Programação Dinâmica

André, Salles

Departamento de Informática
Universidade Federal de Viçosa

INF 333 - 2024/1

Introdução

Recursão

Para aprender recursão, você primeiro deve aprender recursão.

Programação Dinâmica

Para aprender programação dinâmica, você primeiro deve aprender programação dinâmica.

Example: Coin-row problem

Fila de moedas

- Há uma fila de n moedas cujos valores são inteiros positivos $c_1, c_2, \ldots, c_n$ não necessariamente distintos.
- O objetivo é pegar o máximo valor com a restrição de não pegar duas moedas adjacentes.

- Exemplo: 5 1 2 10 6 2
- Solução: 5 1 2 10 6 2, de valor 17.

Example: Coin-row problem

Para a n-ésima moeda há duas opções:

- Pegar: ganha c_n e continua o processo da $n-2$
- Não pegar: continua o processo da $n-1$

$$F(n) = \max\{c_n + F(n-2),\ F(n-1)\} \quad \text{for } n > 1,$$
$$F(0) = 0, \qquad F(1) = c_1.$$

Example: Coin-row problem

ALGORITHM *CoinRow*$(C[1..n])$

//Applies formula (8.3) bottom up to find the maximum amount of money
//that can be picked up from a coin row without picking two adjacent coins
//Input: Array $C[1..n]$ of positive integers indicating the coin values
//Output: The maximum amount of money that can be picked up
$F[0] \leftarrow 0; \quad F[1] \leftarrow C[1]$
for $i \leftarrow 2$ **to** n **do**
    $F[i] \leftarrow \max(C[i] + F[i-2], F[i-1])$
return $F[n]$

Example: Coin-row problem

$F[0] = 0$, $F[1] = c_1 = 5$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5					

$F[2] = \max\{1 + 0, 5\} = 5$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5				

$F[3] = \max\{2 + 5, 5\} = 7$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7			

$F[4] = \max\{10 + 5, 7\} = 15$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15		

$F[5] = \max\{6 + 7, 15\} = 15$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15	15	

$F[6] = \max\{2 + 15, 15\} = 17$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15	15	**17**

Example: Change-making problem

Troco com mínimo de moedas

- Devolver um troco de valor n com o mínimo de moedas de valores $d_1 < d_2 < \cdots < d_m$, sendo $d_1 = 1$.

- Para as moedas usadas na maioria dos países o método guloso funciona: usar a de maior valor possível até acabar.
- Mas não funciona para $n = 6$ e moedas $1, 3, 4$, por exemplo.

- Programação dinâmica resolve o caso geral de forma eficiente.

Example: Change-making problem

Para um valor n, testar cada valor de moeda $d_j \leq n$

- devolvendo d_j, sobra $n - d_j$ para devolver
- escolher o que dá menor quantidade de moedas

$$F(n) = \min_{j:n \geq d_j} \{F(n - d_j)\} + 1 \quad \text{for } n > 0,$$

$$F(0) = 0.$$

Example: Change-making problem

ALGORITHM *ChangeMaking*$(D[1..m], n)$

//Applies dynamic programming to find the minimum number of coins
//of denominations $d_1 < d_2 < \cdots < d_m$ where $d_1 = 1$ that add up to a
//given amount n
//Input: Positive integer n and array $D[1..m]$ of increasing positive
// integers indicating the coin denominations where $D[1] = 1$
//Output: The minimum number of coins that add up to n
$F[0] \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
  $temp \leftarrow \infty$; $j \leftarrow 1$
  while $j \leq m$ **and** $i \geq D[j]$ **do**
    $temp \leftarrow \min(F[i - D[j]], temp)$
    $j \leftarrow j + 1$
  $F[i] \leftarrow temp + 1$
return $F[n]$

Example: Change-making problem

$F[0] = 0$

n	0	1	2	3	4	5	6
F	0						

$F[1] = \min\{F[1-1]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1					

$F[2] = \min\{F[2-1]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2				

$F[3] = \min\{F[3-1], F[3-3]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1	2	1			

$F[4] = \min\{F[4-1], F[4-3], F[4-4]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1	2	1	1		

$F[5] = \min\{F[5-1], F[5-3], F[5-4]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2	1	1	2	

