15 PROGRAMAÇÃO DINÂMICA

A programação dinâmica, assim como o método de divisão e conquista, resolve problemas combinando as soluções para subproblemas. (Nesse contexto, "programação" se refere a um método tabular, não ao processo de escrever código de computador.) Como vimos nos Capítulos 2 e 4, os algoritmos de divisão e conquista subdividem o problema em subproblemas independentes, resolvem os subproblemas recursivamente e depois combinam suas soluções para resolver o problema original. Ao contrário, a programação dinâmica se aplica quando os subproblemas se sobrepõem, isto é, quando os subproblemas compartilham subsubproblemas. Nesse contexto, um algoritmo de divisão e conquista trabalha mais que o necessário, resolvendo repetidamente os subsubproblemas comuns. Um algoritmo de programação dinâmica resolve cada subsubproblema só uma vez e depois grava sua resposta em uma tabela, evitando assim, o trabalho de recalcular a resposta toda vez que resolver cada subsubproblema.

Em geral, aplicamos a programação dinâmica a ***problemas de otimização***. Tais problemas podem ter muitas soluções possíveis. Cada solução tem um valor, e desejamos encontrar uma solução com o valor ótimo (mínimo ou máximo). Denominamos tal solução "*uma*" solução ótima para o problema, ao contrário de "*a*" solução ótima, já que podem existir várias soluções que alcançam o valor ótimo.

O desenvolvimento de um algoritmo de programação dinâmica segue uma sequência de quatro etapas:

1. Caracterizar a estrutura de uma solução ótima.
2. Definir recursivamente o valor de uma solução ótima.
3. Calcular o valor de uma solução ótima, normalmente de baixo para cima.
4. Construir uma solução ótima com as informações calculadas.

As etapas 1 a 3 formam a base de uma solução de programação dinâmica para um problema. Se precisarmos apenas do valor de uma solução ótima, e não da solução em si, podemos omitir a etapa 4. Quando executamos a etapa 4, às vezes mantemos informações adicionais durante a etapa 3, para facilitar a construção de uma solução ótima.

As seções a seguir, usam o método de programação dinâmica para resolver alguns problemas de otimização. A Seção 15.1 examina o problema de cortar uma haste em hastes de menor comprimento, de modo a maximizar os valores totais. A Seção 15.2 pergunta como podemos multiplicar uma cadeia de matrizes e, ao mesmo tempo, executar o menor número total de multiplicações escalares. Dados esses exemplos de programação dinâmica, a Seção 15.3 discute duas características fundamentais que um problema deve ter para que a programação dinâmica seja uma técnica de solução viável. Em seguida, a Seção 15.4 mostra como determinar a subsequência comum mais longa de duas sequências por programação dinâmica. Finalmente, a Seção 15.5 utiliza programação dinâmica para construir árvores de busca binária que são ótimas, dada uma distribuição conhecida de chaves que devem ser examinadas.

15.1 CORTE DE HASTES

Nosso primeiro exemplo usa programação dinâmica para resolver um problema simples que é decidir onde cortar hastes de aço. A Serling Enterprises compra hastes de aço longas e as corta em hastes mais curtas, para vendê-las. Cada corte é livre. A gerência da Serling Enterprises quer saber qual é o melhor modo de cortar as hastes.

Suponhamos que conhecemos, para $i = 1, 2, ..., n$ o preço p_i em dólares que a Serling Enterprises cobra por uma haste de i polegadas de comprimento. Os comprimentos das hastes são sempre um número inteiro de polegadas. A Figura 15.1 apresenta uma amostra de tabela de preços.

O ***problema do corte de hastes*** é o seguinte. Dada uma haste de n polegadas de comprimento e uma tabela de preços p_i para $i = 1, 2, ..., n$, determine a receita máxima r_n que se pode obter cortando a haste e vendendo os pedaços. Observe que, se o preço p_n para uma haste de comprimento n for suficientemente grande, uma solução ótima pode exigir que ela não seja cortada.

Considere o caso quando $n = 4$. A Figura 15.2 mostra todas as maneiras de cortar uma haste de 4 polegadas de comprimento, incluindo não cortá-la. Vemos que cortar uma haste de 4 polegadas em duas peças de 2 polegadas produz a receita $p_2 + p_2 = 5 + 5 = 10$, que é ótima.

Podemos cortar uma haste de comprimento n de 2^{n-1} modos diferentes — já que temos uma opção independente de cortar ou não cortar — à distância de i polegadas da extremidade esquerda, para $i = 1, 2, ..., n - 1$.[1] Denotamos um desdobramento em pedaços usando notação de adição comum, portanto $7 = 2 + 2 + 3$ indica que uma haste de comprimento 7 foi cortada em três peças — duas de comprimento 2 e uma de comprimento 3. Se uma solução ótima cortar a haste em k pedaços, para algum $1 \le k \le n$, então o desdobramento ótimo

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

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

Figura 15.1 Uma amostra de tabela de preços para hastes. Cada haste de comprimento i polegadas rende p_i dólares de receita para a empresa.

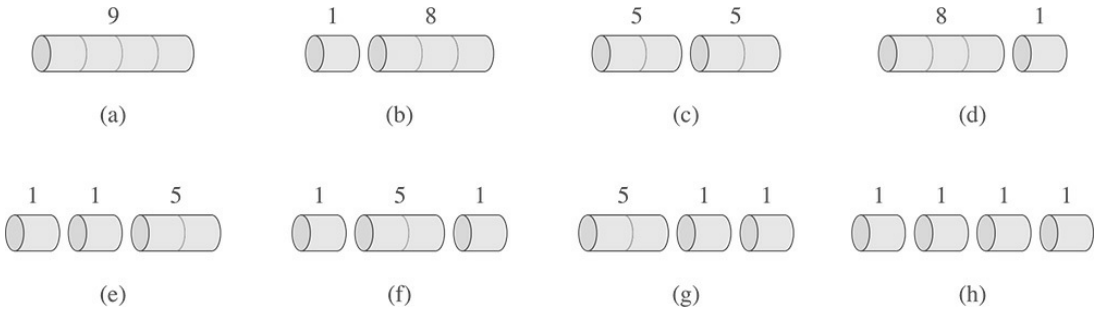


Figura 15.2 Os oito modos possíveis de cortar uma haste de comprimento 4. Acima de cada pedaço está o valor de cada um, de acordo com a amostra de tabela de preços na Figura 15.1. A estratégia ótima é a parte (c) — cortar a haste em duas peças de comprimento 2 — cujo valor total é 10.

da haste em peças de comprimentos $i_1, i_2, ..., i_k$ dá a receita máxima correspondente

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

Em nossa problema-amostra podemos determinar os números da receita ótima r_i , para $i = 1, 2, ..., 10$, por inspeção, com os correspondentes desdobramentos ótimos

$r_1 = 1$ pela solução $1 = 1$ (nenhum corte),
$r_2 = 5$ pela solução $2 = 2$ (nenhum corte),
$r_3 = 8$ pela solução $3 = 3$ (nenhum corte),
$r_4 = 10$ pela solução $4 = 2 + 2$,
$r_5 = 13$ pela solução $5 = 2 + 3$,
$r_6 = 17$ pela solução $6 = 6$ (nenhum corte),
$r_7 = 18$ pela solução $7 = 1 + 6$ ou $7 + 2 + 2 + 3$,
$r_8 = 22$ pela solução $8 = 2 + 6$,
$r_9 = 25$ pela solução $9 = 3 + 6$,
$r_{10} = 30$ pela solução $10 = 10$ (nenhum corte).

De modo mais geral, podemos enquadrar os valores r_n para $n \geq 1$ em termos de receitas ótimas advindas de hastes mais curtas:

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1) \tag{15.1}$$

O primeiro argumento, p_n, corresponde a não fazer nenhum corte e vender a haste de comprimento n como tal. Os outros $n - 1$ argumentos para max correspondem à receita máxima obtida de um corte inicial da haste em duas peças de tamanhos i e $n - i$, para cada $i = 1, 2, \ldots, n - 1$ e, prosseguindo com o corte ótimo dessas peças, obtendo as receitas r_i e r_{n-i} dessas duas peças. Visto que não sabemos de antemão qual é o valor de i que otimiza a receita, temos de considerar todos os valores possíveis para i e escolher aquele que maximize a receita. Temos também a opção de não escolher nenhum i se pudermos obter mais receita vendendo a haste inteira.

Observe que, para resolver o problema original de tamanho n, resolvemos problemas menores do mesmo tipo, porém de tamanhos menores. Uma vez executado o primeiro corte, podemos considerar os dois pedaços como instâncias independentes do problema do corte da haste. A solução ótima global incorpora soluções ótimas para os dois subproblemas relacionados, maximizando a receita gerada por esses dois pedaços. Dizemos que o problema do corte de hastes exibe ***subestrutura ótima***: soluções ótimas para um problema incorporam soluções ótimas para subproblemas relacionados, que podemos resolver independentemente.

Um modo relacionado, mas ligeiramente mais simples de arranjar uma estrutura recursiva para o problema do corte de hastes considera que um desdobramento consiste em uma primeira peça de comprimento i cortada da extremidade esquerda e o que restou do lado direito, com comprimento $n - i$. Somente o resto, e não a primeira peça, pode continuar a ser dividido. Podemos considerar cada desdobramento de uma haste de comprimento n desse modo: uma primeira peça seguida por algum desdobramento do resto. Quando fazemos isso, podemos expressar a solução que não contém nenhum corte dizendo que a primeira peça tem tamanho $i = n$ e receita p_n e que o resto tem tamanho 0 com receita correspondente $r_0 = 0$. Assim, obtemos a seguinte versão mais simples da equação (15.1):

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i}). \tag{15.2}$$

Nessa formulação, uma solução ótima incorpora a solução para somente *um* subproblema relacionado — o resto — em vez de dois.

Implementação recursiva de cima para baixo

O procedimento a seguir, implementa o cálculo implícito na equação (15.2) de um modo direto, recursivo, de cima para baixo.

```
CUT-ROD(p, n)
1  if n == 0
2      return 0
3  q = -∞
4  for i = 1 to n
5      q = max(q, p[i] + CUT-ROD(p, n − i))
6  return q
```

O procedimento CUT-ROD toma como entrada um arranjo $p[1..n]$ de preços e um inteiro n, e retorna a máxima receita possível para uma haste de comprimento n. Se $n = 0$, nenhuma receita é possível e, portanto, CUT-ROD retorna 0 na linha 2. A linha 3 inicializa a receita máxima q para $-\infty$, de modo que o laço **for** na linhas 4-5 calcula corretamente $q = \max^{1 \le i \le n}(pi + \text{CUT-ROD}(p, n, i))$; então, a linha 6 retorna esse valor. Uma simples indução em n prova que essa resposta é igual à resposta desejada r_n, dada pela equação (15.2).

Se você tivesse de codificar CUT-ROD em sua linguagem de programação favorita e executá-lo em seu computador, veria que, tão logo o tamanho se torna moderadamente grande, seu programa levaria um longo tempo para executá-lo. Para $n = 40$, você verificaria que seu programa demoraria no mínimo vários minutos e, o que é mais provável, mais de uma hora. Na verdade, constataria que cada vez que n aumentasse de 1, o tempo de execução de seu programa seria aproximadamente duas vezes maior.

Por que CUT-ROD é tão ineficiente? O problema é que CUT-ROD chama a si mesmo recursivamente repetidas vezes com os mesmos valores de parâmetros; resolve os mesmos problemas repetidamente. A Figura 15.3 ilustra o que acontece para $n = 4$:

CUT-ROD(p, n) chama CUT-ROD$(p, n - i)$ para $i = 1, 2, ..., n$. Equivalentemente, CUT-ROD (p, n) chama CUT-ROD(p, j) para cada $j = 0, 1,..., n - 1$. Quando esse processo se desenrola recursivamente, a quantidade de trabalho realizada, em função de n, cresce explosivamente.

Para analisar o tempo de execução de CUT-ROD, denotamos por $T(n)$ o número total de chamadas feitas a CUT-ROD se chamado quando seu segundo parâmetro é igual a n. Essa expressão é igual ao número de nós em uma subárvore cuja raiz é identificada por n na árvore de recursão. A contagem inclui a chamada inicial à raiz. Assim, $T(0) = 1$ e

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j). \qquad (15.3)$$

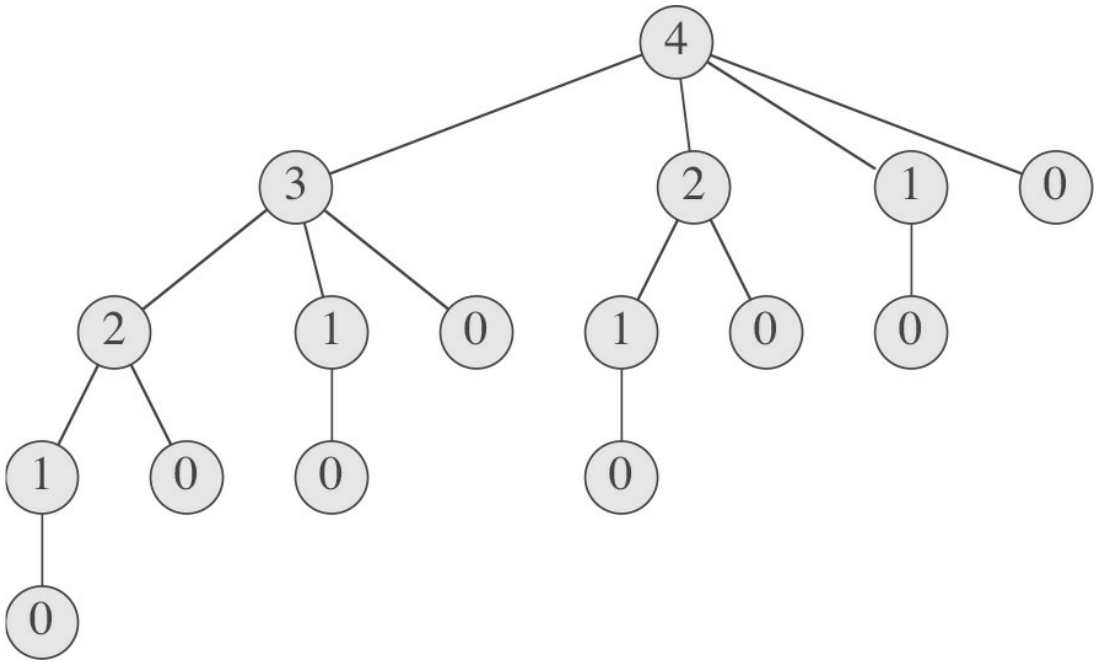


Figura 15.3 A árvore de recursão que mostra chamadas recursivas resultantes de uma chamada a CUT-ROD(p, n) para $n = 4$. Cada rótulo dá o tamanho n do subproblema correspondente, de modo que uma aresta de um pai com rótulo s para um filho com rótulo t corresponde a cortar uma peça inicial de tamanho $s - t$ e deixar um subproblema restante de tamanho t. Um caminho da raiz a uma folha corresponde a um dos 2^{n-1} modos de cortar uma haste de comprimento n. Em geral, essa árvore de recursão tem 2^n nós e 2^{n-1} folhas.

O 1 inicial é para a chamada na raiz, e o termo $T(j)$ conta o número de chamadas (incluindo chamadas recursivas) resultantes da chamada CUT-ROD(p, $n - i$), onde $j = n - i$.

Como o Exercício 15.1-1 pede que você mostre,

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j). \tag{15.3}$$

e, assim, o tempo de execução de CUT-ROD é exponencial em n.

Em retrospectiva, esse tempo de execução exponencial não é surpresa. CUT-ROD considera explicitamente todos os 2^{n-1} modos possíveis de cortar uma haste de comprimento n. A árvore de chamadas recursivas tem 2^{n-1} folhas, uma para cada modo possível de cortar a haste. Os rótulos no caminho simples da raiz até uma folha dão os tamanhos de cada pedaço restante do lado direito da haste antes de cada corte. Isto é, os rótulos dão os pontos de corte correspondentes, medidos em relação à extremidade direita da haste.

Utilização de programação dinâmica para o corte ótimo de hastes

Agora, mostramos como converter CUT-ROD em um algoritmo eficiente usando programação dinâmica.

O método da programação dinâmica funciona da seguinte maneira: agora que já observamos que uma solução recursiva ingênua é ineficiente porque resolve os mesmos problemas repetidas vezes, nós a adaptamos para resolver cada problema somente *uma vez* e armazenar sua solução. Se precisarmos nos referir novamente a esse problema mais tarde, bastará que o examinemos, em vez de o recalcular. Assim, a programação dinâmica usa memória adicional para poupar tempo de computação; serve como exemplo de uma ***permuta tempo-memória.*** As economias de tempo podem ser espetaculares: uma solução de tempo exponencial pode ser transformada em uma solução de tempo polinomial. Uma abordagem de programação dinâmica é executada em tempo polinomial quando o número de problemas *distintos* envolvido é polinomial no tamanho da entrada e podemos resolver cada subproblema em tempo polinomial.

Normalmente, há dois modos equivalentes de implementar uma abordagem de programação dinâmica. Ilustraremos ambos com nosso exemplo do corte de hastes.

A primeira abordagem é ***de cima para baixo com memoização***.[2] Nessa abordagem, escrevemos o procedimento recursivamente de maneira natural, porém modificado para salvar o resultado de cada subproblema (normalmente em um arranjo ou tabela hash). Agora, o procedimento primeiro verifica se já resolveu anteriormente esse subproblema. Se já o resolveu, retorna o valor salvo, poupando computação adicional nesse nível; se ainda não o resolveu, o procedimento calcula o valor da maneira usual. Dizemos que o procedimento recursivo foi ***memoizado***; ele "lembra" quais resultados já calculou anteriormente.

A segunda abordagem é o ***método de baixo para cima***. Essa abordagem depende tipicamente de alguma noção natural do "tamanho" de um subproblema, tal que resolver qualquer subproblema particular depende somente de resolver subproblemas "menores". Ordenamos os subproblemas por tamanho e os resolvemos em ordem de tamanho, o menor primeiro. Ao resolvemos um determinado problema, já resolvemos todos os subproblemas menores dos quais sua solução depende, e já salvamos suas soluções. Resolvemos cada subproblema somente uma vez e, na primeira vez que o vemos, já estão resolvidos todos os seus subproblemas pré-requisitados.

Essas duas abordagens produzem algoritmos com o mesmo tempo de execução assintótico, exceto em circunstâncias incomuns, quando a abordagem de cima para baixo não executa recursão propriamente dita para

examinar todos os subproblemas possíveis. Muitas vezes, a abordagem de baixo para cima tem fatores constantes muito melhores, visto que tem menos sobrecarga para chamadas de procedimento.

Apresentamos a seguir, o pseudocódigo para o procedimento CUT-ROD de cima para baixo, acrescido de memoização:

```
MEMOIZED-CUT-ROD(p, n)
1  seja r[0 .. n] um novo arranjo
2  for i = 0 to n
3     r[i] = -∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)

MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2  return r[n]
3  if n == 0
4     q = 0
5  else q = -∞
6     for i = 1 to n
7        q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
8     r[n] = q
9  return q
```

Aqui, o procedimento principal MEMOIZED-CUT-ROD inicializa um novo arranjo auxiliar $r[0 .. n]$ com o valor -∞, uma escolha conveniente para denotar "desconhecido". (Valores de receita conhecidos são sempre não negativos.) Então, ele chama sua rotina auxiliar, MEMOIZED-CUT-ROD-AUX.

O procedimento MEMOIZED-CUT-ROD-AUX é apenas a versão memoizada de nosso procedimento anterior, CUT-ROD. Primeiro ele consulta a linha 1 para verificar se o valor desejado já é conhecido; se for, a linha 2 retorna esse valor. Caso contrário, as linhas 3-7 calculam o valor desejado q na maneira usual, a linha 8 o salva em $r[n]$, e a linha 9 o retorna.

A versão de baixo para cima é ainda mais simples:

```
BOTTOM-UP-CUT-ROD(p, n)
1  seja r[0 .. n] um novo arranjo
2  r[0] = 0
3  for j = 1 to n
4     q = -∞
5     for i = 1 to j
6        q = max(q, p[i] + r[j − i])
7     r[j] = q
8  return r[n]
```

No caso da abordagem da programação dinâmica de baixo para cima , BOTTOM-UP-CUT--ROD usa a ordenação natural dos subproblemas: um subproblema de tamanho i é "menor" do que um subproblema de tamanho j se $i < j$. Assim, o procedimento resolve subproblemas de tamanhos $j = 0, 1, ..., n$, naquela ordem.

