UNIVERSIDADE FEDERAL DO CEARÁ - CAMPUS SOBRAL
CURSO DE ENGENHARIA DE COMPUTAÇÃO
DISCIPLINA: CONSTRUÇÃO E ANÁLISE DE ALGORITMOS
PROFESSOR: DR. ANTÔNIO JOSEFRAN DE OLIVEIRA BASTOS

TROCO ÓTIMO: UM COMPARATIVO ENTRE ABORDAGEM GULOSA E PROGRAMAÇÃO DINÂMICA

CAIO VINÍCIUS MAGALHÃES LUSTOZA 509872
PEDRO RICKSON FERNANDES ARAGÃO 542619
RAFAEL BENVINDO HOLANDA MENDES 415546

SOBRAL - CE
2025

SUMÁRIO

1 DESCRIÇÃO DO PROBLEMA

O problema do “Troco Ótimo“ é um problema clássico de otimização combinatória, consistindo em dado um conjunto de moedas com diferentes valores e um valor alvo específico, determinar a quantidade mínima de moedas necessária para compor o valor requerido. Esse problema possui diversas soluções, dentre elas as mais comuns são abordagens recursivas, abordagens gulosas e programação dinâmica. Neste estudo, iremos abordar especificamente as soluções que utilizam abordagem gulosa e programação dinâmica.

1.1 Definição Formal

- **Entrada:** um conjunto de números inteiros positivos $S = \{s_1, s_2, \ldots, s_n\}$ ordenados crescentemente [1], representando os valores disponíveis das moedas e um valor alvo $V \in \mathbb{N}^+$.
- **Saída:** o menor número inteiro $k \in \mathbb{N}$ tal que existam coeficientes $x_1, x_2, \ldots, x_n \in \mathbb{N}$ que satisfaçam a equação $\sum_{i=1}^{n} x_i \cdot s_i = V$, onde $k = \sum_{i=1}^{n} x_i$. Caso não exista tal combinação, indica-se que V não pode ser formado com as moedas disponíveis.

[1] Com o conjunto de moedas ordenado, evita-se a aplicação de algoritmos de ordenação no conjunto s.

2 ABORDAGEM GULOSA

```
1 def greedy_coin_change(s, v):
2     k = 0
3 
4     for i in range(len(s) - 1, -1, -1):
5         while v >= s[i]:
6             v -= s[i]
7             k += 1
8 
9     return k if v == 0 else None
```

A função `greedy_coin_change(s, v)` acima propõe uma solução baseada em abordagem gulosa, tomando sempre a maior moeda possível para compor a soma do valor alvo. A linha 2 inicializa a variável `k` em zero, é a partir dela que irá ser feita a contagem da quantidade total de moedas utilizadas na construção da solução.

A linha 4 inicia uma iteração sobre o vetor `s`, que representa o conjunto de moedas disponíveis. Como a entrada do problema exige que o conjunto de moedas esteja ordenado de maneira crescente, a iteração é feita em ordem decrescente dos indíces do conjunto `s`, partindo do último elemento até o primeiro, assim o último elemento representa a moeda de maior valor e o primeiro elemento representa a moeda de menor valor.

Na linha 5, é avaliada a possibilidade de utilizar a moeda atual `s[i]` para compor a soma do valor alvo `v`. Enquanto a moeda puder ser utilizada na soma do valor alvo, isto é, enquanto `v` for maior ou igual a `s[i]`, o algoritmo entra no laço e utiliza essa moeda na solução.

A linha 6 executa a subtração do valor da moeda `s[i]` do valor restante `v`, efetivamente reduzindo o problema para um subproblema menor, no qual parte do valor já foi solucionado. Vale destacar que na primeira execução do laço o valor `v` representa o valor alvo inserido na entrada do problema. Entretanto, com o passar das execuções esse valor `v` vai sendo atualizado de maneira a representar o restante que falta para atingir o valor alvo inicial após o uso de cada moeda.

