动态规划

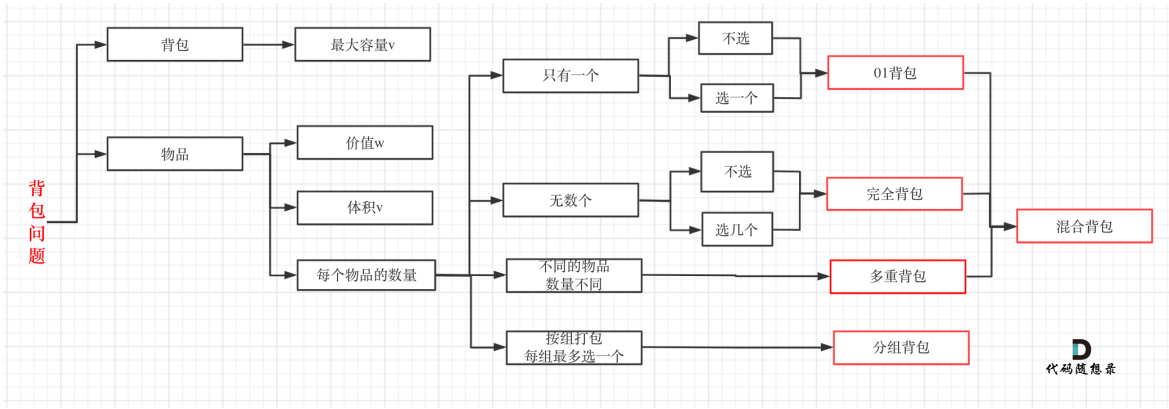
算法

背包问题

背包问题有多种背包方式，常见的有：01背包、完全背包、多重背包、分组背包和混合背包等等。

要注意题目描述中商品是不是可以重复放入。

即一个商品如果可以重复多次放入是完全背包，而只能放入一次是01背包，写法还是不一样的。



01背包

问题描述

有n件物品和一个最多能背重量为w 的背包。第i件物品的重量是weight[i]，得到的价值是value[i] 。**每件物品只能用一次**，求解将哪些物品装入背包里物品价值总和最大。

举例：背包最大重量为4。

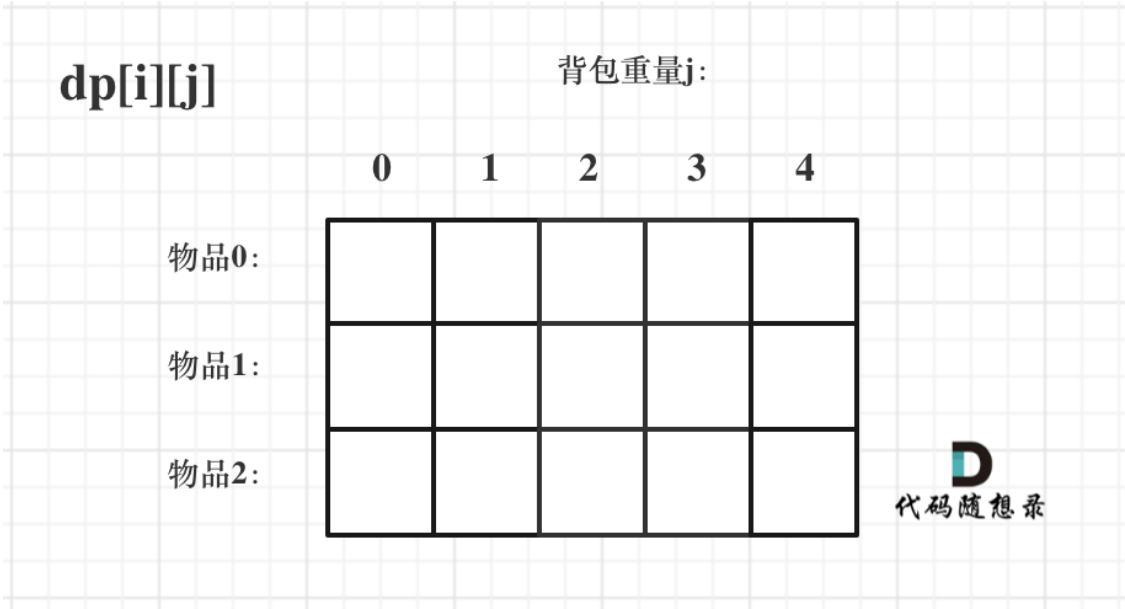
物品为：

	重量	价值
物品0	1	15
物品1	3	20
物品2	4	30

问背包能背的物品最大价值是多少？

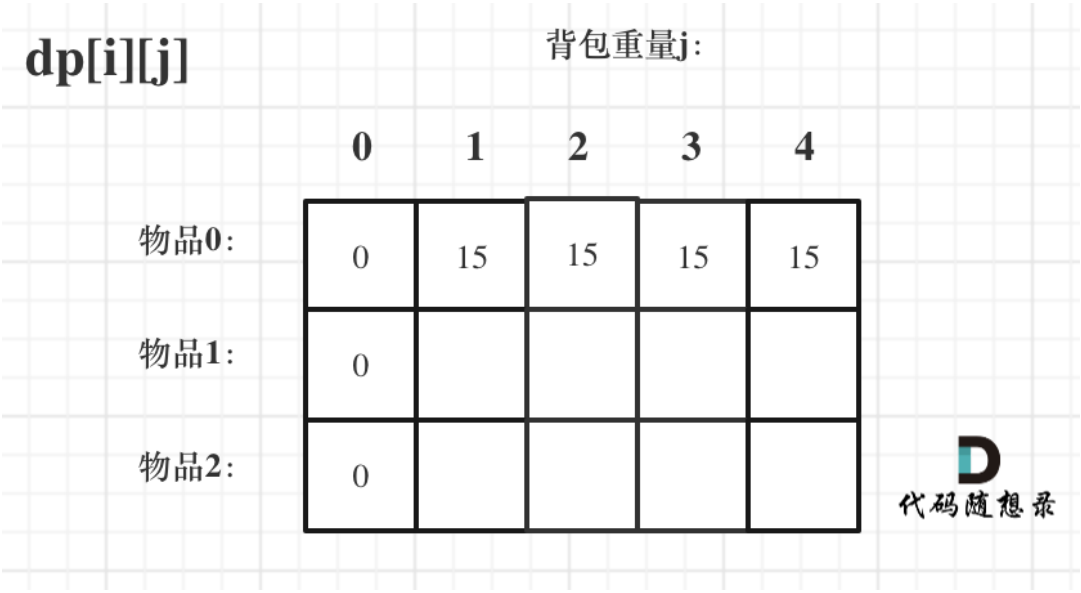
分析

1. 确定 dp 数组定义，`dp[i][j]` 表示从 `[0 - i]` 个物品中任取，放进容量为 `j` 的背包，得到的最大价值。



2. 确定递推公式，`dp[i][j]` 可以从当前状态反推
    1. 选择当前物品 `i` ，则 `dp[i][j] = dp[i - 1][j - weights[i]] + values[i]`，即选择当前物品 `i` 后，表示只能用容量为 `j - weights[i]` 的背包从 `[0, i - 1]` 个物品中任取了。这里要注意， `j >= weights[i]`
    2. 不选择物品 `i`，则 `dp[i][j] = dp[i - 1][j]`，表示用容量为 `j` 的背包从物品 `[0, i]` 个物品中任取，这里 `j < weights[i]`

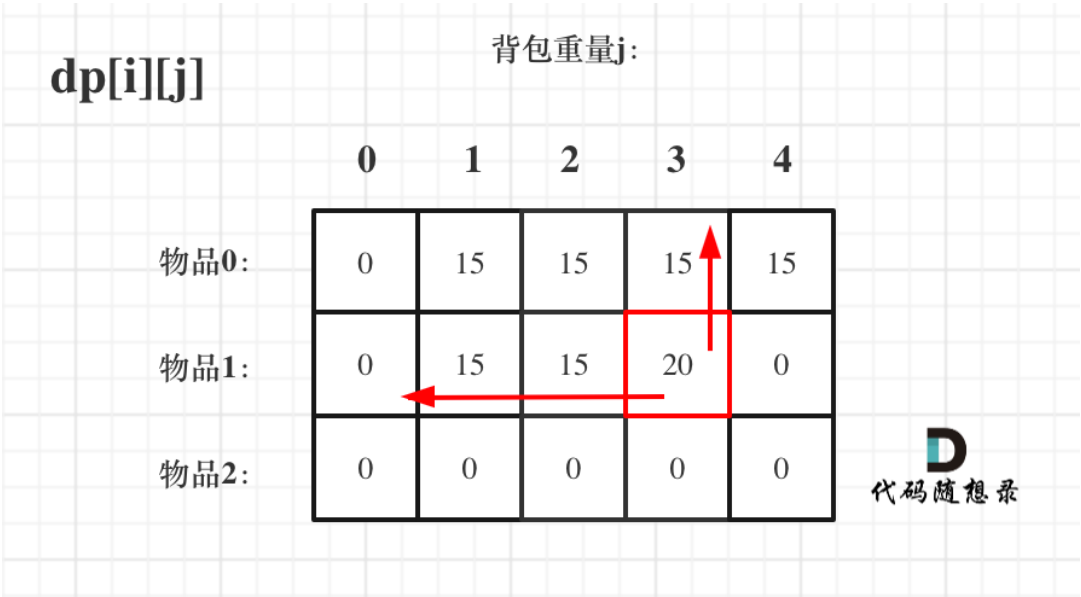
    所以递推公式 `dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weights[i]] + values[i])`
3. `dp` 数组初始化
    1. `dp[i][0] = 0` 背包容量为 0 的背包，任何物品都放不下
    2. `dp[0][j]` 当背包容量 `j > weights[0]` 时，`dp[0][j] = weights[0]`，其余 `dp[0][j] = 0`



