动态规划

（DYNAMIC PROGRAMMING）

什么是动态规划?

- 是运筹学的一个分支，是**求解决策过程最优化的过程。**20世纪50年代初，由美国数学家贝尔曼等人在研究多阶段决策过程的优化问题时提出。
- **动态规划(算法)是一大类算法**，与分治算法的自顶向下求解和与贪心算法寻找局部最优解有本质的区别。**通过循环做出每一步的最优解从而自底向上的得出对问题的整体最优解**;

动态规划的三要素如下:

(1)重叠子问题——子问题的计算方式大致相同

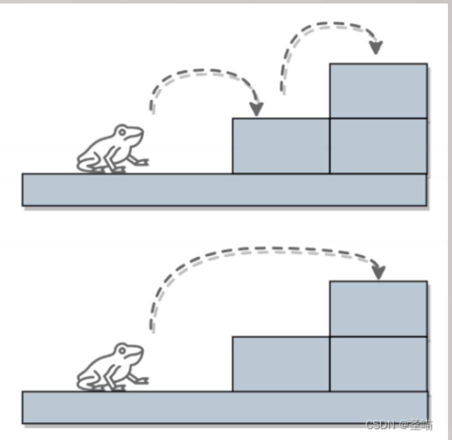
(2)复合最优子结构——可以通过子问题的最优解,推导出问题的最优解

(3)状态转移方程——描述怎么从子问题最优解推导当前问题的最优解

动态规划解题模式

- **确定定义 -> 找初始值 -> 思考关系 -> 写代码解**

基础问题：青蛙跳台阶



【问题描述】一只青蛙一次可以跳上1级台阶，也可以跳上2级台阶。求该青蛙跳上一个 n 级的台阶总共有多少种跳法？

- 初始值：

  dp[0]=1， dp[1]=1，

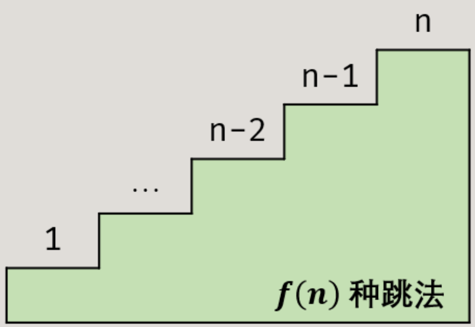
  dp[2]=2 （跳2个1级台阶，或跳1个2级台阶）

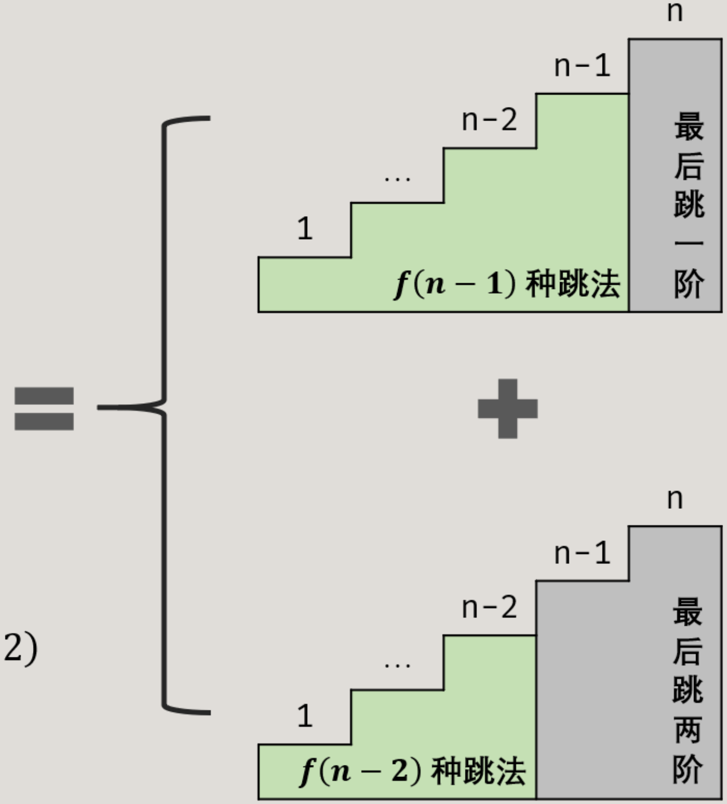
- 状态转移方程

  总跳法=最后跳一步的跳法数 + 最后跳两步的跳法数

  dp[n]=dp[n-1]+dp[n-2]