A linha 1 do procedimento BOTTOM-UP-CUT-ROD cria um novo arranjo $r[0 .. n]$ no qual salvar os resultados dos subproblemas, e a linha 2 inicializa $r[0]$ com 0, visto que uma haste de comprimento 0 não rende nenhuma receita. As linhas 3-6 resolvem cada subproblema de tama>nho j, para $j = 1, 2, ..., n$, em ordem crescente de tamanho. A abordagem usada para resolver um problema de determinado tamanho j é a mesma usada por CUT-ROD, exceto que agora a linha 6 referencia diretamente a entrada $r[j - i]$ em vez de fazer uma chamada recursiva para resolver o subproblema de tamanho $j - i$. A linha 7 salva em $r[j]$ a solução do subproblema de tamanho j. Finalmente, a linha 8 retorna $r[n]$, que é igual ao valor ótimo r_n.

As versões de baixo para cima e de cima para baixo têm o mesmo tempo de execução assintótico. O tempo de execução do procedimento BOTTOM-UP-CUT-ROD é $\Theta(n^2)$, devido à sua estrutura de laço duplamente aninhado. O número de iterações de seu laço **for** interno nas linhas 5-6 forma uma série aritmética. O tempo de execução de sua contraparte de cima para baixo, MEMOIZED-CUT-ROD, também é $\Theta(n^2)$, embora esse tempo de execução possa ser um pouco mais difícil de ver. Como uma chamada recursiva para resolver um subproblema já resolvido antes retorna imediatamente, MEMOIZED-CUT-ROD resolve cada subproblema apenas uma vez. Esse procedimento resolve subproblemas para tamanhos 0, 1, ..., n. Para resolver um subproblema de tamanho n, o laço **for** das linhas 6-7 itera n vezes. Assim, o número total de iterações desse laço **for**, para todas as chamadas recursivas de MEMOIZED-CUT-ROD, forma uma série aritmética que dá um total de $\Theta(n^2)$ iterações, exatamente como o laço **for** interno de BOTTOM-UP-CUT-ROD. (Na verdade, aqui estamos usando uma forma de análise agregada. Veremos os detalhes da análise agregada na Seção 17.1.)

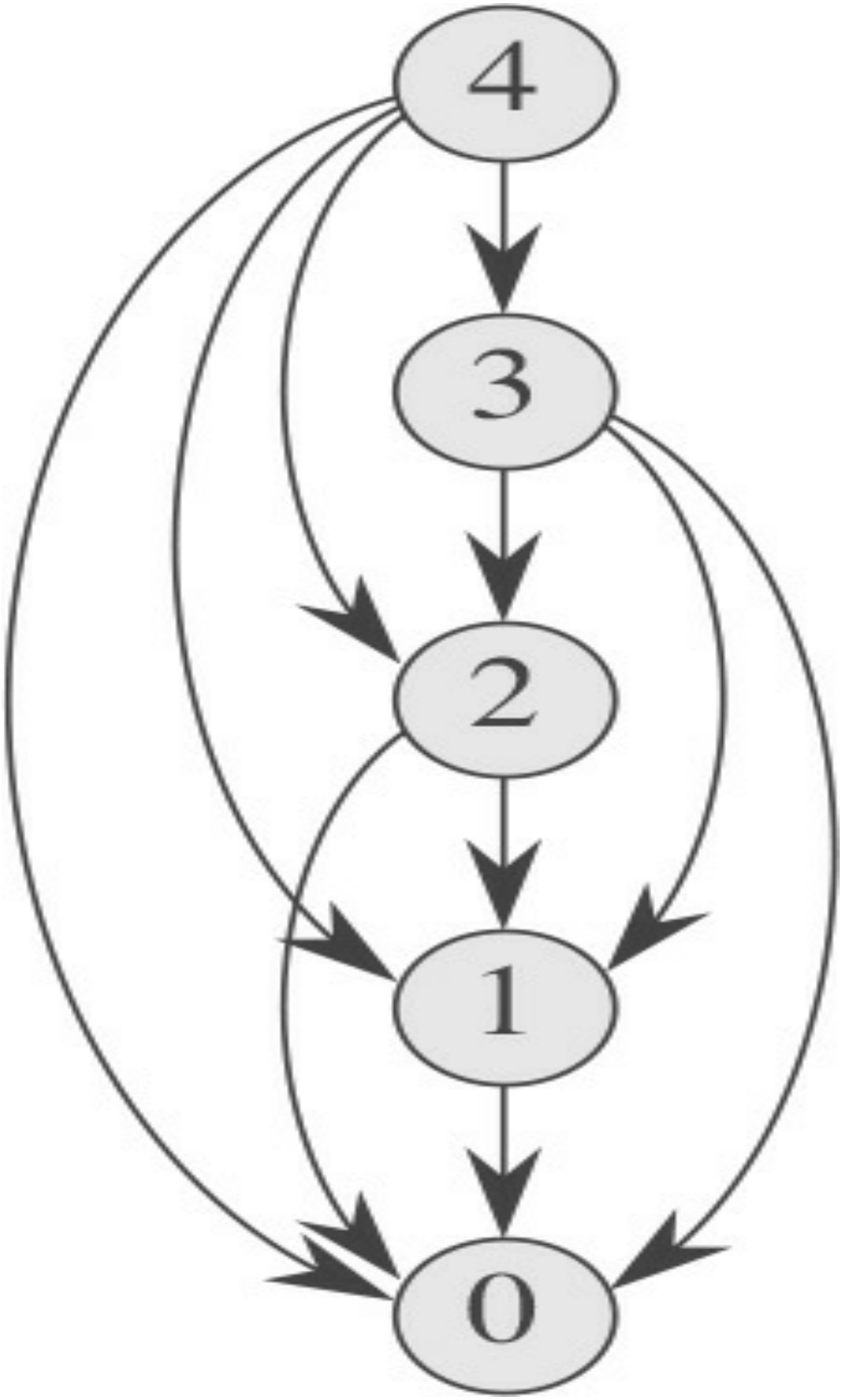

4
3
2
1
0

Figura 15.4 O grafo do subproblema para o problema do corte de haste com $n = 4$. Os rótulos nos vértices dão os tamanhos dos subproblemas correspondentes. Um vértice dirigido (x, y) indica que precisamos de uma solução para o subproblema y quando resolvemos o subproblema x. Esse grafo é uma versão reduzida da árvore da Figura 15.3, na qual todos os nós que têm o mesmo rótulo são integrados em um único vértice e todas as arestas vão de pai para filho.

Grafos de subproblemas

Quando pensamos em um problema de programação dinâmica, temos de entender o conjunto de subproblemas envolvido e como os subproblemas dependem uns dos outros.

O ***grafo de subproblemas*** para o problema incorpora exatamente essa informação. A Figura 15.4 mostra o grafo de subproblemas para o problema do corte de haste com $n = 4$. É um grafo dirigido, que contém um vértice para cada subproblema distinto. O grafo de subproblemas tem um vértice dirigido do vértice para o subproblema x até o vértice para o subproblema y se a determinação de uma solução ótima para o subproblema x envolver considerar diretamente uma solução ótima para o subproblema y. Por exemplo, o grafo de subproblema contém um vértice de x a y se um procedimento recursivo de cima para baixo para resolver x diretamente chamar a si mesmo para resolver y. Podemos imaginar o grafo de subproblema como uma versão "reduzida" ou "colapsada" da árvore de recursão para o método recursivo de cima para baixo, na qual reunimos todos os nós para o mesmo subproblema em um único vértice e orientamos todos os vértices de pai para filho.

O método de baixo para cima para programação dinâmica considera os vértices do grafo de subproblema em uma ordem tal que resolvemos os subproblemas y adjacentes a um dado subproblema x antes de resolvermos o subproblema x. (Lembre-se de que dissemos, na Seção B.4, que a relação de adjacência não é necessariamente simétrica.) Usando a terminologia do Capítulo 22, em um algoritmo de programação dinâmica de baixo para cima, consideramos os vértices do grafo de subproblema em uma ordem que é uma "ordenação topológica reversa" ou uma "ordenação topológica do transposto" (veja a Seção 22.4) do grafo de subproblema. Em outras palavras, nenhum subproblema é considerado até que todos os subproblemas dos quais ele depende tenham sido resolvidos. De modo semelhante, usando noções do mesmo capítulo, podemos considerar o método de cima para baixo (com memoização) para programação dinâmica como uma "busca em profundidade" do grafo de subproblema (veja a Seção 22.3).

O tamanho do grafo de subproblema $G = (V, E)$ pode nos ajudar a determinar o tempo de execução do algoritmo de programação dinâmica. Visto que resolvemos cada subproblema apenas uma vez, o tempo de execução é a soma dos tempos necessários para resolver cada subproblema. Normalmente, o tempo para calcular a solução de um subproblema é proporcional ao grau (número de vértices dirigidos para fora) do vértice correspondente no grafo de subproblema, e o número de subproblemas é igual ao número de vértices no grafo de subproblema. Nesse caso comum, o tempo de execução da programação dinâmica é linear em relação ao número de vértices e arestas.

Reconstruindo uma solução

Nossas soluções de programação dinâmica para o corte de hastes devolvem o valor de uma solução ótima, mas não devolvem uma solução propriamente dita: uma lista de tamanhos de peças. Podemos estender a abordagem da programação dinâmica para registrar não apenas o *valor* ótimo calculado para cada subproblema, mas também uma *escolha* que levou ao valor ótimo. Com essa informação, podemos imprimir imediatamente uma solução ótima.

Apresentamos a seguir uma versão estendida de BOTTOM-UP-CUT-ROD que calcula, para cada tamanho de haste j, não somente a receita máxima r_j, mas também s_j, o tamanho ótimo da primeira peça a ser cortada.

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
1   sejam r[0 .. n] e s[0 .. n] novos arranjos
2   r[0] = 0
3   for j = 1 to n
4       q = -∞
5       for i = 1 to j
6           if q < p[i] + r[j − i]
7               q = p[i] + r[j − i]
8               s[j] = i
9       r[j] = q
10  return r and s
```

Esse procedimento é semelhante a BOTTOM-UP-CUT-ROD, exceto pela criação do arranjo s na linha 1 e pela atualização de $s[j]$ na linha 8 para conter o tamanho ótimo i da primeira peça a cortar ao resolver um subproblema de tamanho j.

O procedimento apresentado a seguir toma um tabela de preços p e um tamanho de haste n e chama EXTENDED-BOTTOM-UP-CUT-ROD para calcular o arranjo $s[1 .. n]$ de tamanhos ótimos da primeira peça e depois imprime a lista completa de tamanho de peças conforme um desdobramento ótimo de uma haste de comprimento n:

```
PRINT-CUT-ROD-SOLUTION(p, n)
1   (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2   while n > 0
3       print s[n]
4       n = n − s[n]
```

Em nosso exemplo do corte de haste, a chamada EXTENDED-BOTTOM-UP-CUT-ROD(p,10) retornaria os seguintes arranjos:

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

Uma chamada a PRINT-CUT-ROD-SOLUTION(p,10) imprimiria apenas 10, mas uma chamada com $n = 7$ imprimiria os cortes 1 e 6, correspondentes ao primeiro desdobramento ótimo para r_7 dado anteriormente.

Exercícios

15.1-1 Mostre que a equação (15.4) decorre da equação (15.3) e da condição inicial $T(0) = 1$.

15.1-2 Mostre, por meio de um contraexemplo, que a seguinte estratégia "gulosa" nem sempre determina um modo ótimo de cortar hastes. Defina a ***densidade*** de uma haste de comprimento i como p_i/i, isto é, seu valor por polegada. A estratégia gulosa para uma haste de comprimento n corta uma primeira peça de comprimento i, onde $1 \leq i \leq n$, om densidade máxima. Então, continua aplicando a estratégia gulosa à peça resultante de comprimento $n - i$.

15.1-3 Considere uma modificação do problema do corte da haste no qual, além de um preço p_i para cada haste, cada corte incorre em um custo fixo c. A receita associada à solução agora é a soma dos preços das peças menos os custos da execução dos cortes. Dê um algoritmo de programação dinâmica para resolver esse problema modificado.

15.1-4 Modifique MEMOIZED-CUT-ROD para retornar não somente o valor, mas também a solução propriamente dita.

15.1-5 Os números de Fibonacci são definidos pela recorrência (3.22). Dê um algoritmo de programação dinâmica de tempo $O[n]$ para calcular o n–ésimo número de Fibonacci. Desenhe o grafo de subproblema. Quantos vértices e arestas existem no grafo?

15.2 Multiplicação de cadeias de matrizes

Nosso próximo exemplo de programação dinâmica é um algoritmo que resolve o problema de multiplicação de cadeias de matrizes. Temos uma sequência (cadeia) $\langle A_1, A_2, ..., A_n \rangle$ de n matrizes para multiplicar e desejamos calcular o produto

$$A_1, A_2 \cdots A_n . \tag{15.5}$$

Podemos avaliar a expressão (15.5) usando o algoritmo-padrão para multiplicação de pares de matrizes como uma sub-rotina, tão logo a tenhamos parentizado para resolver todas as ambiguidades relativas à multiplicação das matrizes entre si. A multiplicação de matrizes é associativa e, portanto, não importa como são colocadas entre parênteses; o produto entre elas é sempre o mesmo. Um produto de matrizes é ***totalmente parentizado*** se for uma única matriz ou o produto de dois produtos de matrizes totalmente parentizadas também expresso entre parênteses. Por exemplo, se a cadeia de matrizes é $\langle A_1, A_2, A_3, A_4 \rangle$, podemos expressar o produto $A_1 A_2 A_3 A_4$ como totalmente parentizado de cinco modos distintos:

$$(A_1(A_2(A_3A_4))) ,$$
$$(A_1((A_2A_3)A_4)) ,$$
$$((A_1A_2)(A_3A_4)) ,$$
$$((A_1(A_2A_3))A_4) ,$$
$$(((A_1A_2)A_3)A_4) .$$

O modo como colocamos parênteses em uma cadeia de matrizes pode ter um impacto expressivo sobre o custo de avaliação do produto. Considere primeiro o custo de multiplicar duas matrizes. O algoritmo-padrão é dado pelo pseudocódigo a seguir, que generaliza o procedimento Square-Matrix-Multiply da Seção 4.2. Os atributos *linhas* e *colunas* são os números de linhas e colunas em uma matriz.

