1. Busca Binária

Problema: Encontrar o índice de um valor em uma lista ordenada.

Solução Lógica:

1. **Inicialize:** Defina os ponteiros `esquerda` e `direita` para o início e fim da lista.
2. **Enquanto:** `esquerda` for menor ou igual a `direita`, faça o seguinte:
    - Calcule o índice `meio` como a média de `esquerda` e `direita`.
    - Compare o valor no índice `meio` com o valor alvo.
    - Se for igual, retorne `meio`.
    - Se o valor alvo for menor, ajuste `direita` para `meio - 1`.
    - Se o valor alvo for maior, ajuste `esquerda` para `meio + 1`.
3. **Se:** O valor não for encontrado, retorne `-1`.

EXEMPLO DE CÓDIGO:

```
def busca_binaria(lista, alvo):
    esquerda, direita = 0, len(lista) - 1  # Inicializa os ponteiros para o início e fim da lista
    while esquerda <= direita:  # Enquanto a busca não terminar
        meio = (esquerda + direita) // 2  # Encontra o índice do meio
        if lista[meio] == alvo:  # Se o valor no meio é o alvo
            return meio  # Retorna o índice do alvo
        elif lista[meio] < alvo:  # Se o valor no meio é menor que o alvo
            esquerda = meio + 1  # Ajusta o ponteiro da esquerda para o meio + 1
        else:  # Se o valor no meio é maior que o alvo
            direita = meio - 1  # Ajusta o ponteiro da direita para o meio - 1
    return -1  # Se o valor não for encontrado, retorna -1

# Exemplo de uso
lista = [1, 3, 5, 7, 9, 11]
alvo = 7
print(busca_binaria(lista, alvo))  # Saída: 3
```

Explicação:

1. **Inicialização: Define os ponteiros `esquerda` e `direita` para o início e fim da lista.**
2. **Loop de Busca: Enquanto `esquerda` é menor ou igual a `direita`, calcula o índice `meio` e compara o valor no índice `meio` com o valor alvo.**
3. **Ajuste dos Ponteiros: Ajusta os ponteiros dependendo se o valor no meio é menor ou maior que o alvo.**
4. **Retorno: Se o valor for encontrado, retorna o índice. Caso contrário, retorna `-1` se o alvo não estiver na lista.**

2. Problema da Mochila (Knapsack)

Problema: Dada uma mochila com capacidade limitada e uma lista de itens com peso e valor, determine o valor máximo que você pode carregar.

Solução Lógica:

1. **Inicialize: Crie uma tabela `dp` onde `dp[i][w]` representa o valor máximo com os primeiros `i` itens e capacidade `w`.**
2. **Preencha a Tabela:**
   - **Para cada item e cada capacidade, verifique se o item pode ser incluído.**
   - **Atualize o valor máximo com base na inclusão ou exclusão do item.**
3. **Resultado: O valor máximo para a capacidade total é `dp[n][capacidade]`.**

```
def knapsack(pesos, valores, capacidade):
    n = len(pesos)  # Número de itens
    dp = [[0] * (capacidade + 1) for _ in range(n + 1)]  # Tabela de DP

    for i in range(1, n + 1):  # Itera sobre todos os itens
        for w in range(capacidade + 1):  # Itera sobre todas as capacidades
            if pesos[i - 1] <= w:  # Se o item cabe na capacidade atual
                # Escolhe o máximo entre incluir ou não o item
                dp[i][w] = max(valores[i - 1] + dp[i - 1][w - pesos[i - 1]], dp[i - 1][w])
            else:
                dp[i][w] = dp[i - 1][w]  # Não inclui o item

    return dp[n][capacidade]  # Retorna o valor máximo que pode ser carregado

# Exemplo de uso
pesos = [1, 2, 3]
valores = [60, 100, 120]
capacidade = 5
print(knapsack(pesos, valores, capacidade))  # Saída: 220
```

Explicação:

1. **Inicialização:** Cria uma tabela `dp` onde `dp[i][w]` representa o valor máximo possível com os primeiros `i` itens e capacidade `w`.
2. **Preenchimento da Tabela:** Para cada item e capacidade, decide se inclui o item na mochila ou não, baseado na maximização do valor.
3. **Resultado:** Retorna o valor máximo que pode ser carregado com a capacidade total dada.

3. Busca em Profundidade (DFS)

Problema: Dado um grafo não direcionado, percorra todos os nós a partir de um nó inicial.

