MC458 — Projeto e Análise de Algoritmos I

C.C. de Souza   C.N. da Silva   O. Lee

Antes de mais nada...

- Uma versão anterior deste conjunto de slides foi preparada por Cid Carvalho de Souza e Cândida Nunes da Silva para uma instância anterior desta disciplina.
- O que vocês tem em mãos é uma versão modificada preparada para atender a meus gostos.
- Nunca é demais enfatizar que o material é apenas um guia e não deve ser usado como única fonte de estudo. Para isso consultem a bibliografia (em especial o CLR ou CLRS).

Orlando Lee

Agradecimentos (Cid e Cândida)

- Várias pessoas contribuíram direta ou indiretamente com a preparação deste material.
- Algumas destas pessoas cederam gentilmente seus arquivos digitais enquanto outras cederam gentilmente o seu tempo fazendo correções e dando sugestões.
- Uma lista destes “colaboradores” (em ordem alfabética) é dada abaixo:
  - Célia Picinin de Mello
  - José Coelho de Pina
  - Orlando Lee
  - Paulo Feofiloff
  - Pedro Rezende
  - Ricardo Dahab
  - Zanoni Dias

Programação Dinâmica

Programação dinâmica

- Tipicamente o paradigma de programação dinâmica se aplica a problemas de otimização combinatória.

Programação dinâmica

- Tipicamente o paradigma de programação dinâmica se aplica a problemas de otimização combinatória.
- Informalmente, em um problema de otimização combinatória temos um conjunto finito de soluções (viáveis) e queremos encontrar entre elas, uma que minimize (ou maximize) uma certa função objetivo. Tal solução é chamada de solução ótima.

Programação dinâmica

- Tipicamente o paradigma de programação dinâmica se aplica a problemas de otimização combinatória.
- Informalmente, em um problema de otimização combinatória temos um conjunto finito de soluções (viáveis) e queremos encontrar entre elas, uma que minimize (ou maximize) uma certa função objetivo. Tal solução é chamada de solução ótima.
- A dificuldade do problema é que o conjunto de soluções (viáveis) em geral é muito grande. Isto torna inviável a ideia de testar cada possível solução. O que se deseja é encontrar uma solução ótima, sem examinar todas as soluções.

Um exemplo

Problema da mochila: seja $I = \{1, 2, \ldots, n\}$ um conjunto de itens. Cada item i tem um peso w_i e um valor c_i. Suponha que temos uma mochila com capacidade W (peso máximo que suporta).

Um exemplo

Problema da mochila: seja $I = \{1, 2, \ldots, n\}$ um conjunto de itens. Cada item i tem um peso w_i e um valor c_i. Suponha que temos uma mochila com capacidade W (peso máximo que suporta).

O objetivo é escolher um subconjunto $S \subseteq I$ de itens tais que

- a soma dos pesos dos itens em S não ultrapassa W (cabe na mochila) e
- a soma dos valores dos itens em S é máximo.

Um exemplo

Problema da mochila: seja $I = \{1, 2, \ldots, n\}$ um conjunto de itens. Cada item i tem um peso w_i e um valor c_i. Suponha que temos uma mochila com capacidade W (peso máximo que suporta).

O objetivo é escolher um subconjunto $S \subseteq I$ de itens tais que

- a soma dos pesos dos itens em S não ultrapassa W (cabe na mochila) e
- a soma dos valores dos itens em S é máximo.

Note que as soluções viáveis do problema são os subconjuntos de I que satisfazem (a). Este conjunto pode ser potencialmente grande, da ordem de $\Omega(2^n)$.

Programação dinâmica

- O paradigma de programação dinâmica pode ser aplicado a certos problemas de otimização combinatória para obter algoritmos eficientes.

Programação dinâmica

- O paradigma de programação dinâmica pode ser aplicado a certos problemas de otimização combinatória para obter algoritmos eficientes.
- No projeto de um algoritmo de programação dinâmica seguimos os seguintes passos:

Programação dinâmica

- O paradigma de programação dinâmica pode ser aplicado a certos problemas de otimização combinatória para obter algoritmos eficientes.
- No projeto de um algoritmo de programação dinâmica seguimos os seguintes passos:
  1. Caracterizamos a estrutura de uma solução ótima.

Programação dinâmica

- O paradigma de programação dinâmica pode ser aplicado a certos problemas de otimização combinatória para obter algoritmos eficientes.
- No projeto de um algoritmo de programação dinâmica seguimos os seguintes passos:
  1. Caracterizamos a estrutura de uma solução ótima.
  2. Recursivamente definimos o valor de uma solução ótima.

Programação dinâmica

- O paradigma de programação dinâmica pode ser aplicado a certos problemas de otimização combinatória para obter algoritmos eficientes.
- No projeto de um algoritmo de programação dinâmica seguimos os seguintes passos:
  1. Caracterizamos a estrutura de uma solução ótima.
  2. Recursivamente definimos o valor de uma solução ótima.
  3. Computamos o valor de uma solução ótima, em geral, de modo bottom-up.

Programação dinâmica

- O paradigma de programação dinâmica pode ser aplicado a certos problemas de otimização combinatória para obter algoritmos eficientes.
- No projeto de um algoritmo de programação dinâmica seguimos os seguintes passos:
    1. Caracterizamos a estrutura de uma solução ótima.
    2. Recursivamente definimos o valor de uma solução ótima.
    3. Computamos o valor de uma solução ótima, em geral, de modo bottom-up.
    4. Construímos uma solução ótima, a partir da informação computada.

Programação dinâmica

- O paradigma de programação dinâmica pode ser aplicado a certos problemas de otimização combinatória para obter algoritmos eficientes.
- No projeto de um algoritmo de programação dinâmica seguimos os seguintes passos:
  1. Caracterizamos a estrutura de uma solução ótima.
  2. Recursivamente definimos o valor de uma solução ótima.
  3. Computamos o valor de uma solução ótima, em geral, de modo bottom-up.
  4. Construímos uma solução ótima, a partir da informação computada.
- Os passos 1-3 formam a base de uma solução por programação dinâmica.

Problema do corte de barra (Rod Cutting)

- A Cortadora de Pedaços Inteiros (CPI) é uma empresa que se especializou na venda de barras cilíndricas de aço.

Problema do corte de barra (Rod Cutting)

- A Cortadora de Pedaços Inteiros (CPI) é uma empresa que se especializou na venda de barras cilíndricas de aço.
- A CPI compra longas barras de aço de uma fornecedora (*Phantom Enterprises*) e então corta cada barra em barras mais curtas, para então vendê-las. Cada corte é grátis.
  A CPI quer saber a melhor maneira de cortar uma barra.

