动态规划（Dynamic Programming）

动态规划（Dynamic Programming）

- 动态规划，简称DP
- 是求解最优化问题的一种常用策略

- 通常的使用套路（一步一步优化）

① 暴力递归（自顶向下，出现了重叠子问题）

② 记忆化搜索（自顶向下）

③ 递推（自底向上）

动态规划的常规步骤

■ 动态规划中的“动态”可以理解为是“会变化的状态”

① 定义状态（状态是原问题、子问题的解）

✓ 比如定义 dp(i) 的含义

② 设置初始状态（边界）

✓ 比如设置 dp(0) 的值

③ 确定状态转移方程

✓ 比如确定 dp(i) 和 dp(i – 1) 的关系

动态规划的一些相关概念

■ 来自维基百科的解释

□ **Dynamic Programming** is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions.

① 将复杂的原问题拆解成若干个简单的子问题

② 每个子问题仅仅解决1次，并保存它们的解

③ 最后推导出原问题的解

■ 可以用动态规划来解决的问题，通常具备2个特点

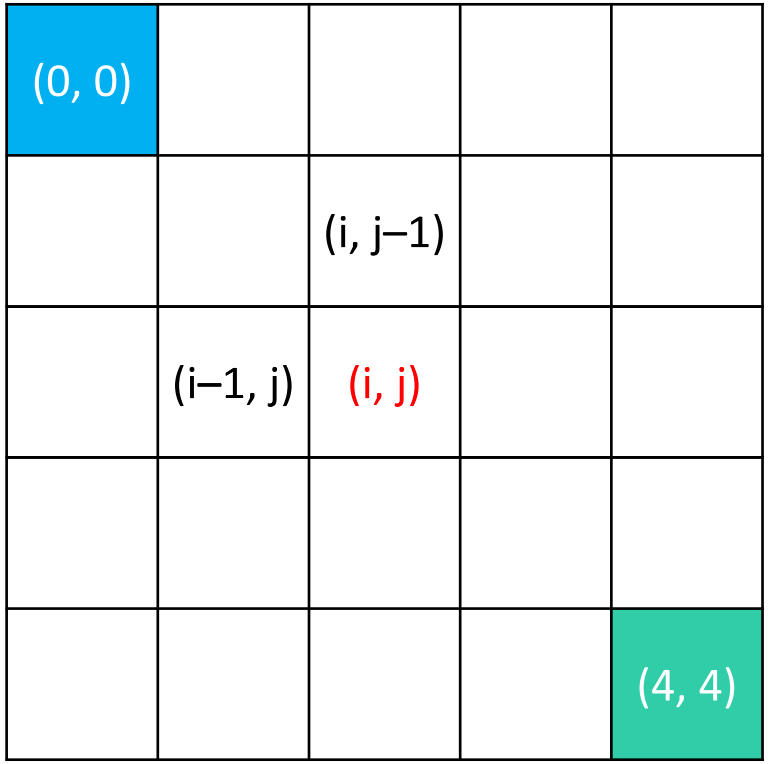
□ 最优子结构（最优化原理）：通过求解子问题的最优解，可以获得原问题的最优解

□ 无后效性

✓ 某阶段的状态一旦确定，则此后过程的演变不再受此前各状态及决策的影响（未来与过去无关）

✓ 在推导后面阶段的状态时，只关心前面阶段的具体状态值，不关心这个状态是怎么一步步推导出来的

无后效性

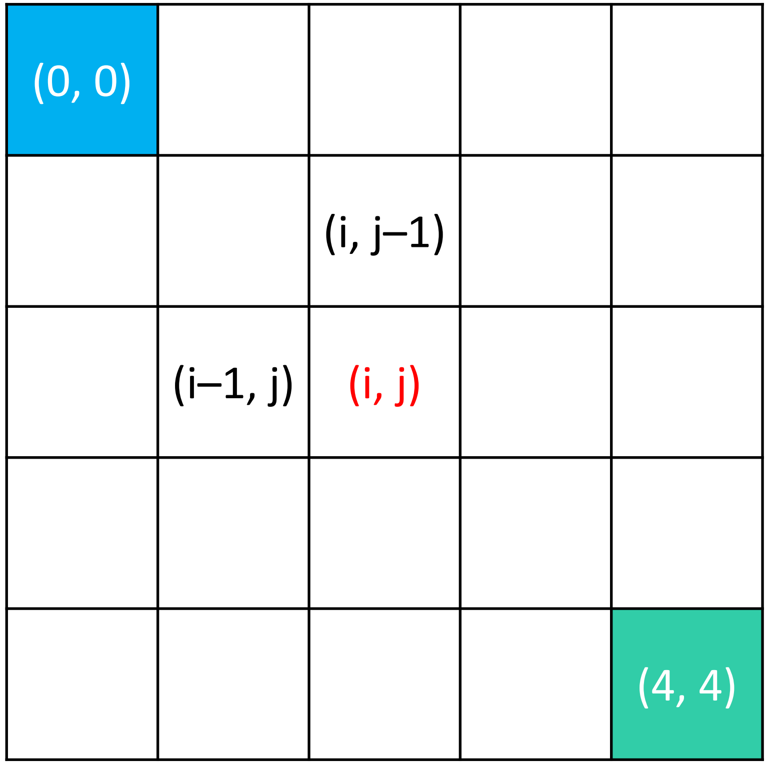


- 从起点（0, 0）走到终点（4, 4）一共有多少种走法？只能向右、向下走

- 假设 dp(i, j) 是从（0, 0）走到（i, j）的走法
  - $dp(i, 0) = dp(0, j) = 1$
  - $dp(i, j) = dp(i, j-1) + dp(i-1, j)$

- 无后效性
  - 推导 dp(i, j) 时只需要用到 dp(i, j – 1)、dp(i – 1, j) 的值
