Trabalho Prático 1 - Estruturas de Dados

Luiz Philippe Pereira Amaral [1]

[1] DCC – Universidade Federal de Minas Gerais

luizphilippe@dcc.ufmg.br

Abstract. *Rick Sanchez's measurement problem is a scenario where, from different containers that can be used to add or remove a certain amount of a substance from his measurement, we want to make a measurement of x ml. The objective of the work is to implement an algorithm capable of providing the minimum number of actions (each action being the act of adding or subtracting the total capacity of the flask from our measure) necessary to measure the desired quantity. The algorithm architecture presented below is built using two lists of elements that are traversed. We were able to verify a high complexity for solving robust instances of the problem. Therefore, it can be stated that as the measurement becomes more extensive, the proposed algorithm becomes inefficient quickly.*
Keywords: *Data Structures, Linked List, C ++.*

Resumo. *O problema da medição de Rick Sanchez trata-se de um cenário onde, a partir de diferentes recipientes que podem ser utilizados para adicionar ou remover uma certa quantidade de uma substância da sua medida, desejamos realizar uma medida de x ml. O objetivo do trabalho é implementar um algorítimo capaz de fornecer o número mínimo de ações (sendo cada ação o ato de adicionar ou subtrair a capacidade total do frasco da nossa medida) necessárias para que possa ser medida a quantidade desejada. A arquitetura do algorítimo apresentada a seguir é construída utilizando duas listas de elementos que são percorridas. Foi possível verificar uma alta complexidade para a resolução de instâncias robustas do problema. Portanto, é possível afirmar que conforme a medição se torna mais extensa, o algorítmo proposto para o trabalho deixa de ser eficiente rapidamente.*
Palavras-chave: *Estruturas de Dados, Lista Encadeada, 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 e solução do problema foram implementadas três classes para as estruturas de dados: célula (*Cell*), lista encadeada (*LinkedList*) e file (Queue). Todas as estruturas foram implementadas utilizando templates, de forma que a estrutura funciona independe do conteúdo armazenado na célula (e, consequentemente, na lista ou na fila). Também foram implementadas classes que auxiliam na representação das entidades do sistema, mais especificamente dos recipientes (*Vessel*), das operações de medida (*Operation*) e do problema em si (*Measure*).

1.1. Lista Encadeada

A classe lista contém um inteiro que armazena a quantidade de objetos inseridos na lista e dois ponteiros "*first*" e "*last*" que apontam para duas células, a primeira e a última da lista respectivamente. A lista possui um método get que retorna seu tamanho. A adição de objetos é feita por padrão no fim da lista com o método "*add*" (complexidade O(1)) e a remoção é feita por meio de um índice *i* recebido na função "*remove*" (complexidade O(n) para o pior caso e O(1) para o melhor, sendo os melhores casos aqueles onde *i* é igual a primeira ou a última posição da lista). Também é possível inserir em qualquer posição da lista com complexidade O(1) caso seja fornecida uma referência para uma célula utilizando as funções "*insert_after*" e "*insert_before*". É possível pesquisar células e objetos na lista com complexidade O(n). A lista pode ser liberada de forma recursiva e iterativa.

1.2. Fila

A fila é semelhante a lista, porém mais otimizada para adição de elementos no fim da estrutura e remoção do início. Ela contém os mesmos 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 também pode ser liberada recursivamente ou iterativamente.

1.3. 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 *LinkedList* é declarada como "*friend*" a célula, já que são intrinsecamente ligadas.

1.4. Recipiente

Um recipiente é composto simplesmente de um inteiro que representa sua capacidade. O construtor para recipiente recebe esse inteiro e a classe possui um get para esse valor.

1.5. Operação

Uma operação representa a combinação de várias ações de adição e subtração dos recipientes. Seu construtor recebe dois inteiros, o número de ações associadas a operação e a quantidade medida. Se temos como exemplo um cenário onde temos dois recipientes de 30ml e 10ml e utilizamos $1 \times 30 + 2 \times 10$ para medir 50ml, temos uma operação onde o número de recipientes é 3 e a quantidade medida é 50.

