Programação Dinâmica

- Tipicamente o paradigma de programação dinâmica aplica-se a problemas de **otimização**.
- Podemos utilizar programação dinâmica em problemas onde há:
  - **Subestrutura Ótima:** As soluções ótimas do problema incluem soluções ótimas de subproblemas.
  - **Sobreposição de Subproblemas:** O cálculo da solução através de recursão implica no recálculo de subproblemas.

Programação Dinâmica: Conceitos Básicos (Cont.)

- A técnica de programação dinâmica visa evitar o recálculo desnecessário das soluções dos subproblemas.
- Para isso, soluções de subproblemas são armazenadas em tabelas.
- Logo, para que o algoritmo de programação dinâmica seja eficiente, é preciso que o número total de subproblemas que devem ser resolvidos seja pequeno (polinomial no tamanho da entrada).

Memorização x Programação Dinâmica

- Existem duas técnicas para evitar o recálculo de subproblemas:
  - **Memorização:** Consiste em manter a estrutura recursiva do algoritmo, registrando em uma tabela o valor ótimo para subproblemas já computados e verificando, antes de cada chamada recursiva, se o subproblema a ser resolvido já foi computado.
  - **Programação Dinâmica:** Consiste em preencher uma tabela que registra o valor ótimo para cada subproblema de forma apropriada, isto é, a computação do valor ótimo de cada subproblema depende somente de subproblemas já previamente computados. Elimina completamente a recursão.

O Problema Binário da Mochila

O Problema da Mochila

Dada uma mochila de capacidade W (inteiro) e um conjunto de n itens com tamanho w_i (inteiro) e valor c_i associado a cada item i, queremos determinar quais itens devem ser colocados na mochila de modo a **maximizar** o valor total transportado, respeitando sua capacidade.

- Podemos fazer as seguintes suposições:
  - $\sum_{i=1}^{n} w_i > W$;
  - $0 < w_i \leq W$, para todo $i = 1, \ldots, n$.

O Problema Binário da Mochila

- Podemos resolver o problema da mochila com Programação Linear Inteira:
  - Criamos uma variável x_i para cada item: $x_i = 1$ se o item i estiver na solução ótima e $x_i = 0$ caso contrário.
  - A modelagem do problema é simples:

$$\max \sum_{i=1}^{n} c_i x_i \tag{1}$$

$$\sum_{i=1}^{n} w_i x_i \leq W \tag{2}$$

- (1) é a função objetivo e (2) o conjunto de restrições.

O Problema Binário da Mochila

- Como podemos projetar um algoritmo para resolver o problema ?
- Existem 2^n possíveis subconjuntos de itens: um algoritmo de força bruta é impraticável !
- É um problema de otimização. Será que tem subestrutura ótima ?
- Se o item n estiver na solução ótima, o valor desta solução será c_n mais o valor da melhor solução do problema da mochila com capacidade $W - w_n$ considerando-se só os $n - 1$ primeiros itens.
- Se o item n não estiver na solução ótima, o valor ótimo será dado pelo valor da melhor solução do problema da mochila com capacidade W considerando-se só os $n - 1$ primeiros itens.

- Seja $z[k, d]$ o valor ótimo do problema da mochila considerando-se uma capacidade d para a mochila que contém um subconjunto dos k primeiros itens da instância original.

O Problema Binário da Mochila

- Seja $z[k, d]$ o valor ótimo do problema da mochila considerando-se uma capacidade d para a mochila que contém um subconjunto dos k primeiros itens da instância original.
- A fórmula de recorrência para computar $z[k, d]$ para todo valor de d e k é:

- Seja $z[k, d]$ o valor ótimo do problema da mochila considerando-se uma capacidade d para a mochila que contém um subconjunto dos k primeiros itens da instância original.
- A fórmula de recorrência para computar $z[k, d]$ para todo valor de d e k é:

$$z[0, d] = 0$$

$$z[k, 0] = 0$$

$$z[k, d] = \begin{cases} z[k-1, d], & \text{se } w_k > d \\ \max\{z[k-1, d], z[k-1, d-w_k] + c_k\}, & \text{se } w_k \leq d \end{cases}$$