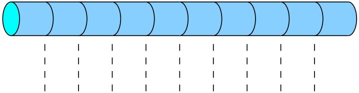
Problema do corte de barra (Rod Cutting)

- A Cortadora de Pedaços Inteiros (CPI) é uma empresa que se especializou na venda de barras cilíndricas de aço.
- A CPI compra longas barras de aço de uma fornecedora (*Phantom Enterprises*) e então corta cada barra em barras mais curtas, para então vendê-las. Cada corte é grátis.
  A CPI quer saber a melhor maneira de cortar uma barra.
- Os comprimentos (em metros) das barras são inteiros. A CPI cobra um preço p_i por uma barra de i metros, $i = 1, 2, \ldots$.

Exemplo:

i	1	2	3	4	5	6	7	8	9	10
p_i	1	5	8	9	10	17	17	20	24	30

Problema do corte de barra

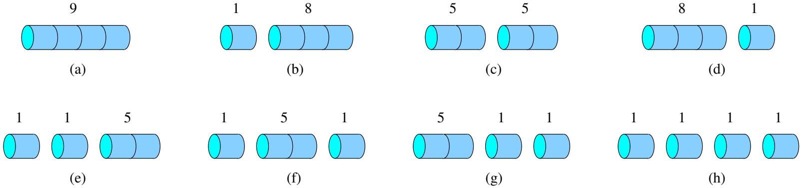


compr. i	1	2	3	4	5	6	7	8	9	10
preço p_i	1	5	8	9	10	17	17	20	24	30

Dada uma barra de n metros e uma tabela de preços p_i, $i = 1, 2, \ldots, n$, a CPI quer maximizar o lucro que pode ser obtido cortando-se a barra e vendendo os pedaços.

Exemplo de corte de barra para $n = 4$

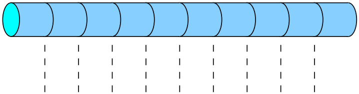
Há 2^3 maneiras distintas de cortar uma barra de comprimento 4.



compr. i	1	2	3	4
preço p_i	1	5	8	9

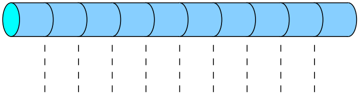
No exemplo acima, a solução ótima é dada pelo corte (c) com lucro 10.

Problema do corte de barra



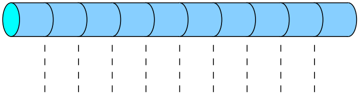
- De modo geral, para uma barra de comprimento n, há $n-1$ possíveis cortes.

Problema do corte de barra



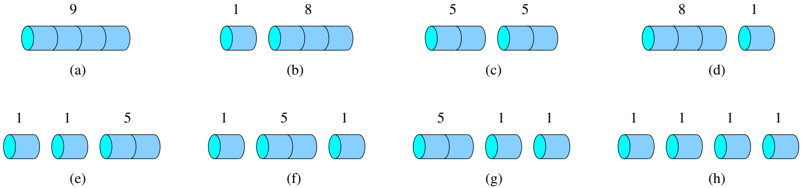
- De modo geral, para uma barra de comprimento n, há $n - 1$ possíveis cortes.
- Assim, o número de possíveis maneiras de cortar uma barra de comprimento n é 2^{n-1}.

Problema do corte de barra



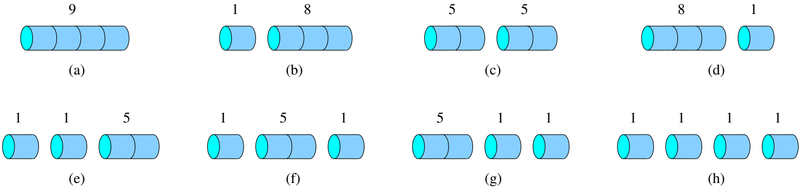
- De modo geral, para uma barra de comprimento n, há $n - 1$ possíveis cortes.
- Assim, o número de possíveis maneiras de cortar uma barra de comprimento n é 2^{n-1}.
- Assim, é impraticável enumerar todas elas para determinar uma solução ótima!

Problema do corte de barra



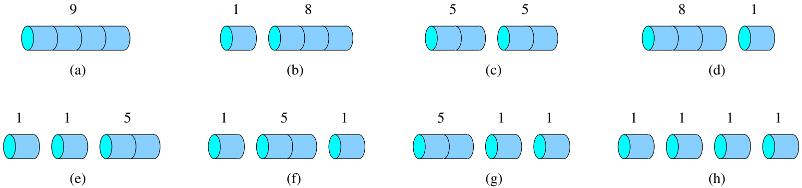
- Poderíamos considerar apenas cortes em ordem não-decrescente de comprimento. Deste modo, haveria menos maneiras de cortar uma barra de comprimento n.

Problema do corte de barra



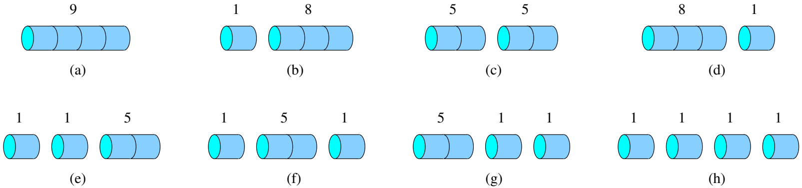
- Poderíamos considerar apenas cortes em ordem não-decrescente de comprimento. Deste modo, haveria menos maneiras de cortar uma barra de comprimento n.
- Para $n = 4$ as maneiras válidas são (a), (b), (c), (e) e (h).

Problema do corte de barra



- Poderíamos considerar apenas cortes em ordem não-decrescente de comprimento. Deste modo, haveria menos maneiras de cortar uma barra de comprimento n.
- Para $n = 4$ as maneiras válidas são (a), (b), (c), (e) e (h).
- O número de maneiras de cortar uma barra desta forma é dada pela função de partição $\approx e^{\pi\sqrt{2n/3}}/4n\sqrt{3}$ que é menor que 2^{n-1} mas ainda muito maior que qualquer polinômio em n.

Problema do corte de barra



- Poderíamos considerar apenas cortes em ordem não-decrescente de comprimento. Deste modo, haveria menos maneiras de cortar uma barra de comprimento n.
- Para $n = 4$ as maneiras válidas são (a), (b), (c), (e) e (h).
- O número de maneiras de cortar uma barra desta forma é dada pela função de partição $\approx e^{\pi\sqrt{2n/3}}/4n\sqrt{3}$ que é menor que 2^{n-1} mas ainda muito maior que qualquer polinômio em n.
- Não seguiremos com esta linha de raciocínio.

