

# Programação Dinâmica

## Parte II

---

Laboratório de Programação Competitiva I

Pedro Henrique Paiola

Rene Pegoraro

Wilson M Yonezawa

Arisa Yoshida

Nicolas Barbosa Gomes

Luis Henrique Morelli

# Revisando

- Algoritmos com estrutura recursiva, divisão e conquista
- Ideia: armazenar a solução de subproblemas para a resolução de problemas futuros
- Ideia é simples, o desafio é aplicar isso em diferentes problemas.
- Estratégias e dicas

# Estratégia geral

1. Definir os subproblemas
2. Escrever a recorrência que relaciona os subproblemas
3. Reconhecer e solucionar os casos base

# Estratégia geral

- Ou, de forma análoga, podemos considerar os seguintes passos para resolver um problema de PD:
  1. Identificar se é um problema de PD
  2. Definir os estados do problema
  3. Definir a relação entre os estados
  4. Implementar a solução usando tabulation ou memoization

# Identificando o problema

- Na aula passada trabalhamos este passo com mais detalhes.
- Para podermos aplicar PD, o problema deve ter uma estrutura recursiva e apresentar as seguintes propriedades:
  - Subestrutura ótima
  - Sobreposição de subproblemas
- Problemas típicos:
  - Encontrar a solução ótima de um problema, que maximize ou minimize um determinado valor.
  - Contar o número de soluções possíveis.

# Definir os estados do problema

- Problemas de PD podem ser caracterizados por estados e transições
  - Corte do bastão
    - Estado: bastões resultantes dos cortes
    - Transição: novo corte
  - Troco
    - Estado: moedas utilizadas até o momento e valor que falta a ser trocado
    - Transição: adição de uma nova moeda
- A representação dos estados do problema devem ser feitos com um certo cuidado, pois isso vai afetar diretamente a próxima etapa, de definir a relação entre os estados, as transições

# Definir os estados do problema

- **Estado:** um estado pode ser definido como um conjunto de parâmetros que identifica, unicamente, uma posição ou “situação” de um dado problema.
- Esse conjunto de parâmetros deve ser o menor possível, especialmente para reduzir o espaço necessário para armazenar as soluções dos subproblemas.
- Exemplos:
  - Fibonacci:  $i$  ( $i$ -ésimo termo)
  - Troco: valor a ser trocado
  - Bastões: tamanho do bastão ou pedaço a ser cortado

# Definir os estados do problema

- Se o estado que define uma certa instância (ou subinstância) do seu problema apresenta muitos parâmetros, sua solução deve ser ineficiente.
  - A solução pode também ser ineficiente em relação ao tempo, mas principalmente em relação ao espaço, demandando muita memória para armazenar as soluções de subproblemas.
- Definir o melhor estado possível para as instâncias do problema é um passo primordial da aplicação de PD.
- Atente-se a parâmetros redundantes ou inúteis.



# Definir a relação entre os estados

- Provavelmente a parte mais difícil na resolução de um problema de PD.
- Dependendo do problema, requer bastante intuição, observação e prática.
- Como ocorrem as transições entre estados? Quais são as decisões possíveis?
- Como definir a solução de um problema em função dos seus subproblemas?

# Definir a relação entre os estados

- Encare o problema como sendo um problema de decisão: a partir de um certo estado, há um conjunto de decisões (ações, transições) possíveis, e deve-se determinar qual decisão leva a solução ótima.
- Cada transição leva a um determinado estado. Supondo que se conhece a solução de todos os estados para quais podemos avançar, como posso combinar essas informações para obter a solução do estado atual?

# Definir a relação entre os estados

- Problema do troco:
  - Estado: valor a ser trocado
  - Decisões possíveis: moedas disponíveis
  - Para cada moeda adicionada, temos um novo valor a ser trocado. Como estamos adicionando apenas uma moeda, a melhor decisão é aquela que leva ao valor que precisa do menor número de moedas para ser trocado.

$$troco(x) = 1 + \min \begin{cases} troco(x - m_1) \\ \dots \\ troco(x - m_n) \end{cases}$$

*Handwritten notes:*

- An arrow points from the text "estado atual" (current state) to the variable  $x$  in the function  $troco(x)$ .
- An arrow points from the text "novo estado" (new state) to the expression  $x - m_1$  in the first case of the minimum function.

