Análise de algoritmos

Programação dinâmica

Conteúdo I

Conteúdo II

Introdução: Programação dinâmica I

- Programação dinâmica (PD) é uma técnica de projeto de algoritmos (como divisão e conquista) e não um algoritmo específico
- "Programação" se refere a um método tabular, não ao processo de escrever código de computador
- Usada para problemas de otimização:
  - Encontrar *uma* solução com *o* valor ótimo
  - Minimização ou maximização

Introdução: Programação dinâmica II

Método de quatro etapas:

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

Considerações:

- Etapas 1 a 3 formam a base de uma solução PD para um problema
- Se precisarmos apenas valor da solução ótima: podemos omitir a etapa 4
- Quando precisamos executar a etapa 4, às vezes mantemos informações adicionais durante a etapa 3, para facilitar a construção de uma solução ótima

Introdução: Corte de hastes I

Contexto:

- Decidir onde cortar hastes de aço
- Serling Enterprises compra hastes de aço longas e as corta em hastes mais curtas, para vender
- Cada corte é gratuito
- A gerência quer saber qual é o melhor modo de cortar as hastes

Supondo que conhecemos, para $i = 1, 2, \ldots$, o preço p_i de venda de uma haste de i polegadas, como segue:

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

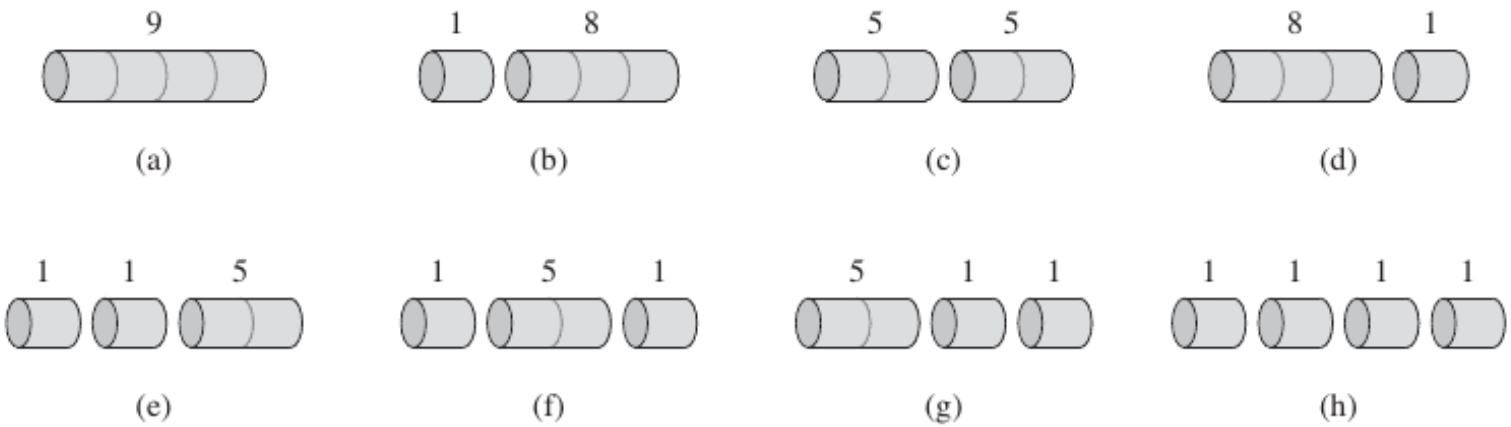
Introdução: Corte de hastes II

Problema de corte de hastes

Dada uma haste de n polegadas de comprimento e uma tabela 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.

Introdução: Corte de hastes III

- Considere o caso em que $n = 4$
- A seguir, todas as maneiras possíveis de cortá-la, incluindo não cortá-la
- Cortar em duas peças de 2 polegadas produz a receita $p_2 + p_2 = 5 + 5 = 10$, que é ótima



Introdução: Corte de hastes IV

- Podemos cortar uma haste de comprimento n de 2^{n-1} modos diferentes
- Dado que temos uma opção independente de cortar, ou não cortar, à distância de i polegadas das extremidade esquerda para $i = 1, 2, ..., n - 1$
- $7 = 2 + 2 + 3$ indica que uma haste de comprimento 7 foi cortada em três peças
- Se uma solução ótima cortar a haste em k pedaços, para algum $1 \leq k \leq n$ então o desdobramento ótimo

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

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

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

Introdução: Corte de hastes V

Exemplo (...)

Introdução: Corte de hastes VI

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

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

- O primeiro argumento corresponde a não fazer nenhum corte
- Os outros $n - 1$ argumentos correspondem à receita máxima de um corte inicial da haste em duas peças de tamanhos i e $n - i$, para cada $i = 1, 2, ..., n - 1$ e,
- prosseguindo com o corte ótimo dessas peças obtendo as receitas r_i e r_{n-i}
- 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 maximiza a receita

Introdução: Corte de hastes VII

- Temos a opção de não escolher nenhum i, se pudermos obter mais receita vendendo a haste inteira

Prosseguindo:

- 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
- A solução ótima global incorpora soluções ótimas para os dois subproblemas relacionados, maximizando a receita gerada por esses dois pedaços
- Dizemos que o problema de corte de hastes exibe **subestrutura ótima**: soluções ótimas para um problema incorporam soluções ótimas para subproblemas relacionados, que podemos resolver independentemente

Introdução: Corte de hastes VIII

Estrutura recursiva:

- Um modo mais simples: 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
- Assim, 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 $r_0 = 0$

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

Introdução: Corte de hastes IX

- 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 I

- Implementação do cálculo implícito na equação anterior, de modo direto, recursivo, de cima para baixo
- Retorna a receita r_n ótima

```
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
```

