Descobrindo padrões — a jornada para sincronizar dados complexos

Grupo: Gabriel Arpini, Marcos Vinicius, Riquelme e Victor Rowilson

Introdução

Encontrar uma solução para o problema de sincronização dos dados;

Implementar a solução em uma interface web (HTML,CSS, Javascript);

Definição do problema

Encontrar as maiores subsequências comuns para um par de entradas (Helena e Marcus) de modo que satisfaça o processamento de D conjuntos de pares. Onde:

1 <= D <= 10

Exemplo:

Entrada:	Saída:
1	ijiji ijiki ijkji
ijkijkii	ikiji ikiki ikjii ikjk
ikjikji	

A saída deve estar em ordem alfabética garantindo que haverá pelo menos uma subsequência válida, e o número total de subsequências distintas não ultrapassará 1000.

A interface

Foi utilizado uma interface web que utiliza HTML, CSS e Javascript;

Os elementos textuais da interface são processados pelo script em Javascript;

Os elementos gráficos para inserção dos textos de Helena e Marcus são gerados dinamicamente pelo script em Javascript;

Programação Dinâmica

```javascript
// Cria uma matriz dp para armazenar o comprimento das subsequências comuns entre prefixos de a e b
function construirMatrizDP(a, b) {
  const m = a.length, n = b.length;
  const dp = Array.from({ length: m + 1 }, () => Array(n + 1).fill(0));
  for (let i = 1; i <= m; i++) {
    for (let j = 1; j <= n; j++) {
      if (a[i - 1] === b[j - 1]) {
        // Se os caracteres forem iguais, soma 1 ao valor da diagonal anterior
        dp[i][j] = dp[i - 1][j - 1] + 1;
      } else {
        // Caso contrário, pega o maior valor entre cima ou esquerda
        dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
      }
    }
  }
  return dp;
}
```

Visualização da tabela

		i	k	j	i	k	j	i
	0	0	0	0	0	0	0	0
i	0	1	1	1	1	1	1	1
j	0	1	2	2	2	2	2	2
k	0	1	2	2	2	3	3	3
i	0	1	2	2	3	3	3	4
j	0	1	2	3	3	3	4	4
k	0	1	2	3	3	4	4	4
i	0	1	2	3	4	4	4	5
i	0	1	2	3	4	4	4	5

Backtrack

```
// Função recursiva que reconstrói todas as subsequências usando backtracking
function backtrack(dp, a, b, i, j, memo) {
  // Caso base: chegou ao início de alguma das strings
  if (i === 0 || j === 0) return new Set([""]);
  const key = `${i},${j}`;
  if (memo.has(key)) return memo.get(key);

  const result = new Set();

  // Se os caracteres forem iguais, segue na diagonal e adiciona o caractere
  if (a[i - 1] === b[j - 1]) {
    for (const sub of backtrack(dp, a, b, i - 1, j - 1, memo)) {
      result.add(sub + a[i - 1]);
    }
  } else {
    // Se os caminhos de cima ou da esquerda forem válidos (mantêm o comprimento máximo), explora ambos
    if (dp[i - 1][j] >= dp[i][j - 1]) {
      for (const sub of backtrack(dp, a, b, i - 1, j, memo)) result.add(sub);
    }
    if (dp[i][j - 1] >= dp[i - 1][j]) {
      for (const sub of backtrack(dp, a, b, i, j - 1, memo)) result.add(sub);
    }
  }

  memo.set(key, result); // Memoriza o resultado
  return result;
}
```

Visualização de um exemplo de Backtrack

		i	k	j	i	k	j	i
	0	0	0	0	0	0	0	0
i	0	1*	1	1	1	1	1	1
j	0	1	2*	2	2	2	2	2
k	0	1	2	2*	2	3	3	3
i	0	1	2	2	3*	3	3	4
j	0	1	2	3	3	3*	4	4
k	0	1	2	3	3	4	4*	4
i	0	1	2	3	4	4	4	5*
i	0	1	2	3	4	4	4	5

Subsequência:

ijkji

Conclusão

A programação dinâmica em conjunto com o Backtrack trouxe uma eficiência muito maior para a obtenção do resultado esperado;

Permite generalizar para qualquer string de qualquer tamanho;