Algoritmos e Estruturas de Dados II

Lucila Bento
lucila.bento [at] ime.uerj.br

Programação Dinâmica

Se tivermos uma recorrência para resolver determinado problema, a técnica de Programação Dinâmica procura responder a seguinte pergunta:

É possível implementar a recorrência sem o uso de recursão?

A programação dinâmica evita o uso de recursão, usando "memorização", só que resolvendo e tabelando os subproblemas de forma "bottom-up" de tal maneira que, ao se precisar resolver dado problema, os subproblemas menores necessários já foram previamente tabelados na matriz de memorização.

Por forma "bottom-up" deve-se entender que a solução dos subproblemas deve ser tabelada por ordem de tamanho, dos menores para os maiores.

Programação Dinâmica

Moedas

Programação Dinâmica

Moedas: Dados os m tipos de moedas de um país, determinar o número de maneiras distintas para dar um troco de valor n.

Exemplo:
m = 6
V = {1, 5, 10, 25, 50, 100}
n = 26
Há 13 maneiras distintas.

25	1					
10	10	5	1			
10	10	1	...	1		
10	5	5	5	1		
10	5	5	1	...	1	
10	5	1	...	1		
10	1	...	1			
5	5	5	5	5	1	
5	5	5	5	1	,,,	1
5	5	5	1	...	1	
5	5	1	...	1		
5	1	...	1			
1	...	1				

Programação Dinâmica

Moedas

Formulação recursiva: dados m, V[m], n

T(p, n) = formas distintas de dar um troco n, usando os p tipos iniciais de moedas, V[1]...V[p]

$T(p, n) = 0, (n < 0)$

$T(p, n) = 1, (n = 0)$

$T(p, n) = T(p, n - V[p]) + T(p-1, n), (n > 0)$

A solução do problema é obter T(m, n).

Observe que a formulação é a mesma feita no estudo de memorização.

Programação Dinâmica

Exemplo:

V = {1, 5, 10, 25, 50, 100}

m = 6

n = 26

Há 13 maneiras distintas:

T(4,26) =T(4,1)+...=1+..	→	25, 1
T(3,26)=T(3,16)+...=6+..	→	10, 10, 5, 1
		10, 10, 1...1
		10, 5, 5, 5, 1
		10, 5, 5, 1...1
		10, 5, 1...1
		10, 1...1
T(2,26)=T(2,21)+...=5 + ...	→	5, 5, 5, 5, 5, 1
		5, 5, 5, 5, 1..1
		5, 5, 5, 1..1
		5, 5, 1..1
		5, 1..1
T(1,26) = 1	→	1,1...1

Programação Dinâmica

Exemplo: V = {1, 5, 10, 25, 50, 100} m = 6, n = 20

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	2	2	2	2	2	3	3	3	3	3	4	4	4	4	4	5
3	1	1	1	1	1	2	2	2	2	2	4	4	4	4	4	6	6	6	6	6	9

Observando os subproblemas necessários para calcular T[i,j], verificamos que eles estão tabelados na linha i, à esquerda do ponto [i,j] e na linha i-1, acima do ponto [i,j]. Logo, se preenchermos a matriz, sucessivamente, tabelando os subproblemas por linha ou por coluna, para calcular T[i,j], já temos os resultados necessários. Portanto, a programação dinâmica pode ser utilizada.

Programação Dinâmica

Moedas $T[p,n] = T[p, n-V[p]] + T[p-1,n]$

A solução por PD consiste em preencher toda a tabela m x n por ordem crescente do tamanho dos subproblemas, sem necessidade de recursão. Há duas formas de fazer isso para Moedas:

→ + ↓ ou ↓ + →

Ex. da 1ª forma:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	2	2	2	2	2	3	3	3	3	3	4	4	4	4	4	5
3	1	1	1	1	1	2	2	2	2	2	4	4	4	4	4	6	6	6	6	6	9

Programação Dinâmica

Complexidade: $O(m.n)$

Moedas

$T(p, n) = 0, (n < 0)$ ou $(p = 0)$

$T(p, n) = 1, (n = 0)$

$$T(p, n) = T(p, n - V[p]) + T(p-1,n) , (n > 0)$$

```
Algoritmo, com preenchimento linha x coluna:
Moedas():
   para j ← 0 até n incl.:
           T[0, j] ← 0
    para i ← 1 até m incl.:
       T[i, 0] ← 1
       para j ← 1 até n incl.:
         se j ≥ V[i]:
               T[i,j] ← T[i,j-V[i]] + T[i-1,j]
         senão:
               T[i, j] ← T[i-1,j]
```

Programação Dinâmica

Moedas PD x Memorização

V = {1, 5, 10, 25, 50, 100}

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	1	-1	-1	-1	-1	2	-1	-1	-1	-1	3	-1	-1	-1	-1	4	-1	-1	-1	-1	5
3	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	4	-1	-1	-1	-1	-1	-1	-1	-1	-1	9
4	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	9
5	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	9
6	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	9

Programação Dinâmica

Moedas

V = {1, 5, 10, 25, 50, 100} m = 6, n = 20

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	2	2	2	2	2	3	3	3	3	3	4	4	4	4	4	5
3	1	1	1	1	1	2	2	2	2	2	4	4	4	4	4	6	6	6	6	6	9
4	1	1	1	1	1	2	2	2	2	2	4	4	4	4	4	6	6	6	6	6	9
5	1	1	1	1	1	2	2	2	2	2	4	4	4	4	4	6	6	6	6	6	9
6	1	1	1	1	1	2	2	2	2	2	4	4	4	4	4	6	6	6	6	6	9

Programação Dinâmica

Moedas

Outra formulação recursiva: dados m, n

$T(p, n)$ = formas distintas de dar um troco n, usando os p tipos iniciais de moedas, $V[1]...V[p]$

$T(p, n) = 0, (n < 0)$

$T(p, n) = 1, (n = 0)$

$T(p, n) = \Sigma\, T(i, n - V[i]), (n > 0), 1 \leq i \leq p$

A solução do problema é obter $T(m, n)$.

Observe que a formulação é diferente daquela feita no estudo de memorização.

Programação Dinâmica

Moedas

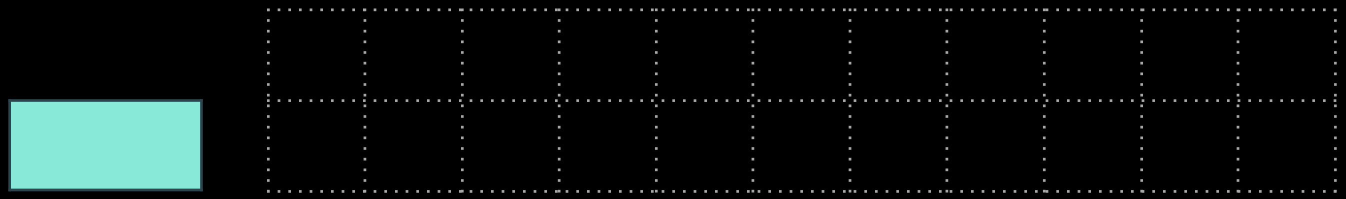
V = {1, 5, 10, 25, 50, 100} m = 6, n = 20

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	2	2	2	2	2	3	3	3	3	3	4	4	4	4	4	5
3	1	1	1	1	1	2	2	2	2	2	4	4	4	4	4	6	6	6	6	6	9
4	1	1	1	1	1	2	2	2	2	2	4	4	4	4	4	6	6	6	6	6	9
5	1	1	1	1	1	2	2	2	2	2	4	4	4	4	4	6	6	6	6	6	9
6	1	1	1	1	1	2	2	2	2	2	4	4	4	4	4	6	6	6	6	6	9

Exercício recomendado

Escrever uma recorrência para o seguinte problema:

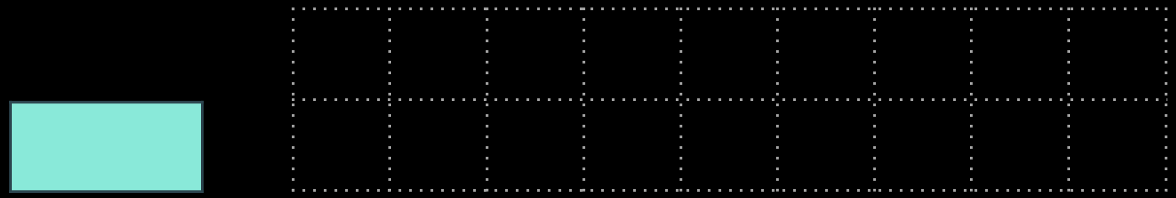
Dado um conjunto de azulejos de dimensões 2 x 1, quer-se saber de quantas maneiras distintas pode-se azulejar uma parede de dimensões 2 x n, com esses azulejos.



