Universidade do Minho
Escola de Engenharia

Laboratórios de Informática III

Fase 2

Grupo 58

Ano letivo 2024/2025

Samuel Gibson Da Cunha Figueiredo Lobato - A106907
Lucas Rafael da Cunha Franco Robertson - A89467
Marco Rocha Ferreira - A106857

Conteúdo

Introdução

Este relatório configura-se como um material de apoio para o projeto realizado no âmbito da UC Laboratórios de Informática III, pelo grupo 58. O intuito deste trabalho resume-se a:

- Desenvolver um conhecimento mais profundo acerca da linguagem de programação C, com ênfase na aplicação do conceito de modularidade no código e no uso de estratégias de encapsulamento;

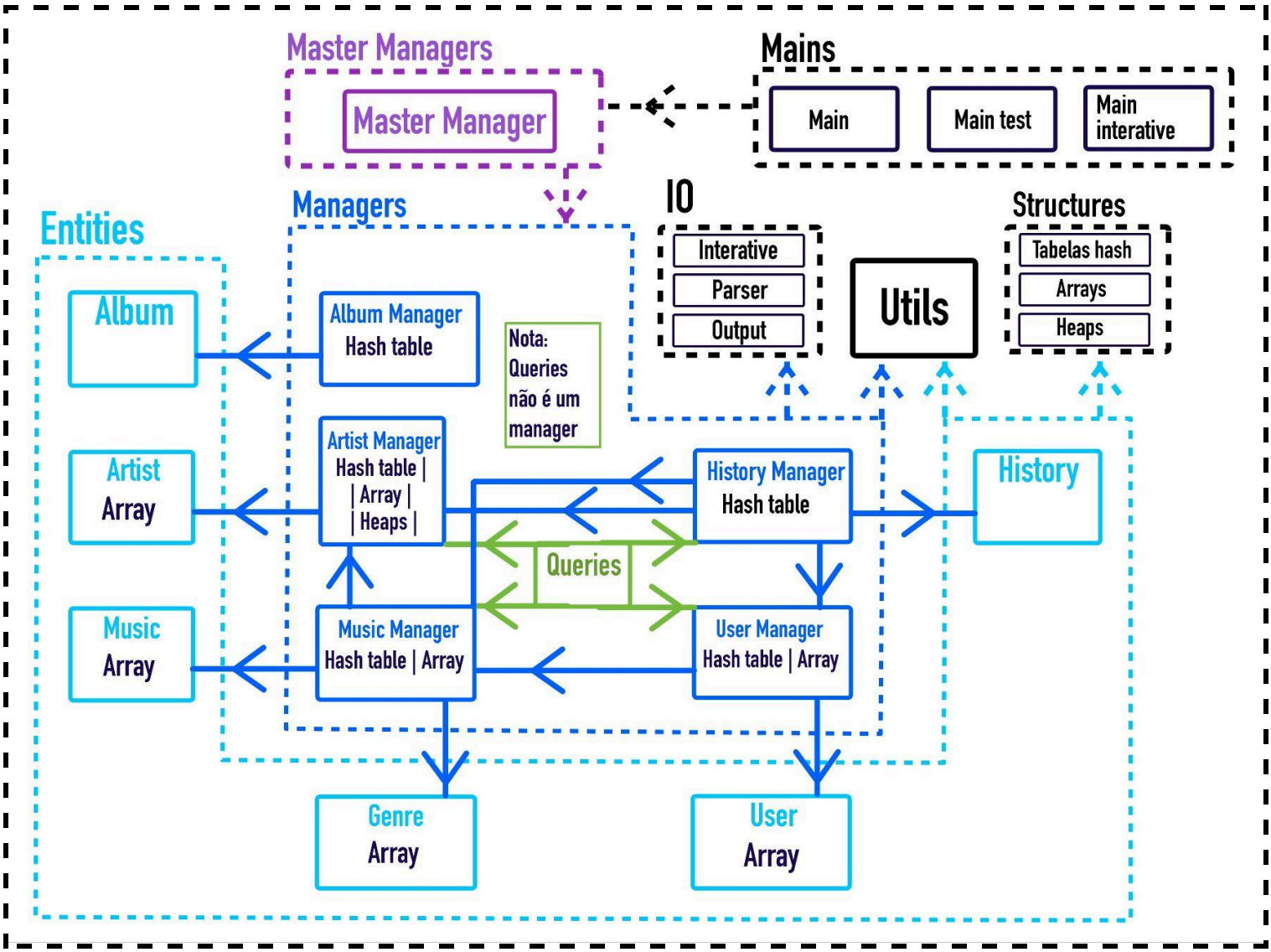
- Aprender a usar ferramentas que permitem melhorar a gestão de memória, com o principal objetivo de evitar *memory leaks*, que tornam o programa inseguro e instável.

