Análise de Algoritmos

Ronaldo Cristiano Prati

Bloco A, sala 513-2

ronaldo.prati@ufabc.edu.br

Programação dinâmica

- Armazena uma tabela com solução de (sub)problemas já resolvidos
  - Essa estratégia é chamada de **memorização**
- Usa as soluções da tabela para resolver problemas ainda não resolvidos
- Ao final, usamos a informação coletada no caminho para resolver o problema completo

Programação dinâmica

- Duas abordagens
  - Top Down: geralmente a transcrição direta de um algoritmo recursivo
  - Bottom up: inicia pelos problemas menores e usa essa solução para resolver problemas maiores

Programação dinâmica

- O nome "dinâmica" se refere ao fato que o problema é resolvido em multiplos estágios
- O nome foi atribuído (junto com a criação da técnica) nos anos de 1950
- Precisava de um "nome bonito" para conseguir financiamento da força aérea americana:

> *"It's impossible to use the word, dynamic, in the pejorative sense...I thought dynamic programming was a good name. It was something not even a Congressman could object to."*

Subsequência Comum

- A sequência BDFH é uma subsquência de ABCDEFGH
  - Observe que não precisam, necessariamente, serem contínuos
- Se X e Y são sequências, uma subsequência comum é uma subsequência de ambas as sequências.
  - BDFH é uma subsequência de ABCDEFGH e ABDFGHI

Subsequência Comum Máxima

- A maior subsequência comum é uma sequência que é comum e é a mais longa.
  - A maoir subsequência comum entre ABCDEFGH and ABDFGHI é ABDFGH.
- Problema: Dados duas sequências $X = \{X_1, X_2, \cdots, X_m\}$ e $Y = \{Y_1, Y_2, \cdots, Y_n\}$, encontrar uma subsquência de maior tamanho $LCS(X, Y)$

Aplicações

- Bioinformática:
  - Encontrar a semelhança genética entre espécies; Encontrar sequências comuns para uma detarminada característica/doença genética
- Diferença entre arquivos:
  - Comando `diff` do do unix
- Merge em controlo de versões
  - `svn`, `git`, etc.
- Processamento de Língua Natural
  - pareamento de sentenças

Maior Subsequência comum

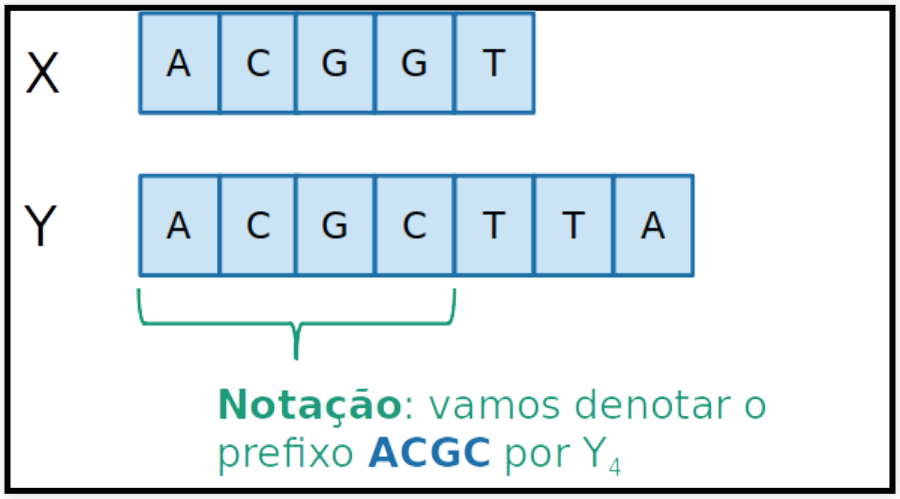
- Solução por força bruta:
  - Enumerar todas as possíveis subsequências de X. Checar se cada uma dessas subsquências também é subsequência de Y, guardando a de maior comprimento.
- Complexidade?
  - Existem 2^m subsequências de X
  - Tempo linear para verificar se a subsequência de X está em Y
  - $O(n2^m)$, portanto ineficiente

Programação dinâmica

1. Identificar subestrutura ótima
2. Encontrar uma formulação recursiva
3. Usar programação dinâmica para encontrar o valor da função ótima
4. Se necessário, armazenar informação adicional de tal maneira que no passo 3 podemos encontrar a solução ótima