Programação Dinâmica

Escrever uma recorrência para o seguinte problema:

Dado um conjunto de azulejos de dimensões 2 x 1, quer-se saber de quantas maneiras distintas pode-se azulejar uma parede de dimensões 2 x n, com esses azulejos.



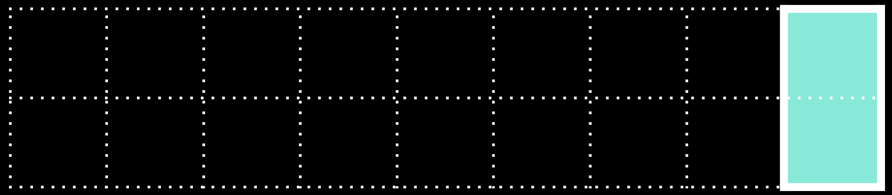
Solução:

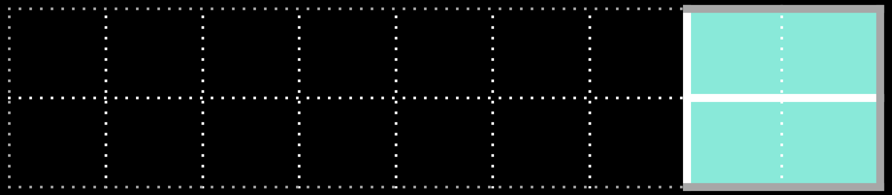
$T(n)$ = número de maneiras distintas para azulejar a parede.

$T(1) = 1$

$T(2) = 2$

$T(n) = ?$





$$T(n) = T(n-1) + T(n-2)$$

Programação Dinâmica

Ideia e abordagens

Programação Dinâmica

"Quem não se lembra do passado está condenado a repeti-lo"

Em algoritmos que usam divisão e conquista é comum haver a repetição de subproblemas. Isso pode acabar gerando muito recálculo.

A Programação Dinâmica vem tentar resolver este problema.

Programação Dinâmica

IDEIA:
Armazenar a solução dos subproblemas para serem utilizados no futuro.

Pode ser feito por duas abordagens:
Top Down (Memorização)
Botton up (tabulação)

IMPORTANTE:
Para que esse paradigma possa ser aplicado, é necessário que o problema tenha uma estrutura recursiva, a solução de toda instância do problema deve "conter" soluções de subinstância dessa instância.

Memorização (Top Down)

Do geral para o específico, de cima para baixo.

- Normalmente essa abordagem é a mais simples de se aplicar, pois ainda faz o uso de algoritmos recursivos.
- Visita apenas os estados requisitados.

Método:

- O problema é dividido em subproblemas.
- Cada subproblema é resolvido recursivamente.
- Quando um subproblema é resolvido, o resultado é armazenado para possíveis utilizações no futuro.

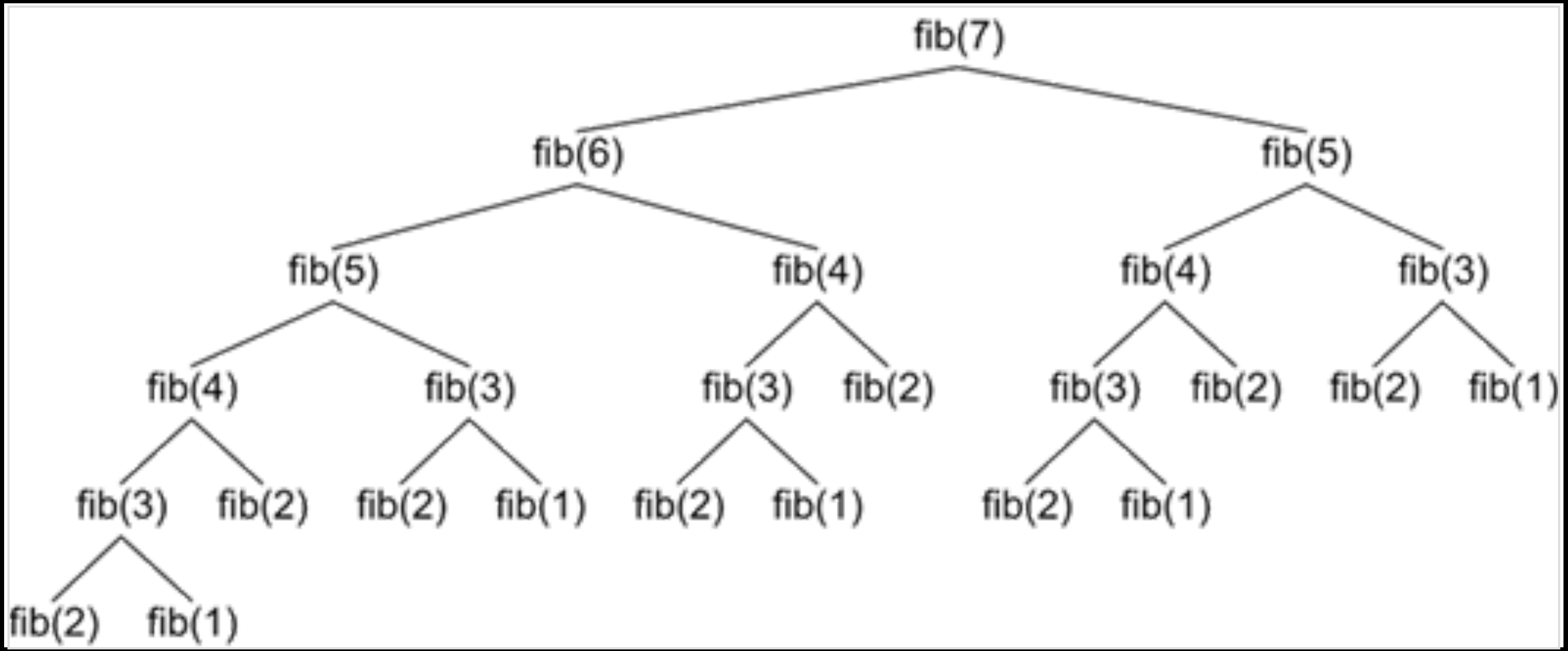
Memorização (Top Down)

```
int memo[] = {1,1,-1,-1,-1,-1,...} //-1 =
não calculado

int fib(int i){
    if (memo[i] == -1)
        memo[i] = fib(i-1) + fib (i-2);
    return memo[i];
}
```

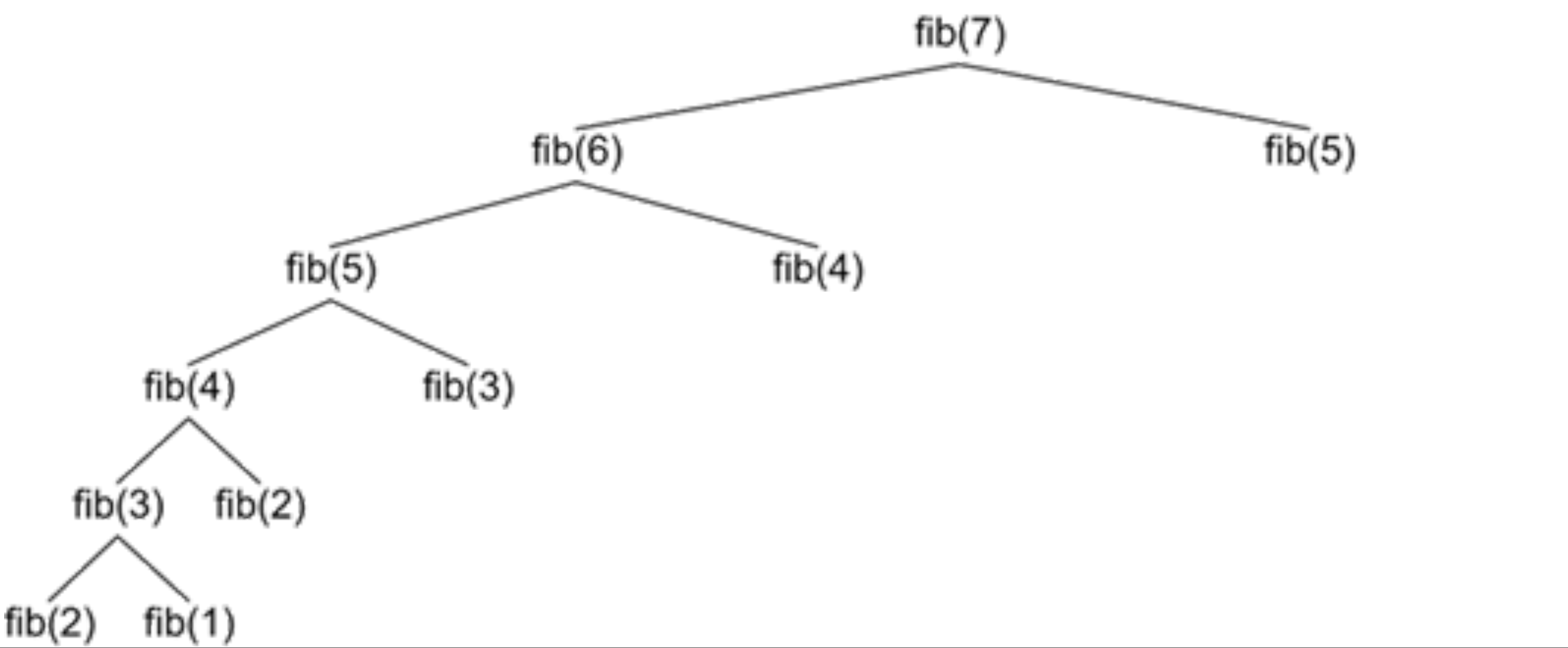
Memorização (Top Down)

