Programação Dinâmica

Programação Dinâmica

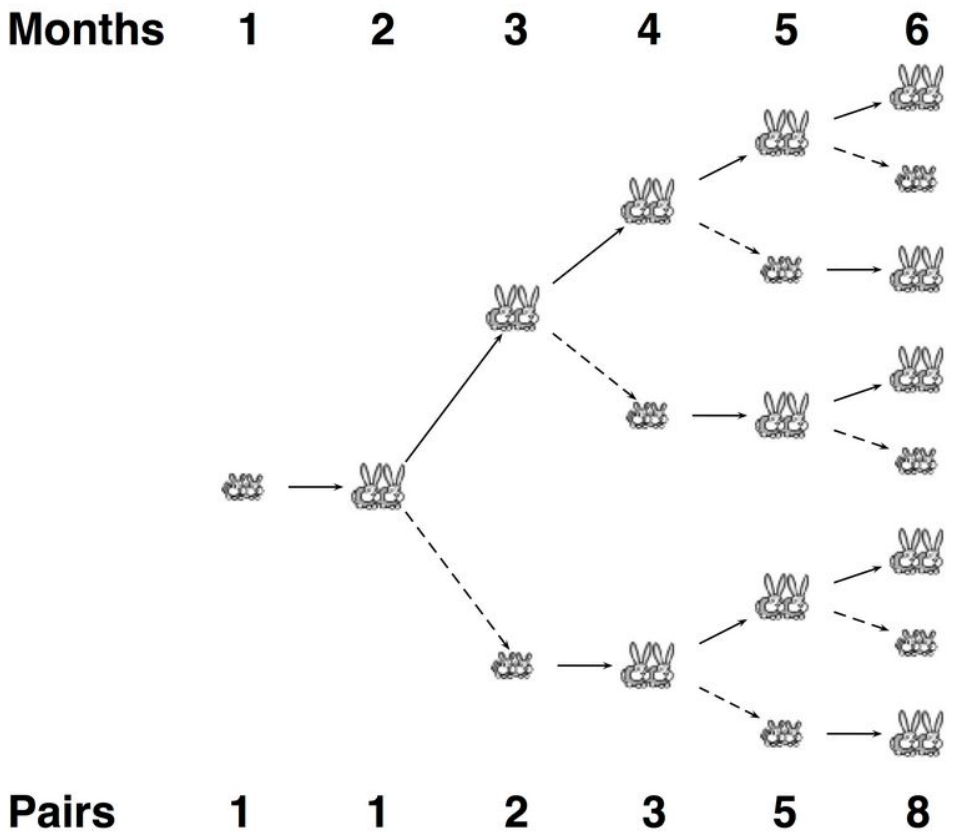
- Problemas de otimização
  - Requer soluções que retornam a melhor solução
  - Problemas difíceis
  - Algoritmos gulosos: nem sempre garantem a melhor a solução
  - Algoritmos de busca exaustiva encontram o resultado ótimo, mas muitas vezes é impraticável
  - Programação dinâmica permite o projeto de algoritmos customizados que buscam todas possibilidades enquanto armazenam resultados para evitar cálculos repetitivos

Programação Dinâmica (DP)

- Paradigma de algoritmos aplicável a uma ampla gama de problemas
  - Identificação de uma coleção de subproblemas do problema maior
    - Subestrutura ótima: as soluções ótimas do problema incluem soluções ótimas de subcasos
    - Sobreposição de subproblemas: O cálculo da solução por recursão implica no recálculo de subproblemas:
  - Resolução dos menores primeiro
  - Uso das respostas aos menores problemas para responder problemas maiores
    - Memorização
  - Duas formas de se aplicar DP
    - Top-down recursivo + memorização
    - Bottom-up

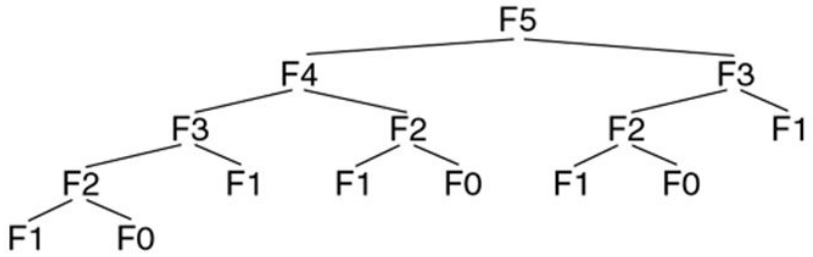
Sequência de Fibonacci

- Reprodução de uma população de coelhos



Exemplo

- Fibonacci




```
Fibonacci(n){
   if n≤1 then
      return n;
   else
      return (Fibonacci(n-1)+Fibonacci(n-2));
}
```

$$T(n) = \begin{cases} \Theta(1) \text{ se } n \leq 1 \\ T(n-1) + T(n-2) + \Theta(1) \text{ se } n > 1 \end{cases}$$

