练习4 – 最长公共子序列（LCS）

- 最长公共子序列（Longest Common Subsequence，LCS）

- leetcode_1143_最长公共子序列：https://leetcode-cn.com/problems/longest-common-subsequence/

- 求两个序列的最长公共子序列长度
  - [1, 3, 5, 9, 10] 和 [1, 4, 9, 10] 的最长公共子序列是 [1, 9, 10]，长度为 3
  - ABCBDAB 和 BDCABA 的最长公共子序列长度是 4，可能是
    - ✓ ABCBDAB 和 BDCABA > BDAB
    - ✓ ABCBDAB 和 BDCABA > BDAB
    - ✓ ABCBDAB 和 BDCABA > BCAB
    - ✓ ABCBDAB 和 BDCABA > BCBA

最长公共子序列 – 思路

- 假设 2 个序列分别是 nums1、nums2
  - i ∈ [1, nums1.length]
  - j ∈ [1, nums2.length]

- 假设 dp(i, j) 是【nums1 前 i 个元素】与【nums2 前 j 个元素】的最长公共子序列长度
  - dp(i, 0)、dp(0, j) 初始值均为 0
  - 如果 nums1[i – 1] = nums2[j – 1]，那么 dp(i, j) = dp(i – 1, j – 1) + 1
  - 如果 nums1[i – 1] ≠ nums2[j – 1]，那么 dp(i, j) = max { dp(i – 1, j), dp(i, j – 1) }

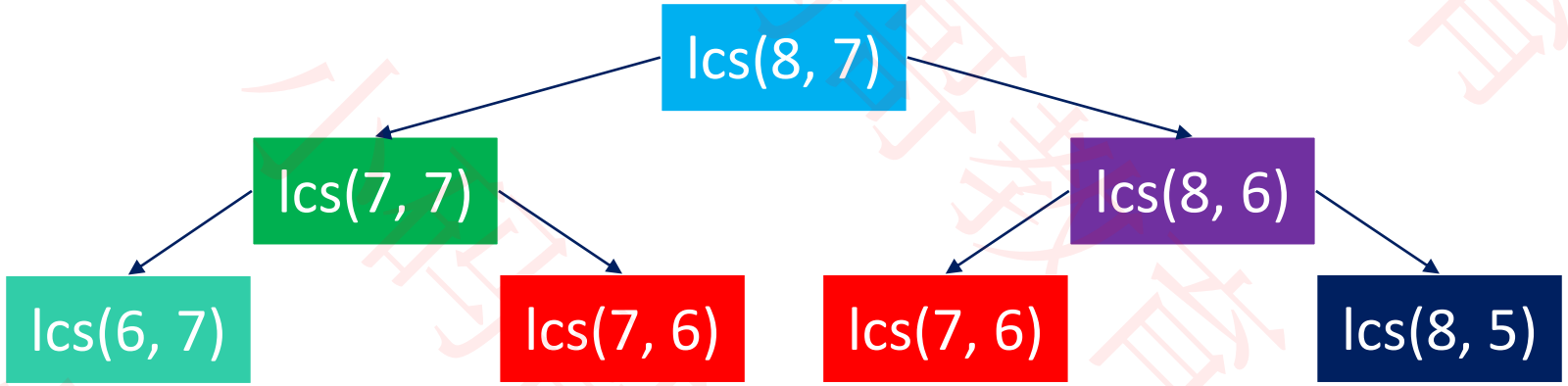
最长公共子序列 – 递归实现

```
int lcs(int[] nums1, int[] nums2) {
    if (nums1 == null || nums1.length == 0) return 0;
    if (nums2 == null || nums2.length == 0) return 0;
    return lcs(nums1, nums1.length, nums2, nums2.length);
}

int lcs(int[] nums1, int i,
        int[] nums2, int j) {
    if (i == 0 || j == 0) return 0;
    if (nums1[i - 1] != nums2[j - 1]) {
        return Math.max(
                lcs(nums1, i - 1, nums2, j),
                lcs(nums1, i, nums2, j - 1));
    }
    return lcs(nums1, i - 1, nums2, j - 1) + 1;
}
```