Subestrutura ótima

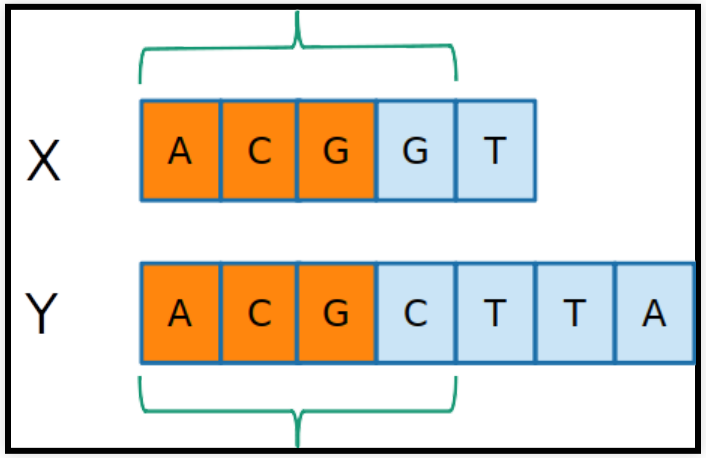
- Vamos definir um prefixo como uma subsequência que se inicia na primeira, até uma certa posição i



Subestrutura ótima

- Seja $C[i, j]$ o tamanho da maior subsqüência comum entre X_i e Y_j

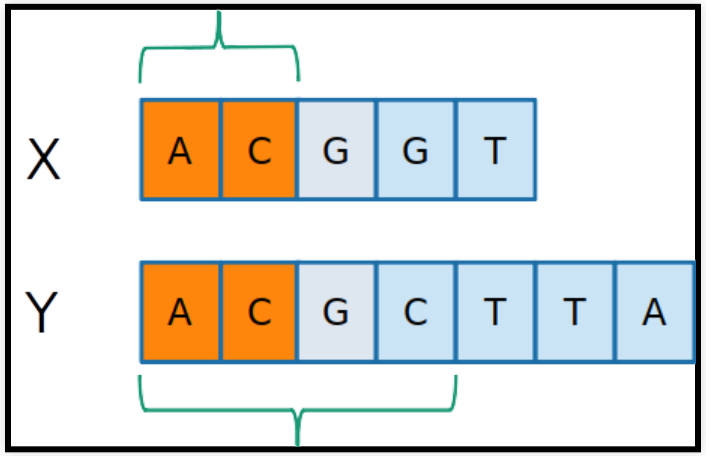
$$C[4, 4] = 3$$



Subestrutura ótima

- Seja $C[i, j]$ o tamanho da maior subsqüência comum entre X_i e Y_j

$$C[2, 3] = 4$$



Formulação recursiva

- Seja X e Y duas sequências de tamanho i e j, nas quais queremos encontrar a maior sequência comum.
- Vamos tentar escrever $C[i, j]$ em função de problemas menores
  - Seja $X[i]$ e $Y[j]$ os dois últimos caracteres da sequência:

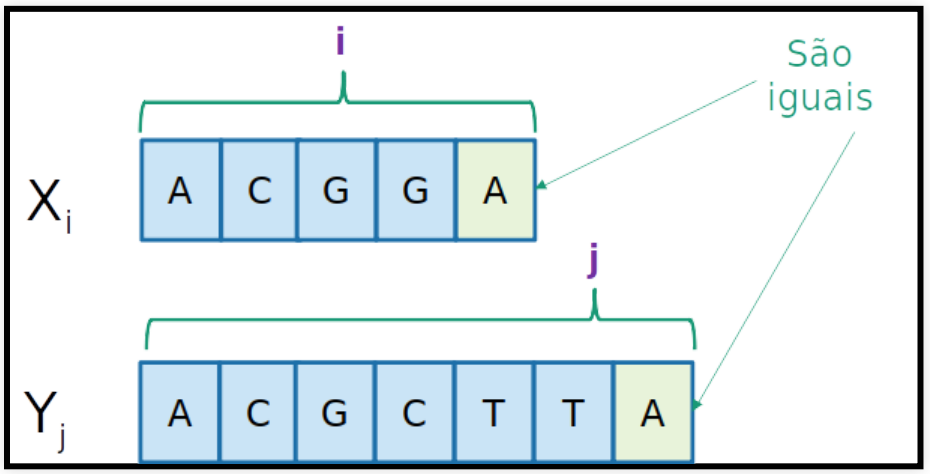
$$C_{i,j} = \begin{cases} 0 & \text{se } i = 0 \text{ ou } j = 0 \\ 1 + C_{i-1,j-1} & \text{se } X[i] = Y[j] \\ \max(C_{i-1,j}, C_{i,j-1}) & \text{se } X[i] \neq Y[j] \end{cases}$$