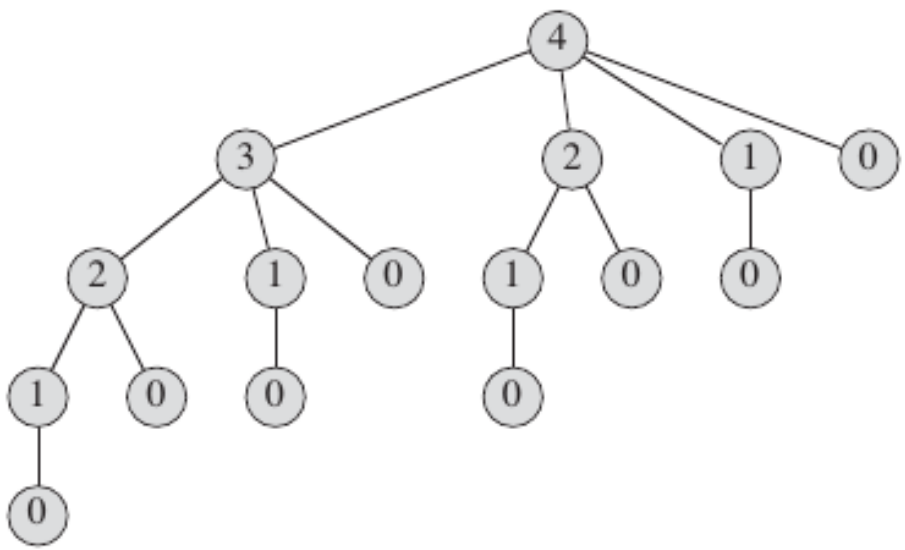
- Laço das linhas 4-5 computa corretamente $q = \max_{1 \leq i \leq n}(p_i + \texttt{Cut-Rod}(p, n - i))$
- Uma indução em n prova que a resposta é igual a r_n (da equação anterior)

Implementação recursiva de cima para baixo II

- Tão logo o tamanho do problema se torna moderadamente grande, seu programa levaria logo tempo
- Para $n = 40$, demoraria no mínimo vários minutos, ou mais de uma hora
- Quanto n aumenta 1, o tempo de execução é aprox. 2 vezes maior

Implementação recursiva de cima para baixo III

- Por que `Cut-Rod` é tão ineficiente?
  - `Cut-Rod` chama ele mesmo repetidamente, mesmo para subproblemas que ele já havia resolvido
  - Vários subproblemas repetidos
  - Resolve o subproblema de tamanho 2 duas vezes
  - ...de tamanho 1 quatro vezes
  - ...de tamanho 0 oito vezes



Implementação recursiva de cima para baixo IV

- Crescimento exponencial
  - $T(n)$ representa o número de chamadas a `Cut-Rod` com segundo parâmetro n, assim:

$$T(n) = \begin{cases} 1 & \text{se } n = 0, \\ 1 + \sum_{j=0}^{n-1} T(j) & \text{se } n \geq 1. \end{cases}$$

  - Somatório conta chamadas onde o segundo parâmetro é $j = n - i$
  - Solução para recorrência: $T(n) = 2^n$

Utilização de **programação dinâmica** para o corte ótimo de hastes I

- Ao invés de resolver o mesmo problema repetidamente, adaptamos para resolver cada subproblema apenas uma vez
- Armazenar a solução para um subproblema em uma tabela, referir-se a tabela toda vez que revisitamos o subproblema
- "Armazenar, não recomputar": **permuta tempo-memória**
- Pode transformar uma solução de tempo exponencial em uma solução de tempo polinomial
  - Uma abordagem PD tem tempo polinomial quando o número de subproblemas *distintos* envolvidos é polinomial no tamanho da entrada e podemos resolver cada subproblema em tempo polinomial
- Duas abordagem básicas:
  De cima para baixo com memoização, e de baixo para cima

Utilização de **programação dinâmica** para o corte ótimo de hastes II

- **De cima para baixo com memoização**
  - Resolver recursivamente, mas armazenar cada resultado em uma tabela
  - Para encontrar a solução para um subproblema, olhar primeiro na tabela. Se a resposta estiver lá, use-a. Caso contrário, computar a solução para o subproblema e então armazenar a solução na tabela para uso futuro
  - **Memoização** é relembrar o que havíamos calculado anteriormente
  - A seguir, versão memoizada do algoritmo recursivo, armazenando a solução para o subproblema de tamanho i em uma entrada de arranjo $r[i]$:

Utilização de **programação dinâmica** para o corte ótimo de hastes III

```
MEMOIZED-CUT-ROD(p, n)
1  let r[0..n] be a new array
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
```

Utilização de **programação dinâmica** para o corte ótimo de hastes IV

- **De baixo para cima**
  - Ordenar os subproblemas por tamanho e resolver os menores primeiro
  - Dessa forma, quando da resolução de um subproblema, já terão sido resolvidos os subproblemas de tamanhos menores necessários

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
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]
```

Utilização de **programação dinâmica** para o corte ótimo de hastes V

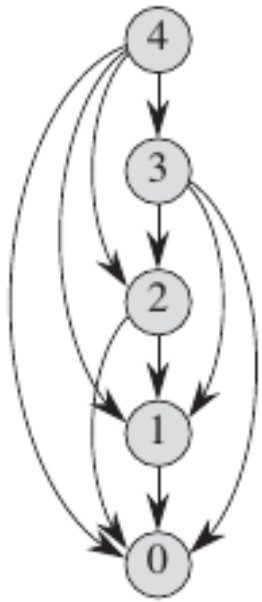
- **Tempo de execução**
  - As versões *de cima para baixo* e *de baixo para cima* executam em tempo $\Theta(n^2)$
  - **De baixo para cima**: Laços duplamente aninhados. Número de iterações do laço `for` interno forma uma série aritmética
  - **De cima para baixo**: `Memoized-Cut-Rod` resolve cada problema apenas uma vez, e faz isso para subproblemas de tamanhos $0, 1, \ldots, n$. Para resolver um subproblema de tamanho n, o laço `for` itera n vezes $\rightarrow$ considerando todas as chamadas recursivas, o número total de iterações forma uma série aritmética (usando análise agregada)

Grafos de subproblemas I

- Grafo de subproblemas: Uma maneira de compreender os subproblemas envolvidos e como eles dependem uns dos outros
- Grafo orientado
  - Um vértice distinto para cada subproblema
  - Possui uma aresta orientada (arco) (x, y) se a computação de uma solução ótima para o subproblema x requer conhecer *diretamente* uma solução ótima para o subproblema y

Grafos de subproblemas II

- Exemplo: problema de corte de hastes com $n = 4$:



Grafos de subproblemas III

- Podemos pensar em um grafo de subproblemas como uma versão "reduzida" ou "colapsada" de uma árvore de chamadas recursivas, onde todos os nós para o mesmo problema são reunidos em um único vértice, e orientamos todas as arestas para irem de pai para o filho
- Grafo de subproblema pode ajudar a determinar o tempo de execução. Pelo fato de resolvermos cada subproblema apenas uma vez, o tempo de execução é a soma dos tempos necessários para resolver cada subproblema
  - Tempo para computar a solução para um subproblema é tipicamente linear no grau de saída de seu vértice
  - Número de subproblemas é igual ao número de vértices
- Quando essas condições são válidas, o tempo de execução é linear no número de vértices e arestas

Reconstruindo uma solução I

- Até agora, focamos em computar o *valor* de uma solução ótima, ao invés das *escolhas* que produzem uma solução ótima
- Estender a abordagem de baixo para cima para armazenar não apenas valores ótimos, mas também escolhas ótimas. Salvar as escolhas ótimas em uma tabela separada. Então, usar um procedimento separado para imprimir os valores das escolhas ótimas
- Segue algoritmo:

Reconstruindo uma solução II

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
 1  let r[0..n] and s[0..n] be new arrays
 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
```

