Estudo, Análise e Implementação do algoritmo

LCS (Longest Common Subsequence)

Alunos: Henrique Schultz e Augusto Baldino

Descrição do problema:

O problema da Maior Subsequência Comum (**LCS - *Longest Common Subsequence***) consiste em encontrar a maior subsequência (não necessariamente contínua) comum entre duas sequências.

Por exemplo, entre as strings "**ABC**" e "**AC**", a maior subsequência comum é "**AC**", de comprimento **2**.

Exemplos de casos do problema: (esses são os exemplos utilizados posteriormente para demonstrar a execução)

String 1	String 2	LCS esperada
ABC	AC	2 (“AC”)
AGGTAB	GXTXAYB	4 ("GTAB")
ABCDEFGHI	JKLMNOPQR	0
ABCDEFGHIJKLMNOPQRSTUVWXYZ	ZZZABCZZZ	4 ("ABCZ")

Algoritmo Recursivo (Força Bruta)

O algoritmo recursivo busca todas as possibilidades de subsequências comuns, testando caractere a caractere:

```java
public class LCSRecursivo{

    static int chamadas = 0;

    public static int lcsRecursivo(String s1, String s2,int m, int n){
        chamadas++;

        // Caso base: se alguma string acabar
        if(m==0 || n==0){
            return 0;
        }

        // Se o último caractere for igual
        if(s1.charAt(m-1) == s2.charAt(n-1)){
            return 1 + lcsRecursivo(s1,s2,m-1,n-1);
        } else{
            // Se forem diferentes, tenta as duas possibilidades
            return Math.max(lcsRecursivo(s1, s2, m-1, n), lcsRecursivo(s1, s2, m, n-1));
        }
    }
}
```

- O método `lcsRecursivo` é estático para permitir o controle global do número de chamadas recursivas.
- **Caso base**: Se qualquer uma das strings estiver vazia (comprimento 0), retorna 0, pois não há subsequência comum possível.
- **Se** o último caractere das duas strings **for igual**, somamos 1 ao comprimento e chamamos a função recursivamente com os comprimentos reduzidos (m-1 e n-1).
- **Se** os caracteres **forem diferentes**, chamamos recursivamente duas opções (remover um caractere de cada string) e escolhemos o maior resultado usando Math.max.
- Assim, o algoritmo explora **todas as possibilidades possíveis** e retorna o maior comprimento encontrado.

Exemplo:
Para s1 = "ABC" e s2 = "AC", o algoritmo segue essas chamadas:

```
lcsRecursivo("ABC", "AC") -> "C" == "C" -> 1 + lcsRecursivo("AB", "A")
lcsRecursivo("AB", "A") -> "B" != "A" -> Math.max(lcsRecursivo("A", "A"), lcsRecursivo("AB", ""))
lcsRecursivo("A", "A") -> "A" == "A" -> 1 + lcsRecursivo("", "")
lcsRecursivo("AB", "") -> caso base
lcsRecursivo("", "") -> caso base

Resultado: 1 + 1 = 2 (AC)
```

Complexidade: O(2^n) no pior caso, devido à exploração de todas as possibilidades.

Algoritmo com Programação Dinâmica

O algoritmo com programação dinâmica resolve o LCS usando uma matriz dp[m+1][n+1] que armazena os tamanhos das subsequências comuns entre partes das strings, evitando cálculos repetidos. É mais eficaz com strings maiores.

```java
public class LCSDinamico {

    static int iteracoes = 0; // contador de iterações como campo static

    public static int lcsDinamico(String s1, String s2) {
        int m = s1.length();
        int n = s2.length();

        int[][] dp = new int[m + 1][n + 1];
        iteracoes = 0; // zerar antes de calcular

        // Preenchendo a matriz DP
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                iteracoes++;
                if (s1.charAt(i - 1) == s2.charAt(j - 1)) {
                    dp[i][j] = 1 + dp[i - 1][j - 1];
                } else {
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
                }
            }
        }

        return dp[m][n];
    }
}
```

- Primeiro, calculamos o **tamanho** das duas strings (m e n) e criamos uma **matriz dp[m+1][n+1]** para armazenar os resultados parciais.
- Utilizando **dois laços for**, percorremos toda a matriz.
- Para cada posição (i, j):
- Se os caracteres i e j das strings forem **iguais**, armazenamos **1 + dp[i-1][j-1]**.
- Se forem **diferentes**, armazenamos o maior valor entre **dp[i-1][j]** e **dp[i][j-1]**.
- Ao final, o valor de **dp[m][n]** será o comprimento da maior subsequência comum.
- Diferente do algoritmo recursivo, aqui não fazemos chamadas recursivas. Tudo é resolvido iterativamente, preenchendo a tabela **dp** de forma crescente.