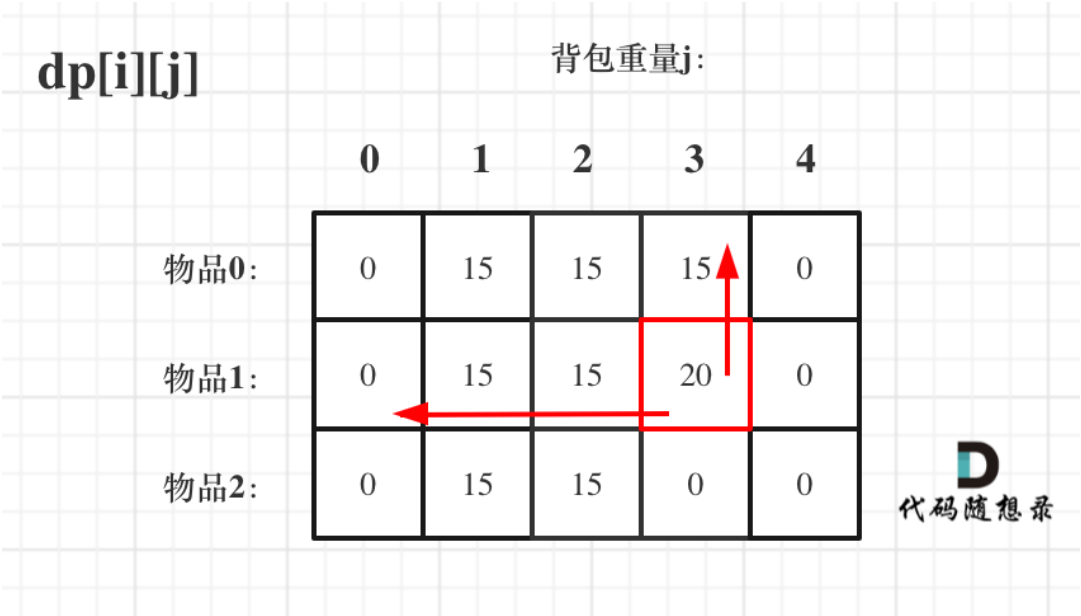
其他都默认初始化为 0 即可

4. 遍历顺序：从 dp 数组示意图看，存在两个遍历的方向，背包容量、物品；先遍历谁呢？这里都可以，因为根据递推公式 `dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weights[i]] + values[i])`，`dp[i][j]` 从由 `dp[i - 1][j]` 和 `dp[i - 1][j - weights[i]] + values[i]` 得到的，都在其正上或左上角方向。而先遍历谁都能得到其左上角的值

    1. 先遍历物品，再遍历背包



    2. 先遍历背包，再遍历物品



5. 打印 dp 数组

6. 确定可优化方向：空间（滚动数组）

优化方向

空间优化- 滚动数组

对于背包问题其实状态都是可以压缩的。

在使用二维数组的时候，递推公式：`dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weight[i]] + value[i]);`

其实可以发现如果把`dp[i - 1]`那一层拷贝到`dp[i]`上，表达式完全可以是：`dp[i][j] = max(dp[i][j], dp[i][j - weight[i]] + value[i])`；

与其把`dp[i - 1]`这一层拷贝到`dp[i]`上，不如只用一个一维数组了，只用`dp[j]`（一维数组，也可以理解是一个滚动数组）。

这就是滚动数组的由来，需要满足的条件是上一层可以重复利用，直接拷贝到当前层。

一定要切记，`dp[i][j]`的含义，`i`表示物品，`j`表示背包容量，`dp[i][j]`表示从物品`[0, i]`中任务，放进容量为`j`的背包，最大价值是多少

那么，一维数组下的五部曲为

1. 确定dp数组含义：`dp[j]`任取物品，放进容量为`j`的背包，最大价值为`dp[j]`
2. 递推公式: 注意这里也是两层循环，遍历物品 / 遍历背包容量，所以还是存在`i`的
    1. 和之前一样，如果取当前物品`i`，则表示`j >= weights[i]`，那么`dp[j] = max(dp[j], dp[j - weights[i]] + values[i])`，即只能用容量为`j - weights[i]`的背包从剩下物品任取了
    2. 不取当前物品，则`dp[j] = dp[j]`不做任何操作，还是用容量为`j`的背包从剩下的物品中任取
3. 初始化：因为`dp[j]`表示容量为`j`的背包所能放物品的最大价值，如果背包容量为`0`，则价值为`0`，所以`dp[0] = 0`
4. 遍历顺序： 还是按照先遍历物品，后遍历背包的顺序

    但要注意，背包容量是从大到小遍历的，不像二维dp中从小到大。

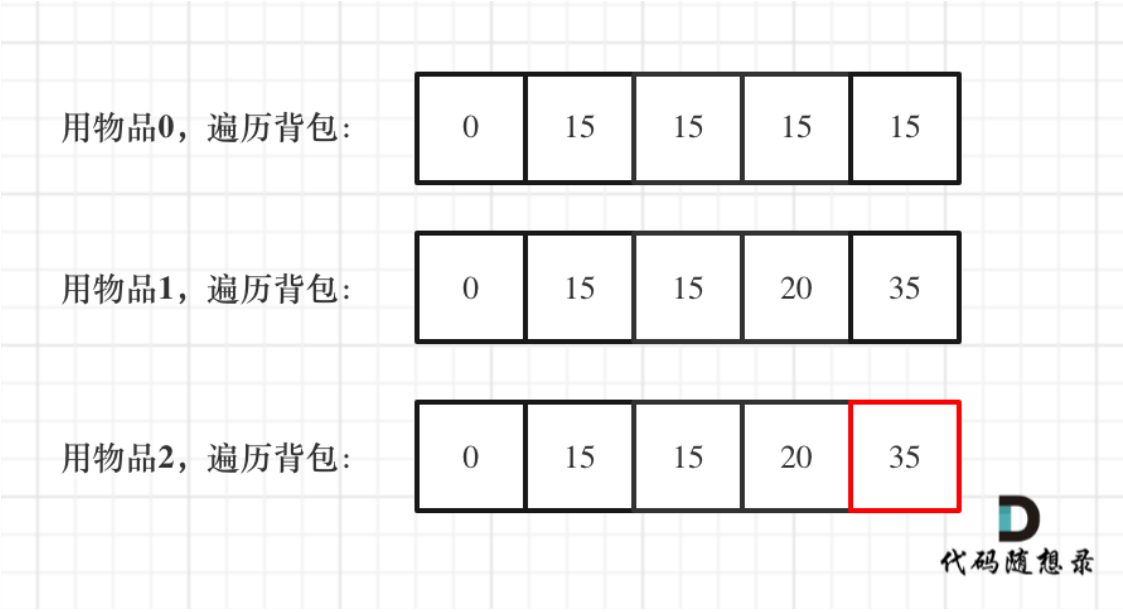
    因为根据递推公式`dp[j] = max(dp[j], dp[j - weights[i]]) + values[i]`，计算`dp[j]`需要依赖到`dp[j - weights[i]]`，注意这里的`dp[j - weights[i]]`还是上一行的`dp[j - weights[i]]`即`dp[i - 1][j - weights[i]]`。

    如果从小到大遍历，那么当前行的`dp[j - weights[i]]`会先被计算，覆盖了上一行的`dp[j - weights[i]]`，那么在计算后续的`d[j]`时，使用的就是当前行的`dp[j - weights[i]]`，而不是正确的上一行的`dp[j - weights[i]]`了

    所以遍历代码

    ```
    for(int i = 0; i < weight.size(); i++) { // 遍历物品
        for(int j = bagweight; j >= weight[i]; j--) { // 遍历背包容量
            dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);

        }
    }
    ```

5. 打印dp数组



题目

力扣没有01背包题目，可看这个https://kamacoder.com/problempage.php?pid=1046

46. 携带研究材料（第六期模拟笔试）

> 小明是一位科学家，他需要参加一场重要的国际科学大会，以展示自己的最新研究成果。他需要带一些研究材料，但是他的行李箱空间有限。这些研究材料包括实验设备、文献资料和实验样本等等，它们各自占据不同的空间，并且具有不同的价值。
>
> 小明的行李空间为 N，问小明应该如何抉择，才能携带最大价值的研究材料，每种研究材料只能选择一次，并且只有选与不选两种选择，不能进行切割。
>
> 第一行包含两个正整数，第一个整数 M 代表研究材料的种类，第二个正整数 N，代表小明的行李空间。
>
> 第二行包含 M 个正整数，代表每种研究材料的所占空间。
>
> 第三行包含 M 个正整数，代表每种研究材料的价值。
>
> 输出描述
>
> 输出一个整数，代表小明能够携带的研究材料的最大价值。
>
> 输入示例
>
> ```
> 6 1
> 2 2 3 1 5 2
> 2 3 1 5 4 3
> ```
>
> 输出示例
>
> ```
> 5
> ```
>
> 提示信息
>
> 小明能够携带 6 种研究材料，但是行李空间只有 1，而占用空间为 1 的研究材料价值为 5，所以最终答案输出 5。
>
> 数据范围:
> 1 <= N <= 5000
> 1 <= M <= 5000
> 研究材料占用空间和价值都小于等于 1000

动态规划

```
#include <iostream>
#include <vector>

int GetMaxValue2D(int types, int bag, std::vector<int> &weights, std::vector<int> &values) {
    //  1. dp[i][j] 表示在背包容量为j时，从0-i个物品中选择到的最大价值
    std::vector<std::vector<int>> dp(types, std::vector<int>(bag + 1, 0));

    //  2. 递推公式  dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weights[i] + values[i])
    //  2.1 选当前物品 dp[i][j] = dp[i - 1][j - weights[i] + values[i],
    //      可以理解为反向选择，选了i后，要在背包容量为j - weights[i]时，从 0-(i - 1)个物品中能选择到的最大价值
    //  2.2 不选当前物品 dp[i][j] = dp[i - 1][j]，表示当背包容量为j时，从 0-(i - 1)个物品中能选择到的最大价值


    //  3. 初始化
    //  3.1 dp[i][0] = 0，背包容量为0时，所有物品都无法选择，故dp[i][0] = 0
    //  3.2 dp[0][j] ? 背包容量为j时，选择物品0的最大价值，要比较j和物品0重量，j < weights[0]，则dp[0][j] = 0; 否则
dp[0][j] = weights[0]

    // 这里因为构造dp都初始化为0，可以省略
    for (int i = 0; i < types; ++i) {
        dp[i][0] = 0;
    }
    for (int j = 0; j <= bag; ++j) {
        if (j >= weights[0]) {
            dp[0][j] = weights[0];
        }
    }

    // 4. 遍历顺序，因为dp[i][j]由dp[i - 1][j]和dp[i - 1][j - weights[i]得到，在二维数组中可以想象由其左上角得到，古行前
往后遍历
    // 先遍历背包或者物品都可以，因为计算dp[i][j]时，不论先遍历谁，依赖的元素都已计算
    for (int i = 1; i < types; ++i) {
        for (int j = 1; j <= bag; ++j) {
            if (j >= weights[i]) {
                dp[i][j] = std::max(dp[i - 1][j], dp[i - 1][j - weights[i]] + values[i]);
            } else {
                dp[i][j] = dp[i - 1][j];
            }
        }

    }
```

```
    for (int i = 0; i < types; ++i) {
        for (int j = 0; j <= bag; ++j) {
            std::cout << dp[i][j] << " ";
        }
        std::cout << std::endl;
    }

    return dp[types - 1][bag];
}

int GetMaxValue1D(int types, int bag, std::vector<int> &weights, std::vector<int> &values) {
    //  1. dp[j] 表示在背包容量为j时，从0-i个物品中选择到的最大价值
    std::vector<int> dp(bag + 1, 0);

    //  2. 递推公式  dp[j] = max(dp[j], dp[j - weights[i] + values[i])
    //  2.1 选当前物品 dp[j] = dp[j - weights[i] + values[i],
    //      可以理解为反向选择，选了i后，要在背包容量为j - weights[i]时，从 0-(i - 1)个物品中能选择到的最大价值
    //  2.2 不选当前物品 dp[j] = dp[j]，表示当背包容量为j时，从 0-(i - 1)个物品中能选择到的最大价值


    //  3. 初始化 dp[0] = 0，背包容量为0时，所有物品都无法选择，故dp[0] = 0
    dp[0] = 0;

    // 4. 遍历顺序，因为dp[j]由dp[j]和dp[j - weights[i]得到，在二维数组中可以想象由其左上角得到，古行前往后遍历
    // 先遍历背包或者物品都可以，因为计算dp[j]时，不论先遍历谁，依赖的元素都已计算
    for (int i = 0; i < types; ++i) {
        for (int j = bag; j >= 0; ++j) {
            if (j >= weights[i]) {
                dp[j] = std::max(dp[j], dp[j - weights[i]] + values[i]);
            } else {
                dp[j] = dp[j];
            }
        }

    }

    return dp[bag];
}

int main() {

    std::vector<int> weights{2, 2, 3, 1, 5, 2};
    std::vector<int> values{2, 3, 1, 5, 4, 3};

    int res = GetMaxValue2D(6, 1, weights, values);
    std::cout << res << std::endl;

    return 0;
}
```

