MAC 5711 - Análise de Algoritmos

Rodrigo Augusto Dias Faria
Departamento de Ciência da Computação - IME/USP

15 de outubro de 2015

Lista 6

1. (**CLRS 15.2-1**) Determine a parentização ótima de um produto de cadeia de matrizes cuja sequência de dimensões é $\langle 5, 10, 3, 12, 5, 50, 6\rangle$.

Resposta: Como a sequência tem 7 elementos, o valor de $n = 6$. A figura 1 mostra cada iteração da entrada para geração da matriz m que contém o número mínimo de multiplicações escalares necessário para calcular o produto $A_1..A_n$.

l	i	n-l+1	j	k	q
2	1	5	2	1	150
	2	5	3	2	360
	3	5	4	3	180
	4	5	5	4	3000
	5	5	6	5	1500
3	1	4	3	1	960
	1	4	3	2	330
	2	4	4	2	330
	2	4	4	3	960
	3	4	5	3	4800
	3	4	5	4	930
	4	4	6	4	1860
	4	4	6	5	6600
4	1	3	4	1	580
	1	3	4	2	405
	1	3	4	3	630
	2	3	5	2	2430
	2	3	5	3	9360
	2	3	5	4	2830
	3	3	6	3	2076
	3	3	6	4	1770
	3	3	6	5	1830
5	1	2	5	1	4930
	1	2	5	2	1830
	1	2	5	3	6330
	1	2	5	4	1655
	2	2	6	2	1950
	2	2	6	3	2940
	2	2	6	4	2130
	2	2	6	5	5430
6	1	1	6	1	2250
	1	1	6	2	2010
	1	1	6	3	2550
	1	1	6	4	2055
	1	1	6	4	3155

Figura 1: Cálculo do número mínimo de multiplicações escalares

Os valores destacados em azul são aqueles menores para uma dada iteração i, j e, consequentemente, são armazenados na matriz $m[i, j]$, como pode ser visto na figura 2.

	1	2	3	4	5	6
1	0	150	330	405	1655	2010
2		0	360	330	2430	1950
3			0	180	930	1770
4				0	3000	1860
5					0	1500
6						0

Figura 2: Matriz m preenchida após todas as iterações

A figura 3 mostra uma segunda matriz s utilizada para armazenar o valor de k, de forma que possamos rastrear a sequência ótima para multiplicação posteriormente.

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

Figura 3: Cálculo do número mínimo de multiplicações escalares

Sendo assim, a sequência ótima é $((A_1A_2)((A_3A_4)(A_5A_6)))$.

2. (**15.2-2**) Adapte o algoritmo dado em aula para que calcule uma matriz s que pode ser usada para determinar uma ordem ótima de multiplicação das matrizes. Dê um algoritmo recursivo que recebe as matrizes $A_1, \ldots, A_n$ em uma matriz tridimensional A, recebe a matriz s e índices i e j, com $i \leq j$, e faz o produto das matrizes $A_i \ldots A_j$ usando a ordem ótima dada em s.

Resposta: Uma modificação do algoritmo dado no CLRS p. 375 já realiza o procedimento com a matriz s, sendo o parâmetro p a sequência com as dimensões das matrizes e n o tamanho de $p - 1$.

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

Agora, basta calcular o resultado das multiplicações das matrizes na sequência ótima dada por s. Quando $i = j$, estamos na base da recursão, ou seja, no ponto em que uma das matrizes de A deve ser retornada para realização da multiplicação. Caso contrário, efetuamos o produto dos pares de matrizes e do resultado do produto dos pares de matrizes que, como é realizado *bottom-up*, termina com uma única matriz quando todas as chamadas recursivas forem concluídas.

```
MATRIX-CHAIN-MULTIPLY(A, s, i, j)
1  if i == j
2       return A[i]
3  else a = MATRIX-CHAIN-MULTIPLY(A, s, i, s[i, j])
4       b = MATRIX-CHAIN-MULTIPLY(A, s, s[i, j] + 1, j)
5       return a * b
```

4. (**15.4-5**) Escreva um algoritmo que, dado n e um vetor $v[1..n]$ de inteiros, determina uma subsequência crescente mais longa de v. Seu algoritmo deve consumir tempo $O(n^2)$. Se puder, dê um algoritmo para este problema que consome tempo $O(nlgn)$.