```
MATRIX-MULTIPLY(A,B)
1   if A.colunas ≠ B.linhas
2      error "dimensões incompatíveis"
3   else seja C uma nova A.linhas • B.colunas
4      for i = 0 to A.linhas
5         for j = 1 to B.colunas
6            c_ij = 0
7            for k = 1 to A.colunas
8               c_ij = c_ij + a_ik · a_jk
9      return C
```

Podemos multiplicar duas matrizes A e B somente se elas forem ***compatíveis***: o número de colunas de A deve ser igual ao número de linhas de B. Se A é uma matriz $p \cdot q$ e B é uma matriz $q \cdot r$, a matriz resultante C é uma matriz $p \cdot r$. O tempo para calcular C é dominado pelo número de multiplicações escalares na linha 8, que é pqr. A seguir, expressaremos os custos em termos do número de multiplicações escalares.

Para ilustrar os diferentes custos incorridos pelas diferentes posições dos parênteses em um produto de matrizes, considere o problema de uma cadeia $\langle A_1, A_2, A_3 \rangle$ de três matrizes. Suponha que as dimensões das matrizes sejam $10 \cdot 100$, $100 \cdot 5$ e $5 \cdot 50$, respectivamente. Se multiplicarmos as matrizes de acordo com a posição dos parênteses $((A_1A_2)A_3)$, executaremos $10 \cdot 100 \cdot 5 = 5.000$ multiplicações escalares para calcular o produto $10 \cdot 5$ de matrizes A_1 A_2, mais outras $10 \cdot 5 \cdot 50 = 2.500$ multiplicações escalares para multiplicar essa matriz por A_3, produzindo um total de 7.500 multiplicações escalares. Se, em vez disso, multiplicarmos de acordo com a posição dos parênteses $(A_1(A_2 A_3))$, executaremos $100 \cdot 5 \cdot 50 = 25.000$ multiplicações escalares para calcular o produto $100 \cdot 50$ de matrizes $A_2 A_3$, mais outras $10 \cdot 100 \cdot 50 = 50.000$ multiplicações escalares para multiplicar A_1 por essa matriz, dando um total de 75.000 multiplicações escalares. Assim, o cálculo do produto de acordo com a primeira posição dos parênteses é 10 vezes mais rápido.

Enunciamos o ***problema de multiplicação de cadeias de matrizes*** da seguinte maneira: dada uma cadeia $\langle A_1, A_2, \ldots A_n \rangle$ de n matrizes onde, para $i = 1, 2, \ldots, n$, a matriz A_i tem dimensão $p_{i-1} \cdot p_i$, expresse o produto $A_1, A_2, \ldots A_n$ como um produto totalmente parentizado de modo a minimizar o número de multiplicações escalares.

Observe que no problema de multiplicação de cadeias de matrizes, não estamos realmente multiplicando matrizes. Nossa meta é apenas determinar uma ordem para multiplicar matrizes que tenha o custo mais baixo. Em geral, o tempo investido na determinação dessa ordem ótima é mais que compensado pelo tempo economizado mais tarde, quando as multiplicações de matrizes são de fato executadas (por exemplo, executar apenas 7.500 multiplicações escalares em vez de 75.000).

Contagem do número de parentizações

Antes de resolver o problema de multiplicação de cadeias de matrizes por programação dinâmica, temos de nos convencer de que a verificação exaustiva de todas as possíveis parentizações não resulta em um algoritmo eficiente. Vamos representar por $P(n)$ o número de parentizações alternativas de uma sequência de n matrizes. Quando $n = 1$, há apenas uma matriz e, portanto, somente um modo de parentizar totalmente o produto de matrizes. Quando $n = 2$, um produto de matrizes totalmente parentizado é o produto de dois subprodutos de matrizes totalmente parentizados, e a separação entre os dois subprodutos pode ocorrer entre a k-ésima e a $(k + 1)$-ésima matrizes para qualquer $k = 1, 2, \ldots, n - 1$. Assim, obtemos a recorrência

$$P(n) = \begin{cases} 1 & \text{se } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{se } n \geq 2. \end{cases} \tag{15.6}$$

O Problema 12-4 pediu para mostrar que a solução para uma recorrência semelhante é a sequência de ***números de Catalan***, que cresce como $(4^n/n_{3/2})$. Um exercício mais simples (veja o Exercício 15.2-3) é mostrar que a solução para a recorrência (15.6) é (2^n). Portanto, o número de soluções é exponencial em n e o método de força bruta de busca exaustiva é uma estratégia ruim para determinar a parentização ótima de uma cadeia de matrizes.

Aplicação da programação dinâmica

Usaremos o método da programação dinâmica para determinar a parentização ótima de uma cadeia de matrizes. Para tal, seguiremos a sequência de quatro etapas declaradas no início deste capítulo:

1. Caracterizar a estrutura de uma solução ótima.
2. Definir recursivamente o valor de uma solução ótima.
3. Calcular o valor de uma solução ótima.
4. Construir uma solução ótima com as informações calculadas.

Percorreremos essas etapas em ordem, demonstrando claramente como aplicar cada etapa ao problema.

Etapa 1: A estrutura de uma parentização ótima

Em nossa primeira etapa do paradigma de programação dinâmica, determinamos a subestrutura ótima e depois a usamos para construir uma solução ótima para o problema partindo de soluções ótimas para subproblemas. No problema de multiplicação de cadeias de matrizes, podemos executar essa etapa da maneira descrita a seguir. Por conveniência, vamos adotar a notação $A_{i..j}$, onde $i \leq j$ para a matriz que resulta da avaliação do produto $A_i A_{i+1} \cdots A_j$. Observe que, se o problema é não trivial, isto é, se $i < j$, então, para parentizar o produto $A_i A_{i+1} \cdots A_j$ temos de separá-lo entre A_k e A_{k+1} para algum inteiro k no intervalo $i \leq k < j$. Isto é, para algum valor de k, primeiro calculamos as matrizes $A_{i..k}$ e $A_{k+1..j}$, e depois multiplicamos as duas para gerar o produto final $A_{i..j}$. O custo dessa parentização é o custo de calcular a matriz $A_{i..k}$, mais o custo de calcular $A_{k+1..j}$, mais o custo de multiplicá-las uma pela outra.

A subestrutura ótima desse problema é dada a seguir. Suponha que para efetuar a parentização ótima de $A_i A_{i+1} \cdots A_j$ separamos o produto entre A_k e A_{k+1}. Então, o modo como posicionamos os parênteses na subcadeia "prefixo" $A_i A_{i+1} \cdots A_k$ dentro dessa parentização ótima de $A_i A_{i+1} \cdots A_j$ deve ser uma parentização ótima de $A_i A_{i+1} \cdots A_k$. Por quê? Se existisse um modo menos dispendioso de parentizar $A_i A_{i+1} \cdots A_k$, então poderíamos substituir essa parentização na parentização ótima de $A_i A_{i+1} \cdots A_j$ para produzir um outro modo de parentizar $A_i A_{i+1} \cdots A_j$ cujo custo seria mais baixo que o custo ótimo: uma contradição. Uma observação semelhante é válida para parentizar a subcadeia $A_{k+1} A_{k+2} \cdots A_j$ na parentização ótima de $A_i A_{i+1} \cdots A_j$: ela deve ser uma parentização ótima de $A_{k+1} A_{k+2} \cdots A_j$.

Agora, usamos nossa subestrutura ótima para mostrar que podemos construir uma solução ótima para o problema pelas soluções ótimas para subproblemas. Vimos que qualquer solução para uma instância não trivial do problema de multiplicação de cadeias de matrizes requer que separemos o produto e que qualquer solução ótima contém em si soluções ótimas para instâncias de subproblemas. Assim, podemos construir uma solução ótima para uma instância do problema de multiplicação de cadeias de matrizes separando o problema em dois subproblemas (pela parentização ótima de $A_i A_{i+1} \cdots A_k$ e $A_{k+1} A_{k+2} \cdots A_j$), determinando soluções ótimas para instâncias de subproblemas e depois combinando essas soluções ótimas de subproblemas. Devemos assegurar que, quando procurarmos o lugar correto para separar o produto, consideremos todos os lugares possíveis, para termos certeza de que examinamos a opção ótima.

Etapa 2: Uma solução recursiva

Em seguida, definimos recursivamente o custo de uma solução ótima em termos das soluções ótimas para subproblemas. No caso do problema de multiplicação de cadeias de matrizes, escolhemos como nossos subproblemas os problemas da determinação do custo mínimo da parentização de $A_i A_{i+1} \cdots A_j$ para $1 \le i \le j \le n$. Seja $m[i, j]$ o número mínimo de multiplicações escalares necessárias para calcular a matriz $A_{i..j}$; para o problema completo, o custo mínimo para calcular $A_{1..n}$ seria, portanto, $m[1, n]$.

Podemos definir $m[i, j]$ recursivamente da maneira descrita a seguir. Se $i = j$, o problema é trivial; a cadeia consiste em apenas uma matriz $A_{i..i} = A_i$, de modo que nenhuma multiplicação escalar é necessária para calcular o produto. Assim, $m[i, i] = 0$ para $i = 1, 2, ..., n$. Para calcular $m[i, j]$ quando $i < j$, tiramos proveito da estrutura de uma solução ótima da etapa 1. Vamos considerar que, para obter a parentização ótima, separamos o produto $A_i A_{i+1} \cdots A_j$ entre A_k e A_{k+1}, onde $i \le k < j$. Então, $m[i, j]$ é igual ao custo mínimo para calcular os subprodutos $A_{i..k}$ e $A_{k+1..j}$, mais o custo de multiplicar essas duas matrizes. Recordando que cada matriz A_i é $p_{i-1} \cdot p_i$, vemos que o cálculo do produto de matrizes $A_{i..k} A_{k+1..j}$ exige $p_{i-1} p_k p_j$ multiplicações escalares. Assim, obtemos

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j .$$

Essa equação recursiva supõe que conhecemos o valor de k, o que não é verdade. Porém, há somente $j - i$ valores possíveis para k, isto é, $k = i, i + 1, ..., j - 1$. Visto que a parentização ótima deve usar um desses valores para k, precisamos apenas verificar todos eles para determinar o melhor. Assim, nossa definição recursiva para o custo mínimo de colocar o produto $A_i A_{i+1} \cdots A_j$ entre parênteses se torna

$$m[i, j] = \begin{cases} 0 & \text{se } i = j, \\ \min\limits_{i \le k \le j} \{m[i, k] + m[k+1, j] + p_{i-1} p_k p_j\} & \text{se } i < j. \end{cases} \tag{15.7}$$

Os valores $m[i, j]$ dão os custos de soluções ótimas para subproblemas, mas não dão todas as informações de que necessitamos para construir uma solução ótima. Para nos ajudar a fazer isso, definimos $s[i, j]$ como um valor de k no qual separamos o produto $A_i A_{i+1} \cdots A_j$ para obter uma parentização ótima. Isto é, $s[i, j]$ é igual a um valor k tal que $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$.

Etapa 3: Cálculo dos custos ótimos

Neste ponto, seria fácil escrever um algoritmo recursivo baseado na recorrência (15.7) para calcular o custo mínimo $m[1, n]$ para multiplicar $A_1 A_2 \cdots A_n$. Como vimos no problema do corte de hastes de aço, e como veremos na Seção 15.3, esse algoritmo recursivo demora um tempo exponencial, o que não é nada melhor que o método da força bruta de verificar cada maneira de parentizar o produto.

Observe que temos um número relativamente pequeno de subproblemas distintos: um problema para cada escolha de i e j que satisfaça $1 \le i \le j \le n$ ou $\binom{n}{2} + n = \Theta(n^2)$ no total. Um algoritmo recursivo pode encontrar cada subproblema, muitas vezes, em diferentes ramos de sua árvore de recursão. Essa propriedade de sobrepor subproblemas é a segunda indicação da aplicabilidade da programação dinâmica (a subestrutura ótima é a primeira).

Em vez de calcular a solução para a recorrência (15.7) recursivamente, calculamos o custo ótimo usando uma abordagem tabular de baixo para cima. (Na Seção 15.3, apresentaremos a abordagem de cima para baixo correspondente usando memoização.)

Implementaremos o método tabular, de cima para baixo, no procedimento MATRIX-CHAIN-ORDER, que aparece mais adiante. Esse procedimento supõe que a matriz A_i tem dimensões $p_{i-1} \cdot p_i$ para $i = 1, 2, ..., n$. Sua entrada é uma sequência $p = \langle p_0, p_1, ..., p_n \rangle$, onde $p.comprimento = n + 1$. O procedimento utiliza uma tabela auxiliar $m[1 .. n, 1 .. n]$

para armazenar os custos $m[i, j]$ e uma outra tabela auxiliar $s[1 .. n - 1 .. n]$ que registra qual índice de k alcançou o custo ótimo no cálculo de $m[i, j]$. Usaremos a tabela s para construir uma solução ótima.

Para implementar a abordagem de baixo para cima, devemos determinar a quais entradas da tabela nos referimos para calcular $m[i, j]$. A equação (15.7) mostra que o custo $m[i, j]$ de calcular um produto de cadeias de $j - i + 1$ matrizes só depende dos custos de calcular os produtos de cadeias de menos que $j - i + 1$ matrizes. Isto é, para $k = i, i + 1, ..., j - 1$, a matriz $A_{i..k}$ é um produto de $k - i + 1 < j - i + 1$ matrizes, e a matriz $A_{k+1..j}$ é um produto de $j - k < j - i + 1$ matrizes. Assim, o algoritmo deve preencher a tabela m de um modo que corresponda a resolver o problema da parentização em cadeias de matrizes de comprimento crescente. Para o subproblema da parentização ótima da cadeia $A_i A_{i+1} \cdots A_j$, admitimos que o tamanho do subproblema é o comprimento $j - i + 1$ da cadeia.

```
MATRIX-CHAIN-ORDER (p)
1   n = p.comprimento − 1
2   sejam m[1 .. n, 1 .. n] e s[1 .. n − 1,2 .. n] tabelas novas
3   for i = 1 to n
4       m[i, i] = 0
5   for l = 2 to n                  // l é o comprimento da cadeia
6       for i = 1 to n − l + 1
7           j = i + l − 1
8           m[i, j] = ∞
9           for k = i to j − 1
10              q = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j
11              if q < m[i, j]
12                  m[i, j] = q
13                  s[i, j] = k
14  return m e s
```

O algoritmo calcula primeiro $m[i, i] = 0$ para $i = 1, 2, ..., n$ (os custos mínimos para cadeias de comprimento 1) nas linhas 3-4. Então, usa a recorrência (15.7) para calcular $m[i, i + 1]$ para $i = 1, 2, ..., n - 1$ (os custos mínimos para cadeias de comprimento $l = 2$) durante a primeira execução do laço **for** nas linhas 5-13. Na segunda passagem pelo laço, o algoritmo calcula $m[i, i + 2]$ para $i = 1, 2, ..., n - 2$ (os custos mínimos para cadeias de comprimento $l = 3$), e assim por diante. Em cada etapa, o custo $m[i, j]$ calculado nas linhas 10-13 depende apenas das entradas de tabela $m[i, k]$ e $m[k + 1, j]$ já calculadas.

A Figura 15.5 ilustra esse procedimento em uma cadeia de $n = 6$ matrizes. Visto que definimos $m[i, j]$ somente para $i = j$, apenas a porção da tabela m estritamente acima da diagonal principal é usada. A tabela mostrada na figura sofreu uma rotação para colocar a diagonal principal na posição horizontal. A lista ao longo da parte inferior da figura mostra a cadeia de matrizes. Usando esse leiaute, podemos determinar o custo mínimo $m[i, j]$ para multiplicar uma subcadeia de matrizes $A_i A_{i+1} \cdots A_j$ na interseção de linhas que partem de A_i na direção nordeste e de A_j. Cada linha horizontal na tabela contém as entradas para cadeias de matrizes do mesmo comprimento. MATRIX-CHAIN-ORDER calcula as linhas de baixo para cima e da esquerda para a direita dentro de cada linha. Calcula cada entrada $m[i, j]$ usando os produtos $p_i - {}^1 p_k {}^{pj}$ para $k = i, i + 1, ..., j - 1$ e todas as entradas a sudoeste e a sudeste de $m[i, j]$.

Uma simples inspeção da estrutura de laços aninhados de MATRIX-CHAIN-ORDER produz um tempo de execução de $O(n_3)$ para o algoritmo. Os laços estão aninhados com profundidade três, e cada índice de laço (l, i e k) adota no máximo n valores. O Exercício 15.2-5 pede que você mostre que o tempo de execução desse algoritmo é mesmo (n_3). O algoritmo requer espaço $Q(n_2)$ para armazenar as tabelas m e s. Assim, MATRIX-CHAIN-ORDER é muito mais eficiente que o método de tempo exponencial que enumera todas as possíveis parentizações e verifica cada uma delas.

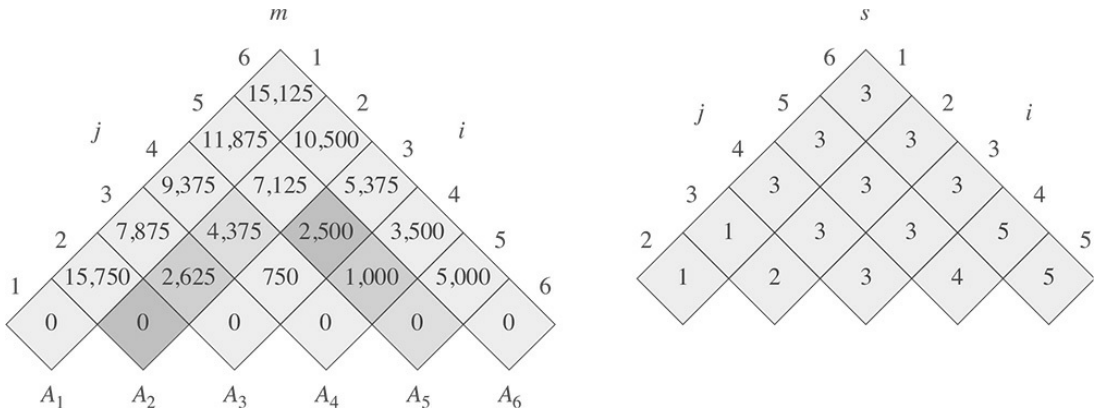


Figura 15.5 As tabelas *m* e *s* calculadas por MATRIX-CHAIN-ORDER para $n = 6$ e as seguintes dimensões de matrizes:

matriz	A_1	A_2	A_3	A_4	A_5	A_6
dimensão	$30 \cdot 35$	$35 \cdot 15$	$15 \cdot 5$	$5 \cdot 10$	$10 \cdot 20$	$20 \cdot 25$

As tabelas sofreram uma rotação para colocar a diagonal principal na posição horizontal. A tabela *m* usa somente a diagonal principal e o triângulo superior, e a tabela *s* usa somente o triângulo superior. O número mínimo de multiplicações escalares para multiplicar as seis matrizes é $m[1, 6] = 15.125$. Entre as entradas sombreadas nos três tons mais escuros, os pares que têm o mesmo sombreado são tomados juntos na linha 10 quando se calcula

$$\begin{aligned} m[2,5] &= \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13{,}000, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11{,}375 \end{cases} \\ &= 7125. \end{aligned}$$

Etapa 4: Construção de uma solução ótima

Embora determine o número ótimo de multiplicações escalares necessárias para calcular um produto de cadeias de matrizes, MATRIX-CHAIN-ORDER não mostra diretamente como multiplicar as matrizes. A tabela $s[1 .. n - 1, 2, ..., n]$ nos dá a informação que precisamos para fazer isso. Cada entrada $s[i, j]$ registra um valor de k tal que uma parentização ótima de $A_i A_{i+1} \cdots A_j$ separa o produto entre A_k e A_{k+1}. Assim, sabemos que a multiplicação de matrizes final no cálculo ótimo de $A_{1..n}$, é $A_{1..s[1,n]} A_{s[1,n]+1..n}$. Podemos determinar as multiplicações de matrizes anteriores recursivamente, já que $s[1, s[1, n$ determina a última multiplicação de matrizes no cálculo de $A_{1..s[1,n]}$ e $s[s[1, n] + 1, n]$ determina a última multiplicação de matrizes no cálculo de $A_{s[1,n]+1..n}$. O procedimento recursivo a seguir imprime uma parentização ótima de $\langle A_i, A_{i+1}, ..., A_j \rangle$, dada a tabela *s* calculada por MATRIX-CHAIN-ORDER e os índices *i* e *j*. A chamada inicial PRINT-OPTIMAL-PARENS$(s, 1, n)$ imprime uma parentização ótima de $\langle A_1, A_2, ..., An \rangle$.

PRINT-OPTIMAL-PARENS(s, i, j)

