Trabalho Prático 2 - Estruturas de Dados

Luiz Philippe Pereira Amaral [1]

[1] Departamento de Ciência da Computação – Universidade Federal de Minas Gerais

luizphilippe@dcc.ufmg.br

Abstract. *Rick Sanchez's travel schedule problem is a scenario where we have a list of planets to visit, each with their own name and time to visit, and we also have a maximum time allocated to visit planets on a month. From this information, we need to organize an agenda with the planets to be visited each month so that:*

- *The number of planets visited in a month is always greater than or equal to the number of planets visited in a month later.*
- *Planets are visited within a month following the alphabetical order of their names.*

The objective of the work is to implement an algorithm capable of organizing such a schedule in a timely manner for large lists of planets. The following algorithm architecture is built using a vector of planets, a month row, and merge sort ordering.
Keywords: Data Structures, Sorting Algorithms, C ++.

Resumo. *O problema da agenda de viagens de Rick Sanchez trata-se de um cenário onde temos uma lista de planetas a serem visitados, cada um com o seu nome e tempo necessário para a visita, e temos também um tempo máximo alocado para visitar planetas em um mês. A partir dessas informações, precisamos organizar uma agenda com os planetas a serem visitados em cada mês de forma que:*

- *O número n_1 de planetas visitados em um mês m_1 seja sempre maior ou igual ao número de planetas n_2 visitados em um mês posterior m_2.*
- *Os planetas sejam visitados em um mês seguindo a ordem alfabética de seus nomes.*

O objetivo do trabalho é implementar um algoritmo capaz de organizar tal agenda em tempo hábil para grandes listas de planetas. A arquitetura do algoritmo apresentada a seguir é construída utilizando um vetor de planetas, uma fila de mêses e ordenação merge sort.
Palavras-chave: *Estruturas de Dados, Algoritmos de Ordenação, C++.*

1. Implementação

A implementação do código foi feita em inglês devido ao benefício de se tratar de um idioma que não utiliza acentos.

Para a solução do problema foram implementadas uma fila (*Queue)* e dois algoritmos de ordenação (*Quick sort* e *Merge sort*). Inicialmente as ordenações - tanto pelo

tempo da visita quanto pelo nome dos planetas - era feita com o método do *Quick sort*, no entanto esse algoritmo foi posteriormente substituído pelo *Merge sort* por se tratar de um algoritmo estável e que lida melhor com entradas maiores. Também foram implementadas classes que auxiliam na representação das entidades do sistema, mais especificamente dos planetas(*Planet*), dos meses (*Month*) e da agenda (*TravelGuide*).

1.1. Merge sort

O algoritmo de ordenação *merge sort* implementado consiste em duas funções: "*merge_sort*" responsável por chamar recursivamente a si própria para cada uma das metades do vetor recebido até que se trate de um vetor de ordenação trivial e "*merge*" que realiza a união (ou o merge) das metades já ordenadas. Este procedimento tem complexidade $O(nlog(n))$.

1.2. Radix sort

O radix sort, utilizado para ordenar os planetas alfabeticamente utiliza um vetor de filas para classificar os planetas de acordo com a letra em uma posição específica do seu nome, começando pela letra mais a direita (LSB) até a letra mais a esquerda (MSB). A fila é esvaziada e seus elementos colcados em ordem de volta ao vetor de onde vieram. Este procedimento tem complexidade $O(n \cdot k)$, onde k é o número de caracteres dos nomes.

1.3. Fila

A fila contém os ponteiros "first" e "last" que apontam para as células do fim e do início e o inteiro relativo ao seu tamanho. A adição é feita ao fim da fila por meio do método "add" (complexidade $O(1)$) e a remoção é feita do início da fila com o método "remove" (complexidade $O(1)$). A fila pode ser liberada recursivamente ou iterativamente.

1.4. Célula

Uma célula é composta por um objeto de um tipo T* genético e dois ponteiros "next" e "prev" que apontam para as células a direita e a esquerda respectivamente. Célula possui um construtor que recebe o objeto T* a ser armazenado pela estrutura e possui métodos get para seus atributos. A classe *Queue* é declarada como "friend" a célula, já que são intrinsecamente ligadas.

1.5. Planeta

Um planeta é composto por uma cadeia de caracteres que representa seu nome e um inteiro que representa o tempo da duração de sua visita..

1.6. Mês

Um mês possui um vetor de planetas (que são os planetas a serem visitados, em ordem) e o número de planetas visitados (o tamanho do seu vetor de planetas).

1.7. Guia de viagens

A classe guia de viagens foi criada para conter funções estáticas que auxiliam na execução do problema bem como os dados relacionados ao ambiente do problema. É nessa classe que se encontram as funções de ordenação, o vetor de planetas e o procedimento para o cálculo da agenda.

1.8. Resolução do problema

A função main do código aloca um vetor de planetas recebido pela entrada e o atribui a propriedade estática da classe *TravelGuide*, em seguida faz uma chamada a função "*TravelGuide::visit_planets*" para receber a fila de meses para a visitação. Nessa função, começamos utilizando o *merge sort* para ordenar todos os planetas do vetor baseado em seu tempo de visita. Em seguida acompanhamos o número de planetas já visitados e enquanto ele é menor que o número total de planetas a visitar, iteramos sobre nosso vetor de planetas e agrupamos os n_1 primeiros planetas onde a soma do tempo de visita desses planetas seja menor ou igual a T onde T é o número máximo de tempo de visitas em um mês e n_1 é o maior possível. Então usamos o *radix sort* para ordenar o trecho do vetor de planetas das posições 0 até $n_1 - 1$ baseado nos nomes dos planetas e criamos um mês m_1 com o trecho do vetor que acabamos de ordenar de tamanho n_1 sem que para isso seja necessária a alocação de outro vetor na memória, utilizamos apenas o endereço de memória do início do trecho desejado. Em seguida repetimos o procedimento com para um grupo de n_2 planetas que se encontram nas posições n_1 a $n_2 - 1$ e onde $n_1 \geq n_2$ e n_2 seja o maior possível tal que a soma do tempo de visita desses planetas seja menor ou igual a T e assim por diante até termos agrupado todos os planetas do vetor em algum mês.