Formulação recursiva

- Se $X[i] = Y[j]$, então

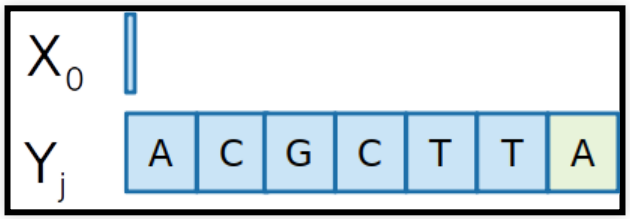
$$C_{i,j} = 1 + C_{i-1,j-1}$$

uma vez que as duas últimas posições são iguais (portanto a subsequência máxima entre os dois últimos caracteres é 1),



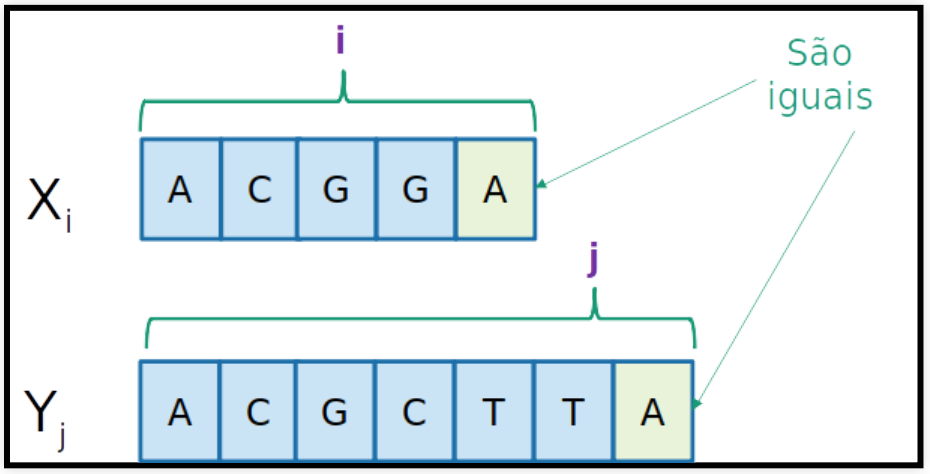
Formulação recursiva

- Se $i = 0$ ou $j = 0$, então $C[i, j] = 0$ uma vez que
  - Ou X ou Y não contém nenhum caracter, e não tem nenhuma subsequência em comum.



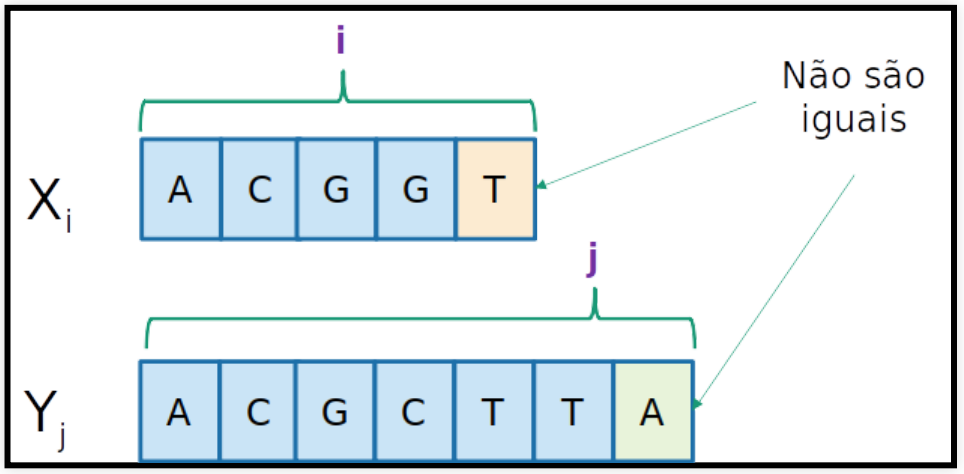
Formulação recursiva

- Se $X[i] = Y[j]$, então $C_{i,j} = 1 + C_{i-1,j-1}$
  uma vez que
  - as duas últimas posições são iguais (portanto a subsequência máxima entre os dois últimos caracteres é 1),
  - e devemos continuar verificando o restante da subsquência