Problema do corte de barra

- Usaremos uma notação aditiva para representar uma decomposição (maneira de cortar) de uma barra. Por exemplo, $7 = 2 + 2 + 3$ indica que uma barra de comprimento 7 foi cortada em três pedaços: duas barras de comprimento 2 e uma barra de comprimento 3.

Problema do corte de barra

- Usaremos uma notação aditiva para representar uma decomposição (maneira de cortar) de uma barra. Por exemplo, $7 = 2 + 2 + 3$ indica que uma barra de comprimento 7 foi cortada em três pedaços: duas barras de comprimento 2 e uma barra de comprimento 3.
- Suponha que uma solução ótima corta uma barra de comprimento n em k pedaços, para algum k, $1 \leq k \leq n$, da seguinte forma:

$$n = i_1 + i_2 + \cdots + i_k.$$

Problema do corte de barra

- Usaremos uma notação aditiva para representar uma decomposição (maneira de cortar) de uma barra. Por exemplo, $7 = 2 + 2 + 3$ indica que uma barra de comprimento 7 foi cortada em três pedaços: duas barras de comprimento 2 e uma barra de comprimento 3.
- Suponha que uma solução ótima corta uma barra de comprimento n em k pedaços, para algum k, $1 \leq k \leq n$, da seguinte forma:

$$n = i_1 + i_2 + \cdots + i_k.$$

- O lucro (máximo) r_n obtido com esta decomposição é:

$$r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}.$$

Problema do corte de barra

compr. i	1	2	3	4	5	6	7	8	9	10
preço p_i	1	5	8	9	10	17	17	20	24	30

Podemos verificar (por inspeção) que:

Problema do corte de barra

compr. i	1	2	3	4	5	6	7	8	9	10
preço p_i	1	5	8	9	10	17	17	20	24	30

Podemos verificar (por inspeção) que:

n	solução ótima	lucro máximo
1	$1 = 1$	$r_1 = p_1 = 1$
2	$2 = 2$	$r_2 = p_2 = 5$
3	$3 = 3$	$r_3 = p_3 = 8$
4	$4 = 2 + 2$	$r_4 = r_2 + r_2 = 10$
5	$5 = 2 + 3$	$r_5 = r_2 + r_3 = 13$
6	$6 = 6$	$r_6 = p_6 = 17$
7	$7 = 1 + 6$ ou $7 = 2 + 2 + 3$	$r_7 = r_1 + r_6 = r_2 + r_2 + r_3 = 18$
8	$8 = 2 + 6$	$r_8 = r_2 + r_6 = 22$
9	$9 = 3 + 6$	$r_9 = r_3 + r_6 = 25$
10	$10 = 10$	$r_{10} = 30$

Subestrutura ótima

Subestrutura ótima

- Podemos expressar o lucro máximo r_n em função de r_i, para $i = 1, 2, \ldots, n-1$, através da recorrência:

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_i + r_{n-i}, \ldots, r_{n-1} + r_1).$$

Subestrutura ótima

- Podemos expressar o lucro máximo r_n em função de r_i, para $i = 1, 2, \ldots, n-1$, através da recorrência:

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_i + r_{n-i}, \ldots, r_{n-1} + r_1).$$

- O primeiro argumento p_n representa a solução sem cortes. Os demais $n-1$ argumentos representam a solução em que o corte inicial divide a barra em duas barras de comprimento i e $n-i$, para $i = 1, 2, \ldots, n-1$, e então corta de modo ótimo as duas barras obtendo lucro $r_i + r_{n-i}$.

Subestrutura ótima

- Podemos expressar o lucro máximo r_n em função de r_i, para $i = 1, 2, \ldots, n-1$, através da recorrência:

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_i + r_{n-i}, \ldots, r_{n-1} + r_1).$$

- O primeiro argumento p_n representa a solução sem cortes. Os demais $n-1$ argumentos representam a solução em que o corte inicial divide a barra em duas barras de comprimento i e $n-i$, para $i = 1, 2, \ldots, n-1$, e então corta de modo ótimo as duas barras obtendo lucro $r_i + r_{n-i}$.
- Como a priori não sabemos qual é a melhor maneira de dividir, é necessário considerar todos os possíveis valores de i e escolher aquele que maximiza o lucro. Podemos também decidir por não fazer nenhum corte.

Subestrutura ótima

- Podemos expressar o lucro máximo r_n em função de r_i, para $i = 1, 2, \ldots, n-1$, através da recorrência:

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_i + r_{n-i}, \ldots, r_{n-1} + r_1).$$

Subestrutura ótima

- Podemos expressar o lucro máximo r_n em função de r_i, para $i = 1, 2, \ldots, n-1$, através da recorrência:

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_i + r_{n-i}, \ldots, r_{n-1} + r_1).$$

- Note que para resolver o problema original de tamanho n, temos que resolver problemas do mesmo tipo, mas de tamanho menor.

Subestrutura ótima

- Podemos expressar o lucro máximo r_n em função de r_i, para $i = 1, 2, \ldots, n-1$, através da recorrência:

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_i + r_{n-i}, \ldots, r_{n-1} + r_1).$$

- Note que para resolver o problema original de tamanho n, temos que resolver problemas do mesmo tipo, mas de tamanho menor.
- A solução ótima incorpora soluções de dois subproblemas relacionados, que maximizam o lucro em cada pedaço.

Subestrutura ótima

- Podemos expressar o lucro máximo r_n em função de r_i, para $i = 1, 2, \ldots, n-1$, através da recorrência:

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_i + r_{n-i}, \ldots, r_{n-1} + r_1).$$

- Note que para resolver o problema original de tamanho n, temos que resolver problemas do mesmo tipo, mas de tamanho menor.
- A solução ótima incorpora soluções de dois subproblemas relacionados, que maximizam o lucro em cada pedaço.
- Dizemos então que o problema do corte de barra tem subestrutura ótima: soluções ótimas contém soluções ótimas de subproblemas relacionados.

Subestrutura ótima: recorrência mais simples

- Há uma maneira mais simples de explorar a natureza recursiva do problema do corte de barra. Como a ordem dos cortes é irrelevante, podemos pensar que o corte inicial é aquele feito mais à esquerda.

Subestrutura ótima: recorrência mais simples

- Há uma maneira mais simples de explorar a natureza recursiva do problema do corte de barra. Como a ordem dos cortes é irrelevante, podemos pensar que o corte inicial é aquele feito mais à esquerda.
- Assim, uma decomposição consiste em cortar fora um pedaço de comprimento i e então resolver o subproblema para a barra de comprimento $n - i$ resultante.