# Definir a relação entre os estados

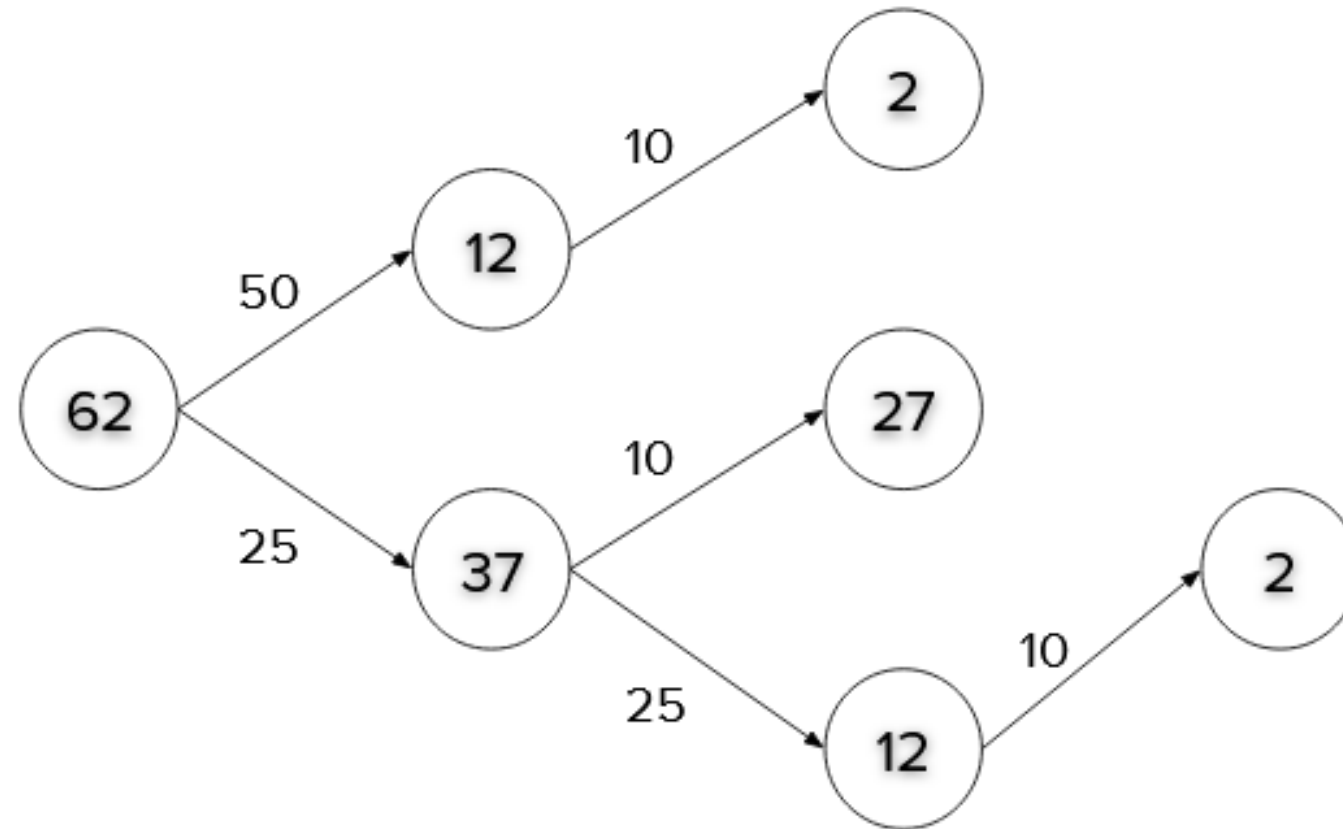
- Problema do corte do bastão:
  - Estado: tamanho do bastão
  - Decisões possíveis: cortes
  - Vamos considerar que em cada passo realizamos apenas um corte. A melhor decisão é o corte obtém o maior valor total, ou seja, considerando o preço do corte atual, com o maior valor possível de ser obtido com o bastão que sobrou.

$$rod(x) = \max \begin{cases} p_1 + rod(x - 1) \\ p_2 + rod(x - 2) \\ \dots \\ p_x + rod(0) \end{cases}$$

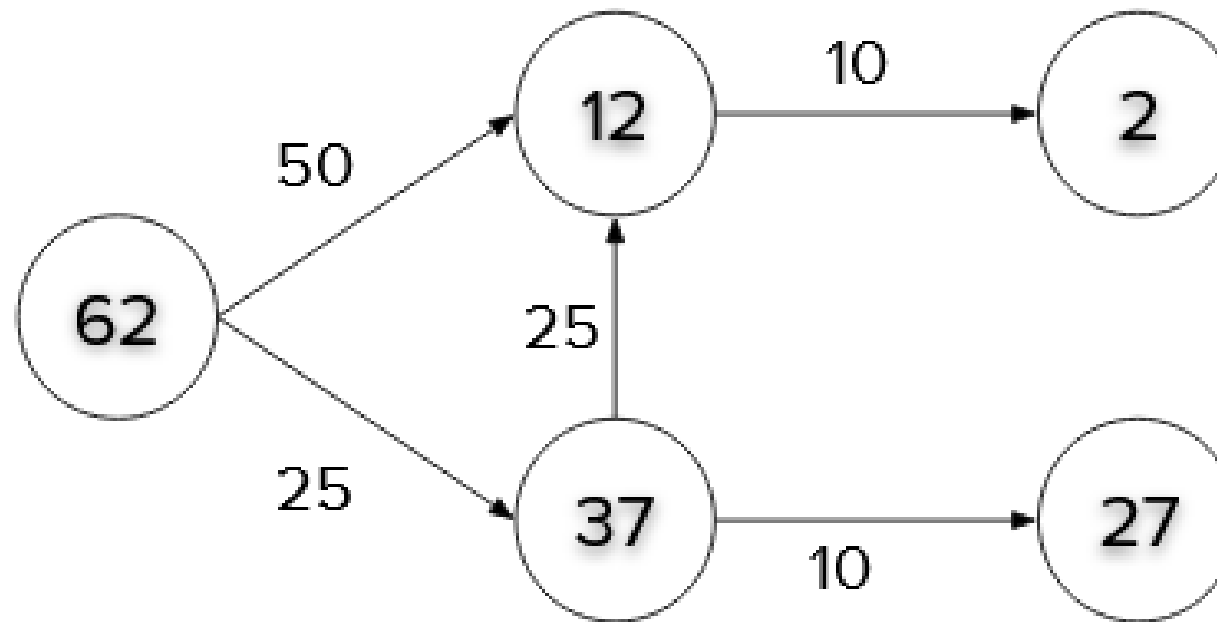
# Definir a relação entre os estados

- Em alguns casos, tentar elaborar um diagrama de como o problema se comporta, buscando estabelecer relações entre os estados, pode ajudar a obter *insights* de como solucionar o problema.