题目

416. 分割等和子集

> 给你一个 **只包含正整数** 的 **非空** 数组 `nums` 。请你判断是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。
>
> **示例 1：**
>
> ```
> 输入：nums = [1,5,11,5]
> 输出：true
> 解释：数组可以分割成 [1, 5, 5] 和 [11] 。
> ```
>
> **示例 2：**
>
> ```
> 输入：nums = [1,2,3,5]
> 输出：false
> 解释：数组不能分割成两个元素和相等的子集。
> ```

回溯法

任一元素都有两种选择：取或者不取，可以使用回溯

```
class Solution {
public:
    // 回溯法
    bool canPartition(vector<int>& nums) {
        int target = 0;
        for (auto num : nums) {
            target += num;
        }
        if (target % 2 != 0) { return false; }
        target /= 2;
        return backtrack(nums, 0, 0, target);
    }

    bool backtrack(vector<int> &nums, int idx, int sum, int target) {
        if (idx == nums.size()) {
            return sum == target;
        }

        return backtrack(nums, idx + 1, sum + nums[idx], target) ||
               backtrack(nums, idx + 1, sum, target);
    }
};
```

超时

动态规划

题目难在如何将题目转化为0-1背包问题；题目可以理解为

> 从非空数组`nums`中任取元素，每个元素最多取一次，每个元素的重量为`nums[i]`，是否可以填满容量为`sum/2`的背包

二维DP

```
class Solution {
public:
    // 动态规划 - 01背包
    bool canPartition(vector<int>& nums) {
        int sum = 0;
        for (auto num : nums) {
            sum += num;
        }

        // 边界条件：如果 sum 为奇数，则不存在子集
        if (sum % 2 != 0) { return false; }

        int bag = sum / 2;

        // 1. dp[i][j]含义：从物品[0, i]中任取，是否可以恰好填满背包容量为 sum/2 的背包
        vector<vector<int>> dp(nums.size(), vector<int>(bag + 1, false));

        // 2. 递推公式：dp[i][j] = dp[i - 1][j] || dp[i - 1][j - nums[i]]

        // 3. 初始化：dp[i][0] = true 表示容量为0时，不取任何元素即可
        for (int i = 0; i < nums.size(); ++i) {
            dp[i][0] = true;
        }
        // 3. 初始化：填充一下 j = nums[0] 的情况，不加也能过
        // if (nums[0] <= bag) {
        //     dp[0][nums[0]] = true;
        // }

        // // 4. 遍历顺序
        for (int i = 1; i < nums.size(); ++i) {
            for (int j = 1; j <= bag; ++j) {
                if (j >= nums[i]) {
                    dp[i][j] = dp[i - 1][j] || dp[i - 1][j - nums[i]];
                } else {
                    dp[i][j] = dp[i - 1][j];
                }
            }
        }

        // 5. 打印dp数组
        // for (int i = 0; i < nums.size(); ++i) {
```

```
        //     for (int j = 0; j < bag; ++j) {
        //         std::cout << dp[i][j] <<" ";
        //     }
        //     std::cout << std::endl;
        // }

        return dp[nums.size() - 1][bag];
    }
};
```

时间复杂度：`O(n * target)`，其中 `n` 是数组的长度，`target` 是整个数组的元素和的一半。需要计算出所有的状态，每个状态在进行转移时的时间复杂度为 `O(1)`

空间复杂度：`O(n * target)`，其中 `n` 是数组的长度，`target` 是整个数组的元素和的一半。需要计算出所有的状态。

一维DP

```
class Solution {
public:
    // 动态规划 - 01背包 - 一维DP
    bool canPartition(vector<int>& nums) {
        int sum = 0;
        for (auto num : nums) {
            sum += num;
        }

        // 边界条件：如果 sum 为奇数，则不存在子集
        if (sum % 2 != 0) { return false; }

        int bag = sum / 2;

        // 1. dp[j]含义：任取物品，是否可以恰好填满背包容量为 sum/2 的背包
        vector<bool> dp(bag + 1, false);

        // 2. 递推公式：dp[j] = dp[j] || dp[j - nums[i]]

        // 3. 初始化：dp[0] = true 表示容量为0时，不取任何元素即可
        dp[0] = true;

        // // 4. 遍历顺序
        for (int i = 0; i < nums.size(); ++i) {
            for (int j = bag; j >= 0; --j) {
                if (j >= nums[i]) {
                    dp[j] = dp[j] || dp[j - nums[i]];
                } else {
                    // 不做修改
                    // dp[j] = dp[j];
                }
            }
            std::cout << std::endl;
        }

        return dp[bag];
    }
};
```

时间复杂度：`O(n * target)`，其中 `n` 是数组的长度，`target` 是整个数组的元素和的一半。需要计算出所有的状态，每个状态在进行转移时的时间复杂度为 `O(1)`

空间复杂度：`O(target)`，其中 `n` 是数组的长度，`target` 是整个数组的元素和的一半。需要计算出所有的状态。

1049. 最后一块石头的重量 II

> 有一堆石头，用整数数组 `stones` 表示。其中 `stones[i]` 表示第 `i` 块石头的重量。
>
> 每一回合，从中选出**任意两块石头**，然后将它们一起粉碎。假设石头的重量分别为 `x` 和 `y`，且 `x <= y`。那么粉碎的可能结果如下：
>
> - 如果 `x == y`，那么两块石头都会被完全粉碎；
> - 如果 `x != y`，那么重量为 `x` 的石头将会完全粉碎，而重量为 `y` 的石头新重量为 `y-x`。
>
> 最后，**最多只会剩下一块** 石头。返回此石头 **最小的可能重量** 。如果没有石头剩下，就返回 `0`。
>
> **示例 1：**

```
输入：stones = [2,7,4,1,8,1]
输出：1
解释：
组合 2 和 4，得到 2，所以数组转化为 [2,7,1,8,1]，
组合 7 和 8，得到 1，所以数组转化为 [2,1,1,1]，
组合 2 和 1，得到 1，所以数组转化为 [1,1,1]，
组合 1 和 1，得到 0，所以数组转化为 [1]，这就是最优值。
```

示例 2：

```
输入：stones = [31,26,33,21,40]
输出：5
```

动态规划

问题的难点如何转化成 背包问题：问题可以抽象成将`n`块石头分为两堆，求两堆石头重量总和的最小差值。石头即物品，石头重量为物品重量，石头重量也为物品价值，背包容量为 石头总重量的一半，理想状态下，如果存在石头可以刚好装满背包，则表示两堆石头总和的最小差值为0

sumA + sumB = sum, 假设sumA <= sum/2, sumB >= sum/2，则要求最小diff为 sumB - sumA = sum - sumA - sumA = sum - 2 * sum A

二维动态规划

```cpp
class Solution {
public:
    // 动态规划 - 01背包 - 二维数组
    int lastStoneWeightII(vector<int>& stones) {
        int sum = 0;
        for (auto stone : stones) {
            sum += stone;
        }
        int bag = sum / 2;
        // 1. 确定dp数组定义 dp[i][j] 表示从石头[0, i]中任选，放入容量为j的背包，得到的最大价值
        vector<vector<int>> dp(stones.size(), vector<int>(bag + 1, 0));

        // 2. 递推公式 dp[i][j] = max(dp[i - ][j], dp[i - 1][j - stones[i]] + stones[i])

        // 3. 初始化 dp[i][0] = 0;
        for (int i = 0; i < stones.size(); ++i) {
            dp[i][0] = 0;
        }
        for (int j = stones[0]; j <= bag; ++j) {
            dp[0][j] = stones[0];
        }

        // 4. 遍历顺序
        for (int i = 1; i < stones.size(); ++i) {
            for (int j = 1; j <= bag; ++j) {
                if (j >= stones[i]) {
                    dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - stones[i]] + stones[i]);
                } else {
                    dp[i][j] = dp[i - 1][j];
                }
            }
        }

        return sum - dp[stones.size() - 1][bag] * 2;
    }
};
```

一维动态规划

```cpp
class Solution {
public:
    // 动态规划 - 01背包 - 滚动数组 一维dp
    int lastStoneWeightII(vector<int>& stones) {
        int sum = 0;
        for (auto stone : stones) {
            sum += stone;
        }
        int bag = sum / 2;

        // 1. 确定dp含义， dp[j]表示容量为 j 的背包，所放石头的最大重量
        vector<int>dp(bag + 1, 0);
```

```
        // 2. 递推公式 dp[j] = max(dp[j], dp[j - weights[i]] + values[i])

        // 3. 初始化 dp[0] = 0;
        dp[0] = 0;

        // 4. 遍历顺序
        for (int i = 0; i < stones.size(); ++i) {
            for (int j = bag; j >= 0; --j) {
                std::cout << dp[j] << " ";
                if (j >= stones[i]) {
                    dp[j] = max(dp[j], dp[j - stones[i]] + stones[i]);
                }
            }
            std::cout << std::endl;
        }

        return sum - dp[bag] * 2;
    }
};
```

494. 目标和

给你一个非负整数数组 `nums` 和一个整数 `target` 。

向数组中的每个整数前添加 `'+'` 或 `'-'` ，然后串联起所有整数，可以构造一个 **表达式** ：

- 例如，`nums = [2, 1]` ，可以在 `2` 之前添加 `'+'` ，在 `1` 之前添加 `'-'` ，然后串联起来得到表达式 `"+2-1"` 。