  - 不需要关心 dp(i, j – 1)、dp(i – 1, j) 的值是怎么求出来的

有后效性



- 如果可以向左、向右、向上、向下走，并且同一个格子不能走 2 次

- 有后效性

□ dp(i, j) 下一步要怎么走，还要关心上一步是怎么来的

✓ 也就是还要关心 dp(i, j – 1)、dp(i – 1, j) 是怎么来的?

练习1 – 找零钱

- leetcode_322_零钱兑换：https://leetcode-cn.com/problems/coin-change/

- 假设有25分、20分、5分、1分的硬币，现要找给客户41分的零钱，如何办到硬币个数最少?
  - 此前用贪心策略得到的并非是最优解（贪心得到的解是 5 枚硬币）

- 假设 dp(n) 是凑到 n 分需要的最少硬币个数
  - 如果第 1 次选择了 25 分的硬币，那么 dp(n) = dp(n – 25) + 1
  - 如果第 1 次选择了 20 分的硬币，那么 dp(n) = dp(n – 20) + 1
  - 如果第 1 次选择了 5 分的硬币，那么 dp(n) = dp(n – 5) + 1
  - 如果第 1 次选择了 1 分的硬币，那么 dp(n) = dp(n – 1) + 1
  - 所以 dp(n) = min { dp(n – 25), dp(n – 20), dp(n – 5), dp(n – 1) } + 1

找零钱 – 暴力递归

```java
int coins(int n) {
    if (n < 1) return Integer.MAX_VALUE;
    if (n == 1 || n == 5 || n == 20 || n == 25) return 1;
    int min1 = Math.min(coins(n - 1), coins(n - 5));
    int min2 = Math.min(coins(n - 20), coins(n - 25));
    return Math.min(min1, min2) + 1;
}
```