- O algoritmo salva o primeiro corte feito em uma solução ótima para o problema de tamanho i em $s[i]$

Reconstruindo uma solução III

- Para imprimir os cortes feitos em uma solução ótima:

```
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]
```

- No exemplo, a chamada `Extended-Bottom-Up-Cut-Rod(p, 10)` retorna 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

Reconstruindo uma solução IV

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, correspondente ao primeiro desdobramento ótimo para r_7 dado anteriormente

Exercícios I

3^a ed.: 15.1-1 a 15.1-5

Introdução: Multiplicação de cadeia de matrizes I

- Sequência (cadeia) $\langle A_1, A_2, ..., A_n \rangle$ de n matrizes para multiplicar e desejamos calcular o produto $A_1 A_2 ... A_n$
- Associativa
- Totalmente parentizado se for uma única matriz ou o produto de dois produtos de matrizes totalmente parentizados também expresso entre parênteses
- $\langle A_1, A_2, A_3, A_4 \rangle$, podemos expressar o produto $A_1 A_2 A_3 A_4$ como:
  $(A_1(A_2(A_3 A_4)))$
  $(A_1((A_2 A_3)A_4))$
  $((A_1 A_2)(A_3 A_4))$
  $((A_1(A_2 A_3))A_4)$
  $(((A_1 A_2)A_3)A_4)$

Introdução: Multiplicação de cadeia de matrizes II

- Algoritmo-padrão para efetuar multiplicação de duas matrizes

```
MATRIX-MULTIPLY(A, B)
1  if A.columns ≠ B.rows
2      error "incompatible dimensions"
3  else let C be a new A.rows × B.columns matrix
4      for i = 1 to A.rows
5          for j = 1 to B.columns
6              c_ij = 0
7              for k = 1 to A.columns
8                  c_ij = c_ij + a_ik · b_kj
9      return C
```

- Compatíveis: *A.colunas* == *B.linhas* (*A.columns* == *B.rows*)
- A é $p \times q$ e B é $q \times r$
- C é $p \times r$

Introdução: Multiplicação de cadeia de matrizes III

- Exemplo de multiplicação de matrizes de dimensões 2×5 e 5×3 (...)
- O tempo para calcular C é dominado pelo número de multiplicações escalares na linha 8, que é pqr

Introdução: Multiplicação de cadeia de matrizes IV

Ilustração

- Cadeia: $\langle A_1, A_2, A_3 \rangle$
- A_1: dimensão 10×100
- A_2: dimensão 100×5
- A_3: dimensão 5×50

Primeira parentização:

- $((A_1 A_2) A_3)$
- $A_1 A_2$
  - Resulta em matriz de dimensão 10×5
  - Executa $10 \cdot 100 \cdot 5 = 5000$ multiplicações escalares
- Multiplicar o resultado de $A_1 A_2$ por A_3
  - Resulta em matriz dimensão 10×50
  - Executa $10 \cdot 5 \cdot 50 = 2500$ multiplicações escalares
- Total de $5000 + 2500 = 7500$ multiplicações escalares

Introdução: Multiplicação de cadeia de matrizes V

Segunda parentização:

- $(A_1(A_2A_3))$
- A_2A_3
  - Resulta em matriz de dimensão 100×50
  - Executa $100 \cdot 5 \cdot 50 = 25000$ multiplicações escalares
- Multiplicar A_1 pelo resultado de $A_2 \cdot A_3$
  - Resulta em matriz de dimensão 10×50
  - Executa $10 \cdot 100 \cdot 50 = 50000$ multiplicações escalares
- Total de $25000 + 50000 = 75000$ multiplicações escalares

Computar o produto de acordo com a primeira parentização é 10 vezes mais rápido

Introdução: Multiplicação de cadeia de matrizes VI

Problema de multiplicação de cadeias de matrizes

Dada uma cadeia $\langle A_1, A_2, ..., A_n \rangle$ de n matrizes, onde para $i = 1, 2, ..., n$, matriz A_i tem dimensão $p_{i-1} \times p_i$, expresse de modo totalmente parentizado o produto [1] $A_1 A_2 ... A_n$ de maneira a minimizar o número de multiplicações escalares.

- As dimensões das matrizes são expressas pela sequência $p = \langle p_0, p_1, ..., p_n \rangle$
- Para o exemplo: $p = \langle 10, 100, 5, 50 \rangle$

- Não estamos realmente multiplicando matrizes
- Determinar uma ordem com o custo mais baixo
- Em geral, o tempo compensa

[1] parentize totalmente o produto

Contagem do número de parentizações I

- $P(n)$: Número de parentizações alternativas de um sequência de n matrizes
- $n = 1$: Somente um modo de parentizar totalmente
- $n = 2$: 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 $k = 1, 2, ..., n-1$. Assim:

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

- Soluções para a recorrência:
  - $\Omega(4^n/n^{3/2})$
  - $\Omega(2^n)$
- 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 de programação dinâmica I

Sequência de etapas sugerida:

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

O problema deve exibir as propriedades de indicação de aplicabilidade de programação dinâmica:

- Subestrutura ótima
- Sobreposição de subproblemas

Etapa 1: A estrutura de uma parentização ótima I

- Determinamos a subestrutura ótima...
- ...e depois a usamos para construir uma solução ótima para o problema a partir de soluções ótimas para subproblemas
- Por conveniência, notação $A_{i..j}$ onde $i \leq j$, para $A_i A_{i+1} ... A_j$
- Se $i < j$ (não trivial), para parentizar $A_i A_{i+1} ... A_j$ temos de separá-los entre A_k e A_{k+1} para algum inteiro k em $i \leq k < j$
- Para algum valor de k, primeiro computamos as matrizes $A_{i..k}$ e $A_{k+1..j}$ e depois multiplicamos as duas para gerar $A_{i..j}$
- O custo dessa parentização é:
  - O custo de computar a matriz $A_{i..k}$
  - Mais o custo de computar $A_{k+1..j}$
  - Mais o custo de multiplicá-las uma pela outra