Na linha 7, o contador `k` é incrementado em uma unidade, pois uma moeda foi efetivamente utilizada na composição do valor alvo. Esse incremento ocorre dentro do laço `while`, o que significa que uma mesma moeda poderá ser utilizada múltiplas vezes enquanto for válida para o valor restante.

Por fim, a linha 9 realiza o retorno da solução. Caso o valor `v` restante seja zero, então uma combinação válida foi encontrada e o número total de moedas utilizadas `k` é retornado. Caso contrário, se ainda restar algum valor `v` diferente de zero, a função retorna `None`, indicando que a estratégia gulosa não conseguiu encontrar uma solução viável para o problema dado o conjunto de moedas `s` fornecido na entrada.

2.1 Exemplos

A seguir são apresentados dois exemplos que demonstram o comportamento da abordagem gulosa para o problema do "Troco Ótimo". O primeiro exemplo mostra um caso em que a estratégia gulosa encontra a solução ótima, enquanto o segundo evidencia uma situação em que a abordagem falha, retornando uma solução subótima

2.1.1 Caso Ideial

Considere o conjunto de moedas $s = \{1, 5, 10, 25\}$ e o valor alvo $v = 30$. Para este caso a solução ótima utiliza apenas 2 moedas, sendo esta solução $25 + 5 = 30$. Vejamos a baixo como o algoritmo guloso, funciona corretamente para esse conjunto de moedas e valor alvo.

Inicialização

$$k = 0, \quad v = 30$$

Iteração para i = 3

$$v = 30 \geq s[i] = 25 \Rightarrow v = 30 - 25 = 5, k = 1$$

$$v = 5 \ngeq s[i] = 25 \Rightarrow \texttt{while} \text{ interrompido, } \texttt{i} \text{ é decrementado}$$

Iteração para i = 2

$$v = 5 \not\geq s[i] = 10 \Rightarrow \texttt{while} \text{ interrompido, } \texttt{i} \text{ é decrementado}$$

Iteração para i = 1

$$v = 5 \geq s[i] = 5 \Rightarrow v = 5 - 5 = 0, k = 2$$

$$v = 0 \not\geq s[i] = 5 \Rightarrow \texttt{while} \text{ interrompido, } \texttt{i} \text{ é decrementado}$$

Iteração para i = 0

$$v = 0 \not\geq s[i] = 1 \Rightarrow \texttt{while} \text{ interrompido, } \texttt{i} \text{ é decrementado}$$

Resultado Retornado

k = 2

2.1.2 Caso Falhos

Considere agora o conjunto de moedas $s = \{1,3,4\}$ e o valor alvo $v = 6$. Neste caso, a solução ótima utiliza-se apenas de 2 moedas, sendo essa solução representada pela soma $3+3 = 6$. Entretanto, como iremos verificar a baixo, a abordagem gulosa falhará neste caso por não considerar o impacto futuro de uma escolha local, tendo como resposta final $k = 3$, ou seja $4+1+1 = 6$

Inicialização

$$k = 0, \quad v = 6$$

Iteração para i = 2

$$v = 6 \geq s[i] = 4 \Rightarrow v = 6 - 4 = 2, k = 1$$

$$v = 2 \ngeq s[i] = 4 \Rightarrow \texttt{while} \text{ interrompido, } \texttt{i} \text{ é decrementado}$$

Iteração para i = 1

$$v = 2 \ngeq s[i] = 3 \Rightarrow \texttt{while} \text{ interrompido, } \texttt{i} \text{ é decrementado}$$

Iteração para i = 0

$$v = 2 \geq s[i] = 1 \Rightarrow v = 2 - 1 = 1, k = 2$$

$$v = 1 \geq s[i] = 1 \Rightarrow v = 1 - 1 = 0, k = 3$$

$$v = 0 \ngeq s[i] = 1 \Rightarrow \texttt{while} \text{ interrompido, } \texttt{i} \text{ é decrementado}$$