- 类似于斐波那契数列的递归版，会有大量的重复计算，时间复杂度较高

找零钱 – 记忆化搜索

```java
int coins(int n) {
    if (n < 1) return -1;
    int[] dp = new int[n + 1];
    int[] faces = {1, 5, 20, 25};
    for (int face : faces) {
        if (n < face) break;
        dp[face] = 1;
    }
    return coins(n, dp);
}

static int coins(int n, int[] dp) {
    if (n < 1) return Integer.MAX_VALUE;
    if (dp[n] == 0) {
        int min1 = Math.min(coins(n - 25, dp), coins(n - 20, dp));
        int min2 = Math.min(coins(n - 5, dp), coins(n - 1, dp));
        dp[n] = Math.min(min1, min2) + 1;
    }
    return dp[n];
}
```

找零钱 – 递推

```java
int coins(int n) {
    if (n < 1) return -1;
    int[] dp = new int[n + 1];
    for (int i = 1; i <= n; i++) {
        int min = dp[i - 1];
        if (i >= 5) min = Math.min(dp[i - 5], min);
        if (i >= 20) min = Math.min(dp[i - 20], min);
        if (i >= 25) min = Math.min(dp[i - 25], min);
        dp[i] = min + 1;
    }
    return dp[n];
}
```

- 时间复杂度、空间复杂度：O(n)

思考题：请输出找零钱的具体方案（具体是用了哪些面值的硬币）

```java
int coins(int n) {
    if (n < 1) return -1;
    int[] dp = new int[n + 1];
    int[] faces = new int[dp.length];
    for (int i = 1; i <= n; i++) {
        int min = dp[i - 1];
        faces[i] = 1;
        if (i >= 5 && dp[i - 5] < min) {
            min = dp[i - 5];
            faces[i] = 5;
        }
        if (i >= 20 && dp[i - 20] < min) {
            min = dp[i - 20];
            faces[i] = 20;
        }
        if (i >= 25 && dp[i - 25] < min) {
            min = dp[i - 25];
            faces[i] = 25;
        }
        dp[i] = min + 1;
    }
    print(faces, n);
    return dp[n];
}
```

```java
void print(int[] faces, int i) {
    while (i > 0) {
        System.out.print(faces[i] + " ");
        i -= faces[i];
    }
    System.out.println();
}
```

找零钱 – 通用实现

```java
int coins(int n, int[] faces) {
    if (n < 1 || faces == null || faces.length == 0) return -1;
    int[] dp = new int[n + 1];
    for (int i = 1; i <= n; i++) {
        int min = Integer.MAX_VALUE;
        for (int face : faces) {
            if (i < face) continue;
            if (dp[i - face] < 0 || dp[i - face] >= min) continue;
            min = dp[i - face];
        }
        if (min == Integer.MAX_VALUE) {
            dp[i] = -1;
        } else {
            dp[i] = min + 1;
        }
    }
    return dp[n];
}
```

练习2 – 最大连续子序列和

- 给定一个长度为 n 的整数序列，求它的最大连续子序列和
  - 比如 –2、1、–3、4、–1、2、1、–5、4 的最大连续子序列和是 4 + (–1) + 2 + 1 = 6

- 状态定义
  - 假设 dp(i) 是以 nums[i] 结尾的最大连续子序列和（nums是整个序列）
  - ✓ 以 nums[0] –2 结尾的最大连续子序列是 –2，所以 dp(0) = –2
  - ✓ 以 nums[1] 1 结尾的最大连续子序列是 1，所以 dp(1) = 1
  - ✓ 以 nums[2] –3 结尾的最大连续子序列是 1、–3，所以 dp(2) = dp(1) + (–3) = –2
  - ✓ 以 nums[3] 4 结尾的最大连续子序列是 4，所以 dp(3) = 4
  - ✓ 以 nums[4] –1 结尾的最大连续子序列是 4、–1，所以 dp(4) = dp(3) + (–1) = 3
  - ✓ 以 nums[5] 2 结尾的最大连续子序列是 4、–1、2，所以 dp(5) = dp(4) + 2 = 5
  - ✓ 以 nums[6] 1 结尾的最大连续子序列是 4、–1、2、1，所以 dp(6) = dp(5) + 1 = 6
  - ✓ 以 nums[7] –5 结尾的最大连续子序列是 4、–1、2、1、–5，所以 dp(7) = dp(6) + (–5) = 1
  - ✓ 以 nums[8] 4 结尾的最大连续子序列是 4、–1、2、1、–5、4，所以 dp(8) = dp(7) + 4 = 5

