Trabalho Prático 1: O problema da medição de Rick Sanchez

Daniel Ferreira Abadi
2018088062

1. Introdução

O problema abordado consiste em uma forma de armazenar todos os recipientes e seus respectivos tamanhos, em mililitros, para achar o menor número de combinações que devem ser feitas a fim de achar uma determinada medida. Para resolver o problema, o programa utiliza duas listas encadeadas, uma para armazenar o tamanho dos frascos e as medidas. Para realizar as combinações são utilizados 2 laços, encadeados, alocando as combinações.

2. Implementação

O programa utiliza duas estruturas de dados, chamadas de "no" e de "lista". A estrutura "lista" contém 5 métodos que serão vistos mais à frente.

2.1. Estruturas

A primeira estrutura é chamada de "no", ela guarda apenas três dados, um apontador para outro "no" e dois inteiros que representam a medida e o número de operações para se chegar naquela quantidade. Ela não possui métodos.

A segunda estrutura se chama "lista", ela contém dois apontadores do tipo "no", que recebem o endereço para o primeiro elemento e último elemento, respectivamente. Ela contém 5 métodos que serão vistos na próxima seção.

2.2. Métodos

Todos os métodos são da estrutura "lista", 3 deles são "padrões" em outras listas, sendo eles "void insere_no()", "void remove_no(int)" e "void deleta_lista()", que servem para adicionar um novo elemento ao final da lista, retirar um determinado elemento e apagar todos os elementos da lista, respectivamente. Tendo esta informação, irei comentar apenas sobre os outros dois métodos, que são os principais e foram feitos exatamente para esse propósito.

2.2.1. Método "combina_frascos"

Este método recebe como parâmetro um inteiro, o endereço de uma lista, que representam a medida que quero achar e a lista de frascos, respectivamente. E retorna um inteiro, que representa o número de combinações necessárias para alcançar a medida recebida.

```
int lista::combina_frascos(int mililitros, lista* lista_frascos){
    if(mililitros == 0){
        return 0;
    }
    no* no_auxiliar_frascos = lista_frascos->cabeca;
    no* no_auxiliar_medidas = this->cabeca;
    int soma;
    int subtracao;

    while(no_auxiliar_medidas != nullptr){
        no_auxiliar_frascos = lista_frascos->cabeca;
        while(no_auxiliar_frascos != nullptr){
            soma = no_auxiliar_medidas->mililitros + no_auxiliar_frascos->mililitros;
            subtracao = no_auxiliar_medidas->mililitros - no_auxiliar_frascos->mililitros;

            if(soma > 0){
            this->insere_no(soma, no_auxiliar_medidas->operacoes + 1);
            }

            if(subtracao > 0){
                this->insere_no(subtracao, no_auxiliar_medidas->operacoes + 1);
            }
            if(soma == mililitros || subtracao == mililitros){
                return no_auxiliar_medidas->operacoes + 1;
            }
            no_auxiliar_frascos = no_auxiliar_frascos->proximo;
        }
        no_auxiliar_medidas = no_auxiliar_medidas->proximo;
    }
    return 0;
}
```

Faz dois loops encadeados percorrendo ambas as listas fazendo as combinações. As combinações são feitas da “lista” de medidas para a “lista” de frascos, e as combinações são armazenadas na lista de medidas utilizando o método “insere_no”. A função termina quando é encontrado o valor recebido como parâmetro e retorna o número de operações que ele necessitou para ser formado.

2.2.2. Método "esvazia_lista"

Este método não recebe nada como parâmetro e não retorna nada.

```
void lista::esvazia_lista(){
    no* auxiliar_fixo = this->cabeca;
    no* auxiliar;

    while(auxiliar_fixo->proximo != nullptr){
        auxiliar = auxiliar_fixo->proximo;
        auxiliar_fixo->proximo = auxiliar->proximo;
        if(auxiliar->proximo == nullptr){
            this->cauda = auxiliar_fixo;
        }
        delete auxiliar;
    }
}
```

Tem um loop que percorre toda a lista deletando todos os “nos”, exceto o primeiro, pois ele é o caso base que deverá ter em todas as chamadas. É sempre chamado quando queremos fazer uma nova medição.