返回可以通过上述方法构造的、运算结果等于 `target` 的不同 **表达式** 的数目。

示例 1：

```
输入：nums = [1,1,1,1,1], target = 3
输出：5
解释：一共有 5 种方法让最终目标和为 3 。
-1 + 1 + 1 + 1 + 1 = 3
+1 - 1 + 1 + 1 + 1 = 3
+1 + 1 - 1 + 1 + 1 = 3
+1 + 1 + 1 - 1 + 1 = 3
+1 + 1 + 1 + 1 - 1 = 3
```

示例 2：

```
输入：nums = [1], target = 1
输出：1
```

回溯法

```
class Solution {
public:
    // 回溯法
    int findTargetSumWays(vector<int>& nums, int target) {
        return backtrack(nums, 0, 0, target);
    }

    int backtrack(vector<int>& nums, int idx, int sum, int target) {
        if (idx == nums.size()) {
            if (sum == target) {
                return 1;
            }
            return 0;
        }
        return backtrack(nums, idx + 1, sum + nums[idx], target) + backtrack(nums, idx + 1, sum - nums[idx],
target);
    }
};
```

时间复杂度：`O(2^n)`，其中 `n` 是数组 `nums` 的长度。回溯需要遍历所有不同的表达式，共有 `2^n` 种不同的表达式，每种表达式计算结果需要 `O(1)` 的时间，因此总时间复杂度是 `O(2^n)`

空间复杂度：`O(n)`，其中 `n` 是数组 `nums` 的长度。空间复杂度主要取决于递归调用的栈空间，栈的深度不超过 `n`。

动态规划-01背包-二维dp

```
class Solution {
public:
    int findTargetSumWays(vector<int>& nums, int target) {
        int sum = 0;
        for (auto num : nums) {
            sum += num;
        }
        int diff = sum - target;
        if (diff < 0 || diff % 2 != 0) {
            return 0;
        }
        int bag = diff / 2;

        // 1. 确定dp数组含义 dp[i][j] 表示从物品[0,i]中选择，填满背包 j 的数目
        vector<vector<int>> dp(nums.size(), vector<int>(bag + 1, 0));
        // 2. 递推函数 dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weights[i]] + values[i])
        // 3. 初始化 dp[i][0] = 1 不论物品有多少，都不选，组合数=1
        for (int i = 0; i < nums.size(); ++i) {
            dp[i][0] = 1;
        }
        // 3. 当 j == nums[0]时，背包只放nums[0]，组合数+=1
        // 因为是组合数，万一nums[0] = 0，则dp[0][0] = 2，所以这里用++的方式
        if (nums[0] <= bag) {
            dp[0][nums[0]] += 1;
        }

        // 4. 遍历顺序
        for (int i = 1; i < nums.size(); ++i) {
            // 注意这里 j 是从[0, bag]，防止出现nums[i] == 0的情况
            for (int j = 0; j <= bag; ++j) {
                if (j >= nums[i]) {
                    dp[i][j] = dp[i - 1][j] + dp[i - 1][j - nums[i]];
                } else {
                    dp[i][j] = dp[i - 1][j];
                }
            }
        }
        // 5. 打印dp数组
        for (int i = 0; i < nums.size(); ++i) {
            for (int j = 0; j <= bag; ++j) {
                std::cout << dp[i][j] << " ";
            }
            std::cout << std::endl;
        }
        return dp[nums.size() - 1][bag];
    }
};
```

动态规划-01背包-一维dp

```
class Solution {
public:
    // 动态规划 - 01背包 -一维DP
    int findTargetSumWays(vector<int>& nums, int target) {
        int sum = 0;
        for (auto num : nums) {
            sum += num;
        }
        int diff = sum - target;
        if (diff < 0 || diff % 2 != 0) {
            return 0;
        }
        int bag = diff / 2;
        // 1. 确定dp含义 dp[j] 表示装满容量为j的背包的组合数
        vector<int> dp(bag + 1, 0);
        // 2. 递推公式 dp[j] = dp[j] + dp[j - nums[i]] if j >= nums[i]
        // 3. 初始化
        dp[0] = 1;

        // 4. 遍历顺序
        for (int i = 0; i < nums.size(); ++i) {
            for (int j = bag; j >= 0; --j) {
                if (j >= nums[i]) {
                    dp[j] = dp[j] + dp[j - nums[i]];
                }
            }
        }
```

```
        return dp[bag];
    }
};
```

474. 一和零

给你一个二进制字符串数组 `strs` 和两个整数 `m` 和 `n` 。

请你找出并返回 `strs` 的最大子集的长度，该子集中 **最多** 有 `m` 个 `0` 和 `n` 个 `1` 。

如果 `x` 的所有元素也是 `y` 的元素，集合 `x` 是集合 `y` 的 **子集** 。

示例 1：

```
输入：strs = ["10", "0001", "111001", "1", "0"], m = 5, n = 3
输出：4
解释：最多有 5 个 0 和 3 个 1 的最大子集是 {"10","0001","1","0"} ，因此答案是 4 。
其他满足题意但较小的子集包括 {"0001","1"} 和 {"10","1","0"} 。{"111001"} 不满足题意，因为它含 4 个 1 ，大于 n 的值 3 。
```

示例 2：

```
输入：strs = ["10", "0", "1"], m = 1, n = 1
输出：2
解释：最大的子集是 {"0", "1"} ，所以答案是 2 。
```

回溯（超时）

```
class Solution {
public:
    std::unordered_map<std::string, int> str_0_cnt_, str_1_cnt_;
    // 回溯
    int findMaxForm(vector<string>& strs, int m, int n) {
        for (const auto &str : strs) {
            str_0_cnt_[str] = std::count(str.begin(), str.end(), '0');
            str_1_cnt_[str] = std::count(str.begin(), str.end(), '1');
        }
        return 0;
        // return backtrack(strs, 0, 0, 0, m, n, 0);
    }

    int backtrack(vector<string> &strs, int idx, int tm, int tn, int m, int n, int len) {
        if (idx == strs.size()) {
            if (tm <= m && tn <= n) {
                return len;
            } else {
                return 0;
            }
        }
        return max(backtrack(strs, idx + 1, tm + str_0_cnt_[strs[idx]], tn + str_1_cnt_[strs[idx]], m, n, len
+ 1),
                   backtrack(strs, idx + 1, tm, tn, m, n, len));
    }
};
```

动态规划-01背包

思路：把总共的 0 和 1 的个数视为背包的容量，每一个字符串视为装进背包的物品。这道题就可以使用 0-1 背包问题的思路完成，这里的目标值是能放进背包的字符串的数量。

```
class Solution {
public:
    std::unordered_map<std::string, int> str_0_cnt_, str_1_cnt_;
    int findMaxForm(vector<string>& strs, int m, int n) {
        //  0. 预处理字符
        for (const auto &str : strs) {
            str_0_cnt_[str] = std::count(str.begin(), str.end(), '0');
            str_1_cnt_[str] = std::count(str.begin(), str.end(), '1');
        }
        //  1. 确定dp数组含义 dp[i][j][k] 表示从物品 [0, i]中任选，放入容量为[j] && [k]的背包，能放的物品个数
        vector<vector<vector<int>>> dp(strs.size(), vector<vector<int>>(m + 1, vector<int>(n + 1, 0)));
        //  2. 递推公式  选当前物品i，则dp[i][j][k] = dp[i - 1][j - m][k - n] + 1，表示用剩余容量[j-m][k-n]继续从剩下
的物品[0,i-1]中选择
        //               不选当前物品i，则dp[i][j][k] = dp[i - 1][j][k]，表示用容量[j][k]继续从剩下的物品[0,i-1]中选择
        //  3. 初始化
        for (int i = 0; i < strs.size(); ++i) {
```

```
            dp[i][0][0] = 0;
        }
        for (int j = m; j >= str_0_cnt_[strs[0]]; --j) {
            for (int k = n; k >= str_1_cnt_[strs[0]]; --k) {
                dp[0][j][k] = 1;
            }
        }

        //  4. 遍历顺序
        for (int i = 1; i < strs.size(); ++i) {
            for (int j = 0; j <= m; ++j) {
                for (int k = 0; k <= n; ++k) {
                    if (j >= str_0_cnt_[strs[i]] && k >= str_1_cnt_[strs[i]]) {
                        dp[i][j][k] = max(dp[i - 1][j][k], dp[i - 1][j - str_0_cnt_[strs[i]]][k -
str_1_cnt_[strs[i]]] + 1);
                    } else {
                        dp[i][j][k] = dp[i - 1][j][k];
                    }
                }
            }
        }

        return dp[strs.size() - 1][m][n];
    }
};
```

完全背包

有N件物品和一个最多能背重量为W的背包。第i件物品的重量是weight[i]，得到的价值是value[i] 。**每件物品都有无限个（也就是可以放入背包多次）**，求解将哪些物品装入背包里物品价值总和最大。

