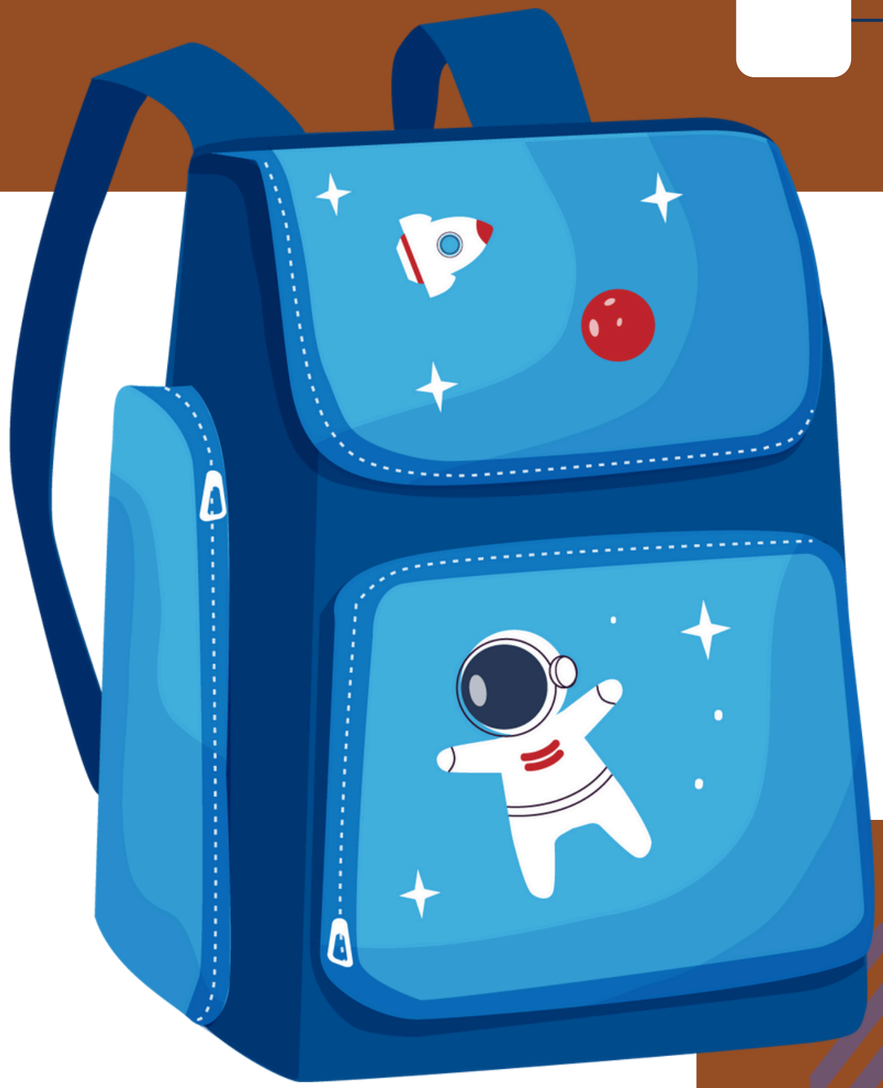
PROBLEMA DA MOCHILA 0/1

Docente: Herbert Oliveira Rocha

Discentes: Eduardo Izidorio e Shelly Leal

Problema da Mochila 0/1

- É um problema clássico de otimização combinatória
- Consiste em selecionar um subconjunto de itens, cada um com um **peso** e um **valor**, de modo que o valor total seja **maximizado** sem que o peso total ultrapasse a capacidade máxima.



Na versão 0/1, cada item pode ser escolhido inteiro ou não escolhido — não é permitido fracionar.

Dado um conjunto de *n* itens com pesos e valores, e uma capacidade, o problema é:

$$\text{Maximizar} \quad \sum_{i=1}^{n} v_i x_i$$

$$\text{sujeito a} \quad \sum_{i=1}^{n} p_i x_i \leq C, \quad x_i \in \{0, 1\}$$

p_i = Pesos

x_i = Valores

C = Capacidade

Artigo “The Price-Elastic Knapsack Problem”

O artigo de Fukasawa, Naoum-Sawaya e Oliveira, introduz uma variação do Problema da Mochila denominada Price-Elastic Knapsack Problem (**PEKP**). Diferentemente da formulação clássica, no PEKP o peso e/ou o valor de cada item não são fixos, mas dependem de um parâmetro denominado preço.