Com Divisão e Conquista



Memorização (Top Down)

Com Programação Dinâmica



Tabulação (Bottom Up)

Do específico para o geral, de baixo para cima.

- Visita todos os estados.

Método:

- O problema é dividido em subproblemas.
- Cada subproblema é resolvido, se iniciando pelos que são base para a solução dos seguintes (de forma geral, isso é feita iterativamente).
- Quando um subproblema é resolvido, o resultado é armazenado para resolver subproblemas futuros, até alcançar o problema original.

Tabulação (Bottom Up)

```
int memo[MAXN];

void preprocess(int n){
    memo[0] = memo[1] = 1;
    for(int i = 2; i<n; i++)
        memo[i] = memo[i-1]+memo[i-2];
}
int fib(int i){
    return memo[i];
}
```

Propriedades necessárias

Subestrutura ótima

A solução ótima do problema é composta pela solução ótima de partes menores e mais simples do problema. Exemplo: fib(n) = fib(n-1)+fib(n-2)

Lembrando que nem sempre é fácil ou intuitivo ver como as soluções dos subproblemas devem ser combinadas para obter a solução do problema original.

Propriedades necessárias

Sobreposição de subproblemas

É necessário haver a repetição de subproblemas, do contrário, não faz muito sentido armazenar a solução de um subproblema que nunca mais será necessária.

Exemplo: fib(5) = fib(4)+fib(3)

fib(4) = fib(3)+fib(2),...

fib(3) = fib(2)+fib(1),...

Estratégia geral

Ideia básica é simples, mas é um desafio aplica-la em diferentes problemas.

Não existe um "receita de bolo" para a aplicação, mas existem dicas e estratégias.

Em especial, temos que nos focar em encontrar padrões de recorrência no nosso problema.

Uso convencionais:

- Encontrar uma solução ótima;
- Contar o número de soluções possíveis.

Estratégia geral

Definir os subproblemas (estados)

Escrever a recorrência que relaciona os subproblemas (estados)

Reconhecer e solucionar o caso base

Estratégia geral

Conceito aplicado à Sequência de Fibonacci

1. Definir os subproblemas

   fib(i) -> subproblemas: fib(i-1) e fib(i-2)

2. Escrever a recorrência que relaciona os subproblemas

   fib(i) = fib(i-1) + fib(i-2)

3. Reconhecer e solucionar os casos base

   fib(0) = 1 e fib(1)=1

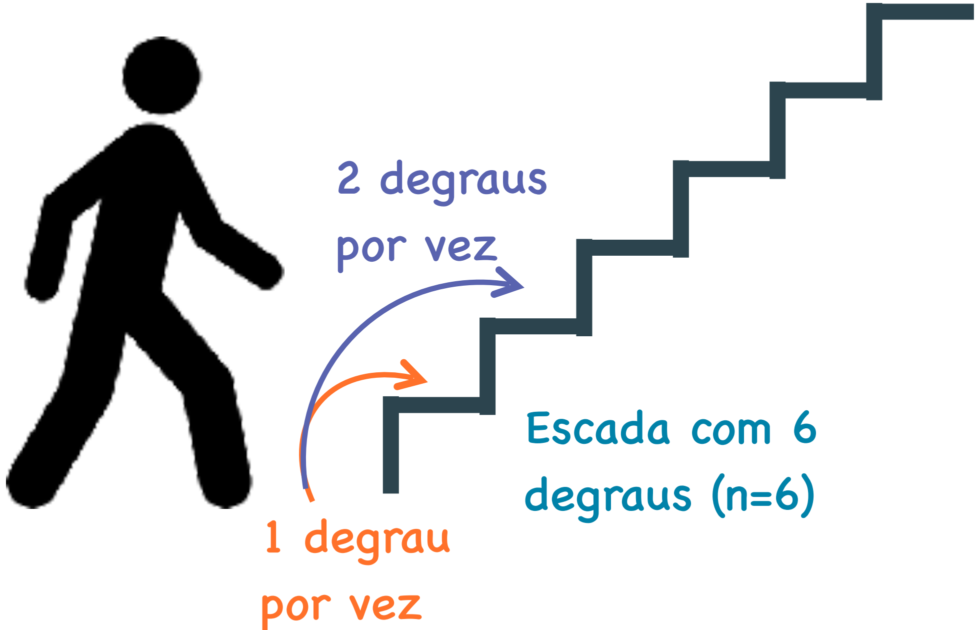
$$fib(i) = \begin{cases} 1,\ i = 0\ ou\ i = 1 \\ fib(i-1) + fib(i-2),\ i > 1 \end{cases}$$

Programação Dinâmica

Problema da Escada

Problema da Escada

Quantas forma há para subir uma escada de n degraus, sendo que em cada passo pode-se subir 1 ou 2 degraus por vez?



Problema da Escada

Quantas forma há para subir uma escada de n degraus, sendo que em cada passo pode-se subir 1 ou 2 degraus por vez?

Exemplos de possibilidades para n = 6:

