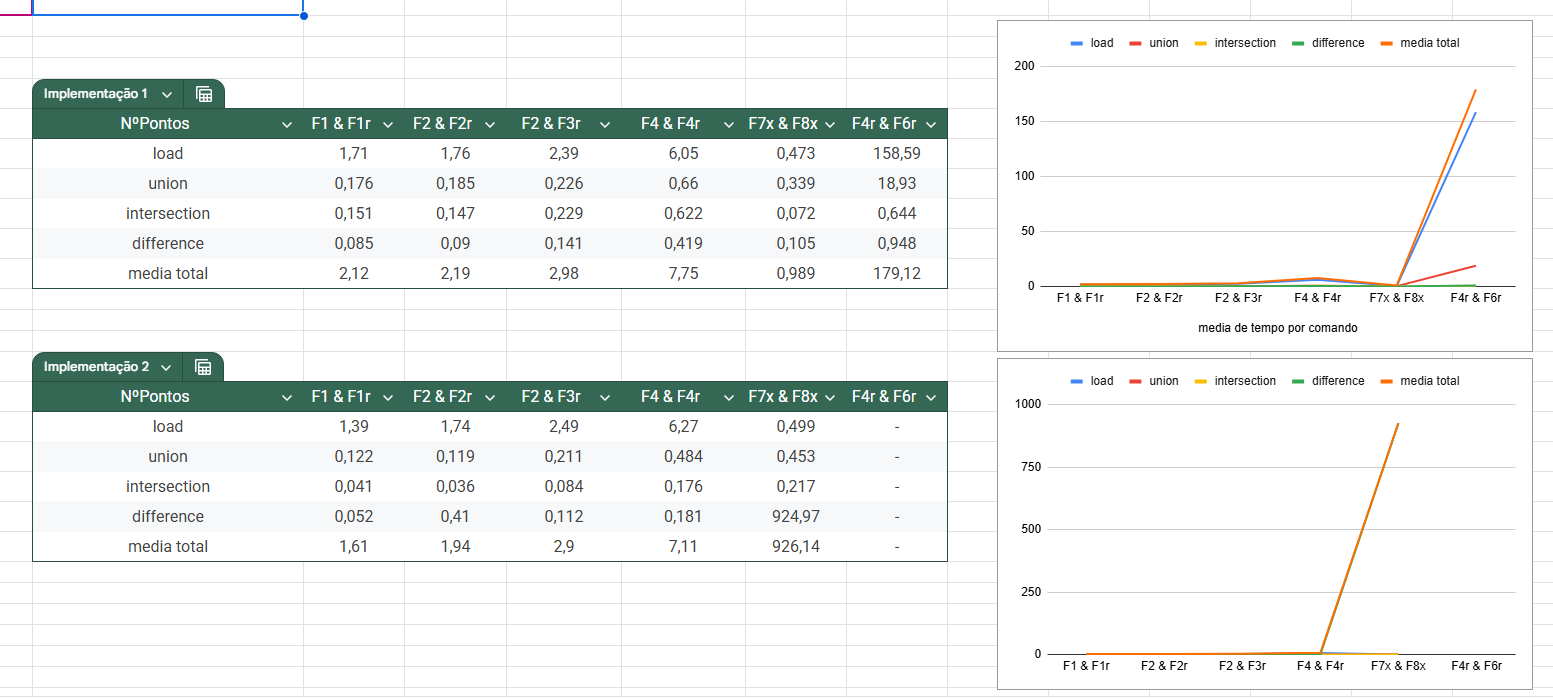


Figura 3: Comparação dos tempos de execução da Implementação 2.

# Resultados

Como podemos ver nos resultados obtidos as duas implementações são semelhantes mas a implementação 1 é mais otimizada já que usa extensões do kotlin já a implementação 2 é mais lenta porque a lógica dos comandos usados na implementação 1 foi criada então demora mais a chegar aos resultados.

A execução com os ficheiros F4r e F6r não pode ser realizada já que a memória do java não é suficiente para que este processo seja possível de ser realizado e mesmo que fosse possível levaria horas até que acaba-se.

# 

# **Conclusões**

Por fim, as duas implementações são semelhantes, mas a implementação 1 é mais otimizada e a implementação 2 é mais lenta quanto maior é o tamanho dos ficheiros até que chega a uma parte que fica obsoleto. Como podemos ver nos gráficos o tempo que leva o processo a ser realizado é proporcional ao tamanho dos ficheiros. A execução com os ficheiros F4r e F6r não pode ser realizada e mesmo que fosse possível levaria horas até que acaba-se.

# 

# **Referências**

[1] “Disciplina: Algoritmos e Estruturas de Dados - 2425SV,” Moodle 2024/25. [Online]. Available: https://2425.moodle.isel.pt. [Accessed: 16-03-2023].

[2] Introduction to Algorithms, 3º Edition. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. MIT Press.

# 