Resposta: Para este exercício, basta utilizarmos os algoritmos LCS-LENGTH e PRINT-LCS disponíveis no CLRS 15.4, sendo que o parâmetro u é uma cópia do vetor v ordenado por algum algoritmo de ordenação que, neste caso, usamos o QUICKSORT.

```
BUILD-LONGEST-INCREASING-SUBSEQUENCE(v, n)
1  u[1..n] vetor auxiliar utilizado para copiar e ordenar v
2  for i = 1 to n
3       u[i] = v[i]
4  QUICKSORT(u, 1, u.length)
5  c, b = LCS-LENGTH(v, u)
6  PRINT-LCS(b, v, v.length, u.length)
```

Como nós queremos encontrar a maior subsequência crescente de v, e u é a cópia ordenada de v em ordem crescente, o LCS-Length deverá, então, encontrá-la dentre todas as subsequências crescentes dadas por u no vetor original v.

A tabela 1 mostra o tempo de execução do algoritmo Build-Longest-Increasing-Subsequence.

Linha	Tempo
1	$\Theta(1)$
2	$\Theta(n)$
3	$\Theta(n)$
4	$\Theta(nlogn)$
5	$O(n * n)$
6	$O(n + n)$
Total	$O(n^2 + 2n) + \Theta(nlogn) + \Theta(2n) = O(n^2)$

Tabela 1: Tempo de execução

Originalmente, o LCS-Length tem tempo de execução em função de m e n ($O(mn)$), porém, como sabemos que v e u têm o mesmo tamanho (n), teremos um tempo de execução $O(n^2)$. A mesma analogia vale para o algoritmo Print-LCS que é executado na linha 6.

5. Quantas árvores de busca binária existem que guardam 6 diferentes chaves?

Resposta: Vimos em sala que podemos representar 3 nós em 5 árvores de busca binária distintas.

Vamos definir por $t(n)$ o número de diferentes árvores de busca binária para n nós. Sendo assim, o número de diferentes árvores de busca binária que podemos obter deve ter uma raiz, uma subárvore à esquerda com i nós e outra à direita com $n - 1 - i$ nós para cada i, o que nos dá:

$$t(n) = t(0)t(n-1) + t(1)t(n-2) + ... + t(n-1)t(0)$$

Logo, podemos estabelecer $t(n)$ como uma recorrência:

$$t(n) = \begin{cases} 1, & n = 0 \text{ ou } n = 1 \\ \sum_{i=1}^{n} t(i-1)t(n-i), & n > 1 \end{cases}$$

A base da recorrência nos diz que há apenas uma árvore de busca binária com nenhum ou um nó.

Sendo assim, para 6 nós, temos 132 árvores de busca binária:

$$
\begin{aligned}
&t(0) = 1\\
&t(1) = t(0)t(0) = 1\\
&t(2) = t(0)t(1) + t(1)t(0) = 2\\
&t(3) = t(0)t(2) + t(1)t(1) + t(2)t(0) = 5\\
&t(4) = t(0)t(3) + t(1)t(2) + t(2)t(1) + t(3)t(0) = 14\\
&t(5) = t(0)t(4) + t(1)t(3) + t(2)t(2) + t(3)t(1) + t(4)t(0) = 42\\
&t(6) = t(0)t(5) + t(1)t(4) + t(2)t(3) + t(3)t(2) + t(4)t(1) + t(5)t(0) = 132
\end{aligned}
$$

O número de árvores de busca binária também pode ser calculado como o n-*ésimo* número de catalão:

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{n!(n+1)!} = \frac{(2\times 6)!}{6!(6+1)!} = \frac{12\times 11\times 10\times 9\times 8\times 7!}{6!7!} = 132$$

8. Pode-se generalizar o problema da árvore de busca binária ótima para que inclua na sua descrição uma estimativa do número de buscas mal-sucedidas. Mais concretamente, nessa variante do problema, seriam dados não apenas os números de acessos a_i de cada chave i, mas também números b_i, para $i = 0, ..., n$, representando uma estimativa do número de buscas mal-sucedidas a elementos entre i e $i + 1$ (considere 0 como $-\infty$ e $n - 1$ como $+\infty$ para que b_0 e b_n sejam o que se espera). Encontre a recorrência para o custo de uma árvore de busca binária ótima para esta variante do problema. Escreva o algoritmo de programação dinâmica gerado a partir dessa recorrência. Quanto tempo consome o seu algoritmo?

