动态规划解题套路框架

核心问题：求解动态规划的核心问题是穷举。因为要求最值，肯定要把所有可行的答案穷举出来，然后在其中找最值呗。
重叠子问题、最优子结构、状态转移方程就是动态规划三要素。
动态规划的一般流程就是三步：暴力的递归解法->带备忘录的递归解法->迭代的动态规划解法。

一、斐波那契数列 509. 斐波那契数

斐波那契数，通常用 F(n) 表示，形成的序列称为斐波那契数列。
该数列由 0 和 1 开始，后面的每一项数字都是前面两项数字的和。也就是：
F(0) = 0，F(1) = 1F(N) = F(N - 1) + F(N - 2), 其中 N > 1.给定 N，计算 F(N)。
来自<https://leetcode-cn.com/problems/fibonacci-number/>

1、暴力递归【o(n^2)】：

```
int fib(int N) {
    if (N == 1 || N == 2) return 1;
    return fib(N - 1) + fib(N - 2);
}
```

2、带备忘录的递归解法【o(n)】

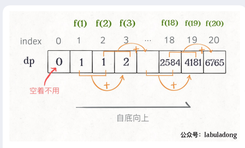
```
int fib(int N) {
    if (N < 1) return 0;
    // 备忘录全初始化为 0
    vector<int> memo(N + 1, 0);
    // 初始化最简情况
    return helper(memo, N);
}

int helper(vector<int>& memo, int n) {
    // base case
    if (n == 1 || n == 2) return 1;
    // 已经计算过
    if (memo[n] != 0) return memo[n];
    memo[n] = helper(memo, n - 1) +
              helper(memo, n - 2);
    return memo[n];
}
```

3、dp 数组的迭代解法【time：O(1)】

```
int fib(int N) {
    vector<int> dp(N + 1, 0);
    // base case
    dp[1] = dp[2] = 1;
    for (int i = 3; i <= N; i++)
        dp[i] = dp[i - 1] + dp[i - 2];
    return dp[N];
}
```

讲一个细节优化。细心的读者会发现，根据斐波那契数列的状态转移方程，
当前状态只和之前的两个状态有关，其实并不需要那么长的一个 DP-table来存储所有的状态，只要想办法存储之前的两个状态就行了。
所以，可以进一步优化，把空间复杂度降为O(1)：

```
int fib(int n) {
    if (n == 2 || n == 1)
        return 1;
    int prev = 1, curr = 1;
    for (int i = 3; i <= n; i++) {
        int sum = prev + curr;
        prev = curr;
        curr = sum;
    }
    return curr;
}
```



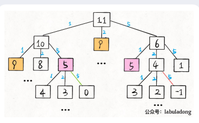
Subtopic

二、凑零钱问题 322. 零钱兑换

给定不同面额的硬币coins和一个总金额amount。
编写一个函数来计算可以凑成总金额所需的最少的硬币个数。
如果没有任何一种硬币组合能组成总金额，返回-1。
示例 1:
输入: coins = [1, 2, 5],amount=11
输出:3 解释: 11 = 5 + 5 + 1
链接：https://leetcode-cn.com/problems/coin-change

1、暴力递归【time：O(k*n^k)】

```
def coinChange(coins: List[int], amount: int):
    def dp(n):
        # base case
        if n == 0: return 0
        if n < 0: return -1
        # 求最小值，所以初始化为正无穷
        res = float('INF')
        for coin in coins:
            subproblem = dp(n - coin)
            # 子问题无解，跳过
            if subproblem == -1: continue
            res = min(res, 1 + subproblem)
        return res if res != float('INF') else -1
    return dp(amount)
```



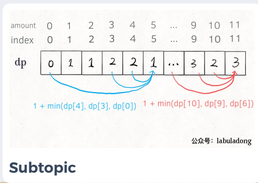
Subtopic

2、带备忘录的递归【time：O(kn)】

```
def coinChange(coins: List[int], amount: int):
    # 备忘录
    memo = dict()
    def dp(n):
        # 查备忘录，避免重复计算
        if n in memo: return memo[n]
        if n == 0: return 0
        if n < 0: return -1
        res = float('INF')
        for coin in coins:
            subproblem = dp(n - coin)
            if subproblem == -1: continue
            res = min(res, 1 + subproblem)
        # 记入备忘录
        memo[n] = res if res != float('INF') else -1
        return memo[n]
    return dp(amount)
```

3、dp 数组的迭代解法【time：O(n)】

```
int coinChange(vector<int>& coins, int amount) {
    // 数组大小为 amount + 1，初始值也为 amount + 1
    vector<int> dp(amount + 1, amount + 1);
    // base case
    dp[0] = 0;
    for (int i = 0; i < dp.size(); i++) {
        // 内层 for 在求所有子问题 + 1 的最小值
        for (int coin : coins) {
            // 子问题无解，跳过
            if (i - coin < 0) continue;
            dp[i] = min(dp[i], 1 + dp[i - coin]);
        }
    }
    return (dp[amount] == amount + 1) ? -1 : dp[amount];
}
```



Subtopic

动态规划之背包问题

给你一个可装载重量为 W 的背包和N个物品，每个物品有重量和价值两个属性。其中第 i 个物品的重量为 wt[i]，价值为val[i]，现在让你用这个背包装物品，最多能装的价值是多少？