最大连续子序列和 – 状态转移方程和初始状态

■ 状态转移方程

□ 如果 dp(i – 1) ≤ 0，那么 dp(i) = nums[i]

□ 如果 dp(i – 1) > 0，那么 dp(i) = dp(i – 1) + nums[i]

■ 初始状态

□ dp(0) 的值是 nums[0]

■ 最终的解

□ 最大连续子序列和是所有 dp(i) 中的最大值 max { dp(i) }，i ∈ [0, nums.length)

最大连续子序列和 – 动态规划 – 实现

```java
int maxSubArray(int[] nums) {
    if (nums == null || nums.length == 0) return 0;
    int[] dp = new int[nums.length];
    int max = dp[0] = nums[0];
    for (int i = 1; i < dp.length; i++) {
        int prev = dp[i - 1];
        if (prev > 0) {
            dp[i] = prev + nums[i];
        } else {
            dp[i] = nums[i];
        }
        max = Math.max(max, dp[i]);
    }
    return max;
}
```

- 空间复杂度：O(n)，时间复杂度：O(n)

最大连续子序列和 – 动态规划 – 优化实现

```java
int maxSubArray(int[] nums) {
    if (nums == null || nums.length == 0) return 0;
    int dp = nums[0];
    int max = dp;
    for (int i = 1; i < nums.length; i++) {
        if (dp > 0) {
            dp = dp + nums[i];
        } else {
            dp = nums[i];
        }
        max = Math.max(max, dp);
    }
    return max;
}
```

■ 空间复杂度：O(1)，时间复杂度：O(n)

练习3 – 最长上升子序列（LIS）

- 最长上升子序列（最长递增子序列，Longest Increasing Subsequence，LIS）

- leetcode_300_最长上升子序列：https://leetcode-cn.com/problems/longest-increasing-subsequence/

- 给定一个无序的整数序列，求出它最长上升子序列的长度（要求严格上升）
  - 比如 [10, 2, 2, 5, 1, 7, 101, 18] 的最长上升子序列是 [2, 5, 7, 101]、[2, 5, 7, 18]，长度是 4

最长上升子序列 – 动态规划 – 状态定义

- 假设数组是 nums， [10, 2, 2, 5, 1, 7, 101, 18]
  - dp(i) 是以 nums[i] 结尾的最长上升子序列的长度，i ∈ [0, nums.length)
  - ✓ 以 nums[0] 10 结尾的最长上升子序列是 10，所以 dp(0) = 1
  - ✓ 以 nums[1] 2 结尾的最长上升子序列是 2，所以 dp(1) = 1
  - ✓ 以 nums[2] 2 结尾的最长上升子序列是 2，所以 dp(2) = 1
  - ✓ 以 nums[3] 5 结尾的最长上升子序列是 2、5，所以 dp(3) = dp(1) + 1 = dp(2) + 1 = 2
  - ✓ 以 nums[4] 1 结尾的最长上升子序列是 1，所以 dp(4) = 1
  - ✓ 以 nums[5] 7 结尾的最长上升子序列是 2、5、7，所以 dp(5) = dp(3) + 1 = 3
  - ✓ 以 nums[6] 101 结尾的最长上升子序列是 2、5、7、101，所以 dp(6) = dp(5) + 1 = 4
  - ✓ 以 nums[7] 18 结尾的最长上升子序列是 2、5、7、18，所以 dp(7) = dp(5) + 1 = 4

- 最长上升子序列的长度是所有 dp(i) 中的最大值 max { dp(i) }，i ∈ [0, nums.length)

最长上升子序列 – 动态规划 – 状态转移方程

■ 遍历 j ∈ [0, i)