Resposta: Para generalizarmos o problema da ABB ótima, tal que possamos incluir estimativas de acesso mal sucedidas, vamos pensar em uma ABB que armazena as chaves $k = \langle k_1, k_2, ..., k_n\rangle$ ordenadas e, para cada chave, tenhamos a estimativa do número de acessos a cada elemento de k dado por a_i, para todo $i = 1, 2, ..., n$. Essa representação é a que nós já vimos em sala de aula e pode ser vista na figura 4.

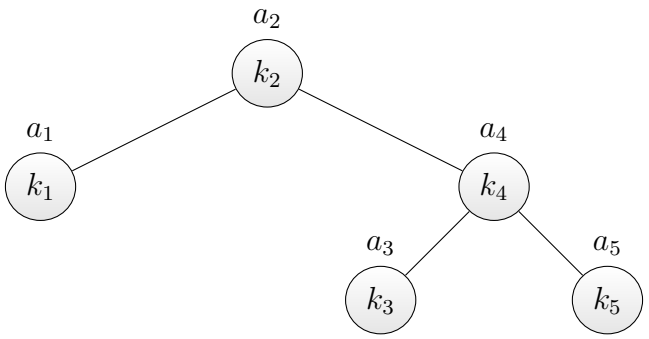

Figura 4: ABB com número de acessos a cada chave

Como temos buscas que são mal sucedidas, ou seja, valores que não estão em k, nós devemos estender a ABB de forma a incluir $n+1$ chaves que representam todas essas buscas que, neste caso, denominamos por $q_0, q_1, ..., q_n$. A figura 5 mostra essa nova representação. Note que, cada nó terminal k_i na estrutura anterior agora recebe novos nós (folhas) que representam as buscas mal sucedidas entre k_i e k_{i+1}. Vale ressaltar que, q_0 representa todas os valores menores que k_1 e q_n representa todos os valores maiores que k_n.

Da mesma forma que nas buscas bem sucedidas, temos a estimativa do número de acessos associado a cada chave q_i não encontrada em k que é dado por $b_0, b_1, ..., b_n$. Por exemplo, se tivéssemos $k = \langle 3, 5, 7, 11, 12 \rangle$, se em uma dada ABB $k_3 = 7$ e $k_4 = 11$, o nó terminal q_3 representaria os valores não encontrados $\{8, 9, 10\}$.

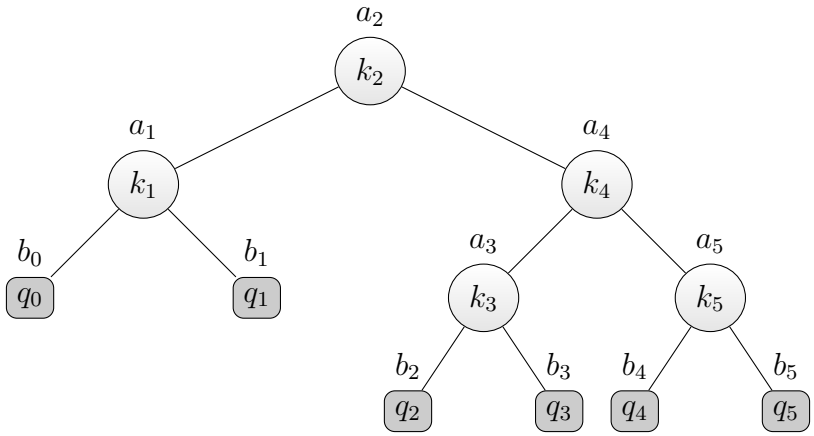

Figura 5: ABB estendida adicionando número de buscas mal-sucedidas

Visto isso, podemos definir a **subestrutura ótima** do problema: se uma ABB é ótima, então as subárvores esquerda e direita também são ótimas. Vale ressaltar que, quando escolhemos uma das chaves, digamos k_r $(i \leq r \leq j)$, como a raiz de uma subárvore $k_i, ..., k_j$, a subárvore à esquerda de k_r deve conter as chaves $k_i, ..., k_{r-1}$, bem como as representantes de buscas mal sucedidas $q_{i-1}, ..., q_{r-1}$ e a subárvore à direita de k_r deve conter as chaves $k_{r+1}, ..., k_j$ e $q_r, ..., q_j$.