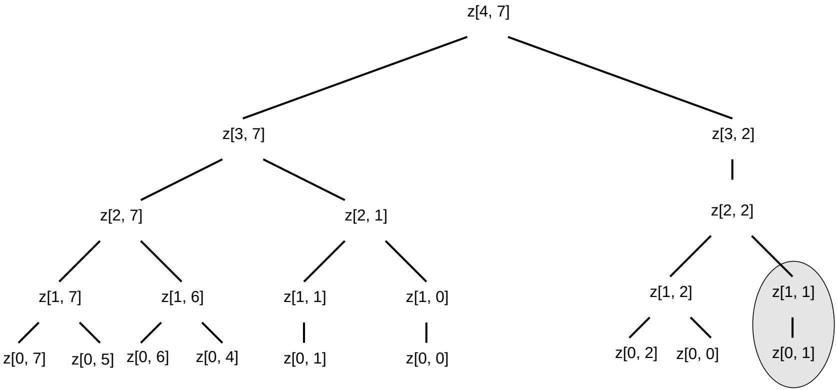
- A complexidade do algoritmo recursivo para este problema no pior caso é dada pela recorrência:

$$T(k,d) = \begin{cases} 1, & k = 0 \text{ ou } d = 0 \\ T(k-1,d) + T(k-1,d-w_k) + 1 & k > 0 \text{ e } d > 0. \end{cases}$$

- Portanto, no pior caso, o algoritmo recursivo tem complexidade $\Omega(2^n)$. É impraticável !
- Mas há sobreposição de subproblemas: o recálculo de subproblemas pode ser evitado !

Mochila - Sobreposição de Subproblemas

- Considere vetor de tamanhos $w = \{2, 1, 6, 5\}$ e capacidade da mochila $W = 7$. A árvore de recursão seria:

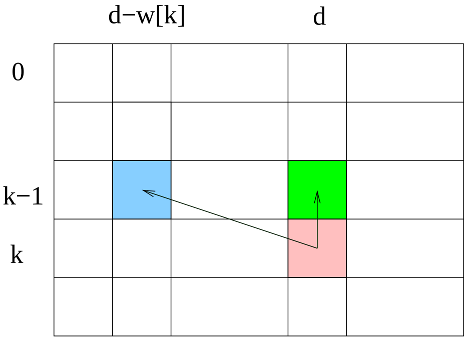


- O subproblema $z[1, 1]$ é computado duas vezes.

Mochila - Programação Dinâmica

- O número total máximo de subproblemas a serem computados é nW.
- Isso porque tanto o tamanho dos itens quanto a capacidade da mochila são inteiros !
- Podemos então usar programação dinâmica para evitar o recálculo de subproblemas.
- Como o cálculo de $z[k, d]$ depende de $z[k-1, d]$ e $z[k-1, d-w_k]$, preenchemos a tabela linha a linha.

Mochila



$$\boxed{z[k,d]} = \max \{ \boxed{z[k-1,d]} \;,\; \boxed{z[k-1,d-w[k]] + c[k]} \}$$

O Problema Binário da Mochila - Algoritmo

MochilaMaximo(c, w, W, n)

▷ **Entrada:** Vetores c e w com valor e tamanho de cada item, capacidade W da mochila e número de itens n.

▷ **Saída:** O valor máximo do total de itens colocados na mochila.

para $d := 0$ **até** W **faça** $z[0, d] := 0$
para $k := 1$ **até** n **faça** $z[k, 0] := 0$
para $k := 1$ **até** n **faça**
    para $d := 1$ **até** W **faça**
        $z[k, d] := z[k-1, d]$
        se $w_k \leq d$ **e** $c_k + z[k-1, d-w_k] > z[k, d]$ **então**
            $z[k, d] := c_k + z[k-1, d-w_k]$
retorne($z[n, W]$)

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0							
2	0							
3	0							
4	0							

Mochila - Exemplo

- Exemplo: $c = \{10, 7, 25, 24\}$, $w = \{2, 1, 6, 5\}$ e $W = 7$.