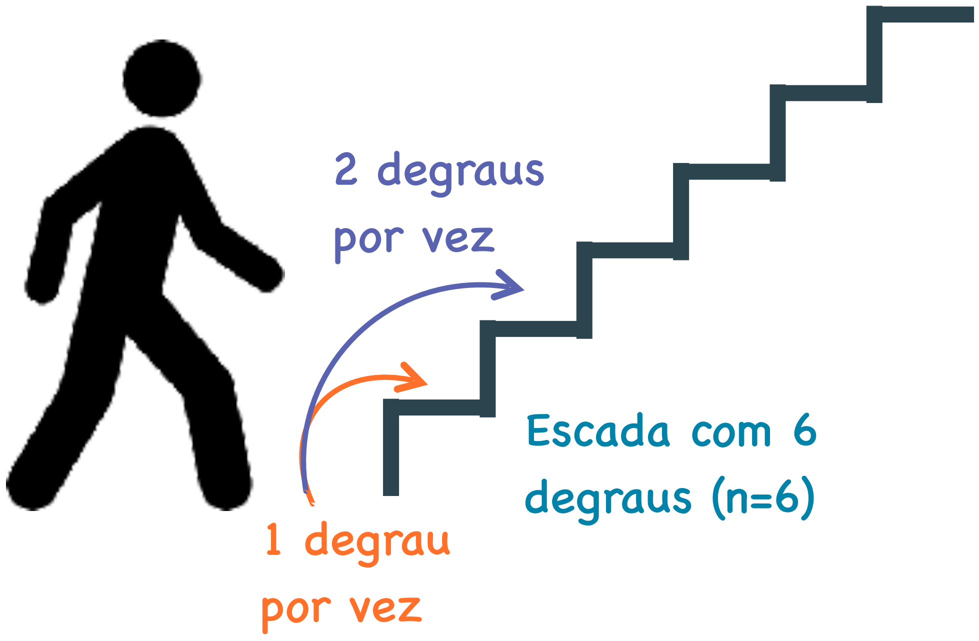
1, 1, 1, 1, 1, 1

2, 1, 1, 1, 1

2, 2, 2

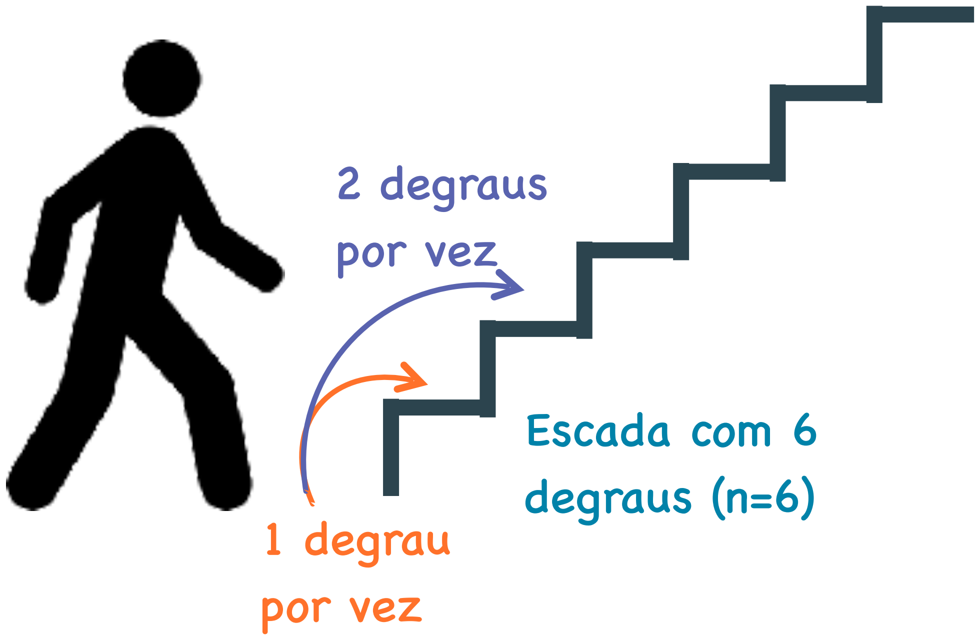
2, 1, 1, 2

1, 1, 1, 1, 2



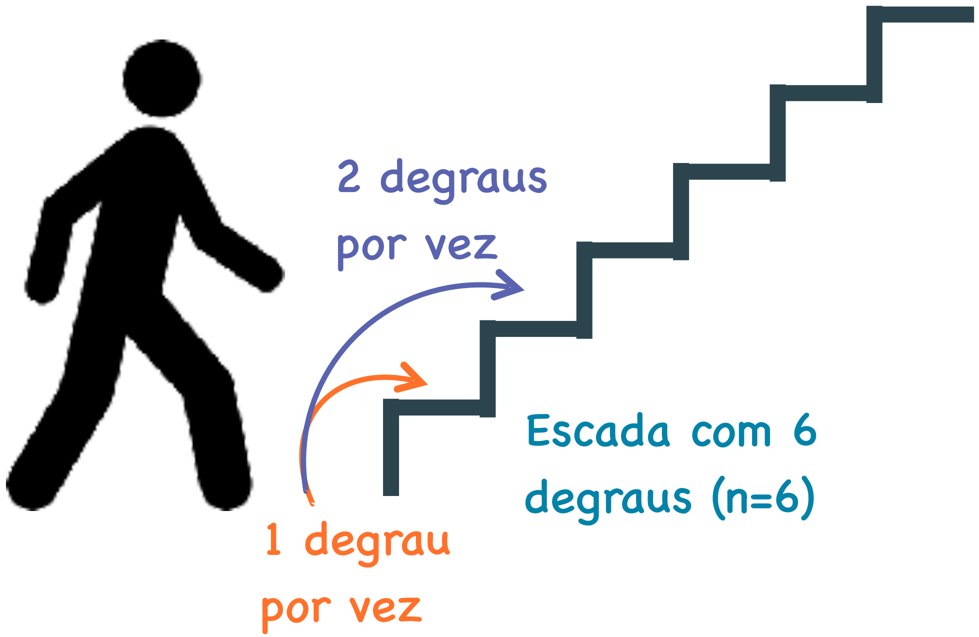
Problema da Escada

- Vamos considerar que o problema será resolvido por uma função $f(n)$, onde n é o número de degraus.
- Considerando que já estamos no degrau n, em quais degraus poderíamos estar no passo anterior?



Problema da Escada

$$f(n) = \begin{cases} 1, & se\ n = 0 \\ 1, & se\ n = 1 \\ f(n-1) + f(n-2), & se\ n > 1 \end{cases}$$



Programação Dinâmica

Problema do troco

Problema do Troco

Problema: É necessário dar troco de um valor x com o menor número de moedas possível.

Para moedas M = {1, 5, 10, 25} e troco = 26, qual é a árvore de recursão?

Problema do Troco

Relação de recorrência

$$f(w) = \begin{cases} 0, \;\; se\ w = 0 \\ 1 + \min\left\{ f(w - moeda[i])\right\}, \; i = 0,\ldots m - 1 \end{cases}$$

Problema do Troco

Implementação Top-down

Em C++

```
vector<int> moedas = {50, 25, 10, 5,1};
map<int, int> memo;
int troco(int x){
    if (x == 0)
        return 0
    if (memo.count(x))
        return memo[x];
    memo[x] = INT_MAX
    for(int m : moedas){
        if (m > x)
            continue;
        memo[x] = min(memo[x], 1 + troco(x-m));
    }
   return memo[x];
}
```

Map em C++ armazena elementos formados por uma combinado de chave-valor

Função do C++ que procura pela chave x e retorna o número de correspondências

Problema do Troco

Implementação Bottom-up

Em C++

```cpp
int minmoedas(vector<int>& moedas, int w){
  int n = moedas.size();
  vector<int> dp(w+1, INT_MAX);

  dp[0] = 0;
  for (int i = 1;  i <= w; i++)
    for (int j = 0; j < n; j++)
      if (moedas[j] <= i)
        dp[i] = min(dp[i], dp[i-moedas[j]]+1);
    return dp[w];
}
```

Complexidade:
O(n.w)

Programação Dinâmica

Partição de Inteiros

Partição de Inteiros

Dado n inteiro, determinar o número de maneiras de particionar n, tal que cada parcela seja $\leq n$.

Exemplo 1:

n = 4

5 maneiras distintas:

4

3 + 1

2 + 2

2 + 1 + 1

1 + 1 + 1 + 1

Exemplo 2:

n = 6

11 maneiras distintas:

6

5 + 1

4 + 2

4 + 1 + 1

3 + 3

3 + 2 + 1

3 + 1 + 1

2 + 2 + 2

2 + 2 + 1 + 1

2 + 1 + 1 + 1 + 1

1 + 1 + 1 + 1 + 1 + 1

Partição de Inteiros

Formulação recursiva: dados n

$T(p, n)$ = número de partições de n onde a maior parcela $\leq p$

$T(p, n) = 0$, se $(n < 0)$ ou $(n > 0$ e $p = 0)$

$T(p, n) = 1$, se $(n = 0)$

$T(p, n) = T(p, n - p) + T(p-1, n)$, $n > 0$, $p > 0$

Procura-se obter $T(n, n)$.

Pode-se implementar a recursão com "memorização".

Partição de Inteiros

$T(p, n) = 0$, se $(n < 0)$ ou $(n > 0, p = 0)$
$T(p, n) = 1$, se $(n = 0)$
$T(p, n) = T(p, n - p) + T(p-1, n)$, $n > 0$, $p > 0$
Quer-se obter $T(n, n)$.

A idéia da programação dinâmica (PD) é evitar totalmente o uso de recursão, resolvendo os subproblemas de forma "bottom-up". Para tanto, todos os sub-problemas menores necessários já devem ter sido resolvidos antes.

Como vimos, por forma "bottom-up" deve-se entender que a solução dos subproblemas deve ser feita por ordem de tamanho, dos menores para os maiores.

Neste caso, é possível usar a idéia da PD!

Partição de Inteiros

$T(p, n) = 0$, se $(n < 0)$ ou $(n > 0, p = 0)$
$T(p, n) = 1$, se $(n = 0)$
$T(p, n) = T(p, n - p) + T(p-1, n)$, $n > 0$, $p > 0$
Quer-se obter $T(n, n)$.

Para implementar a programação dinâmica sem uso de recursão, deve-se calcular $T(p, n)$ em ordem crescente de n e p, começando por qualquer um dos dois parâmetros.