Subestrutura ótima: recorrência mais simples

- Há uma maneira mais simples de explorar a natureza recursiva do problema do corte de barra. Como a ordem dos cortes é irrelevante, podemos pensar que o corte inicial é aquele feito mais à esquerda.
- Assim, uma decomposição consiste em cortar fora um pedaço de comprimento i e então resolver o subproblema para a barra de comprimento $n - i$ resultante.
- Deste modo, podemos pensar na solução sem cortes como aquela em que o primeiro pedaço tem comprimento $i = n$ e o subproblema resultante tem tamanho 0 e lucro máximo $r_0 = 0$.

Subestrutura ótima: recorrência mais simples

- Há uma maneira mais simples de explorar a natureza recursiva do problema do corte de barra. Como a ordem dos cortes é irrelevante, podemos pensar que o corte inicial é aquele feito mais à esquerda.
- Assim, uma decomposição consiste em cortar fora um pedaço de comprimento i e então resolver o subproblema para a barra de comprimento $n - i$ resultante.
- Deste modo, podemos pensar na solução sem cortes como aquela em que o primeiro pedaço tem comprimento $i = n$ e o subproblema resultante tem tamanho 0 e lucro máximo $r_0 = 0$.
- Obtemos então a recorrência mais simples:

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i}).$$

Implementação recursiva topdown

A partir da recorrência

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i}),$$

é trivial implementar um algoritmo recursivo para determinar r_n.

Implementação recursiva topdown

A partir da recorrência

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i}),$$

é trivial implementar um algoritmo recursivo para determinar r_n.

```
Corta-Barra(p, n)
    se n = 0
        então devolva 0
    q ← −∞
    para i = 1 até n faça
        q = max(q, p_i + Corta-Barra(p, n − i))
    devolva q
```

A corretude de Corta-Barra segue por indução em n e da fórmula de recorrência (Exercício).

Implementação recursiva topdown

```
CORTA-BARRA(p, n)
    se n = 0
        então devolva 0
    q ← −∞
    para i = 1 até n faça
        q = max(q, p_i + CORTA-BARRA(p, n − i))
    devolva q
```

$\text{CORTA-BARRA}(p, n)$
1. **se** $n = 0$
2. **então devolva** 0
3. $q \leftarrow -\infty$
4. **para** $i = 1$ **até** n **faça**
5. $q = \max(q, p_i + \text{CORTA-BARRA}(p, n - i))$
6. **devolva** q

Se você implementar CORTA-BARRA em sua linguagem de programação favorita e executá-lo em seu computador para instâncias de tamanho moderado, você notará que ele demora muito.

Implementação recursiva topdown

$\textsc{Corta-Barra}(p, n)$

```
se n = 0
    então devolva 0
q ← −∞
para i = 1 até n faça
    q = max(q, p_i + CORTA-BARRA(p, n − i))
devolva q
```

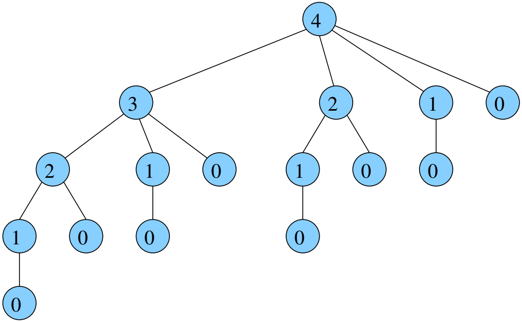
Se você implementar $\textsc{Corta-Barra}$ em sua linguagem de programação favorita e executá-lo em seu computador para instâncias de tamanho moderado, você notará que ele demora muito.

Para $n = 40$ provavelmente, seu programa demorará minutos (talvez mais que uma hora). Se você aumentar n de 1, perceberá que o tempo dobrará.

Corta-Barra(p, n)
1. **se** $n = 0$
2. **então devolva** 0
3. $q \leftarrow -\infty$
4. **para** $i = 1$ **até** n **faça**
5. $q = \max(q, p_i + \text{Corta-Barra}(p, n - i))$
6. **devolva** q

Por que Corta-Barra é tão ineficiente? O motivo é que Corta-Barra chama a si mesmo várias e várias vezes com os mesmos parâmetros. Ele resolve os subproblemas repetidas vezes.

Ávore de recursão de Corta-Barra



Cada nó com rótulo i representa um subproblema de tamanho i. As arestas indicam onde é feito o corte inicial. Um caminho da raiz até uma das 2^{n-1} folhas representa uma maneira de cortar a barra. A árvore de recursão tem 2^n nós e 2^{n-1} folhas. (!!)

Consumo de tempo de CORTA-BARRA

CORTA-BARRA(p, n)
```
se n = 0
    então devolva 0
q ← −∞
para i = 1 até n faça
    q = max(q, p_i + CORTA-BARRA(p, n − i))
devolva q
```

Seja $T(n)$ o número de chamadas feitas a CORTA-BARRA quando chamado inicialmente com segundo parâmetro igual a n.

Isto é igual ao número de nós em uma subárvore com uma raiz de rótulo n.

Consumo de tempo de CORTA-BARRA

CORTA-BARRA(p, n)

```
se n = 0
    então devolva 0
q ← −∞
para i = 1 até n faça
    q = max(q, p_i + CORTA-BARRA(p, n − i))
devolva q
```

Então $T(0) = 1$ e $T(n) = 1 + \sum_{j=0}^{n-1} T(j)$ para $n \geq 1$.

O 1 corresponde à chamada inicial e $T(j)$ corresponde à chamada CORTA-BARRA($p, n - i$) com $j = n - i$.

Consumo de tempo de CORTA-BARRA

CORTA-BARRA(p, n)
se $n = 0$
    então devolva 0
$q \leftarrow -\infty$
para $i \leftarrow 1$ **até** n **faça**
    $q \leftarrow \max(q, p_i + \text{CORTA-BARRA}(p, n - i))$
devolva q

Chuto que $T(n) = 2^n$. (método da substituição)

Consumo de tempo de Corta-Barra

```
Corta-Barra(p, n)
    se n = 0
        então devolva 0
    q ← −∞
    para i ← 1 até n faça
        q ← max(q, p_i + Corta-Barra(p, n − i))
    devolva q
```

Chuto que $T(n) = 2^n$. (método da substituição)

De fato, $T(0) = 1 = 2^0$ e

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) = 1 + \sum_{j=0}^{n-1} 2^j = 1 + (2^n - 1) = 2^n.$$ (Yeess!)