Agora, devemos definir uma solução recursiva para encontrar uma ABB ótima para as chaves $k_1, k_2, ..., k_n$ cujo número esperado de comparações para uma busca bem sucedida $a_1, a_2, ..., a_n$ e o número esperado de comparações para uma busca mal sucedida $b_0, b_1, b_2, ..., b_n$ seja mínimo.

Seja $e[i, j]$ o custo esperado para uma busca em uma ABB ótima com as chaves $i, i+1, ..., j$. Vale a seguinte recorrência para $e[i, j]$:

$$e[i,j] = \begin{cases} b_{i-1}, & \text{se } i > j \\ \min\limits_{i \leq r \leq j} e[i, r-1] + e[r+1, j] + w[i,j], & \text{se } i \leq j \end{cases}$$

Onde:

$$w[i,j] = \begin{cases} b_{i-1}, & \text{se } i > j \\ w[i, j-1] + a_j + b_j, & \text{se } i \leq j \end{cases}$$

Ou seja, $w[i,j]$ é a soma das estimativas do número de buscas bem e mal sucedidas para um dado i,j:

$$w[i,j] = \sum_{l=i}^{j} a_l + \sum_{l=i-1}^{j} b_l$$

Agora, basta efetuarmos as alterações necessárias no algoritmo dado em sala de aula para calcularmos as matrizes w e e. Também vamos armazenar o valor de r em uma matriz *root*, ou seja, qual a raiz foi escolhida para uma determinada subárvore $k_i, ..., k_j$. O algoritmo EXTENDED-OPTIMAL-BST mostra o resultado.

EXTENDED-OPTIMAL-BST(a, b, n)

```
 1  for i = 1 to n + 1
 2      e[i, i − 1] = b_{i−1}
 3      w[i, i − 1] = b_{i−1}
 4  for l = 1 to n
 5      for i = 1 to n − l + 1
 6          j = i + l − 1
 7          w[i, j] = w[i, j − 1] + a_j + b_j
 8          e[i, j] = ∞
 9          for r = i to j
10              aux = e[i, r − 1] + e[r + 1, j] + w[i, j]
11              if aux < e[i, j]
12                  e[i, j] = aux
13                  root[i, j] = r
14  return e, root
```

O custo mínimo da ABB ótima é retornado na posição $e[1, n]$. A tabela 2 mostra o tempo de execução do algoritmo EXTENDED-OPTIMAL-BST. As linhas 4-13 têm 3 *loops* encadeados, que tomam no máximo n iterações cada um, o que nos dá um consumo total $\Theta(n^3)$.

Linha	Tempo
1-3	$\Theta(n)$
4-13	$\Theta(n^3)$
14	$\Theta(1)$
Total	$\Theta(n) + \Theta(n^3) = \Theta(n^3)$

Tabela 2: Tempo de execução

12. Escreva uma função que recebe como parâmetros um inteiro $n > 0$ e um vetor que armazena uma sequência de n inteiros e devolve o comprimento de uma subsequência crescente da sequência dada de soma máxima. Sua função deve consumir tempo $O(n^2)$. Modifique a sua função (sem piorar a sua complexidade assintótica) para que ela devolva também uma subsequência crescente da sequência dada de soma máxima.

Resposta: Assim como no exercício 4, podemos usar uma variação do LCS-LENGTH para resolver este problema.

Vamos, então, definir a **subestrutura ótima** do problema. Seja $X[1..n]$ um vetor com n inteiros e $Y[1..n]$ uma cópia do vetor X ordenado, temos que $Z[1..k]$ é uma subsequência crescente de X de soma máxima, se:

- $X[n] = Y[n]$, então $Z[k] = X[n] = Y[n]$ e $Z[1..k-1]$ é subsequência crescente de soma máxima de $X[1..n-1]$ e $Y[1..n-1]$
- $X[n] \neq Y[n]$, então $Z[k] \neq X[n]$ implica que $Z[1..k]$ é subsequência crescente de soma máxima de $X[1..n-1]$ e $Y[1..n]$
- $X[n] \neq Y[n]$, então $Z[k] \neq Y[n]$ implica que $Z[1..k]$ é subsequência crescente de soma máxima de $X[1..n]$ e $Y[1..n-1]$

Logo, percebemos que a subestrutura ótima do problema original não muda. Vamos definir por $c[i, j]$ a soma dos elementos de uma subsequência crescente de soma máxima de $X[1..i]$ e $Y[1..j]$. Vejamos, então, a **recorrência** para calcular o comprimento da subsequência crescente de soma máxima:

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

A diferença para a recorrência original se dá no caso 2, onde o tamanho da LCS era contabilizada. Neste caso, somamos cada elemento de X[i] em uma dada subsequência $X[i..j]$.