```
1  if i == j
2     print "A"_i
3  else print "("
4     PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5     PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6     print ")"
```

No exemplo da Figura 15.5 a chamada PRINT-OPTIMAL-PARENS$(s, 1, 6)$ imprime a parentização de $((A_1 (A_2 A_3)) ((A_4 A_5) A_6))$.

Exercícios

15.2-1 Determine uma parentização ótima de um produto de cadeias de matrizes cuja sequência de dimensões é $\langle 5, 10, 3, 12, 5, 50, 6\rangle$.

15.2-2 Dê um algoritmo recursivo MATRIX-CHAIN-MULTIPLY(A, s, i, j) que realmente execute a multiplicação ótima de cadeias de matrizes, dadas a sequência de matrizes $\langle A_1, A_2, ..., A_n\rangle$, a tabela s calculada por MATRIX-CHAIN-ORDER e os índices i e j. (A chamada inicial seria MATRIX-CHAIN-MULTIPLY$(A, s, 1, n)$.)

15.2-3 Use o método de substituição para mostrar que a solução para a recorrência (15.6) é $\Omega(2^n)$.

15.2-4 Descreva o grafo de subproblema para multiplicação de cadeia de matrizes com uma cadeia de entrada de comprimento n. Quantos vértices ele tem? Quantas arestas ele tem e quais são essas arestas?

15.2-5 Seja $R(i, j)$ o número de vezes que a entrada de tabela $m[i, j]$ é referenciada durante o cálculo de outras entradas de tabela em uma chamada de MATRIX-CHAIN-ORDER. Mostre que o número total de referências para a tabela inteira é

$$\sum_{i=j}^{n}\sum_{j=i}^{n} R(i,j) = \frac{n^3 - n}{3}.$$

(*Sugestão*: A equação (A.3) pode ser útil.)

15.2-6 Mostre que uma parentização completa de uma expressão de n elementos tem exatamente $n - 1$ pares de parênteses.

15.3 ELEMENTOS DE PROGRAMAÇÃO DINÂMICA

Embora tenhamos acabado de analisar dois exemplos do método de programação dinâmica, é bem possível que você ainda esteja imaginando exatamente quando aplicar o método. Do ponto de vista da engenharia, quando devemos procurar uma solução de programação dinâmica para um problema? Nesta seção, examinamos os dois elementos fundamentais que um problema de otimização deve ter para que a programação dinâmica seja aplicável: subestrutura ótima e sobreposição de subproblemas. Também voltaremos a discutir mais completamente como a memoização pode

nos ajudar a aproveitar a propriedade de sobreposição de subproblemas em uma abordagem recursiva de cima para baixo.

Subestrutura ótima

O primeiro passo para resolver um problema de otimização por programação dinâmica é caracterizar a estrutura de uma solução ótima. Lembramos que um problema apresenta uma ***subestrutura ótima*** se uma solução ótima para o problema contiver soluções ótimas para subproblemas. Sempre que um problema exibir subestrutura ótima, temos uma boa indicação de que a programação dinâmica pode se aplicar. (Porém, como discutiremos no Capítulo 16, isso também pode significar que uma estratégia gulosa é aplicável.) Em programação dinâmica, construímos uma solução ótima para o problema partindo de soluções ótimas para subproblemas. Consequentemente, devemos ter o cuidado de garantir que a faixa de subproblemas que consideramos inclui aqueles usados em uma solução ótima.

Encontramos uma subestrutura ótima em ambos os problemas examinados neste capítulo até agora. Na Seção 15.1, observamos que o modo mais rápido de cortar uma haste de comprimento n (se a cortarmos) envolve cortar otimamente as duas peças resultantes do primeiro corte. Na Seção 15.2, observamos que uma parentização ótima de $A_i A_{i+1} \cdots A_j$ que separa o produto entre A_k e A_{k+1} contém soluções ótimas para os problemas de parentização de $A_i A_{i+1} \cdots A_k$ e $A_{k+1} A_{k+2} \cdots A_j$.

Você perceberá que está seguindo um padrão comum para descobrir a subestrutura ótima:

1. Mostrar que uma solução para o problema consiste em fazer uma escolha, como a de escolher um corte inicial em uma haste ou um índice no qual separar a cadeia de matrizes. Essa escolha produz um ou mais subproblemas a resolver.
2. Supor que, para um dado problema, existe uma escolha que resulta em uma solução ótima. Você ainda não se preocupa com a maneira de determinar essa escolha. Basta supor que ela existe.
3. Dada essa escolha, determinar quais subproblemas dela decorrem e como caracterizar melhor o espaço de subproblemas resultante.
4. Mostrar que as soluções para os subproblemas usados dentro de uma solução ótima para o problema também devem ser ótimas utilizando uma técnica de "recortar e colar". Para tal, você supõe que alguma das soluções de subproblemas não é ótima e, então, deduz uma contradição. Em particular, "recortando" a solução não ótima para cada subproblema e "colando" a solução ótima, você mostra que pode conseguir uma solução melhor para o problema original, o que contradiz a suposição de que você já tinha uma solução ótima. Se uma solução ótima der origem a mais de um subproblema, normalmente eles serão tão semelhantes que você pode modificar o argumento "recortar e colar" usado para um deles e aplicá-lo aos outros com pouco esforço.

Para caracterizar o espaço de subproblemas, uma boa regra prática é tentar manter o espaço tão simples quanto possível e depois expandi-lo conforme necessário. Por exemplo, o espaço de subproblemas que consideramos para o problema do corte da haste continha os problemas de determinar o corte ótimo para uma haste de comprimento i para cada tamanho i. Esse espaço de subproblemas funcionou bem e não tivemos nenhuma necessidade de tentar um espaço de subproblemas mais geral.

Ao contrário, suponha que tivéssemos tentado restringir nosso espaço de subproblemas para a multiplicação de cadeias de matrizes a produtos de matrizes da forma $A_1 A_2 \cdots A_j$. Como antes, uma parentização ótima deve separar esse produto entre A_k e A_{k+1} para algum $1 \le k \le j$. A menos que possamos garantir que k é sempre igual a $j - 1$, constataremos que tínhamos subproblemas da forma $A_1 A_2 \cdots A_k$ e $A_{k+1} A_{k+2} \cdots Aj$, e que este último subproblema não é da forma $A_1 A_2 \cdots A_j$. Para esse problema, tivemos que permitir que nossos subproblemas variassem em "ambas as extremidades", isto é, permitir que i e j variassem no subproblema $A_i A_{i+1} \cdots A_j$.

A subestrutura ótima varia nos domínios de problemas de duas maneiras:

1. o número de subproblemas usados em uma solução ótima para o problema original.
2. o número de opções que temos para determinar qual(is) subproblema(s) usar em uma solução ótima.

No problema do corte da haste, uma solução ótima para cortar uma haste de tamanho n usa apenas um subproblema (de tamanho $n - i$), mas temos de considerar n escolhas para i para determinar qual deles produz uma solução ótima. A multiplicação de cadeias de matrizes para a subcadeia $A_i A_{i+1} \cdots A_j$ serve como um exemplo com dois subproblemas e $j - i$ escolhas. Para uma dada matriz A_k na qual separamos o produto, temos dois subproblemas — a parentização de $A_i A_{i+1} \cdots A_k$ e a parentização de $A_{k+1} A_{k+2} \cdots A_j$ — e devemos resolver *ambos* otimamente. Uma vez determinadas as soluções ótimas para subproblemas, escolhemos entre $j - i$ candidatos para o índice k.

Informalmente, o tempo de execução de um algoritmo de programação dinâmica depende do produto de dois fatores: o número global de subproblemas e quantas escolhas consideramos que existem para cada subproblema. No corte de hastes tínhamos $\Theta(n)$ subproblemas no total e no máximo n escolhas para examinar para cada um, resultando no tempo de execução $O(n^2)$. Na multiplicação de cadeias de matrizes, tínhamos $\Theta(n^2)$ subproblemas no total e, em cada um deles, tínhamos no máximo $n - 1$ escolhas, dando um tempo de execução $O(n^3)$ (na verdade, um tempo de execução $\Theta(n^3)$ pelo Exercício 15.2-5).

Normalmente, o grafo de subproblemas dá um modo alternativo de executar a mesma análise. Cada vértice corresponde a um subproblema, e as escolhas para um subproblema são as arestas que nele incidem. Lembre-se de que, no corte de hastes, o grafo do subproblema tinha n vértices e no máximo n arestas por vértice, resultando no tempo de execução $O(n^2)$. Na multiplicação de cadeia de matrizes, o grafo de subproblemas, se o tivéssemos desenhado, teria $\Theta(n^2)$ vértices e cada vértice teria um grau de no máximo $n - 1$, o que resultaria em um total de $O(n^3)$ vértices e arestas.

A programação dinâmica usa frequentemente a subestrutura ótima de baixo para cima. Isto é, primeiro encontramos soluções ótimas para subproblemas e, resolvidos os subproblemas, encontramos uma solução ótima para o problema. Encontrar uma solução ótima para o problema acarreta escolher, entre os subproblemas, aqueles que usaremos na solução do problema. Normalmente, o custo da solução do problema é igual aos custos dos subproblemas, mais um custo atribuível diretamente à escolha em si. Por exemplo, no corte de hastes, primeiro resolvemos os subproblemas de determinar maneiras ótimas de cortar hastes de comprimento i para $i = 0, 1, \ldots, n - 1$ e depois determinamos qual subproblema produz uma solução ótima para uma haste de comprimento n, usando a equação (15.2). O custo atribuível à escolha em si é o termo p_i na equação (15.2). Na multiplicação de cadeias de matrizes, determinamos a parentização ótima de subcadeias de $A_i A_{i+1} \cdots A_j$, e então escolhemos a matriz A_k na qual separar o produto. O custo atribuível à escolha propriamente dita é o termo $p_{i-1} p_k p_j$.

No Capítulo 16, examinaremos os "algoritmos gulosos", que guardam muitas semelhanças com a programação dinâmica. Em particular, os problemas aos quais os algoritmos gulosos se aplicam têm subestrutura ótima. Uma diferença notável entre algoritmos gulosos e programação dinâmica é que, em vez de primeiro encontrar soluções ótimas para subproblemas e depois fazer uma escolha informada, os algoritmos gulosos primeiro fazem uma escolha "gulosa" — a que parece melhor no momento — e depois resolvem um problema resultante, sem se darem ao trabalho de resolver todos os possíveis subproblemas menores relacionados.

Surpreendentemente, em alguns casos a estratégia funciona!

Sutilezas

Devemos ter cuidado para não presumir que a subestrutura ótima seja aplicável quando não é. Considere os dois problemas a seguir, nos quais temos um grafo dirigido $G = (V, E)$ e vértices $u, v \in V$.

Caminho mais curto não ponderado:[3] Encontrar um caminho de u para v que consista no menor número de arestas. Tal caminho deve ser simples, já que remover um ciclo de um caminho produz um caminho com menos arestas.

Caminho simples mais longo não ponderado: Encontrar um caminho simples de u para v que consista no maior número de arestas. Precisamos incluir o requisito de simplicidade porque, do contrário, acabamos percorrendo um ciclo tantas vezes quantas quisermos para criar caminhos com um número arbitrariamente grande de arestas.

O problema do caminho mais curto não ponderado exibe subestrutura ótima, como mostramos a seguir. Suponha que $u \neq v$, de modo que o problema é não trivial. Então, qualquer caminho p de u para v deve conter um vértice intermediário, digamos w. (Observe que w pode ser u ou v.) Assim, podemos decompor o caminho $u \; p \; v$ em subcaminhos $u \; p_1 \; w \; p_2 \; v$. É claro que o número de arestas em p é igual ao número de arestas em p_1 mais o número de arestas em p_2. Afirmamos que, se p é um caminho ótimo (isto é, o mais curto) de u para v, então p_1 deve ser um caminho mais curto de u para w. Por quê? Usamos um argumento de "recortar e colar": se existisse outro caminho, digamos de u para w com menos arestas que p'_1, poderíamos recortar p_1 e colar em p'_1 para produzir um caminho $u \; p'_1 \; w \; p_2 \; v$ com menos arestas que p, assim contra-dizendo a otimalidade de p. Simetricamente, p_2 deve ser um caminho mais curto de w para v. Assim, podemos encontrar um caminho mais curto de u para v considerando todos os vértices intermediários w, encontrando um caminho mais curto de u para w e um caminho mais curto de w para v, e escolhendo um vértice intermediário w que produza o caminho mais curto global. Na Seção 25.2, usamos uma variante dessa observação de subestrutura ótima para encontrar um caminho mais curto entre cada par de vértices em um grafo ponderado e dirigido.

É tentador supor que o problema de encontrar um caminho simples mais longo não ponderado também exibe subestrutura ótima. Afinal, se desdobrarmos um caminho simples mais longo $u \overset{p}{\rightsquigarrow} v$ em subcaminhos $u \overset{p_1}{\rightsquigarrow} w \overset{p_2}{\rightsquigarrow} v$, então p não deve ser um caminho simples mais longo de u para w, e p_2 não deve ser um caminho simples mais longo de w para v? A resposta é não! A Figura 15.6 nos dá um exemplo. Considere o caminho $q \rightarrow r \rightarrow t$, que é um caminho simples mais longo de q para t. O caminho $q \rightarrow r$ é um caminho simples mais longo de q para r? Não, já que o caminho $q \rightarrow s \rightarrow t \rightarrow r$ é um caminho simples mais longo. $r \rightarrow t$ é um caminho simples mais longo de r para t? Novamente não, já que o caminho $r \rightarrow q \rightarrow s \rightarrow t$ é um caminho simples mais longo.

Esse exemplo mostra que, para caminhos simples mais longos, não apenas falta uma subestrutura ótima para o problema, mas tampouco podemos montar necessariamente uma solução "legítima" para o problema a partir de soluções para subproblemas. Se combinarmos os caminhos simples mais longos $q \rightarrow s \rightarrow t \rightarrow r$ e $r \rightarrow q \rightarrow s \rightarrow t$, obteremos o caminho $q \rightarrow s \rightarrow t \rightarrow r \rightarrow q \rightarrow s \rightarrow t$, que não é simples. Na realidade, o problema de encontrar um caminho simples mais longo não ponderado não parece ter nenhum tipo de subestrutura ótima. Nenhum algoritmo eficiente de programação dinâmica foi encontrado para esse problema até hoje. De fato, esse problema é NP-completo, que — como veremos no Capítulo 34 — significa que é improvável que ele possa ser resolvido em tempo polinomial.

Por que a subestrutura de um caminho simples mais longo é tão diferente da subestrutura de um caminho mais curto? Embora uma solução para um problema para ambos, o caminho mais longo e o mais curto, use dois subproblemas, os subproblemas para encontrar o caminho simples mais longo não são ***independentes***, mas o são para caminhos mais curtos. O que significa subproblemas independentes? Significa que a solução para um subproblema não afeta a solução para outro subproblema do mesmo problema. No exemplo da Figura 15.6, temos o problema de encontrar um caminho simples mais longo de q para t com dois subproblemas: encontrar caminhos simples mais longos de q para r e de r para t. Para o primeiro desses subproblemas, escolhemos o caminho $q \rightarrow s \rightarrow t \rightarrow r$ e, portanto, também usamos os vértices s e t. Não podemos mais usar esses vértices no segundo subproblema, já que a combinação das duas soluções para subproblemas produziria um caminho que não é simples. Se não podemos usar o vértice t no segundo problema, não podemos resolvê-lo, já que t tem de estar no caminho que encontrarmos, e ele não é o vértice no qual estamos "encaixando" as soluções do subproblema (esse vértice é r). Como usamos os vértices s e t em uma solução de subproblema, não podemos usá-los na solução do outro subproblema. Porém, temos de usar, no mínimo, um deles para resolver o outro subproblema e temos de usar ambos para resolvê-lo otimamente. Assim, dizemos que esses subproblemas não são independentes. Visto de outro modo, usar recursos para resolver um subproblema (sendo esses recursos os vértices) torna-os indisponíveis para o outro subproblema.

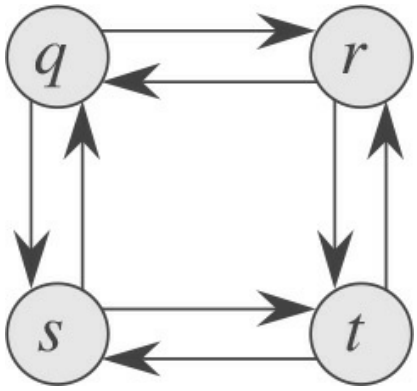


Figura 15.6 Um grafo dirigido mostrando que o problema de encontrar um caminho simples mais longo em um grafo dirigido não ponderado não tem subestrutura ótima. O caminho $q \to r \to t$ é um caminho simples mais longo de q para t, mas o subcaminho $q \to r$ não é um caminho simples mais longo de q para r nem o subcaminho $r \to t$ é um caminho simples mais longo de r para t.

Então, por que os subproblemas são independentes para encontrar um caminho mais curto? A resposta é que, por natureza, os subproblemas não compartilham recursos. Afirmamos que, se um vértice w está em um caminho mais curto p de u para v, então podemos emendar *qualquer* caminho mais curto $u \overset{p_1}{\rightsquigarrow} w$ e *qualquer* caminho mais curto $w \overset{p_2}{\rightsquigarrow} v$ para produzir um caminho mais curto de u para v. Estamos seguros de que, além de w, nenhum vértice pode aparecer nos caminhos p_1 e p_2. Por quê? Suponha que algum vértice $x \neq w$ apareça tanto em p_1 quanto em p_2, de modo que podemos decompor p_1 como $u \overset{p_{ux}}{\rightsquigarrow} x \rightsquigarrow w$ e p_2 como $w \overset{p_{xv}}{\rightsquigarrow} v$. Pela subestrutura ótima desse problema, o caminho p tem tantas arestas quanto p_1 e p_2 juntos; vamos dizer que p tenha e arestas. Agora, vamos construir um caminho $p' = u\, p_{ux}\, w\, p_{xv}\, v$ de u para v. Como cortamos os caminhos de x para w e de w para x e cada um deles contém no mínimo uma aresta, o caminho p_0 contém no máximo $e - 2$ arestas, o que contradiz a hipótese de p ser um caminho mais curto. Assim, estamos seguros de que os subproblemas para o problema do caminho mais curto são independentes.

Ambos os problemas examinados nas Seções 15.1 e 15.2 têm subproblemas independentes. Na multiplicação de cadeias de matrizes, os subproblemas são multiplicar subcadeias $A_i A_{i+1} \cdots A_k$ e $A_{k+1} A_{k+2} \cdots A_j$. Essas subcadeias são disjuntas, de modo que não haveria possibilidade de alguma matriz ser incluída em ambas. No corte de hastes, para determinar o melhor modo de cortar uma haste de comprimento n, consideramos os melhores modos de cortar hastes de comprimento i para $i = 0, 1, \ldots, n - 1$. Como uma solução ótima para o problema do comprimento n inclui apenas uma dessas soluções de subproblemas (após termos cortado o primeiro pedaço), a independência de subproblemas nem entra no assunto.

Subproblemas sobrepostos

O segundo elemento que um problema de otimização deve ter para a programação dinâmica ser aplicável é que o espaço de subproblemas deve ser "pequeno", no sentido de que um algoritmo recursivo para o problema resolve os mesmos subproblemas repetidas vezes, em lugar de sempre gerar novos subproblemas. Em geral, o número total de subproblemas distintos é um polinômio no tamanho de entrada. Quando um algoritmo recursivo reexamina o mesmo problema repetidamente, dizemos que o problema de otimização tem ***subproblemas sobrepostos***.[4] Ao contrário, um problema para o qual uma abordagem de divisão e conquista é adequada, normalmente gera problemas absolutamente novos em cada etapa da recursão. Algoritmos de programação dinâmica, normalmente tiram proveito de subproblemas sobrepostos resolvendo cada subproblema uma vez e depois armazenando a solução em uma tabela onde ela pode ser examinada quando necessário, usando um tempo constante por busca.

Na Seção 15.1, examinamos brevemente como uma solução recursiva para o problema do corte de hastes faz exponencialmente muitas chamadas para encontrar soluções para problemas menores. Nossa solução de programação dinâmica toma um algoritmo recursivo de tempo exponencial e o reduz a um algoritmo de tempo linear.

Para ilustrar a propriedade de subproblemas sobrepostos com mais detalhes, vamos examinar novamente o problema de multiplicação de cadeias de matrizes. Consultando mais uma vez a Figura 15.5, observe que MATRIX-CHAIN-ORDER consulta repetidamente a solução para subproblemas em linhas inferiores quando está resolvendo subproblemas em linhas superiores. Por exemplo, referencia a entrada $m[3, 4]$ quatro vezes: durante os cálculos de $m[2, 4]$, $m[1, 4]$, $m[3, 5]$ e $m[3, 6]$. Se tivéssemos de recalcular $m[3, 4]$ toda vez, em vez de apenas consultá-la, o tempo de execução aumentaria expressivamente. Para ver como, considere o seguinte procedimento recursivo (ineficiente) que determina $m[i, j]$, o número mínimo de multiplicações escalares necessárias para calcular o produto de cadeias de matrizes $A_{i..j} = A_i A_{i+1} \cdots A_j$. O procedimento é diretamente baseado na recorrência (15.7).

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
1  if i == j
2     return 0
3  m[i, j] = ∞
4  for k = i to j − 1
5     q = RECURSIVE-MATRIX-CHAIN(p, i, k)
          + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
          + p_{i−1} p_k p_j
6     if q < m[i, j]
7        m[i, j] = q
8  return m[i, j]
```

A Figura 15.7 mostra a árvore de recursão produzida pela chamada RECURSIVE-MATRIX--CHAIN(p, 1, 4). Cada nó é identificado pelos valores dos parâmetros i e j. Observe que alguns pares de valores ocorrem muitas vezes.

De fato, podemos mostrar que o tempo para calcular $m[1, n]$ por esse procedimento recursivo é no mínimo exponencial em n. Seja $T(n)$ o tempo tomado por RECURSIVE-MATRIX-CHAIN para calcular uma parentização ótima de uma cadeia de n matrizes. Como cada uma das execuções das linhas 1-2 e das linhas 6-7 demora no mínimo tempo unitário, assim como a multiplicação na linha 5, a inspeção do procedimento produz a recorrência

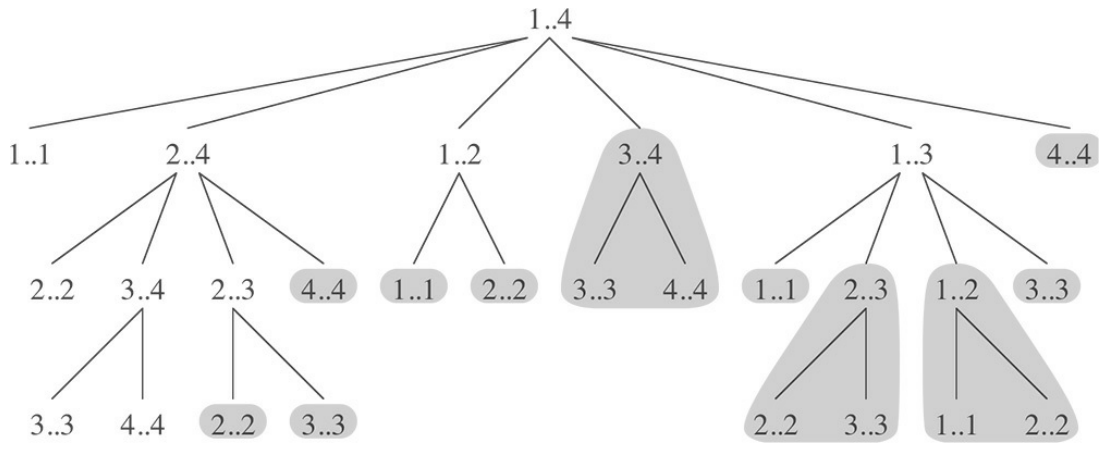


Figura 15.7 A árvore de recursão para a execução de RECURSIVE-MATRIX-CHAIN(p, 1, 4). Cada nó contém os parâmetros i e j. Os cálculos executados em uma subárvore sombreada são substituídos por uma única consulta à tabela em MEMOIZED-MATRIX-CHAIN.

$$T(1) \geq 1,$$
$$T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \quad \text{para } n > 1.$$

Observando que, para $i = 1, 2, ..., n - 1$, cada termo $T(i)$ aparece uma vez como $T(k)$ e uma vez como $T(n - k)$, e reunindo os $n - 1$ valores 1 no somatório com o valor 1 à frente, podemos

$$T(n) \geq 2\sum_{i=1}^{n-1} T(i) + n. \tag{15.8}$$

Provaremos que $T(n) = \Omega(2_n)$ usando o método de substituição. Especificamente, mostraremos que $T(n) \geq 2_n - 1$ para todo $n \geq 1$. A base é fácil, já que $T(1) \geq 1 = 2^0$. Por indução, para $n \geq 2$, temos

$$\begin{aligned} T(n) &\geq 2\sum_{i=1}^{n-1} 2^{i-1} + n \\ &= 2\sum_{i=0}^{n-2} 2^i + n \\ &= 2(2^{n-1} - 1) + n \quad \text{(pela equação (A.5))} \\ &= 2^n - 2 + n \\ &\geq 2^{n-1}, \end{aligned}$$

o que conclui a prova. Assim, a quantidade total de trabalho executado pela chamada RECURSIVE-MATRIX-CHAIN(p, 1, n) é no mínimo exponencial em n.

Compare esse algoritmo recursivo de cima para baixo (sem memoização) com o algoritmo de programação dinâmica de baixo para cima. Este último é mais eficiente porque tira proveito da propriedade de subproblemas sobrepostos. A multiplicação de cadeias de matrizes tem somente $Q(n_2)$ subproblemas distintos, e o algoritmo de programação dinâmica resolve cada um deles exatamente uma vez. Por outro lado, novamente o algoritmo recursivo deve resolver cada subproblema toda vez que ele reaparece na árvore de recursão. Sempre que uma árvore de recursão para a solução recursiva natural para um problema contiver o mesmo subproblema repetidamente e o número total de subproblemas distintos for pequeno, a programação dinâmica pode melhorar a eficiência, às vezes, expressivamente.