Etapa 1: A estrutura de uma parentização ótima II

Subestrutura ótima

- Suponha que para efetuar a parentização ótima de $A_i A_{i+1} ... A_j$ separamos o produto entre A_k e A_{k+1}.
- Então, o modo como posicionamos os parênteses na subcadeia precedente ("prefixa") $A_i A_{i+1} ... A_k$ dentro dessa parentização ótima de $A_i A_{i+1} ... A_j$ deve ser uma parentização ótima de $A_i A_{i+1} ... A_k$. Por quê?
- Se existisse um modo menos custoso de parentizar $A_i A_{i+1} ... A_k$ então poderíamos substituir essa parentização na parentização ótima de $A_i A_{i+1} ... A_j$ para produzir um outro modo de parentizar $A_i A_{i+1} ... A_j$ cujo custo seria mais baixo que o custo ótimo: **Uma contradição**
- Uma observação semelhante é válida para parentizar a subcadeia $A_{k+1} A_{k+2} ... A_j$ na parentização ótima de $A_i A_{i+1} ... A_j$

Etapa 1: A estrutura de uma parentização ótima III

Usar a subestrutura ótima para construir uma solução ótima para o problema a partir de soluções ótimas para subproblemas

- Vimos que qualquer solução para uma instância não trivial requer que separemos o produto e que qualquer solução ótima contém em si soluções ótimas para instâncias de subproblemas
- Podemos construir uma solução ótima para uma instância do problema:
  - Separando o problema em dois subproblemas (pela parentização ótima de $A_i A_{i+1} ... A_k$ e $A_{k+1} A_{k+2} ... A_j$)
  - Determinando as soluções ótimas para instâncias de subproblemas e
  - Depois combinando essas soluções ótimas de subproblemas
- Devemos considerar todos os lugares possíveis para separar o produto, para termos certeza de que examinamos a opção ótima

Etapa 2: Uma solução recursiva I

Definir recursivamente o custo de uma solução ótima em termos das soluções ótimas para subproblemas

- Escolhemos como subproblemas os problemas de: Determinar o custo mínimo da parentização de $A_i A_{i+1} ... A_j$ para $1 \leq i \leq j \leq n$
- Seja $m[i, j]$ o número mínimo de multiplicações escalares necessárias para computar a matriz $A_{i..j}$
- Para o problema completo, o custo mínimo para computar $A_{1..n}$ é $m[1, n]$

Etapa 2: Uma solução recursiva II

Definir $m[i, j]$ recursivamente

- Se $i = j$ o problema é trivial: Consiste de $A_{i..i} = A_i$ e nenhuma multiplicação é necessária
  - Assim, $m[i, i] = 0$ para $i = 1, 2, ..., n$
- Computar $m[i, j]$ quando $i < j$. P/obter a parentização ótima
  - Vamos considerar que separamos o produto $A_i A_{i+1} ... A_j$ entre A_k e A_{k+1}, onde $i \leq k < j$
  - Então, $m[i..j]$ é igual ao custo de computar os subprodutos $A_{i..k}$ e $A_{k+1..j}$, mais o custo de multiplicar essas duas matrizes
  - Recordando: A_i é $p_{i-1} \times p_i$
  - Computar $A_{i..k} A_{k+1..j}$ executa $p_{i-1} p_k p_j$ multiplicações escalares
  - Assim obtemos:

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

Etapa 2: Uma solução recursiva III

- Essa equação recursiva supõe que conhecemos o valor de k, o que não é verdade
- Há somente $j - i$ valores possível para k: $k = i, i+1, ..., j-1$
- A parentização ótima usa um desses valores para k
- Precisamos verificar todos eles para determinar o melhor
- Assim, a definição recursiva para o custo mínimo de parentizar $A_i A_{i+1} ... A_j$ se torna

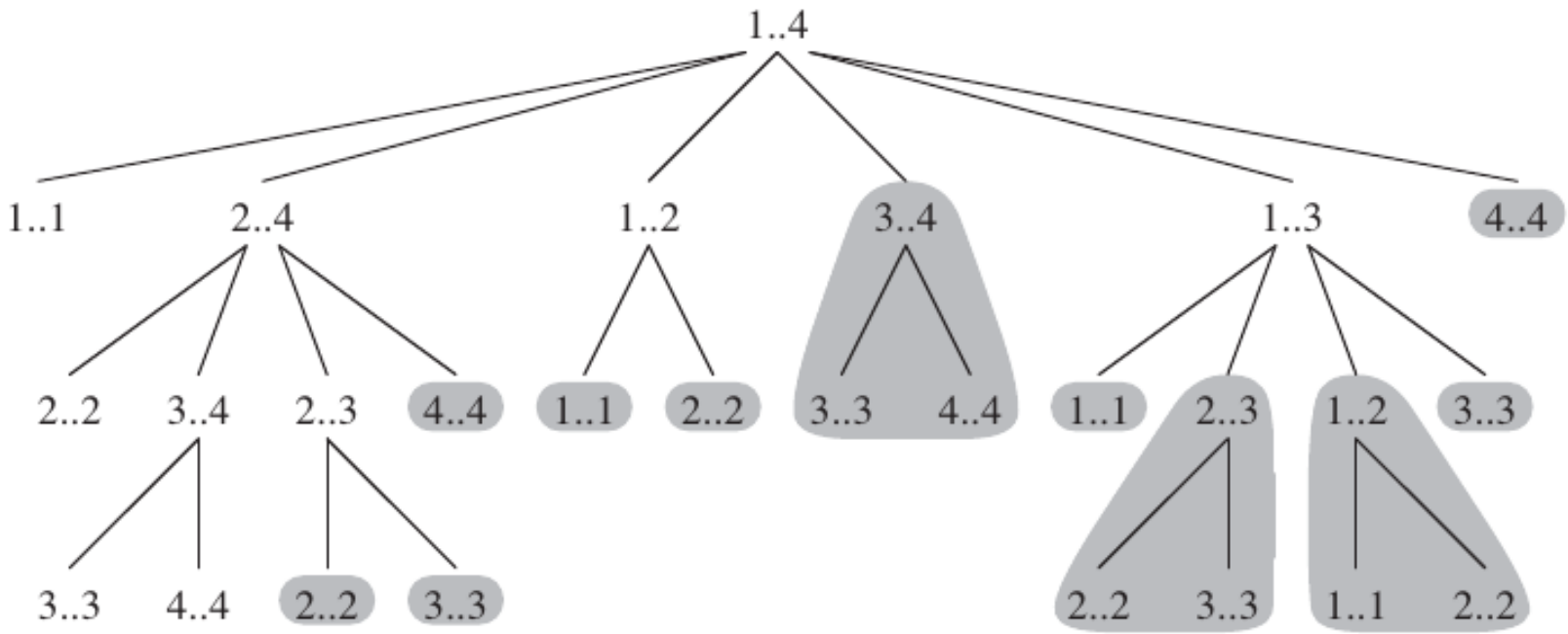
$$m[i,j] = \begin{cases} 0 & \text{se} i = j \\ \min_{i \leq k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{se} i < j \end{cases} \quad (1)$$