Número de chamadas recursivas = Número de Fibonacci

Complexidade de tempo e espaço: $f(n) = O(\Phi^n)$

$$\Phi = \left(\frac{1+\sqrt{5}}{2}\right) = 1{,}61803 \ldots$$

Golden ratio

Complexidade Exponencial!

Top-down com memorização

- Solução recursiva com memorização
  - Antes de fazer as chamadas recursivas para resolver um subproblema, checa em uma tabela se o resultado deste subproblema já foi resolvido
    - Caso já estiver resolvido, retornar o valor calculado
    - Senão, resolve recursivamente

Fibonacci com memorização

```
long f[46]; // inicializado com -1 em todas posições

long fib (int n)
{
   int x;
   if(f[n] >= 0) return f[n]; // se f[n] já foi computado
    if(n <= 1) x = n;
   else x = fib(n-1) + fib(n-2);
   f[n] = x;
   return(f[n]);
}
```

- Quantas chamadas recursivas?

Programação dinâmica

- Solução bottom-up
  - Determinar a ordem para se resolver os subproblemas
    - Resolver subproblemas menores primeiro
  - Quando for resolver um subproblema maior, as soluções dos subproblemas menores que a solução do subproblema maior depende já estarão calculados
  - É preciso saber quais são os subproblemas e como encontrar as suas soluções
  - Cada problema possui uma certa forma de se acessar os subproblemas
    - É importante conhecer a solução recursiva para entender a estrutura do problema
    - Sabendo a recursão, podemos obter a solução iterativa

Fibonacci por DP

```
long fib_dp(int n)
{
   int i;
   long f[46]; /* array to cache computed fib values */
   f[0] = 0;
   f[1] = 1;
   for (i=2; i<=n; i++) f[i] = f[i-1]+f[i-2];
   return(f[n]);
}
```

- Calculando o fibonacci na ordem inversa elimina a necessidade da recursão
- Qual a complexidade?

Fibonacci – outra solução

- Não é preciso armazenar todos os valores intermediários durante toda a execução

```
long fib_ultimate(int n)
{
   int i; /* counter */
   long back2=0, back1=1; /* last two values of f[n] */
   long next; /* placeholder for sum */
   if (n == 0) return (0);
   for (i=2; i<n; i++) {
      next = back1+back2;
      back2 = back1;
      back1 = next;
    }
   return(back1+back2);
}
```