Resultado Retornado

k = 3

2.2 Complexidade

A abordagem gulosa tenta construir uma solução viável selecionando iterativamente as maiores moedas possíveis até que o valor-alvo seja atingido ou não possa mais ser reduzido. Essa estratégia apresenta complexidade de tempo $O(n+k)$, em que n refere-se ao número de moedas no conjunto s ordenado crescentemente, e k é o número de moedas efetivamente utilizadas na composição do troco. A complexidade de espaço é $O(1)$, pois o algoritmo utiliza apenas variáveis escalares para controle da execução e não aloca nenhuma estrutura auxiliar. Embora essa abordagem seja eficiente e muito mais leve em termos computacionais, ela não garante soluções corretas para todos os casos.

2.3 Desmonstração de Corretude

A heurística aplicada na abordagem gulosa deste estudo, seleciona a maior moeda possível a cada iteração que não ultrapasse o valor restante a ser alcançado. Apesar de ser eficiente, essa abordagem não garante corretude para todos os conjuntos de moedas. A seguir, demonstramos a corretude do algoritmo sob a suposição de que o conjunto de moedas escolhido alcançará a solução ótima.

Seja $S = \{s_1, s_2, \ldots, s_n\}$ um conjunto de moedas ordenado em ordem crescente, e $V \in \mathbb{N}^+$ o valor alvo. O algoritmo guloso percorre o conjunto S em ordem decrescente, subtraindo iterativamente a maior moeda s_i tal que $s_i \leq V$ até que o valor residual v seja zero. A cada subtração, o contador de moedas utilizado é aumentado em um.

Para demonstrar a corretude do algoritmo sob tais condições, adotamos a técnica do **laço invariante**, complementada por uma **prova por contradição**.

Invariante de Laço

Ao início de cada iteração do laço externo (que percorre o conjunto de moedas de forma decrescente), o valor residual v representa o montante que ainda precisa ser formado, e o número de moedas utilizadas até então, k, corresponde a uma solução parcial que utiliza moedas exclusivamente do subconjunto $\{s_{i+1}, \ldots, s_n\}$.

Inicialização

Antes da primeira iteração, $v = V$ e $k = 0$, ou seja, nenhuma moeda foi utilizada. O invariante é válido nesse ponto.

Manutenção

Em cada iteração, o algoritmo verifica se $s_i \leq v$. Caso positivo, subtrai s_i de v e incrementa k. A nova solução parcial continua utilizando moedas no subconjunto $\{s_i, s_{i+1}, \ldots, s_n\}$, mantendo o invariante.

Término

Ao fim do algoritmo, temos duas possibilidades: (i) $v = 0$, e portanto uma combinação válida foi encontrada com k moedas; ou (ii) $v \neq 0$, e não foi possível formar o valor alvo com as moedas disponíveis.

Conclusão

Suponha que o algoritmo retorna uma solução com k moedas, mas exista uma solução ótima com $k' < k$ moedas. Provaria-se então, que para o conjunto de moedas s essa solução alternativa utilizaria combinações menos eficientes, contradizendo a escolha gulosa da maior moeda possível a cada passo. Assim, por contradição, conclui-se que a solução encontrada é ótima. Portanto, a corretude do algoritmo guloso é garantida apenas sob a suposição de que o conjunto de moedas escolhido resulta numa solução ótima.

3 ABORDAGEM COM PROGRAMAÇÃO DINÂMICA

```python
1 def dp_coin_change(s, v):
2     dp = [v] * (v + 1)
3     dp[0] = 0
4 
5     for i in range(1, v + 1):
6         for coin in s:
7             if i - coin >= 0:
8                 dp[i] = min(dp[i], 1 + dp[i - coin])
9 
10     return dp[-1] if dp[-1] != v else None
```

A função `dp_coin_change(s, v)` acima propõe uma solução algorítmica baseada em Programação Dinâmica (*Dynamic Programming* - DP) para o problema do "Troco Ótimo". Na linha 2 o vetor `dp` é inicializado, ele será responsável por armazenar as soluções ótimas para todos os subproblemas de valor de `0` até `v`. O vetor possui `v + 1` posições, todas inicialmente preenchidas com o valor `v`, que representa um estado inviável ou um pior caso artificial, isto é, assumir que seriam necessárias `v` moedas de valor 1 para atingir o montante `v`. Essa escolha permite que a operação de minimização, realizada nas etapas seguintes, funcione corretamente desde a primeira iteração.