Literary review 1

Essa dependência cria relações não lineares na função objetivo e/ou nas restrições, tornando o problema mais complexo.

Artigo “The Price-Elastic Knapsack Problem”

Os autores formulam o PEKP como um problema de otimização não linear e investigam três casos específicos:

- Caso resolvível em tempo polinomial — uma configuração simplificada da dependência entre preço e parâmetros do item que permite solução eficiente.
- Caso de peso afim do preço — o peso varia linearmente com o preço. Esse caso resulta em um Quadratic Program (QP), de difícil resolução direta. Para contornar a dificuldade, o problema é decomposto em três Programações Inteiras Mistas (MIP) independentes, resolvidas separadamente.
- Caso de peso piecewise-linear — também resulta em QP, tratado pela mesma estratégia de decomposição.

Nos experimentos computacionais, a decomposição em MIPs apresentou desempenho superior em relação à resolução direta dos QPs, especialmente em instâncias de maior porte.

Complexidade

O Problema da Mochila 0/1 é **NP-Completo**:

- Não existe algoritmo conhecido que resolva todas as instâncias em tempo polinomial.
- A complexidade cresce exponencialmente com ***n***.
- Métodos exatos como **Backtracking e Programação Dinâmica** são viáveis para instâncias pequenas ou médias.

Algoritmos Implementados

Backtracking

Explora todas as combinações possíveis, usando poda quando o peso excede C.

- Complexidade: **O (2^n)** no pior caso.
- Vantagem: encontra sempre a solução ótima.

Programação Dinâmica

Usa tabela dp [i] [c] representando o melhor valor com os primeiros *i* itens e capacidade *c*.

- Complexidade **O(n · C)**.
- Vantagem: eficiente para capacidades moderadas.

Pseudocódigo - Backtracking

```
Algoritmo 1: Mochila_Backtracking
Entrada: peso[1..n], valor[1..n], C
Saída: MelhorValor, MelhorEscolha[1..n]

1  MelhorValor ← 0
2  MelhorPeso  ← 0
3  EscolhaAtual[1..n] ← 0
4  MelhorEscolha[1..n] ← 0

5  procedimento DFS(idx, pesoAtual, valorAtual):
6     se pesoAtual > C então
7        retornar                    // Poda por capacidade
8     fim-se
9     se idx > n então
10       se valorAtual > MelhorValor então
11          MelhorValor ← valorAtual
12          MelhorPeso  ← pesoAtual
13          MelhorEscolha ← cópia(EscolhaAtual)
14       fim-se
15       retornar
16    fim-se
17
```

Pseudocódigo - Backtracking

```
18 // Caso A: não pegar o item idx
19 EscolhaAtual[idx] ← 0
20 DFS(idx + 1, pesoAtual, valorAtual)
21
22 // Caso B: pegar o item idx
23 EscolhaAtual[idx] ← 1
24 DFS(idx + 1, pesoAtual + peso[idx], valorAtual + valor[idx])
25 // Programa principal
26 ler n, C e os vetores peso, valor
27 DFS(1, 0, 0)
28 retornar (MelhorValor, MelhorEscolha)
```

Pseudocódigo - Programação Dinâmica

Algoritmo 2: Mochila_DP_2D
Entrada: peso[1..n], valor[1..n], C
Saída: ValorÓtimo, Escolha[1..n]

```
1  criar matriz dp[0..n][0..C] inicializada com 0
2
3  para i ← 1 até n faça
4     para c ← 0 até C faça
5        naoPega ← dp[i-1][c]
6        se peso[i] ⩽ c então
7           pega ← dp[i-1][c - peso[i]] + valor[i]
8        senão
9           pega ← -∞   // ou simplesmente naoPega
10       fim-se
11       dp[i][c] ← max(naoPega, pega)
12    fim-para
13 fim-para
14
15 ValorÓtimo ← dp[n][C]
16 Escolha[1..n] ← 0
17 c ← C
```

Pseudocódigo - Programação Dinâmica

```
18 para i ← n até 1 passo -1 faça
19 se dp[i][c] ≠ dp[i-1][c] então // item i foi escolhido
20 Escolha[i] ← 1
21 c ← c - peso[i]
22 fim-se
23 fim-para
24
25 retornar (ValorÓtimo, Escolha)
```

Estudo de Caso – Missão Espacial

Na missão espacial considerada, o rover possui capacidade máxima de 100 kg. A tabela abaixo apresenta os itens disponíveis:

Item	Peso (kg)	Valor (pts)
Câmera alta resolução	**20**	**40**
Braço robótico	**50**	**100**
Analisador de solo	**30**	**60**
Detector de radiação	**10**	**30**
Fonte de energia extra	**40**	**70**

O objetivo é selecionar itens que maximizem o valor científico sem exceder 100 kg.

