INSTITUTO DE INFORMÁTICA
Universidade Federal de Goiás

Partição de Números

Prof. André Moura

INSTITUTO DE INFORMÁTICA
UFG

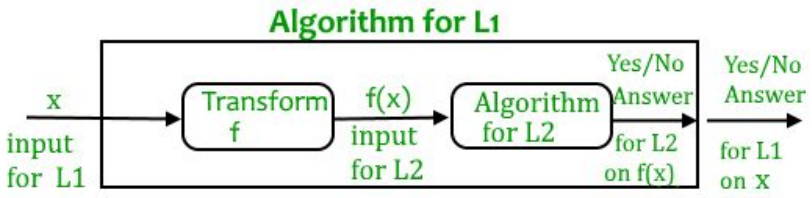
UFG
UNIVERSIDADE FEDERAL DE GOIÁS

Participantes

- ➢ Daired Almeida Cruz - 201912945
- ➢ Gustavo Rodrigues Ribeiro - 202003570
- ➢ Keven Lucas Vieira Gondim - 202000181
- ➢ Pedro Paulo Pereira Souza - 202103770

Introdução

O que é ser NP ?

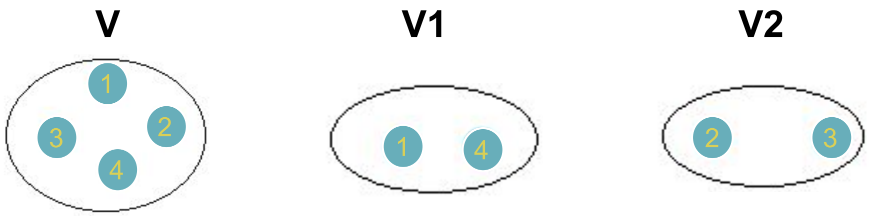


- **Um problema é NP se:**
  - Se existe um algoritmo não determinístico que o resolve em tempo polinomial.
- **Algoritmo não determinístico:**
  - Escolhas aleatórias;
  - Para uma mesma entrada, o algoritmo pode gerar saídas diferentes.
- **Logo, para qualquer entrada:**
  - O algoritmo é capaz de:
    - Solução correta em tempo polinomial;
    - Ou, não há solução em tempo polinomial.
- **Salto-nd:**
  - É uma instrução para fazer uma escolha aleatória;
  - Se houver solução, ele fornecerá um conjunto que resolve;
  - Se não houver solução, ele fornecerá um conjunto que não resolve.

Problema da *k*-Partição de Números (TWNPP)

- Dada uma sequência numérica *V* de inteiros positivos:
  - Encontrar uma partição de dimensão *k*, de modo que as somas dos elementos de cada parte sejam as mais próximas possíveis umas das outras;
  - Maior soma **aproximando-se** da menor soma;
  - Encontrar dois subconjuntos *V1* e *V2*, de modo que:
    - *V1* e *V2* sejam disjuntos;
    - |*V1*| + |*V2*| = |*V*|;
    - |*V1*| - |*V2*| seja minimizado.

- **EXEMPLO:**
  - Para *V* = {1, 2, 3, 4}, uma solução seria:
    - *V1* = {1, 4}
    - *V2* = {2, 3}
    - |*V1*| - |*V2*| = 0.



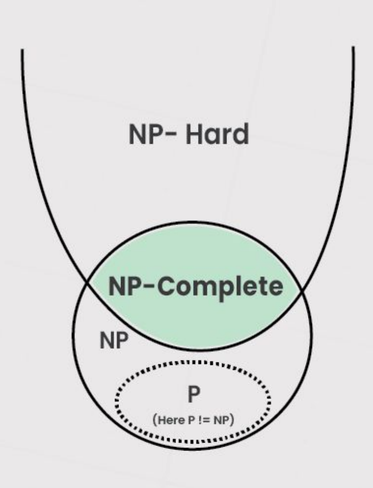
Aplicação do problema

- Alocação de recursos;
- Planejamento de produção;
- Otimização de portfólio;
- Teoria da informação.