Na linha 3, define-se a condição base da abordagem de programação dinâmica, com `dp[0] = 0`. Isso indica que o número mínimo de moedas necessárias para formar o valor zero é exatamente zero. Essa base é fundamental para o funcionamento correto da estratégia de *bottom-up*, pois os demais valores de `dp[i]` dependerão dos resultados previamente computados, especialmente `dp[i - coin]`.

A linha 5 dá início à construção iterativa da solução, percorrendo todos os subvalores de `1` até `v + 1`. A cada iteração do laço externo, o algoritmo busca determinar a menor quantidade de moedas necessárias para compor o valor atual `i`, que vai crescendo unitariamente até assumir o valor alvo `v` utilizando as combinações possíveis de moedas do conjunto `s`. Essa abordagem assegura que os subproblemas sejam resolvidos de forma incremental e reutilizável.

Na linha 6, executa-se uma segunda iteração sobre cada moeda disponível no conjunto `s`, testando-as como candidatas a compor a soma atual `i`. Para cada moeda `coin`, a linha 7 verifica se a subtração `i - coin` resulta em um valor não negativo. Essa verificação é essencial para garantir que apenas moedas viáveis sejam consideradas para a construção do valor `i`.

A linha 8 contém o passo central do algoritmo, responsável pela atualização do estado `dp[i]`. A expressão `dp[i] = min(dp[i], 1 + dp[i - coin])` compara a solução armazenada atualmente em `dp[i]` com uma nova possibilidade, a de adicionar uma moeda `coin` à melhor solução conhecida para `i - coin`. Essa operação reflete a relação de recorrência característica da programação dinâmica, onde a solução de um problema depende das soluções ótimas de subproblemas menores.

Por fim, a linha 10 realiza a verificação final. Se `dp[v]` for diferente de `v`, significa que uma combinação de moedas foi encontrada para compor exatamente o valor `v`, e o valor mínimo de moedas é retornado. Caso contrário, retorna-se `None`, indicando que nenhuma combinação possível de moedas foi capaz de atingir exatamente o valor-alvo, e portanto o problema é considerado sem solução no domínio fornecido.

3.1 Subestrutura Ótima

No caso do algoritmo `dp_coin_change(s, v)`, a subestrutura ótima é claramente observável. Já que, a cada valor `i` no intervalo de `1` a `v`, o algoritmo procura a melhor solução possível utilizando a melhor solução previamente computada para `i - coin`, onde `coin` representa um dos valores disponíveis no conjunto de moedas `s`. A recorrência utilizada para expressar essa ideia é dada por:

$$dp[i] = \min\left(dp[i], 1 + dp[i - \text{coin}]\right), \quad \text{para todo coin} \in s \text{ tal que } i - \text{coin} \geq 0$$

Essa equação descreve como o valor ótimo para o subproblema de valor `i` depende diretamente das soluções ótimas de subproblemas de valores menores. A lógica consiste em adicionar uma moeda `coin` à solução previamente conhecida para `i - coin` e, em seguida, verificar se essa nova combinação reduz a quantidade total de moedas necessárias para compor o valor `i`. Como o algoritmo itera sobre todas as moedas possíveis para cada valor `i`, ele garante que todas as possibilidades viáveis serão avaliadas.