- Assim:
  - Os $m[i,j]$ dão os custos de soluções ótimas de subproblemas
  - Mas não dão todas as informações p/**construir** uma sol.ótima
  - Definimos $s[i,j]$ como um valor de k no qual separamos $A_i A_{i+1} ... A_j$ para obter uma parentização ótima
  - $s[i,j]$ é um k que $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$

Etapa 3: Cálculo dos custos ótimos I

- Seria fácil escrever um algoritmo recursivo com base na recorrência 1 para computar $m[1, n]$
- Demora tempo exponencial (no livro)
- Número relativamente pequeno de subproblemas distintos:
  - Um subproblema para cada escolha de i e j que satisfaz $1 \leq i \leq j \leq n$
  - Ou $\binom{n}{2} + n = \Theta(n^2)$ no total
- Um algoritmo recursivo pode encontrar cada subproblema, muitas vezes, em diferentes ramos de sua árvores de recursão

Etapa 3: Cálculo dos custos ótimos II



Etapa 3: Cálculo dos custos ótimos III

- Essa propriedade de **subproblemas superpostos** é a segunda indicação da aplicabilidade de programação dinâmica (a primeira é subestrutura ótima)

Etapa 3: Cálculo dos custos ótimos IV

- Em vez de computar a recorrência 1 recursivamente
- Computamos o custo ótimo usando uma abordagem tabular de baixo para cima
- Procedimento `Matrix-Chain-Order`
  - Supõe que a matriz A_i tem dimensões $p_{i-1} \times p_i$ para $i = 1, 2, ..., n$
  - Sua entrada é uma sequência $p = \langle p_0, p_1, ..., p_n \rangle$ onde $p.comprimento = n + 1$

Etapa 3: Cálculo dos custos ótimos V

```
MATRIX-CHAIN-ORDER(p)
 1  n = p.length − 1
 2  let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
 3  for i = 1 to n
 4      m[i, i] = 0
 5  for l = 2 to n                    // l is the chain length
 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 and s
```

Etapa 3: Cálculo dos custos ótimos VI

- O procedimento usa as tabelas auxiliares:
  - $m[1..n, 1..n]$ para armazenar os custos $m[i, j]$
  - $s[1..n-1, 2..n]$ que registra qual k alcançou o custo ótimo na computação de $m[i, j]$
    - Usaremos a tabela s para construir uma solução ótima

- Abordagem de baixo para cima: Quais as entradas da tabela no referimos para computar $m[i, j]$?
- A Equação 1 mostra que
  o custo $m[i, j]$ de computar um produto de cadeias de $j - i + 1$ matrizes **depende somente**
  dos custos de computar produtos de cadeias de menos que $j - i + 1$ matrizes
- I.e. p/ $k = i, i + 1, ..., j - 1$, 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

Etapa 3: Cálculo dos custos ótimos VII

- Preencher a tabela m de modo a resolver o problema de parentização em cadeias de comprimento crescente
- Para o subproblema de parentização ótima da cadeia $A_i A_{i+1} ... A_j$, consideramos o tamanho do subproblema é o comprimento $j - i + 1$ da cadeia