Formulação recursiva

- Se $X[i] \neq Y[j]$, então $\max(C_{i-1,j}, C_{i,j-1})$ uma vez que
  - A subsequência máxima pode se dar ignorando o último caracter da primera sequência
  - A subsequência máxima pode se dar ignorando o último caracter da segunda sequência

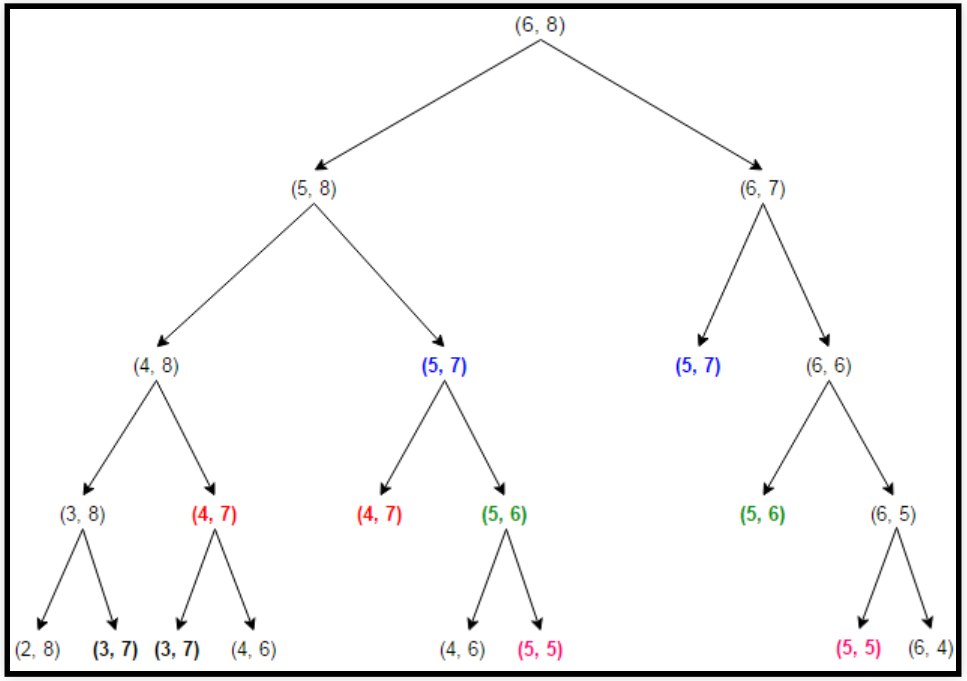


Formulação recursiva

- Uma $LCS(X, Y)$ pode ser obtida recursivamente da seguinte maneira:
  - Se $X_m = Y_n$, eles fazem parta de LCS, e deve-se continuar procurando em $LCS(X_{m-1}, Y_{n-1})$
  - Se $X_m \neq Y_n$, então temos dois subproblemas
    - Encontrar uma $LCS(X_{m-1}, Y_n)$;
    - Encontrar uma $LCS(X_m, Y_{n-1})$;
    - $LCS(X, Y)$ é a maior entre essas duas.
- A complexidade desse algoritmo é $O(2^{(m+n)})$