Partição de Inteiros - Cálculo de T(5, 5)

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

T(2,5) = T(2,5−2)+T(1,5)

T(4,5) = T(4,5−4)+T(3,5)

Partição de Inteiros

Algoritmo Bottom-up

$T(p, n) = 0$, se $(n < 0)$ ou

$(n > 0, p = 0)$

$T(p, n) = 1$, se $(n = 0)$

$T(p, n) = T(p, n - p) + T(p-1, n)$,

$n > 0, p > 0$

Complexidade: $O(n^2)$

```
Algoritmo (por coluna):

Partição():
    para p ← 0 até n incl.:
       T[p, 0] ← 1
    para j ← 1 até n incl.:
       T[0, j] ← 0
       para p ← 1 até n incl.:
          se (j  ≥ p):
             T[p, j] ← T[p, j-p] + T[p-1, j]
          senão:
             T[p, j] ← T[p-1, j]
```

Partição de Inteiros: n = 7

	0	1	2	3	4	5	6	7
0	1	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1
2	1	1	2	2	3	3	4	4
3	1	1	2	3	4	5	7	8
4	1	1	2	3	5	6	9	11
5	1	1	2	3	5	7	10	13
6	1	1	2	3	5	7	11	14
7	1	1	2	3	5	7	11	15

Partição de Inteiros

Conferindo a solução para T(7,7)

7 = 1+1+1+1+1+1+1

7 = 1+1+1+1+1+2

7 = 1+1+1+2+2

7 = 1+2+2+2

7 = 1+1+1+1+3

7 = 1+1+2+3

7 = 2+2+3

7 = 1+3+3

7 = 1+1+1+4

7 = 3+4

7 = 1+2+4

7 = 1+1+5

7 = 2+5

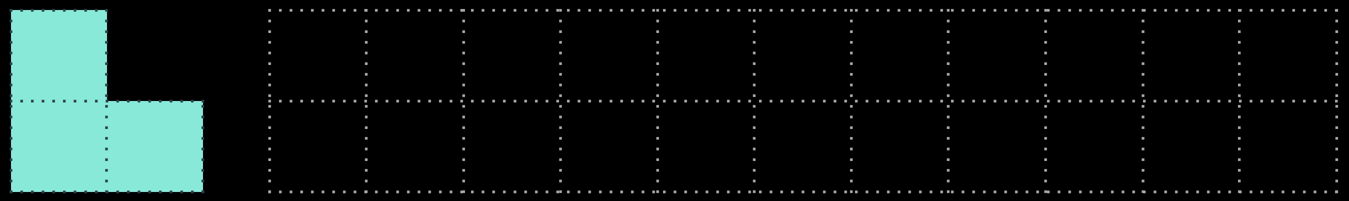
7 = 1+6

7 = 7

Exercício recomendado

Escrever uma recorrência para o seguinte problema:

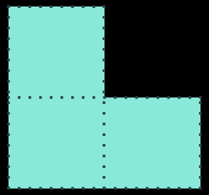
Dado um conjunto de azulejos do tipo trominos 2 x 2, com 1 canto cortado, quer-se saber de quantas maneiras distintas pode-se azulejar uma parede de dimensões 2 x n, com esses azulejos.

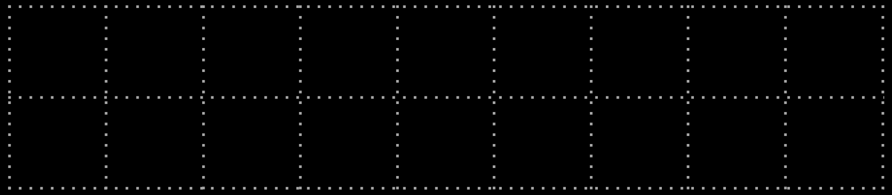


Exercício recomendado

Escrever uma recorrência para o seguinte problema:

Dado um conjunto de azulejos do tipo trominos 2 x 2, com 1 canto cortado, quer-se saber de quantas maneiras distintas pode-se azulejar uma parede de dimensões 2 x n, com esses azulejos.





Solução:

T(n) = número de maneiras distintas para o azulejamento.

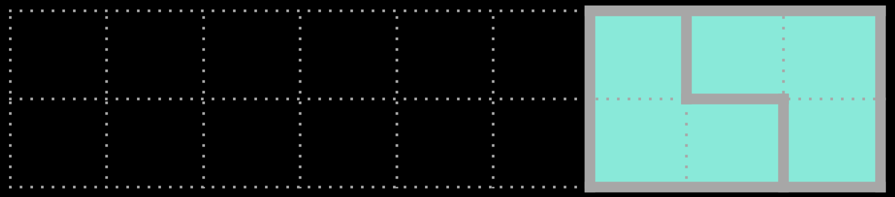
T(1) = 0

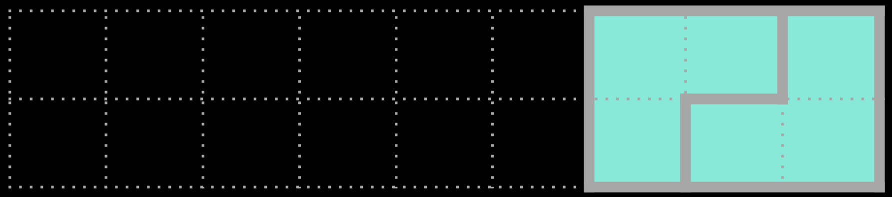
T(2) = 0

T(3) = 2

T(n) = ?

T(n) = 2T(n-3)





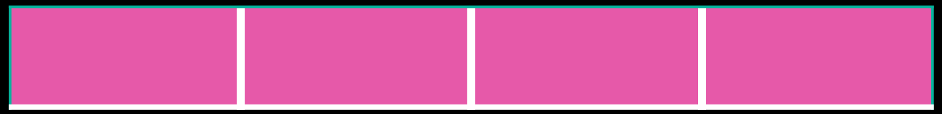
Programação Dinâmica

Corte do bastão

Corte do bastão

Dado um bastão de madeira de comprimento n e uma tabela de preços (venda) de cortes de n.

Objetivo: Determinar o valor máximo obtido cortando o bastão e vendendo os pedaços (cortes) ou o bastão inteiro.

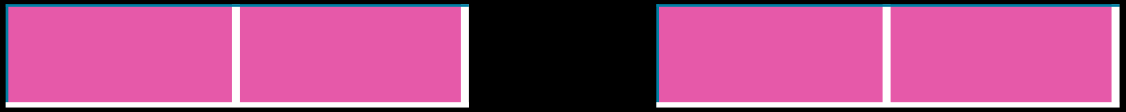


Tamanho (corte)	Preço
1	R$ 1,00
2	R$ 5,00
3	R$ 8,00
4	R$ 9,00

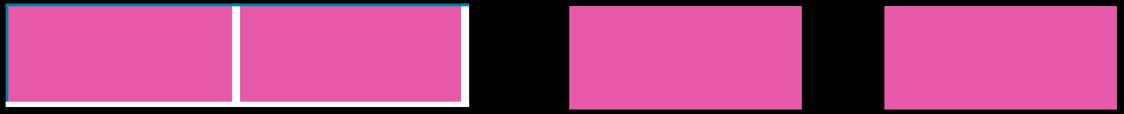
Corte do bastão

Algumas possibilidades

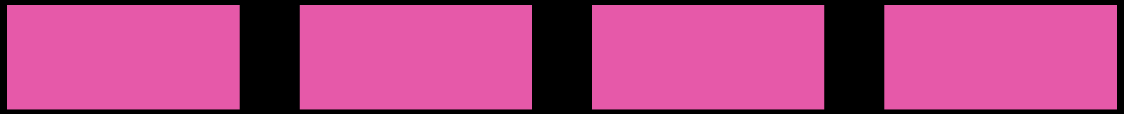
8 + 1 = 9



5 + 5 = 10



5 + 1 + 1 = 7



1 + 1 + 1 + 1 = 4

Tamanho (corte)	Preço
1	R$ 1,00
2	R$ 5,00
3	R$ 8,00
4	R$ 9,00

Corte do bastão

Tentando encontrar a recorrência do problema:

- Nosso objetivo é maximizar o valor obtido de um bastão de tamanho n, vamos considerar que isso seja o resultado da função rod(n)
- Se fizermos um corte de tamanho i nesse bastão, vamos obter um bastão de tamanho i e um novo bastão de tamanho n-i