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

Mochila - Complexidade

- Claramente, a complexidade do algoritmo de programação dinâmica para o problema da mochila é $O(nW)$.
- É um algoritmo pseudo-polinomial: sua complexidade depende do valor de W, parte da entrada do problema.
- O algoritmo não dá o subconjunto de valor total máximo, apenas o valor máximo.
- É fácil recuperar o subconjunto a partir da tabela z preenchida.

Mochila - Recuperação da Solução

MochilaSolucao(z, n, W)

$\triangleright$ **Entrada:** Tabela z preenchida, capacidade W da mochila e número de itens n.

$\triangleright$ **Saída:** O vetor x que indica os itens colocados na mochila.

para $i := 1$ **até** n **faça** $x[i] := 0$
$MochilaSolucaoAux(x, z, n, W)$
retorne(x)

MochilaSolucaoAux(x, z, k, d)

se $k \neq 0$ **então**
  se $z[k, d] = z[k-1, d]$ **então**
    $x[k] := 0$; $MochilaSolucaoAux(x, z, k-1, d)$
  se não
    $x[k] := 1$; $MochilaSolucaoAux(x, z, k-1, d-w_k)$

Mochila - Exemplo

k \ d	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	10	10	10	10	10	10
2	0	7	10	17	17	17	17	17
3	0	7	10	17	17	17	25	32
4	0	7	10	17	17	24	31	34

x[1] = x[4] = 1 , x[2] = x[3] = 0

Mochila - Complexidade

- O algoritmo de recuperação da solução tem complexidade $O(n)$.
- Portanto, a complexidade de tempo e de espaço do algoritmo de programação dinâmica para o problema da mochila é $O(nW)$.
- É possível economizar memória, registrando duas linhas: a que está sendo preenchida e a anterior. Mas isso inviabiliza a recuperação da solução.

Algoritmo exato de Programação Dinâmica

- Componentes básicas de um algoritmo de Programação Dinâmica:
  - Uma solução ótima contém soluções ótimas de subproblemas semelhantes ao problema original;
  - O valor ótimo está relacionado aos valores ótimos destes subproblemas por meio de uma fórmula de recorrência;
  - Os valores subótimos são tabelados de modo a impedir recálculos;
- Valores tabelados para BKP
  - $A[i, d]$: peso do subconjunto *mais leve* dos i primeiros itens com valor exatamente d;
  - $A[i, d] = \infty$ se não existir tal conjunto;
  - **idéia**: dadas duas soluções de mesmo custo, dá-se preferência àquela que tem menos peso;

Algoritmo exato de Programação Dinâmica

- Fórmula de recorrência:

$$A[i,d] = \begin{cases} \min\{A[i-1,d], A[i-1,d-c_i] + w_i\}, & \text{se } c_i < d, \\ A[i-1,d], & \text{caso contrário.} \end{cases}$$

- Dimensão da matriz A: $(n+1) \times (nC+1)$.
- Inicialização de A:
  - Definição: $C = \max_{i \in N}\{c_i\}$;
  - *Primeira coluna*: $A[i,0] = 0$, para todo $i \in N \cup \{0\}$;
  - *Primeira linha*: $A[0,d] = \infty$, para todo $d \in \{1,\ldots,nC\}$;
- Valor ótimo: $C^* = \max\{d : A[n,d] \leq W\}$

  (maior índice de uma coluna cujo valor da célula na última linha não excede W)

Algoritmo exato de Programação Dinâmica

- Preenchimento da matriz A:

	0	1		$d-c_i$		d		nC
0	0	∞	$\cdots$	∞	$\cdots$	∞	$\cdots$	∞
	$\cdots$							
$i-1$	0							
i	0							
	$\cdots$							
n	0							

$$A[i,d] = \begin{cases} \min\{A[i-1,d], A[i-1,d-c_i] + w_i\}, c_i < d, \\ A[i-1,d], \text{ caso contrário.} \end{cases}$$

Algoritmo exato de Programação Dinâmica

- Preenchimento da matriz A:

	0	1		$d - c_i$		d		nC
0	0	∞	$\cdots$	∞	$\cdots$	∞	$\cdots$	∞
	$\cdots$							
$i-1$	0							
i	0							
	$\cdots$							
n	0							

Complexidade: $O(n^2 C)$ (*pseudopolinomial !*)