O algoritmo BUILD-MAX-SUM-LCS efetua, então, o cálculo desejado a partir de um vetor $X[1..n]$. Inicialmente, uma cópia de X é criada e ordenada com um algoritmo de ordenação que, neste caso, utilizamos o QUICKSORT. Posteriormente, chamamos a função LCS-MAX-SUM-LENGTH que calcula e retorna a matriz c com a soma e a matriz b para que possamos rastrear a subsequência crescente, respectivamente.

Assim que a matriz b é calculada, ela é submetida à rotina GET-LCS-MAX-SUM que retorna, então, o vetor Z com a subsequência crescente de soma máxima, bem como o tamanho da subsequência.

```
BUILD-MAX-SUM-LCS(X, n)
1  Y[1..n] vetor auxiliar utilizado para copiar e ordenar X
2  for i = 1 to n
3        Y[i] = X[i]
4  QUICKSORT(Y, 1, Y.length)
5  c, b = LCS-MAX-SUM-LENGTH(X, Y)
6  Z, l = GET-LCS-MAX-SUM(X, X.length, b)
7  return Z, l
```

LCS-MAX-SUM-LENGTH(X, Y)

```
 1  n = X.length
 2  for i = 0 to n
 3       c[i, 0] = 0
 4  for j = 1 to n
 5       c[0, j] = 0
 6  for i = 1 to n
 7       for j = 1 to n
 8            if X[i] == Y[j]
 9                 c[i, j] = c[i − 1, j − 1] + X[i]
10                 b[i, j] = " ↖ "
11            elseif c[i − 1, j] ≥ c[i, j − 1]
12                 c[i, j] = c[i − 1, j]
13                 b[i, j] = " ↑ "
14            else c[i, j] = c[i, j − 1]
15                 b[i, j] = " ← "
16  return c, b
```

A última linha do algoritmo GET-LCS-MAX-SUM nos devolve a partição de Z que foi preenchida e o tamanho do vetor resultante, ou seja, $n - k$.

GET-LCS-MAX-SUM(X, n, b)

```
 1  k = n
 2  i = n
 3  j = n
 4  while i > 0 and j > 0
 5       if b[i, j] == " ↖ "
 6            Z[k] = X[i]
 7            k = k − 1
 8            i = i − 1
 9            j = j − 1
10       elseif b[i, j] == " ↑ "
11            i = i − 1
12       else j = j − 1
13  return Z[k..n], n-k
```

Sabemos que o tempo de execução do QUICKSORT, LCS-LENGTH e GET-LCS é $\Theta(nlogn)$, $\Theta(mn)$ e $O(m+n)$, respectivamente. As alterações que fizemos em LCS-MAX-SUM-LENGTH e GET-LCS-MAX-SUM das versões originais dos respectivos algoritmos não alteram o comportamento assintótico.

Vale lembrar que, o comportamento assintótico do LCS-LENGTH e GET-LCS são em função de m e n originalmente, como ambos vetores X e Y são de tamanho n, o tempo de execução passa a ser em função de n, exclusivamente.

Portanto, o consumo de tempo total do BUILD-MAX-SUM-LCS é dado na tabela 3:

Linha	Tempo
1	$\Theta(0)$
2-3	$\Theta(n)$
4	$\Theta(nlogn)$
5	$\Theta(n^2)$
6	$O(n+n)$
7	$\Theta(1)$
Total	$\Theta(n) + \Theta(nlogn) + \Theta(n^2) + O(2n) = \Theta(n^2)$

Tabela 3: Tempo de execução

14. Problema 14-4 do CLRS (Planejando 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, a relação de supervisores forma uma árvore com raiz no presidente. O pessoal do escritório classificou cada funcionário com uma avaliação de sociabilidade, que é um número real. Para tornar a festa divertida para todos os partocipantes, o presidente não deseja que um funcionário e seu supervisor imediato participem.