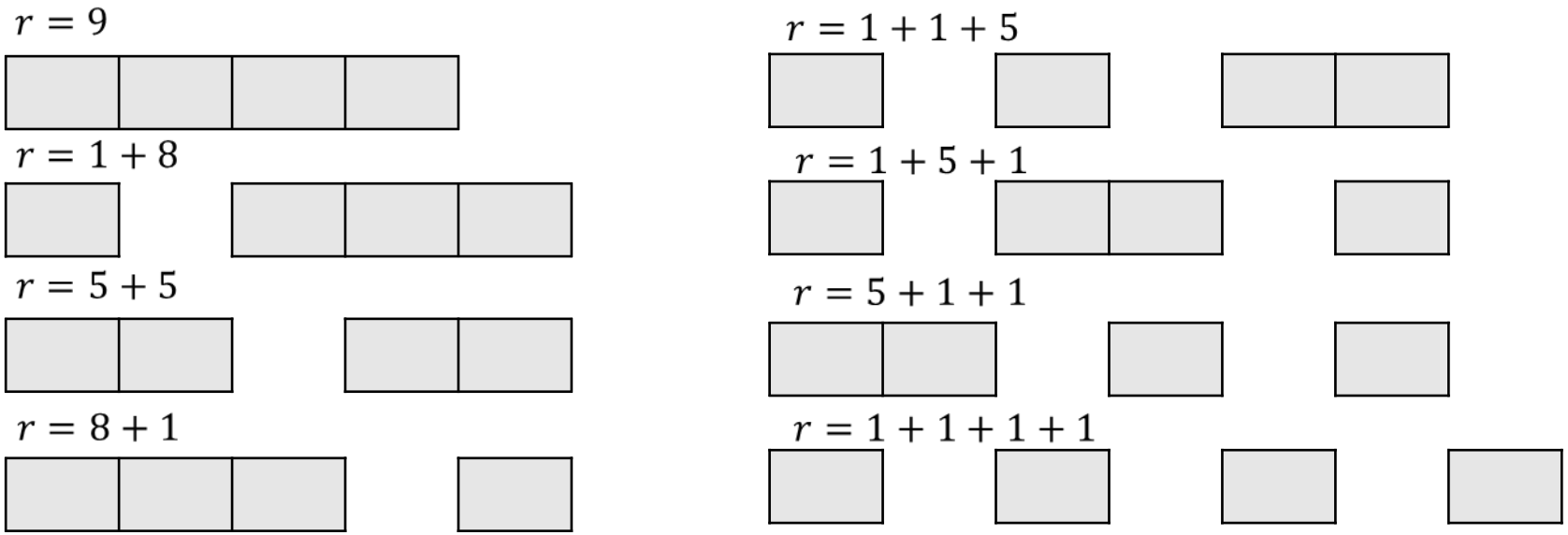
Cortando Barras

- Problema: Dada uma barra de aço de comprimento n e uma tabela de preços p_i para $i = 1, 2, \ldots, n$ que uma empresa cobra por uma barra de aço de comprimento i, determinar a receita máxima r_n que essa empresa pode obter ao se cortar a barra de comprimento n.

Exemplo

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

- $n = 4$



Obtendo a solução ótima

- A cada posição i distante i metros da extremidade esquerda para $i = 1,2,..,n-1$, temos 2 opções: cortar ou não cortar nesta posição. Então, existem 2^{n-1} diferentes possibilidades.
- Seja uma decomposição ótima com a receita máxima r_n

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

Subestrutura ótima

$$r_1 = p_1 = 1$$
$$r_2 = \max(p_2, r_1 + r_1) = \max(5,2) = 5$$
$$r_3 = \max(p_3, r_1 + r_2, r_2 + r_1) = \max(8,6,6) = 8$$
$$r_4 = \max(p_4, r_1 + r_3, r_2 + r_2, r_3 + r_1) = \max(9,9,10,9) = 10$$
$$\dots$$
$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots, r_{n-1} + r_1)$$

i	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
p_i	1	5	8	9	10	17	17	20	24	30
r_i	1	5	8	10	13	17	18	22	25	30

Subestrutura ótima

- Para um dado comprimento n, verificamos em qual das possibilidades obtém-se a maior receita
  - Não realizar nenhum corte: p_n
  - Cortar em uma posição $1 \leq i \leq n-1$: $r_i + r_{n-i}$
    - Para cada possível posição de corte i, obtemos duas barras menores de tamanhos i e $n-i$. Para cada uma dessas barras, podemos realizar o mesmo procedimento (recursivamente)

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

  - Substrutura ótima
    - Soluções ótimas incorporam soluções ótimas de subproblemas relacionados

Subestrutura ótima

- Podemos simplificar a estrutura recursiva realizando-se o corte em uma posição i $(1 \leq i \leq n)$ e continuar os cortes apenas na segunda metade de comprimento $n - i$.
  - A solução de não se realizar nenhum corte consiste de se cortar em $i = n$.
  - Cada corte em uma posição i gera apenas um subproblema ao invés de dois.

