1 O Problema do Troco

O *problema do troco* consiste em determinar como formar um valor monetário específico utilizando moedas de diferentes denominações disponíveis.

Por exemplo, suponha que se deseja obter um valor total de 6 unidades, e há moedas de valores 1, 3 e 4. Algumas combinações possíveis são $1 + 1 + 4$, $3 + 3$ ou $4 + 1 + 1$. Dependendo do objetivo, pode-se querer apenas saber se é possível formar o valor, quantas formas diferentes existem, ou qual é o menor número de moedas necessário.

O desafio surge porque escolher sempre a moeda de maior valor nem sempre leva à melhor solução. Assim, é necessário avaliar combinações de moedas que, juntas, somam exatamente o valor desejado.

2 Uma Solução Gulosa

Uma estratégia natural é usar uma abordagem *gulosa*: em cada passo, escolhe-se a moeda de maior valor que não ultrapasse o valor restante. A ideia é "pegar o máximo possível" a cada escolha, até completar o valor desejado.

Exemplo Intuitivo

Suponha moedas de valores $1, 3, 4$ e que se deseja formar o valor 6:

- A moeda de maior valor menor ou igual a 6 é 4. Pegamos 4, restando 2.
- A moeda de maior valor menor ou igual a 2 é 1. Pegamos 1, restando 1.
- Novamente, pegamos 1. Restam 0 unidades e terminamos.

Resultado: $4 + 1 + 1$ (3 moedas). Note que esta solução não é ótima, pois a melhor solução seria $3 + 3$ (2 moedas).

Pseudocódigo

```
function troco_guloso(valor, moedas):
    troco = []
    moedas = sort(moedas, decrescente)

    while valor > 0:
        for moeda in moedas:
            if moeda <= valor:
                troco.append(moeda)
                valor -= moeda
                break
    return troco
```

Observação: A abordagem gulosa é simples e rápida, mas nem sempre produz o menor número de moedas. Ela funciona de forma ótima apenas para alguns conjuntos de moedas, como o sistema monetário padrão.

3 Programação Dinâmica para o Problema do Troco

Para encontrar o **número mínimo de moedas** necessário para formar um valor x, podemos construir uma relação de recorrência baseada nas escolhas de moedas.

Seja $solve(x)$ o número mínimo de moedas para formar o valor x, e seja C o conjunto de moedas disponíveis. Então, podemos escrever:

$$solve(x) = \begin{cases} \infty, & \text{se } x < 0, \\ 0, & \text{se } x = 0, \\ \min_{c \in C}\{solve(x-c)+1\}, & \text{se } x > 0. \end{cases}$$

A ideia é simples:

- Para $x = 0$, não é necessária nenhuma moeda.
- Para $x < 0$, a solução é inválida (∞).
- Para $x > 0$, tentamos cada moeda $c \in C$, e adicionamos 1 à solução do valor restante $x - c$. O mínimo entre todas essas opções dá a solução ótima.

Exemplo

Suponha moedas $C = \{1, 3, 4\}$ e queremos $x = 10$:

$$solve(10) = \min\{solve(9) + 1, solve(7) + 1, solve(6) + 1\}.$$

Cada termo da recorrência corresponde a usar uma moeda diferente e resolver o subproblema restante. Construindo a tabela de valores de $solve(x)$ do menor para o maior, garantimos encontrar a solução ótima de forma eficiente.

4 Otimização com Memoization

A implementação recursiva direta da função $solve(x)$ recalcula repetidamente os mesmos subproblemas. Por exemplo, ao calcular $solve(10)$ com moedas $C = \{1, 3, 4\}$, subproblemas como $solve(6)$ ou $solve(7)$ podem ser resolvidos várias vezes, tornando o algoritmo exponencial em tempo.

A **memoization** consiste em armazenar os resultados de subproblemas já resolvidos em uma estrutura de dados, como um vetor ou dicionário, de modo que, se o mesmo subproblema for necessário novamente, podemos simplesmente recuperar o resultado armazenado.

Em C++, isso pode ser implementado da seguinte forma:

```
int solve(int x, const std::vector<int>& moedas, std::vector<int>& memo) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    if (memo[x] != -1) return memo[x]; // resultado já calculado

    int best = INF;
    for (auto c : moedas)
        best = std::min(best, solve(x - c, moedas, memo) + 1);

    return memo[x] = best;
}
```

Aqui, o vetor `memo` é inicializado com um valor indicador (por exemplo, -1) para todos os índices. Antes de calcular $solve(x)$, verificamos se já existe um valor armazenado. Se existir, retornamos imediatamente, evitando recomputações desnecessárias.

Com memoization, a complexidade do algoritmo passa a ser $\mathcal{O}(valor \times |C|)$, garantindo que cada subproblema seja resolvido no máximo uma vez. Assim, conseguimos combinar a clareza da abordagem recursiva com a eficiência de uma implementação dinâmica.