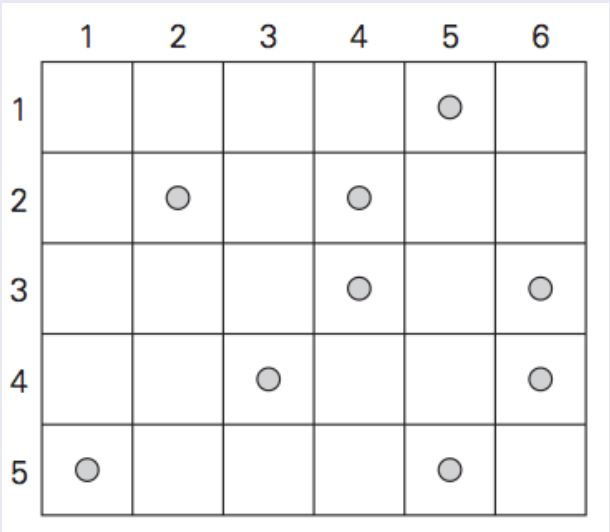
$F[6] = \min\{F[6-1], F[6-3], F[6-4]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2	1	1	2	**2**

Example: Coin-collecting problem

Coletor de moedas

- Várias moedas são espalhadas numa grade de $n \times m$ casas
- Um robô, localizado no canto superior-esquerdo, deve coletar o máximo de moedas e levá-las até o canto inferior-direito
- Ele coleta as moedas das casas que visita e em cada passo pode ir de sua posição atual para a casa à direita ou a casa abaixo



- O objetivo é determinar o máximo de moedas que o robô pode coletar (e o caminho pelo qual ele consegue isto)

Example: Coin-collecting problem

Para cada casa, escolher se é melhor vir de cima ou da esquerda

$$F(i, j) = \max\{F(i-1, j), F(i, j-1)\} + c_{ij} \quad \text{for } 1 \le i \le n, \quad 1 \le j \le m$$
$$F(0, j) = 0 \text{ for } 1 \le j \le m \quad \text{and} \quad F(i, 0) = 0 \text{ for } 1 \le i \le n,$$

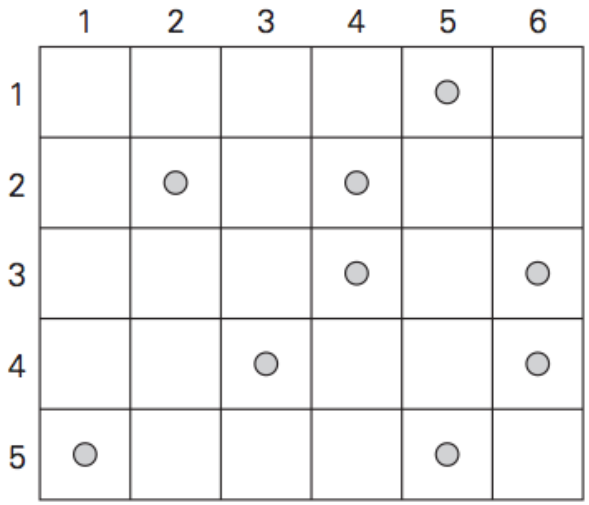
Example: Coin-collecting problem

ALGORITHM *RobotCoinCollection*$(C[1..n, 1..m])$

//Applies dynamic programming to compute the largest number of
//coins a robot can collect on an $n \times m$ board by starting at (1, 1)
//and moving right and down from upper left to down right corner
//Input: Matrix $C[1..n, 1..m]$ whose elements are equal to 1 and 0
//for cells with and without a coin, respectively
//Output: Largest number of coins the robot can bring to cell (n, m)

$F[1, 1] \leftarrow C[1, 1]$; **for** $j \leftarrow 2$ **to** m **do** $F[1, j] \leftarrow F[1, j-1] + C[1, j]$
for $i \leftarrow 2$ **to** n **do**
    $F[i, 1] \leftarrow F[i-1, 1] + C[i, 1]$
    for $j \leftarrow 2$ **to** m **do**
        $F[i, j] \leftarrow \max(F[i-1, j], F[i, j-1]) + C[i, j]$
return $F[n, m]$

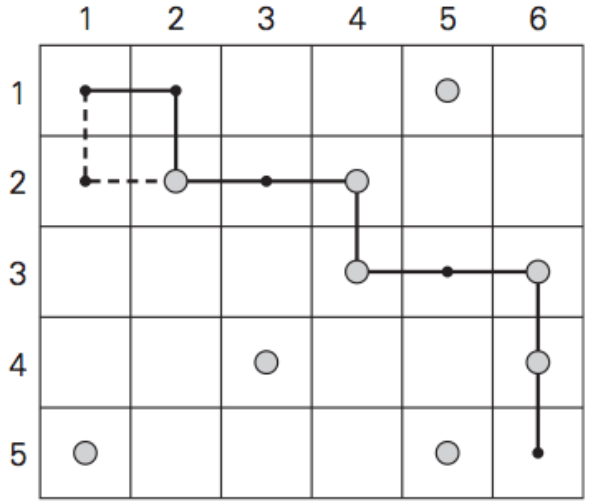
Example: Coin-collecting problem



(a)