Reconstrução de uma solução ótima

Como regra prática, muitas vezes, armazenamos em uma tabela a opção que escolhemos em cada subproblema, de modo que não tenhamos de reconstruir essa informação com base nos custos que armazenamos.

Na multiplicação de cadeias de matrizes, a tabela $s[i, j]$ poupa uma quantidade significativa de trabalho durante a reconstrução de uma solução ótima. Suponha que não mantivéssemos a tabela $s[i, j]$, tendo preenchido apenas a tabela $m[i, j]$, que contém custos de subproblemas ótimos. Escolhemos entre $j - i$ possibilidades quando determinamos quais subproblemas usar em uma solução ótima para parentizar $A_i A_{i+1} \cdots A_j$, e $j - i$ não é uma constante. Portanto, demoraria o tempo $Q(j - i) = \omega(1)$ para reconstruir os subproblemas que escolhemos para uma solução de um problema dado. Armazenando em $s[i, j]$ o índice da matriz nos quais separamos o produto $A_i A_{i+1} \cdots A_j$, podemos reconstruir cada escolha no tempo $O(1)$.

Memoização

Como vimos no problema do corte de hastes, há uma abordagem alternativa para a programação dinâmica que frequentemente oferece a eficiência da abordagem de programação dinâmica de baixo para cima e ao mesmo tempo mantém uma estratégia de cima para baixo. A ideia é ***memoizar*** o algoritmo recursivo natural mas ineficiente. Como na abordagem de baixo para cima, mantemos uma tabela com soluções de subproblemas, mas a estrutura de controle para preencher a tabela é mais semelhante ao algoritmo recursivo.

Um algoritmo recursivo memoizado mantém uma entrada em uma tabela para a solução de cada subproblema. Cada entrada da tabela contém inicialmente um valor especial para indicar que a entrada ainda tem de ser preenchida. Quando o subproblema é encontrado pela primeira vez durante a execução do algoritmo recursivo, sua solução é calculada e depois armazenada na tabela. Cada vez subsequente que encontrarmos esse subproblema, simplesmente consultamos o valor armazenado na tabela e o devolvemos.[5]

Apresentamos a seguir, uma versão memoizada de RECURSIVE-MATRIX-CHAIN. Observe os pontos de semelhança com o método de cima para baixo memoizado para o problema do corte de hastes.

```
MEMOIZED-MATRIX-CHAIN (p)
1  n = p.comprimento − 1
2  seja m[1 .. n, 1 .. n] uma nova tabela
3  for i = 1 to n
4      for j = i to n
5          m[i, j] = ∞
6  return LOOKUP-CHAIN (m, p, 1, n)

LOOKUP-CHAIN (m, p, i, j)
1  if m[i, j] < ∞
2      return m[i, j]
3  if i == j
4      m[i, j] = 0
5  else for k = i to j − 1
6          q = LOOKUP-CHAIN (m, p, i, k)
               + LOOKUP-CHAIN (m, p, k + 1, j) + p_{i−1} p_k p_j
7          if q < m[i, j]
8              m[i, j] = q
9  return m[i, j]
```

O procedimento MEMOIZED-MATRIX-CHAIN, assim como o procedimento MATRIX-CHAIN-ORDER, mantém uma tabela $m[1..n, 1..n]$ de valores calculados de $m[i, j]$, o número mínimo de multiplicações escalares necessárias para calcular a matriz $A_{i..j}$. Cada entrada de tabela contém inicialmente o valor ∞ para indicar que a entrada ainda tem de ser preenchida. Quando a chamada LOOKUP-CHAIN(p, i, j) é executada, se a linha 1 verificar que $m[i, j] < \infty$, o procedimento simplesmente retorna o custo $m[i, j]$ calculado anteriormente na linha 2. Caso contrário, o custo é calculado como em RECURSIVE-MATRIX-CHAIN, armazenado em $m[i, j]$ e retornado. Assim, LOOKUP-CHAIN(p, i, j) sempre retorna o valor de $m[i, j]$, mas só o calcula na primeira chamada de LOOKUP-CHAIN que tenha esses valores específicos de i e j.

A Figura 15.7 ilustra como MEMOIZED-MATRIX-CHAIN poupa tempo em comparação com RECURSIVE-MATRIX-CHAIN. As subárvores sombreadas representam valores que o procedimento consulta em vez de recalcular.

Do mesmo modo que o algoritmo de programação dinâmica de baixo para cima MATRIX-CHAIN-ORDER, o procedimento MEMOIZED-MATRIX-CHAIN é executado em tempo $O(n_3)$. A linha 5 de MEMOIZED-MATRIX-CHAIN é executada $O(n_2)$ vezes. Podemos classificar as chamadas de LOOKUP-CHAIN em dois tipos:

1. Chamadas nas quais $mi, j = \infty$, de modo que as linhas 3-9 são executadas.
2. Chamadas nas quais $mi, j < \infty$, de modo que LOOKUP-CHAIN simplesmente retorna na linha 2.

Há $Q(n_2)$ chamadas do primeiro tipo, uma por entrada de tabela. Todas as chamadas do segundo tipo são feitas como chamadas recursivas por chamadas do primeiro tipo. Sempre que uma dada chamada de LOOKUP-CHAIN faz chamadas recursivas, ela faz $O(n)$ chamadas. Assim, há ao todo $O(n_3)$ chamadas do segundo tipo. Cada chamada do segundo tipo demora o tempo $O(1)$, e cada chamada do primeiro tipo demora o tempo $O(n)$ mais o tempo gasto em suas chamadas recursivas. Portanto, o tempo total é $O(n_3)$. Assim, a memoização transforma um algoritmo de tempo (2_n) em um algoritmo de tempo $O(n_3)$.

Resumindo, podemos resolver o problema de multiplicação de cadeias de matrizes no tempo $O(n_3)$ por um algoritmo de programação dinâmica de cima para baixo memoizado ou por um algoritmo de programação dinâmica de baixo para cima. Ambos os métodos tiram proveito da propriedade dos subproblemas sobrepostos. Há apenas $Q(n_2)$ subproblemas distintos no total e qualquer um desses métodos calcula a solução para cada subproblema somente uma vez. Sem memoização, o algoritmo recursivo natural é executado em tempo exponencial, já que subproblemas resolvidos são resolvidos repetidas vezes.

Na prática geral, se todos os subproblemas devem ser resolvidos no mínimo uma vez, o desempenho de um algoritmo de programação dinâmica de baixo para cima, normalmente supera o de um algoritmo de cima para baixo memoizado por um fator constante porque o algoritmo de baixo para cima não tem nenhuma sobrecarga para recursão e a sobrecarga associada à manutenção da tabela é menor. Além disso, em alguns problemas podemos explorar o padrão regular de acessos a tabelas no algoritmo de programação dinâmica para reduzir ainda mais os requisitos de tempo ou espaço. Alternativamente, se alguns subproblemas no espaço de subproblemas não precisarem ser resolvidos de modo algum, a solução memoizada tem a vantagem de resolver somente os subproblemas que são definitivamente necessários.

Exercícios

15.3-1 Qual modo é mais eficiente para determinar o número ótimo de multiplicações em um problema de multiplicação de cadeias de matrizes: enumerar todos os modos de parentizar o produto e calcular o número de multiplicações para cada um ou executar RECURSIVE-MATRIX-CHAIN? Justifique sua resposta.

15.3-2 Desenhe a árvore de recursão para o procedimento MERGE-SORT da Seção 2.3.1 em um arranjo de 16 elementos. Explique por que a memoização não aumenta a velocidade de um bom algoritmo de divisão e conquista como MERGE-SORT.

15.3-3 Considere uma variante do problema da multiplicação de cadeias de matrizes na qual a meta é parentizar a sequência de matrizes de modo a maximizar, em vez de minimizar, o número de multiplicações escalares. Esse problema exibe subestrutura ótima?

15.3-4 Como já dissemos, em programação dinâmica primeiro resolvemos os subproblemas e depois escolhemos qual deles utilizar em uma solução ótima para o problema. A professora Capulet afirma que nem sempre é necessário resolver todos os subproblemas para encontrar uma solução ótima. Ela sugere que podemos encontrar uma solução ótima para o problema de multiplicação de cadeias de matrizes escolhendo sempre a matriz A_k na qual separar o subproduto $A_i A_{i+1} \cdots A_j$ (selecionando k para minimizar a quantidade $p_i - 1 p_k p_j$) *antes* de resolver os subproblemas. Determine uma instância do problema de multiplicação de cadeias de matrizes para a qual essa abordagem gulosa produz uma solução subótima.

15.3-5 Suponha que, no problema do corte de hastes da Seção 15.1, tivéssemos também o limite l_i para o número de peças de comprimento i que era permitido produzir, para $i = 1, 2,...,n$. Mostre que a propriedade de subestrutura ótima descrita na Seção 15.1 deixa de ser válida.

15.3-6 Imagine que você queira fazer o câmbio de uma moeda por outra e percebe que, em vez de trocar diretamente uma moeda por outra, seria melhor efetuar uma série de trocas intermediárias por outras moedas, por fim tendo em mãos a moeda que queria. Suponha que você possa trocar n moedas diferentes, numeradas de 1,2,...,n, que começará com a moeda 1 e quer terminar com a moeda n. Você tem, para cada par de moedas i e j, uma taxa de câmbio r_{ij} , o que significa que, se você começar com d unidades da moeda i, poderá trocá-las por dr_{ij} unidades da moeda j. Uma sequência de trocas pode acarretar uma comissão, que depende do número de trocas que você faz. Seja c_k a comissão cobrada quando você faz k trocas. Mostre que, se $c_k = 0$ para todo $k = 1,2,...,n$, o problema de determinar a melhor sequência de trocas da moeda 1 para a moeda n exibe subestrutura ótima. Então, mostre que, se as comissões c_k são valores arbitrários, o problema de determinar a melhor sequência de trocas da moeda 1 para a moeda n não exibe necessariamente subestrutura ótima.

15.4 Subsequência comum mais longa

Em aplicações biológicas, muitas vezes, é preciso comparar o DNA de dois (ou mais) organismos diferentes. Um filamento de DNA consiste em uma cadeia de moléculas denominadas ***bases***, na qual as bases possíveis são adenina, guanina, citosina e timina. Representando cada uma dessas bases por sua letra inicial, podemos expressar um filamento de DNA como uma cadeia no conjunto finito {A,C,G,T}. (O Apêndice C dá a definição de uma cadeia.) Por exemplo, o DNA de um organismo pode ser S_1 = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA, e o DNA de outro organismo pode ser S_2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA. Uma razão para a comparação de dois filamentos de DNA é determinar o grau de "semelhança" entre eles, que serve como alguma medida da magnitude da relação entre os dois organismos. Podemos definir (e definimos) a semelhança de muitas maneiras diferentes. Por exemplo, podemos dizer que dois filamentos de DNA são semelhantes se um deles for uma subcadeia do outro. (O Capítulo 32 explora algoritmos para resolver esse problema.) Em nosso exemplo, nem S_1 nem S_2 é uma subcadeia do outro. Alternativamente, poderíamos dizer que dois filamentos são semelhantes se o número de mudanças necessárias para transformar um no outro for pequeno. (O Problema 15-3 explora essa noção.) Ainda uma outra maneira de medir a semelhança entre filamentos S_1 e S_2 é encontrar um terceiro filamento S_3 no qual as bases em S_3 aparecem em cada um dos filamentos S_1 e S_2; essas bases devem aparecer na mesma ordem, mas não precisam ser necessariamente consecutivas. Quanto mais longo o filamento S_3 que pudermos encontrar, maior será a semelhança entre S_1 e S_2 . Em nosso exemplo, o filamento S_3 mais longo é GTCGTCGGAAGCCGGCCGAA.

Formalizamos essa última noção de semelhança como o problema da subsequência comum mais longa. Uma subsequência de uma determinada sequência é apenas a sequência dada na qual foram omitidos zero ou mais elementos. Em termos formais, dada uma sequência $X = \langle x_1, x_2, ..., x_m \rangle$, uma outra sequência $Z = \langle z_1, z_2, ..., z_k \rangle$ é uma ***subsequência*** de X se existir uma sequência estritamente crescente $\langle i_1, i_2, ..., i_k \rangle$ de índices de X tais que, para todo $j = 1, 2, ..., k$, temos $x_i = z_j$. Por exemplo, $Z = \langle B, C, D, B \rangle$ é uma subsequência de $X = \langle A, B, C, B, D, A, B \rangle$ com sequência de índices correspondente $\langle 2, 3, 5, 7 \rangle$.

Dadas duas sequências X e Y, dizemos que uma sequência Z é uma ***subsequência comum*** de X e Y se Z é uma subsequência de X e Y. Por exemplo, se $X = \langle A, B, C, B, D, A, B \rangle$ e $Y = \langle B, D, C, A, B, A \rangle$, a sequência $\langle B, C, A \rangle$ é uma subsequência comum das sequências X e Y. Porém, a sequência $\langle B, C, A \rangle$ não é uma subsequência comum *mais longa* (LCS — longest common subsequence) de X e Y, já que tem comprimento 3, e a sequência $\langle B, C, B, A \rangle$, que também é comum a X e Y, tem comprimento 4. A sequência $\langle B, C, B, A \rangle$ é uma LCS de X e Y, assim como a sequência $\langle B, D, A, B \rangle$, visto que não existe nenhuma subsequência comum de comprimento 5 ou maior.

No ***problema da subsequência comum mais longa***, temos duas sequências $X = \langle x_1, x_2, ..., x_m \rangle$ e $Y = \langle y_1, y_2, ..., y_n \rangle$, e desejamos encontrar uma subsequência comum de comprimento máximo de X e Y. Esta seção mostra como resolver o problema da LCS eficientemente, usando programação dinâmica.

Etapa 1: Caracterização de uma subsequência comum mais longa

Uma abordagem de força bruta para resolver o problema da LCS seria enumerar todas as subsequências de X e conferir cada subsequência para ver se ela também é uma subsequência de Y, sem perder de vista a subsequência mais longa encontrada. Cada subsequência de X corresponde a um subconjunto dos índices $\{1, 2, ..., m\}$ de X. Como X tem 2^m subsequências, essa abordagem requer tempo exponencial, o que a torna impraticável para sequências longas.

Porém, o problema da LCS tem uma propriedade de subestrutura ótima, como mostra o teorema a seguir. Como veremos, as classes naturais de subproblemas correspondem a pares de "prefixos" das duas sequências de entrada. Mais precisamente, dada uma sequência $X = \langle x_1, x_2, .., x_m \rangle$, definimos o i-ésimo ***prefixo*** de X, para $i = 0, 1, ..., m$, como $X_i = \langle x_1, x_2, ..., x_i \rangle$. Por exemplo, se $X = \langle A, B, C, B, D, A, B \rangle$, então $X_4 = \langle A, B, C, B \rangle$ e X_0 é a sequência vazia.

Teorema 15.1 (Subestrutura ótima de uma LCS)

Sejam $X = \langle x_1, x_2, ..., x_m \rangle$ e $Y = \langle y_1, y_2, ..., y_n \rangle$ as sequências, e seja $Z = \langle z_1, z_2, ..., z_k \rangle$ qualquer LCS de X e Y.

1. Se $x_m = y_n$, então $z_k = x_m = y_n$ e Z_{k-1} é uma LCS de X_{m-1} e Y_{n-1}.
2. Se $x_m \neq y_n$, então $z_k \neq x_m$ implica que Z é uma LCS de X_{m-1} e Y.
3. Se $x_m \neq y_n$, então $z_k \neq y_m$ implica que Z é uma LCS de X e Y_{n-1}.

Prova (1) Se $z_k \neq x_m$, então podemos anexar $x_m = y_n$ a Z para obter uma subsequência comum de X e Y de comprimento $k + 1$, contradizendo a suposição de que Z é uma subsequência comum *mais longa* de X e Y. Assim, devemos ter $z_k = x_m = y_n$. Agora, o prefixo Z_{k-1} é uma subsequência comum de comprimento $(k - 1)$ de X_{m-1} e Y_{n-1}. Desejamos mostrar que ela é uma LCS. Suponha, por contradição, que exista uma subsequência comum W de X_{m-1} e Y_{n-1} com comprimento maior que $k - 1$. Então, anexar $x_m = y_n$ a W produz uma subsequência comum de X e Y cujo comprimento é maior que k, o que é uma contradição.

(2) Se $z_k \neq x_m$, então Z é uma subsequência comum de X_{m-1} e Y. Se existisse uma subsequência comum W de X_{m-1} e Y com comprimento maior que k, então W seria também uma subsequência comum de X_m e Y, contradizendo a suposição de que Z é uma LCS de X e Y.

(3) A prova é simétrica a (2).

O modo como o Teorema 15.1 caracteriza subsequências comuns mais longas nos diz que uma LCS de duas sequências contém uma LCS de prefixos das duas sequências. Assim, o problema de LCS tem uma propriedade de subestrutura ótima. Uma solução recursiva também tem a propriedade de subproblemas sobrepostos, como veremos em breve.

Etapa 2: Uma solução recursiva

O Teorema 15.1 subentende que devemos examinar um ou dois subproblemas quando queremos encontrar uma LCS de $X = \langle x_1, x_2, ..., x_m \rangle$ e $Y = \langle y_1, y_2, ..., y_n \rangle$. Se $x_m = y_n$, devemos encontrar uma LCS de X_{m-1} e Y_{n-1}. Anexar $x_m = y_n$ a essa LCS produz uma LCS de X e Y. Se $x_m \neq y_n$, então devemos resolver dois subproblemas: encontrar uma LCS de X_{m-1} e Y e encontrar uma LCS de X e Y_{n-1}. A mais longa de qualquer dessas duas LCS é uma LCS de X e Y. Como esses casos esgotam todas as possibilidades, sabemos que uma das soluções ótimas de subproblemas certamente aparecerá dentro de uma LCS de X e Y.

É fácil ver a propriedade de subproblemas sobrepostos no problema da LCS. Para encontrar uma LCS de X e Y, pode ser necessário encontrar as LCS de X e Y_{n-1} e de X_{m-1} e Y. Porém, cada um desses subproblemas tem o subsubproblema de encontrar uma LCS de X_{m-1} e Y_{n-1}. Muitos outros subproblemas compartilham subsubproblemas.

Como ocorreu no problema de multiplicação de cadeias de matrizes, nossa solução recursiva para o problema da LCS envolve estabelecer uma recorrência para o valor de uma solução ótima. Vamos definir $c[i, j]$ como o

comprimento de uma LCS das sequências X_i e Y_j. Se $i = 0$ ou $j = 0$, uma das sequências tem comprimento 0 e, portanto, a LCS tem comprimento 0. A subestrutura ótima do problema da LCS dá a fórmula recursiva

$$c[i,j] = \begin{cases} 0 & \text{se } i = 0 \text{ ou } j = 0, \\ c[i-1,j-1]+1 & \text{se } i,j > 0 \text{ e } x_i = y_j, \\ \max(c[i,j-1], c[i-1,\ j]) & \text{se } i,j > 0 \text{ e } x_i \neq y_j. \end{cases} \tag{15.9}$$

Observe que, nessa formulação recursiva, uma condição no problema restringe os subproblemas que podemos considerar. Quando $x_i = y_j$, podemos e devemos considerar o subproblema de encontrar a LCS de X_i^{-1} e Y_j^{-1}. Caso contrário, consideramos os dois subproblemas de encontrar uma LCS de X_i e Y_j^{-1} e de X_i^{-1} e Y_j. Nos algoritmos de programação dinâmica que já examinamos — para corte de hastes e para multiplicação de cadeias de matrizes —, não descartamos nenhum subproblema por causa de condições no problema. O algoritmo para encontrar uma LCS não é o único algoritmo de programação dinâmica que descarta subproblemas com base em condições no problema. Por exemplo, o problema da distância de edição (ver o Problema 15-3) tem essa característica.

Etapa 3: Cálculo do comprimento de uma LCS

Tendo como base a equação (15.9), seria fácil escrever um algoritmo recursivo de tempo exponencial para calcular o comprimento de uma LCS de duas sequências. Contudo, visto que o problema da LCS tem somente $Q(mn)$ subproblemas distintos, podemos usar programação dinâmica para calcular as soluções de baixo para cima.