Demonstração de pertencimento à classe NP

Prova

- **O problema da Partição de Números:**
  - Divide um conjunto de *n* números em dois subconjuntos.
    - A soma dos subconjuntos deve ser a mesma.
- **Seja S um conjunto de números:**
  - Subconjunto *A* de *n* números com soma *S1*;
  - Outro subconjunto com os elementos (*S* - *A*) com soma *S2*;
  - *S1* é igual a *S2*.
- **Devemos provar que:**
  - O problema em si é da classe NP;
  - Se for NP-Completo, provar que é NP-Difícil:
    - Todos os outros problemas da classe NP podem ser redutíveis a ele em tempo polinomial.



Prova

- **O Problema da Partição de Números está em NP:**
  - Dado um 'certificado':
    - Solução;
    - Instância do problema **(um conjunto *S* e duas partições *A* e *A'*)**.
  - Pode-se provar que o certificado está em tempo polinomial. Da seguinte forma:
    - **1.** Para cada elemento *x* em *A* e *x'* em *A'*, verificar que todos os elementos pertencentes a *S* estão cobertos;
    - **2.** Seja *S1* igual a 0, *S2* igual a 0;
    - **3.** Para cada elemento *x* em *A*, adicionar esse valor a *S1*;
    - **4.** Para cada elemento *x'* em *A'*, adicionar esse valor a *S2*;
    - **5.** Verificar que *S1* é igual a *S2*.
  - O algoritmo demora um **tempo linear** em relação ao **tamanho do conjunto de números.**
  - Logo, o problema da Partição de Números é um **NP**.
  - Para **NP-Difícil**, reduzir o Problema da Soma de Subconjuntos.

Algoritmos

Algoritmo guloso

```python
import random


def two_way_number_partitioning(V):
  """
  Resolve o problema da partição de números usando um algoritmo guloso.

  Args:
    V: Um conjunto de números inteiros positivos.

  Returns:
    A solução para o problema, representada por um tuplo (V1, V2), onde V1 e V2
    são os subconjuntos resultantes.
  """

  V1 = []
  V2 = []

  # Adiciona o número mais pequeno de V a um dos subconjuntos.

  n = min(V)
  V1.append(n)
  V.remove(n)

```

Algoritmo guloso

```
  # Adiciona os números restantes a um dos subconjuntos, de modo a minimizar
  # a diferença das somas dos subconjuntos.

  while V:
    if sum(V1) > sum(V2):
      V2.append(V.pop())
    else:
      V1.append(V.pop())

  return (V1, V2)


V = [i for i in range(1, 101)]

(V1, V2) = two_way_number_partitioning(V)
difference = len(V1) - len(V2)


print(V1, end="\n\n\n")
print(V2, end="\n\n\n")
print("|V1|-|V2| = {}".format(abs(difference)))
```

Algoritmo guloso - Descrição

Função two_way_number_partitioning

- A função two_way_number_partitioning, adiciona o número mais pequeno de V a um dos subconjuntos.
- Adiciona os números restantes a um dos subconjuntos, de modo a minimizar a diferença das somas dos subconjuntos.
- Para minimizar a diferença das somas dos subconjuntos, o algoritmo escolhe o subconjunto com a soma mais baixa para adicionar o próximo número. Se a soma do subconjunto com a soma mais baixa for maior ou igual à soma do subconjunto com a soma mais alta, o algoritmo escolhe o subconjunto com a soma mais alta para adicionar o próximo número.

Algoritmo guloso - Complexidade

- O que determina a complexidade do algoritmo será o laço while ao percorrer o vetor v, no entanto há presença das funções auxiliares append e pop, não poderão interferir em sua complexidade.

- No caso do número mais pequeno, a complexidade de tempo é constante (O(1)). No caso dos números restantes, a complexidade de tempo é linear (O(n)) no pior caso, pois o algoritmo precisa iterar sobre todos os números restantes para decidir qual subconjunto adicionar cada um deles.