	1	2	3	4	5	6
1	0	0	0	0	1	1
2	0	1	1	2	2	2
3	0	1	1	3	3	4
4	0	1	2	3	3	5
5	1	1	2	3	4	**5**

(b)



(c)

obs.: caminho pode ser recuperado começando do final e verificando a cada passo se é melhor (valor maior na tabela) vir de cimda ou da direita.

Exemplo - Problema da Mochila (*knapsack problem*)

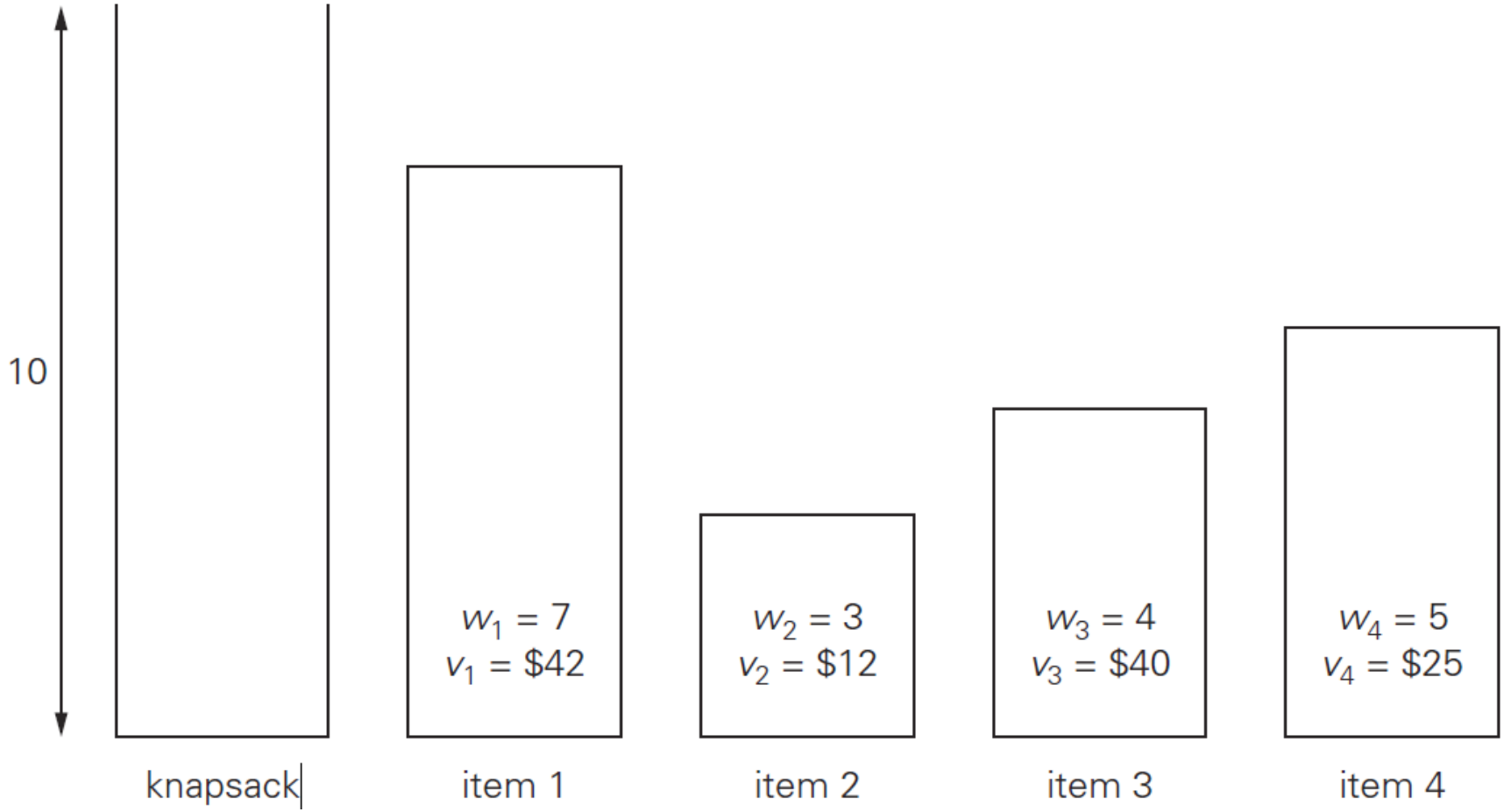
Problema da mochila

Dados n itens:

- Pesos: $w_1, w_2, \ldots, w_n$.
- Valores: $v_1, v_2, \ldots, v_n$.
- e uma mochila de capacidade W.