O procedimento LCS-LENGTH toma duas sequências $X = \langle x_1, x_2, ..., x_m \rangle$ e $Y = \langle y_1, y_2, ..., y_n \rangle$ como entradas. Armazena os valores $c[i, j]$ em uma tabela $c[0 .. m, 0 .. n]$ e calcula as entradas em ***ordem orientada por linha*** (isto é, preenche a primeira linha de c da esquerda para a direita, depois a segunda linha, e assim por diante). O procedimento também mantém a tabela $b[1 .. m, 1 .. n]$ para ajudar a construir uma solução ótima. Intuitivamente, $b[i, j]$ aponta para a entrada da tabela correspondente à solução ótima de subproblema escolhida ao se calcular $c[i, j]$. O procedimento retorna as tabelas b e c; $c[m, n]$ contém o comprimento de uma LCS de X e Y.

```
LCS-LENGTH (X, Y)
1   m = X.comprimento
2   n = Y.comprimento
3   sejam b[1..m, 1..n] e c[0..m, 0..n] tabelas novas
4   for i = 1 to m
5       c[i,0] = 0
6   for j = 0 to n
7       c[0, j] = 0
8   for i = 1 to m
9       for j = 1 to n
10          if x_i == y_j
11              c[i, j] = c[i − 1, j − 1] + 1
12              b[i, j] = "↖"
13          elseif c[i − 1, j] ≥ c[i, j − 1]
14              c[i, j] = c[i − 1, j]
15              b[i, j] = "↑"
16          else c[i, j] = c[i, j- 1]
17              b[i, j] = "←"
18  return c, b
```

A Figura 15.8 mostra as tabelas produzidas por LCS-LENGTH nas sequências $X = \langle A, B, C, B, D, A, B \rangle$ e $Y = \langle B, D, C, A, B, A \rangle$. O tempo de execução do procedimento é $\Theta(mn)$, já que cada entrada de tabela demora o tempo $\Theta(1)$ para ser calculada.

Etapa 4: Construção de uma LCS

A tabela b retornada por LCS-LENGTH nos habilita a construir rapidamente uma LCS de $X = \langle x_1, x_2, ..., x_m \rangle$ e $Y = \langle y_1, y_2, ..., y_n \rangle$. Simplesmente começamos em $b[m, n]$ e percorremos a tabela seguindo as setas. Sempre que encontramos uma "↖" na entrada $b[i, j]$, ela implica que $x_i = y_j$ é um elemento da LCS que LCS-LENGTH encontrou. Com esse método, encontramos os elementos da LCS em ordem inversa. O procedimento recursivo a seguir imprime uma LCS de X e Y na ordem direta adequada. A invocação inicial é PRINT-LCS(b, X, *X.comprimento*, *Y.comprimento*).

```
PRINT-LCS(b, X, i, j)
1   if i == 0 ou j == 0
2      return
3   if b[i, j] == "↖"
4      PRINT-LCS(b, X, i − 1, j − 1)
5      print x_i
6   elseif b[i, j] == "↑"
7      PRINT-LCS(b, X, i − 1, j)
8   else PRINT-LCS(b, X, i, j − 1)
```

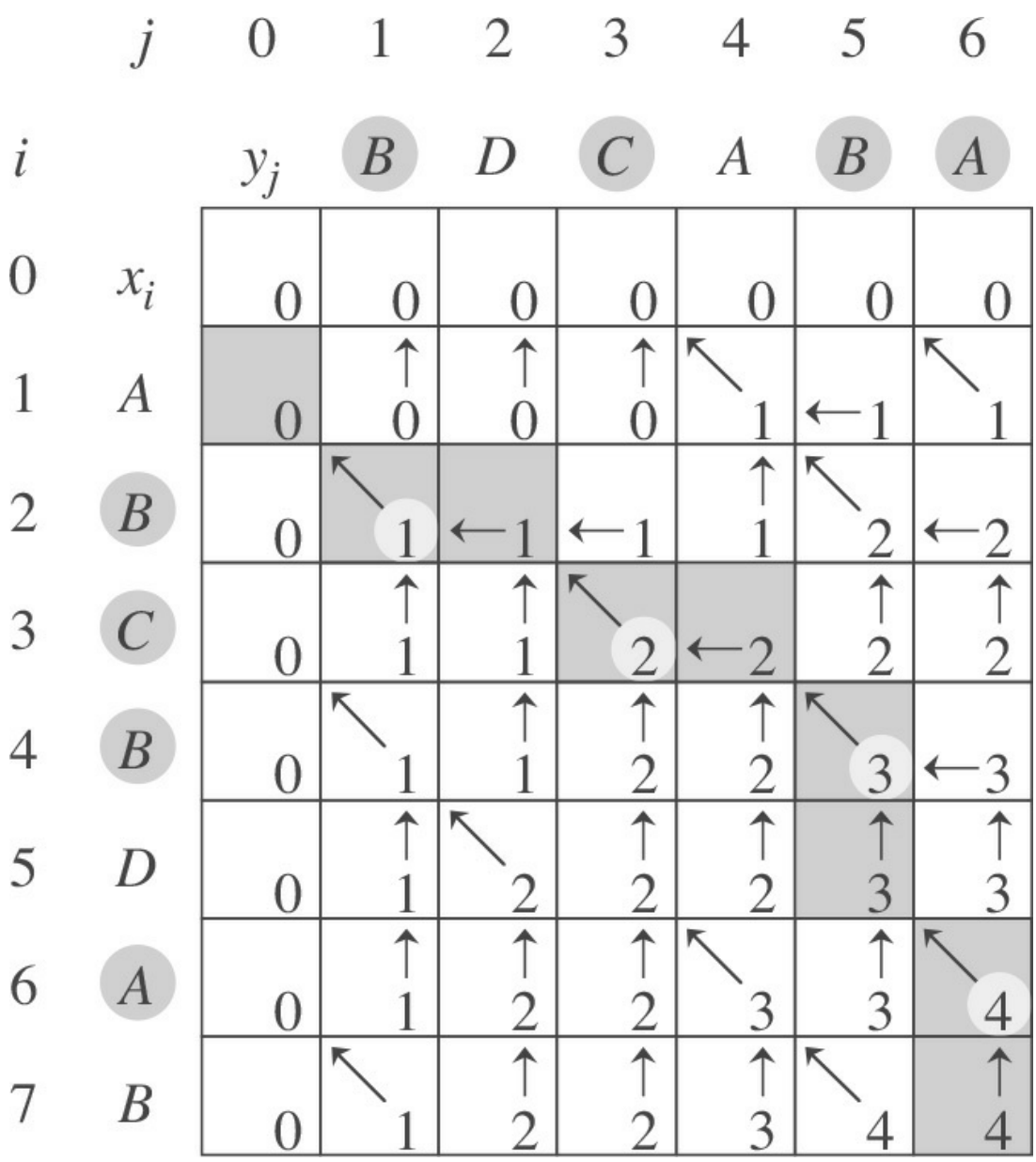


Figura 15.8 As tabelas c e b calculadas por LCS-LENGTH para as sequências $X = \langle A, B, C, \text{B, D, A, B} \rangle$ e $Y = \langle B, D, C, A, B, A \rangle$. O quadrado na linha i e coluna j contém o valor de ci, j e a seta adequada para o valor de bi, j. A entrada 4 em $c7, 6$ — o canto inferior direito da tabela — é o comprimento de uma LCS $\langle B, C, B, A \rangle$ de X e Y. Para $i, j > 0$, a entrada ci, j depende apenas de $x_i = y_j$ e dos valores nas entradas $ci - 1, j$, $ci, j - 1$ e $ci - 1, j - 1$, que são calculados antes de ci, j. Para reconstruir os elementos de uma LCS, siga as setas bi, j desde o canto inferior direito; a sequência está sombreada. Cada "\" na sequência sombreada corresponde a uma entrada (destacada) para a qual $x_i = y_j$ é membro de uma LCS.

Para a tabela b na Figura 15.8, esse procedimento imprime $BCBA$. O procedimento demora o tempo $O(m + n)$, já que decrementa no mínimo um de i e j em cada fase da recursão.

Melhorando o código

Depois de ter desenvolvido um algoritmo, você, muitas vezes, constatará que é possível melhorar o tempo ou o espaço que ele utiliza. Algumas mudanças podem simplificar o código e melhorar fatores constantes, porém, quanto ao

mais, não produzem nenhuma melhora assintótica no desempenho. Outras podem resultar em economias assintóticas significativas de tempo e de espaço.

Por exemplo, no algoritmo LCS podemos eliminar totalmente a tabela b. Cada entrada $c[i, j]$ depende apenas de três outras entradas na tabela $c[i - 1, j - 1]$, $c[i - 1, j]$ e $c[i, j - 1]$. Dado o valor de $c[i, j]$, podemos determinar em tempo $O(1)$ de qual desses três valores foi usado para calcular $c[i, j]$, sem inspecionar a tabela b. Assim, podemos reconstruir uma LCS em tempo $O(m + n)$ usando um procedimento semelhante a PRINT-LCS. (O Exercício 15.4-2 pede que você dê o pseudocódigo.) Embora economizemos espaço $Q(mn)$ por esse método, o requisito de espaço auxiliar para calcular uma LCS não diminui assintoticamente, já que, de qualquer modo, precisamos do espaço $Q(mn)$ para a tabela c.

Entretanto, podemos reduzir os requisitos de espaço assintótico para LCS-LENGTH, já que esse procedimento só precisa de duas linhas da tabela c por vez: a linha que está sendo calculada e a linha anterior. (De fato, como o Exercício 15.4-4 pede que você mostre, podemos usar apenas um pouquinho mais que o espaço para uma linha de c para calcular o comprimento de uma LCS.) Esse aperfeiçoamento funciona se necessitamos apenas do comprimento de uma LCS; se >precisarmos reconstruir os elementos de uma LCS, a tabela menor não guardará informações suficientes para reconstituir nossas etapas no tempo $O(m + n)$.

Exercícios

15.4-1 Determine uma LCS de $\langle 1, 0, 0, 1, 0, 1, 0, 1\rangle$ e $\langle 0, 1, 0, 1, 1, 0, 1, 1, 0\rangle$.

15.4-2 Dê o pseudocódigo para reconstruir uma LCS partindo da tabela c concluída e das sequências originais $X = \langle x_1, x_2, ..., x_m\rangle$ e $Y = \langle y_1, y_2, ..., y_n\rangle$ em tempo $O(m + n)$, sem usar a tabela b.

15.4-3 Dê uma versão memoizada de LCS-LENGTH que seja executada no tempo $O(mn)$.

15.4-4 Mostre como calcular o comprimento de uma LCS usando apenas $2 \cdot \min(m, n)$ entradas na tabela c mais o espaço adicional $O(1)$. Em seguida, mostre como fazer a mesma coisa usando $\min(m, n)$ entradas mais o espaço adicional $O(1)$.

15.4-5 Dê um algoritmo de tempo $O(n_2)$ para encontrar a subsequência monotonicamente crescente mais longa de uma sequência de n números.

15.4-6 ★ Dê um algoritmo de tempo $O(n \lg n)$ para encontrar a subsequência mais longa monotonicamente crescente de uma sequência de n números. (*Sugestão*: Observe que o último elemento de uma subsequência candidata de comprimento i é no mínimo tão grande quanto o último elemento de uma subsequência candidata de comprimento $i - 1$. Mantenha as subsequências candidatas ligando-as por meio da sequência de entrada.)

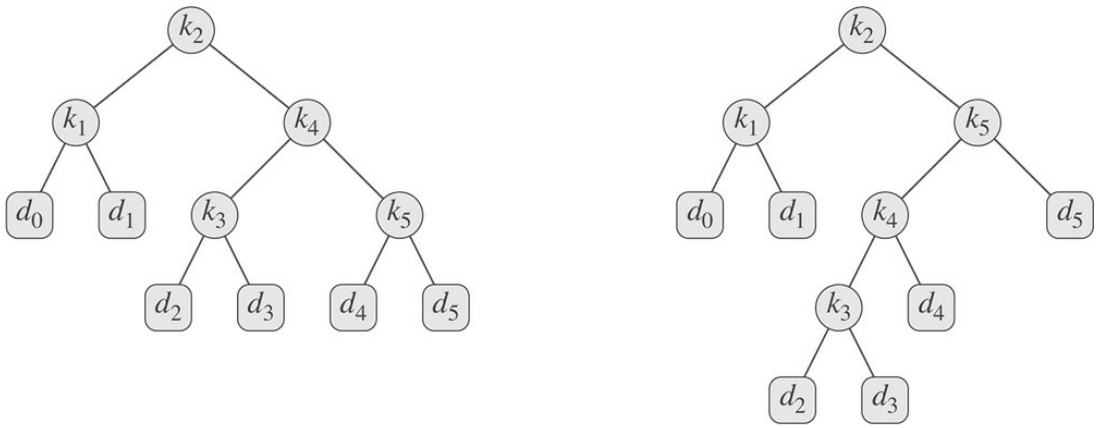
15.5 ÁRVORES DE BUSCA BINÁRIA ÓTIMAS

Suponha que estejamos projetando um programa para traduzir texto do inglês para o francês. Para cada ocorrência de cada palavra inglesa no texto, precisamos procurar sua equivalente em francês. Um modo de executar essas operações de busca é construir uma árvore de busca binária com n palavras inglesas como chaves e suas equivalentes francesas como dados satélites. Como pesquisaremos a árvore para cada palavra individual no texto, queremos que o tempo total gasto na busca seja tão baixo quanto possível. Poderíamos assegurar um tempo de busca $O(\lg n)$ por ocorrência usando uma árvore vermelho-preto ou qualquer outra árvore de busca binária balanceada. Porém, as palavras aparecem com frequências diferentes, e uma palavra usada frequentemente como *the* pode aparecer longe da raiz, enquanto uma palavra raramente usada como *machicolation* apareceria perto da raiz. Tal organização reduziria a velocidade da tradução, já que o número de nós visitados durante a busca de uma chave em uma árvore de

busca binária é igual a uma unidade mais a profundidade do nó que contém a chave. Queremos que palavras que ocorrem com frequência no texto sejam colocadas mais próximas à raiz.[6] Além disso, algumas palavras no texto podem não ter nenhuma tradução para o francês[7] e, portanto, não apareceriam em nenhum lugar na árvore de busca binária. Como organizamos uma árvore de busca binária para minimizar o número de nós visitados em todas as buscas, considerando que sabemos com que frequência cada palavra ocorre?

O que precisamos é conhecido como ***árvore de busca binária ótima***. Formalmente, temos uma sequência $K = \langle k_1, k_2, ..., k_n \rangle$ de n chaves distintas em sequência ordenada (de modo que $k_1 < k_2 < ... < k_n$), e desejamos construir uma árvore de busca binária com essas chaves. Para cada chave k_i, temos uma probabilidade p_i de que uma busca seja para k_i. Algumas buscas podem ser para valores que não estão em K, e então também temos $n + 1$ "chaves fictícias" $d_0, d_1, d_2, ..., dn$ que representam valores que não estão em K. Em particular, d_0 representa todos os valores menores que k_1, d_n representa todos os valores maiores que k_n e, para $i = 1, 2, ..., n - 1$, a chave fictícia d_i representa todos os valores entre k_i e k_i+1. Para cada chave fictícia d_i, temos uma probabilidade q_i de que uma busca corresponda a d_i. A Figura 15.9 mostra duas árvores de busca binária para um conjunto $n = 5$ chaves. Cada chave k_i é um nó interno e cada chave fictícia d_i é uma folha. Toda busca é bem sucedida (quando encontra alguma chave k_i) ou mal sucedida (quando encontra alguma chave fictícia d_i) e, então, temos

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1 . \qquad (15.10)$$



i	0	1	2	3	4	5
p_i		0,15	0,10	0,05	0,10	0,20
q_i	0,05	0,10	0,05	0,05	0,05	0,10

Figura 15.9 Duas árvores de busca binária para um conjunto de $n = 5$ chaves com as seguintes probabilidades: (**a**) Uma árvore de busca binária com custo de busca esperado 2,80. (**b**) Uma árvore de busca binária com custo de busca esperado 2,75. Essa árvore é ótima.

Como temos probabilidades de busca para cada chave e cada chave fictícia, podemos determinar o custo esperado de uma busca em uma árvore de busca binária dada T. Vamos supor que o custo real de uma busca seja igual ao número de nós examinados, isto é, a profundidade do nó encontrado pela busca em T mais 1. Então, o custo esperado de uma busca em T é

$$\begin{aligned}\text{E[custo de busca em } T] &= \sum_{i=1}^{n} (profundidade_T(k_i)+1)\cdot p_i + \sum_{i=0}^{n} (profundidade_T(d_i)+1)\cdot q_i \\ &= 1 + \sum_{i=1}^{n} profundidade_T(k_i)\cdot p_i + \sum_{i=0}^{n} profundidade_T(d_i)\cdot q_i, \end{aligned} \tag{15.11}$$

onde $profundidade_T$ denota a profundidade de um nó na árvore T. A última igualdade decorre da equação (15.10). Na Figura 15.9(a), podemos calcular o custo esperado da busca por nó:

nó	profundidade	probabilidade	contribuição
k_1	1	0,15	0,30
k_2	0	0,10	0,10
k_3	2	0,05	0,15
k_4	1	0,10	0,20
k_5	2	0,20	0,60
d_0	2	0,05	0,15
d_1	2	0,10	0,30
d_2	3	0,05	0,20
d_3	3	0,05	0,20
d_4	3	0,05	0,20
d_5	3	0,10	0,40
Total			2,80

Para um dado conjunto de probabilidades, queremos construir uma árvore de busca binária cujo custo de busca esperado seja o menor de todos. Damos a essa árvore o nome de ***árvore de busca binária ótima***. A Figura 15.9(b) mostra uma árvore de busca binária ótima para as probabilidades dadas na legenda da figura; seu custo esperado é 2,75. Esse exemplo mostra que uma árvore de busca binária ótima não é necessariamente uma árvore cuja altura global seja a menor. Nem funciona necessariamente construir uma árvore de busca binária ótima sempre colocando a chave com maior probabilidade na raiz. Aqui, a chave k_5 tem a maior probabilidade de busca de qualquer chave, entretanto a raiz da árvore de busca binária ótima mostrada é k_2. (O custo esperado mais baixo de qualquer árvore de busca binária com k_5 na raiz é 2,85.)

Como ocorre com a multiplicação de cadeias de matrizes, a verificação exaustiva de todas as possibilidades não produz um algoritmo eficiente. Podemos identificar os nós de qualquer árvore binária de n nós com as chaves k_1, k_2,..., k_n para construir uma árvore de busca binária e depois adicionar as chaves fictícias como folhas. No Problema 12-4, vimos que o número de árvores binárias com n nós é $(4^n/n_{3/2})$ e, portanto, em uma busca exaustiva teríamos de examinar um número exponencial de árvores de busca binária. Então, não é nenhuma surpresa que resolvamos esse problema com programação dinâmica.

Etapa 1: A estrutura de uma árvore de busca binária ótima

Para caracterizar a subestrutura ótima de árvores de busca binária ótima, começamos com uma observação sobre subárvores. Considere qualquer subárvore de uma árvore de busca binária. Ela deve conter chaves em uma faixa contígua k_i,...,k_j, para algum $1 \le i \le j \le n$. Além disso, uma subárvore que contém chaves k_i,...,k_j também deve ter como folhas as chaves fictícias d_i-1,...,d_j.

Agora podemos expressar a subestrutura ótima: se uma árvore de busca binária ótima T tem uma subárvore T' que contém chaves k_i,...,k_j, então essa subárvore T' deve também ser ótima para o subproblema com chaves k_i,...,k_j e

chaves fictícias $d_i^{-1},...,d_j$. O argumento habitual de recortar e colar é aplicável. Se houvesse uma subárvore T'' cujo custo esperado fosse mais baixo que o de T', poderíamos recortar T' de T e colar T'' no seu lugar, resultando em uma árvore de busca binária de custo esperado mais baixo que T, contradizendo assim a otimalidade de T.

Precisamos usar a subestrutura ótima para mostrar que podemos construir uma solução ótima para o problema partindo de soluções ótimas para subproblemas. Dadas as chaves $k_i,...,k_j$, uma dessas chaves, digamos k_r ($i \le r \le j$), é a raiz de uma subárvore ótima que contém essas chaves. A subárvore esquerda da raiz k_r contém as chaves $k_i,...,k_r^{-1}$ (e chaves fictícias $d_i^{-1},...,d_r^{-1}$), e a subárvore direita contém as chaves $k_r^{+1},...,k_j$ (e chaves fictícias $d_r,...,d_j$). Desde que examinemos todas as raízes candidatas k_r, onde $i \le r \le j$, e determinemos todas as árvores de busca binária ótimas que contêm $k_i,...,k_r^{-1}$ e as que contêm $k_r^{+1},...,k_j$, é garantido que encontraremos uma árvore de busca binária ótima.