Nesse relatório, detalharemos todas as principais mudanças relativas à primeira versão do projeto, nomeadamente sua arquitetura atualizada, novas estruturas de dados e novos mecanismos de execução do programa. Somado a essas mudanças, apresentaremos também os testes de desempenho que tais alterações implicam.

Em destaque, estão as três novas queries, que acreditamos que são respondidas de forma rápida e com uso eficiente de memória, e um programa personalizado capaz de respondê-las de forma interativa, via terminal.

Sistema

Diagrama de arquitetura



Gestores

- Master Manager: Comporta os cinco gestores abaixo descritos e controla a execução do programa;
- Artist Manager: Inclui a tabela de hash dos artistas, um array dos mesmos ordenado segundo a discografia total de cada um e a semana mais antiga onde existe, pelo menos, um artista com tempo de reprodução;

- Music Manager: Inclui a tabela de hash das músicas, um array dos géneros musicais e o número total de géneros diferentes existentes;
- User Manager: Inclui a tabela de hash dos utilizadores, um array com os IDs de todos os utilizadores na forma de string, o número total de utilizadores e o caminho para o ficheiro CSV desta entidade.
- Album Manager: Inclui a tabela de hash dos álbuns.
- History Manager: Inclui a tabela de hash dos históricos, a matriz de reproduções e o seu tamanho (nº de linhas e nº de colunas);

Cada um dos cinco gestores de entidades contém a lógica de armazenamento e a validação lógica das mesmas (só após ser validada é que uma entidade é efetivamente armazenada).

Nota: as cinco tabelas de hash estão organizadas segundo os IDs das entidades correspondentes.

Entidades

- Artists;
- Musics;
- Genre;
- User;
- History;
- Album.

Cada entidade é instanciável e contém os dados presentes nos ficheiros CSV que são relevantes. No que toca a funções, todos estes módulos disponibilizam *getters* e *setters* para que o acesso aos dados seja controlado, i.e., para que outros módulos não alterem os dados das entidades. O Artists, o Genre e o User possuem também funções para imprimir dados (recorrendo à função de output geral, presente no módulo de Output). Por fim, cada módulo contém ainda as funções de validação sintática relativas à sua entidade.

Queries

Este módulo tem uma estrutura geral que se ramifica em cada tipo de query e é única, ou seja, dependendo do tipo de query a ser processado, os campos são atualizados com a informação relevante dessa query. Depois, na execução de uma nova query, os campos anteriores são sinalizados como inválidos, e assim sucessivamente.