Algoritmo guloso - Complexidade O(n)

- Portanto, a complexidade de tempo total do algoritmo de partição no melhor caso é:

$$O(1) + O(n) = O(n)$$

Algoritmo guloso - Complexidade $\Omega(n)$

- No pior caso, o algoritmo precisa iterar sobre todos os números do conjunto original, resultando em uma complexidade de tempo linear $\Omega(n)$.

- No entanto a função append também será de custo $\Omega(n)$, logo o custo e de $2\Omega(n)$.

Problemas com o algoritmo guloso

- Exemplo:

$$V = [1, 2, 3, 4, 5]$$

- O algoritmo guloso iria resolver este problema adicionando o número mais pequeno do conjunto, que é 1, a um dos subconjuntos. Este é um passo localmente ótimo, pois reduz a diferença entre as somas dos subconjuntos. No entanto, o algoritmo guloso continuaria adicionando os números restantes do conjunto ao mesmo subconjunto.

Problemas com o algoritmo guloso

- Como resultado, o algoritmo guloso retornaria um conjunto com apenas dois elementos, que são 1 e 5. Esta não é a solução ótima, pois a solução ótima é um conjunto com três elementos, que são 1, 2 e 3.

- Em geral, o algoritmo guloso pode falhar em casos em que a solução ótima não é a solução localmente ótima.

Algoritmo com Heuristica

```
def heuristic_number_partitioning(V):
    # Inicializa dois subconjuntos vazios
    V1 = []
    V2 = []

    # Ordena os números em ordem decrescente
    sorted_V = sorted(V, reverse=True)

    # Adiciona os números de forma alternada aos subconjuntos
    for i, num in enumerate(sorted_V):
        if i % 2 == 0:
            V1.append(num)  # Adiciona ao V1 se o índice for par
        else:
            V2.append(num)  # Adiciona ao V2 se o índice for ímpar

    return (V1, V2)

# Cria um conjunto de números de 1 a 100
V = [i for i in range(1, 101)]

# Chama a função heurística de partição
(V1, V2) = heuristic_number_partitioning(V)

# Calcula a diferença entre os tamanhos dos subconjuntos
difference = len(V1) - len(V2)

# Imprime os subconjuntos e a diferença absoluta entre seus tamanhos
print(V1, end="\n\n\n")
print(V2, end="\n\n\n")
print("|V1|-|V2| = {}".format(abs(difference)))
```

Algoritmo com heuristica - Descrição

Função heuristic_number_partitioning

- Inicialização de Subconjuntos:
  - V1 e V2 são inicializados como listas vazias que representarão os dois subconjuntos.
- Ordenação em Ordem Decrescente:
  - A função cria uma cópia ordenada decrescente do conjunto original V usando sorted(V, reverse=True). Isso coloca os maiores números no início da lista.

Algoritmo com heuristica - Deascrição

Função heuristic_number_partitioning

- Distribuição Alternada:
    - Utilizando um loop for e a função enumerate, os números ordenados são adicionados de forma alternada aos subconjuntos V1 e V2.
    - Se o índice (i) for par, o número é adicionado a V1.
    - Se o índice for ímpar, o número é adicionado a V2.
    - Retorno do Resultado:
    - A função retorna o par de subconjuntos (V1, V2).

Algoritmo com heuristica - Complexidade O(n)

- A complexidade do algoritmo depende do método de ordenação aplicado, como foi usado a biblioteca sort padrão do python, então o método de ordenação utilizado foi o algoritmo Tim Sort cuja complexidade é *O(n log n)*.

- Além disso, a distribuição alternada dos elementos na lista ordenada tem uma complexidade linear *O(n)*, onde n é o número de elementos.

- Portanto, a complexidade total do algoritmo é aproximadamente *O(n log n) + O(n)*.

