Problema da Mochila **(Knapsack problem)**

- **Instância:** Pesos $p[1 \ldots n]$ e valores $v[1 \ldots n]$ e Capacidade C da mochila.
- **Objetivo:** Achar vetor $x[1 \ldots n]$ de 0's e 1's tal que $\mathbf{x} \cdot \mathbf{p} = \sum_{k=1}^{n} x_k p_k \leq \mathbf{C}$ e $\mathbf{x} \cdot \mathbf{v} = \sum_{k=1}^{n} x_k v_k$ seja máximo
- **Intuição:** $x[k] = 1$ (item k selecionado) e $x[k] = 0$ (cc).
- **Exemplo:** $C = 5$, $p = [4, 2, 1, 3]$ e $v = [50, 40, 30, 45]$. Ótimo: $x = [0, 1, 0, 1]$, pois $2 + 3 \leq 5$ e $40 + 45 = 85$ é o valor máximo.

1. Propriedade da Subestrutura Ótima

- Imagine uma solução ótima, com um item k. Considere a instância sem o item k e com capacidade $C' = C - p_k$ na mochila.
- Então a solução ótima anterior sem o item k será uma solução ótima dessa nova instância. Ou seja, removendo o item k da mochila ótima, o que estiver na mochila será uma solução ótima caso a mochila fosse menor, com capacidade C'.
- Se houvesse uma solução melhor para mochila com capacidade C', obteríamos uma solução melhor para a mochila com capacidade C, simplesmente incluindo de volta o item k. Absurdo.

2. Equação de Recorrência - Algoritmo recursivo simples

- Seja $V[k, C']$ o valor máximo considerando apenas os itens de 1 a k a mochila tendo capacidade C'. **Objetivo:** Calcular $V[n, C]$.
- $V[0, C'] = 0$; $\qquad V[k, 0] = 0$
- Verificar se é melhor com o item k ou não.
- Se $p_k > C'$: $V[k, C'] = V[k-1, C']$. **Caso contrário:**

$$V[k, C'] \;=\; \max\Big\{V[k-1, C'],\;\; v_k \,+\, V[k-1, C'-p_k]\Big\}.$$

Mochila-REC(p, v, C', k)

1 **se** $(k = 0)$ **então retorne** 0
2 **se** $(C' = 0)$ **então retorne** 0
3 $A \longleftarrow$ *Mochila-REC*$(p, v, C', k-1)$
4 **se** $(p_k > C')$ **então retorne** A
5 $B \longleftarrow v_k +$ *Mochila-REC*$(p, v, C'-p_k, k-1)$
6 **se** $(A \geq B)$ **então retorne** A
7 **retorne** B

2. Equação de Recorrência - Algoritmo recursivo simples

Sobreposição de Subproblemas

- **Chamada inicial**: $Mochila - REC(p, v, C, n)$
- **Superposição**: Se todos os pesos são iguais a 1, a instância $(C-1, n-2)$ é chamada por $(C, n-1)$ e $(C-1, n-1)$, que são chamadas por (C, n).
- **Tempo**: Exponencial $\Omega(2^{\min\{n,C\}}) \Rightarrow \Omega(2^n)$ para $C = n$

Mochila-REC(p, v, C', k)

1 **se** $(k = 0)$ **então retorne** 0
2 **se** $(C' = 0)$ **então retorne** 0
3 $A \longleftarrow$ *Mochila-REC*$(p, v, C', k-1)$
4 **se** $(p_k > C')$ **então retorne** A
5 $B \longleftarrow v_k +$ *Mochila-REC*$(p, v, C' - p_k, k-1)$
6 **se** $(A \geq B)$ **então retorne** A
7 **retorne** B

2b. Memoização (Alg. recursivo + memória) - top Down

Mochila-memo(p, v, C, n, V)

```
1  para C' ← 0 até C:    V[0, C'] ← 0
2  para k ← 1 até n:     V[k, 0] ← 0
3  para k ← 1 até n:
4        para C' ← 1 até C:
5              V[k, C'] ← −1
6  retorne Mochila-REC-memo(p, v, C, n, V)
```