□ 当 nums[i] > nums[j]

✓ nums[i] 可以接在 nums[j] 后面，形成一个比 dp(j) 更长的上升子序列，长度为 dp(j) + 1

✓ dp(i) = max { dp(i), dp(j) + 1 }

□ 当 nums[i] ≤ nums[j]

✓ nums[i] 不能接在 nums[j] 后面，跳过此次遍历（continue）

■ 状态的初始值

□ dp(0) = 1

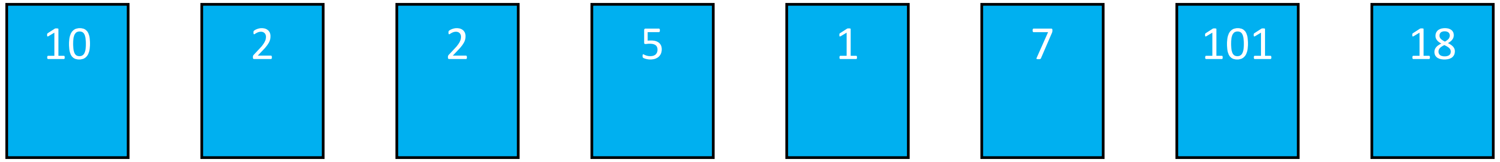
□ 所有的 dp(i) 默认都初始化为 1

最长上升子序列 – 动态规划 – 实现

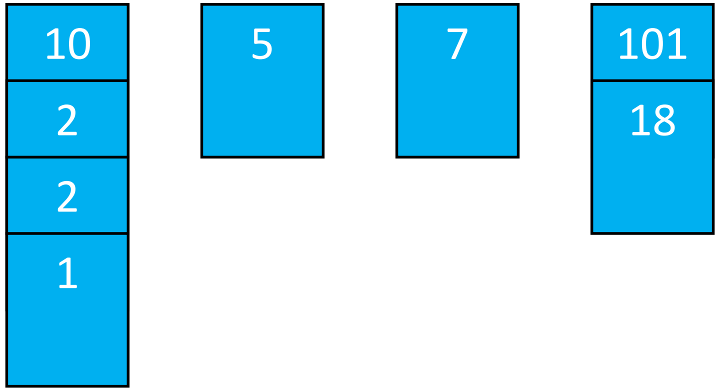
```java
int lengthOfLIS(int[] nums) {
    if (nums == null || nums.length == 0) return 0;
    int[] dp = new int[nums.length];
    int max = dp[0] = 1;
    for (int i = 1; i < dp.length; i++) {
        dp[i] = 1;
        for (int j = 0; j < i; j++) {
            if (nums[i] <= nums[j]) continue;
            dp[i] = Math.max(dp[i], dp[j] + 1);
        }
        max = Math.max(dp[i], max);
    }
    return max;
}
```

- 空间复杂度：$O(n)$，时间复杂度：$O(n^2)$

最长上升子序列 – 二分搜索 – 思路



- 把每个数字看做是一张扑克牌，从左到右按顺序处理每一个扑克牌
  - 将它压在（从左边数过来）第一个牌顶 ≥ 它的牌堆上面
  - 如果找不到牌顶 ≥ 它的牌堆，就在最右边新建一个牌堆，将它放入这个新牌堆中



- 当处理完所有牌，最终牌堆的数量就是最长上升子序列的长度

最长上升子序列 – 二分搜索 – 思路

■ 思路（假设数组是 nums，也就是最初的牌数组）

- top[i] 是第 i 个牌堆的牌顶，len 是牌堆的数量，初始值为 0
- 遍历每一张牌 num
  - ✓ 利用二分搜索找出 num 最终要放入的牌堆位置 index
  - ✓ num 作为第 index 个牌堆的牌顶，top[index] = num
  - ✓ 如果 index 等于 len，相当于新建一个牌堆，牌堆数量 +1，也就是 len++