Que itens colocar na mochila para obter valor máximo sem ultrapassar sua capacidade?

Exemplo - Problema da Mochila (*knapsack problem*)



Exemplo - Problema da Mochila (*knapsack problem*)

Solução gulosa

- escolher por ordem decrescente de valor
  - não funciona! verifique...
- escolher por ordem crescente de peso
  - não funciona! verifique...
- escolher por ordem de custo-benefício: valor/peso
  - funciona este exemplo, mas não em todos
  - por exemplo, se $w_1 = 6$ e $W = 6$

WA: não existe método guloso que funciona para qualquer entrada

Exemplo - Problema da Mochila (*knapsack problem*)

Solução força-bruta: testar todas as alternativas

- considere todos os subconjuntos dos n itens
- calcule o peso total de cada subconjunto para identificar os que cabem na mochila
- calcule o valor total de cada subconjunto para identificar o de maior valor entre eles
  (ilustrado no próximo slide)

TLE: método $O(2^n)$, muito lento para n grande

Exemplo - Problema da Mochila (*knapsack problem*)

Subset	Total weight	Total value
∅	0	$ 0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1, 2}	10	$54
{1, 3}	11	not feasible
{1, 4}	12	not feasible
{2, 3}	7	$52
{2, 4}	8	$37
{3, 4}	**9**	**$65**
{1, 2, 3}	14	not feasible
{1, 2, 4}	15	not feasible
{1, 3, 4}	16	not feasible
{2, 3, 4}	12	not feasible
{1, 2, 3, 4}	19	not feasible

Exemplo - Problema da Mochila (*knapsack problem*)

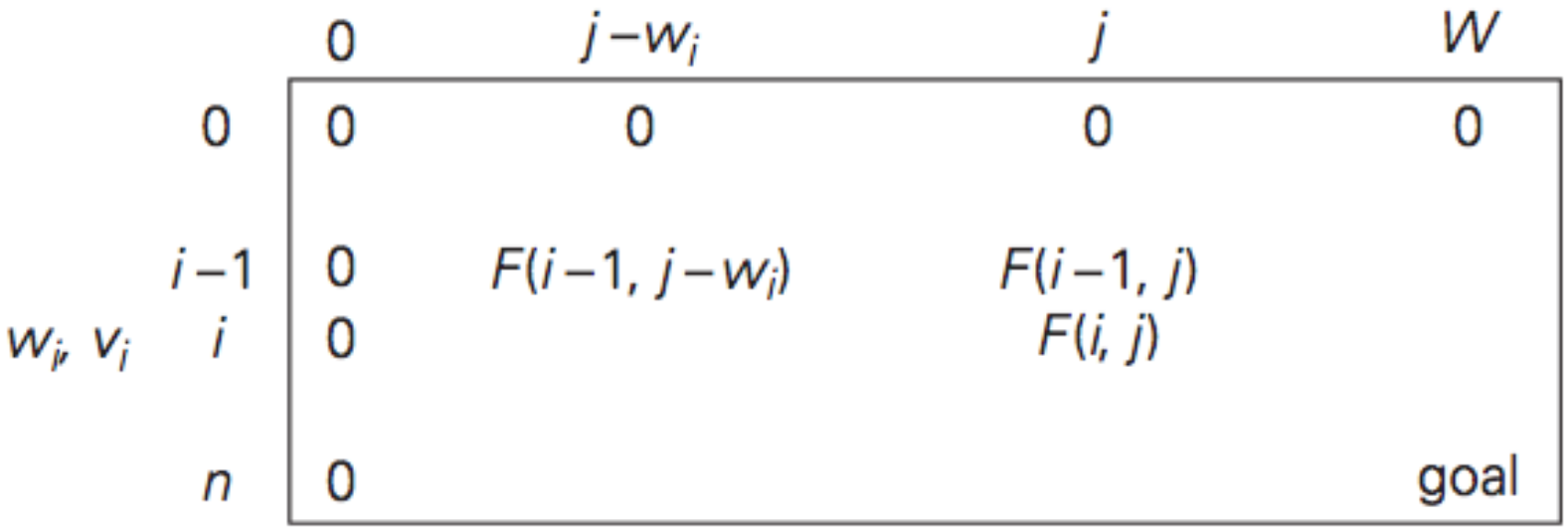
Solução por programação dinâmica

- Seja $F(i, j)$ o melhor valor com itens $1 \ldots i$ e capacidade j
- Um opção é não usar o item i
  - Teríamos o valor de $F(i-1, j)$
- Outra opção é usar o item i
  - Ganharíamos o valor v_i...
  - ... mas usaríamos w_i da capacidade
  - Teríamos então o valor $v_i + F(i-1, j-w_j)$
- A melhor opção é a que der maior valor