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

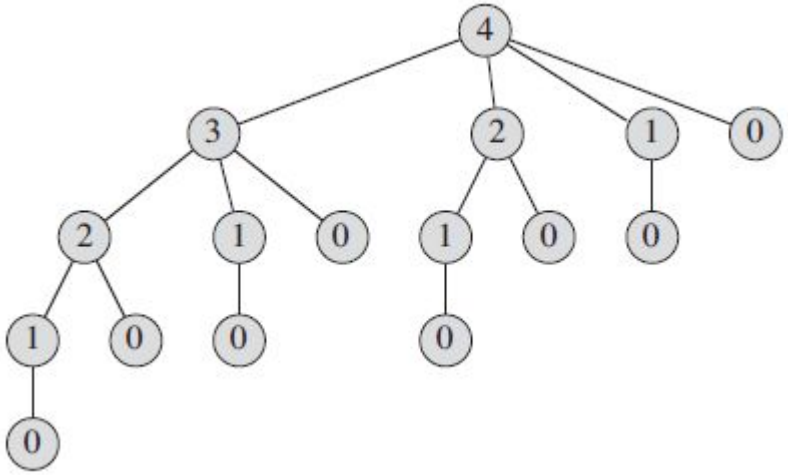
Solução recursiva

- Solução top-down

```
Cut-Rod(p, n){
  //entrada: arranjo p[1..n] de preços e um inteiro n
  //retorno: receita máxima para uma barra de comprimento n
  if (n == 0){
    return 0
  }
  q = -∞
  for i=1 to n do
    q = max(q, p[i]+Cut-Rod(p,n-i))
  return q
}
```

Quais chamadas recursivas são feitas para um $n = 4$?

Sobreposições entre subproblemas



- Árvore de recursão com os valores de n para um $n = 4$ inicial
  - Um subproblema pode estar sendo resolvido repetidas vezes
- Cada caminho da raiz a uma folha representa uma das 2^{n-1} sequências para se cortar uma barra de comprimento n.
- Recorrência do número de chamadas da função Cut-Rod()

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

- Versão recursiva + memorização

```
Memoized-Cut-Rod(p, n){
  //entrada: arranjo p[1..n] de preços e um inteiro n
  //retorno: receita máxima para uma barra de comprimento n
  //Esta função inicializa um arranjo r[0..n] e faz a chamada
  //inicial da função recursiva
  for i=0 to n do
    r[i] = -∞      //valor não calculado (negativo)
  return Memoized-Cut-Rod-Aux(p,n,r)
}
Memoized-Cut-Rod-Aux(p, n, r){
  if(r[n] >= 0) return r[n]    //valor já calculado
  if (n == 0)
    q = 0
  else{
    q = -∞
    for i=1 to n do
      q = max(q, p[i]+Memoized-Cut-Rod-Aux(p,n-i,r))
  }
  r[n] = q
  return q
}
```

- Versão bottom-up

```
Bottom-up-Cut-Rod(p, n){
  //entrada: arranjo p[1..n] de preços e um inteiro n
  //retorno: receita máxima para uma barra de comprimento n
  //Esta função utiliza um arranjo r[0..n] para armazenar
  //as soluções dos subproblemas de tamanho n=i em r[i]
  r[0] = 0
  for j=1 to n do{
    q = -∞
    for i=1 to j do
      q = max(q, p[i]+r[j-i])
    r[j] = q
  }
  return r[n]
}
```

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

Reconstruindo a solução

- Podemos guardar a escolha que fazemos para se obter as soluções ótimas dos subproblemas

```
Bottom-up-Cut-Rod-2(p, n){
  //entrada: arranjo p[1..n] de preços e um inteiro n
  //retorno: receita máxima para uma barra de comprimento n
  //Esta função utiliza um arranjo r[0..n] e s[0..n] para armazenar
  //as soluções dos subproblemas de tamanho n=i em r[i] e as decisões em s[i]
  r[0] = 0
  for j=1 to n do{
    q = -∞
    for i=1 to j do{
      if(q < p[i]+r[j-i]){
        q = p[i]+r[j-i]
        s[j] = i
      }
    }
    r[j] = q
  }
  return r[n]
}
```

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

Longest Increasing Subsequence (LIS)

- Problema: Dada uma sequência de n elementos $(x_1, x_2, \dots, x_n)$, determinar a sua maior subsequência crescente.
- A subsequência não é necessariamente contígua ou única
- Ex: $n = 8, S = \{-7, 10, 9, 2, 3, 8, 8, 1\}$
  - LIS é $\{-7, 2, 3, 8\}$ de tamanho 4.

Subsequências

- Subsequência: dada uma sequência $X = \langle x_1, x_2, \ldots, x_m \rangle$, uma outra sequência $Z = \langle z_1, z_2, \ldots, z_k \rangle$ é uma subsequência de X se existir uma sequência estritamente crescente de índices $\langle i_1, i_2, \ldots, i_k \rangle$ de X tal que para todo $j = 1,2, \ldots, k$, temos que $x_{i_j} = z_j$.
- Exemplo:

  $X = \langle A, B, C, B, D, A, B \rangle$

  $Z = \langle A, C, D, B \rangle$ é uma subsequência de X com índices $\langle 1,3,5,7 \rangle$.