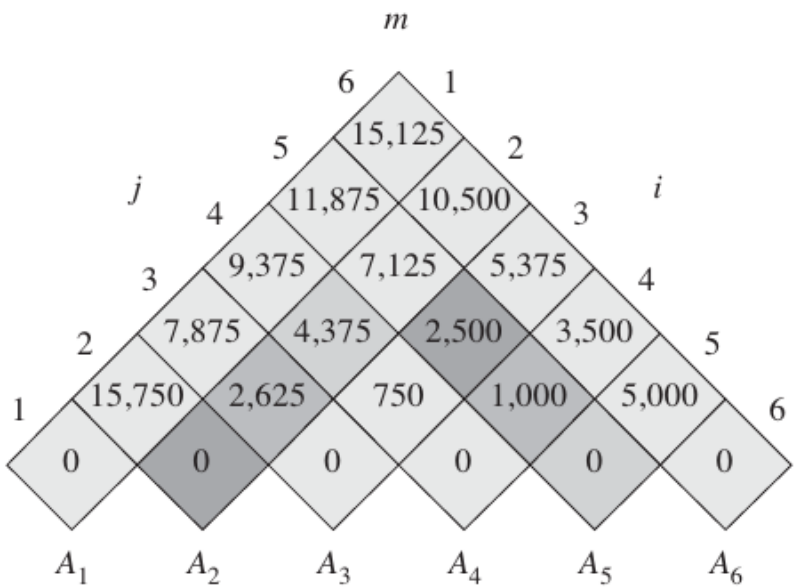
Etapa 3: Cálculo dos custos ótimos VIII

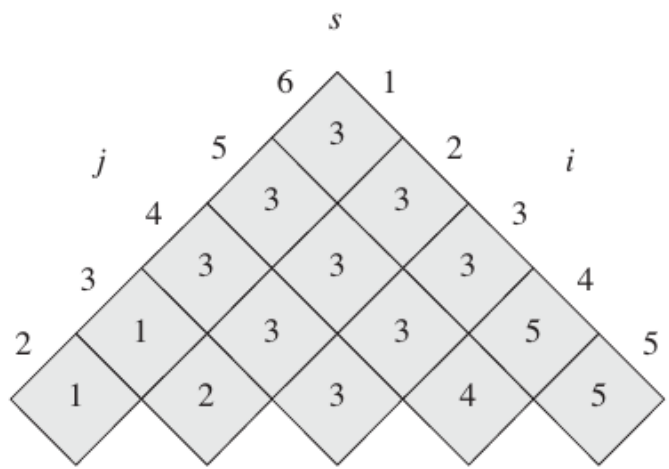
- Primeiro computa $m[i, i] = 0$, para $i = 1, 2, ..., n$ (cadeias de compr. 1)
- Usa a recorrência 1 para computar $m[i, i + 1]$ para $i = 1, 2, ..., n - 1$ (cadeias de compr. $l = 2$; 1ª pass. 5-13)
- $m[i, i + 2]$ para $i = 1, 2, ..., n - 2$ (cadeias de compr. $l = 3$; 2ª pass. 5-13)
- ...
- Custo $m[i, j]$ computado nas linhas 10-13 depende apenas de $m[i, k]$ e $m[k + 1, j]$, já calculadas
- (...)
- Exemplo:

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

Etapa 3: Cálculo dos custos ótimos IX

- Para $n = 6$
- Definimos $m[i, j]$ somente para $i \leq j$
- m: diagonal principal e triângulo superior
- s: triângulo superior
- Tabela rotacionada com diagonal principal na horizontal





Etapa 3: Cálculo dos custos ótimos X

- Custo mínimo $m[i, j]$ para multiplicar subcadeia $A_i A_{i+1} ... A_j$: Interseção partindo a nordeste de A_i e a noroeste de A_j
- Cada linha horizontal na tabela (da figura) contêm entradas para matrizes de mesmo comprimento
- O algoritmo computa as linhas de baixo para cima e da esquerda para a direita dentro de cada linha
- Computa cada entrada $m[i, j]$ usando os produtos $p_{i-1} p_k p_j$ e todas as entrada a sudoeste e a sudeste de $m[i, j]$

Etapa 3: Cálculo dos custos ótimos XI

No exemplo:

- $m[1,6] = 15125$
- Na linha 10, computando:

$$m[2,5] = min \begin{cases} m[2,2]+m[3,5]+p_1 p_2 p_5 = 0+2500+35 \cdot 15 \cdot 20 = 13000 \\ m[2,3]+m[4,5]+p_1 p_3 p_5 = 2625+1000+35 \cdot 5 \cdot 20 = 7125 \\ m[2,4]+m[5,5]+p_1 p_4 p_5 = 4375+0+35 \cdot 10 \cdot 20 = 11375 \end{cases}$$

Etapa 3: Cálculo dos custos ótimos XII

Complexidade:

- Os laços estão aninhados com profundidade 3, e cada índice de laço (l, i e k) assumem no máximo $n-1$ valores
- Tempo de execução: $O(n^3)$
- Espaço: $\Theta(n^2)$, para armazenar as tabelas m e s
- Muito mais eficiente que o tempo exponencial do método que enumera todas as possíveis parentizações e verifica cada uma delas

Etapa 4: Construção de uma solução ótima I

- `Matrix-Chain-Order` determina o número ótimo de multiplicação escalares
- Mas não mostra diretamente como multiplicar
- $s[1..n-1, 2..n]$ provém informação para mostrar como multiplicar
- $s[i, j]$ contém um k de modo que uma parentização ótima de $A_i A_{i+1} ... A_j$ separa o produto entre A_k e A_{k+1}
- Multiplicação de matrizes final na computação de $A_{1..n}$ de forma ótima é $A_{1..s[1,n]} A_{s[1,n]+1..n}$
- Podemos mostrar as multiplicações de matrizes anteriores recursivamente
- Já que $s[1, s[1, n]]$ determina a última mult. na computação de $A_{1..s[1,n]}$
- E $s[s[1, n] + 1, n]$ determina a última mult. na computação de $A_{s[1,n]+1..n}$