- 问题本质是求斐波那契数列的变体解问题





即 $f(n)=f(n-1)+f(n-2)$

连续子数组的最大和

【问题描述】输入一个整型数组，数组中的一个或连续多个整数组成一个子数组。求所有子数组的和的最大值。要求时间复杂度为O(n)。

- 示例:

输入: nums = [-2,1,-3,4,-1,2,1,-5,4]
输出: 6
解释: 连续子数组 [4,-1,2,1] 的和最大，为 6。

解题分析

- 【状态定义】：设动态规划列表 dp[i] 代表以元素 nums[i] 为结尾的连续子数组最大和。
- 【状态转移方程】：
  - 若 dp[i−1]≤0， 说明 dp[i−1] 对 dp[i] 产生负贡献，即 dp[i−1]+nums[i] 还不如 nums[i] 本身大。执行 dp[i]=nums[i]；
  - 当 dp[i−1]>0 时：执行 dp[i]=dp[i−1]+nums[i]；
- 【初始状态】： dp[0]=nums[0]

示例：输入NUMS

nums	-2	1	-3	4	-1	2	1	-5	4
dp	-2	1	-2	4	3	5	6	1	5

状态定义：

$dp[i]$ 代表以元素 $nums[i]$ 为结尾的连续子数组最大和

转移方程：

$$dp[i] = \begin{cases} dp[i-1] + nums[i] & , dp[i-1] > 0 \\ nums[i] & , dp[i-1] \leq 0 \end{cases}$$

返回值：

返回 dp 列表中的最大值，代表全局最大值。

DP空间优化：

直接在nums上进行修改dp值，省去 dp 列表使用的额外空间。

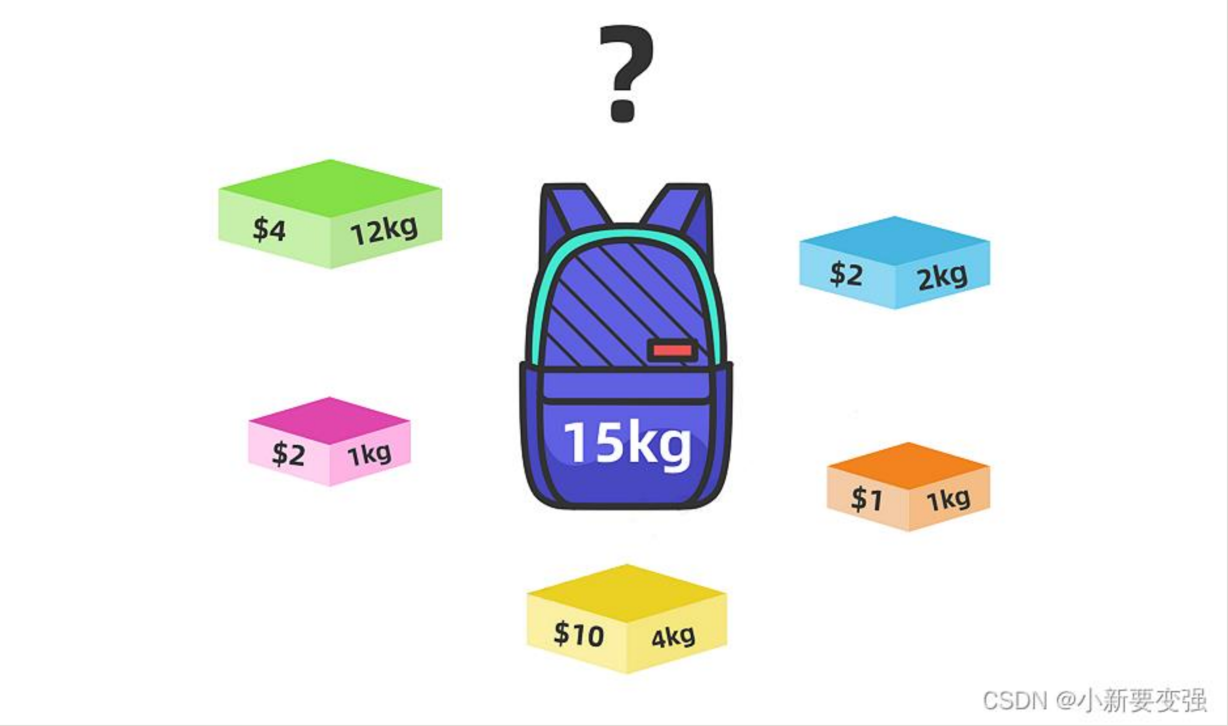
DP空间优化：dp[i] 只与 dp[i−1] 和 nums[i] 有关系，因此可以将原数组 nums 用作 dp 列表，即直接在 nums 上修改即可。