O professor Stewart recebe uma árvore que descreve a estrutura hierárquica da corporação, usando a representação de filho da esquerda, irmão da direita, usanda para armazenamento de árvores enraizadas (olhe na seção 10.4 se precisar). Cada nó da árvore contém além dos ponteiros, o nome de um funcionário e a ordem de sociabilidade desse funcionário. Escreva um algoritmo para compor a lista de convidados que maximize a soma das avaliações de sociabilidade dos convidados. Analise o tempo de execução do seu algoritmo.

Resposta: Vamos resolver esse problema com a técnica de programação dinâmica.

Sub-estrutura ótima: Seja T a árvore hierarquica da empresa, sabemos que dado a solução ótima S que maximiza a soma de sociabilidade da empresa, se a raiz de T pertece a essa solução S a sociabilidade dessa solução é a soma da sociabilidade do presidente (raiz da árvore) e das soluções ótimas das subárvores enraizadas em seus subordinados diretamente, note que seus subordinados não podem ser convidados. Se o presidente não pertence a solução ótima desse problema então a sociabilidade da solução ótima é a soma das sociabilidades das soluções ótimas das subárvores enraizadas em seus subordinados direitos. **Prova:** Vamos assumir por contradição que na solução ótima, dado as subárvores enraizadas nos subordinados diretos do presidente exista uma delas que não seja ótima, então com certeza podemos trocar essa subárvore que não é ótima pela subárvore ótima e conseguir uma solução com sociabilidade maior que a solução ótima assumida que é uma contradição! Já que a solução ótima maximiza a sociabilidade.

Recorrência: Vamos definir identificar cada nó de nossa árvore como o nome do funcionário que o nó representa. Então definimos $c[x]$ como:

$c[x]$:= é a soma maxíma da sociabilidade dos convidados da festa da árvore enraizada em x tal que dado que um nó foi convidado seu pai não foi.

definimos $p(x)$ como o grau de sociabilidade do nó x, $netos(x)$ como a operação que devolve os netos do nó x na árvore, e $filhos(x)$ a operação que devolve os nós filhos de x na árvore. Também definimos que $netos(x) = \emptyset$ e $filhos(x) = \emptyset$ quando x não tem nós netos ou filhos respectivamente.

Agora definimos a recorrência da seguinte forma:

$$c[x] = \begin{cases} 0 & \text{,se } x = \emptyset \\ p(x) & \text{,se } x \text{ é uma folha} \\ \max\left(p(x) + \sum_{y \in netos(x)} c[y], \sum_{y \in filhos(x)} c[y]\right) & \text{,caso contrário} \end{cases}$$

Algoritmo de programação dinâmica: Abaixo segue o algoritmo de programação dinâmica.

15. PC 111105 (Cortes de tora) Você deve cortar uma tora de madeira em vários pedaços. A empresa mais em conta para fazer isso é a *Analog Cutting Machinery* (ACM), que cobra de acordo com o comprimento da tora a ser cortada. A máquina de corte deles permite que apenas um corte seja feito por vez. Se queremos fazer vários cortes, é fácil ver que ordens diferentes destes cortes levam a preços diferentes. Por exemplo, considere uma tora com 10 metros de comprimento, que tem que ser cortada a 2, 4 e 7 metros de uma de suas extremidades. Há várias possibilidades. Podemos primeiramente fazer o corte dos 2 metros, depois dos 4 e depois dos 7. Tal ordem custa 10+8+6 = 24, porque a primeira tora tinha comprimento 10, o que restou tinha 8 metros de comprimento e o último pedaço tinha comprimento 6. Se cortássemos na ordem 4, depois 2, depois 7, pagaríamos 10 + 4 + 6 = 20, que é mais barato.
Seu chefe encomendou um programa que, dado o comprimento l da tora e k pontos $p_1, ..., p_k$ de corte da tora, encontre o custo mínimo para executar esses cortes na ACM.

Resposta: Assumindo que os pontos de corte estão ordenados, ou seja, $p_1 \leq p_2 \leq p_3 \leq ... \leq p_n$, podemos olhar este problema de forma semelhante ao da ABB ótima.