Etapa 4: Construção de uma solução ótima II

- Imprime uma parentização ótima de $\langle A_i, A_{i+1}, ..., A_j \rangle$
- Dados s – computado por `Matrix-Chain-Order` – e os índices i e j
- Chamada inicial `Print-Optimal-Parens(s, 1, n)`

```
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, a chamada `Print-Optimal-Parens(s, 1, 6)` imprime a parentização $((A_1(A_2A_3))((A_4A_5)A_6))$

Exercícios I

2^a ed.: 15.2-1 a 15.2-5
3^a ed.: 15.2-1 a 15.2-6 (15.2-4 da 3^a ed. é novo)

15.2-1 (3^a ed. == 2^a ed.) 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 (3^a ed. == 2^a ed.) Elabore 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)`.)

Exercícios II

15.2-3 (3^a ed. == 2^a ed.) Use o método de substituição para mostrar que a solução para a recorrência (15.6, na 3^a ed.; 15.11, na 2^a ed.) é $\Omega(2^n)$.
Obs.: a recorrência 15.6 é:

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

15.2-4 (3^a ed.) Descreva o grafo de subproblemas para a 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?

Introdução: Elementos de programação dinâmica I

Dois elementos fundamentais:

- Subestrutura ótima
- Sobreposição de subproblemas

Subestrutura ótima I

Para descobrir a **subestrutura ótima**:

- Mostrar que a solução de um problema consiste de fazer uma escolha, o que deixa um ou mais subproblemas a serem resolvidos
- Suponha que lhe foi dada a última escolha que leva a uma solução ótima
- Dada esta escolha, determine quais subproblemas surgem e como caracterizar o espaço de subproblemas resultantes
- Mostre que a solução para subproblemas usados dentro da solução ótima devem elas mesmas serem ótimas. Geralmente usa-se o método de recortar e colar:
  - Suponha que umas das soluções dos subproblemas não é ótima
  - Recorte ela
  - Cole no lugar uma solução ótima
  - Obtenha uma solução melhor para o problema original. Isto contradiz a otimalidade da solução do problema

Subestrutura ótima II

É necessário assegurar que consideramos um intervalo suficiente de escolhas e subproblemas. Tente todas as escolhas, resolva todos os subproblemas resultantes de cada escolha, e selecione a escolha cuja solução, juntamente com as soluções dos subproblemas, é a melhor.
Como caracterizar o espaço de subproblemas?

- Mantenha o espaço tão simples quanto possível
- Expanda-o quando necessário

Exemplos:

- Corte de hastes
- Multiplicação de matrizes

Subestrutura ótima III

Subestrutura ótima varia de acordo com domínios de problemas:

1. *Quantos subproblemas* são usados em uma solução ótima
2. *Quantas escolhas* são feitas para determinar qual(is) subproblema(s) usar

Exemplos:

- Corte de hastes
- Multiplicação de matrizes

Subestrutura ótima IV

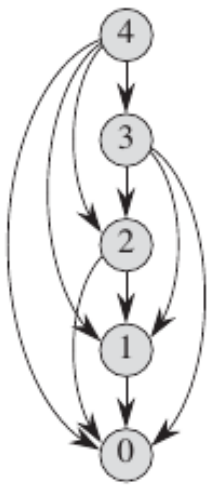
Informalmente, o tempo de execução depende do produto:
(número total de subproblemas) $\cdot$
(quantas escolhas consideramos para cada subproblema)
Exemplo:

- Corte de hastes:
  - $\Theta(n)$ subproblemas no total e
  - No máximo n escolhas para examinar cada um
  - Total: $O(n^2)$

Subestrutura ótima V

Grafo de subproblemas pode ser modo alternativo de fazer a mesma análise

- Cada vértice corresponde a um subproblema
- As escolhas para um subproblemas são as arestas que nele incidem
- No corte de hastes temos n vértices e no máximo n arestas por vértice: $O(n^2)$



Subestrutura ótima VI

- Na multiplicação de cadeia de matrizes:
  - $\Theta(n^2)$ subproblemas no total
  - Em cada um deles, no máximo $n - 1$ escolhas
  - Tempo total: $O(n^3)$ ($\Theta(n^3)$, pelo exercício 15.2-5)

Subestrutura ótima VII

Programação dinâmica frequentemente usa a subestrutura ótima de *baixo para cima* (obs.: determinamos em que ordem iremos acessar a tabela, e preenchemos ela dessa forma):

- *Primeiro* encontra soluções ótimas para subproblemas
- *Então* escolhe qual(is) usar na solução ótima para o problema

Normalmente o custo da solução do problema é igual aos custos dos subproblemas, mais um custo diretamente atribuível a escolha em si.
Exemplos:

- Corte de hastes
- Multiplicação de matrizes

Subestrutura ótima VIII

Algoritmos gulosos trabalham de forma de *cima para baixo*:

- *Primeiro* faz uma escolha que parece melhor
- *Então* resolve o subproblema resultante

Obs.: se o problema apresenta subestrutura ótima, também pode significar que uma estratégia gulosa é aplicável (que será visto posteriormente).

Subestrutura ótima IX

Análise da subestrutura ótima de um problema

- Cuidado para não presumir que a subestrutura ótima seja aplicável quando não é
- Dois problemas que parecem similar
- Considere um grafo orientado (sem peso nas arestas) $G = (V, E)$
- Encontrar um **caminho** (sequência de arestas conectadas, contendo os vértices) do vértice u até o vértice v

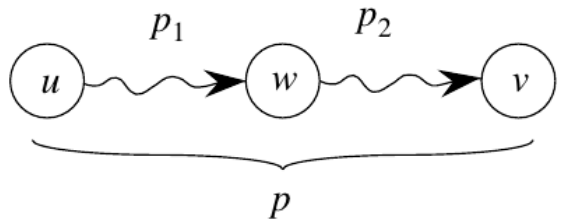
Subestrutura ótima X

Os problemas:

- **Caminho mais curto**: Encontrar um caminho $u \rightsquigarrow v$ que consista no menor número de arestas. Deve ser **simples** (sem ciclos), já que remover um ciclo de um caminho produz um caminho com menos arestas.
- **Caminho mais longo**: Encontrar um caminho *simples* $u \rightsquigarrow v$ que consista no maior número de arestas. Precisamos incluir o requisito de simplicidade, porque, do contrário, acabamos percorrendo um ciclo tantas vezes quanto quisermos para criar um caminho com um número arbitrariamente grande de arestas.

Subestrutura ótima XI

Caminho mais curto tem subestrutura ótima.



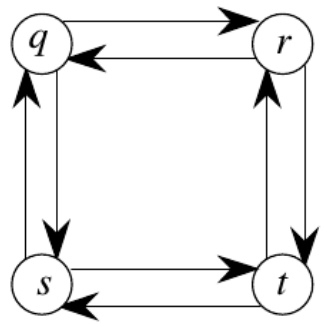
- Suponha $u \neq v$
- Suponha que p é o menor caminho $u \rightsquigarrow v$
- w é qualquer vértice em p
- p_1 é a porção de p dada por $u \rightsquigarrow w$
- Afirmamos que se p é um caminho ótimo de u para v, então p_1 deve ser um caminho mais curto de u para w. Por quê?

Subestrutura ótima XII

- Recortar e colar:
  - Se existisse outro caminho, p_1', $u \rightsquigarrow w$, mais curto
  - Poderíamos recortar p_1 e colar p_1' e o caminho resultante teria menos arestas que p
  - Mas, isto contradiz a otimalidade de p
- Simetricamente, p_2 deve ser um caminho mais curto $w \rightsquigarrow v$
- Assim, podemos encontrar um caminho mais curto $u \rightsquigarrow v$ considerando todos os vértices intermediários w, encontrando um caminho mais curto $u \rightsquigarrow w$ e um caminho mais curto $w \rightsquigarrow v$, e escolhendo w que produza o caminho mais curto global
  - Obs.: Futuramente, usaremos uma variante dessa observação de subestrutura ótima para encontrar um caminho mais curto entre cada par de vértices em um grafo ponderado e orientado.

Subestrutura ótima XIII

Caminho mais longo tem subestrutura ótima? Parece ... mas não tem



- Considere $q \to r \to t$, que é o caminho mais longo $q \rightsquigarrow t$. Seus subcaminhos são caminhos mais longos?
  - Subcaminho $q \rightsquigarrow r$ de $q \rightsquigarrow t$ é $q \to r$
  - Subcaminho $r \rightsquigarrow t$ de $q \rightsquigarrow t$ é $r \to t$
  - Caminho simples mais longo $q \rightsquigarrow r$ é $q \to s \to t \to r$
  - Caminho simples mais longo $r \rightsquigarrow t$ é $r \to q \to s \to t$
  - **Resposta: Os subcaminhos de $q \rightsquigarrow t$ não são os caminhos mais longos!**

Subestrutura ótima XIV

- E ainda, não podemos montar uma solução válida a partir de soluções para subproblemas
- Combinando os caminhos simples mais longos:
  $q \to s \to s \to t \to r \to q \to s \to t$
- Que não é simples
- De fato, este problema é *NP*-difícil (significa que é improvável que ele possa ser resolvido em tempo polinomial)

Subestrutura ótima XV

Qual é a grande diferença entre o caminho mais curto e o caminho mais longo?

- Caminho mais curto tem subproblemas **independentes**
- Solução para um subproblema não afeta a solução para outro subproblema do mesmo problema
- Caminho simples mais longo: subproblemas não são independentes
- Considere subproblemas de caminho simples mais longo $q \rightsquigarrow r$ e $r \rightsquigarrow t$
- Caminho simples mais longo $q \rightsquigarrow r$ usa s e t
- Não podemos usar s e t para resolver o caminho simples mais longo $r \rightsquigarrow t$, já que a combinação das duas soluções para subproblemas produziria um caminho que não é simples
- Mas nós temos que usar t para encontrar o caminho simples mais longo $r \rightsquigarrow t$

Subestrutura ótima XVI

- O uso de recursos (vértices) em um subproblema os torna indisponíveis para resolver outro subproblema

Exemplos:

- Multiplicação de cadeias de matrizes
- Corte de hastes

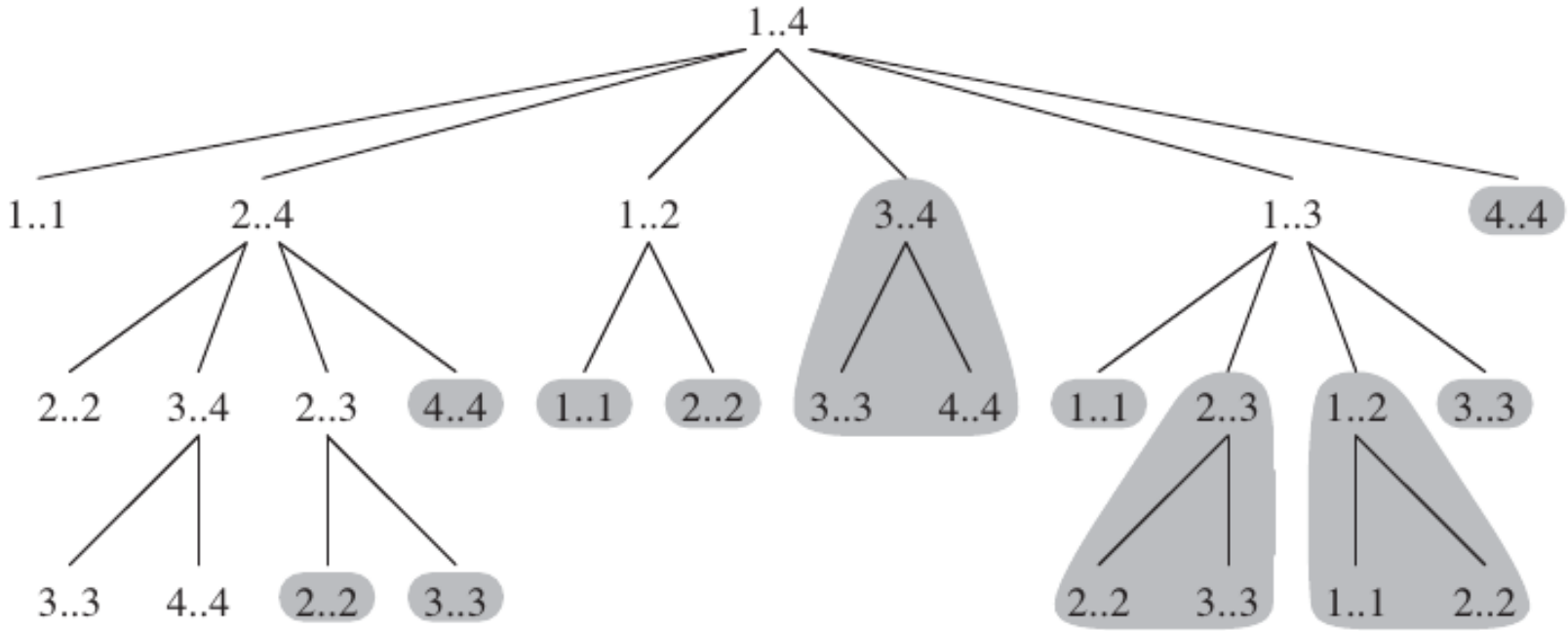
Subproblemas superpostos I

- Espaço de subproblemas deve ser "pequeno"
- Um algoritmo recursivo resolve os mesmos problemas repetidas vezes, em lugar de sempre gerar novos problemas (como fazem bons algoritmos de divisão e conquita, que é o caso do `Mergesort`)
- Quando um algoritmo recursiva reexamina o mesmo problema repetidas vezes, dizemos que o problema de otimização tem **subproblemas superpostos**

Subproblemas superpostos II