IO

- Parser: Módulo que serve para leitura de ficheiros, instanciando estruturas Parser que contém um apontador para um ficheiro e um campo que indica o número de bytes lidos no último acesso a esse ficheiro. Este último campo tem como propósito verificar se foi, efetivamente, lida alguma coisa e também navegar no ficheiro (recorrendo à função fseek).
- Output: Contrariamente, serve para escrita, quer para o *standard output*, quer para ficheiros. Neste último caso, são instanciadas estruturas Output que contém um apontador para o ficheiro (criando-o caso não exista), o separador a ser usado na impressão de resultados (dependendo do tipo de query - S) e um booleano que indica se o resultado deverá ser também impresso para a janela do *NCurses*.
- Interativo: Por fim, este módulo controla o programa interativo, recorrendo à biblioteca *NCurses*. Contém funções que pedem ao utilizador os dados necessários ao longo da execução (e.g., caminho para os dados, tipo da query a executar, etc).

Utils

O módulo Utils contém várias funções usadas por vários outros módulos, como por exemplo, calcular uma duração a partir da string ou validações sintáticas necessárias para várias entidades.

Testagem

Este módulo contém funções específicas ao programa de testes, cujo objetivo é comparar os resultados obtidos com os que eram esperados, imprimindo para o terminal os ficheiros e a linha dentro deste onde ocorreram os erros.

Data Structures

Nas estruturas de dados, implementamos apenas Heaps (que podem ser tanto min-heaps como max-heaps, dependendo da função de comparação passada pelo utilizador). Estas estruturas são usadas na resolução da Query 4 e também no nosso recomendador, como será explicado na secção da discussão.

Discussão

Configurações dos computadores

	PC 1	PC 2	PC 3
Processador	Intel® Core™ i7-1165G7	Intel® Core™ i7-1355U	AMD Ryzen 7 5700U
Frêquencia do CPU	4.7 GHz	5 GHz	4.3 GHz
Capacidade da cache	12 MB	12 MB	8 MB
Números de *cores*	4	10	8
Números de *threads*	8	12	16

Análise de desempenho

Dataset Pequeno

	Q 1[1] (ms)	Q 4[1] (ms)	Q 5[1] (ms)	Q 6[1] (ms)	Armazenamento(s)[2]	Resposta a Queries(s)[3]	Free(s)[4]	Total(s)[5]
PC 1	0.006142	0.213328	63.06648	0.005254	4.062225	0.644507	0.572323	5.272353
PC 2	0.003065	0.12535	54.6395	0.003344	3.288938	0.554234	0.454804	4.297977
PC 3	0.008059	0.358361	69.1646	0.006315	5.153959	0.707312	0.682420	6.543694

Utilização de memória: ~320 MB

[1] As colunas “Q1”, “Q4”, “Q5” e “Q6” fazem referência ao tempo médio necessário para a execução das queries.
[2] Tempo necessário para o armazenamento, validação e organização dos dados.
[3] Tempo total da lógica de resposta.
[4] Tempo necessário para a libertação da memória usada.
[5] Tempo total necessário para a execução do programa.

Dataset Grande

	Q 1[6] (ms)	Q 4[1] (ms)	Q 5[1] (ms)	Q 6[1] (ms)	Armazenamento(s)[7]	Resposta a Queries(s)[8]	Free(s)[9]	Total(s)[10]
PC 1	0.003311	0.174451	71.03627	0.008913	29.926573	3.598440	4.624387	38.149401
PC 2	0.00072	0.165897	57.66219	0.007127	23.486418	2.918703	3.492932	29.898052
PC 3	0.008930	0.475539	73.6730	0.011319	40.649797.	3.781298.	4.878327	49.309426

Utilização de memória: ~1330 MB

Os resultados das tabelas acima foram obtidos a partir de 5 medições, sendo o resultado apresentado uma média. Além disso, todos os testes foram efetuados em modo de desempenho, com o mínimo de processos a correr ao mesmo tempo e utilizando os *datasets* com erros.

Como era de esperar, os melhores resultados foram obtidos pelo PC 2, uma vez que possui a maior frequência de relógio, a maior cache e maior número de *cores*. Os segundos melhores resultados, bastante semelhantes ao PC 2, pertencem ao PC 1, que possui características bastante semelhantes, sendo que a única diferença significativa está no número de *cores*. Por fim, o PC 3, que possui características inferiores aos outros dois, obteve os piores resultados.