最长上升子序列 – 二分搜索 – 实现

```java
int lengthOfLIS(int[] nums) {
    if (nums == null || nums.length == 0) return 0;
    int[] top = new int[nums.length];
    int len = 0;
    for (int num : nums) {
        int begin = 0, end = len;
        while (begin < end) {
            int mid = (begin + end) >> 1;
            if (num <= top[mid]) {
                end = mid;
            } else {
                begin = mid + 1;
            }
        }
        top[begin] = num;
        if (begin == len) len++;
    }
    return len;
}
```

- 空间复杂度：O(n)
- 时间复杂度：O(nlogn)

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]

前i – 1个元素	nums1[i – 1]
前j – 1个元素	nums2[j – 1]

- 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) }

前i – 1个元素
前j – 1个元素

前i – 1个元素	
前j – 1个元素	nums2[j – 1]

前i – 1个元素	nums1[i – 1]
前j – 1个元素	

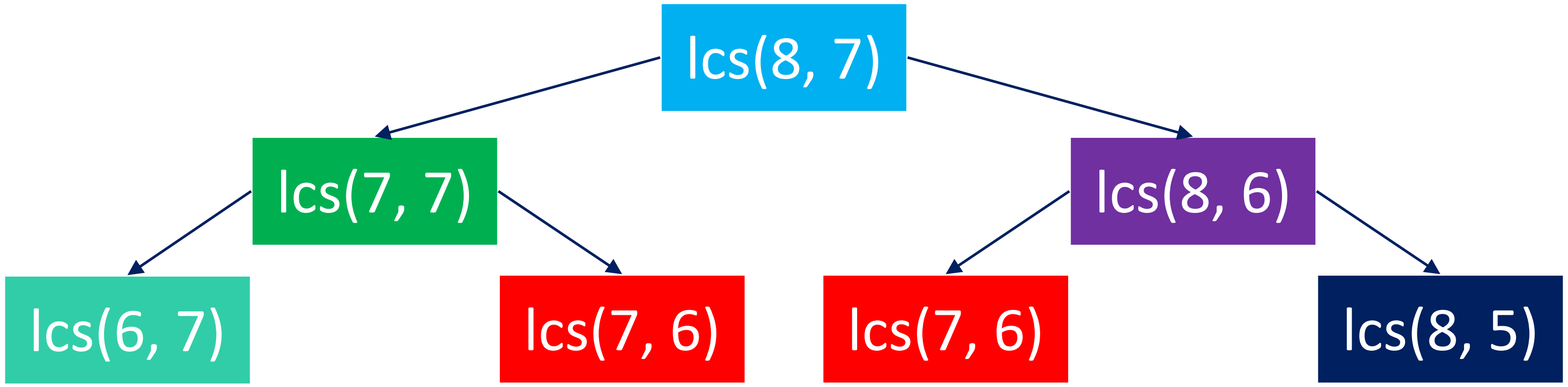
最长公共子序列 – 递归实现

```java
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

最长公共子序列 – 非递归实现

```java
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

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

```java
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

```java
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 $O(k)$，$k = \min\{n, m\}$

```java
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];
}
```

练习5 – 最长公共子串

- 最长公共子串（Longest Common Substring）
  - 子串是连续的子序列

- 求两个字符串的最长公共子串长度
  - ABCBA 和 BABCA 的最长公共子串是 ABC，长度为 3

最长公共子串 – 思路

- 假设 2 个字符串分别是 str1、str2
  - $i \in [1, str1.length]$
  - $j \in [1, str2.length]$

- 假设 dp(i, j) 是以 str1[i – 1]、str2[j – 1] 结尾的最长公共子串长度
  - dp(i, 0)、dp(0, j) 初始值均为 0
  - 如果 str1[i – 1] = str2[j – 1]，那么 dp(i, j) = dp(i – 1, j – 1) + 1
  - 如果 str1[i – 1] ≠ str2[j – 1]，那么 dp(i, j) = 0

