PUCRS – Projeto e Otimização de Algoritmos

Trabalho 1 – Programação Dinâmica – 2025/1

Yanny Elise, Mykelly Barros, Samara Tavares

---

1. Descrição do Problema

O problema da Maior Subsequência Comum (LCS - Longest Common Subsequence) trata de identificar, entre duas sequências de símbolos, a maior sequência que aparece na mesma ordem em ambas, mesmo que não necessariamente de forma contínua. Este conceito é fundamental para medir o grau de similaridade entre duas sequências e possui uma vasta gama de aplicações práticas. Entre as aplicações mais relevantes, destacam-se a comparação de sequenciamentos genéticos; a análise de similaridade entre textos, a comparação de códigos-fonte de programas de computador, a identificação de padrões em imagens e texturas e a correção ou sinalização de erros de digitação em sistemas de busca.

Além de suas aplicações diretas, o problema da LCS também está relacionado a outras questões fundamentais da Ciência da Computação, como a distância de edição, o problema da maior subsequência crescente (LIS) etc. Em todos esses casos, a manipulação e análise de sequências são pontos centrais para a solução dos problemas.

O LCS é um problema de natureza combinatória com complexidade de tempo quadrática, o que significa que o tempo necessário para sua resolução cresce proporcionalmente ao produto dos tamanhos das duas sequências. Como consequência, à medida que o tamanho das strings aumenta, a solução prática do problema se torna progressivamente mais desafiadora.

2. Exemplos de Casos

Para ilustrar o conceito de LCS, analisamos os seguintes exemplos:

- **Exemplo 1:**
    - **S1 = "AGGTAB"**, **S2 = "GXTXAYB"**
    - LCS = "GTAB" (comprimento 4)
- **Exemplo 2:**
    - **S1 = "ABC"**, **S2 = "ABC"**
    - LCS = "ABC"
- **Exemplo 3:**
    - **S1 = "ABCBDAB", S2 = "BDCAB"**

- LCS = "BCAB"

Esses exemplos demonstram que a subsequência comum pode envolver saltos de caracteres e que podem existir múltiplas soluções ótimas de mesmo comprimento.

3. Algoritmo Recursivo (Força Bruta)

Uma primeira abordagem para resolver o problema da LCS é utilizar uma solução recursiva baseada em força bruta. Essa estratégia consiste em explorar todas as combinações possíveis de subsequências, comparando caractere por caractere de ambas as strings. Se os últimos caracteres forem iguais, o problema é reduzido aos prefixos anteriores, incrementando o tamanho da subsequência. Se forem diferentes, duas possibilidades são consideradas: ignorar o último caractere da primeira string ou da segunda, e escolher a opção que resultar em uma subsequência maior.

Embora conceitualmente simples, essa abordagem recursiva apresenta complexidade exponencial, tornando-se rapidamente inviável à medida que o tamanho das sequências cresce. Cada decisão gera duas novas chamadas recursivas, levando a um número total de combinações que cresce de maneira insustentável para strings maiores.

4. Algoritmo com Programação Dinâmica

Para contornar a ineficiência da abordagem recursiva, emprega-se a técnica de Programação Dinâmica. Esta técnica baseia-se em resolver e armazenar as soluções de subproblemas menores, evitando recomputações desnecessárias. Ao utilizar Programação Dinâmica para a LCS, constrói-se uma matriz `dp` onde cada célula `dp[i][j]` armazena o comprimento da maior sequência comum entre os prefixos das duas strings até as posições `i` e `j`, respectivamente.

Inicialmente, toda a primeira linha e a primeira coluna da matriz são preenchidas com zeros, uma vez que a subsequência comum entre qualquer string e uma string vazia é vazia, de comprimento zero. O preenchimento da matriz segue uma lógica sequencial da esquerda para a direita e de cima para baixo. Se os caracteres **`S1[i-1]` e `S2[j-1]`** forem iguais, então **`dp[i][j]`** recebe o valor de **`dp[i-1][j-1] + 1`**, indicando que a subsequência foi estendida. Caso contrário, **`dp[i][j]`** é igual ao maior valor entre **`dp[i-1][j]`** e **`dp[i][j-1]`**, representando o maior LCS obtido até aquele ponto, sem considerar o caractere atual.

Formalmente, a fórmula de transição é descrita como:

Sejam duas strings S1 e S2, e dp[i][j] a matriz que armazena o comprimento da maior subsequência comum entre os prefixos S1[0..i-1] e S2[0..j-1].

A fórmula de preenchimento é:

Se **`S1[i-1] == S2[j-1]:`**

`dp[i][j] = dp[i-1][j-1] + 1`

Caso contrário:

`dp[i][j] = max(dp[i-1][j], dp[i][j-1])`

Ou seja, se os caracteres atuais forem iguais, adiciona-se 1 ao comprimento da subsequência anterior (diagonal superior à esquerda). Caso contrário, considera-se o maior comprimento entre ignorar o último caractere de S1 ou ignorar o último caractere de S2.

O uso da Programação Dinâmica reduz a complexidade do problema para O(m × n), onde m e n são os tamanhos das strings comparadas. Em implementações em linguagem C, essa abordagem se torna ainda mais natural, pois as strings em C são indexadas a partir de zero e finalizadas com o caractere especial NULL, o que facilita o tratamento de sufixos e prefixos durante o preenchimento da matriz.