Soluções eficientes para corte de barra

- A implementação ingênua CORTA-BARRA é ineficiente pois resolve cada subproblema várias vezes.

Soluções eficientes para corte de barra

- A implementação ingênua Corta-Barra é ineficiente pois resolve cada subproblema várias vezes.
- Veremos dois métodos para converter Corta-Barra em um algoritmo eficiente.

Soluções eficientes para corte de barra

- A implementação ingênua Corta-Barra é ineficiente pois resolve cada subproblema várias vezes.
- Veremos dois métodos para converter Corta-Barra em um algoritmo eficiente.
- A ideia principal é garantir que cada subproblema seja resolvido apenas uma vez, armazenando sua solução. Se o subproblema aparecer de novo, podemos simplesmente olhar a solução em vez de resolver o subproblema.

Soluções eficientes para corte de barra

Soluções eficientes para corte de barra

- Trade-off de tempo-memória: economizamos tempo mas aumentamos o espaço (memória) usado.

Soluções eficientes para corte de barra

- Trade-off de tempo-memória: economizamos tempo mas aumentamos o espaço (memória) usado.
- Para que o método resultante seja eficiente, é necessário que o número de subproblemas seja polinomial no tamanho da entrada e que saibamos resolver cada subproblema.

Soluções eficientes para corte de barra

- Trade-off de tempo-memória: economizamos tempo mas aumentamos o espaço (memória) usado.
- Para que o método resultante seja eficiente, é necessário que o número de subproblemas seja polinomial no tamanho da entrada e que saibamos resolver cada subproblema.
- No problema de corte de barra o número de subproblemas é linear em n. Veremos que é necessário apenas memória adicional de $\Theta(n)$ para armazenar as soluções ótimas dos subproblemas.

Método 1: programação dinâmica (bottom-up)

Método 1: programação dinâmica (bottom-up)

- Conceitualmente a ideia é simples: resolvemos os subproblemas em ordem crescente do tamanho do problema, guardando suas soluções.

Método 1: programação dinâmica (bottom-up)

- Conceitualmente a ideia é simples: resolvemos os subproblemas em ordem crescente do tamanho do problema, guardando suas soluções.
- Isto permite resolver cada subproblema uma única vez.

Método 1: programação dinâmica (bottom-up)

- Conceitualmente a ideia é simples: resolvemos os subproblemas em ordem crescente do tamanho do problema, guardando suas soluções.
- Isto permite resolver cada subproblema uma única vez.
- A ordem em que os subproblemas devem ser resolvidos para isto acontecer, em geral, segue da fórmula de recorrência.

```
PD-CORTA-BARRA(p, n)
  r[0] ← 0
  para j ← 1 até n faça
      q ← −∞
      para i ← 1 até j faça
          q ← max(q, p[i] + r[j − i])
      r[j] ← q
  devolva r[n]
```

i	1	2	3	4	5	6	7	8	9	10
p_i	1	5	8	9	10	17	17	20	24	30

j	0	1	2	3	4	5	6	7	8	9	10
$r[j]$											

PD-CORTA-BARRA(p, n)

1. $r[0] \leftarrow 0$
2. **para** $j \leftarrow 1$ **até** n **faça**
3. $\quad q \leftarrow -\infty$
4. $\quad$ **para** $i \leftarrow 1$ **até** j **faça**
5. $\quad\quad q \leftarrow \max(q, p[i] + r[j-i])$
6. $\quad r[j] \leftarrow q$
7. **devolva** $r[n]$

i	1	2	3	4	5	6	7	8	9	10
p_i	1	5	8	9	10	17	17	20	24	30

j	0	1	2	3	4	5	6	7	8	9	10
$r[j]$	0										

Programação dinâmica (bottom-up)

PD-CORTA-BARRA(p, n)

1. $r[0] \leftarrow 0$
2. **para** $j \leftarrow 1$ **até** n **faça**
3. $\quad q \leftarrow -\infty$
4. $\quad$ **para** $i \leftarrow 1$ **até** j **faça**
5. $\quad\quad q \leftarrow \max(q, p[i] + r[j - i])$
6. $\quad r[j] \leftarrow q$
7. **devolva** $r[n]$

i	1	2	3	4	5	6	7	8	9	10
p_i	1	5	8	9	10	17	17	20	24	30

j	0	1	2	3	4	5	6	7	8	9	10
$r[j]$	0	1									

Programação dinâmica (bottom-up)

PD-CORTA-BARRA(p, n)
1. $r[0] \leftarrow 0$
2. **para** $j \leftarrow 1$ **até** n **faça**
3. $\quad q \leftarrow -\infty$
4. $\quad$ **para** $i \leftarrow 1$ **até** j **faça**
5. $\quad\quad q \leftarrow \max(q, p[i] + r[j-i])$
6. $\quad r[j] \leftarrow q$
7. **devolva** $r[n]$

i	1	2	3	4	5	6	7	8	9	10
p_i	1	5	8	9	10	17	17	20	24	30

j	0	1	2	3	4	5	6	7	8	9	10
$r[j]$	0	1	5								

PD-CORTA-BARRA(p, n)

1. $r[0] \leftarrow 0$
2. **para** $j \leftarrow 1$ **até** n **faça**
3. $\quad q \leftarrow -\infty$
4. $\quad$ **para** $i \leftarrow 1$ **até** j **faça**
5. $\quad\quad q \leftarrow \max(q, p[i] + r[j-i])$
6. $\quad r[j] \leftarrow q$
7. **devolva** $r[n]$

i	1	2	3	4	5	6	7	8	9	10
p_i	1	5	8	9	10	17	17	20	24	30

j	0	1	2	3	4	5	6	7	8	9	10
$r[j]$	0	1	5	8							

PD-CORTA-BARRA(p, n)

1. $r[0] \leftarrow 0$
2. **para** $j \leftarrow 1$ **até** n **faça**
3. $\quad q \leftarrow -\infty$
4. $\quad$ **para** $i \leftarrow 1$ **até** j **faça**
5. $\quad\quad q \leftarrow \max(q, p[i] + r[j-i])$
6. $\quad r[j] \leftarrow q$
7. **devolva** $r[n]$

i	1	2	3	4	5	6	7	8	9	10
p_i	1	5	8	9	10	17	17	20	24	30

j	0	1	2	3	4	5	6	7	8	9	10
$r[j]$	0	1	5	8	10						

PD-CORTA-BARRA(p, n)