完全背包和01背包问题唯一不同的地方就是，每种物品有无限件。

同样leetcode上没有纯完全背包问题，都是需要完全背包的各种应用，需要转化成完全背包问题，所以我这里还是以纯完全背包问题进行讲解理论和原理。

背包最大重量为4。

物品为：

	重量	价值
物品0	1	15
物品1	3	20
物品2	4	30

每件商品都有无限个！

问背包能背的物品最大价值是多少？

01背包和完全背包唯一不同就是体现在遍历顺序上，所以本文就不去做动规五部曲了，我们直接针对遍历顺序经行分析！首先再回顾一下01背包的核心代码

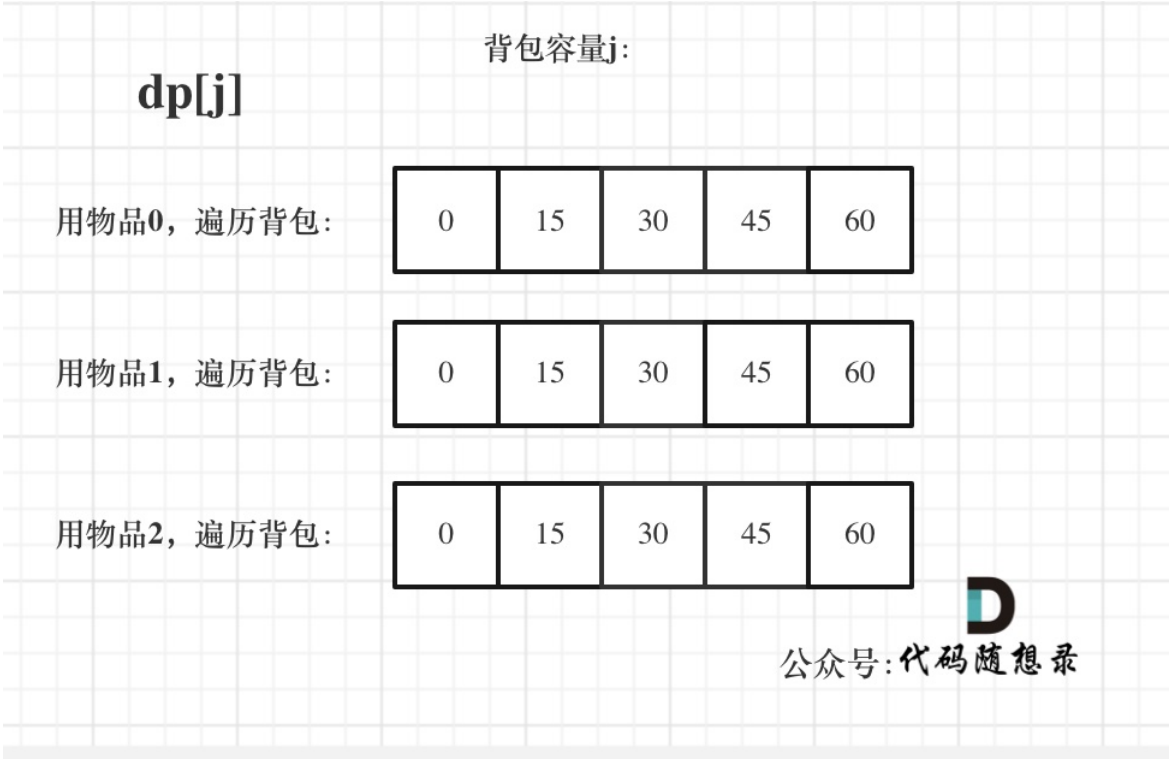
```
for(int i = 0; i < weight.size(); i++) { // 遍历物品
    for(int j = bagweight; j >= weight[i]; j--) { // 遍历背包容量
        dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);
    }
}
```

我们知道01背包内嵌的循环是从大到小遍历，为了保证每个物品仅被添加一次。

而完全背包的物品是可以添加多次的，所以要从小到大去遍历，即：

```
// 先遍历物品，再遍历背包
for(int i = 0; i < weight.size(); i++) { // 遍历物品
    for(int j = weight[i]; j <= bagWeight ; j++) { // 遍历背包容量
        dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);

    }
}
```

dp状态图如下：



还有一个很重要的问题，为什么遍历物品在外层循环，遍历背包容量在内层循环？

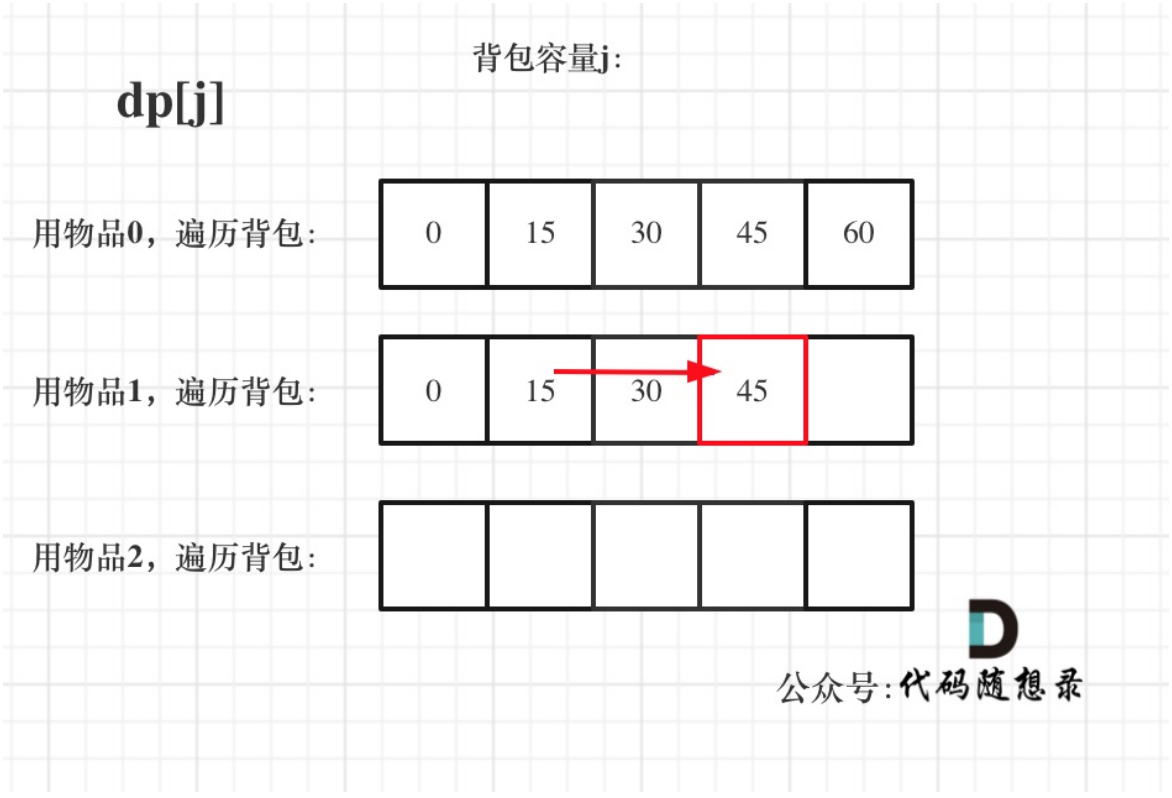
这个问题很多题解关于这里都是轻描淡写就略过了，大家都默认 遍历物品在外层，遍历背包容量在内层，好像本应该如此一样，那么为什么呢？

难道就不能遍历背包容量在外层，遍历物品在内层？01背包中二维dp数组的两个for遍历的先后循序是可以颠倒了，一维dp数组的两个for循环先后循序一定是先遍历物品，再遍历背包容量。

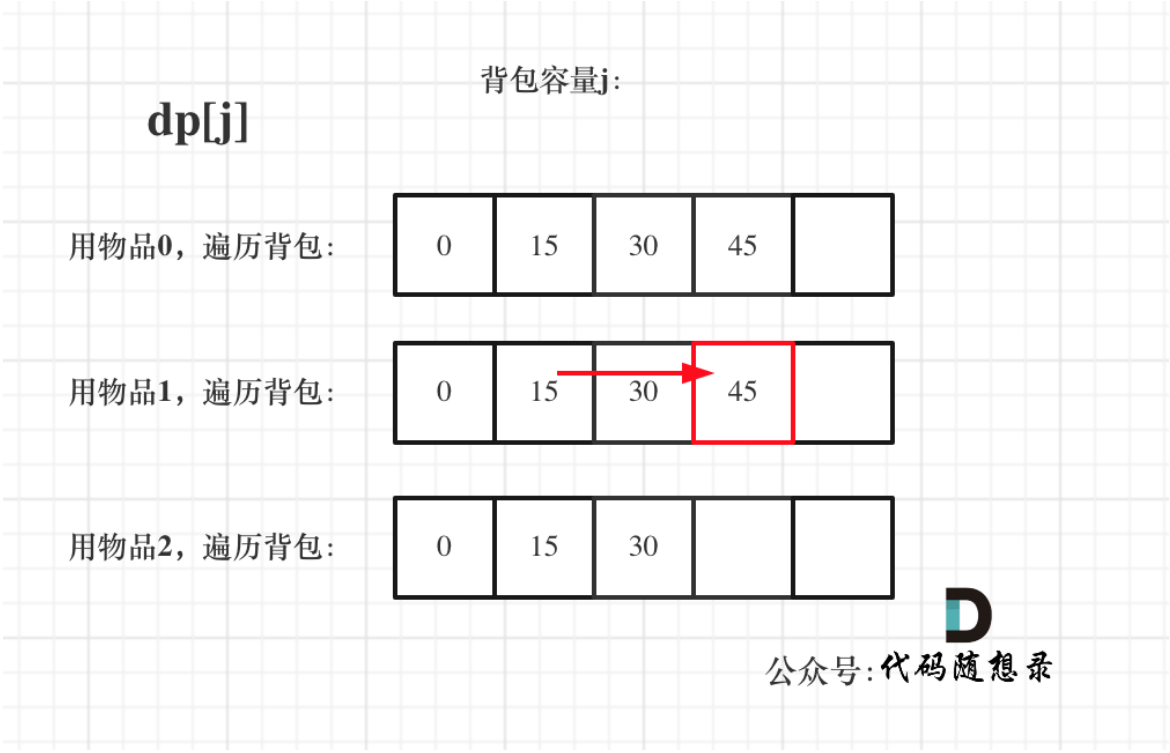
在完全背包中，对于一维dp数组来说，其实两个for循环嵌套顺序是无所谓的！

因为dp[j] 是根据 下标j之前所对应的dp[j]计算出来的。 只要保证下标j之前的dp[j]都是经过计算的就可以了。

遍历物品在外层循环，遍历背包容量在内层循环，状态如图：



遍历背包容量在外层循环，遍历物品在内层循环，状态如图：



看了这两个图，大家就会理解，完全背包中，两个for循环的先后循序，都不影响计算dp[j]所需要的值（这个值就是下标j之前所对应的dp[j]）。

先遍历背包在遍历物品，代码如下：

```
// 先遍历背包，再遍历物品
for(int j = 0; j <= bagweight; j++) { // 遍历背包容量
    for(int i = 0; i < weight.size(); i++) { // 遍历物品
        if (j - weight[i] >= 0) dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);
    }
    cout << endl;
}
```

题目

[518. 零钱兑换 II]

给你一个整数数组 `coins` 表示不同面额的硬币，另给一个整数 `amount` 表示总金额。

请你计算并返回可以凑成总金额的硬币组合数。如果任何硬币组合都无法凑出总金额，返回 `0` 。

假设每一种面额的硬币有无限个。

题目数据保证结果符合 32 位带符号整数。

示例 1：

```
输入：amount = 5, coins = [1, 2, 5]
输出：4
解释：有四种方式可以凑成总金额：
5=5
5=2+2+1
5=2+1+1+1
5=1+1+1+1+1
```

示例 2：

```
输入：amount = 3, coins = [2]
输出：0
解释：只用面额 2 的硬币不能凑成总金额 3 。
```

示例 3：

```
输入：amount = 10, coins = [10]
输出：1
```

回溯

动态规划-完全背包-二维DP

	背包容量0	背包容量1	背包容量2	背包容量3	背包容量4	背包容量5
物品0 1	1	1	1	1	1	1
物品1 2	1	1	2	2	3	3
物品2 5	1	1	2	2	3	4

	物品0 - 1	物品1 - 2	物品2 - 5
背包容量0	1	1	1
背包容量1	1	1	1
背包容量2	1	2	2
背包容量3	2		
背包容量4			
背包容量5			