Algoritmo 1 – Mochila 0/1 com Backtracking (com poda de capacidade)

Entrada: Lista Itens[0..n−1] (cada item com peso e valor), número de itens n, capacidade máxima C.
Saída: MelhorValor (valor ótimo), MelhorEscolha[0..n−1] (vetor binário indicando itens escolhidos).

1. MelhorValor ← 0
2. MelhorEscolha[0..n−1] ← 0
3. EscolhaAtual[0..n−1] ← 0
4. Definir função DFS(indice, pesoAtual, valorAtual)
4.1. Se pesoAtual > C, retorne
4.2. Se indice == n então
a) Se valorAtual > MelhorValor então
MelhorValor ← valorAtual
MelhorEscolha ← copia(EscolhaAtual)
b) retorne
4.3. EscolhaAtual[indice] ← 0
Chamar DFS(indice+1, pesoAtual, valorAtual)
4.4. EscolhaAtual[indice] ← 1
Chamar DFS(indice+1, pesoAtual + Itens[indice].peso, valorAtual + Itens[indice].valor)
5. Chamar DFS(0, 0, 0)
6. Retornar (MelhorValor, MelhorEscolha)

Complexidade: Tempo O(2^n), Espaço O(n).

Algoritmo 2 – Mochila 0/1 com Programação Dinâmica (tabela 2D e reconstrução)

Entrada: Lista Itens[0..n−1] (cada item com peso e valor), número de itens n, capacidade máxima C.
Saída: ValorTotal, PesoTotal, Escolha[0..n−1] (itens escolhidos).

1. Criar tabela dp[0..n][0..C] inicializada com 0
2. Para i ← 1 até n faça
   2.1. Para c ← 0 até C faça
      a) naoPega ← dp[i−1][c]
      b) Se Itens[i−1].peso ⩽ c então
         pega ← dp[i−1][c − Itens[i−1].peso] + Itens[i−1].valor
      Senão
         pega ← naoPega
      c) dp[i][c] ← max(naoPega, pega)
3. ValorTotal ← dp[n][C]
4. Inicializar Escolha[0..n−1] ← 0, PesoTotal ← 0, capacidadeRestante ← C
5. Para i ← n até 1 (decrescente) faça
   5.1. Se dp[i][capacidadeRestante] ≠ dp[i−1][capacidadeRestante] então
      a) Escolha[i−1] ← 1
      b) PesoTotal ← PesoTotal + Itens[i−1].peso
      c) capacidadeRestante ← capacidadeRestante − Itens[i−1].peso
6. Retornar (ValorTotal, PesoTotal, Escolha)

Complexidade: Tempo O(n · C), Espaço O(n · C) — pode ser reduzido para O(C) com vetor 1D.

Qual é a melhor solução?

Em termos de resultado, os dois chegam ao mesmo valor (porque ambos são métodos exatos para a Mochila 0/1).

Em termos de desempenho:

- Backtracking tem complexidade temporal exponencial $O(2^{n)}$ e pode ficar muito lento para muitos itens.
- Programação Dinâmica tem complexidade temporal polinomial $O(n \cdot C)$ e é mais eficiente quando a capacidade não é muito grande, especialmente se quisermos rodar com muitos itens.

Existem várias soluções ótimas com 200 pontos e 100 kg, por exemplo:
(Braço 50,100) + (Analisador 30,60) + (Câmera 20,40) = 100 kg, 200 pts
(Braço 50,100) + (Fonte 40,70) + (Detector 10,30) = 100 kg, 200 pts
(Câmera 20,40) + (Analisador 30,60) + (Detector 10,30) + (Fonte 40,70) = 100 kg, 200 pts

O valor ótimo é o mesmo em ambos os métodos porque ambos são exatos; a combinação específica pode mudar conforme a ordem dos itens/empates na reconstrução.
Para instâncias maiores, PD tende a ser mais eficiente quando CCC é moderado; Backtracking é simples, mas exponencial.

Obrigado!

Link do repositório Github:
https://github.com/EhoKira/FinalProject_DCC606_Problema_da_Mochila_8_RR_2025.git