第一步要明确两点，「状态」和「选择」。

先说状态，如何才能描述一个问题局面？只要给几个物品和一个背包的容量限制，就形成了一个背包问题呀。所以状态有两个，就是「背包的容量」和「可选择的物品」。

```
for 状态1 in 状态1的所有取值：
    for 状态2 in 状态2的所有取值：
        for ...
            dp[状态1][状态2][...] = 择优(选择1，选择2...)
```

第二步要明确 dp 数组的定义。
首先看看刚才找到的「状态」，有两个，也就是说我们需要一个二维 dp 数组。
dp[i][w] 的定义如下：对于前 i 个物品，当前背包的容量为 w，
这种情况下可以装的最大价值是dp[i][w]。
比如说，如果 dp[3][5] = 6，其含义为：对于给定的一系列物品中，若只对前 3 个物品进行选择，当背包容量为 5 时，最多可以装下的价值为 6。
根据这个定义，我们想求的最终答案就是 dp[N][W]。base case 就是 dp[0][..] = dp[..][0] = 0，因为没有物品或者背包没有空间的时候，能装的最大价值就是 0。

```
int dp[N+1][W+1]
dp[0][..] = 0
dp[..][0] = 0

for i in [1..N]:
    for w in [1..W]:
        dp[i][w] = max(
            把物品 i 装进背包,
            不把物品 i 装进背包
        )
return dp[N][W]
```

第三步，根据「选择」，思考状态转移的逻辑。

简单说就是，上面伪码中「把物品 i 装进背包」和「不把物品 i 装进背包」怎么用代码体现出来呢？
先重申一下刚才我们的 dp 数组的定义：

dp[i][w] 表示：对于前 i 个物品，当前背包的容量为 w 时，这种情况下可以装下的最大价值是 dp[i][w]。

如果你没有把这第 i 个物品装入背包，那么很显然，最大价值 dp[i][w] 应该等于 dp[i-1][w]，继承之前的结果。

如果你把这第 i 个物品装入了背包，那么 dp[i][w] 应该等于 dp[i-1][w - wt[i-1]] + val[i-1]。

首先，由于 i 是从 1 开始的，所以 val 和 wt 的索引是 i-1 时表示第 i 个物品的价值和重量。

而 dp[i-1][w - wt[i-1]] 也很好理解：你如果装了第 i 个物品，就要寻求剩余重量 w - wt[i-1] 限制下的最大价值，加上第 i 个物品的价值 val[i-1]。

```
for i in [1..N]:
    for w in [1..W]:
        dp[i][w] = max(
            dp[i-1][w],
            dp[i-1][w - wt[i-1]] + val[i-1]
        )
return dp[N][W]
```

```
int knapsack(int W, int N, vector<int>& wt, vector<int>& val) {
    // base case 已初始化
    vector<vector<int>> dp(N + 1, vector<int>(W + 1, 0));
    for (int i = 1; i <= N; i++) {
        for (int w = 1; w <= W; w++) {
            if (w - wt[i-1] < 0) {
                // 这种情况下只能选择不装入背包
                dp[i][w] = dp[i - 1][w];
            } else {
                // 装入或者不装入背包，择优
                dp[i][w] = max(dp[i - 1][w - wt[i-1]] + val[i-1],
                               dp[i - 1][w]);
            }
        }
    }

    return dp[N][W];
}
```

300. 最长上升子序列

给定一个无序的整数数组，找到其中最长上升子序列的长度。
示例:输入: [10,9,2,5,3,7,101,18] 输出:4
解释:最长的上升子序列是 [2,3,7,101]，它的长度是 4。
说明:可能会有多种最长上升子序列的组合，你只需要输出对应的长度即可。
你算法的时间复杂度应该为 O(n^2) 。
进阶: 你能将算法的时间复杂度降低到 O(n log n) 吗?
链接：https://leetcode-cn.com/problems/longest-increasing-subsequence

1、动态规划解法【time：O(n^2)】

```
public int lengthOfLIS(int[] nums) {
    int[] dp = new int[nums.length];
    // dp 数组全都初始化为 1
    Arrays.fill(dp, 1);
    for (int i = 0; i < nums.length; i++) {
        for (int j = 0; j < i; j++) {
            if (nums[i] > nums[j])
                dp[i] = Math.max(dp[i], dp[j] + 1);
        }
    }

    int res = 0;
    for (int i = 0; i < dp.length; i++) {
        res = Math.max(res, dp[i]);
    }
    return res;
}
```

2、二分查找解法【time：O(NlogN)】

```
public int lengthOfLIS(int[] nums) {
    int[] top = new int[nums.length];
    // 牌堆数初始化为 0
    int piles = 0;
    for (int i = 0; i < nums.length; i++) {
        // 要处理的扑克牌
        int poker = nums[i];

        /***** 搜索左侧边界的二分查找 *****/
        int left = 0, right = piles;
        while (left < right) {
            int mid = (left + right) / 2;
            if (top[mid] > poker) {
                right = mid;
            } else if (top[mid] < poker) {
                left = mid + 1;
            } else {
                right = mid;
            }
        }
        /*********************************/

        // 没找到合适的牌堆，新建一堆
        if (left == piles) piles++;
        // 把这张牌放到牌堆顶
        top[left] = poker;
    }
    // 牌堆数就是 LIS 长度
    return piles;
}
```