动态规划-完全背包-一维DP

```
class Solution {
public:
    // 动态规划
    int change(int amount, vector<int>& coins) {
        // 1. 确定dp含义，dp[j]表示从物品[0, i]中任选，每个物品可选任意次，填满容量为j的背包的组合数
        vector<int> dp(amount + 1, 0);

        // 2. 递推公式 dp[j] = dp[j] + dp[j - coins[i]]

        // 3. 初始化：任意元素都不选，可得一种方法
        dp[0] = 1;

        // 4. 遍历顺序：求组合数需要先遍历物品，后遍历背包，
        for (int i = 0; i < coins.size(); ++i) {
            for (int j = 0; j <= amount; ++j) {
                if (j >= coins[i]) {
                    dp[j] = dp[j] + dp[j - coins[i]];
                } else {
                    // dp[j] = dp[j];
                }
            }
        }
        return dp[amount];
    }
};
```

题目

5. 最长回文子串

> 给你一个字符串 s，找到 s 中最长的回文子串。
>
> 示例 1：

> 输入：s = "babad"
> 输出："bab"
> 解释："aba" 同样是符合题意的答案。
> 示例 2：
>
> 输入：s = "cbbd"
> 输出："bb"
> 示例 3：
>
> 输入：s = "a"
> 输出："a"
> 示例 4：
>
> 输入：s = "ac"
> 输出："a"

- 动态规划
  - 思路
    - 对于一个子串而言，若是回文串，并且长度大于2，那么将其首尾字母去除后，它仍然是回文串。例如字符串 `ababa`，如果已经知道 `bab` 是回文串，那么 `ababa` 一定也是回文串，因为其首尾字母都是 `a`
    - 根据上面的思路，可使用 **动态规划**的方法解决本题
    - **$P(i, j)$定义**：表示字符串 `s` 的第 `i` 到 `j` 个字母组成的子串即 `s[i:j]` 是否为回文串

$$P(i,j)=\begin{cases} true, & \text{如果子串} S_i \dots S_j \text{是回文串} \\ false, & s[i,j] \text{本身不是回文串或者} i > j, \text{此时} s[i,j] \text{不合法} \end{cases}$$

    - **状态转移方程**：

$$P(i,j)=P(i+1,j-1) \cap (S_i == S_j)$$

      也就是说，只有$s[i + 1: j - 1]$是回文串，并且s的第i个字符和第j个字符相同时，$s[i:j]$才会使回文串
    - **初始状态**：上述讨论的前提是子串长度大于2，故子串的边界情况就是子串的长度为1或2时。对于长度为1的子串，显然是个回文串；对于长度为2的子串，只有它的两个字符相同，也是回文串、那么动态规划的边界条件如下

$$\begin{cases} P(i,i)=true, & \text{即长度为1的子串} \\ P(i,i+1)=(S_i == S_{i+1}), & \text{即长度为2的子串，且两个字符相同} \end{cases}$$

    - 根据上述思路，动态规划解法就完成了。最终答案是所有$P(i, j) = true$中$j - i + 1$（即子串长度）的最大值。注意：**在状态转移方程中，是从长度较短的字符串向长度较长的字符串进行转义的，因此需要注意动态规划的循环顺序**
  - 代码

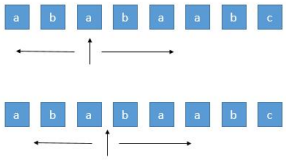
```
// 原理参考 https://leetcode-cn.com/problems/longest-palindromic-substring/solution/zui-chang-hui-wen-zi-chuan-by-leetcode-solution/
// 代码参考 https://leetcode-cn.com/problems/longest-palindromic-substring/solution/zhong-xin-kuo-san-fa-he-dong-tai-gui-hua-by-reedfa/
// 动态规划解法1
class Solution {
public:
    string longestPalindrome(string s) {
        int n = s.size();
        if (n < 2) {
            return s;
        }

        // 最长回文子串长度
        int max_size = 1;
        // 最长回文子串起始位置
        int begin = 0;

        // dp[i][j]表示s[i:j]是否为回文串
        vector<vector<int>> dp(n, vector<int>(n));

        for (int r = 1; r < n; ++r) {
            //l: 子串起始位置, r: 子串终止位置
            for (int l = 0; l < r; ++l) {
                if (s[l] == s[r] && (r - l <= 2 || dp[l + 1][r - 1])) {
                    dp[l][r] = true;
                    if (r - l + 1 > max_size) {
                        max_size = r - l + 1;
                        begin = l;
                    }
                }
            }
        }

        // 获取最长回文子串
        return s.substr(begin, max_size);
```

```
    }
};
```

- 复杂度
  - 时间复杂度：$O(n^2)$，其中n是字符串长度
  - 空间复杂度：$O(n^2)$

- 中心扩展算法
  - 思路

    回文串一定是对称的，故每次可选择一个中心，进行左右扩展，判断左右字符是否相等



    注意：**由于存在奇数长度和偶数长度的字符串，故需要分别从一个字符和两个字符之间开始扩展，所以总共有n + n - 1个中心**

  - 代码

```
// 原理参考 https://leetcode-cn.com/problems/longest-palindromic-substring/solution/xiang-xi-tong-su-de-si-lu-fen-xi-duo-jie-fa-bao-gu/
// 代码参考 https://leetcode-cn.com/problems/longest-palindromic-substring/solution/zui-chang-hui-wen-zi-chuan-by-leetcode-solution/
// 中心扩展法
class Solution {
public:
    string longestPalindrome(string s) {
        int start = 0, end = 0;
        for (int i = 0; i < s.size(); ++i) {
            auto [left1, right1] = helper(s, i, i);
            auto [left2, right2] = helper(s, i, i + 1);

            if (right1 - left1 > end - start) {
                start = left1;
                end = right1;
            }

            if (right2 - left2 > end - start) {
                start = left2;
                end = right2;
            }
        }

        return s.substr(start, end - start + 1);
    }

    pair<int, int> helper(const string &s, int left, int right) {
        while (left >= 0 && right < s.size() && s[left] == s[right]) {
            --left;
            ++right;
        }
        return {left + 1, right - 1};
    }
};
```

  - 复杂度
    - 时间复杂度：$O(n^2)$，其中n是字符串长度，长度为1和2的回文中心分别有n个和$n - 1$个，每个回文中心最多向外扩展$O(n)$次
    - 空间复杂度：$O(1)$

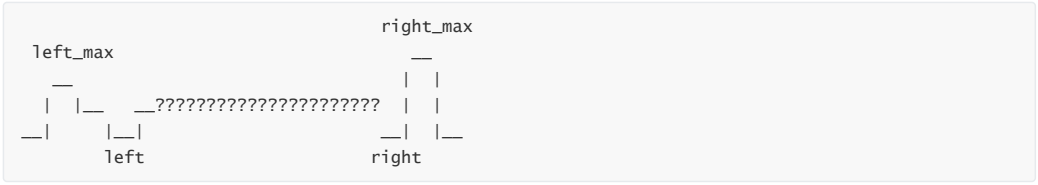
- Manacher算法：比较复杂，有时间再仔细研究吧

42. 接雨水

> 给定 n 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。
>
> 示例 1：

> 输入：height = [0,1,0,2,1,0,1,3,2,1,2,1]
> 输出：6
> 解释：上面是由数组 [0,1,0,2,1,0,1,3,2,1,2,1] 表示的高度图，在这种情况下，可以接 6 个单位的雨水（蓝色部分表示雨水）。
> 示例 2：
>
> 输入：height = [4,2,0,3,2,5]
> 输出：9

- 双指针法
  - 思路
    - 明确变量含义：
      - `left_max`：左边最大值，从左往右遍历时得到；
      - `right_max`：右边最大值，从右往左遍历时得到
      - `left`：从左往右处理的当前下标
      - `right`：从右往左处理的当前下标
    - **定理一**：在某个位置`i`处，它能存的水，取决于它左右两边的最大值中较小的一个
    - **定理二**：当从左往右处理到`left`下标时，左边的最大值`left_max`对它而言是可信的，但`right_max`是不可信的。因为`left_max`是从左往右遍历得到，而`right_max`却在不知多远的右边，中间会出现各种位置情况，right_max不一定是它右边最大的值；比如下图

      对于位置`left`而言，左边最大值一定是`left_max`，右边最大值 **大于等于**`right_max`，这时候，如果`left_max < right_max`，那么它就知道自己能存多少水了。无论右边将来会不会出现更大的`right_max`，都不影响这个结果。故当`left_max < right_max`时，我们希望去处理`left`下标，反之，我们希望去处理`right`下标
  - 代码

```cpp
// https://leetcode-cn.com/problems/trapping-rain-water/solution/jie-yu-shui-by-leetcode/327718
// 双指针法
class Solution {
public:
    int trap(vector<int>& height) {
        int left = 0, right = height.size() - 1;
        int left_max = 0, right_max = 0;
        int res = 0;

        // 小于等于，否则会漏掉 left == right的那块雨水
        while (left <= right) {
            if (left_max < right_max) {
                res += max(0, left_max - height[left]);
                left_max = max(left_max, height[left]);
                ++left;
            } else {
                res += max(0, right_max - height[right]);
                right_max = max(right_max, height[right]);
                --right;
            }
        }

        return res;
    }
};
```

  - 复杂度
    - 时间复杂度：$O(n)$，
    - 空间复杂度：$O(1)$

53. 最大子序和

> 给定一个整数数组 nums ，找到一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。
>
> 示例 1：
>
> 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 。
> 示例 2：

> 输入：nums = [1]
> 输出：1
> 示例 3：
>
> 输入：nums = [0]
> 输出：0
> 示例 4：
>
> 输入：nums = [-1]
> 输出：-1
> 示例 5：
>
> 输入：nums = [-100000]
> 输出：-100000

- 动态规划
  - 思路
    - **f(i)定义**：表示以第i个数结尾的 连续子数组的最大和，故目标就是求出 $sum_{0 \leq i \leq n-1} f(i)$
    - **状态转移方程**：每个元素$nums[i]$可以选择相加或不加，取决于$nums[i]$和$f(i - 1) + nums[i]$的大小，当$f(i) < 0$时，说明当前$num故方程为

$$f(i) = max(f(i-1) + nums[i], nums[i])$$

- **初始状态**：初始状态是$f(-1) = 0$，需要比较$f(0)和f(-1)$，$f(-1)$并不是真的有-1状态
- 代码

```
// 参考 https://leetcode-cn.com/problems/maximum-subarray/solution/zui-da-zi-xu-he-by-leetcode-solution/
// 动态规划
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int pre = 0, res = nums[0];
        for (auto num : nums) {
            pre = max(pre + num, num);
            res = max(res, pre);
        }
        return res;
    }
};
```