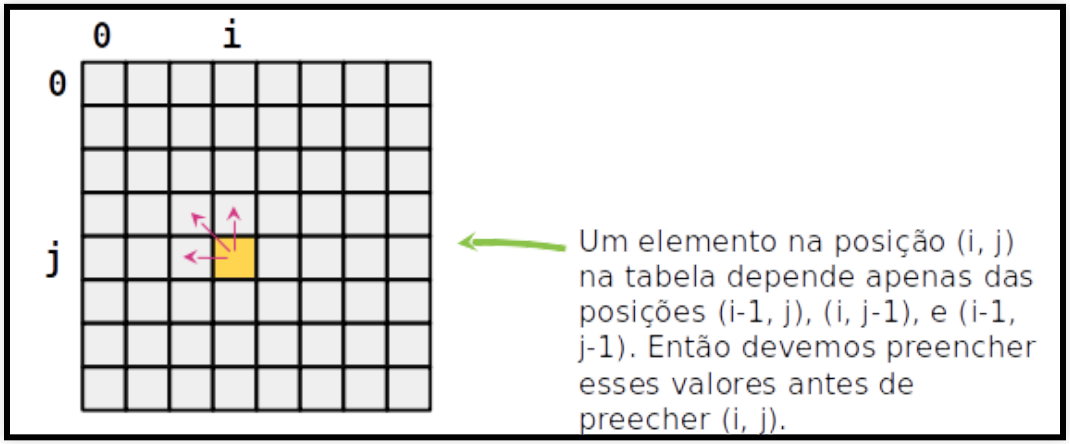
Formulação recursiva

- Observe também que há um overlap entre os subproblemas



Programação Dinâmica

- Assim com no problema da mochila inteira, vamos usar uma matriz para armazenar os valores de $C_{i,j}$ já calculados



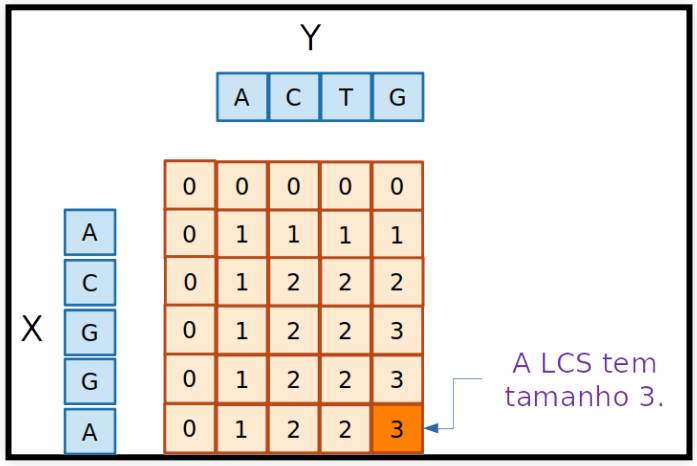
$$C_{i,j} = \begin{cases} 0 & \text{se } i = 0 \text{ ou } j = 0 \\ 1 + C_{i-1,j-1} & \text{se } X[i] = Y[j] \\ \max(C_{i-1,j}, C_{i,j-1}) & \text{se } X[i] \neq Y[j] \end{cases}$$