1. $r[0] \leftarrow 0$
2. **para** $j \leftarrow 1$ **até** n **faça**
3. $\quad q \leftarrow -\infty$
4. $\quad$ **para** $i \leftarrow 1$ **até** j **faça**
5. $\quad\quad q \leftarrow \max(q, p[i] + r[j-i])$
6. $\quad r[j] \leftarrow q$
7. **devolva** $r[n]$

i	1	2	3	4	5	6	7	8	9	10
p_i	1	5	8	9	10	17	17	20	24	30

j	0	1	2	3	4	5	6	7	8	9	10
$r[j]$	0	1	5	8	10	13					

Programação dinâmica (bottom-up)

PD-CORTA-BARRA(p, n)

1. $r[0] \leftarrow 0$
2. **para** $j \leftarrow 1$ **até** n **faça**
3. $\quad q \leftarrow -\infty$
4. $\quad$ **para** $i \leftarrow 1$ **até** j **faça**
5. $\quad\quad q \leftarrow \max(q, p[i] + r[j-i])$
6. $\quad r[j] \leftarrow q$
7. **devolva** $r[n]$

i	1	2	3	4	5	6	7	8	9	10
p_i	1	5	8	9	10	17	17	20	24	30

j	0	1	2	3	4	5	6	7	8	9	10
$r[j]$	0	1	5	8	10	13	17				

Programação dinâmica (bottom-up)

PD-CORTA-BARRA(p, n)
1. $r[0] \leftarrow 0$
2. **para** $j \leftarrow 1$ **até** n **faça**
3. $\quad q \leftarrow -\infty$
4. $\quad$ **para** $i \leftarrow 1$ **até** j **faça**
5. $\quad\quad q \leftarrow \max(q, p[i] + r[j-i])$
6. $\quad r[j] \leftarrow q$
7. **devolva** $r[n]$

i	1	2	3	4	5	6	7	8	9	10
p_i	1	5	8	9	10	17	17	20	24	30

j	0	1	2	3	4	5	6	7	8	9	10
$r[j]$	0	1	5	8	10	13	17	18			

Programação dinâmica (bottom-up)

PD-CORTA-BARRA(p, n)
1. $r[0] \leftarrow 0$
2. **para** $j \leftarrow 1$ **até** n **faça**
3. $\quad q \leftarrow -\infty$
4. $\quad$ **para** $i \leftarrow 1$ **até** j **faça**
5. $\quad\quad q \leftarrow \max(q, p[i] + r[j-i])$
6. $\quad r[j] \leftarrow q$
7. **devolva** $r[n]$

i	1	2	3	4	5	6	7	8	9	10
p_i	1	5	8	9	10	17	17	20	24	30

j	0	1	2	3	4	5	6	7	8	9	10
$r[j]$	0	1	5	8	10	13	17	18	22		

PD-CORTA-BARRA(p, n)

1. $r[0] \leftarrow 0$
2. **para** $j \leftarrow 1$ **até** n **faça**
3. $\quad q \leftarrow -\infty$
4. $\quad$ **para** $i \leftarrow 1$ **até** j **faça**
5. $\quad\quad q \leftarrow \max(q, p[i] + r[j-i])$
6. $\quad r[j] \leftarrow q$
7. **devolva** $r[n]$

i	1	2	3	4	5	6	7	8	9	10
p_i	1	5	8	9	10	17	17	20	24	30

j	0	1	2	3	4	5	6	7	8	9	10
$r[j]$	0	1	5	8	10	13	17	18	22	25	

Programação dinâmica (bottom-up)

PD-CORTA-BARRA(p, n)

```
r[0] ← 0
para j ← 1 até n faça
    q ← −∞
    para i ← 1 até j faça
        q ← max(q, p[i] + r[j − i])
    r[j] ← q
devolva r[n]
```

i	1	2	3	4	5	6	7	8	9	10
p_i	1	5	8	9	10	17	17	20	24	30

j	0	1	2	3	4	5	6	7	8	9	10
$r[j]$	0	1	5	8	10	13	17	18	22	25	30

```
PD-Corta-Barra(p, n)
r[0] ← 0
para j ← 1 até n faça
    q ← −∞
    para i ← 1 até j faça
        q ← max(q, p[i] + r[j − i])
    r[j] ← q
devolva r[n]
```

Programação dinâmica (bottom-up)

PD-CORTA-BARRA(p, n)

1. $r[0] \leftarrow 0$
2. **para** $j \leftarrow 1$ **até** n **faça**
3. $\quad q \leftarrow -\infty$
4. $\quad$ **para** $i \leftarrow 1$ **até** j **faça**
5. $\quad\quad q \leftarrow \max(q, p[i] + r[j-i])$
6. $\quad r[j] \leftarrow q$
7. **devolva** $r[n]$

As linhas 3-6 computam r_j, guardando em $r[j]$. Pela fórmula de recorrência, r_j depende apenas dos r_i, para $i = 1, 2 \ldots, j-1$, que foram calculados nas iterações anteriores.

Invariante: no início de cada iteração da linha 2, $r[i] = r_i$ para $i = 0, 1, 2, \ldots, j-1$.

Programação dinâmica (bottom-up)

$\text{PD-CORTA-BARRA}(p, n)$

$r[0] \leftarrow 0$	?
para $j \leftarrow 1$ **até** n **faça**	?
$\quad q \leftarrow -\infty$	?
$\quad$ **para** $i \leftarrow 1$ **até** j **faça**	?
$\quad\quad q \leftarrow \max(q, p[i] + r[j - i])$	?
$\quad r[j] \leftarrow q$	?
devolva $r[n]$	?

Complexidade de PD-CORTA-BARRA: ???

PD-CORTA-BARRA(p, n)

$r[0] \leftarrow 0$	$\Theta(1)$
para $j \leftarrow 1$ **até** n **faça**	$\Theta(n)$
$\quad q \leftarrow -\infty$	$\Theta(n)$
$\quad$ **para** $i \leftarrow 1$ **até** j **faça**	$\sum_{j=1}^{n} j$
$\quad\quad q \leftarrow \max(q, p[i] + r[j-i])$	$\sum_{j=1}^{n} j$
$\quad r[j] \leftarrow q$	$\Theta(n)$
devolva $r[n]$	$\Theta(1)$

Complexidade de PD-CORTA-BARRA:
$2\sum_{j=1}^{n} j + \Theta(n) = 2n(n+1)/2 + \Theta(n) = \Theta(n^2)$.

Método 2: topdown com memorização

Método 2: topdown com memorização

- Neste método usamos a mesma implementação recursiva, mas a adaptamos para que ele salve a solução de um subproblema (em um vetor ou tabela de hashing).