Antes de mais nada, devemos inserir o valor 0 na posição 0 de p e o tamanho l da tora na posição p_{k+1}, para que possamos calcular o tamanho restante da tora após um corte.

Podemos. então, caracterizar a **suesbtrutura ótima** da seguinte maneira: se o custo de corte da tora t de t$[i..j]$ é mínimo, então, o custo de corte da tora de $t[i..m]$ e $t[m..j]$ também é mínimo.

Vamos definir por $c[i, j]$ o custo mínimo de corte da tora de $t[i..j]$. Vejamos agora a **recorrência** para calcular o custo mínimo de corte da tora:

$$c[i,j] = \begin{cases} 0, & \text{se } i+1 = j \\ \min\limits_{i<m<j} c[i,m] + c[m,j] + p_j - p_i, & \text{se } i+1 < j \end{cases}$$

A figura 6 mostra a tabela $c[i, j]$ preenchida após todas as iterações para a entrada $p = \langle 4, 5, 7, 8\rangle$ e $l = 10$. A matriz será preenchida pelas diagonais, de cima para baixo. Vale lembrar que inserimos os valores 0 e l em p antes de executar a recorrência, resultando em $p' = \langle 0, 4, 5, 7, 8, 10\rangle$.

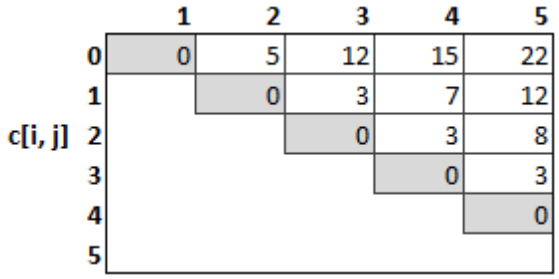

Figura 6: Matriz c preenchida após todas as iterações

O algorimto Analog-Cutting-Machinery calcula o custo mínimo de corte da tora, com base na recorrência dada por $c[i, j]$.

Analog-Cutting-Machinery(p, l)

```
 1  Insert(p, 0, 0)
 2  Insert(p, p.length, l)
 3  n = p.length
 4  for i = 0 to n − 1
 5        c[i, i + 1] = 0
 6  for k = 1 to n
 7        for i = 0 to n − k − 1
 8              j = i + k + 1
 9              c[i, j] = ∞
10              for m = i + 1 to j
11                    aux = c[i, m] + c[m, j]
12                    if aux < c[i, j]
13                          c[i, j] = aux
14              c[i, j] = c[i, j] + p[j] − p[i]
15  return c[0, n]
```

O custo mínimo do corte da tora é retornado na posição $c[0, n]$. A tabela 4 mostra o tempo de execução do algoritmo Analog-Cutting-Machinery. Como as inserções em p nas linhas 1-2 podem ser feitas em tempo $\Theta(n)$, o consumo total é $\Theta(n^3)$. já que as linhas 6-14 têm 3 *loops* encadeados que tomam, no máximo, n iterações cada um.

Linha	Tempo
1-2	$\Theta(n)$
3	$\Theta(1)$
4-5	$\Theta(n)$
6-14	$\Theta(n^3)$
15	$\Theta(1)$
Total	$\Theta(3n) + \Theta(n^3) = \Theta(n^3)$

Tabela 4: Tempo de execução

21. (Bandeiras) No dia da Bandeira na Rússia o proprietário de uma loja decidiu decorar a vitrine de sua loja com faixas de tecido das cores branca, azul e vermelha. Ele deseja satisfazer as seguintes condições: faixas da mesma cor não podem ser colocadas uma ao lado da outra. Uma faixa azul sempre está entre uma branca e uma vermelha, ou uma vermelha e uma branca. Escreva um programa que, dado o número n de faixas a serem colocadas na vitrine, calcule o número de maneiras de satisfazer as condições do proprietário.

Exemplo: Para $n = 3$, o resultado são as seguintes combinações: BVB, VBV, BAV, VAB.

Resposta: Podemos pensar no processo de formação da combinação das cores da seguinte maneira: a primeira posição, obrigatoriamente, deve conter um B ou um V. Vamos fazer uma simulação iniciando com B que, ao final, basta multiplicarmos por 2 para termos o número total de combinações. No segundo nível teremos então duas possibilidades: V e A.