Há um detalhe que vale a pena observar sobre subárvores "vazias". Suponha que em uma subárvore com chaves $k_i,...,k_j$, selecionemos k_i como a raiz. Pelo argumento que acabamos de apresentar, a subárvore esquerda de k_i contém as chaves $k_i,...,k_i^{-1}$. Interpretamos que essa sequência não contém nenhuma chave. Contudo, lembre-se de que subárvores também contêm chaves fictícias. Adotamos a seguinte convenção: uma subárvore que contém chaves $k_i,...,k_i^{-1}$ não tem nenhuma chave real, mas contém a única chave fictícia d_i^{-1}. Simetricamente, se selecionarmos k_j como a raiz, a subárvore direita de k_j contém as chaves $k_j^{+1},...,k_j$; essa subárvore direita não contém nenhuma chave real, mas contém a chave fictícia d_j.

Etapa 2: Uma solução recursiva

Estamos prontos para definir o valor de uma solução ótima recursivamente. Escolhemos, como domínio de nosso subproblema, encontrar uma árvore de busca binária ótima que contenha as chaves $k_i,...,k_j$, onde $i \ge 1$, $j \le n$ e $j \ge i - 1$. (Quando $j = i - 1$ não existe nenhuma chave real; temos apenas a chave fictícia d_i^{-1}.) Vamos definir $e[i, j]$ como o custo esperado de pesquisar uma árvore de busca binária ótima que contenha as chaves $k_i,...,k_J$. Em última análise, desejamos calcular $e[1, n]$.

O caso fácil ocorre quando $j = i - 1$. Então, temos apenas a chave fictícia d_i^{-1}. O custo de busca esperado é $e[i, i - 1] = q_i^{-1}$.

Quando $j \ge i$, precisamos selecionar uma raiz k que esteja entre $k_i,...,kj$ e fazer de uma árvore de busca binária ótima com chaves $k_i,...,k_r^{-1}$ sua subárvore esquerda e de uma árvore de busca binária ótima com chaves $k_r^{+1},...,k_j$ sua subárvore direita. O que acontece com o custo de busca esperado de uma subárvore quando ela se torna uma subárvore de um nó? A profundidade de cada nó na subárvore aumenta de 1. Pela equação (15.11), o custo de busca esperado dessa subárvore aumenta de uma quantidade igual à soma de todas as probabilidades na subárvore. Para uma subárvore com chaves $k_i,...,k_j$, vamos denotar essa soma de probabilidades como

$$w(i,j) = \sum_{l=i}^{j} p_1 + \sum_{l=i-1}^{j} q_1. \tag{15.12}$$

Assim, se k_r é a raiz de uma subárvore ótima contendo chaves $k_i,...,k_j$, temos

$$e[i,j] = p_r + (e[i,r-1] + w(i,r-1)) + (e[r+1,j] + w(r+1,j)).$$

Observando que

$$w(i,j) = w(i,r-1) + p_r + w(r+1,j),$$

reescrevemos $e[i, j]$ como

$$e[i,j] = w[i,r-1] + e[r+1,j] + w(i,j). \tag{15.13}$$

A equação recursiva (15.13) pressupõe que sabemos qual nó k_r usar como raiz. Escolhemos a raiz que dá o custo de busca esperado mais baixo, o que dá nossa formulação recursiva final:

$$e[i,j] = \begin{cases} q_{i-1} & \text{se } j = i-1, \\ \min\limits_{i \le r \le j} \{e[i,r-1] + e[r+1,j] + w(i,j)\} & \text{se } i \le j. \end{cases} \tag{15.14}$$

Os valores $e[i, j]$ dão os custos de busca esperados em árvores de busca binária ótimas. Para ajudar a controlar a estrutura de árvores de busca binária ótimas, definimos $raiz[i, j]$, para $1 \le i \le j \le n$, como o índice r para o qual k_r é a raiz de uma árvore de busca binária ótima contendo chaves $k_i,...,k_j$. Veremos como calcular os valores de $raiz[i, j]$, mas deixamos a construção da árvore de busca binária ótima com esses valores para o Exercício 15.5-1.

Etapa 3: Cálculo do custo de busca esperado de uma árvore de busca binária ótima

Até aqui, você deve ter notado algumas semelhanças entre as caracterizações que fizemos de árvores de busca binária ótimas e multiplicação de cadeias de matrizes. Para ambos os domínios de problemas, nossos subproblemas consistem em subfaixas de índices contíguos. Uma implementação recursiva direta da equação (15.14) seria tão ineficiente quanto um algoritmo recursivo direto de multiplicação de cadeias de matrizes. Em vez disso, armazenamos os valores $e[i, j]$ em uma tabela $e[1 .. n + 1, 0 .. n]$. O primeiro índice precisa ir até $n + 1$ em vez de n porque, para ter uma subárvore contendo apenas a chave fictícia d_n, precisaremos calcular e armazenar $e[n + 1, n]$. O segundo índice tem de começar de 0 porque, para ter uma subárvore contendo apenas a chave fictícia d_0, precisaremos calcular e armazenar $e[1, 0]$. Usamos somente as entradas $e[i, j]$ para as quais $j \ge i - 1$. Empregamos também uma tabela $raiz[i, j]$ para registrar a raiz da subárvore que contém as chaves $k_i,...,k_j$. Essa tabela utiliza somente as entradas para as quais $1 \le i \le j \le n$.

Precisaremos de uma outra tabela para eficiência. Em vez de calcular o valor de $w(i, j)$ desde o início toda vez que estamos calculando $e[i, j]$ — o que exigiria $Q(j - i)$ adições —, armazenamos esses valores em uma tabela $w[1..n + 1, 0..n]$. Para o caso-base, calculamos $w[i, i - 1] = q_i^{-1}$ para $1 \le i \le n$. Para $j \ge i$, calculamos

$$w[i,j] = w[i,j-1] + p_j + q_j . \tag{15.15}$$

Assim, podemos calcular cada um dos $Q(n_2)$ valores de $w[i, j]$ no tempo $Q(1)$.

O pseudocódigo a seguir, toma como entradas as probabilidades $p_1,...,p_n$ e $q_0,...,q_n$ e o tamanho n, e retorna as tabelas e e $raiz$.

```
OPTIMAL-BST (p, q, n)
1   sejam e[1..n + 1,0..n], w[1..n + 1, 0..n],
              e raiz [1..n, 1..n] tabelas novas
2   for i = 1 to n + 1
3      e[i, i − 1] = q_{i-1}
4      w[i, i − 1] = q_{i-1}
5   for l = 1 to n
6      for i = 1 to n − l + 1
7          j = i + l − 1
8          e[i, j] = ∞
9          w[i, j] = w[i, j − 1] + p_j + q_j
10         for r = i to j
11             t = e[i, r − 1] + e[r + 1, j] + w[i, j]
12             if t < e[i, j]
13                 e[i, j] = t
14                 raiz[i, j] = r
15     return e, raiz
```

Pela descrição anterior e pela semelhança com o procedimento MATRIX-CHAIN-ORDER da Seção 15.2, você constatará que a operação desse procedimento é razoavelmente direta. O laço **for** das linhas 2-4 inicializa os valores de $e[i, i-1]$ e $w[i, i-1]$. Então, o laço **for** das linhas 5-14 usa as recorrências (15.14) e (15.15) para calcular $e[i, j]$ e $w[i, j]$ para $1 \le i \le j \le n$. Na primeira iteração, quando $l = 1$, o laço calcula $e[i, i]$ e $w[i, i]$ para $i = 1, 2,...,n$. A segunda iteração, com $l = 2$, calcula $e[i, i+1]$ e $w[i, i+1]$ para $i = 1, 2,...,n-1$, e assim por diante. O laço **for** mais interno, nas linhas 10-14, experimenta cada índice candidato r para determinar que chave k_r usar como raiz de uma árvore de busca binária ótima contendo chaves $k_i,...,k_j$. Esse laço **for** salva o valor atual do índice r em $raiz[i, j]$ sempre que encontra uma chave melhor para usar como raiz.

A Figura 15.10 mostra as tabelas $e[i, j]$, $w[i, j]$ e $raiz[i, j]$ calculadas pelo procedimento OPTIMAL-BST para a distribuição de chaves mostrada na Figura 15.9. Como no exemplo de multiplicação de cadeias de matrizes da Figura 15.5, as tabelas sofreram uma rotação para colocar a diagonal principal na posição horizontal. OPTIMAL-BST calcula as linhas de baixo para cima e da esquerda para a direita dentro de cada linha.

O procedimento OPTIMAL-BST demora o tempo $\Theta(n_3)$, exatamente como MATRIX-CHAIN-ORDER. É fácil verificar que o tempo de execução é $O(n_3)$, já que seus laços **for** estão aninhados em profundidade três e cada índice de laço exige no máximo n valores. Os índices de laços em OPTIMAL-BST não têm exatamente os mesmos limites que os de MATRIX-CHAIN-ORDER, mas eles estão abrem no máximo 1 em todas as direções. Assim, exatamente como MATRIX-CHAIN-ORDER, o procedimento OPTIMAL-BST demora o tempo $\Theta(n_3)$.

Exercícios

15.5-1 Escreva o pseudocódigo para o procedimento CONSTRCUT-OPTIMAL-BST(*raiz*) que, dada a tabela *raiz*, dê como saída a estrutura de uma árvore de busca binária ótima. No exemplo da Figura 15.10, seu procedimento deve imprimir a estrutura

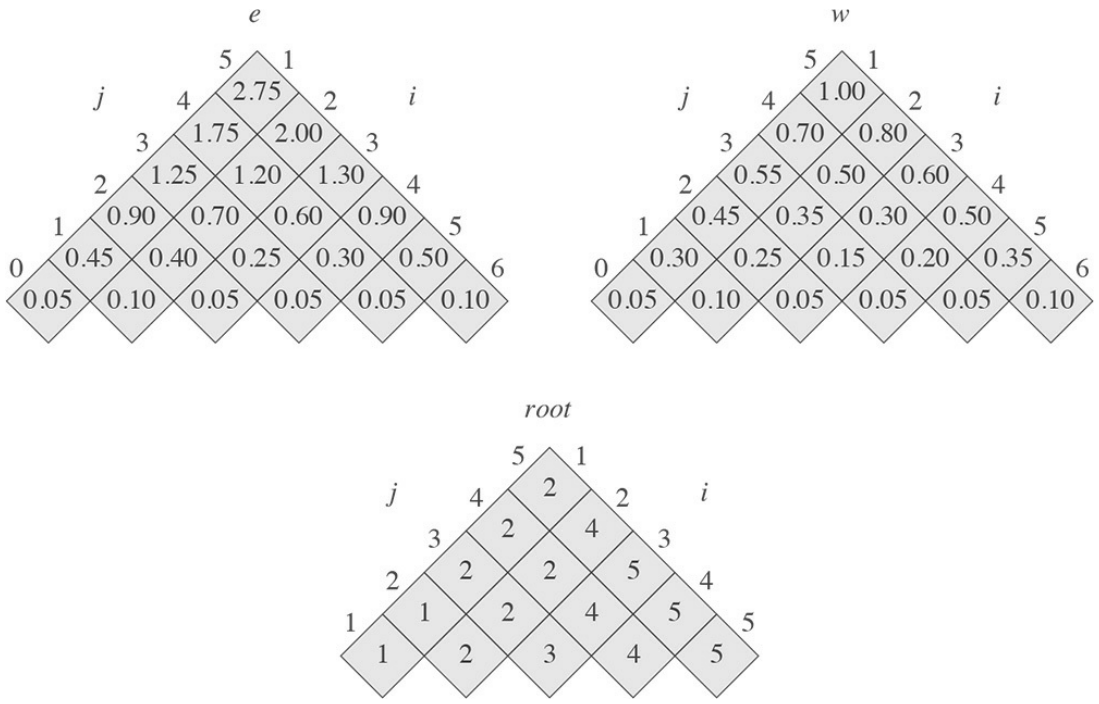


Figura 15.10 As tabelas ei, j , wi, j e $raizi, j$ calculadas por OPTIMAL-BST para a distribuição de chaves mostrada na Figura 15.9. As tabelas sofreram uma rotação para colocar as diagonais principais na posição horizontal.

k_2 é a raiz

k_1 é o filho à esquerda de k_2

d_0 é o filho à esquerda de k_1

d_1 é o filho à direita de k_1

k_5 é o filho à direita de k_2

k_4 é o filho à esquerda de k_5

k_3 é o filho à esquerda de k_4

d_2 é o filho à esquerda de k_3

d_3 é o filho à direita de k_3

d_4 é o filho à direita de k_4

d_5 é o filho à direita de k_5

correspondente à árvore de busca binária ótima mostrada na Figura 15.9(b).

15.5-2 Determine o custo e a estrutura de uma árvore de busca binária ótima para um conjunto de $n = 7$ chaves com as seguintes probabilidades:

i	0	1	2	3	4	5	6	7
p_i		0,04	0,06	0,08	0,02	0,10	0,12	0,14
q_i	0,06	0,06	0,06	0,06	0,05	0,05	0,05	0,05

15.5-3 Suponha que, em vez de manter a tabela $w[i, j]$, calculássemos o valor de $w(i, j)$ diretamente da equação (15.12) na linha 9 de OPTIMAL-BST e utilizássemos esse valor calculado na linha 11. Como essa mudança afetaria o tempo de execução assintótico de OPTIMAL-BST?

15.5-4 ★Knuth [214] mostrou que sempre existem raízes de subárvores ótimas tais que $raiz[i, j - 1] \leq raiz[i, j] \leq raiz[i + 1, j]$ para todo $1 \leq i < j \leq n$. Use esse fato para modificar o procedimento OPTIMAL-BST de modo que ele seja executado no tempo $\Theta(n_2)$.

Problemas

15.1 Caminho simples mais longo em um grafo acíclico dirigido

Suponha que tenhamos um grafo acíclico dirigido $G = (V, E)$ com pesos de arestas com valores reais e dois vértices distinguidos s e t. Descreva uma abordagem de programação dinâmica para determinar um caminho simples ponderado mais longo de s a t. Qual é a aparência do grafo de subproblema? Qual é a eficiência do seu algoritmo?

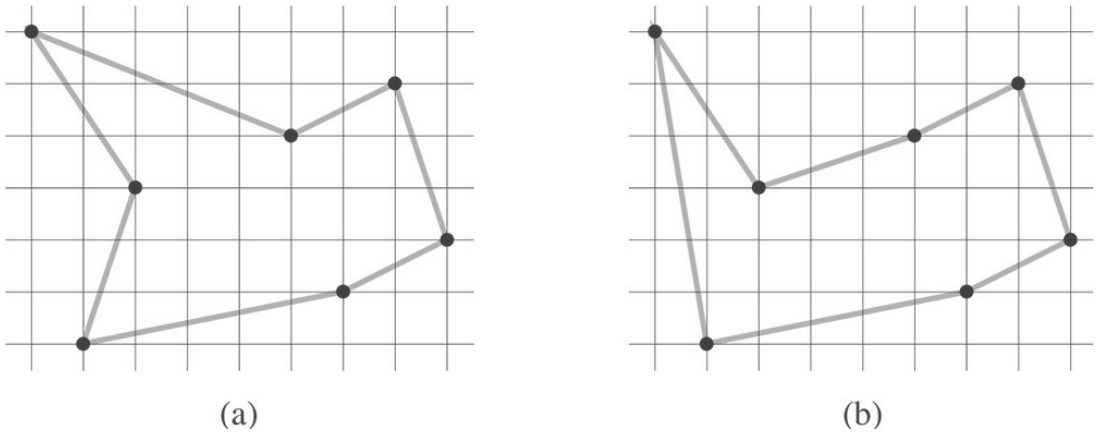


Figura 15.11 Sete pontos no plano, mostrados sobre uma grade unitária. **(a)** O caminho fechado mais curto, com comprimento aproximado de 24,89. Esse caminho não é bitônico. **(b)** O caminho fechado bitônico mais curto para o mesmo conjunto de pontos. Seu comprimento é aproximadamente 25,58.

15.2 Subsequência palíndromo mais longa

Um ***palíndromo*** é uma cadeia não vazia em algum alfabeto que é lida do mesmo modo da esquerda para a direita ou da direita para a esquerda. Exemplos de palíndromos são todas cadeias de comprimento 1, radar, asa, reter e oco.

Dê um algoritmo eficiente para encontrar o palíndromo mais longo que é uma subsequência de uma cadeia de entrada dada. Por exemplo, dada a entrada character nosso algoritmo retornaria carac. Qual é o tempo de execução do seu algoritmo?

15.3 Problema do caixeiro-viajante euclidiano bitônico:

No ***problema do caixeiro-viajante euclidiano*** temos um conjunto de n pontos no plano e queremos determinar o caminho fechado mais curto que conecta todos os n pontos. A Figura 15.11(a) mostra a solução para um problema de sete pontos. O problema geral é NP-difícil e, portanto, acredita-se que sua solução requeira mais que tempo polinomial (ver Capítulo 34).

J. L. Bentley sugeriu que simplificássemos o problema restringindo nossa atenção a ***caminhos fechados bitônicos***, isto é, caminhos fechados que começam no ponto da extrema esquerda, seguem estritamente da esquerda para a direita até o ponto da extrema direita e depois voltam estritamente da direita para a esquerda até o ponto de partida. A Figura 15.11(b) mostra o caminho fechado bitônico mais curto para os mesmos sete pontos. Nesse caso, é possível um algoritmo de tempo polinomial.

Descreva um algoritmo de tempo $O(n_2)$ para determinar um caminho fechado bitônico ótimo. Você pode considerar que não existem dois pontos com a mesma coordenada x. (*Sugestão*: Desloque-se da esquerda para a direita, mantendo possibilidades ótimas para as duas partes do caminho fechado.)

15.4 Como obter uma impressão nítida

Considere o problema de obter, em uma impressora, uma impressão nítida de um parágrafo em fonte monoespaçada (todos os caracteres têm a mesma largura). O texto de entrada é uma sequência de n palavras de comprimentos $l_1, l_2, \ldots l_n$, medidos em caracteres. Queremos imprimir esse parágrafo nitidamente em uma série de linhas que contêm no máximo M caracteres cada uma. Nosso critério de "nitidez" é dado a seguir. Se determinada linha contém palavras de i até j, onde $i \leq j$, e deixamos exatamente um espaço entre as palavras, o número de caracteres de espaço extras no final da linha é $M - j + i - \sum_{k=1}^{j} l_k$, que deve ser não negativo para que as palavras caibam na linha. Desejamos minimizar a soma em todas as linhas, exceto a última, dos cubos dos números de caracteres de espaço extras nas extremidades das linhas. Dê um algoritmo de programação dinâmica para imprimir um parágrafo de n palavras nitidamente em uma impressora. Analise o tempo de execução e os requisitos de espaço do seu algoritmo.

15.5 Distância de edição

Para transformar uma cadeia de texto de origem $x[1..m]$ na cadeia de texto que desejamos $y[1..n]$, podemos executar várias operações de transformação. Nossa meta é, dados x e y, produzir uma série de transformações que mudam x para y. Usamos um arranjo z — considerado grande o suficiente para conter todos os caracteres de que precisará — para conter os resultados intermediários. Inicialmente, z está vazio e, no término, devemos ter $z[j] = y[j]$ para $j = 1, 2, \ldots, n$. Mantemos índices atuais i em x e j em z, e as operações podem alterar z e esses índices. Inicialmente, $i = j = 1$. Temos de examinar cada caractere em x durante a transformação, o que significa que no fim da sequência de operações de transformação devemos ter $i = m + 1$.

Podemos escolher entre seis operações de transformação:

Copiar um caractere de x para z fazendo $z[j] = x[i]$ e incrementar i e j. Essa operação examina $x[i]$.

Substituir um caractere de x por outro caractere c fazendo $z[j] = c$ e incrementar i e j. Essa operação examina $x[i]$.

Excluir um caractere de x incrementando i, mas deixando j inalterado. Essa operação examina $x[i]$.

Inserir o caractere c em z definindo $z[j] = c$ e incrementar j, mas deixar i inalterado. Essa operação não examina nenhum caractere de x.