# Definir a relação entre os estados



# Definir a relação entre os estados



# Aplicar *tabulation* ou *memoization*

- Realizar a implementação da solução em si, considerando as abordagens apresentadas na aula anterior: Top Down e Bottom Up.
- As soluções dos subproblemas devem ser armazenadas de alguma forma, no caso mais geral temos:

**memo[estado] = solução do estado**



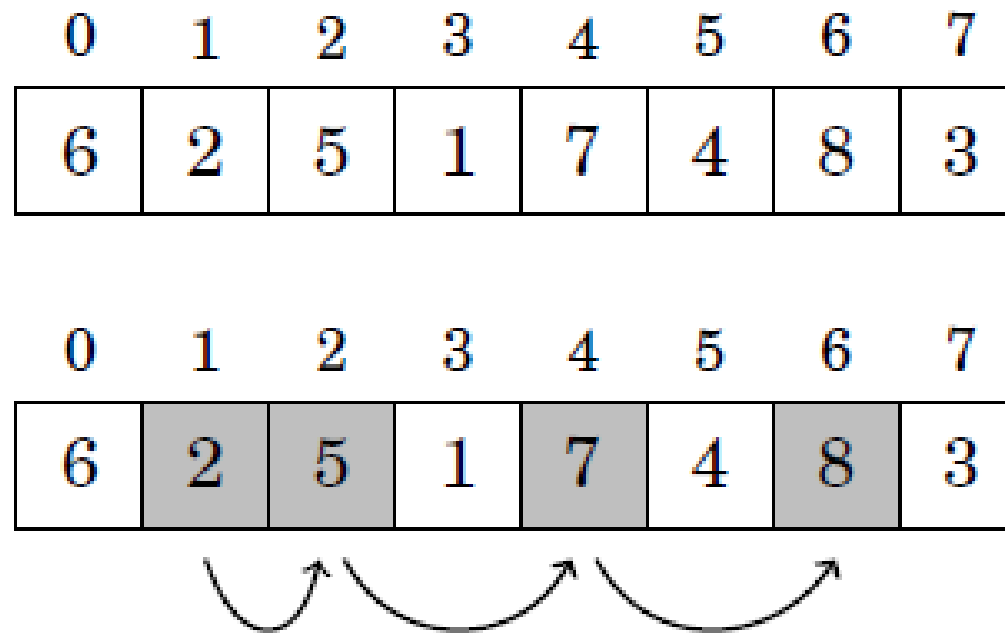
# Maior subsequência crescente (LIS)

- **Subsequência:** uma subsequência de uma sequência de elementos  $X$  é uma sequência  $X'$  com zero ou mais elementos de  $X$  removidos.
- É uma sequência de elementos de  $X$  não necessariamente contíguos.
- Exemplo:

$$\begin{aligned}
 X &= \{A, B, C, B, D, C, B\} \\
 X' &= \{A, C, D, C\}
 \end{aligned}$$

# Maior subseqüência crescente (LIS)

- Maior subseqüência crescente:** dado uma seqüência de números, determinar a maior subseqüência de valores crescentes.



# Maior subsequência crescente (LIS)

- Inicialmente, vamos tentar verificar se este é um problema de PD, definindo a relação de recorrência mais intuitiva possível, sem nos preocupar com a eficiência da solução ainda.
- Se pensarmos um pouco, não é tão difícil perceber que a subsequência máxima de um vetor  $v[0 \dots n - 1]$  pode ser definida a partir das subsequências máximas dos vetores  $v[0 \dots n - 2]$ ,  $v[0 \dots n - 3]$ , ... .

# Maior subsequência crescente (LIS)

0	1	2	3	4	5	6	7
3	4	1	2	6	4	2	5

i	Maior subsequência até i
0	3
1	3, 4
2	3, 4
3	3,4 ou 1,2
4	1, 2, 6
5	1, 2, 6 ou 1, 2, 4
6	1, 2, 6 ou 1, 2, 4
7	

# Maior subsequência crescente (LIS)

0	1	2	3	4	5	6	7
3	4	1	2	6	4	2	5

i	Maior subsequência até i
0	3, 5
1	3, 4, 5
2	3, 4, 5
3	3, 4 ou 1, 2, 5
4	1, 2, 6
5	1, 2, 6 ou 1, 2, 4, 5
6	1, 2, 6 ou 1, 2, 4, 5
7	1, 2, 4, 5



# Maior subsequência crescente (LIS)

0	1	2	3	4	5	6	7
3	4	1	2	6	4	2	5

i	Maior subsequência até i
0	3
1	3, 4
2	3, 4
3	3,4 ou 1,2
4	1, 2, 6
5	1, 2, 6 ou 1, 2, 4
6	1, 2, 6 ou 1, 2, 4
7	1, 2, 4, 5

# Maior subsequência crescente (LIS)

- Agora que já sabemos que podemos aplicar PD neste problema, vamos utilizar a estratégia apresentada anteriormente para modelá-los da melhor forma possível, visando uma implementação eficiente.

# Maior subseqüência crescente (LIS)

- Definição dos estados
  - No passo anterior, concluímos que podemos determinar a subseqüência máxima do vetor  $v[0 \dots n - 1]$  a partir das subseqüências máximas dos vetores  $v[0 \dots n - 2]$ ,  $v[0 \dots n - 3]$  ...
  - A partir disso, parece interessante definir o estado do nosso problema como o índice em que acaba nosso vetor.
  - Subseqüência máxima que **TERMINA** na posição  $i$ :  $lis(i)$
  - Subseqüência máxima do vetor inteiro:  $max(lis(i)), 0 \leq i < n$



# Maior subsequência crescente (LIS)

- Relação entre os estados
  - Agora temos que definir/encontrar uma relação de recorrência.
  - Problema base:  $lis(0)$ , nesse caso estamos considerando apenas o primeiro elemento do vetor, obviamente a maior subsequência crescente possível é 1 (considerando o único elemento possível)
    - $lis(0) = 1$

# Maior subsequência crescente (LIS)

- Relação entre os estados
  - E o passo da recursão?
  - Para  $lis(i)$  queremos encontrar a subsequência máxima que termina e contém a posição  $i$ .
  - Para isso, vamos considerar as posições  $j \mid j < i$

# Maior subsequência crescente (LIS)

- Relação entre os estados
  - Se  $a[j] > a[i]$ , não vamos considerar a  $lis(j)$ , pois o elemento  $a[i]$  não pode ser inserida nela.
  - Se  $a[j] \leq a[i]$ , então  $a[i]$  pode ser inserido na  $lis(j)$ , gerando uma subsequência de tamanho  $lis(j) + 1$ .