Percebemos então que, a cada vez que temos um A, temos apenas uma possibilidade no nível seguinte. Quando temos um B ou um V, temos duas novas possibilidades. Devemos ter o cuidado ao finalizar as combinações para que não haja nós adjacentes com o mesmo valor e, também, nenhuma terminação em A.

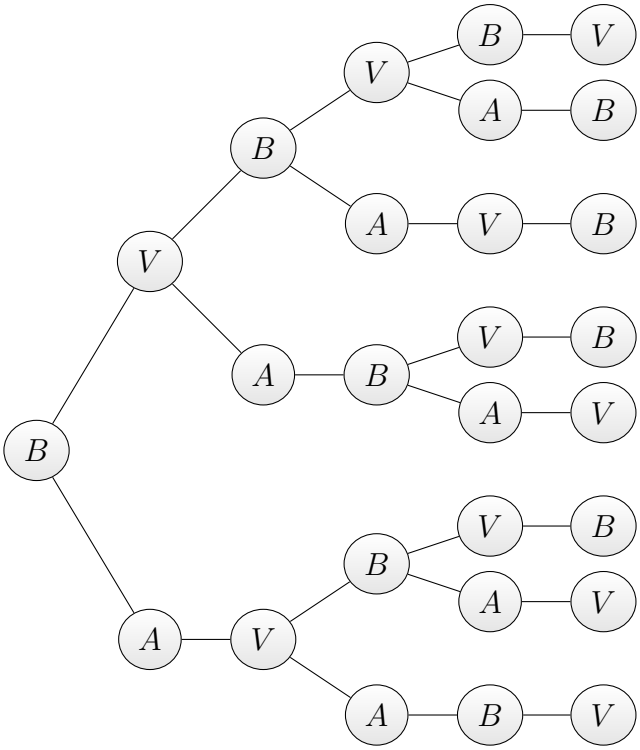

Figura 7: Árvore de combinações possíveis para $n = 6$

Logo, percebemos que o número de combinações de $1..n-1$ cresce conforme a série de Fibonacci e, portanto, podemos calcular o número de maneiras de satisfazer as condições do

proprietário com a própria recorrência que calcula a série de Fibonacci:

$$t(n) = \begin{cases} 0, & \text{se } n = 0 \\ 1, & \text{se } n = 1 \\ t(n-1) + t(n-2) + 1, & \text{se } n \geq 2 \end{cases}$$

Como temos duas possibilidades (iniciando a série com B ou V), basta multiplicarmos o resultado da série de Fibonacci(n) por 2 e teremos, então, o número de diferentes maneiras que podemos organizar as faixas de tecido na vitrine. O algoritmo ARRANGE-FLAGS mostra a solução que, na verdade, é uma pequena alteração do algoritmo original de Fibonacci para programação dinâmica.

```
ARRANGE-FLAGS(n)
1  f[0] = 0
2  f[1] = 1
3  for i = 2 to n
4        f[i] = f[i − 1] + f[i − 2]
5  return 2 * f[n]
```

Sendo assim, no exemplo supra citado, para $n = 6$, temos que o número de combinações possíveis é 16.

Obviamente o consumo de espaço e tempo do algoritmo é $\Theta(n)$, assim como é o original.

CLRS (Outros)

A.1-7 Avalie o produtório $\prod_{k=1}^{n} 2(4^k)$.

$$\prod_{k=1}^{n} 2(4^k) = \prod_{k=1}^{n} 2((2^2)^k) = \prod_{k=1}^{n} 2(2^{2k}) = \prod_{k=1}^{n} 2^{2k+1}$$

Se avaliarmos o produtório para $n = 3$, por exemplo:

$$\prod_{k=1}^{n} 2^{2k+1} = 2^{2+1} \times 2^{4+1} \times 2^{6+1}$$

Percebemos que o expoente de 2 cresce em uma série aritmética:

$$\sum_{k=1}^{n} 2k + 1 = \sum_{k=1}^{n} 2k + \sum_{k=1}^{n} 1 = 2\sum_{k=1}^{n} k + n = 2(\frac{n(n+1)}{2}) + n = n(n+2)$$

Portanto:

$$\prod_{k=1}^{n} 2(4^k) = 2^{n(n+2)}$$