Corte do bastão

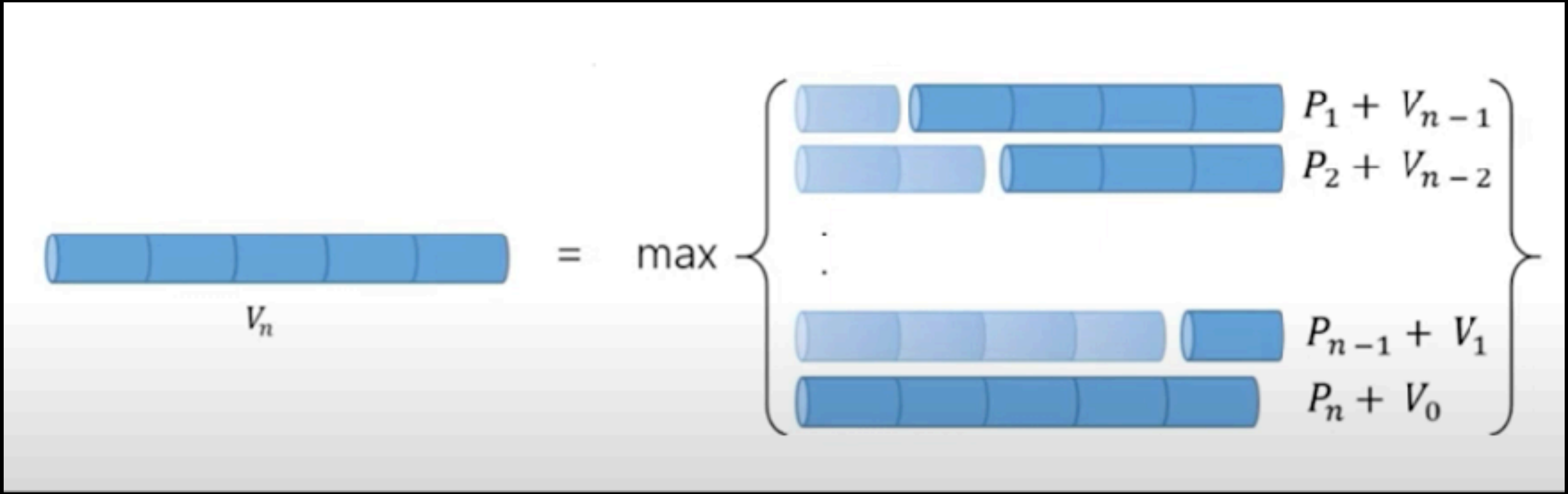
Vamos considerar que não podemos mais fazer cortes no bastão de tamanho i, apenas no bastão de tamanho n-i.

Nesse caso, a solução seria $p[i] + bastao(n - 1)$

A base da nossa solução será fazer isso para todos os possíveis cortes i, então podemos generalizar o problema da seguinte forma:

$$bastao(n) = \begin{cases} 0, \ \ se\ n = 0 \\ \max\left\{p_i + bastao(n - i)\right\}, i = 1, \ldots, n \end{cases}$$

Corte do bastão



Corte do bastão

Implementação Bottom-up

```
int cortebastao(int p[], int n){
    int bastao[n+1];
    bastao[0] = 0;
    for (int i = 1;  i <= n; i++){
       int max_val = -INF;
       for (int j = 1; j < i; j++)
         max_val = max(max_val, p[j] +
                       bastao[i-j]);
       bastao[i] = max_val;
    return bastao[n];
}
```

Corte do bastão

E na abordagem Top-down?
Como fica?

Exercício recomendado

Escrever uma recorrência para determinar de quantas maneiras diferentes para um atleta pode subir uma escada com n degraus, sendo que ele consegue dar pulos até de tamanho k.

Exercício recomendado

Escrever uma recorrência para determinar $T(n)$ = número de maneiras diferentes para um atleta pode subir uma escada com n degraus, sendo que ele consegue dar pulos até de tamanho k.

T(n) = 0, se n < 0

T(0) = 1, se n = 0

T(n) = ∑ T(n-i), 1 ≤ i ≤ n, se n ≤ k

T(n) = ∑ T(n-i), 1 ≤ i ≤ k, se n > k

Exemplo:

n=8, k=4

	0	1	2	3	4	5	6	7	8
T(n)	**1**	**1**	**2**	**4**	**8**	**15**	**29**	**56**	**108**

Programação Dinâmica

Maior Subsequência Crescente

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:

$X = \{A, B, C, B, D, C, B\}$

$X' = \{A, C, D, B\}$

Maior subsequência crescente (LIS)

Maior subsequência crescente: dado uma sequência de números, determinar a maior subsequência de valores crescentes

0	1	2	3	4	5	6	7
6	2	5	1	7	4	8	3

0	1	2	3	4	5	6	7
6	2	5	1	7	4	8	3

Maior subsequência crescente (LIS)

Inicialmente, vamos tentar verificar se este é um problema de PD, definindo a relação de recorrência mais intuitiva possível, sem ainda nos preocupar com a eficiência da solução.

Se pensarmos um pouco, não é tão difícil perceber que a subsequência máxima de um vetor $v[0 \ldots n-1]$ pode ser definida a partir das subsequência máximas dos vetores $v[0 \ldots n-2]$, $v[0 \ldots n-3]$, $\ldots$

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	3, 4, 6 ou 1, 2, 6
5	3, 4, 6 ou 1, 2, 6 ou 1, 2, 4
6	3, 4, 6 ou 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 subsequência crescente (LIS)

Definição dos estados

No passo anterior, concluímos que podemos determinar a subsequência máxima do vetor $v[0 \dots n-1]$ a partir das subsequência máximas dos vetores $v[0 \dots n-2]$, $v[0 \dots n-3]$, $\dots$

A partir disso, parece interessante definir o estado do nosso problema como o índice em que acaba nosso vetor

Subsequência máxima termina na posição i: lis(i)

Subsequência máxima do vetor inteiro: max(lis(i)), 0 <= 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 sequência 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 $a[j] \; < \; a[i]$, então $a[i]$ pode ser inserido na $lis(j)$, gerando uma subsequência de tamanho $lis(j) + 1$

$$lis(\mathrm{i}) = \begin{cases} 1, \, se \; i = 0 \\ 1 + \max\left\{ lis\left(j\right) \right\}, \; para \; \forall j \,|\, 0 \leq j < i \; e \; a_j < a_i \end{cases}$$

Maior subsequência crescente (LIS)

Implementação Top-down

```
Memo[] = {1, -1, -1, -1, ...}

int lis(int i){
   //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], listMax = 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);
      }
      listMax = max(listMax, memo[i]);
    }
    return listMax;
}
```

Complexidade:
$O(n^2)$

Programação Dinâmica

Maior Subsequência Comum

Maior subsequência comum (LCS)

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

Exemplo:

$X = \{A, B, C, B, D, A, B\}$

$Y = \{B, D, C, A, B, A\}$

$Z = \{B, C, B, A\}$

Maior subsequência comum (LCS)

Como dito anteriormente, uma subsequência de X é uma subsequência 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 subsequência para que elas se tornem iguais (ou, de forma equivalente, maximizar o número de elementos inseridos).

Maior subsequência comum (LCS)

Teorema: Seja $Z[1\ldots k]$ uma LCS de $X[1\ldots m]$ e $Y[1\ldots n]$

Se $x_m = y_n$ então $z_k = y_n = x_m$ e $Z[1\ldots k-1]$ é uma LCS de $X[1\ldots m-1]$ e $Y[1\ldots n-1]$

Se $x_m \neq y_n$ então $z_k \neq x_m$ e $Z[1\ldots k]$ é uma LCS de $X[1\ldots m-1]$ e $Y[1\ldots n]$

Se $x_m \neq y_n$ então $z_k \neq y_n$ e $Z[1\ldots k]$ é uma LCS de $X[1\ldots m]$ e $Y[1\ldots n-1]$