Programação Dinâmica

Programação Dinâmica

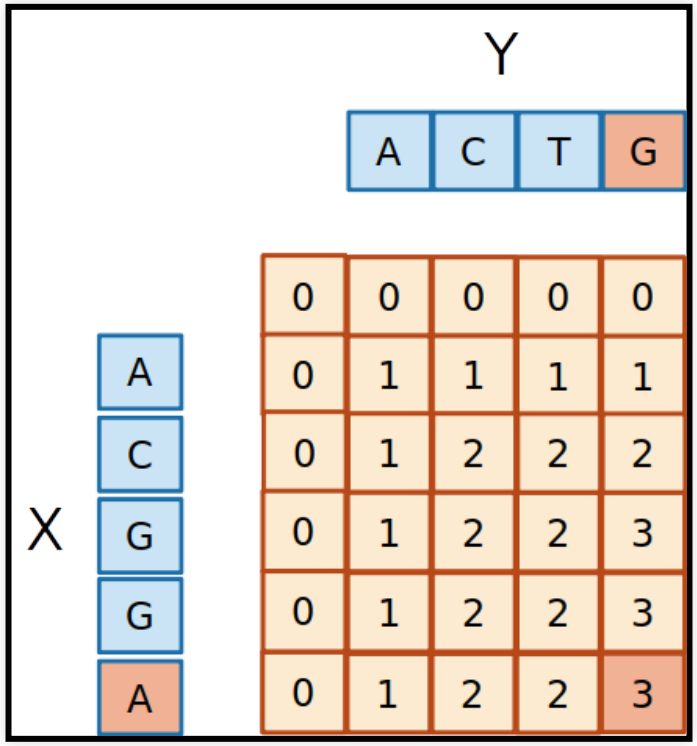
Programação Dinâmica

- O tamanho da maior subsequência comum está na última posição da matriz



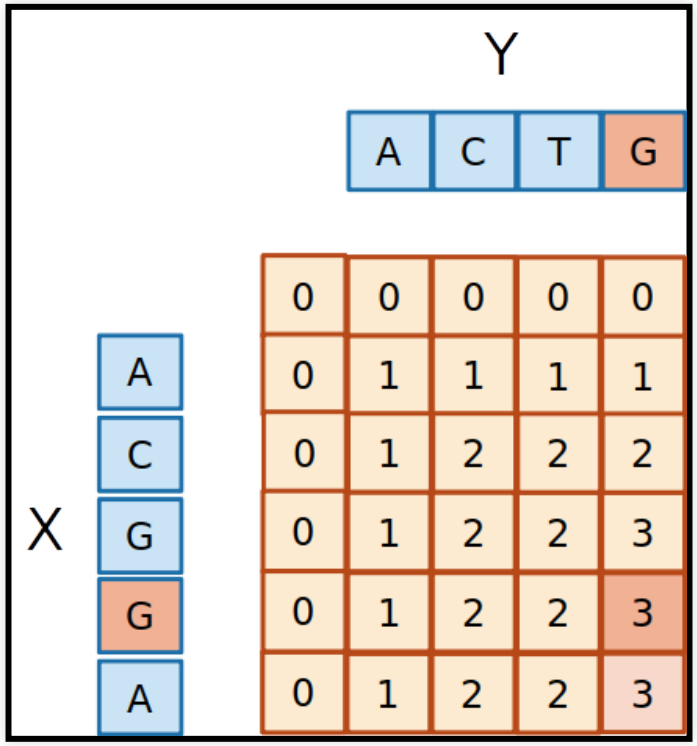
Programação Dinâmica

- Os dois caracteres são diferentes, então o 3 deve ter vindo do 3 de cima



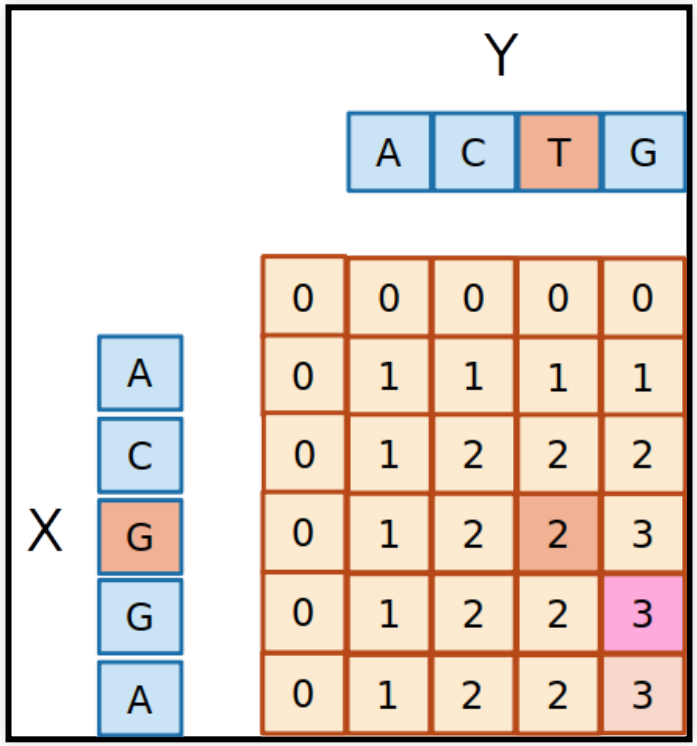
Programação Dinâmica

- Os dois caracteres são iguais, então o 3 deve ter vindo desta célula