Analisando a tabela com algum cuidado, é possível notar que o tempo de armazenamento é o mais significativo, sendo bastante próximo ao tempo total. Isto deve-se ao facto de termos optado por fazer a organização dos dados imediatamente após a validação. Na prática, quando é lida uma linha de dados, a informação dessa linha é validada. Se a validação for satisfeita, a informação é armazenada e, além disso, outros dados relevantes deriváveis a partir da linha do ficheiro são calculados, como por exemplo:

- Idade de um utilizador, que é calculada após a sua validação;
- Likes por género, que vão sendo acumulados a cada utilizador validado;
- Duração de uma música, que é adicionada à discografia total dos seus artistas;
- Nº de álbuns de um dado artista;
- Tempo de reprodução de um artista em cada semana.

Além destes dados, é ainda feita uma ordenação do array de artistas segundo a sua discografia total e são calculados os top 10 de cada semana.

[6] As colunas “Q1”, “Q4”, “Q5” e “Q6” fazem referência ao tempo médio necessário para a execução das queries.

[7] Tempo necessário para o armazenamento, validação e organização dos dados.

[8] Tempo total da lógica de resposta.

[9] Tempo necessário para a libertação da memória usada.

[10] Tempo total necessário para a execução do programa.

Com este armazenamento concluído, as funções de resposta às queries passam a ter toda a informação que precisam “à disposição”. Desta forma, os tempos de execução das queries são bastante reduzidos, porque o “trabalho pesado” é efetuado aquando do armazenamento e validação.

Acerca da “evolução” do desempenho do dataset pequeno para o grande, podemos concluir que o nosso programa cresce de forma bastante “controlada”. Isto porque o dataset aumenta 10 vezes, enquanto que o tempo de execução aumenta, em média, 7,3 vezes e a utilização de memória aumenta 4,2 vezes, evidenciando o uso de estruturas de dados adequadas às necessidades do trabalho.

Funcionamento das queries

Query 1

A query 1 dá-nos o id de um utilizador ou artista para devolvermos certas informações deles ou uma linha vazia caso eles não existam, como nós os guardamos em hash tables cujas keys são os ids, em inteiros, a verificação de se eles estão lá presentes ou não é rapidíssima pois é só dar lookup dos ids na tabela, e se eles existirem obter as suas informações também o é no caso dos artistas, pois no momento de resposta já contém de imediato toda a informação necessária para a resposta, que no caso de strings passamos com getters que devolvem const, sendo algo muitíssimo rápido.

Algo que também ajuda a nossa eficiência é o facto de adicionarmos de imediato +1 ao número de álbuns e somarmos logo a receita da música ao artista logo no momento de leitura dos ficheiros de álbuns e históricos.

Query 4

Quando o ficheiro de históricos é lido, é calculado um valor para a semana para cada entrada, de acordo com o timestamp dessa entrada (a semana 0 é a que contém o dia atual - 9/9/2024 - e quanto mais antiga a data, maior o número da semana). De seguida, a duração, em segundos, é adicionada aos artistas obtidos a partir da música em questão, sendo que estes últimos acumulam o seu tempo de reprodução num array dinámico, inicializado a 0s. Desta forma, se uma dada música foi reproduzida 200 segundos no dia 31/8/2024 (semana 2), todos os artistas desta música vão ter a posição 2 do seu array incrementada em 200.

Após todas as entradas do histórico serem lidas, são calculados os top 10, semana a semana, utilizando um algoritmo com base em min-heaps (mais detalhes na secção da discussão). Após o cálculo de cada top, todos os artistas que o constituem são marcados com -1 na semana (posição do array) devida. Depois de

calcular todos os top 10, é calculada, para cada artista, a frequência acumulada de ocorrências no top. Desta forma, para determinar o artista com mais tops num intervalo, não é necessário somar as ocorrências no intervalo, basta subtrair o valor dos limites do intervalo (no caso do limite inferior, deverá ser uma posição menos, ou 0 caso o inferior já seja 0). Isto é especialmente importante porque um comando da Q4 pode referir-se a todo o tempo. Assim, embora o cálculo da frequência acumulada implique somar tudo, acaba por permitir poupar tempo, uma vez que quanto mais comandos da Q4 são executados, mais somas se evitam.