Método 2: topdown com memorização

- Neste método usamos a mesma implementação recursiva, mas a adaptamos para que ele salve a solução de um subproblema (em um vetor ou tabela de hashing).
- Quando nos deparamos com um subproblema, verificamos se ele já foi resolvido anteriormente.
  Se foi, então devolvemos a solução já computada.
  Se não foi, continuamos a recursão da maneira usual.

Método 2: topdown com memorização

- Neste método usamos a mesma implementação recursiva, mas a adaptamos para que ele salve a solução de um subproblema (em um vetor ou tabela de hashing).
- Quando nos deparamos com um subproblema, verificamos se ele já foi resolvido anteriormente.
  Se foi, então devolvemos a solução já computada.
  Se não foi, continuamos a recursão da maneira usual.
- Dizemos que o procedimento recursivo recebeu memorização: ele lembra das soluções computadas anteriormente.

Implementação recursiva topdown (sem memorização)

```
CORTA-BARRA(p, n)
    se n = 0
        então devolva 0
    q ← −∞
    para i = 1 até n faça
        q = max(q, p_i + CORTA-BARRA(p, n − i))
    devolva q
```

Complexidade: $\Theta(2^n)$.

Topdown com memorização

$\text{MEMO-CORTA-BARRA}(p, n)$

1. **para** $i \leftarrow 0$ **até** n **faça**
2. $\quad r[i] \leftarrow -\infty$
3. **devolva** $\text{MEMO-CORTA-BARRA-AUX}(p, n, r)$

$\text{MEMO-CORTA-BARRA-AUX}(p, n, r)$

1. **se** $r[n] \geq 0$ **então devolva** $r[n]$
2. **se** $n = 0$ **então** $q \leftarrow 0$
3. **senão**
4. $\quad q \leftarrow -\infty$
5. $\quad$ **para** $i \leftarrow 1$ **até** n **faça**
6. $\quad q \leftarrow \max(q, p_i + \text{MEMO-CORTA-BARRA-AUX}(p, n - i))$
7. $r[n] \leftarrow q$
8. **devolva** q

Topdown com memorização

$\textsc{Memo-Corta-Barra-Aux}(p, n, r)$

se $r[n] \geq 0$ **então devolva** $r[n]$
se $n = 0$ **então** $q \leftarrow 0$
senão
    $q \leftarrow -\infty$
    para $i \leftarrow 1$ **até** n **faça**
    $q \leftarrow \max(q, p_i + \textsc{Memo-Corta-Barra-Aux}(p, n - i))$
$r[n] \leftarrow q$
devolva q

Complexidade de $\textsc{Memo-Corta-Barra-Aux}$: ???

Topdown com memorização

$\textsc{Memo-Corta-Barra-Aux}(p, n, r)$

se $r[n] \geq 0$ **então devolva** $r[n]$
se $n = 0$ **então** $q \leftarrow 0$
senão
  $q \leftarrow -\infty$
  para $i \leftarrow 1$ **até** n **faça**
  $q \leftarrow \max(q, p_i + \textsc{Memo-Corta-Barra-Aux}(p, n - i))$
$r[n] \leftarrow q$
devolva q

Complexidade de $\textsc{Memo-Corta-Barra-Aux}$: $\Theta(n^2)$

Topdown com memorização

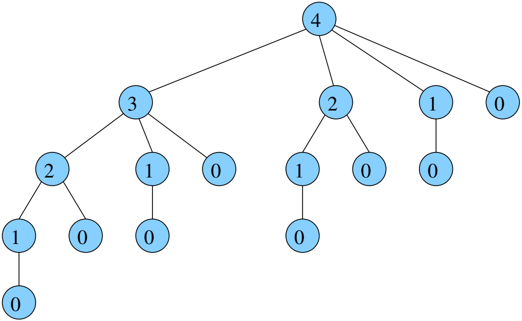
$\text{MEMO-CORTA-BARRA-AUX}(p, n, r)$

1. **se** $r[n] \geq 0$ **então devolva** $r[n]$
2. **se** $n = 0$ **então** $q \leftarrow 0$
3. **senão**
4. $\quad q \leftarrow -\infty$
5. $\quad$ **para** $i \leftarrow 1$ **até** n **faça**
6. $\quad q \leftarrow \max(q, p_i + \text{MEMO-CORTA-BARRA-AUX}(p, n-i))$
7. $r[n] \leftarrow q$
8. **devolva** q

Complexidade de MEMO-CORTA-BARRA-AUX: $\Theta(n^2)$

MEMO-CORTA-BARRA-AUX resolve os subproblema de tamanhos $j = 1, 2, \ldots, n$ exatamente uma vez. Para resolver o subproblema de tamanho j ele executa as linhas 5-6 j vezes. Assim, o tempo total gasto é $\sum_{j=1}^{n} j = \Theta(n^2)$ como no PD-CORTA-BARRA (veja a árvore de recursão na próxima página para entender melhor).

Ávore de recursão de CORTA-BARRA



Na chamada inicial MEMO-CORTA-BARRA-AUX$(p, 4, r)$, é executado o laço com $j = 1, 2, 3, 4$. A memorização evita que subproblemas sejam recalculados.

Memorização versus programação dinâmica

Memorização versus programação dinâmica

- As duas soluções tem complexidade $\Theta(n^2)$, mas a solução por programação dinâmica tem uma constante menor escondida.

Memorização versus programação dinâmica

- As duas soluções tem complexidade $\Theta(n^2)$, mas a solução por programação dinâmica tem uma constante menor escondida.
- Em geral, dada a recorrência para a solução ótima de um problema com subestrutura ótima, é trivial escrever o algoritmo recursivo com memorização.

Memorização versus programação dinâmica

- As duas soluções tem complexidade $\Theta(n^2)$, mas a solução por programação dinâmica tem uma constante menor escondida.
- Em geral, dada a recorrência para a solução ótima de um problema com subestrutura ótima, é trivial escrever o algoritmo recursivo com memorização.
- A solução com programação dinâmica pode ser mais complicada de escrever, mas em geral, é mais rápida que a com memorização.

Reconstruindo uma solução ótima

```
PD-CORTA-BARRA(p, n)
    r[0] ← 0
    para j ← 1 até n faça
        q ← −∞
        para i ← 1 até j faça
            q ← max(q, p[i] + r[j − i])
        r[j] ← q
    devolva r[n]
```

A solução por programação dinâmica acima devolve o lucro máximo, mas não como cortar a barra para atingir esse valor.

