**Mochila Fracionária (Fractional Knapsack Problem)**

Dado n objetos com pesos p1, p2, ..., pn e valores v1, v2, ..., vn e uma mochila com capacidade c, o problema consiste em encontrar o conjunto de itens que podemos armazenar na mochila de tal modo que seu valor seja o maior possível. Para este tipo de problema, podemos escolher uma fração x do objeto (0≤x≤1).

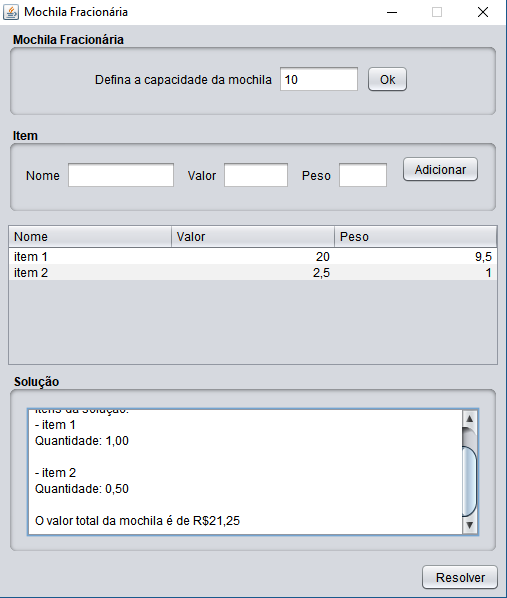
Para encontrarmos a solução ótima, utilizamos a técnica de algoritmos gulosos, ou seja, tomamos as decisões de forma isolada e esperamos que a solução local seja também a solução global. Neste caso, a abordagem gulosa é capaz de nos fornecer a solução ótima global.

Na interface do programa, deve-se especificar a capacidade máxima da mochila e adicionar pelo menos um item, fornecendo o nome, valor e peso do mesmo. Tanto o valor quanto o peso podem ser valores reais. Depois que os itens foram adicionados, o usuário deve clicar em “Resolver” no canto inferior direito.

Os itens especificados pelo usuário são armazenados em um ArrayList. Para implementar a abordagem gulosa, devemos ordenar essa lista de modo decrescente em relação ao valor do item, ou seja, do maior valor para o menor valor. Tivemos a preocupação de utilizar um método de ordenação que não fosse muito custoso, então optamos pelo QuickSort, que possui complexidade O(nlogn).

A função que resolve o problema é simples. Vamos escolhendo sempre os primeiros itens da lista (os de maior valor) e atualizando a capacidade restante. Caso um item não caiba inteiro na mochila, dividimos a capacidade restante pelo peso do objeto. Desse modo, conseguimos determinar a fração daquele item que conseguiremos armazenar.

Como a função que resolve o problema percorre a lista apenas uma vez, sua complexidade é O(n). Entretanto, considerando o método de ordenação implementado, a complexidade acaba sendo O(nlogn).



Exemplo de entrada para o problema da mochila fracionária

**Codificação de Huffman**

A codificação de Huffman é uma técnica gulosa que tem como objetivo reduzir o espaço gasto para armazenar uma sequência de caracteres. Cada caractere recebe um código baseado na frequência com a qual ele aparece no texto. Desse modo, um caractere que aparece mais vezes terá um código menor, e um que aparece menos vezes terá um código maior.

O primeiro passo do algoritmo é criar um vetor de inteiros para armazenar a frequência de cada caractere. Esse vetor possui 93 posições, e usamos os próprios índices para “acessar” um caractere. Vamos supor, por exemplo, que desejamos incrementar a frequência do caractere ‘A’. Sua respectiva posição será [(int) ‘A’ – 32], ou seja, subtraímos 32 do valor decimal da tabela ASCII pois o primeiro caractere que podemos representar é o espaço (cujo código é 32) e o último é o ‘}’ (cujo código é 125). Assim, podemos representar até 125-32=93 caracteres.

Para as próximas etapas, iremos utilizar a estrutura NoHuffman, que possui os seguintes atributos:

private char caractere;

private int frequencia;

private NoHuffman esq;

private NoHuffman dir;

private NoHuffman pai;

private NoHuffman prox;

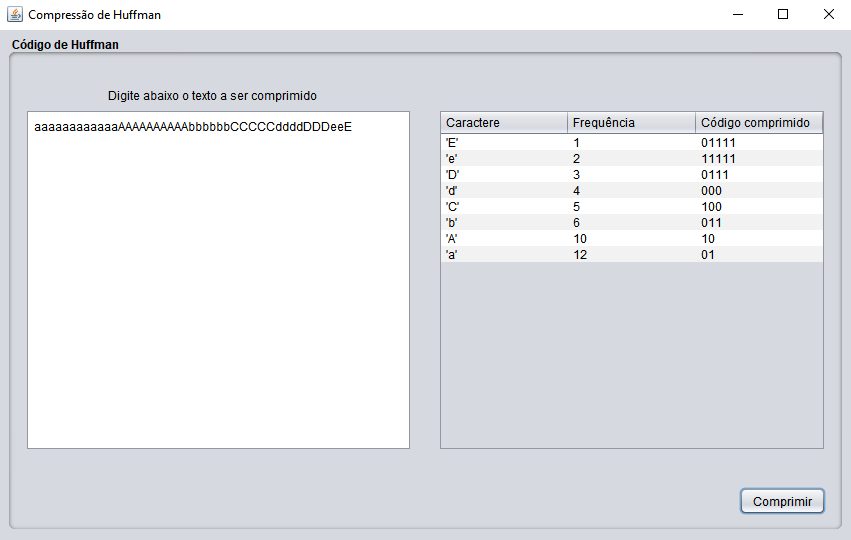
Depois que criamos e atualizamos nosso vetor de frequência, a próxima etapa é adicionar apenas os caracteres utilizados no texto e suas respectivas frequências em uma fila com prioridade. Isso é feito utilizando a estrutura acima. Assim, conseguimos deixar os elementos ordenados crescentemente de acordo com a frequência de cada um.

O próximo passo é montar a árvore. Para isso, removemos os dois primeiros nós da fila, criamos um único pai para os dois e colocamos o pai na fila. Atribuímos o caractere ‘~’ aos nós pais para indicar que eles são auxiliares, e a frequência dos mesmos deve ser a soma da frequência dos dois filhos. Repetimos esse processo até que só tenhamos um nó na fila.

Depois desse processo, nossa árvore de compressão já está criada. Basta agora recuperarmos o código de cada caractere, percorrendo a árvore de baixo para cima. Se o nó em que estamos for filho à esquerda, concatenamos ‘0’ no código. Se ele for filho à direita, concatenamos ‘1’. Fazemos isso até que cheguemos na raiz.

Como um caractere armazenado na tabela ASCII ocupa 8 bits, o código de Huffman faz com que necessitemos de menos bits para armazenar alguns caracteres. Para o exemplo abaixo, nosso código comprimido ocuparia 116 bits contra os 344 bits que seriam necessários caso não houvesse compressão. Nesse caso, temos uma economia de 228 bits.

Por utilizarmos uma estrutura de árvore e realizarmos n iterações, a complexidade O(nlogn).



Exemplo de entrada para o problema da compressão de Huffman