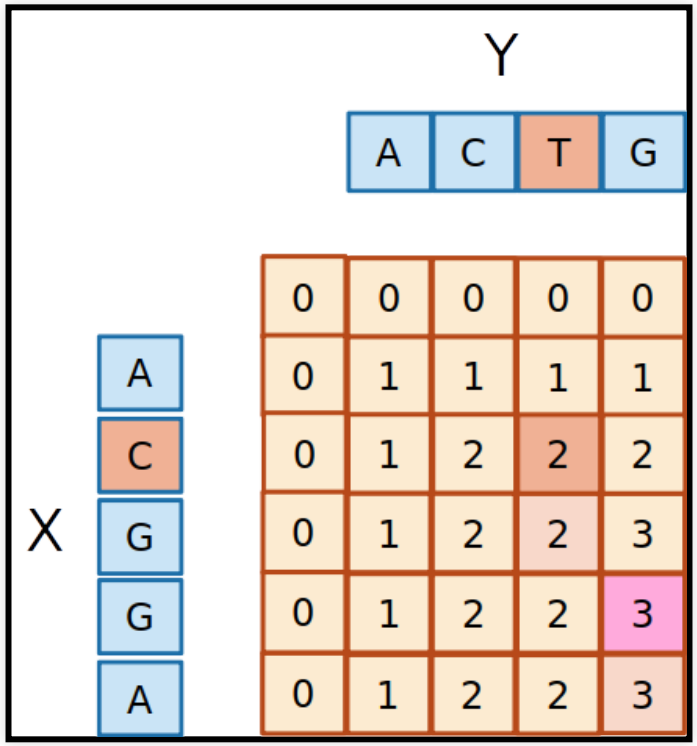
Programação Dinâmica

- Como a última iteração foi um match, devemos nos mover na diagonal
- Como os dois elementos são diferentest, o 2 pode ter vindo tanto da esquerda quanto de cima



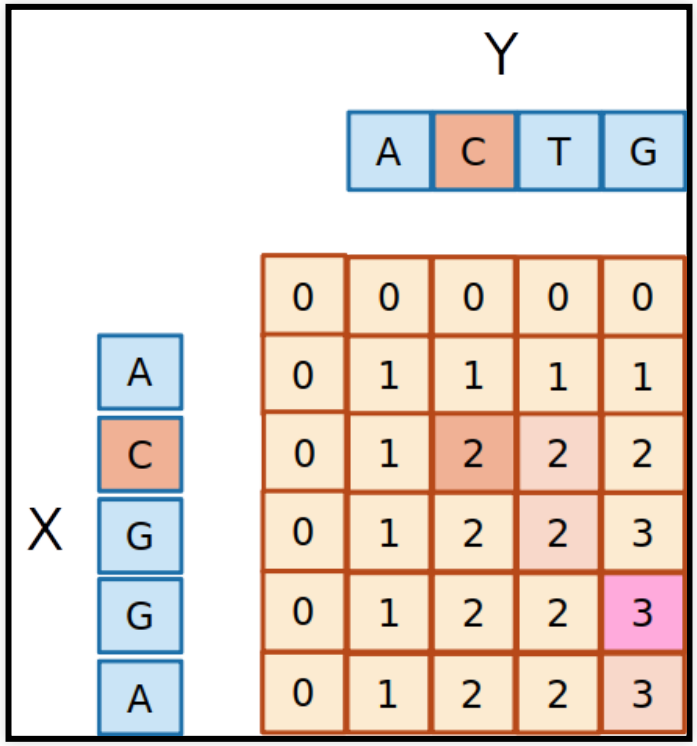
Programação Dinâmica

- Digamos que escolhemos a de cima
- Como os dois caracteres são diferentes, e o valor da célula de cima é menor que o da esquerda, o 2 deve ter vindo da esquerda