Além da subestrutura ótima, o algoritmo também explora a existência de subproblemas sobrepostos, característica igualmente essencial para a aplicação de programação dinâmica. Diferente de abordagens puramente recursivas que podem recalcular os mesmos subproblemas múltiplas vezes, o uso de um vetor `dp` permite armazenar os resultados intermediários e reutilizá-los sempre que necessário. Por exemplo, para calcular `dp[7]`, o algoritmo pode necessitar de `dp[6]`, `dp[5]`, e assim por diante, dependendo do conjunto de moedas. Esses mesmos subvalores `dp[i - coin]` serão reutilizados quando `i` avançar, como ao calcular por exemplo `dp[8]` e `dp[9]`, o que evidencia que diversos estados são revisitados ao longo da computação.

3.2 Exemplo

Considere o conjunto de moedas $s = \{1,3,4\}$ e o valor alvo $v = 6$. O vetor `dp` é inicializado com `v + 1` posições, contendo o valor `v` em todas, exceto em `dp[0]` que recebe `0`.

$$\texttt{dp} = [0,6,6,6,6,6,6]$$

Iteração para $i = 1$

$$\texttt{coin} = 1 \Rightarrow dp[1] = \min(6, 1 + dp[0]) = 1$$

$$\texttt{dp} = [0,1,6,6,6,6,6]$$

Iteração para $i = 2$

$$\texttt{coin} = 1 \Rightarrow dp[2] = \min(6, 1 + dp[1]) = 2$$

$$\texttt{dp} = [0,1,2,6,6,6,6]$$

Iteração para $i = 3$

$$\texttt{coin} = 1 \Rightarrow dp[3] = \min(6, 1 + dp[2]) = 3$$

$$\texttt{coin} = 3 \Rightarrow dp[3] = \min(3, 1 + dp[0]) = 1$$

$$\texttt{dp} = [0,1,2,1,6,6,6]$$

Iteração para $i = 4$

$$\texttt{coin} = 1 \Rightarrow dp[4] = \min(6, 1 + dp[3]) = 2$$

$$\texttt{coin} = 3 \Rightarrow dp[4] = \min(2, 1 + dp[1]) = 2$$

$$\texttt{coin} = 4 \Rightarrow dp[4] = \min(2, 1 + dp[0]) = 1$$

$$\texttt{dp} = [0,1,2,1,1,6,6]$$

Iteração para $i = 5$

$$\texttt{coin} = 1 \Rightarrow dp[5] = \min(6, 1 + dp[4]) = 2$$

$$\texttt{coin} = 3 \Rightarrow dp[5] = \min(2, 1 + dp[2]) = 2$$

$$\texttt{coin} = 4 \Rightarrow dp[5] = \min(2, 1 + dp[1]) = 2$$

$$\texttt{dp} = [0,1,2,1,1,2,6]$$

Iteração para $i = 6$

$$\texttt{coin} = 1 \Rightarrow dp[6] = \min(6, 1 + dp[5]) = 3$$

$$\texttt{coin} = 3 \Rightarrow dp[6] = \min(3, 1 + dp[3]) = 2$$

$$\texttt{coin} = 4 \Rightarrow dp[6] = \min(2, 1 + dp[2]) = 2$$

$$\texttt{dp} = [0,1,2,1,1,2,2]$$

Resultado final

O menor número de moedas para somar exatamente 6 dado o conjunto de moedas inicial é 2, esse valor está representado no último índice do vetor `dp[-1]`. A única combinação possível para antigir o valor alvo a partir do conjunto de moedas dado na entrada é $3+3$.

3.3 Complexidade

A abordagem por programação dinâmica para o problema do "Troco Ótimo" apresenta complexidade de tempo $O(n*v)$, onde n representa a quantidade de denominações de moedas disponíveis no conjunto s, e v é o valor-alvo a ser alcançado. Essa complexidade decorre do fato de que o algoritmo percorre cada subvalor de 1 até v, e para cada um desses valores testa todas as n moedas disponíveis, avaliando se podem contribuir para uma solução ótima. Já a complexidade de espaço é $O(v)$, pois a estratégia emprega um vetor unidimensional dp com $v+1$ posições para armazenar o número mínimo de moedas necessárias para formar cada subvalor. Essa estrutura permite a reutilização eficiente de resultados previamente computados e evita recomputações, tornando a abordagem adequada para problemas em que todas as combinações viáveis devem ser consideradas para garantir uma solução exata, mesmo sob restrições arbitrárias de valores de moedas.