Longest Increasing Subsequence (LIS)

- Solução
  - Construir uma solução recursiva que computa o tamanho da maior subsequência crescente
    - Conhecimento da solução para os primeiros $n-1$ elementos pode ajudar...
    - Precisamos saber o tamanho da maior subsequência crescente que s_n irá estender.
      - Conhecer o tamanho das maiores subsequências terminadas em cada um dos elementos $x_1, x_2 \ldots, x_{n-1}$!
    - Seja $LIS(i)$ o tamanho da maior subsequência crescente **terminada com o elemento $x[i]$**

$$\mathrm{LIS}(i) = \max_{o<j<i} (\mathrm{LIS}(j) + 1), \text{onde } x_j < x_i$$

$$\mathrm{LIS}(i) = 1, \text{para } i = 1$$

Longest Increasing Subsequence

- Seja $LIS(i)$ o LIS terminado no índice i, então temos a seguinte recorrência

$$LIS(1) = 1 \text{ // caso base}$$

$$LIS(i) = \begin{cases} 1, & se\ x_i < x_j \quad \forall j: j < i \\ \max\limits_{j<i}(LIS(j)+1) & \forall j: j < i\ e\ x_i > x_j \end{cases}$$

i	1	2	3	4	5	6	7	8	9
x_i	2	4	3	5	1	7	6	9	8

Longest Increasing Subsequence

- Seja $LIS(i)$ o LIS terminado no índice i, então temos a seguinte recorrência

$$LIS(1) = 1 \text{ // caso base}$$

$$LIS(i) = \begin{cases} 1, & se\ x_i < x_j \quad \forall j: j < i \\ \max\limits_{j<i}(LIS(j)+1) & \forall j: j < i\ e\ x_i > x_j \end{cases}$$

i	1	2	3	4	5	6	7	8	9
x_i	2	4	3	5	1	7	6	9	8
LIS(i)	1	2	2	3	1	4	4	5	5

Longest Increasing Subsequence

- Como reconstruir a maior subsequência ao invés de saber apenas o tamanho dela?
  - Manter informação auxiliar
    - Predecessor p_i: p_i é o índice do predecessor de s_i em um LSI(i)

i	1	2	3	4	5	6	7	8	9
x_i	2	4	3	5	1	7	6	9	8
LIS(i)	1	2	2	3	1	4	4	5	5
p_i	-	1	1	2	-	4	4	6	6

Longest Common Subsequence

- Considere o problema de se comparar sequências de DNA de 2 ou mais organismos. Cada sequência consiste de uma string de bases cujo alfabeto possui as bases Adenina (A), Citosina (C), Guanina (G) e Timina (T).
- Ex:
  - S_1 ACCGGTCGAGTGCGCGGAAGCCGGCCGAA
  - S_2 GTCGTTCGGAATGCCGTTGCTCTGTAAA
- Comparação das sequências para determinar qual "similar" essas sequências são.
- Diferentes formas de se definir similaridade entre S_1 e S_2:
  - Uma sequência ser substring de outra
  - Número de alterações necessárias para se transformar S_1 em S_2
  - Encontrar uma subsequência S_3 que aparece tanto em S_1 e S_2

Subsequências

- Utilizaremos a última definição de similaridade: determinar a maior subsequência comum.
- Subsequência: dada uma sequência $X = \langle x_1, x_2, \dots, x_m \rangle$, uma outra sequência $Z = \langle z_1, z_2, \dots, z_k \rangle$ é uma subsequência de X se existir uma sequência estritamente crescente de índices $\langle i_1, i_2, \dots, i_k \rangle$ de X tal que para todo $j = 1,2, \dots, k$, temos que $x_{i_j} = z_j$.
- Exemplo:

  $X = \langle A, B, C, B, D, A, B \rangle$

  $Z = \langle A, C, D, B \rangle$ é uma subsequência de X com índices $\langle 1,3,5,7 \rangle$.