Exemplo:

Tabela inicial:

	""	A	C
""	0	0	0
A	0		
B	0		
C	0		

Para s1 = "ABC", s2 = "AC":

```
i=1, j=1 → "A" == "A" → dp[1][1] = 1 + dp[0][0] = 1
i=1, j=2 → "A" != "C" → dp[1][2] = Math.max(dp[0][2], dp[1][1]) = Math.max(0,1) = 1
i=2, j=1 → "B" != "A" → dp[2][1] = Math.max(dp[1][1], dp[2][0]) = Math.max(1,0) = 1
i=2, j=2 → "B" != "C" → dp[2][2] = Math.max(dp[1][2], dp[2][1]) = Math.max(1,1) = 1
i=3, j=1 → "C" != "A" → dp[3][1] = Math.max(dp[2][1], dp[3][0]) = Math.max(1,0) = 1
i=3, j=2 → "C" == "C" → dp[3][2] = 1 + dp[2][1] = 1 + 1 = 2
```

Tabela:

	""	A	C
""	0	0	0
A	0	1	1
B	0	1	1
C	0	1	2

Resultado: dp[3][2] = 2

Complexidade: O(m*n), muito mais eficiente em strings longas.

Descrição da Implementação:
O projeto é composto por três arquivos Java:
LCSRecursivo.java: implementa a solução **recursiva** e conta as chamadas
LCSDinamico.java: implementa a solução com PD e conta as iterações
Main.java: interface de execução com entrada via teclado e medição de tempo de execução

---

Exemplos de execução:

Teste 1: Entrada: "ABC" x "AC"

```
=== Executando LCS Recursivo ===
Comprimento da LCS (Recursivo): 2
Número de chamadas recursivas: 5
Tempo de execução: 0.000032 segundos

=== Executando LCS Dinâmico ===
Comprimento da LCS (Dinâmico): 2
Número de iterações: 6
Tempo de execução: 0.000443 segundos
```

Teste 2: Entrada: " AGGTAB" x " GXTXAYB"

```
=== Executando LCS Recursivo ===
Comprimento da LCS (Recursivo): 4
Número de chamadas recursivas: 346
Tempo de execução: 0.000063 segundos

=== Executando LCS Dinâmico ===
Comprimento da LCS (Dinâmico): 4
Número de iterações: 42
Tempo de execução: 0.000569 segundos
```

Teste 3: Entrada: " ABCDEFGHI" x " JKLMNOPQR"

```
=== Executando LCS Recursivo ===
Comprimento da LCS (Recursivo): 0
Número de chamadas recursivas: 97239
Tempo de execução: 0.001668 segundos

=== Executando LCS Dinâmico ===
Comprimento da LCS (Dinâmico): 0
Número de iterações: 81
Tempo de execução: 0.000469 segundos
```

Teste 4: Entrada: " ABCDEFGHIJKLMNOPQRSTUVWXYZ" x " ZZZABCZZZ"

```
=== Executando LCS Recursivo ===
Comprimento da LCS (Recursivo): 4
Número de chamadas recursivas: 27542439
Tempo de execução: 0.056643 segundos

=== Executando LCS Dinâmico ===
Comprimento da LCS (Dinâmico): 4
Número de iterações: 234
Tempo de execução: 0.000542 segundos
```

Conclusão:

O trabalho permitiu visualizar e compreender de forma prática as vantagens da programação dinâmica.

A comparação da abordagem recursiva com a dinâmica evidenciou a importância do armazenamento de resultados parciais para ganho de desempenho.

Constatamos que o algoritmo recursivo só é vantajoso para entradas pequenas, pois a abordagem dinâmica leva um pouco mais de tempo inicialmente para criar a matriz, enquanto o recursivo já inicia comparando as strings.

Porém a solução com programação dinâmica vai ficando significativamente mais eficiente a partir de uma certa quantidade de caracteres, isso é observável no **Teste 3** e principalmente no **Teste 4**.