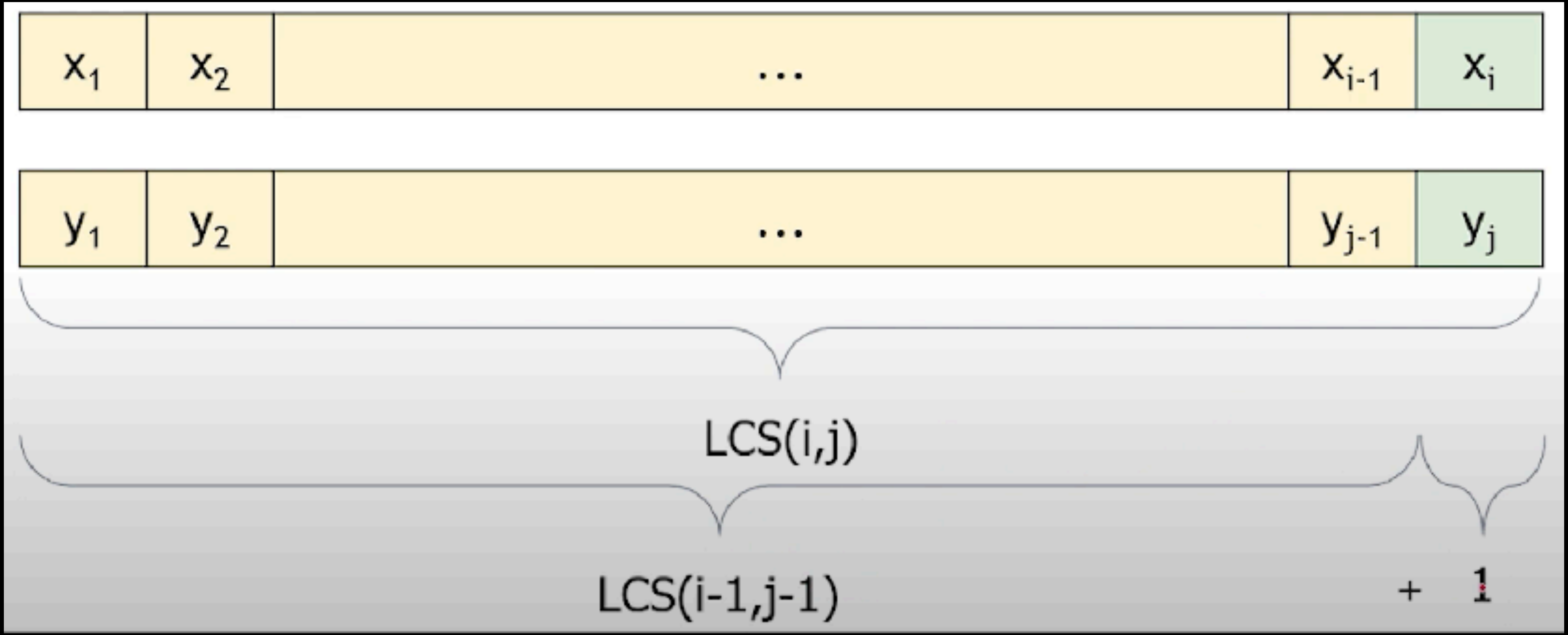
Esse teorema mostra que este problema atende a propriedade da subestrutura ótima.

Maior subsequência comum (LCS)

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

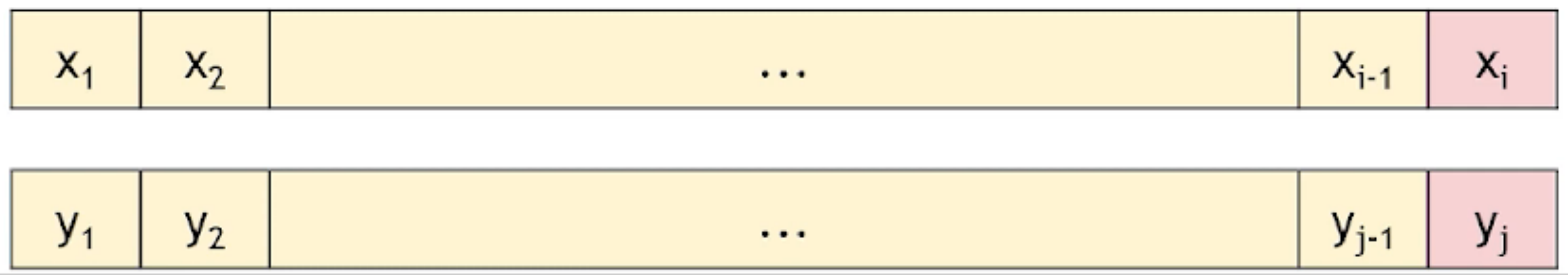
Maior subsequência comum (LCS)

Se $x_i = y_j$

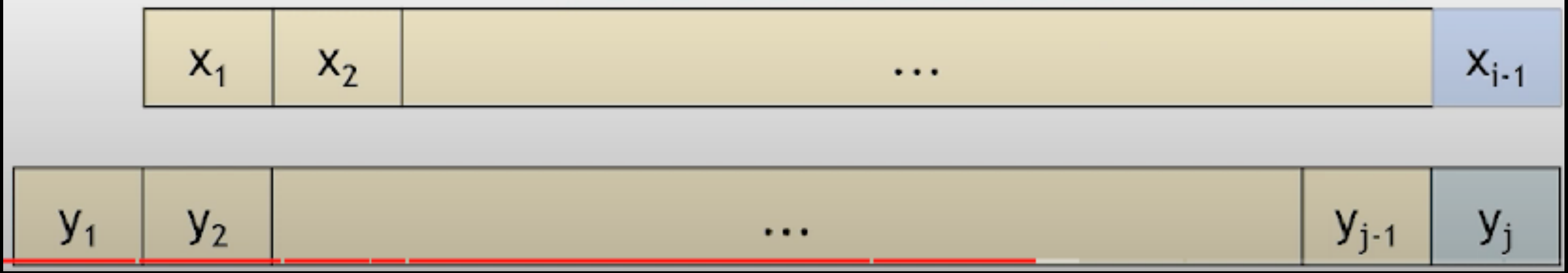


Maior subsequência comum (LCS)

Se $x_i != y_j$

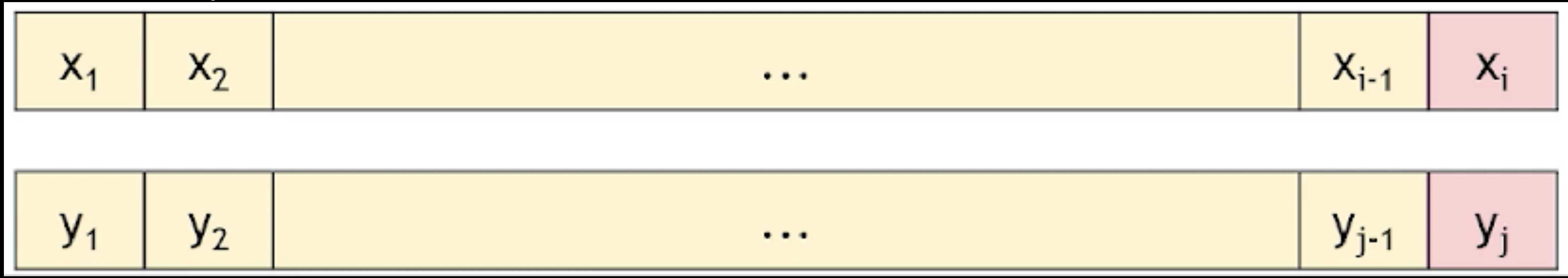


Opção 1: retirar $x_i => LCS(i-1, j)$



Maior subsequência comum (LCS)