$$lis(i) = \begin{cases} 1 & \text{se } i = 0 \\ 1 + \max\{lis(j)\} & \text{para } \forall j \mid 0 \leq j < i \text{ e } a_j \leq a_i \end{cases}$$

# Maior subseqüência crescente (LIS)

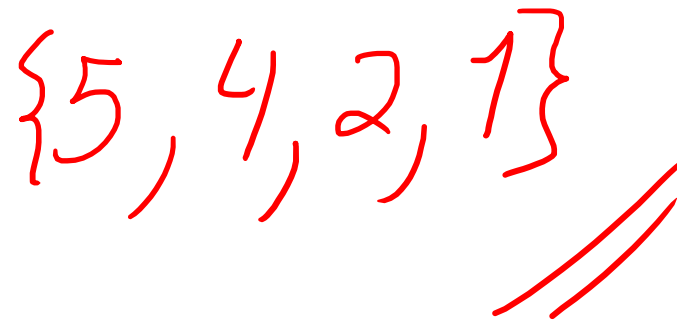
0	1	2	3	4	5	6	7
3	4	1	2	6	4	2	5

i	LIS(i)
0	1 {3}
1	2 {3,4}
2	1 {1}
3	2 {1,2}
4	3 {1, 2, 6}
5	3 {1, 2, 4}
6	2 {1,2}
7	4 {1, 2, 4, 5}

# Maior subseqüência crescente (LIS)

0	1	2	3	4	5	6	7
3	4	1	2	6	4	2	5

i	LIS(i)	
0	1 {3}	0
1	2 {3,4}	0
2	1 {1}	2
3	2 {1,2}	2
4	3 {1, 2, 6}	3
5	3 {1, 2, 4}	3
6	2 {1,2}	2
7	4 {1, 2, 4, 5}	5


  
 {5, 4, 2, 1}

# Maior subsequência crescente (LIS)

- Implementação (Top-down):

memo[] = {1, -1, -1, -1, ...}

```
int lis(int i){                                     //retorna a LIS que termina em a[i]
    if (memo[i] != -1)
        return memo[i];

    memo[i] = 1;
    for(int j = 0; j < i; j++)
        if (a[j] <= a[i])
            memo[i] = max(memo[i], lis(j) + 1);

    return memo[i];
}
```

# Maior subsequência crescente (LIS)

- Implementação (Bottom-up):

```

int lis(int n){
    int memo[n], lisMax = 0;

    for(int i = 0; i < n; i++){
        memo[i] = 1;
        for(int j = 0; j < i; j++){
            if (a[j] <= a[i])
                memo[i] = max(memo[i], memo[j] + 1);
        }
        lisMax = max(lisMax, memo[i]);
    }

    return lisMax;
}
  
```

# Maior subsequência crescente (LIS)

- Esta solução do problema tem complexidade  $O(n^2)$ .
- Por força bruta, teríamos complexidade exponencial (testando todas as possíveis subsequências)
- Existem outras possíveis soluções, utilizando Programação Dinâmica e Busca Binária ou alguma estrutura de dados que trabalhe com *range queries*. Estas soluções atingem complexidade  $O(n \cdot \log n)$ .
- Para mais detalhes:

[https://cp-algorithms.com/sequences/longest\\_increasing\\_subsequence.html](https://cp-algorithms.com/sequences/longest_increasing_subsequence.html)



# Maior subsequência comum (LCS)

- Problema: dadas as sequências  $X[0..m - 1]$  e  $Y[0..n - 1]$ , encontrar uma sequência  $Z$  tal que  $Z$  é subsequência de  $X$  e de  $Y$  e tem comprimento máximo.
- Exemplo:

$$\begin{aligned}
 X &= \{A, \textcolor{red}{B}, \textcolor{red}{C}, \textcolor{red}{B}, D, \textcolor{red}{A}, B\} \\
 Y &= \{\textcolor{red}{B}, D, \textcolor{red}{C}, A, \textcolor{red}{B}, \textcolor{red}{A}\} \\
 Z &= \text{LCS}(X, Y) = \{B, C, B, A\}
 \end{aligned}$$

# Maior subsequência comum (LCS)

- **Força bruta:** testar todas as subsequências de  $X$  para ver se ela também é uma subsequência de  $Y$ .
- Há  $2^m$  subsequências de  $X$  para serem verificadas
- Cada subsequência gasta tempo  $O(n)$  para ser verificada.
- Complexidade total:  $O(n \cdot 2^m)$

# Maior subsequência comum (LCS)

- Como dito anteriormente, uma subsequência de  $X$  é uma sequência  $X'$  com zero ou mais elementos de  $X$  removidos.
- Pensando nisso, nosso objetivo pode ser visto como minimizar o número de elementos removidos de duas sequências para que elas se tornem iguais (ou, de forma equivalente, maximizar o número de elementos inseridos).