Mochila-REC-memo(p, v, C', k, V)

```
1  se (V[k, C'] ≥ 0) então retorne V[k, C']
2  A ← Mochila-REC-memo(p, v, C', k − 1)
3  se (p_k > C') então  V[k, C'] ← A
4  senão
5        B ← v_k + Mochila-REC-memo(p, v, C' − p_k, k − 1, V)
6        se (A < B) então V[k, C'] ← B senão V[k, C'] ← A
7  retorne V[k, C']
```

3. Programação Dinâmica - Valor ótimo (bottom-up)

Mochila-PD(p, v, C, n)

0 Criar matriz $V[0 \ldots n,\ 0 \ldots C]$
1 **para** $C' \leftarrow 0$ **até** C: $V[0, C'] \leftarrow 0$
2 **para** $k \leftarrow 1$ **até** n: $V[k, 0] \leftarrow 0$
3 **para** $k \leftarrow 1$ **até** n:
4 **para** $C' \leftarrow 1$ **até** C:
5 $V[k, C'] \longleftarrow V[k-1, C']$
6 **se** $(p_k \leq C')$ e $(V[k, C'] < v_k + V[k-1, C'-p_k])$ **então**
7 $V[k, C'] \longleftarrow v_k + V[k-1, C'-p_k]$
8 **retorne** $V[n, C]$

Tempo $O(n \cdot C)$

Podemos assumir que $p_k \leq C$ para todo item k. Isso não foi necessário nesse algoritmo, mas será no próximo.

3. Programação Dinâmica - Valor ótimo - variante

Ao invés de buscar conjuntos com valor máximo que cabem na mochila, buscamos conjuntos com peso mínimo e que atingem um valor dado. O maior valor possível é $\leq n \cdot v^*$, onde v^* é o valor do item mais valioso.

Mochila-PD2(p, v, C, n)

0 Criar matriz $P[0 \ldots n,\ 0 \ldots n \cdot v^*]$
1 **para** $V' \leftarrow 1$ **até** $n \cdot v^*$: $P[0, V'] \leftarrow \infty$
2 **para** $k \leftarrow 0$ **até** n **faça**: $P[k, 0] \leftarrow 0$
3 **para** $k \leftarrow 1$ **até** n:
4 **para** $V' \leftarrow 1$ **até** $n \cdot v^*$:
5 $P[k, V'] \longleftarrow P[k-1, V']$
6 **se** $(v_k \leq V')$ e $(P[k, V'] > p_k + P[k-1, V' - v_k])$ **então**
7 $P[k, V'] \longleftarrow p_k + P[k-1, V' - v_k]$
8 **para** $V' \leftarrow n \cdot v^*$ **até** 1 (dec):
9 **se** $(P[n, V'] \leq C)$ **então retorne** V'

Tempo $O(n^2 \cdot v^*)$

3. Programação Dinâmica - Valor ótimo - variante - Exemplo

P	0	1	2	3	4	5	6	7	8	**9**	10
0	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	0	4	4	∞	∞	∞	∞	∞	∞	∞	∞
2	0	2	4	6	∞	∞	∞	∞	∞	∞	∞
3	0	1	1	1	3	5	7	∞	∞	∞	∞
4	0	1	1	1	2	3	3	3	5	7	9
5	0	1	1	1	2	3	3	3	5	**7**	9
k											

Valores V'
(maiores ou iguais)

1 2 3 4 5	itens
4 2 1 2 2	pesos
2 1 3 4 1	valores

P	0	1	2	3	4	5	6	7	8	**9**	10
0	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	0	∞	4	∞	∞	∞	∞	∞	∞	∞	∞
2	0	2	4	6	∞	∞	∞	∞	∞	∞	∞
3	0	2	4	1	3	5	7	∞	∞	∞	∞
4	0	2	4	1	2	4	6	3	5	7	9
5	0	2	4	1	2	4	6	3	5	**7**	9

Valores V'
(exatamante iguais)

Capacidade
C=7 ou 8

3. Programação Dinâmica - Solução Ótima

Ao invés de buscar conjuntos com valor máximo que cabem na mochila, buscamos conjuntos com peso mínimo e que atingem um valor dado. O maior valor possível é $\leq n \cdot v^*$, onde v^* é o valor do item mais valioso.