Reconstruindo uma solução ótima

```
PD-CORTA-BARRA(p, n)
r[0] ← 0
para j ← 1 até n faça
    q ← −∞
    para i ← 1 até j faça
        q ← max(q, p[i] + r[j − i])
    r[j] ← q
devolva r[n]
```

A solução por programação dinâmica acima devolve o lucro máximo, mas não como cortar a barra para atingir esse valor.

Como faço para computar uma solução ótima?

Reconstruindo uma solução ótima

```
PD-CORTA-BARRA(p, n)
1.  r[0] ← 0
2.  para j ← 1 até n faça
3.      q ← −∞
4.      para i ← 1 até j faça
5.          q ← max(q, p[i] + r[j − i])
6.      r[j] ← q
7.  devolva r[n]
```

Nas linhas 2–6 queremos calcular o valor $r[j]$, i.e., o lucro máximo obtido cortando uma barra de comprimento j. Suponha que o valor q na linha 6 ocorreu para um certo i. Isto significa que o primeiro corte de uma solução ótima corta uma barra de comprimento i.

Reconstruindo uma solução ótima

PD-CORTA-BARRA(p, n)

```
1.  r[0] ← 0
2.  para j ← 1 até n faça
3.      q ← −∞
4.      para i ← 1 até j faça
5.          q ← max(q, p[i] + r[j − i])
6.      r[j] ← q
7.  devolva r[n]
```

Nas linhas 2–6 queremos calcular o valor $r[j]$, i.e., o lucro máximo obtido cortando uma barra de comprimento j. Suponha que o valor q na linha 6 ocorreu para um certo i. Isto significa que o primeiro corte de uma solução ótima corta uma barra de comprimento i.

Dizemos que o valor ótimo do subproblema j segue da escolha do i que maximiza q.

Reconstruindo uma solução ótima

- Podemos estender a programação dinâmica para armazenar o valor ótimo de cada subproblema, assim como a escolha que levou àquela solução.

Reconstruindo uma solução ótima

- Podemos estender a programação dinâmica para armazenar o valor ótimo de cada subproblema, assim como a escolha que levou àquela solução.
- Para uma barra de comprimento j, seja s_j o comprimento do primeiro pedaço a ser cortado em uma solução ótima de valor r_j.

Reconstruindo uma solução ótima

- Podemos estender a programação dinâmica para armazenar o valor ótimo de cada subproblema, assim como a escolha que levou àquela solução.
- Para uma barra de comprimento j, seja s_j o comprimento do primeiro pedaço a ser cortado em uma solução ótima de valor r_j.
- Sabendo os valores s_j's é fácil imprimir uma solução ótima para uma barra de comprimento n.

Reconstruindo uma solução ótima

```
PD-CORTA-BARRA-ESTENDIDO(p, n)
1.  r[0] ← 0
2.  para j ← 1 até n faça
3.      q ← −∞
4.      para i ← 1 até j faça
5.          se q < p_i + r[j − i]
6.              então q ← p_i + r[j − i]
7.                    s[j] ← i
8.      r[j] ← q
9.  devolva r e s
```

Invariante: no início de cada iteração da linha 2, $r[i] = r_i$ e $s[i] = s_i$ para $i = 1, 2, \ldots, j - 1$.

Imprimindo uma solução ótima

$\text{IMPRIME-SOLUCAO-CORTA-BARRA}(p, n)$

$(r, s) \leftarrow \text{PD-CORTA-BARRA-ESTENDIDO}(p, n)$
enquanto $n > 0$ **faça**
    imprima $s[n]$
    $n \leftarrow n - s[n]$

Imprimindo uma solução ótima

$\text{IMPRIME-SOLUCAO-CORTA-BARRA}(p, n)$

1. $(r, s) \leftarrow \text{PD-CORTA-BARRA-ESTENDIDO}(p, n)$
2. **enquanto** $n > 0$ **faça**
3. imprima $s[n]$
4. $n \leftarrow n - s[n]$

A chamada $\text{PD-CORTA-BARRA-ESTENDIDO}(p, 10)$ produz:

j	0	1	2	3	4	5	6	7	8	9	10
$r[j]$	0	1	5	8	10	13	17	18	22	25	30
$s[j]$	0	1	2	3	2	2	6	1	2	3	10

$\text{IMPRIME-SOLUCAO-CORTA-BARRA}(p, 10)$: 10
$\text{IMPRIME-SOLUCAO-CORTA-BARRA}(p, 7)$: 1 6

Resumo

Para projetar um algoritmo de programação dinâmica seguimos os seguintes passos:

Resumo

Para projetar um algoritmo de programação dinâmica seguimos os seguintes passos:

1. Subestrutura ótima: caracterizamos a estrutura de uma solução ótima.

Resumo

Para projetar um algoritmo de programação dinâmica seguimos os seguintes passos:

1. Subestrutura ótima: caracterizamos a estrutura de uma solução ótima.
2. Recorrência: recursivamente definimos o valor de uma solução ótima.

Resumo

Para projetar um algoritmo de programação dinâmica seguimos os seguintes passos:

1. Subestrutura ótima: caracterizamos a estrutura de uma solução ótima.
2. Recorrência: recursivamente definimos o valor de uma solução ótima.
3. Algoritmo: computamos o valor de uma solução ótima, de modo bottom-up.

Resumo

Para projetar um algoritmo de programação dinâmica seguimos os seguintes passos:

1. Subestrutura ótima: caracterizamos a estrutura de uma solução ótima.
2. Recorrência: recursivamente definimos o valor de uma solução ótima.
3. Algoritmo: computamos o valor de uma solução ótima, de modo bottom-up.
4. Reconstrução: construímos uma solução ótima, a partir da informação computada.

Quando queremos apenas o valor ótimo, omitimos o passo (4). Se quisermos executar o passo (4) temos que adaptar o passo (3) para lembrar da escolha que leva a uma solução ótima de cada subproblema.

Exercícios

Exercício 1. Considere o seguinte algoritmo que usa uma estratégia gulosa para o problema de corte de barra. Defina a densidade de uma barra de comprimento i como sendo p_i/i, ou seja, seu valor por metro. A estratégia para uma barra de comprimento n é: corte o primeiro pedaço com comprimento i para o qual a densidade p_i/i é máxima; repita o processo para a barra restante de comprimento $n - i$.
Mostre por meio de um contra-exemplo que esta estratégia nem sempre produz uma solução ótima.

Exercícios

Exercício 2. Considere a modificação do problema de corte de barra na qual, além do preço p_i para cada barra, cada corte resulta em um custo fixo c. O lucro total então é a soma dos preços das barras resultantes menos o custo de fazer os cortes. Projete um algoritmo de programação dinâmica que resolve esta versão modificada do problema.