Finalmente, no que toca à função de resposta, esta imprime o resultado vazio caso a semana da data mais recente do comando seja superior à maior semana encontrada em todo o histórico (guardada pelo Artist Manager), i.e., quando o intervalo do comando é completamente fora do intervalo de tempo abrangido pelo histórico. Caso isto não aconteça, é calculado o artista com mais tops, percorrendo todo o array de artistas e comparando as suas ocorrências em tops (calculadas conforme descrito no parágrafo anterior).

Query 5

Primeiramente, verifica se o id do usuário fornecido para a query 5 não existe ou o número de recomendações requeridas é nulo, retornando nesse caso um output vazio.

Se a os argumentos necessários à Query 5 não forem conhecidos, então começa-se a buscar essas informações a partir de um processo de caching, coletando a matriz de curtidas por gênero do gestor de históricos, o número total de usuários assim como os seus identificadores do gestor de usuários, e o nome e o total de gêneros pelo gestor de músicas, reunindo assim cada parâmetro exigido pelas funções de recomendação.

Existem duas funções recomendadoras: a elaborada pela equipa docente e outra desenvolvida pelo grupo 58. Por padrão, convencionou-se a função dos professores como aquela utilizada pela main. Entretanto, quando o programa é executado interativamente, oferece-se ao usuário a opção de selecionar a função de recomendação de sua escolha.

Por fim, independentemente do recomendado escolhido, um array guardará o conjunto de identificadores devolvido pela função selecionada, informação que responde a Query 5.

Query 6

Devolve um resumo anual de um dado utilizador, nomeadamente a sua duração total de reprodução, o dia em que ouviu mais músicas, o seu álbum preferido, o artista mais escutado, a quantidade de faixas ouvidas, o género que mais consumiu e a hora do dia em que costuma ouvir mais música. Pode também ser pedido um resumo extra, filtrado para um dado número N dos seus artistas favoritos, contendo

o id dos artistas, o número de músicas distintas dele que o utilizador ouviu e a quantidade de tempo que o ouviu.

Enquanto por regra geral tentamos sempre preparar as respostas às queries no momento de leitura do ficheiro, neste caso isso não nos foi possível, sendo que conseguimos apenas auxiliá-las, guardando dentro de cada utilizador os históricos que lhe fazem referência em cada ano. De modo a na query 6 apenas analisarmos o grupo de históricos essencial.

Dentro da nossa lógica de resposta à query 6 são efetuados cálculos para cada dado pedido pelo resumo anual, começando com a verificação de se o utilizador pedido tem históricos nesse ano, em caso negativo imprimimos logo uma linha vazia e ignoramos o resto do processo.

No caso do utilizador ter históricos no ano pedido, nós lemos cada histórico um a um e adicionamos os seus dados às variáveis que lhes estão associadas.

No caso do listening time, simplesmente somamos a duração em segundos de cada histórico. No caso da hora favorita, somamos o listening time de cada histórico a um array de tamanho 24 que representa as 24 horas do dia (0-23), na posição equivalente à hora em que o histórico aconteceu e no final calculamos a posição que contém o máximo. No caso do dia favorito fazemos algo semelhante à hora, mas usamos uma matriz [12][31] e apenas somamos 1 por histórico, no dia do mês em que aconteceu, e no final calculamos o seu máximo. Para saber o género favorito fazemos algo semelhante, desta vez com um array de tamanho igual ao array de nomes de géneros usado pelo recomendado, sendo que temos o passo extra de procurar pela posição do género da música ouvida neste histórico no array de nomes de géneros.