Subsequência Comum

- Sejam duas sequências X e Y, uma sequência Z é uma subsequência comum de X e Y se Z é uma subsequência de ambos X e Y.
- Exemplo:

$X = \langle A, B, C, B, D, A, B \rangle$
$Y = \langle B, D, C, A, B, A \rangle$
$Z = \langle B, B, A \rangle$

Longest Common Subsequence

- Problema: Dadas duas sequências $X = \langle x_1, x_2, \dots, x_m \rangle$ e $Y = \langle y_1, y_2, \dots, y_n \rangle$, determinar o comprimento da maior subsequência comum (LCS) entre X e Y.
- No exemplo anterior, Z é LCS de X e Y?
- Exemplo:

$X = \langle A, B, C, B, D, A, B \rangle$

$Y = \langle B, D, C, A, B, A \rangle$

$Z = \langle B, C, B, A \rangle$

Encontrando o LCS

- Força Bruta:
  - Considerar todas as 2^m possibilidades de subconjuntos de índices $\{1,2,..,m\}$ de X.
- Subestrutura ótima
  - Comparar as duas últimas posições de X e Y
  - Sejam os prefixos $X_i = \langle x_1, \dots, x_i \rangle$ e $Y_j = \langle y_1, \dots, y_j \rangle$
  - Se $x_m = y_n$, então podemos incluir x_m no LCS, acrescido do LCS dos prefixos de X e Y sem esses elementos, ou seja, X_{m-1} e Y_{n-1}. Temos 1 subproblema neste caso.
  - Se $x_m \neq y_n$, então temos que resolver 2 subproblemas: encontrar um LCS de X_{m-1} e Y e encontrar um LCS de X e Y_{n-1}. O maior entre os 2 é o LCS de X e Y.

Subestrutura ótima

- Teorema: Seja $Z = \langle z_1, \ldots, z_k \rangle$ o LCS de X e Y.
  1) Se $x_m = y_n$, então $z_k = x_m = y_n$ e Z_{K-1} é um LCS de X_{m-1} e Y_{n-1}.
  2) Se $x_m \neq y_n$, então $z_k \neq x_m \Rightarrow Z$ é um LCS de X_{m-1} e Y.
  3) Se $x_m \neq y_n$, então $z_k \neq y_n \Rightarrow Z$ é um LCS de X e Y_{n-1}.

Subestrutura ótima

- Prova:
  1) Se $x_m = y_n$, então $z_k = x_m = y_n$ e Z_{K-1} é um LCS de X_{m-1} e Y_{n-1}: Primeiro mostraremos que se $x_m = y_n$, então $z_k = x_m = y_n$. Suponhamos que $z_k \neq x_m$, então como $x_m = y_n$ podemos criar $Z' = \langle z_1, \dots, z_k, x_m \rangle$, LCS de X e Y de comprimento $k + 1$. Logo, isso contradiz Z ser um LCS. Em seguida, temos que mostrar que Z_{K-1} é um LCS de X_{m-1} e Y_{n-1}. Suponhamos que exista uma subsequência comum W de X_{m-1} e Y_{n-1} de comprimento $\geq k$. Podemos criar $W' = W + x_m$ (concatenação) de comprimento $k + 1 \Rightarrow$ contradição com Z ser um LCS.
  2) Se $x_m \neq y_n$, então $z_k \neq x_m \Rightarrow Z$ é o LCS de X_{m-1} e Y: Suponha que exista uma subsequência comum W de X_{m-1} e Y de comprimento $> k$. Então W também é uma subsequência comum de X e Y.
  3) Simétrico a 2.