# Maior subsequência comum (LCS)

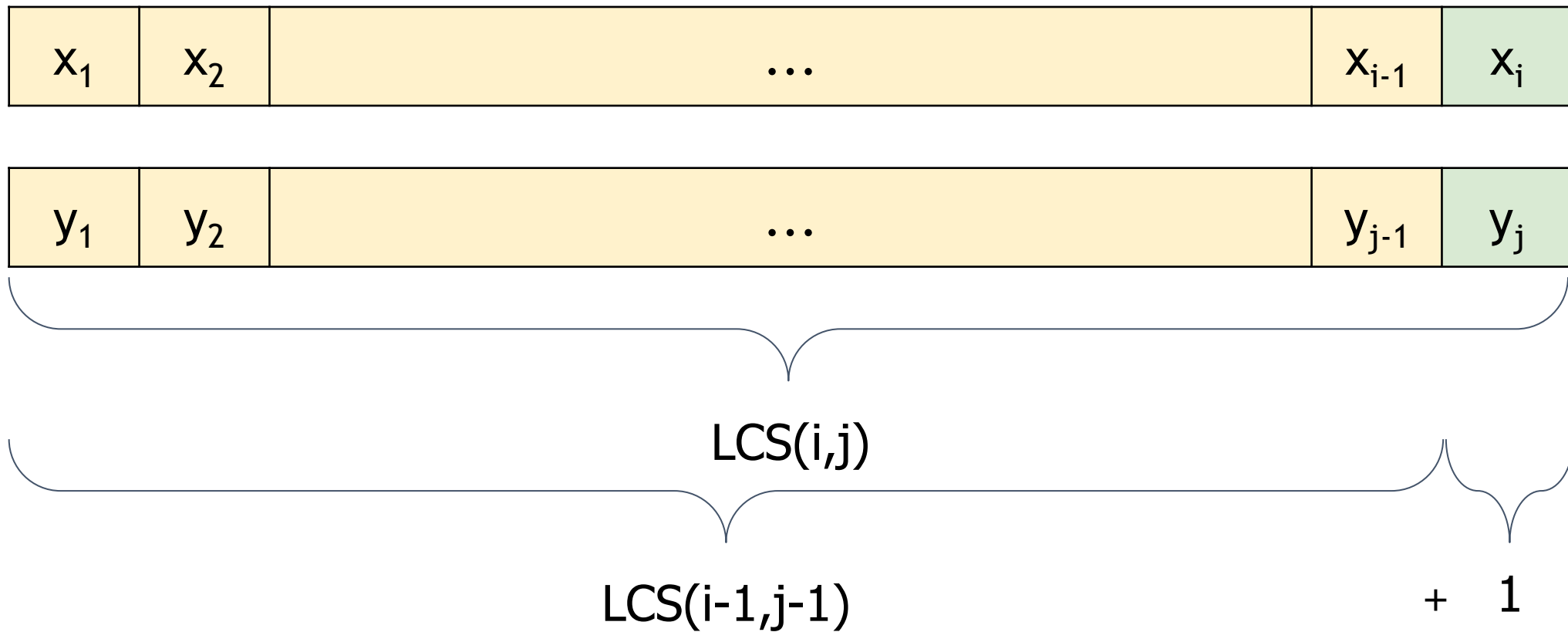
- Teorema:** Seja  $Z[1..k]$  uma LCS de  $X[1..m]$  e  $Y[1..n]$ 
  - Se  $x_m = y_n$  então  $z_k = y_n = x_m$  e  $Z[1..k-1]$  é uma LCS de  $X[1..m-1]$  e  $Y[1..n-1]$
  - Se  $x_m \neq y_n$  então  $z_k \neq x_m$ , sendo assim  $Z[1..k]$  é uma LCS de  $X[1..m-1]$  e  $Y[1..n]$
  - Se  $x_m \neq y_n$  então  $z_k \neq y_n$ , sendo assim  $Z[1..k]$  é uma LCS de  $X[1..m]$  e  $Y[1..n-1]$
- Esse teorema mostra que este problema atende a propriedade da Subestrutura Ótima.

# Maior subsequência comum (LCS)

$$LCS(i, j) = \begin{cases} 0 & \text{se } i = 0 \text{ ou } j = 0 \\ LCS(i - 1, j - 1) + 1 & \text{se } i, j > 0 \text{ e } x_i = y_j \\ \max(LCS(i, j - 1), LCS(i - 1, j)) & \text{se } i, j > 0 \text{ e } x_i \neq y_j \end{cases}$$

# Maior subseqüência comum (LCS)

- Se  $x_i = y_j$



# Maior subseqüência comum (LCS)

- Se  $x_i \neq y_j$

$x_1$	$x_2$	...			$x_{i-1}$	$x_i$
-------	-------	-----	--	--	-----------	-------

$y_1$	$y_2$	...			$y_{j-1}$	$y_j$
-------	-------	-----	--	--	-----------	-------

- Opção 1: retirar  $x_i \Rightarrow \text{LCS}(i-1, j)$

$x_1$	$x_2$	...			$x_{i-1}$
-------	-------	-----	--	--	-----------

$y_1$	$y_2$	...			$y_{j-1}$	$y_j$
-------	-------	-----	--	--	-----------	-------

# Maior subseqüência comum (LCS)

- Se  $x_i \neq y_j$

$x_1$	$x_2$	...			$x_{i-1}$	$x_i$
-------	-------	-----	--	--	-----------	-------

$y_1$	$y_2$	...			$y_{j-1}$	$y_j$
-------	-------	-----	--	--	-----------	-------

- Opção 2: retirar  $y_j \Rightarrow \text{LCS}(i, j-1)$

$x_1$	$x_2$	...			$x_{i-1}$	$x_i$
-------	-------	-----	--	--	-----------	-------

$y_1$	$y_2$	...			$y_{j-1}$
-------	-------	-----	--	--	-----------



# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B						
A						
C						
B						
A						
D						max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B						
A						
C						
B						
A						
D					+1	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B						
A						
C						
B						
A						
D						

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B						
A						
C						
B						
A			+1	max		
D					+1	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B						
A						
C						
B		+1				
A			+1	max		
D					+1	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B						
A						
C	max					
B		+1				
A			+1	max		
D					+1	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B						
A	+1					
C	max					
B		+1				
A			+1	max		
D					+1	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B						
A	1					
C	max					
B		+1				
A			+1	max		
D					+1	max



# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B						
A	1					
C	1					
B		+1				
A			+1	max		
D					+1	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B						
A	1					
C	1					
B		2				
A			+1	max		
D					+1	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B						
A	1					
C	1					
B		2				
A			3	max		
D					+1	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B						
A	1					
C	1					
B		2				
A			3			
D					+1	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B						
A	1					
C	1					
B		2	max	max		
A			3	max		
D					+1	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B						
A	1					
C	1		max			
B		2	max	max		
A			3	max		
D					+1	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B						
A	1					
C	1	max	max			
B		2	max	max		
A			3	max		
D					+1	max

# Maior subseqüência comum (LCS)

	A	B	A	Z	D	C
B						
A	1	max				
C	1	max	max			
B		2	max	max		
A			3	max		
D					+1	max



# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B		+1				
A	1	max				
C	1	max	max			
B		2	max	max		
A			3	max		
D					+1	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B		1				
A	1	max				
C	1	max	max			
B		2	max	max		
A			3	max		
D					+1	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B		1				
A	1	1				
C	1	max	max			
B		2	max	max		
A			3	max		
D					+1	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B		1				
A	1	1				
C	1	1	max			
B		2	max	max		
A			3	max		
D					+1	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B		1				
A	1	1	+1			
C	1	1	max			
B		2	max	max		
A			3	max		
D					+1	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B		1				
A	1	1	2			
C	1	1	max			
B		2	max	max		
A			3	max		
D					+1	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B		1				
A	1	1	2			
C	1	1	2			
B		2	max	max		
A			3	max		
D					+1	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B		1				
A	1	1	2			
C	1	1	2			
B		2	2	max		
A			3	max		
D					+1	max



# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B		1				
A	1	1	2			
C	1	1	2	max		
B		2	2	max		
A			3	max		
D					+1	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B		1				
A	1	1	2	max		
C	1	1	2	max		
B		2	2	max		
A			3	max		
D					+1	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B		1		max		
A	1	1	2	max		
C	1	1	2	max		
B		2	2	max		
A			3	max		
D					+1	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B		1	max	max		
A	1	1	2	max		
C	1	1	2	max		
B		2	2	max		
A			3	max		
D					+1	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B		1	1	max		
A	1	1	2	max		
C	1	1	2	max		
B		2	2	max		
A			3	max		
D					+1	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B		1	1	1		
A	1	1	2	max		
C	1	1	2	max		
B		2	2	max		
A			3	max		
D					+1	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B		1	1	1		
A	1	1	2	2		
C	1	1	2	max		
B		2	2	max		
A			3	max		
D					+1	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B		1	1	1		
A	1	1	2	2		
C	1	1	2	2		
B		2	2	max		
A			3	max		
D					+1	max



# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B		1	1	1		
A	1	1	2	2		
C	1	1	2	2		
B		2	2	2		
A			3	max		
D					+1	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B		1	1	1		
A	1	1	2	2		
C	1	1	2	2		
B		2	2	2		
A			3	3		
D					+1	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B		1	1	1		
A	1	1	2	2		
C	1	1	2	2		
B		2	2	2		
A			3	3		
D					4	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B		1	1	1		
A	1	1	2	2		
C	1	1	2	2		
B		2	2	2		
A			3	3		max
D					4	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B		1	1	1		
A	1	1	2	2		
C	1	1	2	2		
B		2	2	2		
A			3	3	max	max
D					4	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B		1	1	1		
A	1	1	2	2		
C	1	1	2	2		
B		2	2	2	max	
A			3	3	max	max
D					4	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B		1	1	1		
A	1	1	2	2		
C	1	1	2	2	max	
B		2	2	2	max	
A			3	3	max	max
D					4	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B		1	1	1		
A	1	1	2	2	max	
C	1	1	2	2	max	
B		2	2	2	max	
A			3	3	max	max
D					4	max



# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B		1	1	1	max	
A	1	1	2	2	max	
C	1	1	2	2	max	
B		2	2	2	max	
A			3	3	max	max
D					4	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B		1	1	1	1	
A	1	1	2	2	max	
C	1	1	2	2	max	
B		2	2	2	max	
A			3	3	max	max
D					4	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B		1	1	1	1	
A	1	1	2	2	2	
C	1	1	2	2	max	
B		2	2	2	max	
A			3	3	max	max
D					4	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B		1	1	1	1	
A	1	1	2	2	2	
C	1	1	2	2	2	
B		2	2	2	max	
A			3	3	max	max
D					4	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B		1	1	1	1	
A	1	1	2	2	2	
C	1	1	2	2	2	
B		2	2	2	2	
A			3	3	max	max
D					4	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B		1	1	1	1	
A	1	1	2	2	2	
C	1	1	2	2	2	
B		2	2	2	2	
A			3	3	3	max
D					4	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B		1	1	1	1	
A	1	1	2	2	2	
C	1	1	2	2	2	
B		2	2	2	2	max
A			3	3	3	max
D					4	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B		1	1	1	1	
A	1	1	2	2	2	
C	1	1	2	2	2	+1
B		2	2	2	2	max
A			3	3	3	max
D					4	max



# Maior subsequência comum (LCS)

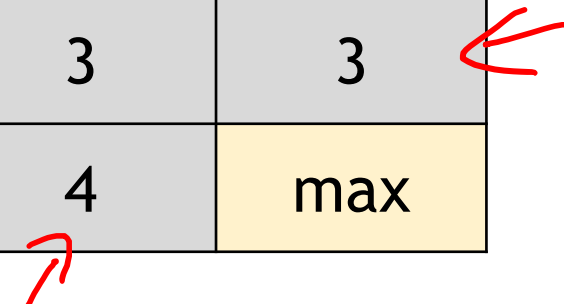
	A	B	A	Z	D	C
B		1	1	1	1	
A	1	1	2	2	2	
C	1	1	2	2	2	3
B		2	2	2	2	max
A			3	3	3	max
D					4	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B		1	1	1	1	
A	1	1	2	2	2	
C	1	1	2	2	2	3
B		2	2	2	2	3
A			3	3	3	max
D					4	max

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B		1	1	1	1	
A	1	1	2	2	2	
C	1	1	2	2	2	3
B		2	2	2	2	3
A			3	3	3	3
D					4	max