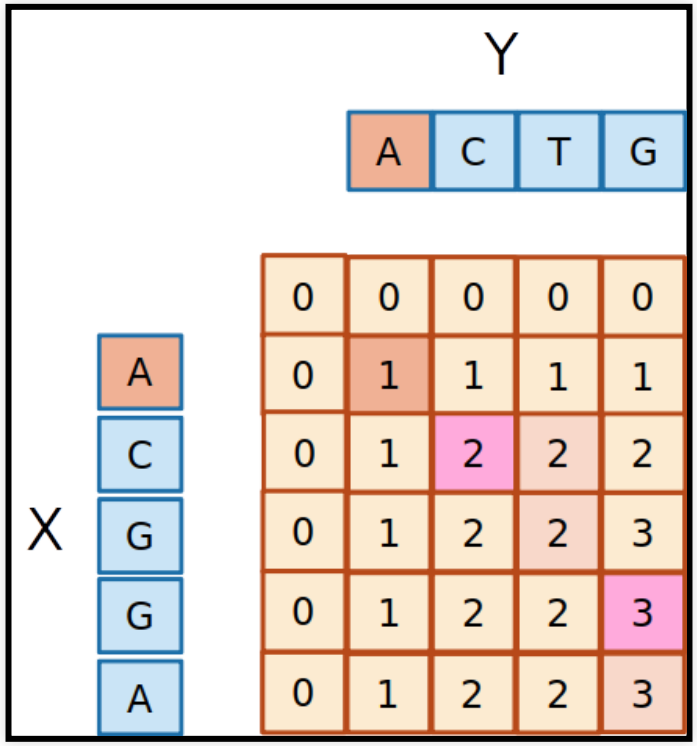
Programação Dinâmica

- Os dois caracteres são iguais, então o 2 deve ter vindo desta célula



Programação Dinâmica

- Os dois caracteres são iguais, então o 1 deve ter vindo desta célula



Programação Dinâmica

Algoritmo

Algoritmo 1: LCS(X,m,Y,n)

```
para i = 1 até m faça
    C[i,0] = 0;
fim
para j = 1 até n faça
    C[0,j] = 0;
fim
para i = 1 até m faça
    para j = 1 até n faça
        se X[i] == Y[j] então
            C[i, j] = C[i − 1, j − 1] + 1
        senão
            C[i, j] = max(C[i − 1, j], C[i, j − 1])
    fim
fim
```

Algoritmo

Algoritmo 2: ImprimeLCS(C,X,Y,i,j)

```
se i == 0 ou i == 0 então
    retorna
se X[i] == Y[j] então
    ImprimeLCS(C,X,Y,i-1,j-1)
    imprimir(X[i])
senão se C[i − 1, j] ≤ C[i, j − 1] então
    ImprimeLCS(C,X,Y,i-1,j)
senão
    ImprimeLCS(C,X,Y,i,j-1)
```

Complexidade

- É fácil ver que a complexidade do algoritmo é $O(mn)$, com um custo adicional de memória de $O(mn)$

Algoritmo Needleman-Wunsch

- Quando dois caracteres são iguais, temos um `match`
- Quando há um buraco, temos um `gap`
- Em algumas situações, queremos dar pesos diferentes ao `match` e ao `gap`
  - Esse é um requisito comum em análise de DNA, em que queremos evitar buracos muito grandes no aninhamento
- Eventualmente, podemos também permitir `match` por similaridade
  - Por exemplo, `s` e `z` tem um som parecido

Algoritmo Needleman-Wunsch

- Seja $\alpha(a, b)$ uma função que retorna o benefício/penalidade de alinhar os caracteres a e b (que podem ou não serem iguais)
- Seja $\alpha(gap)$ a penalidade de alinhar um caracter com um gap
- O Algoritmo de Needleman-Wunsch é similar ao algoritmo de encontrar uma subsequência máxiama, mas com o objetivo é encontrar o alinhamento de pontuação máxima

Algoritmo Needleman-Wunsch

- Para o algoritmo de Needleman-Wunsch, definimos uma matriz de pontuação em que os pesos são definidos por

$$P_{i,j} = \max \begin{cases} \alpha\left(x_i, y_j\right) + P_{i-1,j-1} \\ \alpha(gap) + P_{i-1,j} \\ \alpha(gap) + P_{i,j-1} \end{cases}$$

Algoritmo Needleman-Wunsch

Algoritmo 57: ALINHAMENTO-BOTTOMUP(X, m, Y, n, α)

```
1 Seja M[0..m][0..n] uma matriz
2 para i = 0 até m faça
3     M[i][0] = i × α(gap)
4 para j = 0 até n faça
5     M[0][j] = j × α(gap)
6 para i = 1 até m faça
7     para j = 1 até n faça
8         M[i][j] =
            max{M[i−1][j−1]+α(x_i, y_j), M[i−1][j]+α(gap), M[i][j−1]+α(gap)}
9 retorna M[m][n]
```

Algoritmo Needleman-Wunsch

- É fácil ver que a complexidade é a mesma do algoritmo que encontra a maior subsquência máxima