1.6. Medida

A classe medida foi criada para conter funções estáticas que auxiliam na execução do problema. Essas funções servem para interpretar os dados da entrada e executar as ações solicitadas. O método "*execute*" recebe a lista de recipientes disponíveis e os dados da entrada, um inteiro e um caractere, para cada entrada inserida, em seguida, ela avalia o tipo de instrução a ser executada e chama o método apropriado. O método "*add_vessel*" é

responsável por receber um inteiro e a lista de recipientes e adicionar um novo recipiente com a capacidade recebida na lista. O método "*remove_vessel*" é responsável por remover um recipiente com a quantidade especificada da lista. O método "*min_measure*" por fim, calcula, baseado nos recipientes disponíveis, o número mínimo de ações necessárias para obter a medida desejada.

1.7. Resolução do problema

A função main do código possui uma lista encadeada de recipientes inicialmente vazia, nessa lista são adicionados e removidos recipientes conforme a entrada. Quando uma medição é solicitada, a lista o método *Measure::execute* faz uma chamada a *Measure::min_measure*.

O algoritmo então começa declarando uma fila de operações e itera uma vez sobre a lista de recipientes buscando uma possível solução trivial. Em cada uma dessas iterações, um recipiente é selecionado, caso esse recipiente tenha uma capacidade igual a medida que desejamos realizar, o código retorna 1, caso contrário, uma nova operação é adicionada a fila contendo uma ação e uma quantidade medida igual a capacidade do recipiente. Ao fim dessas iterações, iniciamos um novo loop que ocorre enquanto a fila de operações não é vazia (condição que ocorre somente quando não temos nenhum recipiente disponível, para todos os outros casos, isto é essencialmente um *while(true)*). Nesse loop, no início de cada iteração removemos a primeira operação da fila e iniciamos um loop interno que passa por cada um de nossos recipientes. Nesse loop interno, verificamos se o resultado da soma ou da subtração do recipiente da iteração atual com a operação em mãos é igual a medida desejada, em caso positivo, liberamos a fila de operações e retornamos o número de ações da operação em mãos + 1. Caso negativo, adicionamos as duas operações derivadas do recipiente ao fim da lista de operações (verificando as restrições para isso). Caso a medida seja possível, esse loop eventualmente encontra a solução adequada.

Ao fim da execução do programa, nossa lista de recipientes é liberada.

2. Instruções de compilação e execução

O makefile preparado para a execução do código tem as seguintes opções:

1) make: Compila os arquivos de código fonte e armazena a saída da compilação em uma pasta "*./build/*".
2) make run: Executa o arquivo gerado pela compilação para entrada manual.
3) 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.
4) make test: Executa os script de testes do trabalho.
5) make clean: Limpa a pasta contendo as saídas da compilação.

3. Análise de complexidade

Seja n o número de frascos disponíveis (ou o tamanho da lista de frascos) e m o número de movimentos necessários para que a solução seja encontrada. O código consome da nossa lista de operações de forma contínua, para cada umas das n operações iniciais teremos $2n$ novas operações (cada uma correspondendo a ação de adicionar ou retirar um dos frascos da medida). Dessa forma, até encontrarmos a solução faremos:

$$n \times (2n) \times (2n)...$$

Como essa iteração, ocorrerá m vezes (até que a solução seja encontrada), podemos dizer que esse comportamento iterativo será executado no pior caso em função de:

$$n \times (2n)^{m-1}$$

Portanto, nossa função de complexidade é dada por $f(n,m) = n \times (2n)^{m-1}$ que é $O(n^m)$. Isso significa que apesar do nosso algoritmo se comportar razoavelmente bem conforme n cresce, existe uma perda de eficiência drástica conforme m se torna um número grande. Ou seja, o algoritmo responde de forma bem menos eficiente ao número de ações necessárias para realizar a medida do que ao número de frascos disponíveis para isso.

4. Conclusão

Evidentemente, é impossível dizer de antemão em quais casos o algoritmo será ineficiente, já que em casos onde m se torna grande, executar o código deixa de ser computacionalmente viável e precisamos executar o código para ter seu valor de m associado.

Concluímos portanto que o mais adequado seria trabalhar uma solução heurística que nos forneça uma boa aproximação para a solução ótima existente.

5. Bibliografia

- http://www.cplusplus.com/doc/oldtutorial/templates/
- https://www.geeksforgeeks.org/templates-cpp/