# Maior subsequência comum (LCS)

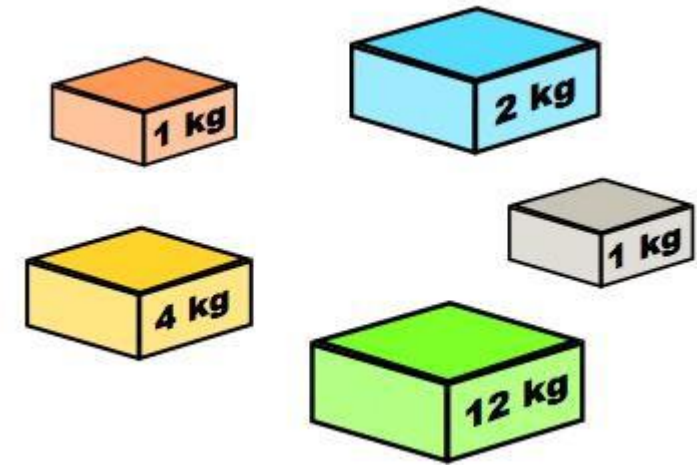
	A	B	A	Z	D	C
B		1	1	1	1	
A	1	1	2	2	2	
C	1	1	2	2	2	3
B		2	2	2	2	3
A			3	3	3	3
D					4	4

# Maior subsequência comum (LCS)

	A	B	A	Z	D	C
B		1	1	1	1	
A	1	1	2	2	2	
C	1	1	2	2	2	3
B		2	2	2	2	3
A			3	3	3	3
D					4	4

# Problema da Mochila

- Problema:
  - Uma mochila suporta até  $W$  quilos
  - Itens devem ser adicionados à mochila
    - Cada item tem um peso  $w_i$  e um valor  $v_i$
    - $w_i$  e  $v_i$  são inteiros
- Objetivo:
  - Qual o valor máximo que não ultrapassa o limite da mochila?



# Problema da Mochila

- Caso base:
  - Se a capacidade da mochila ou a quantidade de itens for zero, então o valor máximo é zero.
- Passo da recursão
  - Senão, há duas opções: incluir ou não incluir (considerando o problema da mochila binária, onde não há repetições de itens)
- Queremos maximizar o valor total carregado sem ultrapassar a capacidade da mochila.

$$\max \sum_{i=0}^n v_i \cdot x_i \quad \text{sujeito a} \quad \sum_{i=0}^n w_i \cdot x_i \leq W \quad x_i \in \{ 0, 1 \}$$

# Problema da Mochila

$w$  = capacidade disponível,  $i$  = item atual

$$f(w, i) = \begin{cases} 0, & w = 0 \text{ ou } i = 0 \\ \max\{ \underbrace{f(w, i - 1)}_{\text{não adicionar item}}, \underbrace{v_i + f(w - w_i, i - 1)}_{\text{adicionar item}} \}, & \text{caso contrário} \end{cases}$$



# Problema da Mochila

- Capacidade da mochila: 12
- $v = \{100, 55, 50\}$
- $w = \{10, 6, 6\}$

$$f(12, 3) = \max\{f(12, 2), 50 + f(6, 2)\}$$

	0	1	2	...	6	...	12
0							
1							
2							
3							max

# Problema da Mochila

- Capacidade da mochila: 12
- $v = \{100, 55, 50\}$
- $w = \{10, 6, 6\}$

$$f(12, 2) = \max\{f(12, 1), 55 + f(6, 1)\}$$

	0	1	2	...	6	...	12
0							
1							
2							max
3							max

# Problema da Mochila

- Capacidade da mochila: 12
- $v = \{100, 55, 50\}$
- $w = \{10, 6, 6\}$

$$f(12, 1) = \max\{f(12, 0), 100 + f(2, 0)\}$$

	0	1	2	...	6	...	12
0			0				0
1							max
2							max
3							max

# Problema da Mochila

- Capacidade da mochila: 12
- $v = \{100, 55, 50\}$
- $w = \{10, 6, 6\}$

$$f(12, 1) = \max\{0, 100 + 0\}$$

	0	1	2	...	6	...	12
0			0				0
1							100
2							max
3							max

# Problema da Mochila

- Capacidade da mochila: 12
- $v = \{100, 55, 50\}$
- $w = \{10, 6, 6\}$

$f(6, 1) = f(6,0)$ , não pode pegar o item 1 pois  $w[0] = 10 > 6$

	0	1	2	...	6	...	12
0			0		0		0
1					$f(6,0)$		100
2							max
3							max

# Problema da Mochila

- Capacidade da mochila: 12
- $v = \{100, 55, 50\}$
- $w = \{10, 6, 6\}$

$f(6, 1) = f(6, 0)$ , não pode pegar o item 0 pois  $w[0] = 10 > 6$

	0	1	2	...	6	...	12
0			0		0		0
1					0		100
2							max
3							max

# Problema da Mochila

- Capacidade da mochila: 12
- $v = \{100, 55, 50\}$
- $w = \{10, 6, 6\}$

$$f(12, 2) = \max\{100, 55 + 0\}$$

	0	1	2	...	6	...	12
0			0		0		0
1					0		100
2							100
3							max

# Problema da Mochila

- Capacidade da mochila: 12
- $v = \{100, 55, 50\}$
- $w = \{10, 6, 6\}$

$$f(6, 2) = \max\{f(6, 1), 55 + f(0, 1)\}$$

	0	1	2	...	6	...	12
0			0		0		0
1					0		100
2					max		100
3							max



# Problema da Mochila

- Capacidade da mochila: 12
- $v = \{100, 55, 50\}$
- $w = \{10, 6, 6\}$

$$f(6, 2) = \max\{0, 55 + 0\}$$

	0	1	2	...	6	...	12
0			0		0		0
1	0				0		100
2					55		100
3							max

# Problema da Mochila

- Capacidade da mochila: 12
- $v = \{100, 55, 50\}$
- $w = \{10, 6, 6\}$

$$f(12, 3) = \max\{f(12, 2), 50 + f(6, 2)\}$$

	0	1	2	...	6	...	12
0			0		0		0
1	0				0		100
2				50 +	55		100
3							max

# Problema da Mochila

- Capacidade da mochila: 12
- $v = \{100, 55, 50\}$
- $w = \{10, 6, 6\}$

$$f(12, 3) = \max\{100, 50 + 55\}$$

	0	1	2	...	6	...	12
0			0		0		0
1	0				0		100
2					55		100
3							105

# Problema da Mochila

- Capacidade da mochila: 12
- $v = \{100, 55, 50\}$
- $w = \{10, 6, 6\}$

$$f(12, 3) = \max\{100, 50 + 55\}$$

	0	1	2	...	6	...	12
0			0		0		0
1	0				0		100
2					55		100
3							105

# Problema da Mochila - Top Down

```
int knapsack(int w, int n){
    if(memo[w][n] != -1)
        return memo[w][n];

    if(w == 0 || n == 0)
        return memo[w][n] = 0;

    if(weight[n-1] > w)
        return memo[w][n] = knapsack(w, n-1);

    return memo[w][n] = max(knapsack(w, n-1), value[n-1] +
                            knapsack(w - weight[n-1], n-1));
}
```