Após a construção da matriz de programação dinâmica, obtém-se a seguinte tabela de comparação:

```
    C A G A X T C A C G A C G T A
  ---------------------------------
T | 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1
C | 0 1 1 1 1 1 1 2 2 2 2 2 2 2 2
G | 0 1 1 2 2 2 2 2 2 2 3 3 3 3 3
T | 0 1 1 2 2 2 3 3 3 3 3 3 4 4 4
G | 0 1 1 2 2 2 3 3 3 3 4 4 4 4 4
T | 0 1 1 2 2 2 3 3 3 3 4 4 4 4 5
T | 0 1 1 2 2 2 3 3 3 3 4 4 4 4 5
G | 0 1 1 2 2 2 3 3 3 3 4 4 4 5 5
C | 0 1 2 2 2 3 3 4 4 4 4 4 5 5 5
G | 0 1 2 2 2 3 3 4 4 4 5 5 5 5 5
T | 0 1 2 2 2 4 4 4 4 4 5 5 5 5 6
G | 0 1 2 2 2 4 4 4 4 4 5 5 5 5 6
Y | 0 1 2 2 2 4 4 4 4 4 5 5 5 5 6
A | 0 1 2 3 3 4 4 4 5 5 5 5 5 6 7
C | 0 1 2 3 3 4 4 4 5 5 5 6 6 6 7
T | 0 1 2 3 3 4 5 5 5 5 5 6 6 7 7
```

Observa-se que o comprimento da maior sequência comum é 8, e as subsequências correspondentes são "**CAGAXTCACGACGTA**" e "**TCGTGTTGCGTGYACT**".

5. Descrição da implementação:

- **Algoritmo Recursivo (Força Bruta)**

O **algoritmo recursivo** para o cálculo da **Maior Subsequência Comum (LCS)** entre duas strings é uma abordagem clássica, onde o problema é dividido em subproblemas menores. A recursão trabalha comparando os caracteres das duas strings de forma sequencial. Se os caracteres são iguais, o valor da subsequência aumenta em 1, e a função recursiva é chamada com os índices das strings reduzidos em 1. Caso contrário, o algoritmo verifica duas possibilidades: uma com o índice da primeira string reduzido e outra com o índice da segunda string reduzido, retornando o máximo entre esses dois resultados.
O tempo de execução desse algoritmo cresce de forma exponencial, pois para cada par de caracteres a função é chamada recursivamente. Isso resulta em uma complexidade de $O(2^n)$, onde n é o tamanho da string.

- **Algoritmo com Programação Dinâmica**

A abordagem de **Programação Dinâmica (DP)** resolve o problema de **LCS** de forma eficiente, armazenando os resultados intermediários em uma tabela para evitar a recomputação dos mesmos

subproblemas. A tabela é preenchida de forma iterativa, onde cada célula L[i][j] contém o comprimento da LCS das substrings X[0...i] e Y[0...j]. Se os caracteres nas posições i-1 e j-1 de ambas as strings forem iguais, o valor na célula é calculado como L[i-1][j-1] + 1. Caso contrário, o valor é o máximo entre L[i-1][j] e L[i][j-1].
Essa abordagem tem uma complexidade de tempo de $0(m * n)$, onde m e n são os tamanhos das duas strings, tornando-o muito mais eficiente do que a versão recursiva.

6. Exemplos de execução:

String 1: **ABCDE**

String 2: **ACE**

```
primeira string: ABCDE
segunda string: ACE

LCS recursivo: 3 | tempo: 0.000000s | interacoes: 15

LCS dinamico: 3 | tempo: 0.000000s | interacoes: 24
```

String 1: **ABCDEFGHIJ**

String 2: **ABDFHIJ**

```
primeira string: ABCDEFGHIJ
segunda string: ABDFHIJ

LCS recursivo: 7 | tempo: 0.000000s | interacoes: 192

LCS dinamico: 7 | tempo: 0.000000s | interacoes: 88
```

String 1: **ABCDEFGHIJKLMNO**

String 2: **ABDFHIJKLMNO**

```
primeira string: ABCDEFGHIJKLMNO
segunda string: ABDFHIJKLMNO

LCS recursivo: 12 | tempo: 0.000000s | interacoes: 197

LCS dinamico: 12 | tempo: 0.000000s | interacoes: 208
```

String 1: **ABCDEFGHIJKLMNOPQRST**

String 2: **ABDFHIJKLMMNOQRST**

Conclusão

A principal diferença entre os dois algoritmos está na eficiência. O algoritmo recursivo, apesar de simples, sofre com a duplicação de cálculos, levando a uma complexidade exponencial O(2^n), o que torna sua execução inviável para strings grandes. Em contraste, o algoritmo de programação dinâmica resolve o problema de forma mais eficiente, armazenando resultados intermediários em uma tabela e evitando recomputação, o que reduz a complexidade para O(m * n), tornando-o significativamente mais rápido e adequado para sequências maiores. A tabela de programação dinâmica melhora a eficiência ao evitar a recalculação de subproblemas, permitindo que o algoritmo reutilize resultados já calculados, o que resulta em um tempo de execução linear em relação ao tamanho das strings. Assim, a abordagem dinâmica é a mais indicada para resolver o problema de LCS, especialmente para entradas de tamanho considerável.