Algoritmo com heuristica - Complexidade $\Omega(n)$

- O pior caso dependerá da função *append* que realiza inserções no fim da lista, durante o laço a função append em python tem custo *$O(1)$*, mas ela ocorre n vezes logo teremos :

$$\Omega(n \log n) + \Omega(n)$$

Algoritmo com heurística - Problemas

- **Não Garante a Solução Ótima:**

Este algoritmo usa uma heurística simples, que não garante encontrar a solução ótima para todos os conjuntos de entrada. Pode produzir uma solução próxima ao ótimo, mas não há garantia.

- **Dependência da Ordenação:**

A eficácia desse algoritmo depende fortemente da qualidade da ordenação inicial. Se a ordenação inicial não for eficiente, isso pode afetar a eficácia da heurística.

- **Sensibilidade à Distribuição Inicial:**

Pequenas variações na ordem dos números no conjunto de entrada podem levar a diferentes soluções.

Algoritmo com heurística - Problemas

- **Limitação do Problema Específico:**

Este algoritmo é específico para o problema da partição de números. Não é uma solução genérica para todos os problemas de particionamento.

- **Não Trata Casos Especiais:**

O código assume que o conjunto de entrada não está vazio. Não trata explicitamente casos especiais, como a entrada de um conjunto vazio.
Este algoritmo presume que os números são inteiros positivos. Se houver a possibilidade de números negativos no conjunto, o algoritmo pode não funcionar corretamente.

- **Sensível à Ordem dos Números:**

Pequenas variações na ordem dos números no conjunto de entrada podem levar a diferentes soluções.

Força Bruta

```python
from itertools import product

def partition_k(X, k):
    n = len(X)

    # Gera todas as combinações possíveis dos índices dos elementos
    index_combinations = product(range(k), repeat=n)

    unique_partitions = set()

    # Verifica se cada combinação atende aos critérios
    for indices in index_combinations:
        subsets = [[] for _ in range(k)]

        # Preenche os subsets com os elementos correspondentes
        for i, index in enumerate(indices):
            subsets[index].append(X[i])

        # Usa tuplas para representar cada subset
        tuple_subsets = [tuple(subset) for subset in subsets]
```

Força Bruta

```
            # Verifica se as somas são iguais e se não há elementos compartilhados entre os subsets
            sums = [sum(subset) for subset in tuple_subsets]
            if len(set(sums)) == 1 and len(set().union(*tuple_subsets)) == n:
                unique_partitions.add(tuple(tuple_subsets))

        # Imprime as partições únicas
        for i, partition in enumerate(unique_partitions):
            # Ordena os elementos dentro de cada subset
            formatted_partition = [', '.join(map(str, sorted(subset))) for subset in partition]
            output = f"{i}: ({'), ('.join(formatted_partition)})"
            print(output)

    # Exemplo de uso
    X = [2, 5, 4, 9, 1, 7, 6, 8]
    k = 3
    partition_k(X, k)
```

Algoritmo com Força Bruta - Descrição

- **Objetivo do Código**
  - O código visa encontrar partições únicas de uma lista *X* em *k* subsets, onde cada subset tem a mesma soma e nenhum elemento é compartilhado entre os subsets.

- **Processo de Geração de Partições:**
  - O código utiliza a função *product* da biblioteca itertools para gerar todas as combinações possíveis dos índices dos elementos da lista *X* em *k* grupos.
  - Para cada combinação de índices, são criados *subsets* correspondentes, e verifica-se se eles atendem às condições de igualdade de soma e ausência de elementos compartilhados.

Algoritmo com Força Bruta - Descrição

- **Verificação das Condições de Partição:**
  - A soma de cada subset é calculada, e é verificado se todas as somas são iguais (*len(set(sums)) == 1*).
  - Além disso, verifica-se se não há elementos compartilhados entre os subsets (*len(set().union(*tuple_subsets)) == n*, onde n é o comprimento da lista *X*).

- **Armazenamento de Partições Únicas:**
  - As partições que atendem às condições são armazenadas em um conjunto (*unique_partitions*) para evitar duplicatas.