- 最长公共子串的长度是所有 dp(i, j) 中的最大值 max { dp(i, j) }

最长公共子串 – 实现

```java
int lcs(String str1, String str2) {
    if (str1 == null || str2 == null) return 0;
    char[] cs1 = str1.toCharArray();
    if (cs1.length == 0) return 0;
    char[] cs2 = str2.toCharArray();
    if (cs2.length == 0) return 0;
    int[][] dp = new int[cs1.length + 1][cs2.length + 1];
    int max = 0;
    for (int i = 1; i <= cs1.length; i++) {
        for (int j = 1; j <= cs2.length; j++) {
            if (cs1[i - 1] != cs2[j - 1]) continue;
            dp[i][j] = dp[i - 1][j - 1] + 1;
            max = Math.max(max, dp[i][j]);
        }
    }
    return max;
}
```

- 空间复杂度：$O(n * m)$
- 时间复杂度：$O(n * m)$

最长公共子串 – 实现

- dp 数组的计算结果如下所示

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

最长公共子串 – 一维数组实现

```java
if (str1 == null || str2 == null) return 0;
char[] cs1 = str1.toCharArray();
if (cs1.length == 0) return 0;
char[] cs2 = str2.toCharArray();
if (cs2.length == 0) return 0;
char[] rowStr = cs1, colStr = cs2;
if (cs1.length < cs2.length) {
    colStr = cs1;
    rowStr = cs2;
}
int[] dp = new int[colStr.length + 1];
```

```java
int max = 0;
for (int i = 1; i <= rowStr.length; i++) {
    for (int j = colStr.length; j >= 1; j--) {
        if (rowStr[i - 1] == colStr[j - 1]) {
            dp[j] = dp[j - 1] + 1;
        } else {
            dp[j] = 0;
        }
        max = Math.max(max, dp[j]);
    }
}
return max;
```

- 空间复杂度：$O(k)$，$k = \min\{n, m\}$
- 时间复杂度：$O(n * m)$

练习6 – 0-1背包

- 有 n 件物品和一个最大承重为 W 的背包，每件物品的重量是 w_{i}、价值是 v_{i}
  - 在保证总重量不超过 W 的前提下，选择某些物品装入背包，背包的最大总价值是多少?
  - 注意：每个物品只有 1 件，也就是每个物品只能选择 0 件或者 1 件

- 假设 values 是价值数组，weights 是重量数组
  - 编号为 k 的物品，价值是 values[k]，重量是 weights[k]，k ∈ [0, n)

- 假设 dp(i, j) 是 **最大承重为 j、有前 i 件物品可选** 时的最大总价值，i ∈ [1, n]，j ∈ [1, W]
  - dp(i, 0)、dp(0, j) 初始值均为 0
  - 如果 j < weights[i – 1]，那么 dp(i, j) = dp(i – 1, j)
  - 如果 j ≥ weights[i – 1]，那么 dp(i, j) = max { dp(i – 1, j), dp(i – 1, j – weights[i – 1]) + values[i – 1] }

0-1背包 – 实现

```java
int select(int[] values, int[] weights, int capacity) {
    if (values == null || values.length == 0) return 0;
    if (weights == null || weights.length == 0) return 0;
    if (weights.length != values.length) return 0;
    if (capacity <= 0) return 0;
    int[][] dp = new int[values.length + 1][capacity + 1];
    for (int i = 1; i <= values.length; i++) {
        for (int j = 1; j <= capacity; j++) {
            if (j < weights[i - 1]) {
                dp[i][j] = dp[i - 1][j];
            } else {
                dp[i][j] = Math.max(dp[i - 1][j],
                        dp[i - 1][j - weights[i - 1]] + values[i - 1]);
            }
        }
    }
    return dp[values.length][capacity];
}
```