Transpor os dois caracteres seguintes, copiando-os de x para z mas na ordem oposta; para tal fazemos $z[j] = x[i + 1]$ e $z[j + 1] = x[i]$, e fazemos $i = i + 2$ e $j = j + 2$. Essa operação examina $x[i]$ e $x[i + 1]$.

Eliminar o restante de x fazendo $i = m + 1$. Essa operação examina todos os caracteres em x que ainda não foram examinados. Essa operação, se executada, deverá ser a última.

Como exemplo, um modo de transformar a corrente de origem algorithm na corrente desejada altruistic é usar a sequência de operações a seguir, onde os caracteres sublinhados são $x[i]$ e $z[j]$ após a operação:

Operação	x	z
cadeias iniciais	a̲lgorithm	_
copiar	al̲gorithm	a_
copiar	alg̲orithm	al_
substituir por *t*	algo̲rithm	alt_
excluir	algor̲ithm	alt_
copiar	algori̲thm	altr_
inserir *u*	algori̲thm	altru_
inserir *i*	algori̲thm	altrui_
inserir *s*	algori̲thm	altruis_
transpor	algorit̲hm	altruisti_
inserir *c*	algorit̲hm	altruistic_
eliminar	algorithm_	altruistic_

Observe que há várias outras sequências de operações de transformação que convertem algorithm em altruistic.

Cada uma das operações de transformação tem um custo associado. O custo de uma operação depende da aplicação específica, mas consideramos que o custo de cada operação é uma constante que conhecemos. Supomos também que os custos individuais das operações copiar e substituir são menores que os custos combinados das operações excluir e inserir, senão as operações de copiar e substituir não seriam usadas. O custo de uma dada sequência de operações de transformação é a soma dos custos das operações individuais na sequência. Para a sequência que estudamos neste exercício, o custo de transformar algorithm em altruistic é

$(3 \cdot \text{custo(copiar)}) + \text{custo(substituir)} + \text{custo(excluir)} + (4 \cdot \text{custo(inserir)}) + \text{custo(girar)} + \text{custo(eliminar)}$.

a. Dadas duas sequências $x1..m$ e $y1..n$ e um conjunto de custos de operação, a ***distância de edição*** de x para y é o custo da sequência menos dispendiosa de operações que transforma x em y. Descreva um algoritmo de programação dinâmica que determine a distância de edição de $x1..m$ para $y1..n$ e imprime uma sequência de operações ótima. Analise o tempo de execução e os requisitos de espaço de seu algoritmo.

O problema da distância de edição generaliza o problema de alinhar duas sequências de DNA (veja, por exemplo, Setubal e Meidanis [310, Seção 3.2]). Há vários métodos para medir a semelhança entre duas sequências de DNA por alinhamento. Um dos métodos para alinhar duas sequências x e y consiste em inserir

espaços em posições arbitrárias nas duas sequências (inclusive em qualquer extremidade) de modo que as sequências resultantes x' e y' tenham o mesmo comprimento, mas não um espaço na mesma posição (isto é, para nenhuma posição j, $x'[j]$ e $y'[j]$ são espaços). Então, atribuímos uma "pontuação" a cada posição. A posição j recebe uma pontuação da seguinte maneira:

- + 1 se $x'[j] = y'[j]$ e nenhum deles é um espaço,
- -1 se $x'j \neq y'j$ e nenhum deles é um espaço,
- -2 se $x'j$ ou $y'j$ é um espaço.

A pontuação para o alinhamento é a soma das pontuações das posições individuais. Por exemplo, dadas as sequências x = GATCGGCAT e y = CAATGTGAATC, um alinhamento é

```
G ATCG GCAT
CAAT GTGAATC
-*++*+*+-++*
```

Um + sob uma posição indica uma pontuação +1 para aquela posição, um – indica a pontuação -1 e um * indica a pontuação -2; portanto, esse alinhamento tem uma pontuação total igual a $6 \cdot 1 - 2 \cdot 1 - 4 \cdot 2 = -4$.

b. Explique como expressar o problema de determinar um alinhamento ótimo como um problema de distância de edição usando um subconjunto das operações de transformação copiar, substituir, excluir, inserir, girar e eliminar.

15.6 Planejamento de uma festa da empresa

O professor Stewart presta consultoria ao presidente de uma corporação que está planejando uma festa da empresa. A empresa tem uma estrutura hierárquica, isto é, as relações entre os supervisores formam uma árvore com raiz no presidente. O pessoal do escritório classificou cada funcionário segundo uma avaliação de sociabilidade que é um número real. Para tornar a festa divertida para todos os participantes, o presidente não quer que um funcionário e seu supervisor imediato participem.

O professor Stewart recebe a árvore que descreve a estrutura da corporação usando a representação de filho à esquerda, irmão à direita descrita na Seção 10.4. Cada nó da árvore contém, além dos ponteiros, o nome de um funcionário e o posto que ele ocupa na escala de classificação de sociabilidade. Descreva um algoritmo para compor uma lista de convidados que maximize a soma das avaliações de sociabilidade dos convidados. Analise o tempo de execução do seu algoritmo.

15.7 Algoritmo de Viterbi

Podemos usar programação dinâmica em um grafo dirigido $G = (V, E)$ para reconhecimento de voz. Cada aresta $(u, v) \in E$ é identificada por um som (u, v) de um conjunto finito Σ de sons. O grafo rotulado é um modelo formal de uma pessoa falando uma linguagem restrita. Cada caminho no grafo que parte de um vértice distinto $v_0 \in V$ corresponde a uma sequência possível de sons produzidos pelo modelo. Definimos o rótulo de um caminho dirigido como a concatenação dos rótulos das arestas nesse caminho.

a. Descreva um algoritmo eficiente que, dado um grafo com arestas rotuladas G contendo um vértice distinto v_0 e uma sequência $s = \langle s_1, s_2, ..., s_k \rangle$ de sons pertencentes ao conjunto Σ, retorne um caminho em

G que começa em v_0 e tenha s como rótulo, se tal caminho existir. Caso contrário, o algoritmo deve retornar NO-SUCH-PATH. Analise o tempo de execução de seu algoritmo. (*Sugestão*: Os conceitos do Capítulo 22 poderão ser úteis.)

Agora, suponha que toda aresta $(u, v) \in E$ tenha uma probabilidade associada não negativa $p(u, v)$ de percorrer a aresta (u, v) desde o vértice u e, assim, produzir o som correspondente. A soma das probabilidades das arestas que saem de qualquer vértice é igual a 1. A probabilidade de um caminho é definida como o produto das probabilidades de suas arestas. Podemos considerar a probabilidade de um caminho que começa em v_0 como a probabilidade de um "percurso aleatório" começando em v_0 seguir o caminho especificado, onde escolhemos aleatoriamente qual aresta que sai de um vértice u tomar de acordo com as probabilidades das arestas disponíveis que partem de u.

b. Amplie sua resposta à parte (a), de modo que, se um caminho for retornado, ele é um *caminho mais provável* que começa em v_0 e tem rótulo s. Analise o tempo de execução do seu algoritmo.

15.8 Compressão de imagem por descostura (seam carving)

Temos uma figura em cores que consiste em um arranjo mn $A[1..m, 1..n]$ de pixels, onde cada pixel especifica uma tripla de intensidades de vermelho, verde e azul (RGB). Suponha que queiramos comprimir ligeiramente essa figura. Especificamente, queremos remover um pixel de cada uma das m linhas, de modo que a figura inteira fique um pixel mais estreita. Porém, para evitar distorção nos efeitos visuais, é necessário que os pixels removidos em duas linhas adjacentes estejam na mesma coluna ou em colunas adjacentes. Os pixels removidos formam uma "costura" da linha superior até a linha inferior. Nessa costura, os pixels sucessivos são adjacentes na vertical e na diagonal.

a. Mostre que o número de tais costuras possíveis cresce no mínimo exponencialmente em m, considerando que $n > 1$.

b. Agora suponha que juntamente com cada pixel Ai, j, calculamos uma medida de distorção de valor real di, j, que indica qual seria o grau de distorção causado pela remoção do pixel Ai, j. Intuitivamente, quanto mais baixa a medida de distorção causada por um pixel, mais semelhante a seus vizinhos é esse pixel. Suponha ainda que definimos a medida de distorção de uma costura como a soma das medidas de distorção de seus pixels.

Dê um algoritmo para encontrar uma costura que tenha a medida de distorção mais baixa. Qual seria a eficiência desse algoritmo?

15.9 Quebra de cadeia

Certa linguagem de processamento de cadeias permite que um programador quebre uma cadeia em dois pedaços. Como essa operação copia a cadeia, quebrar uma cadeia de n caracteres em dois pedaços tem um custo de n unidades de tempo. Suponha que um programador queira quebrar uma cadeia em muitos pedaços. A ordem em que as quebras ocorrem pode afetar a quantidade total de tempo gasto. Por exemplo, suponha que a programadora queira quebrar uma cadeia de 20 caracteres depois dos caracteres 2, 8 e 10 (numerando os caracteres em ordem ascendente a partir da extremidade esquerda e começando de 1). Se ela programar as quebras da esquerda para a direita, a primeira quebra custará 20 unidades de tempo, a segunda quebra custará 18 unidades de tempo (quebra da cadeia dos caracteres 3 a 20 no caractere 8), e a terceira quebra custará 12 unidades de tempo, totalizando 50 unidades de tempo. Entretanto, se ela programar as quebras da direita para a esquerda, a primeira quebra custará 20 unidades de tempo, a segunda quebra custará 10 unidades de tempo, e a terceira quebra custará 8 unidades de tempo, totalizando 38 unidades de tempo.

Ainda em uma outra ordem, ela poderia programar a primeira quebra em 8 (custo 20), quebrar o pedaço à esquerda em 2 (custo 8) e, finalmente, o pedaço à direita em 10 (custo 12), o que dá um custo total de 40. Projete um algoritmo que, dados os números de posição dos caracteres após os quais ocorrerão as quebras, determine uma sequência de menor custo dessas quebras. Mais formalmente, dada uma cadeia S com n caracteres e um arranjo $L[1..m]$ que contém os pontos de quebra, calcule o menor custo para uma sequência de quebras juntamente com uma sequência de quebras que atinja esse custo.

15.10 Planejamento de uma estratégia de investimento

Como você conhece bem algoritmos, consegue um emprego interessante na Acme Computer Company, além de um bônus contratual de $10.000. Você decide investir esse dinheiro com o objetivo de maximizar o retorno em 10 anos. Então, decide contratar a Amalgamated Investment Company para gerenciar os seus investimentos. Essa empresa exige que você observe as regras descritas a seguir. Ela oferece n investimentos diferentes, numerados de 1 a n. Em cada ano j, o investimento dá uma taxa de retorno de r_{ij}. Em outras palavras, se você investiu d dólares no investimento i no ano j, ao final do ano j terá dr_{ij} dólares. As taxas de retorno são garantidas, isto é, a empresa informa todas as taxas de retorno para os próximos 10 anos para cada investimento. Você decide o rumo de seus investimentos uma vez por ano. Ao fim de cada ano, você pode deixar o dinheiro ganho no ano anterior nos mesmos investimentos, ou pode transferir dinheiro para outros investimentos, seja por transferência entre investimentos existentes, ou seja por transferência para um novo investimento. Se você não movimenta o dinheiro entre dois anos consecutivos, você paga uma taxa de f_1 dólares, enquanto que se houver transferência, você paga uma taxa de f_2 dólares, onde $f_2 > f_1$.

a. O problema, como enunciado, permite que você aplique seu dinheiro em vários investimentos a cada ano. Prove que existe uma estratégia de investimento ótima que, a cada ano, investe todo o dinheiro em um único investimento. (Lembre-se de que uma estratégia de investimento ótima maximiza a quantia investida após 10 anos e não se preocupa com outros objetivos, como minimizar riscos.)

b. Prove que o problema de planejar sua estratégia de investimentos ótima exibe subestrutura ótima.

c. Projete um algoritmo que planeje sua estratégia de investimentos ótima. Qual é o tempo de execução desse algoritmo?

d. Suponha que a Amalgamated Investments tenha imposto a seguinte restrição adicional: a qualquer instante, você não pode ter mais de $15.000 em qualquer dos investimentos. Mostre que o problema de maximizar sua receita ao final de 10 anos deixa de exibir subestrutura ótima.

15.11 Planejamento de estoque

A Ricky Dink Company fabrica máquinas para restaurar a superfície de rinques de patinação no gelo. A demanda por tais produtos varia de mês a mês e, por isso, a empresa precisa desenvolver uma estratégia de planejamento de produção dada a demanda flutuante, porém previsível. A empresa quer projetar um plano para os próximos n meses e sabe qual é a demanda d_i para cada mês i, isto é, o número de máquinas que venderá nesse mês. Seja $D = \sum_{i=1}^{n} d_i$ a demanda total para os próximos n meses. A empresa mantém um quadro permanente de funcionários de tempo integral que fornecem a mão de obra para produzir até m máquinas por mês. Se ela precisar fabricar mais de m máquinas em determinado mês, pode contratar mão de obra temporária adicional, a um custo calculado de c dólares por máquina. Além disso, se ao final de um mês a empresa tiver em estoque qualquer número de máquinas não vendidas terá de pagar custos de estoque. O custo de estocar j máquinas é dado como uma função $h(j)$ para $j = 1, 2,..., D$, onde $h(j) \geq 0$

para $1 \le j \le D$ e $h(j) \le h(j+1)$ para $1 \le j \le D-1$.

Dê um algoritmo para calcular um plano de produção para a empresa que minimize seus custos e, ao mesmo tempo, atenda à demanda. O tempo de execução deve ser polinomial em n e D.

15.12 Contratação de jogadores de beisebol donos de seu passe

Suponha que você seja o gerente geral de um time de beisebol da primeira divisão. No período entre temporadas, você precisa contratar para sua equipe alguns jogadores donos de seu próprio passe. O dono do time disponibilizou $\$X$ para gastar com esses jogadores e você pode gastar menos de $\$X$ no total, mas será demitido se gastar mais de $\$X$. Você está considerando N posições diferentes e, para cada posição, há P jogadores disponíveis.[8] Como não quer sobrecarregar seu plantel com muitos jogadores em alguma posição, você decide contratar no máximo um jogador reserva adicional para cada posição. (Se não contratar nenhum jogador para uma determinada posição, você planeja continuar apenas com os jogadores de seu time titular para tal posição.)

Para determinar o valor futuro de um jogador, você decide usar uma estatística sabermétrica[9] conhecida como VORP (*value over replacement player* — valor de um reserva). Um jogador que tenha um VORP mais alto é mais valioso que um jogador com VORP mais baixo. Contratar um jogador que tenha um VORP mais alto não é necessariamente mais caro que contratar um com VORP mais baixo porque há outros fatores que determinam o custo do contrato, além do valor do jogador.

Para cada jogador reserva, você tem três informações:

- a posição do jogador,
- quanto custará contratar o jogador e
- o VORP do jogador.

Projete um algoritmo que maximize o VORP total dos jogadores que você contrata e, ao mesmo tempo, não gaste mais de $\$X$ no total. Suponha que o contrato de cada jogador seja sempre um múltiplo de $100.000. Seu algoritmo deve dar como saída o VORP total dos jogadores contratados, o total de dinheiro gasto e uma lista dos jogadores contratados. Analise o tempo de execução e o requisito de espaço do seu algoritmo.

NOTAS DO CAPÍTULO

R. Bellman começou o estudo sistemático de programação dinâmica em 1955. A palavra "programação", tanto aqui quanto em programação linear, se refere ao uso de um método de solução tabular. Embora as técnicas de otimização que incorporam elementos de programação dinâmica fossem conhecidas antes, Bellman deu à área uma sólida base matemática [37].

Galil e Park [125] classificam algoritmos de programação dinâmica de acordo com o tamanho da tabela e o número de outras entradas de tabela das quais cada entrada depende. Eles denominam um algoritmo de programação dinâmica tD/eD se o tamanho de sua tabela for $O(n_t)$ e cada entrada depender de outras $O(n_e)$ entradas. Por exemplo, o algoritmo de multiplicação de cadeia de matrizes na Seção 15.2 seria $2D/1D$, e o algoritmo da subsequência comum mais longa na Seção 15.4 seria $2D/0D$.

Hu e Shing [182, 183] apresentam um algoritmo de tempo $O(n \lg n)$ para o problema de multiplicação de cadeias de matrizes.

O algoritmo de tempo $O(mn)$ para o problema da subsequência comum mais longa parece ser um algoritmo folclórico. Knuth [70] levantou a questão da existência ou não de algoritmos subquadráticos para o problema da LCS. Masek e Paterson [244] responderam afirmativamente a essa pergunta, dando um algoritmo que é executado no tempo $O(mn/\lg n)$, onde $n \leq m$ e as sequências são extraídas de um conjunto de tamanho limitado. Para o caso especial no qual nenhum elemento aparece mais de uma vez em uma sequência de entrada, Szymanski [326] mostra como resolver o problema no tempo $O((n + m)\lg(n + m))$. Muitos desses resultados se estendem ao problema de calcular distâncias de edição de cadeias (Problema 15-5).

Um artigo anterior sobre codificações binárias de comprimento variável apresentado por Gilbert e Moore [133] teve aplicações na construção de árvores de busca binária ótimas para o caso no qual todas as probabilidades p_i sejam 0; esse artigo contém um algoritmo de tempo $O(n_3)$. Aho, Hopcroft e Ullman [5] apresentam o algoritmo da Seção 15.5. O Exercício 15.5-4 se deve a Knuth [212]. Hu e Tucker [184] criaram um algoritmo para o caso no qual todas as probabilidades p_i sejam 0 e utiliza o tempo $O(n_2)$ e o espaço $O(n)$; mais tarde, Knuth [211] reduziu o tempo para $O(n \lg n)$.

O Problema 15-8 se deve a Avidan e Shamir [27], que apresentaram na Web um maravilhoso vídeo que ilustra essa técnica de compressão de imagem.

---

1 Se exigíssemos que as peças fossem cortadas em ordem não decrescente de tamanho, haveria um número menor de modos a considerar. Para $n = 4$, consideraríamos somente cinco desses modos: partes (a), (b), (c), (e) e (h) na Figura 15.2. O número de modos é denominado ***função partição***; é aproximadamente igual a $e^{\pi\sqrt{2n/3}} / 4n\sqrt{3}$. Essa quantidade é menor que 2^{n+1}, porém ainda muito maior do que qualquer polinômio em n. Todavia, não prosseguiremos nessa linha de raciocínio.

2 Isso não é um erro de ortografia. A palavra é realmente *memoização*, e não *memorização*. *Memoização* vem de *memo*, já que a técnica consiste em gravar um valor de modo que possamos consultá-lo mais tarde.

3 Usamos o termo "não ponderado" para distinguir esse problema do problema de encontrar caminhos mais curtos com arestas ponderadas, que veremos nos Capítulos 24 e 25. Podemos usar a técnica da busca em largura apresentada no Capítulo 22 para resolver o problema não ponderado.

4 Pode parecer estranho que programação dinâmica dependa de subproblemas que são ao mesmo tempo independentes e sobrepostos. Embora possam parecer contraditórios, esses requisitos descrevem duas noções diferentes, em vez de dois pontos no mesmo eixo. Dois subproblemas do mesmo subproblema são independentes se não compartilharem recursos. Dois subproblemas são sobrepostos se realmente forem o mesmo subproblema que ocorre como um subproblema de problemas diferentes.

5 Essa abordagem pressupõe que conhecemos o conjunto de todos os parâmetros de subproblemas possíveis e que estabelecemos a relação entre posições de tabela e subproblemas. Uma outra abordagem, mais geral, é memoizar usando hashing com os parâmetros do subproblema como chave.

6 Se o assunto do texto fosse arquitetura de castelos, talvez quiséssemos que *machicolation* aparecesse perto da raiz.

7 Sim, *machicolation* tem uma contraparte em francês: *mâchicoulis*.

8 Embora haja nove posições em um time de beisebol, N não é necessariamente igual a 9 porque o modo como alguns gerentes gerais pensam sobre posições é peculiar. Por exemplo, um gerente geral poderia considerar que há duas "posições" distintas para arremessadores (*pitchers*), isto é, os arremessadores destros e os arremessadores canhotos, além do primeiro arremessador da partida, dos arremessadores de longo prazo, que podem arremessar por vários turnos, e dos de curto prazo, que normalmente arremessam no máximo um turno.

9 ***Sabermétrica*** é a aplicação de análise estatística a registros de dados de jogos de beisebol. A sabermétrica dá vários modos para comparar valores relativos de jogadores individuais.