2.3. Compilador

O compilador utilizado é o g++. O padrão do Linux.

3. Instruções de compilação e execução

Vá até a pasta src, pelo terminal, e digite “make” para compilar. Para executar o programa digite “./tp1”. Há também a opção “make clean” que apaga o arquivo gerado pela compilação.

Para executar o programa basta digitar “./tp1” e ele já estará funcionando. Para adicionar um frasco digite o tamanho do mesmo, em mililitros, e a letra “i”, para remover digite o tamanho do frasco, novamente em mililitros, e a letra “r”. Para realizar uma medição digite a quantidade e a letra “p” e aparecerá a quantidade de operações necessárias. Para finalizar digite “Ctrl” e “D”.

4. Análise de complexidade

4.1. Função “insere_no”

A função “insere_no” tem sempre a complexidade temporal O(1), já que sempre adiciona no final sem percorrer a lista, pois existe o ponteiro para a última posição. Em relação a complexidade espacial é sempre O(1), pois adiciona um “no”.

4.2. Função “remove_no”

“remove_no” tem complexidade temporal O(1) no melhor caso, que seria retirar o primeiro elemento da lista, visto que seriam feitas apenas duas mudanças de apontadores. No pior caso tem complexidade O(n), pois teria que percorrer toda a lista para procurar o valor a ser removido, fazendo novamente 2 mudanças de apontadores. Para a complexidade espacial a complexidade é O(1), pois instancio dois “nos” para serem auxiliares que são deletados depois.

4.3. Função “deleta_lista”

A função “deleta_lista” tem complexidade temporal O(n) pois percorre toda a lista apagando todos os “nos” após o primeiro e, por último, o primeiro. A complexidade espacial é O(1), pois são alocados dois “nos” para servirem como auxiliares que são deletados depois.

4.4. Função “esvazia_lista”

A função “esvazia_lista” tem complexidade temporal O(n), pois ela percorre toda a lista apagando todos os “nos” exceto o primeiro. Já a complexidade espacial é O(1) pois inicializo dois ponteiros do tipo “no” para serem auxiliares.

4.5. Função “combina_frascos”

Considerando as comparações “indicador de complexidade”, temos que a complexidade temporal no melhor caso será de O(1), visto que o melhor caso é quando a medida é 0. Já no pior caso será de $\Sigma_{i=0}(2n)^i$ sendo n o número de recipientes disponíveis, com i indo até o nível mais profundo da “árvore” formada pelas combinações. Isso se deve ao fato de que para cada recipiente são feitas duas operações, soma e subtração, e ambas são comparadas para poderem ser inseridas utilizando o método “insere_no”, de O(1).

A complexidade espacial, por sua vez, tem quase a mesma complexidade do que a temporal, pois ela não cria apenas os nós que são menores ou iguais a 0. Isso ocorre esporadicamente, não mudando consideravelmente a função.

5. Conclusão

O problema foi resolvido por meio da "força bruta" testando todos os valores de um jeito que apareçam todas as permutações de frascos possíveis até chegar no valor pretendido. A solução encontrada é uma interação entre duas listas até que se encontre o resultado, sendo que pode não haver um resultado em tempo hábil.

Dada a complexidade da função que faz todo o processo de combinações é difícil dizer se há soluções mais eficientes. Mesmo sendo "rápida" a solução ocupa muita memória desnecessária, há uma maneira de ocupar menos, porém iria aumentar muito a complexidade temporal.

No geral não tive grandes dificuldades em implementar o projeto, mas passei alguns dias pensando em como faria tudo antes de começar a implementar.

Referências

T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 2nd edition, MIT Press & McGraw-Hill, 2001.