Mochila-PD3(p, v, C, n)

0 Criar matriz $P[0 \ldots n,\ 0 \ldots n \cdot v^*]$ e vetor $x[1 \ldots n]$
1 **executar** *Mochila-PD2*(p, v, C, n) com a matriz P alocada acima
2 Seja V' o valor ótimo retornado acima
3 **para** $k \leftarrow n$ **decrescendo até** 1:
4 **se** $P[k, V'] = P[k-1, V']$ **então** $x[k] \leftarrow 0$
5 **senão** $x[k] \leftarrow 1$
6 $V' \leftarrow V' - p[k]$
7 **retorne** x

O vetor x indica a presença ou não de cada item na solução ótima.

Tempo $O(n^2 \cdot v^*)$: linha 1

4. Algoritmo aproximativo

Mochila-Aprox$(p, v, C, n, \varepsilon)$

1 $v^* \leftarrow \max\{v_k : k = 1, \ldots, n\}$; $\lambda \leftarrow \varepsilon \cdot v^*/n$
2 **para** $k \leftarrow 1$ **até** n **faça**:
3 $\quad v'_k \leftarrow \lfloor v_k/\lambda \rfloor$
4 **retorne** os itens da solução ótima de *Mochila-PD3*(p, v', C, n)

Tempo $O(n^2 \cdot (v^*/\lambda)) = O(n^3/\varepsilon)$

Como só mudam os valores dos itens, uma solução ótima com valores v' **é uma solução viável** do problema original.

5. Algoritmo $(1-\varepsilon)$-aproximativo

Teorema 2.3: Mochila-Aprox é $(1-\varepsilon)$-aproximativo

Prova: Seja S' a solução ótima retornada pelo algoritmo (com valores v') e seja S uma solução ótima do problema original.

$$v(S') = \sum_{k \in S'} v_k \geq \lambda \sum_{k \in S'} v'_k \geq \lambda \sum_{k \in S} v'_k \geq$$

$$v(S') \geq \lambda \sum_{k \in S} \left(\frac{v_k}{\lambda} - 1 \right) \geq \sum_{k \in S} v_k - \lambda \cdot |S| \geq v(S) - \lambda \cdot n \geq$$

$$v(S') \geq opt - \varepsilon \cdot v^* \geq opt - \varepsilon \cdot opt \geq (1-\varepsilon) \cdot opt$$

6. Conclusão (Mochila)

Mochila-Aprox é um FPTAS

FPTAS: Full Polynomial Time Approximation Scheme.

Esquema de Aproximação Completamente Polinomial

Para cada racional $\varepsilon > 0$, temos um algoritmo de tempo $O(n^3/\varepsilon)$ com fator de aproximação $1 - \varepsilon$.

É um meta-algoritmo, um esquema de aproximação. Um algoritmo para cada ε.

Compromisso entre tempo e aproximação.

Fully polynomial, pois $O(n^3/\varepsilon)$ é polinomial também em $1/\varepsilon$, ao contrário de $O(n^{1/\varepsilon})$, que seria "apenas" um PTAS.

7. Conclusão (até agora)

Escalonamento: Tem algoritmo 2-aproximativo.

Cobertura por Conjuntos: Algoritmo ($\ln n$)-aproximativo. Parece o melhor possível (parece não ter aproximativo para fator constante).

Mochila: FPTAS. É o melhor possível.

- Problemas NP-Difíceis com FPTAS (Mochila)
- Problemas NP-Difíceis com PTAS, que provavelmente não tem FPTAS (falta)
- Problemas NP-Difíceis com α-aproximação (para α const), que provavelmente não tem PTAS (TSP Métrico - falta mostrar)
- Problemas NP-Difíceis com $\alpha(n)$-aproximação (para $\alpha(n)$ **não** const), que provavelmente não tem para α const (Set Cover)
- Problemas NP-Difíceis que provavelmente **não** tem algoritmo $\alpha(n)$-aproximativo nenhum (Caixeiro viajante)