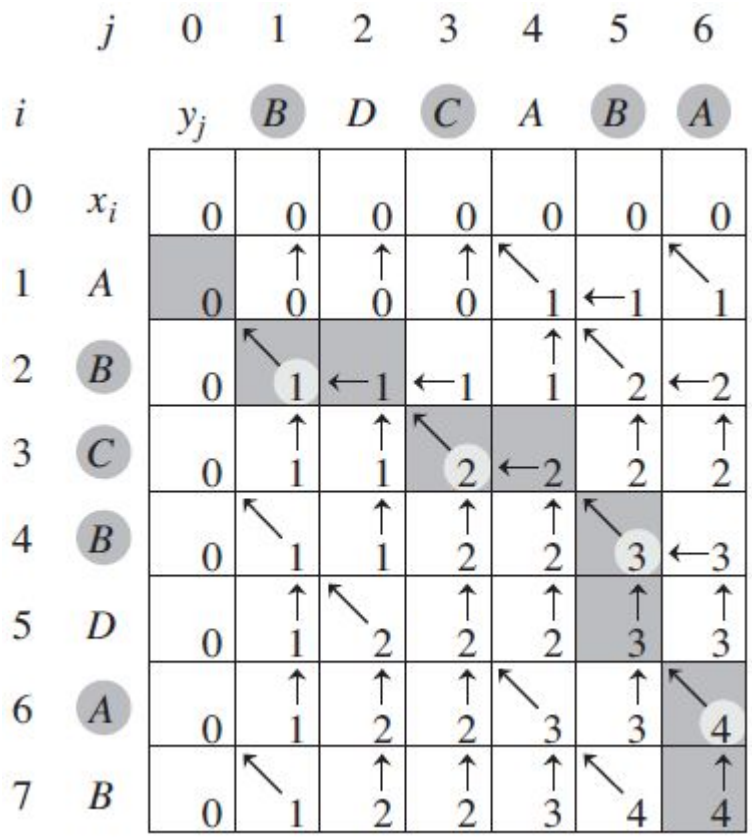
Encontrando o LCS

- Sobreposição entre subproblemas
  - Para encontrar o LCS de X e Y, podemos precisar do LCS de X_{m-1} e Y e do LCS de X e Y_{n-1}. Cada um desses subproblemas inclui o subproblema de encontrar LCS de X_{m-1} e Y_{n-1}.

Solução Recursiva

- $$LCS(X_m, Y_n) = \begin{cases} 0, & \text{se } m = 0 \text{ ou } n = 0 \\ LCS(X_{m-1}, Y_{n-1}) + 1 & \text{senão se } x_m = y_n \\ \max(LCS(X_m, Y_{n-1}), LCS(X_{m-1}, Y_n), & \text{senão se } x_m \neq y_n \end{cases}$$

		0	1	2	3	4	5	6
			B	D	C	A	B	A
0								
1	A							
2	B							
3	C							
4	B							
5	D							
6	A							
7	B							


j 0 1 2 3 4 5 6
i
y_j B D C A B A
0 x_i 0 0 0 0 0 0 0
1 A 0 ↑0 ↑0 ↑0 ↖1 ←1 ↖1
2 B 0 ↖1 ←1 ←1 ↑1 ↖2 ←2
3 C 0 ↑1 ↑1 ↖2 ←2 ↑2 ↑2
4 B 0 ↖1 ↑1 ↑2 ↑2 ↖3 ←3
5 D 0 ↑1 ↖2 ↑2 ↑2 ↑3 ↑3
6 A 0 ↑1 ↑2 ↑2 ↖3 ↑3 ↖4
7 B 0 ↖1 ↑2 ↑2 ↑3 ↖4 ↑4

```
LCS(X, Y){
 //entrada: duas strings X[1..m] e Y[1..n]
 //retorno: comprimento do LCS de X e Y e os indicadores de subproblemas
escolhidos.
 //Esta função utiliza também matrizes c[0..m][0..n] e b[1..m][1..n] para
 //armazenar as soluções dos subproblemas e as decisões tomadas, respectivamente
 for i=1 to m do c[i,0] = 0
 for j=0 to n do c[0,j] = 0
 for i=1 to m do{
   for j=1 to n do{
     if (X[i] == Y[j]){
       c[i][j] = c[i-1][j-1]+1;
       b[i][j] = '\';
     }
     else if(c[i-1][j] >= c[i][j-1]){
       c[i][j] = c[i-1][j];
       b[i][j] = '|';
     }
     else{
       c[i][j] = c[i][j-1];
       b[i][j] = '-';
     }
    }
  }
  return c,b;
}
```

Exercícios

1) Considere o problema de retornar n centavos de troco com o número mínimo de moedas. Projete um algoritmo por programação dinâmica que encontra a quantidade mínima de moedas necessárias para devolver n centavos de troco para qualquer conjunto D que inclua a moeda de 1 centavo.

Exercícios

2) Um palíndromo é uma palavra, frase ou qualquer sequência de caracteres que pode ser lida tanto da esquerda para a direita quanto da direita para a esquerda. Por exemplo, "arara" é um palíndromo, enquanto "araras" não é. Escreva um algoritmo de programação dinâmica para encontrar o palíndromo mais longo que é uma subsequência de uma dada sequência de caracteres. Assim, dada a entrada "character", seu algoritmo deve retornar "carac". Faça uma análise do consumo de tempo de sua solução.

Referências

- CLRS, Introduction to Algorithsm, 3rd ed.
  - Cap. 15, 15.1, 15.3, 15.4
- Skiena, The Algorithm Design Manual, 2nd ed.
  - 8.1, 8.2, 8.3