Se $x_i ! = y_j$



Opção 2: retirar $y_i = > LCS(i, j-1)$

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

Programação Dinâmica

Problema da Mochila

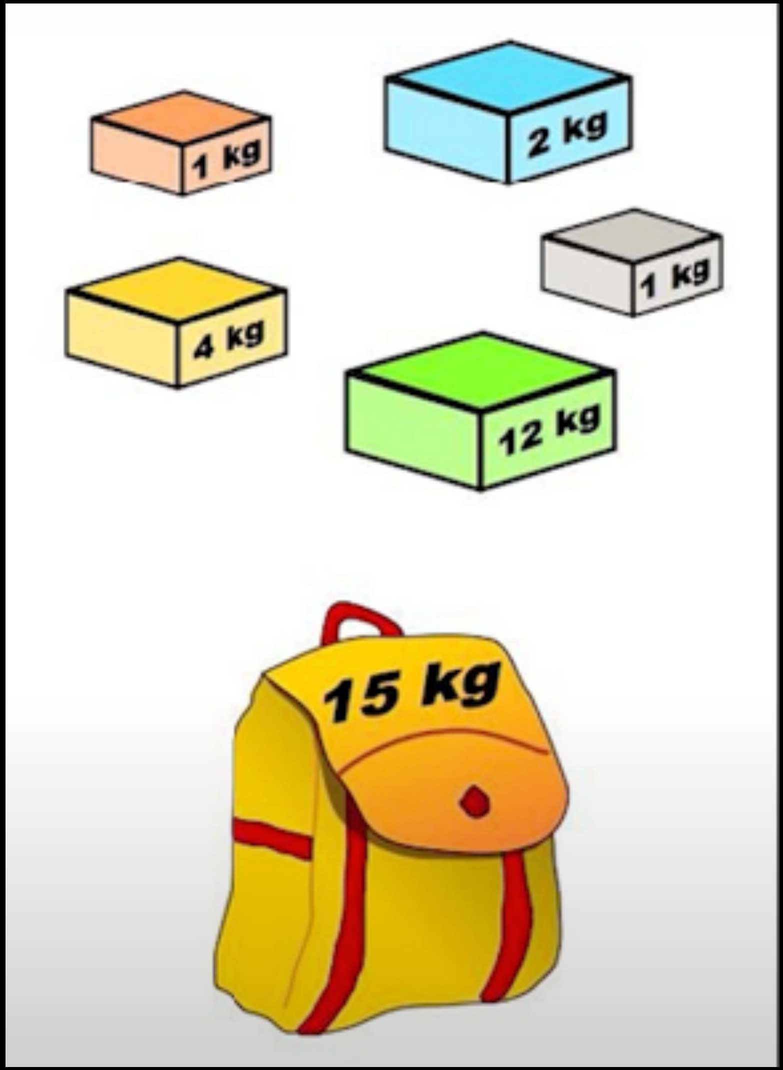
Problema da Mochila

Problema:

- Um 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 transportado que não ultrapassa o limite de peso 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

- Caso contrário, há duas opções: incluir ou não incluir o item (considerando o problema da mochila binária, onde não há repetição de itens)

Queremos maximizar o valor total carregado sem ultrapassar a capacidade da mochila

Problema da Mochila

w = capacidade disponível

i = item atual

$$f(w,i) = \begin{cases} 0,\ se\ w = 0\ ou\ i = 0 \\ \max\left\{ \underbrace{f(w, i-1)}_{\text{Não adiciona item}}, \underbrace{v_i + f\left(w - w_i, i-1\right)}_{\text{Adiciona item}} \right\},\ caso\ contrário \end{cases}$$

Problema da Mochila

Implementação Top-down

```
int mochila(int w, int n){
  if(memo[w][n] != -1)
    return memo[w][n];
  if (w ==0 || n==0)
    return memo[w][n] = 0;
  if (peso[n-1] > w)
    \\Não consigo adicionar
    return memo[w][n] = mochila(w,n-1);
  return memo[w][n] = max(mochila(w, n-1,
            valor[n-1] +
            mochila(w - peso[n-1], n-1));
}
```

Problema da Mochila

Implementação 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(peso[i-1] > j)
      dp[i][j] = dp[i-1][j];
    else
      dp[i][j] = max[dp[i-1][j], dp[i-1][j-
                 peso[i-1]] + valor[i-1];
  }
}
```

Problema da Mochila com Repetição

Variação comum do Problema da Mochila

- Neste caso, podemos considerar que temos uma quantidade ilimitada de de cada item.
- Um mesmo item pode ser colocado mais de uma vez dentro da mochila.

Problema da Mochila com Repetição

A ideia da solução não mudará muito, sendo até mais simples (de certa forma).

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 $(v[j] + dp[i - w[j]])$.

$$f(i) = \begin{cases} 0,\ se\ i = 0 \\ \max\left\{ v[j] + f\left(i - w[j]\right) \right\},\ \forall j \mid w[j] \leq i \end{cases}$$

Capacidade da mochila

Maior valor que pode ser obtido com cada item j

Problema da Mochila com Repetição

Implementação Bottom-up

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

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

Programação Dinâmica

Distância de Edição

Distância de Edição

Dados dois strings A e B, quer-se determinar a menor sequência de operações para transformar A em B.

Os tipos de operação são:

- -inserção de um caracter
- -deleção de um caracter
- -substituição de um caracter

Exemplo:

ERRO transforma-se em ACERTO mediante 3 operações:

-inserção do A:	AERRO
-inserção do C:	ACERRO
-substituição do R:	ACERTO

Distância de Edição - Formulação recursiva

Dados os substrings A_i (primeiros i caracteres) e B_j,

$D(i, j)$ = distância de edição entre A_i e B_j =

$D(i-1, j-1)$, se $a_i = b_j$

$\min(D(i-1, j),\ D(i, j-1),\ D(i-1, j-1)) + 1$, se $a_i \neq b_j$

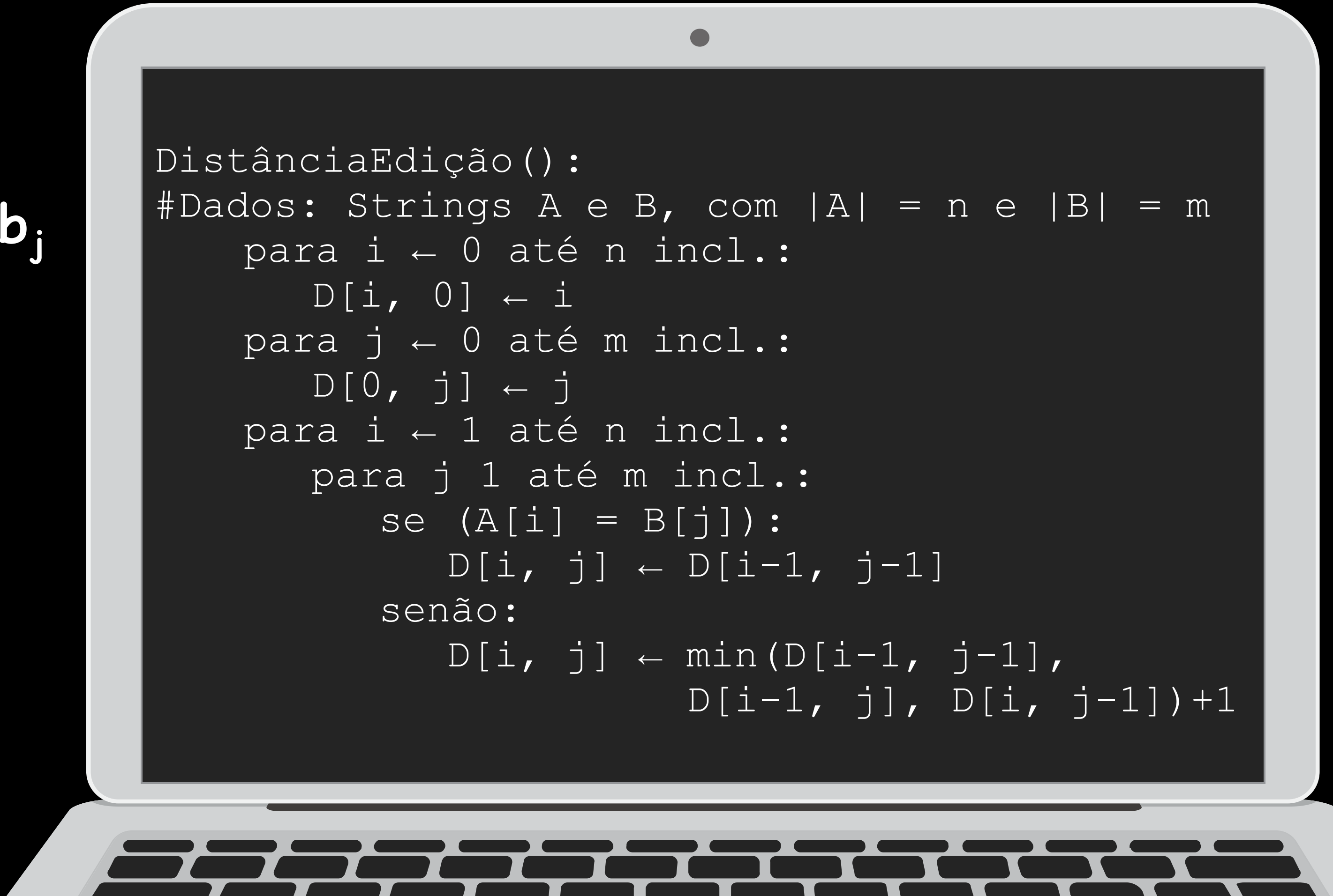
Deleção de a_i Inserção de b_j Substituição de a_i por b_j

Distância de Edição

Complexidade: O(n.m)

$$D(i, j) = D(i-1, j-1) = 0, \text{ se } a_i = b_j$$

$$\min(D(i-1, j), D(i, j-1), D(i-1, j-1)) + 1, \text{ se } a_i \neq b_j$$



Dúvidas?

lucila.bento [at] ime.uerj.br