Por fim removemos os meses da fila e imprimimos seus vetores de planetas como as saídas.

Ao término da execução do programa, nosso vetor de planetas é liberado bem como cada um dos meses da lista.

2. Instruções de compilação e execução

1) O makefile preparado para a execução do código tem as seguintes opções:
2) make: Compila os arquivos de código fonte e armazena a saída da compilação em uma pasta "./build/".
3) make run: Executa o arquivo gerado pela compilação para entrada manual.
4) make mem: Executa o arquivo gerado pela compilação em modo de depuração de memória, assim podemos nos certificar que não existe nenhum vazamento de memória acontecendo durante a execução.
5) make test: Executa os script de testes do trabalho.
6) make clean: Limpa a pasta contendo as saídas da compilação.

3. Análise de complexidade

A leitura das entradas tem sempre complexidade $O(n)$. Em seguida temos a execução do *merge sort* que tem complexidade $O(nlog(n))$ em todos os casos. Então percorremos o nosso vetor, agora ordenado, e aplicamos o *radix sort* para partições do vetor, esse trecho tem complexidade $O(n \cdot k)$. Por fim, imprimimos os n planetas na ordem em que foram visitados com complexidade $O(n)$.

Nossa função de complexidade seria algo similar a $f(n,k) = 2n + nlog(n) + nk$, no entanto sabemos que $f(x) + g(x) = max(O(f(x)), O(g(x)))$ e dessa forma nossa ordem de complexidade é de $O(nlog(n))$ em todos os casos.

4. Conclusão

A primeira observação notável sobre o problema é que, embora o *quick sort* tenha se mostrado mais eficiente para pequenas entradas (em termos de memória utilizada) ele apresentou uma drástica perda de desempenho para entradas maiores como mostrado na figura abaixo:

```
==672== HEAP SUMMARY:
==672==     in use at exit: 0 bytes in 0 blocks
==672==   total heap usage: 14 allocs, 14 frees, 78,112 bytes allocated
==672==
==672== All heap blocks were freed -- no leaks are possible
==672==
==672== For counts of detected and suppressed errors, rerun with: -v
==672== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figura 1. Memória utilizada pelo algoritmo *quick sort* para uma entrada de 3 planetas.

```
==641== HEAP SUMMARY:
==641==     in use at exit: 0 bytes in 0 blocks
==641==   total heap usage: 18 allocs, 18 frees, 78,152 bytes allocated
==641==
==641== All heap blocks were freed -- no leaks are possible
==641==
==641== For counts of detected and suppressed errors, rerun with: -v
==641== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figura 2. Memória utilizada pelo algoritmo *merge sort* para uma entrada de 3 planetas.

Vemos que para pequenas entradas o *quick sort* utiliza uma quantidade menor de memória (40 bytes a menos) em relação ao *merge sort*. Porem, o *quick sort* responde de forma menos eficiente para grandes aumentos no tamanho da entrada como apresentado a seguir:

```
==459== HEAP SUMMARY:
==459==     in use at exit: 0 bytes in 0 blocks
==459==   total heap usage: 18,891,621 allocs, 18,891,621 frees, 330,345,798 bytes allocated
==459==
==459== All heap blocks were freed -- no leaks are possible
==459==
==459== For counts of detected and suppressed errors, rerun with: -v
==459== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figura 3. Memória utilizada pelo algoritmo *quick sort* para uma entrada de 150 mil planetas.

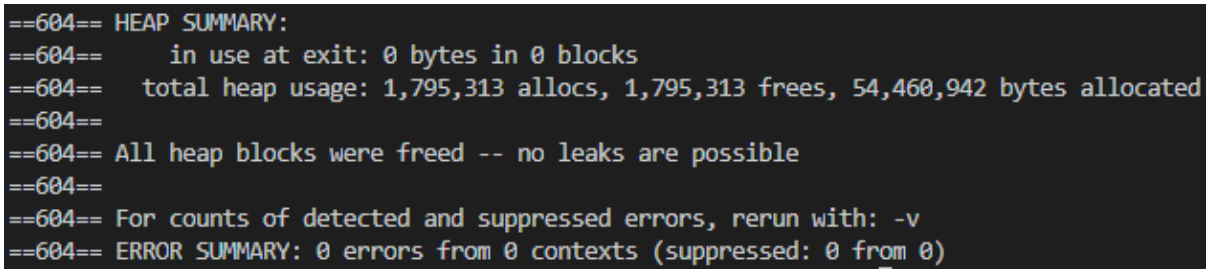

Figura 4. Memória utilizada pelo algoritmo *merge sort* para uma entrada de 150 mil planetas.

Nesse caso, o *merge sort* se apresenta drasticamente mais eficiente já que cerca de 6 vezes menos memória foi alocada.

Em termos de tempo de execução, o algoritmo é viável e moderadamente eficiente para entradas moderadamente grandes já que sua complexidade é quase linear, sendo assim, a solução apresentada é bem viável na grande maioria dos casos.

5. Bibliografia

- https://www.geeksforgeeks.org/quick-sort-vs-merge-sort/
- http://www.cplusplus.com/reference/string/string/compare/