Força Bruta - Análise de Complexidade

Complexidade de Tempo

- A complexidade de tempo da função depende principalmente do número de combinações possíveis dos índices dos elementos, que é determinado por product(range(k), repeat=n). O número total de combinações é k^n, onde n é o número de elementos em X e k é o número de subsets.

Força Bruta - Análise de Complexidade

- **Preenchimento de Subsets (Linha 14-18):**
  - Para cada combinação de índices, a função itera sobre os n elementos de X e os distribui nos k subsets.
  - A complexidade de tempo desta parte é O(n) para cada combinação.
- **Verificação de Condições (Linha 20-23):**
  - Para cada combinação, a função verifica se as somas dos subsets são iguais e se não há elementos compartilhados entre os subsets.
  - A verificação de condições leva O(k) para somas e O(n) para a verificação de elementos compartilhados.

Força Bruta - Análise de Complexidade

- **Adição ao Conjunto de Partições Únicas (Linha 24):**
    - Adicionar um conjunto ao conjunto de partições únicas leva tempo constante O(1) na média, assumindo que a operação de hash é O(1).

- **Impressão (Linha 27-30):**
    - A impressão de cada partição única leva O(k* n logn) para ordenar os elementos dentro de cada subset.

- A complexidade total de tempo é, portanto,

O(k^n*n*k*(n+logn))

Força Bruta - Análise de Complexidade

Complexidade de Espaço

- **Armazenamento de Subsets e Combinações:**
  - O armazenamento de todas as combinações de índices e os subsets correspondentes consome $O(k \cdot n)$ espaço.

- **Armazenamento de Partições Únicas:**
  - O conjunto de partições únicas armazena tuplas de conjuntos (subsets), onde cada conjunto contém $O(n)$ elementos.
  - Portanto, o espaço necessário é O(k*n^2).

Força Bruta - Análise de Complexidade

- **Outras Variáveis Temporárias:**
  - As outras variáveis temporárias, como sums e frozen_subsets, consomem espaço proporcional a O(k*n).

- Portanto, a complexidade total de espaço é

O(k* n^2)

Backtracking

```
from itertools import product

def partition_k(X, k):
    n = len(X)

    # Gera todas as combinações possíveis dos índices dos elementos
    index_combinations = product(range(k), repeat=n)

    unique_partitions = set()

    # Verifica se cada combinação atende aos critérios
    for indices in index_combinations:
        subsets = [[] for _ in range(k)]
```

Backtracking

```python
def is_valid_partition(partitions):
    all_elements = set()
    for subset in partitions:
        for num in subset:
            if num in all_elements:
                return False
            all_elements.add(num)
    return True

def k_partition_backtrack(X, k, target_sum, current_partition, result, index):
    if index == len(X):
        if all(sum(subset) == target_sum for subset in current_partition):
            sorted_partitions = [tuple(sorted(subset)) for subset in current_partition]
            result.add(tuple(sorted(sorted_partitions)))
        return

    for i in range(k):
        current_partition[i].append(X[index])
        k_partition_backtrack(X, k, target_sum, current_partition, result, index + 1)
        current_partition[i].remove(X[index])
```

Backtracking

```python
def k_partition(X, k):
    total_sum = sum(X)
    if total_sum % k != 0:
        return None  # Não é possível encontrar partições com soma igual

    target_sum = total_sum // k
    result = set()
    current_partition = [[] for _ in range(k)]

    k_partition_backtrack(X, k, target_sum, current_partition, result, 0)

    return list(result) if result else None

# Exemplo de uso
X = [2, 5, 4, 9, 1, 7, 6, 8]
k = 3
partitions = k_partition(X, k)

if partitions:
    for i, subset in enumerate(partitions):
        print(f"Subset {i + 1}: {subset}")
else:
    print(f"Não é possível encontrar {k} partições com soma igual.")
```

Algoritmo com Backtracking - Descrição

Função *is_valid_partition*

- Essa função verifica se uma dada partição é válida, ou seja, se nenhum elemento está sendo compartilhado entre as partições.