Para a música e álbum favorito utilizamos uma hash table para cada, no caso da música pois a tabela diz-nos ‘imediatamente’ se a música já lhe foi adicionada ou não, sendo que no final só temos de saber o número de posições ocupadas na hash table, algo que é imediato. E no caso dos álbuns utilizámo-las para, por cada música ouvida, podermos adicionar imediatamente a cada álbum o seu listening time, tendo então no final apenas de percorrer a pequena hash table para devolver o mais ouvido.

No caso do artista favorito, utilizamos um array de estruturas, cujas elas guardam não só o id do artista e o listening time que lhe for associado, como também uma hash table para saber facilmente o número de músicas distintas suas que foram ouvidas. Isto permite-nos tanto devolver o artista mais ouvido, ordenando o array segundo a listening time dos artistas, como também devolver os seus resumos especiais caso seja necessário.

Recomendador

O nosso recomendador começa por preparar três arrays auxiliares de floats, um para guardar a frequência relativa de audições do utilizador alvo, outro semelhante mas para armazenar temporariamente esses valores de outros utilizadores

enquanto percorre a matriz, e por último um que contém a soma das frequência relativas de cada utilizador, filtradas pelo array do utilizador alvo.

Após esta iniciação, começa então por identificar a linha das audições do utilizador alvo e copia-a para o array de floats, e torna-as em frequências relativas. De seguida faz esse mesmo cálculo de frequências relativas para cada outra linha de utilizadores, mas verifica se para cada coluna ultrapassam a frequência relativa do utilizador alvo, sendo que nesse caso limitam-na para esse valor. Enquanto faz isto, também vai somando esses valores no terceiro array de floats, na posição que fizer referência a esse utilizador, de forma a acabar com um número de 0 a 1 lá guardado, que nos dará uma ideia do quão semelhante este utilizador é do nosso alvo.

No final simplesmente desloca os maiores R elementos para o início do array, com um algoritmo baseado em heaps, melhor descrito na secção “Utilização de min-heaps na resolução da Query 4”, sendo R o número de recomendações pedidas.

Otimização da Query 1

Uma vez que parte dos campos que armazenávamos nos utilizadores (primeiro e último nome, país e email) só são necessários para responder à Q1, optamos por deixar de os armazenar. Ao invés disso, o User Manager guarda o caminho para o ficheiro CSV dos utilizadores e cada um destes últimos passa a guardar a posição onde se encontra no ficheiro. Portanto, para responder à query 1 relativa aos utilizadores, começamos por verificar se este existe na tabela de hash; se não existir, é escrito o resultado vazio, e se existir, o ficheiro é aberto e leem-se os dados do utilizador, a partir da posição que este guardou e recorrendo à função *fseek*.

Com esta nova abordagem, conseguimos poupar cerca de 50 MB no dataset pequeno e 70 MB no dataset grande. Quanto ao tempo, embora o acesso aos ficheiros no disco seja mais demorado do que o acesso à memória principal, o tempo de execução do programa manteve-se inalterado, uma vez que este acesso extra só é feito para responder ao comandos da Q1 e deixaram de ser feitos *strdups* dos tokens das strings em causa para todos os utilizadores. Por outras palavras, a redução da quantidade de cópias de strings compensou o acesso aos ficheiros.

Utilização de min-heaps na resolução da Query 4

Conforme foi mencionado acima, para resolver a Q4 de uma forma mais eficiente, optamos por usar um algoritmo que utiliza min-heaps. Para justificar esta escolha, iremos começar por apresentar a evolução das nossas estratégias para a resolução.

Seja A o nº total de elementos (artistas) do array principal e T o nº de maiores elementos a encontrar (para a Q4, T será sempre 10).