```
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]
```

Subproblemas superpostos III



- Exemplo: A entrada $m[3, 4]$
- Tempo é exponencial em n (no livro)

Reconstrução de uma solução ótima I

- 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

Memoização I

- Cada entrada da tabela contém inicialmente uma valor especial para indicar que a entrada ainda tem de ser preenchida
- Quando um subproblema é encontrado pela primeira vez durante a execução de um 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 retornamos

Memoização II

```
MEMOIZED-MATRIX-CHAIN(p)
1  n = p.length − 1
2  let m[1..n, 1..n] be a new table
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]
```

Memoização III

- Na figura, com o de `Memoized-Matrix-Chain`, as subárvores sombreadas representam valores que o procedimento consulta em vez de recalcular
- Assim como o algoritmo de programação dinâmica de baixo para cima, o `Memoized-Matrix-Chain` é $O(n^3)$ (no livro)

Exercícios I

2^a ed.: 15.3-1 a 15.3-5
3^a ed.: 15.3-1 a 15.3-6

- 15.3-4 da 2^a ed. foi removido
- 15.3-4 da 3^a ed. é igual ao 15.3-5 da 2^a ed.
- 15.3-5 e 15.3-6 são novos na 3^a ed.

Referências

- Thomas H. Cormen et al. Introdução a Algoritmos. 3ª edição em português. Capítulo 15.
- Thomas H. Cormen et al. Introdução a Algoritmos. 3ª edição em inglês. Capítulo 15.
- Thomas H. Cormen et al. Introdução a Algoritmos. 2ª edição em português. Capítulo 15.