0-1背包 – 非递归实现

■ dp 数组的计算结果如下所示

	i \ j	0	1	2	3	4	5	6	7	8	9	10
	0	0	0	0	0	0	0	0	0	0	0	0
v=6, w=2	1	0	0	6	6	6	6	6	6	6	6	6
v=3, w=2	2	0	0	6	6	9	9	9	9	9	9	9
v=5, w=6	3	0	0	6	6	9	9	9	9	11	11	14
v=4, w=5	4	0	0	6	6	9	9	9	10	11	13	14
v=6, w=4	5	0	0	6	6	9	9	12	12	15	15	15

0-1背包 – 非递归实现 – 一维数组

- dp(i, j) 都是由 dp(i – 1, k) 推导出来的，也就是说，第 i 行的数据是由它的上一行第 i – 1 行推导出来的
- 因此，可以使用一维数组来优化
- 另外，由于 k ≤ j ，所以 j 的遍历应该由大到小，否则导致数据错乱

```java
int select(int[] values, int[] weights, int capacity) {
    if (values == null || values.length == 0) return 0;
    if (weights == null || weights.length == 0) return 0;
    if (weights.length != values.length) return 0;
    if (capacity <= 0) return 0;
    int[] dp = new int[capacity + 1];
    for (int i = 1; i <= values.length; i++) {
        for (int j = capacity; j >= 1; j--) {
            if (j < weights[i - 1]) continue;
            dp[j] = Math.max(dp[j],
                    dp[j - weights[i - 1]] + values[i - 1]);
        }
    }
    return dp[capacity];
}
```

0-1背包 – 非递归实现 – 一维数组优化

- 观察二维数组表，得出结论：j 的下界可以从 1 改为 weights[i – 1]

```java
int select(int[] values, int[] weights, int capacity) {
    if (values == null || values.length == 0) return 0;
    if (weights == null || weights.length == 0) return 0;
    if (weights.length != values.length) return 0;
    if (capacity <= 0) return 0;
    int[] dp = new int[capacity + 1];
    for (int i = 1; i <= values.length; i++) {
        for (int j = capacity; j >= weights[i - 1]; j--) {
            dp[j] = Math.max(dp[j],
                    dp[j - weights[i - 1]] + values[i - 1]);
        }
    }
    return dp[capacity];
}
```

0-1背包 – 恰好装满

- 有 n 件物品和一个最大承重为 W 的背包，每件物品的重量是 w_i、价值是 v_i
  - 在保证总重量恰好等于 W 的前提下，选择某些物品装入背包，背包的最大总价值是多少?
  - 注意：每个物品只有 1 件，也就是每个物品只能选择 0 件或者 1 件

- dp(i, j) 初始状态调整
  - dp(i, 0) = 0，总重量恰好为 0，最大总价值必然也为 0
  - dp(0, j) = –∞（负无穷），j ≥ 1，负数在这里代表无法恰好装满

	i \ j	0	1	2	3	4	5	6	7	8	9	10
	0	0	-	-	-	-	-	-	-	-	-	-
v=6, w=2	1	0	-	6	-	-	-	-	-	-	-	-
v=3, w=2	2	0	-	6	-	9	-	-	-	-	-	-
v=5, w=6	3	0	-	6	-	9	-	5	-	11	-	14
v=4, w=5	4	0	-	6	-	9	4	5	10	11	13	14
v=6, w=4	5	0	-	6	-	9	4	12	10	15	13	14

0-1背包 – 恰好装满 – 实现

```java
int selectExactly(int[] values, int[] weights, int capacity) {
    if (values == null || values.length == 0) return -1;
    if (weights == null || weights.length == 0) return -1;
    if (weights.length != values.length) return 0;
    if (capacity <= 0) return 0;
    int[] dp = new int[capacity + 1];
    for (int j = 1; j <= capacity; j++) {
        dp[j] = Integer.MIN_VALUE;
    }
    for (int i = 1; i <= values.length; i++) {
        for (int j = capacity; j >= weights[i - 1]; j--) {
            dp[j] = Math.max(dp[j],
                    dp[j - weights[i - 1]] + values[i - 1]);
        }
    }
    return dp[capacity] < 0 ? -1 : dp[capacity];
}
```