A primeira ideia a surgir foi a mais básica, que seria, para cada semana, reordenar o array dos artistas (seguindo a ideia de armazenamento apresentada na descrição da Q4 acima) e depois marcar os 10 primeiros (ou 10 últimos, dependendo da ordenação), de forma semelhante ao que já foi descrito. No entanto, esta abordagem é muito pouco eficiente, uma vez que cada ordenação (usando o quicksort) tem uma complexidade, no caso médio, de O(A.log(A)), e apenas precisamos dos T maiores elementos. Por outras palavras, uma ordenação completa é-nos desnecessária.

Excluindo esta ideia, surgiu uma semelhante (seguindo a estratégia de ordenação), que consiste em criar uma estrutura auxiliar (e.g., uma lista ligada) para armazenar os T maiores elementos, mantendo apenas a estrutura auxiliar ordenada. Desta forma, a complexidade deste algoritmo seria, no pior caso, O(A.T), o que acaba por ser melhor do que a 1ª estratégia, mas apenas para valores de T bastante pequenos em comparação a A (mais concretamente, menores que log2(A)). Embora, para a resolução desta query, esta estratégia já fosse melhor que a anterior, queríamos encontrar uma forma melhor de resolver este problema e que não fosse tão dependente do valor de T.

Foi assim que chegamos ao algoritmo que utilizamos. Tendo em conta que teremos sempre de percorrer todos os elementos para encontrar os maiores, a complexidade será sempre A.x, sendo x referente à estrutura auxiliar referida no parágrafo anterior. Portanto, tentamos usar uma estrutura que fosse mais eficiente a manter a ordenação, e foi assim que chegamos às min-heaps (armazenadas sob a forma de array). Explicação do algoritmo, aplicado a cada semana:

1. Começar a percorrer o array principal, adicionando à heap os primeiros T artistas que tenham tempo de reprodução nesta semana;
2. Se o array foi completamente percorrido sem chegar a encher a heap, então o top está calculado;
3. Caso contrário, a partir do momento em que a heap está cheia, o critério de inserção muda: só são adicionados à heap os artistas que têm maior tempo de reprodução que o 1º (menor) elemento da heap, sendo depois feito um *bubble down* para assegurar as propriedades das min-heaps.

Desta forma, ao chegar ao final do array, a heap contém os 10 artistas com maior reprodução na semana em questão. Como para cada inserção podem ser feitas, no pior caso, log2(T) trocas, a complexidade do pior caso deste algoritmo é O(A.log(T)), o que é melhor e muito menos dependente do valor de T em comparação à segunda solução apresentada. Na pior das hipóteses, T é muito próximo de A (ser igual não faz muito sentido, passa a ser uma ordenação habitual) e então a complexidade temporal deste algoritmo é a mesma que ordenar todo o array. Sendo assim, este algoritmo é o mais adequado às nossas necessidades.

Conclusão

Nesta segunda fase, o desenvolvimento do projeto foi bastante mais “fluído”, tendo em conta a modularidade e o encapsulamento que temos aplicado desde o início do trabalho. Embora, no início, fosse difícil adaptar-nos a estes conceitos, eles provaram-se muito úteis quando a quantidade de código do trabalho começou a aumentar, evidenciando-se, por exemplo, a reutilização de muitas funções e facilidade de vários elementos escreverem código paralelamente.

Após a conquista de 100% em todos os parâmetros da plataforma de testes, focamo-nos em aprimorar ainda mais as funcionalidades do nosso programa, tentando aumentar a sua eficiência a nível de tempo e de memória e melhorando a legibilidade do código.

Mesmo com a imposição do novo dataset, de maiores proporções, o nosso programa continuou eficiente, apresentando um crescimento inferior a linear.

Coletivamente, o nosso grupo concorda que o projeto de LI3 foi responsável por melhorar as nossas capacidades de escrever código em grupo/paralelo, desenvolver as nossas capacidades comunicativas, a nossa perseverança em não nos contentar com o aceitável e, sobretudo, o nosso senso de gestão de memória.