Exemplo - Problema da Mochila (*knapsack problem*)

$$F(i, j) = \begin{cases} \max\{F(i-1, j), v_i + F(i-1, j-w_i)\} & \text{if } j - w_i \geq 0, \\ F(i-1, j) & \text{if } j - w_i < 0. \end{cases}$$

$$F(0, j) = 0 \text{ for } j \geq 0 \quad \text{and} \quad F(i, 0) = 0 \text{ for } i \geq 0.$$



Example: Knapsack problem

ALGORITHM *MFKnapsack*(i, j)

```
//Implements the memory function method for the knapsack problem
//Input: A nonnegative integer i indicating the number of the first
//       items being considered and a nonnegative integer j indicating
//       the knapsack capacity
//Output: The value of an optimal feasible subset of the first i items
//Note: Uses as global variables input arrays Weights[1..n], Values[1..n],
//and table F[0..n, 0..W] whose entries are initialized with −1's except for
//row 0 and column 0 initialized with 0's
if F[i, j] < 0
    if j < Weights[i]
        value ← MFKnapsack(i − 1, j)
    else
        value ← max(MFKnapsack(i − 1, j),
                    Values[i] + MFKnapsack(i − 1, j − Weights[i]))
    F[i, j] ← value
return F[i, j]
```

Exemplo - Problema da Mochila (*knapsack problem*)

item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

capacity $W = 5$.

Exemplo - Problema da Mochila (*knapsack problem*)

Por uma PD bottom-up

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2,\ v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1,\ v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3,\ v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2,\ v_4 = 15$	4	0	10	15	25	30	**37**

Exemplo - Problema da Mochila (*knapsack problem*)

Por uma PD top-down (a do algoritmo mostrado)

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2,\ v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1,\ v_2 = 10$	2	0	—	12	22	—	22
$w_3 = 3,\ v_3 = 20$	3	0	—	—	22	—	32
$w_4 = 2,\ v_4 = 15$	4	0	—	—	—	—	**37**

Exemplo de aplicação de PD

UVa 11450 - Wedding Shopping (link)

Dados diferentes modelos de cada peça de vestuário (ex.: 3 opções de camisa, 2 de gravata, 4 de sapato, ...) e um orçamento limitado, comprar um modelo de cada peça. O valor total da compra deve ser o máximo possível, dentro do orçamento.

Entrada:

- M: orçamento ($1 \leq M \leq 200$)
- C: número de peças de vestuário ($1 \leq C \leq 20$)
- Para cada peça $i \in [0 \ldots C-1]$ de vestuário
  - K_i: quantidade de modelos ($1 \leq K_i \leq 20$) e preço de cada um

Exemplo de entrada:

- $M = 20, C = 3$
- Peça 1 : 3 modelos $\rightarrow$ 6 4 8
- Peça 2 : 2 modelos $\rightarrow$ 5 10
- Peça 3 : 4 modelos $\rightarrow$ 1 5 3 5

Exemplo de aplicação de PD

- Força bruta: tentar todas as alternativas
  - TLE: no pior caso podem ser 20^{20} alternativas
- Guloso: escolher o modelo mais caro que ainda pode comprar
  - WA: considere o exemplo anterior com $M = 12$
- PD Top-down:
  - Como um backtraking, cada chamada aumenta uma peça, testando todos os modelos (diminuindo seu valor do orçamento)
  - A chamada recursiva inclui o índice da peça e o orçamento restante
  - Tabela com resultado já encontrado para peça i e orçamento j
- PD Bottom-up:
  - Matriz de bool: T_{ij} true se é possível gastar j com peças $1 \ldots i$
  - Peça 1, T_{1j} true $\forall j = P_{1k}$, sendo P_{1k} os preços dos modelos
  - Peça $i > 1$, T_{ij} true, $\forall j = r + P_{ik}$, sendo r valores que $T_{i-1,r}$ true
  - Recuperar o máximo j que T_{Cj} é true.