- *all_elements* é um conjunto usado para rastrear elementos já vistos. Se um elemento já está no conjunto, significa que ele está sendo compartilhado e a função retorna *False*.

- Se nenhum elemento é compartilhado, a função retorna *True*.

Algoritmo com Backtracking - Descrição

Função *k_partition_backtrack*

- Esta função é uma função auxiliar usada para realizar o backtracking e encontrar todas as partições válidas.
- A recursão é utilizada para tentar diferentes combinações de elementos nas partições.
- O algoritmo mantém um índice (index) para percorrer os elementos da lista X e tentar adicioná-los a diferentes partições.

Algoritmo com Backtracking - Descrição

Condição de Parada para a Recursão

- Quando o índice (*index*) atinge o comprimento da lista *X*, o algoritmo verifica se a partição atual atende às condições desejadas.
- Se a soma de cada subset for igual à *target_sum* (a soma alvo para cada partição), a partição é considerada válida.
- As partições válidas são formatadas e adicionadas ao conjunto *result*.

Algoritmo com Backtracking - Descrição

Função k_partition

- Calcula a soma total dos elementos da lista X e verifica se é possível dividir a lista em k partições com soma igual.
- Se não for possível, retorna None. Caso contrário, continua com a execução do algoritmo de backtracking.

Backtracking - Análise de Complexidade

Função is_valid_partition

- **Complexidade de Tempo:** A iteração sobre cada subconjunto (subset) é feita uma vez. Para cada número em cada subconjunto, verifica-se a presença no conjunto all_elements. Supondo que o número total de elementos nos subconjuntos seja n a complexidade de tempo é O(n).

- **Complexidade de Espaço:** Utiliza um conjunto (set) chamado all_elements. No pior caso, todos os elementos são armazenados neste conjunto. Portanto, a complexidade de espaço é O(n).

Backtracking - Análise de Complexidade

Função k_partition_backtrack

- **Complexidade de Tempo:** A função utiliza a técnica de backtracking para explorar todas as combinações possíveis. No pior caso, a árvore de recursão é totalmente expandida, e a complexidade de tempo é exponencial, O(k^n), onde n é o número de elementos em X.

- **Complexidade de Espaço:** A profundidade máxima da recursão é n, pois a função é chamada para cada elemento em X. Além disso, há uma cópia local de current_partition para cada chamada recursiva. Portanto, a complexidade de espaço é O(n).

Backtracking - Análise de Complexidade

Função k_partition

- **Complexidade de Tempo:** Chama a função k_partition_backtrack, que tem uma complexidade exponencial no pior caso. Assumindo que n seja o número total de elementos em X, a complexidade de tempo é $O(k^n)$.

- **Complexidade de Espaço:** Utiliza um conjunto (set) chamado result para armazenar partições únicas. A quantidade de espaço utilizado depende do número de partições encontradas. No pior caso, onde há muitas partições, a complexidade de espaço pode ser alta $O(n)$.

Backtracking - Análise de Complexidade

Análise Geral

- **A complexidade de tempo** total do programa é dominada pela função k_partition_backtrack chamada dentro da função k_partition. A função k_partition_backtrack utiliza a técnica de backtracking, explorando todas as combinações possíveis, resultando em uma complexidade exponencial no pior caso. Assumindo que n é o número total de elementos em X e k é o número desejado de partições, a complexidade de tempo total do programa é O(k^n).

- **A complexidade de espaço** total do programa é influenciada principalmente pelo conjunto result na função k_partition_backtrack e pelo conjunto all_elements na função is_valid_partition. Assumindo que n é o número total de elementos em X, a complexidade de espaço total do programa é O(n).