- 复杂度
  - 时间复杂度：$O(n)$，其中n是nums长度
    - 空间复杂度：$O(1)$

62. 不同路径

> 一个机器人位于一个 m x n 网格的左上角 （起始点在下图中标记为 “Start” ）。
>
> 机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为 “Finish” ）。
>
> 问总共有多少条不同的路径?
>
> 示例 1：
>
> 
>
>
> 输入：m = 3, n = 7
> 输出：28
> 示例 2：
>
> 输入：m = 3, n = 2
> 输出：3
> 解释：
> 从左上角开始，总共有 3 条路径可以到达右下角。
>
> 1. 向右 -> 向下 -> 向下
> 2. 向下 -> 向下 -> 向右

> 3. 向下 -> 向右 -> 向下
> 示例 3：
>
> 输入：m = 7, n = 3
> 输出：28
> 示例 4：
>
> 输入：m = 3, n = 3
> 输出：6

- 动态规划
  - 思路
    - **定义函数**：用$f(i, j)$表示从左上角到(i, j)的路径数量，其中$i \in [0, m), j \in [0, n)$
    - **状态转移方程**：由于每步只能向下或向右移动异步，因此要想走到(i, j)，上一步只能是$(i - 1, j) 或 (i, j - 1)$，故方程为

$$f(i,j) = f(i-1,j) + f(i,j-1)$$

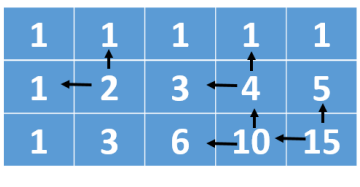
    - **初始状态**：起始位置

$$\begin{cases} f(0,0) = 1, & 起始位置 \\ f(i,0) = 1, & i \in [0,m), 最左一列只能由起始位置一直向下走 \\ f(0,j) = 1, & j \in [0,n), 最上一行只能由起始位置一直向右走 \end{cases}$$

  - 代码

```cpp
// https://leetcode-cn.com/problems/unique-paths/solution/bu-tong-lu-jing-by-leetcode-solution-hzjf/
// 动态规划
class Solution {
public:
    int uniquePaths(int m, int n) {
        vector<vector<int>> dp(m, vector<int>(n));

        for (int i = 0; i < m; ++i) {
            dp[i][0] = 1;
        }

        for (int j = 0; j < n; ++j) {
            dp[0][j] = 1;
        }

        for (int i = 1; i < m; ++i) {
            for (int j = 1; j < n; ++j) {
                dp[i][j] = dp[i-1][j] + dp[i][j-1];
            }
        }

        return dp[m-1][n-1];
    }
};
```

  - 复杂度
    - 时间复杂度：$O(mn)$
    - 空间复杂度：$O(mn)$
- 动态规划-空间优化
  - 思路
    - 二维推倒时发现，`dp[i][j] = dp[i-1][j] + dp[i][j-1]`，即只需要上一行的值即可，不再需要上上一行的了。故可使用滚动数组的方式优化空间



计算dp[i][j]时
需要上方的值+左方的值

前一次计算的dp[j]值
+ 左边dp[j – 1]的值

dp[j] = dp[j] + dp[j – 1]

    - **函数定义**：$f(j)$表示某行第 j 列的路径数量

- **状态转义方程**：$dp[j] = dp[j] + dp[j - 1]$
- **初始状态**：$dp[j] = 1, j \in [0, n)$

- 代码

```
// https://leetcode-cn.com/problems/unique-paths/solution/san-chong-shi-xian-xiang-xi-tu-jie-62-bu-4jz1/
// 动态规划-空间优化
class Solution {
public:
    int uniquePaths(int m, int n) {
        vector<int> dp(n, 1);
        for (int i = 1; i < m; ++i) {
            for (int j = 1; j < n; ++j) {
                dp[j] += dp[j - 1];
            }
        }
        return dp[n - 1];
    }
};
```

- 复杂度
  - 时间复杂度：$O(mn)$
  - 空间复杂度：$O(n)$

- 数学方法
  - 思路

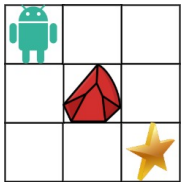
    从左上角到右下角的过程中，需要移动$m + n - 2$次，其中有$m - 1$次向下，$n - 1$次向右移动。故路径总数，等于从$m + n - 2$次移动中选择$m - 1$次向下移动的方案数，即组合数

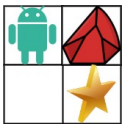
  - 代码

```
// https://leetcode-cn.com/problems/unique-paths/solution/bu-tong-lu-jing-by-leetcode-solution-hzjf/
// 数学方法
class Solution {
public:
    int uniquePaths(int m, int n) {
        long long res = 1;
        for (int x = n, y = 1; y < m; ++x, ++y) {
            res = res * x / y;
        }
        return res;
    }
};
```

  - 复杂度
    - 时间复杂度：$O(m)$
    - 空间复杂度：$O(1)$

63. 不同路径 II

> 一个机器人位于一个 m x n 网格的左上角 （起始点在下图中标记为“Start” ）。
>
> 机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为“Finish”）。
>
> 现在考虑网格中有障碍物。那么从左上角到右下角将会有多少条不同的路径？
>
> 
>
>
> 网格中的障碍物和空位置分别用 1 和 0 来表示。
>
> 示例 1：
>
> 

> 输入：obstacleGrid = [[0,0,0],[0,1,0],[0,0,0]]
> 输出：2
> 解释：
> 3x3 网格的正中间有一个障碍物。
> 从左上角到右下角一共有 2 条不同的路径：
>
> 1. 向右 -> 向右 -> 向下 -> 向下
> 2. 向下 -> 向下 -> 向右 -> 向右
>
> 示例 2：
>
> 
>
> 输入：obstacleGrid = [[0,1],[0,0]]
> 输出：1

- 动态规划
  - 思路
    - **状态定义**：$dp[i][j]$表示从起始点走到格子(i, j)处的路径数
    - **状态转义方程**：

$$dp[i][j] = \begin{cases} dp[i-1,j] + dp[j,j-1], & (i,j)\text{上无障碍物} \\ 0, & (i,j)\text{上有障碍物} \end{cases}$$

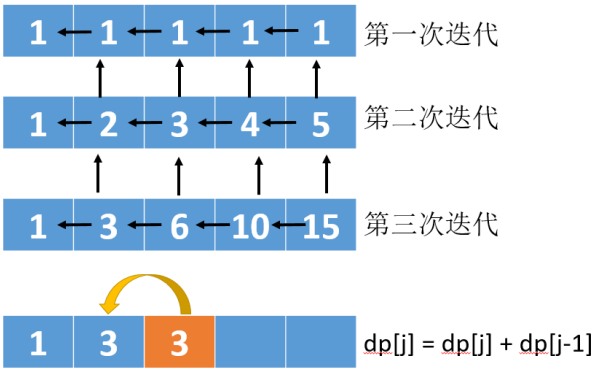
      - 如果网格(i, j)上有障碍物，则$dp[i][j] = 0$，表示走到该格子的路径数为0
      - 如果网格(i, j)可由网格$(i - 1, j) 和 (i, j - 1)$处走过来，表示路径数是它们之和，即$dp[i,j] = dp[i-1, j] + dp[i, j-1]$
    - **初始条件**

$$\begin{cases} f(0,0) = 1, & \text{起始位置，且无障碍物} \\ f(i,0) = 1, & i \in [0,m), \text{最左一列只能由起始位置一直向下走，且不存在障碍物} \\ f(0,j) = 1, & j \in [0,n), \text{最上一行只能由起始位置一直向右走，且不存在障碍物} \end{cases}$$

  - 代码

```cpp
// https://leetcode-cn.com/problems/unique-paths-ii/solution/jian-dan-dpbi-xu-miao-dong-by-sweetiee/
// 动态规划
class Solution {
public:
    int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
        int m = obstacleGrid.size(), n = obstacleGrid[0].size();
        vector<vector<int>> dp(m, vector<int>(n, 0));

        for (int i = 0; i < m && obstacleGrid[i][0] == 0; ++i) {
            dp[i][0] = 1;
        }

        for (int j = 0; j < n && obstacleGrid[0][j] == 0; ++j) {
            dp[0][j] = 1;
        }

        for (int i = 1; i < m; ++i) {
            for (int j = 1; j < n; ++j) {
                if (obstacleGrid[i][j] == 0) {
                    dp[i][j] = dp[i-1][j] + dp[i][j-1];
                }
            }
        }

        return dp[m-1][n-1];
    }
};
```

  - 复杂度
    - 时间复杂度：$O(mn)$
    - 空间复杂度：$O(mn)$
- 动态规划-空间优化
  - 思路
    - 同样由上面的状态转义方程，可知，求第`i`行时，只需要第`i-1`行的值，不再需要第`i-2`及再之前的值了，故可使用**滚动数组**方式进行空间优化
    - **函数定义**：$f(j)$表示某行第`j`列的路径数量，长度为列的长度。



- **状态转义方程**：$dp[j] = dp[j] + dp[j - 1]$。比如上图，第三次迭代时，求第三个网格 6 时，由于左边的值已经是已知的，即$dp[j-1]$，第二次迭代时的同位置的值也已知，即$dp[j]$，故新值计算方式

  > 计算当前值 = 以求出的左边值 + 上一次迭代同位置的值
  >
  > dp[j] = dp[j - 1] + dp[j]

- **初始状态**：$dp[j] = 1, j \in [0, n)$

- 代码

```
// 参考 https://leetcode-cn.com/problems/unique-paths-ii/solution/si-chong-shi-xian-xiang-xi-tu-jie-63-bu-0qyz7/
// 动态规划-空间优化
class Solution {
public:
    int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
        int m = obstacleGrid.size(), n = obstacleGrid[0].size();
        vector<int> dp(n, 0);

        dp[0] = !obstacleGrid[0][0];
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (obstacleGrid[i][j] == 1) {
                    dp[j] = 0;
                } else if (obstacleGrid[i][j] == 0 && j - 1 >= 0) {
                    dp[j] = dp[j] + dp[j-1];
                }
            }
        }

        return dp[n-1];
    }
};
```

- 复杂度
  - 时间复杂度：$O(mn)$
- 空间复杂度：$O(n)$

64. 最小路径和

> 给定一个包含非负整数的 m x n 网格 grid ，请找出一条从左上角到右下角的路径，使得路径上的数字总和为最小。
>
> 说明：每次只能向下或者向右移动一步。
>
> 示例 1：

1	3	1
1	5	1
4	2	1

> 输入：grid = [[1,3,1],[1,5,1],[4,2,1]]
> 输出：7
> 解释：因为路径 1→3→1→1→1 的总和最小。
> 示例 2：
>
> 输入：grid = [[1,2,3],[4,5,6]]
> 输出：12

- 动态规划（无状态压缩）
  - 思路

- **状态定义**：设 `dp` 为大小 `m * n` 矩阵，其中 `dp[i][j]` 表示从左上角走到 `(i, j)` 位置的最小路径和
- **状态转移方程**

$$dp[i][j] = \begin{cases} grid[i][j], & i = 0, j = 0 \\ dp[i-1][j] + grid[i][j], & i \neq 0, j = 0 \\ dp[i][j-1] + grid[i][j], & i = 0, j \neq 0 \\ min(dp[i-1][j], dp[i][j-1]) + grid[i][j], & i \neq 0, j \neq 0 \end{cases}$$