空间复杂度从 O(N) 降至 O(1)

【代码实现】

```
class Solution {
    int maxSubArray(vector<int>& nums) {
        int n=nums.size();
        int res=nums[0];
        for(int i=1;i<n;i++)
        {
            nums[i]+=max(nums[i-1],0);
            res=max(res,nums[i]);
        }
        return res;
    }
};
```

0-1背包问题（动态规划的求解）



0-1背包问题的描述

- 有一个容量为 G 的背包，和一些物品。这些物品分别有两个属性，容量 w 和价值v，每种物品只有一个。要求用这个背包装下价值尽可能多的物品，求该最大价值，背包可以不被装满。

在最优解中，每个物品只有两种可能的情况，即在背包中或者不在背包中（背包中的该物品数为0或1），因此称为0-1背包问题。

问题分析：

- 原问题的子问题：(子问题中物品数和背包容量都应当作为变量) 子问题确定为背包容量为 j 时，求前 i 个物品所能达到最大价值。
  - ✓ 对于每一个物品 i，有两种结果：不能装下或者能装下。
  - ✓ 【第一种结果】，包的容量比物品容量小，不能装下，这时的最大价值和前 i-1个物品的最大价值是一样的。
  - ✓ 【第二种结果】，还有足够的容量装下该物品i，但是装了不一定是最大价值，所以要进行比较。

- 【DP数组】：动态规划的核心问题是【穷举】，在**穷举**过程中，我们需要一个DP table来优化穷举的过程，记录子问题的结果
- 【确定状态】："状态"对应的"值"即为背包容量为 j 时，求前 i 个物品所能达到最大价值，设为dp[i][j]。初始时，dp[0][0]为0，没有物品也就没有价值。

- **【确定状态转移方程】** 由上述分析，第 i 个物品的容量为w[i]，价值为v[i]，则状态转移方程dp(i, j)为

$$
dp(i,j)=\begin{cases} dp(i-1,j), & \text{若}w[i]>j\text{（装不下）} \\ \max(dp(i-1,j-w[i])+v[i],\ dp(i-1,j)), & \text{若}w[i]<=j\text{（可以装下）} \end{cases}
$$

5件物品分别如下：

Value :　v[ ] = {3, 4, 5, 8, 10};　//物品价值

Weight:　w[ ] = {2, 3, 4, 5, 9};　//物品容量

			背包允许容量 *j* （*j*=0~10）时能装的物品最大价值												
物品	**价值**	**容量**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**		
1	3	2	0	0	3	3	3	3	3	3	3	3	3		
2	4	3													
3	5	4													
4	8	5													
5	10	9													

【代码实现】// dp[i][j]含义：背包容量为*j*时，在前*i*件物品中取小于等于*i*件物品，此时取得的物品的价值最大

```
int main( ){
int num = 5;
 int capacity = 10; //背包容量为10
int weight[ ] = {2, 3, 4, 5, 9};   //重量 2 3 4 5 9
int value[ ] = {3, 4, 5, 8, 10};   //价值 3 4 5 8 10
int maxValue = zeroOnePackage(weight, value, num, capacity);
cout<< maxValue ;
}
public static int zeroOnePackage(int[ ] weight,int[ ] value,int num,int capacity) {
  intdp [ ][ ] = new int[num][capacity +1];
  for (int i = 1; i < num; i++) {
          for (int j = 1; j <= capacity; j++) {
               if (weight[i] > j) {dp[i][j] = dp[i - 1][j];   }
               else {
                    dp[i][j] = Max(dp[i - 1][j - weight[i]] + value[i], dp[i -1][j]);  }
                  }
         }
            return dp[num-1][capacity];
}
```

总结

- **动态规划**的基本思想是利用已求解的**子问题的最优解**来推导出更大问题的最优解，从而避免了重复计算。
- 通常采用**自底向上**的方式进行求解，先求解出**小规模的问题**，然后逐步推导出**更大规模的问题**，直到求解出整个**问题的最优解**。

求解动态规划的基本步骤

- **定义状态：**将问题划分为若干个子问题，并定义状态表示子问题的解；
- **定义状态转移方程：**根据子问题之间的关系，设计状态转移方程，即如何从已知状态推导出未知状态的计算过程；
- **确定初始状态：**定义最小的子问题的解；
- **自底向上求解：**按照状态转移方程，计算出所有状态的最优解；
- **根据最优解构造问题的解**。