Solução Lógica:

1. **Inicialize:** Crie um conjunto `visitado` para rastrear os nós visitados.
2. **Recursivamente:** Para cada nó, adicione ao `visitado` e percorra todos os vizinhos não visitados usando DFS.

```
def dfs(grafo, no, visitado=None):
    if visitado is None:
        visitado = set()  # Inicializa o conjunto de nós visitados
    visitado.add(no)  # Marca o nó atual como visitado
    print(no, end=' ')  # Imprime o nó atual

    for vizinho in grafo[no]:  # Itera sobre todos os vizinhos do nó atual
        if vizinho not in visitado:  # Se o vizinho não foi visitado
            dfs(grafo, vizinho, visitado)  # Faz uma chamada recursiva para o vizinho

# Exemplo de uso
grafo = {
    'A': {'B', 'C'},
    'B': {'A', 'D'},
    'C': {'A', 'E'},
    'D': {'B'},
    'E': {'C'}
}
dfs(grafo, 'A')  # Saída: A B D C E
```

Método Steve Halim para Competição de Programação

1. Entendimento do Problema

- **Leia o Problema Cuidadosamente: Compreenda os requisitos e as restrições do problema. Certifique-se de entender o que está sendo pedido antes de começar a implementar uma solução.**
- **Identifique Exemplos e Testes: Analise exemplos fornecidos no problema e crie seus próprios casos de teste para garantir que compreendeu corretamente a tarefa.**

2. Desenvolvimento da Solução

- **Escolha da Estratégia: Identifique a abordagem ou algoritmo apropriado para resolver o problema. Isso pode incluir algoritmos clássicos como busca binária, programação dinâmica, grafos, etc.**
- **Esboço da Solução: Faça um esboço da solução e defina as etapas principais. Pode ser útil criar um diagrama ou pseudocódigo para visualizar a lógica.**
- **Análise de Complexidade: Avalie a complexidade temporal e espacial da solução para garantir que ela se encaixa nos limites impostos pelo problema.**

3. Implementação

- **Código Limpo e Testado: Escreva o código de forma clara e eficiente. Use boas práticas de programação, como nomes de variáveis descritivos e modularidade.**
- **Teste Extensivamente: Teste o código com diversos casos, incluindo casos limite e casos especiais, para garantir que ele funciona corretamente em todas as situações possíveis.**

4. Otimização e Ajustes

- **Ajustes Finais: Faça ajustes para melhorar a eficiência, se necessário. Isso pode incluir otimizações de algoritmos ou estrutura de dados.**
- **Verificação de Soluções: Compare a solução com a solução ideal e verifique se há alguma melhoria possível.**

Exemplos e Aplicações

Vamos ver como aplicar o método Steve Halim a alguns problemas típicos de maratonas de programação.

Exemplo 1: Busca Binária

Problema: Dada uma lista ordenada de números inteiros, encontre a posição de um número alvo.

Estratégia:

- **Entendimento: O problema pode ser resolvido de maneira eficiente usando a busca binária.**
- **Desenvolvimento: A busca binária divide a lista em metades para localizar o número.**
- **Implementação: Abaixo está a implementação em Python.**

python
Copiar código

```
def busca_binaria(lista, alvo):
    esquerda, direita = 0, len(lista) - 1
    while esquerda <= direita:
        meio = (esquerda + direita) // 2
        if lista[meio] == alvo:
            return meio
        elif lista[meio] < alvo:
            esquerda = meio + 1
        else:
            direita = meio - 1
    return -1

# Exemplo de uso
lista = [1, 3, 5, 7, 9]
alvo = 7
print(busca_binaria(lista, alvo))  # Saída: 3
```

Teste: Teste com diferentes listas e valores para garantir que a busca binária funciona corretamente.

Exemplo 2: Programação Dinâmica - Mochila

Problema: Dada uma mochila com capacidade máxima e uma lista de itens com pesos e valores, determine o valor máximo que pode ser carregado na mochila.

Estratégia:

- **Entendimento: Este é um problema clássico de programação dinâmica.**

- **Desenvolvimento: Crie uma tabela `dp` para armazenar o valor máximo para cada capacidade de mochila.**
- **Implementação: Abaixo está a implementação em Python.**

python
Copiar código

```
def knapsack(pesos, valores, capacidade):
    n = len(pesos)
    dp = [[0] * (capacidade + 1) for _ in range(n + 1)]

    for i in range(1, n + 1):
        for w in range(capacidade + 1):
            if pesos[i - 1] <= w:
                dp[i][w] = max(valores[i - 1] + dp[i - 1][w -
pesos[i - 1]], dp[i - 1][w])
            else:
                dp[i][w] = dp[i - 1][w]

    return dp[n][capacidade]

# Exemplo de uso
pesos = [1, 2, 3]
valores = [60, 100, 120]
capacidade = 5
print(knapsack(pesos, valores, capacidade))  # Saída: 220
```

Teste: Teste com diferentes capacidades e listas de pesos e valores.

Aplicabilidade

- **Busca Binária: Usada em problemas onde você precisa encontrar um item em uma lista ordenada, por exemplo, procurar um valor específico ou encontrar a posição onde um item deveria estar.**
- **Programação Dinâmica: Aplicável em problemas de otimização, como o problema da mochila, onde você precisa tomar decisões ótimas baseadas em subproblemas.**

Conclusão

O método Steve Halim oferece uma abordagem estruturada para resolver problemas de programação, desde a compreensão do problema até a implementação e otimização da solução. Seguir esses passos ajuda a resolver problemas de forma eficiente e eficaz, o que é essencial para ter sucesso em competições de programação e desafios de codificação.