3.4 Desmonstração de Corretude

O algoritmo demonstrado nas seções anteriores, resolve o problema do troco ótimo construindo iterativamente soluções para todos os subproblemas de valores i tais que $0 \leq i \leq V$, armazenando o número mínimo de moedas necessárias em um vetor auxiliar dp, de dimensão $V+1$. Definimos $dp[i]$ como o número mínimo de moedas necessárias para formar o valor i. A relação de recorrência utilizada é dada por:

$$dp[i] = \min_{s_j \leq i}\{dp[i],\ 1 + dp[i - s_j]\}, \quad \forall s_j \in S$$

Com a condição inicial $dp[0] = 0$, indica-se que zero moedas são necessárias para formar o valor zero. A seguir, apresentamos uma demonstração de corretude do algoritmo utilizando uma **prova por indução** sobre o valor v.

Base da Indução

Temos que $dp[0] = 0$, o que é correto por definição, pois nenhuma moeda é necessária para formar o valor zero.

Hipótese de Indução

Suponha que para todo $k \leq v$, $dp[k]$ armazena corretamente o número mínimo de moedas necessárias para formar o valor k com as moedas do conjunto s.

Passo Indutivo

Mostremos que $dp[k+1]$ também é correto. O algoritmo considera todas as moedas $s_j \leq v+1$ e avalia a expressão $1 + dp[k+1-s_j]$. Pela hipótese de indução, $dp[k+1-s_j]$ representa corretamente o número mínimo de moedas para o subvalor $k+1-s_j$. Assim, adicionar uma moeda s_j a essa solução resulta em uma solução válida para $dp[v+1]$. A escolha do mínimo entre todas essas possibilidades garante que $dp[k+1]$ seja de fato o menor número de moedas necessário para formar $k+1$.

Conclusão

Por indução, todos os valores de $dp[0]$ até $dp[k]$ são corretamente preenchidos, e a função retorna $dp[k]$ como a solução ótima para o problema. Portanto, o algoritmo de programação dinâmica está provado como correto e ótimo para qualquer conjunto arbitrário de moedas finitas e positivas, independentemente da canonicidade.

4 EXECUÇÃO DO CÓDIGO

Para executar rodar o código deste projeto, é necessário que o interpretador `Python` esteja instalado na máquina local. A verificação pode ser feita executando o comando `python -version` no terminal. Caso o `Python` não esteja instalado, sua obtenção pode ser realizada por meio do site oficial: `https://www.python.org/downloads/`.

Com o ambiente devidamente configurado, o código pode ser obtido clonando o repositório com o comando `git clone https://github.com/caioviniciusml/coin-change.git`. Após a conclusão da clonagem do repositório, é necessário acessar o diretório do projeto com o comando `cd coin-change/` e, então, executar o script principal por meio de `python coin_change.py`.

Ao rodar o programa, o terminal solicitará dois conjuntos de entrada ao usuário. O primeiro corresponde ao conjunto de moedas s, que deve ser inserido em ordem crescente e separado por vírgulas, como no exemplo: `1, 2, 5, 10`. A segunda entrada refere-se a um conjunto de valores-alvo que se deseja analisar, também separados por vírgulas, por exemplo: `12, 129, 324`. O código realizará, para cada valor-alvo informado, a execução das duas abordagens implementadas retornando como saída o menor número de moedas necessárias para cada abordagem. Dessa forma, o usuário poderá comparar os resultados e observar diretamente os efeitos do conjunto de moedas sobre o desempenho e a correção de cada abordagem.