Divisão e Conquista

```
def minDiferenca(arr, l, r, soma1=0, soma2=0):
    # Caso base: se não houver mais elementos para incluir em
subconjuntos
    if l > r:
        return abs(soma1 - soma2)

    # Inclui o elemento atual no subconjunto 1 e recursivamente calcula
a diferença mínima
    inclui_soma1 = minDiferenca(arr, l + 1, r, soma1 + arr[l], soma2)

    # Inclui o elemento atual no subconjunto 2 e recursivamente calcula
a diferença mínima
    inclui_soma2 = minDiferenca(arr, l + 1, r, soma1, soma2 + arr[l])

    # Retorna a diferença mínima
    return min(inclui_soma1, inclui_soma2)

# Testando o código
arr = [3, 1, 4, 2, 2, 1, 5, 2]
n = len(arr)

print("A menor diferença é", minDiferenca(arr, 0, n - 1))
```

Divisão e Conquista - Descrição

função `minDiferenca`

- A função **`minDiferenca`** recebe um array, dois índices (`l` e `r` que representam o início e o fim do array), e duas somas (`soma1` e `soma2` que representam as somas dos elementos nos dois subconjuntos).

- O caso base da recursão é quando não há mais elementos para incluir nos subconjuntos (`l > r`). Nesse caso, a função retorna a diferença absoluta entre `soma1` e `soma2`. A função então chama a si mesma duas vezes: uma vez incluindo o elemento atual no subconjunto 1 (`soma1`) e outra vez incluindo o elemento atual no subconjunto 2 (`soma2`). E então, a função retorna a menor das duas diferenças calculadas.

Divisão e Conquista - Análise de Complexidade

- Para cada elemento, o algoritmo faz duas chamadas recursivas, uma para o caso em que o elemento é incluído no subconjunto 1 e outra para o caso em que o elemento é incluído no subconjunto 2. Portanto, o número total de subproblemas é 2^n.

- Dessa forma a complexidade do algoritmo é

O(2^n)

Programação Dinâmica

```
def minDiferenca(arr, n, somaTotal):
    # Inicializa a tabela dp
    dp = [[0 for i in range(somaTotal + 1)] for j in range(n + 1)]

    # Preenche a tabela dp de maneira ascendente
    for i in range(n + 1):
        dp[i][0] = True

    for j in range(1, somaTotal + 1):
        dp[0][j] = False

    for i in range(1, n + 1):
        for j in range(1, somaTotal + 1):
            dp[i][j] = dp[i - 1][j]

            if arr[i - 1] <= j:
                dp[i][j] |= dp[i - 1][j - arr[i - 1]]

    # Inicializa a diferença da soma das duas partes
    diff = float('inf')
```

Programação Dinâmica

```
    # Encontra a maior j tal que dp[n][j] é True
    # onde j varia de somaTotal/2 até 0
    for j in range(somaTotal // 2, -1, -1):
        if dp[n][j] == True:
            diff = somaTotal - 2 * j
            break

    return diff

# Testando o código
arr = [3, 1, 4, 2, 2, 1, 5, 2]
n = len(arr)

somaTotal = 0
for i in range(0, n):
    somaTotal += arr[i]

print("A menor diferença é", minDiferenca(arr, n, somaTotal))
```

Programação Dinâmica - Descrição

- A função `minDiferenca(arr, n, somaTotal)` recebe uma lista de números (`arr`), o número de elementos na lista (n) e a soma total de todos os números na lista (`somaTotal`). Ela retorna a menor diferença possível entre a soma dos números em dois subconjuntos da lista.

- O código usa programação dinâmica para resolver o problema. Ele constrói uma tabela dp onde `dp[i][j]` é `True` se um subconjunto de `{arr[0], arr[1], ..., arr[i-1]}` tiver soma de subconjunto igual a `j`, caso contrário é `False`.

- Finalmente, o código encontra a maior `j` tal que `dp[n][j]` é `True` onde `j` varia de `somaTotal//2` até `0`. A diferença entre a soma total e duas vezes essa `j` é a menor diferença possível, que é retornada pela função.

Programação Dinâmica - Análise de Complexidade

- O código usa uma abordagem de programação dinâmica para resolver o problema, construindo uma tabela dp de tamanho n * somaTotal. Cada célula da tabela dp é preenchida uma vez, resultando em uma complexidade de tempo de

O(n*somaTotal)