- 空间复杂度：$O(k)$，$k = \min\{n, m\}$，n、m 是 2 个序列的长度
- 时间复杂度：$O(2^n)$，当 $n = m$ 时

最长公共子序列 – 递归实现分析



- 出现了重复的递归调用

小码哥教育 SEEMYGO

最长公共子序列 – 非递归实现

```
int lcs(int[] nums1, int[] nums2) {
    if (nums1 == null || nums1.length == 0) return 0;
    if (nums2 == null || nums2.length == 0) return 0;
    int[][] dp = new int[nums1.length + 1][nums2.length + 1];
    for (int i = 1; i <= nums1.length; i++) {
        for (int j = 1; j <= nums2.length; j++) {
            if (nums1[i - 1] == nums2[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }
    return dp[nums1.length][nums2.length];
}
```

- 空间复杂度：$O(n * m)$
- 时间复杂度：$O(n * m)$

最长公共子序列 – 非递归实现

- dp 数组的计算结果如下所示

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

最长公共子序列 – 非递归实现 – 滚动数组

- 可以使用滚动数组优化空间复杂度

```
int lcs(int[] nums1, int[] nums2) {
    if (nums1 == null || nums1.length == 0) return 0;
    if (nums2 == null || nums2.length == 0) return 0;
    int[][] dp = new int[2][nums2.length + 1];
    for (int i = 1; i <= nums1.length; i++) {
        int row = i & 1;
        int prevRow = (i - 1) & 1;
        for (int j = 1; j <= nums2.length; j++) {
            if (nums1[i - 1] == nums2[j - 1]) {
                dp[row][j] = dp[prevRow][j - 1] + 1;
            } else {
                dp[row][j] = Math.max(dp[prevRow][j], dp[row][j - 1]);
            }
        }
    }
    return dp[nums1.length & 1][nums2.length];
}
```

最长公共子序列 – 非递归实现 – 一维数组

- 可以将 二维数组 优化成 一维数组，进一步降低空间复杂度

```
int lcs(int[] nums1, int[] nums2) {
    if (nums1 == null || nums1.length == 0) return 0;
    if (nums2 == null || nums2.length == 0) return 0;
    int[] dp = new int[nums2.length + 1];
    for (int i = 1; i <= nums1.length; i++) {
        int cur = 0;
        for (int j = 1; j <= nums2.length; j++) {
            int leftTop = cur;
            cur = dp[j];
            if (nums1[i - 1] == nums2[j - 1]) {
                dp[j] = leftTop + 1;
            } else {
                dp[j] = Math.max(dp[j], dp[j - 1]);
            }
        }
    }
    return dp[nums2.length];
}
```

最长公共子序列 – 非递归实现 – 一维数组

- 可以空间复杂度优化至 $O(k)$，$k = \min\{n, m\}$

```
int lcs(int[] nums1, int[] nums2) {
    if (nums1 == null || nums1.length == 0) return 0;
    if (nums2 == null || nums2.length == 0) return 0;
    int[] rowsNums = nums1, colsNums = nums2;
    if (nums1.length < nums2.length) {
        colsNums = nums1;
        rowsNums = nums2;
    }
    int[] dp = new int[colsNums.length + 1];
    for (int i = 1; i <= rowsNums.length; i++) {
        int cur = 0;
        for (int j = 1; j <= colsNums.length; j++) {
            int leftTop = cur;
            cur = dp[j];
            if (rowsNums[i - 1] == colsNums[j - 1]) {
                dp[j] = leftTop + 1;
            } else {
                dp[j] = Math.max(dp[j], dp[j - 1]);
            }
        }
    }
    return dp[colsNums.length];
}
```