# Problema da Mochila - Bottom Up

```

for(int i=0; i<=n; i++)
    dp[i][0] = 0;
for(int j=0; j<=w; j++)
    dp[0][j] = 0;

for(int i=1; i<=n; i++){
    for(int j=1; j<=w; j++){
        if(weight[i-1] > j)
            dp[i][j] = dp[i-1][j];
        else
            dp[i][j] = max(dp[i-1][j], dp[i-1][j-weight[i-1]] + value[i-1]);
    }
}
  
```

# Mochila: otimizando espaço

- Em nossa solução, estamos utilizando uma matriz  $dp[MAX\_W, MAX\_N]$ .
- Dependendo do problema, isso pode ocasionar estouro de memória.
- Existem algumas formas de otimizar nossa solução para não precisarmos de uma matriz tão grande. Veja algumas delas nos seguintes links:

<https://www.geeksforgeeks.org/space-optimized-dp-solution-0-1-knapsack-problem>

<https://codeforces.com/blog/entry/47247?#comment-316200>

<https://medium.com/@ThatOneKevin/knapsack-problems-part-1-8465fb2d53e9>

# Mochila ilimitada (com repetição)

- Uma variação comum do Problema da Mochila.
- Neste caso podemos considerar que temos uma quantidade ilimitada de cada item. Sendo assim, um mesmo item pode ser colocado mais de uma vez dentro da mochila.



# Mochila ilimitada (com repetição)

- A ideia da nossa solução não irá se alterar muito. De certa forma, será até mais simples.
- Para uma certa capacidade  $i$  da mochila, verificamos todos os itens  $j$  que podem ser colocados nela ( $w[j] \leq i$ ) e qual resulta em maior valor ( $v[j] + dp[i - w[j]]$ )

$$f(i) = \begin{cases} 0 & \text{se } i = 0 \\ \max\{v[j] + f(i - w[j])\} & \forall j | w[j] \leq i \end{cases}$$

# Mochila ilimitada (com repetição)

```
int knapsack(int n, int w){
    memset(dp, 0, sizeof(dp));

    for(int j=1; j<=w; j++){
        for(int i=1; i<=n; i++){
            if(weight[i-1] <= j)
                dp[j] = max(dp[j], dp[j-weight[i-1]] + v[i-1]);
        }
    }
    return dp[w];
}
```

# Diving for Gold (UVA - 990)

- **Problema:** Dado  $n$  tesouros representados por pares (profundidade, quantidade de ouro)
- Para pegar um tesouro, leva-se  $3 * w * profundidade$  segundos (sendo  $w$  uma constante dada pela entrada)
- Temos um cilindro de ar que nos permite ficar  $t$  segundos submersos.
- **Objetivo:** determinar o máximo de tesouros que podemos pegar, e quais são estes tesouros.

# Diving for Gold (UVA - 990)

**SPOILER  
ALERT!**

# Referências

Rene Pegoraro. Aulas de Técnicas de Programação.

Rene Pegoraro e Wilson M. Yonezawa. Aulas de Algoritmos Avançados.

Thiago Alexandre Domingues de Souza. Palestra sobre Programação Dinâmica.

Giulia Moura, João Pedro Comini e Pedro H. Paiola. Aulas de Programação Competitiva I.

Bruno Papa, Maurício Scarelli e Rodrigo Rosseti. Seminário sobre Programação Dinâmica.

LAAKSONEN, A. Competitive Programmer's Handbook.

[https://www.ime.usp.br/~pf/analise\\_de\\_algoritmos/aulas/dynamic-programming.html](https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/dynamic-programming.html)

[http://www.decom.ufop.br/anderson/2\\_2012/BCC241/ProgramacaoDinamica.pdf](http://www.decom.ufop.br/anderson/2_2012/BCC241/ProgramacaoDinamica.pdf)

<https://www.geeksforgeeks.org/tabulation-vs-memoization/>

<https://www.geeksforgeeks.org/solve-dynamic-programming-problem/>

# Referências

<https://sites.google.com/site/ldsicufal/disciplinas/programacao-avancada/notas-de-aula---programao-dinmica>

<https://www.geeksforgeeks.org/longest-common-subsequence-dp-4/>

[https://www.tutorialspoint.com/design\\_and\\_analysis\\_of\\_algorithms/design\\_and\\_analysis\\_of\\_algorithms\\_longest\\_common\\_subsequence.htm](https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_longest_common_subsequence.htm)

<https://neps.academy/lesson/164>

[http://www.facom.ufms.br/~marco/analise2007/aula12\\_4.pdf](http://www.facom.ufms.br/~marco/analise2007/aula12_4.pdf)

[https://github.com/icmcgema/gema/blob/master/09-Programacao\\_Dinamica.ipynb](https://github.com/icmcgema/gema/blob/master/09-Programacao_Dinamica.ipynb)

[https://www.ime.usp.br/~pf/analise\\_de\\_algoritmos/aulas/mochila-bool.html](https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/mochila-bool.html)

<https://www.geeksforgeeks.org/space-optimized-dp-solution-0-1-knapsack-problem>

<https://www.geeksforgeeks.org/unbounded-knapsack-repetition-items-allowed/>