- **初始条件**：最上一行都是0，最左一列都是0
- **返回值**：返回 `dp` 矩阵右下角值，即走到终点的最小路径和
- 其实可以不建立 `dp` 矩阵浪费额外空间，直接修改 `grid[i][j]` 即可，因为 `grid[i][j] = min(grid[i-1][j], grid[i][j-1]) + grid[i][j]`；原 `grid` 矩阵元素被覆盖为 `dp` 元素后，不会再被使用

- 代码

```
// https://leetcode-cn.com/problems/minimum-path-sum/solution/zui-xiao-lu-jing-he-dong-tai-gui-hua-gui-fan-liu-c/
// 动态规划
class Solution {
public:
    int minPathSum(vector<vector<int>>& grid) {
        for (int i = 0; i < grid.size(); ++i) {
            for (int j = 0; j < grid[0].size(); ++j) {
                if (i == 0 && j == 0) {
                    continue;
                } else if (i == 0) {
                    grid[i][j] = grid[i][j-1] + grid[i][j];
                } else if (j == 0) {
                    grid[i][j] = grid[i-1][j] + grid[i][j];
                } else {
                    grid[i][j] = min(grid[i-1][j], grid[i][j-1]) + grid[i][j];
                }
            }
        }
        return grid[grid.size() - 1][grid[0].size() - 1];
    }
};
```

- 复杂度
  - 时间复杂度：$O(mn)$
  - 空间复杂度：$O(1)$

- 动态规划（状态压缩）
  - 思路
    - **状态定义**：设 `dp` 为 `m` 大小数组，其中 `dp[i]` 表示从左上角 `(0, 0)` 到第 `i-1` 行的最小路径和
    - **状态转移方程**

$$dp[i][j] = \begin{cases} grid[i][j], & i = 0, j = 0 \\ dp[j] + grid[i][j], & i \neq 0, j = 0 \\ dp[j-1] + grid[i][j], & i = 0, j \neq 0 \\ min(dp[j], dp[j-1]) + grid[i][j], & i \neq 0, j \neq 0 \end{cases}$$

    - **初始条件**：左上角为0
    - 状态压缩可用于将二维dp转为一维dp的情况，因为 `(i, j)` 处的最小路径和只和第 `(i - 1)` 行相关，和第 `(i - 1)` 行之前的都不相关
  - 代码

```
// https://leetcode-cn.com/problems/minimum-path-sum/solution/hui-su-dao-dong-tai-gui-hua-zai-dao-kong-swk9/
// 动态规划-状态压缩
class Solution {
public:
    int minPathSum(vector<vector<int>>& grid) {
        int m = grid.size(), n = grid[0].size();
        vector<int> dp(n, 0);
        dp[0] = grid[0][0];

        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (i == 0 && j != 0) {
                    dp[j] = grid[i][j] + dp[j-1];
                } else if (i != 0 && j == 0) {
                    dp[j] = grid[i][j] + dp[j];
                } else if (i != 0 && j != 0) {
                    dp[j] = grid[i][j] + min(dp[j], dp[j-1]);
                }
```

```
                }
            }
            return dp[n-1];
        }
};
```

- 复杂度
  - 时间复杂度：$O(mn)$
  - 空间复杂度：$O(n)$

70. 爬楼梯

> 假设你正在爬楼梯。需要 n 阶你才能到达楼顶。
>
> 每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢?
>
> 注意：给定 n 是一个正整数。
>
> 示例 1：
>
> 输入： 2
> 输出： 2
> 解释： 有两种方法可以爬到楼顶。
>
> 1. 1 阶 + 1 阶
> 2. 2 阶
>
> 示例 2：
>
> 输入： 3
> 输出： 3
> 解释： 有三种方法可以爬到楼顶。
>
> 1. 1 阶 + 1 阶 + 1 阶
> 2. 1 阶 + 2 阶
> 3. 2 阶 + 1 阶

- 动态规划
  - 思路
    - **状态定义**：`dp[i]`表示爬到第`i`层台阶，有`dp[i]`中方法
    - **状态转移方程**：

$$dp[i] = \begin{cases} 1, & i = 0 \\ 1, & i = 1 \\ dp[i-1] + dp[i-2], & i >= 2 \end{cases}$$

    - **初始条件**: 初始化`dp`大小为`n+1`
    - **返回值**：返回`dp[n]`
  - 代码

```
class Solution {
public:
    int climbStairs(int n) {
        vector<int> dp(n + 1, 0);
        dp[0] = 1;
        dp[1] = 1;
        for (int i = 2; i <= n; ++i) {
            dp[i] = dp[i - 1] + dp[i - 2];
        }
        return dp[n];
    }
};
```

  - 复杂度
    - 时间复杂度：$O(n)$
    - 空间复杂度：$O(n)$
- 动态规划-空间优化
  - 思路

    看上述公式，其实`dp[i]`只和`dp[i-1]，dp[i-2]`有关，故可使用两变量保存这两个值
  - 代码

```
// https://leetcode-cn.com/problems/climbing-stairs/solution/pa-lou-ti-by-leetcode-solution/
// 动态规划-空间优化
class Solution {
public:
    int climbStairs(int n) {
```

```
        int f = 1, s = 1, t = 1;
        for (int i = 2; i <= n; ++i) {
            t = f + s;
            f = s;
            s = t;
        }
        return t;
    }
};
```

- 复杂度
  - 时间复杂度：$O(n)$
  - 空间复杂度：$O(1)$

120. 三角形最小路径和

> 给定一个三角形 triangle ，找出自顶向下的最小路径和。
>
> 每一步只能移动到下一行中相邻的结点上。相邻的结点 在这里指的是 下标 与 上一层结点下标 相同或者等于 上一层结点下标 + 1 的两个结点。也就是说，如果正位于当前行的下标 i ，那么下一步可以移动到下一行的下标 i 或 i + 1 。
>
> 示例 1：
>
> 输入：triangle = [[2],[3,4],[6,5,7],[4,1,8,3]]
> 输出：11
> 解释：如下面简图所示：
> 2
> 3 4
> 6 5 7
> 4 1 8 3
> 自顶向下的最小路径和为 11（即，2 + 3 + 5 + 1 = 11）。
> 示例 2：
>
> 输入：triangle = [[-10]]
> 输出：-10

- 动态规划
  - 思路
    - **状态定义**：`f[i][j]`表示从三角形顶部走到位置`(i, j)`的最小路径和，这里的位置`(i, j)`指的是三角形中第`i`行第`j`列（均从0开始编号）的位置
    - **状态转移方程**

$$f[i][j]=\begin{cases}c[0][0], & i=0,j=0\\ f[i-1][0]+c[i][0], & j=0\\ min(f[i-1][j-1],f[i-1][j])+c[i][j], & i>j\\ f[i-1][j-1]+c[i][j], & i=j\end{cases}$$

    - **初始值**：`f[0][0] = c[0][0]`
    - **返回值**：最后一行中的最小值
  - 代码

```
// https://leetcode-cn.com/problems/triangle/solution/san-jiao-xing-zui-xiao-lu-jing-he-by-leetcode-solu/
// 动态规划
class Solution {
public:
    int minimumTotal(vector<vector<int>>& triangle) {
        int n = triangle.size();

        // f[i][j] 表示从三角形顶部走到位置 (i, j) 的最小路径和
        vector<vector<int>> f(n, vector<int>(n));

        // 初始化
        f[0][0] = triangle[0][0];

        for (int i = 1; i < n; ++i) {
            // 下标(i, 0)
            f[i][0] = f[i - 1][0] + triangle[i][0];

            // 下标(i, j) i > j, j != 0
            for (int j = 1; j < i; ++j) {
                f[i][j] = min(f[i - 1][j - 1], f[i - 1][j]) + triangle[i][j];
            }

            // 下标(i, i)
```

```
            f[i][i] = f[i - 1][i - 1] + triangle[i][i];
        }

        // 返回最后一行最小值
        return *min_element(f[n - 1].begin(), f[n - 1].end());
    }
};
```

- 复杂度
  - 时间复杂度：$O(n^2)$
  - 空间复杂度：$O(n^2)$

- 动态规划-空间优化
  - 思路
    - 回顾上个方法的状态转移方程发现，`f[i][j]`只和`f[i - 1][..]`有关，和`f[i-2][..]`及之前的状态都无关。因此不必存储这些无关状态。
    - 故可使用一维数组存储，但要注意 **按列枚举时，需要递减列下标**

      > 为什么只有在递减地枚举 j 时，才能省去一个一维数组？当我们在计算位置 (i, j) 时，f[j+1]f[j+1] 到 f[i] 已经是第 i 行的值，而 f[0] 到 f[j] 仍然是第 i-1 行的值。此时我们直接通过
      >
      > $ f[j] = \min(f[j-1], f[j]) + c[i][j]$
      >
      > 进行转移，恰好就是在(i−1,j−1) 和 (i−1,j) 中进行选择。但如果我们递增地枚举 j，那么在计算位置 (i, j时，f[0] 到 f[j−1] 已经是第 i 行的值。如果我们仍然使用上述状态转移方程，那么是在 (i, j-1) 和 (i-1, j) 中进行选择，就产生了错误。

  - 代码

```
// https://leetcode-cn.com/problems/triangle/solution/san-jiao-xing-zui-xiao-lu-jing-he-by-leetcode-solu/
// 动态规划-空间优化
class Solution {
public:
    int minimumTotal(vector<vector<int>>& triangle) {
        int n = triangle.size();
        vector<int> dp(n);

        dp[0] = triangle[0][0];

        for (int i = 1; i < n; ++i) {
            dp[i] = dp[i - 1] + triangle[i][i];
            for (int j = i - 1; j > 0; --j) {
                dp[j] = min(dp[j - 1], dp[j]) + triangle[i][j];
            }
            dp[0] += triangle[i][0];
        }

        return *min_element(dp.begin(), dp.end());
    }
};
```

  - 复杂度
    - 时间复杂度：$O(n^2)$
    - 空间复杂度：$O(n)$

- 动态规划-牛皮plus版
  - 思路：由底到顶，最后直接返回，不需要查找最小值
  - 代码

```
// https://leetcode-cn.com/problems/triangle/solution/san-jiao-xing-zui-xiao-lu-jing-he-by-leetcode-solu/491469
// 动态规划-牛皮plus
class Solution {
public:
    int minimumTotal(vector<vector<int>>& triangle) {
        vector<int> dp(triangle.back());
        for (int i = triangle.size() - 2; i >= 0; --i) {
            for (int j = 0; j <= i; ++j) {
                dp[j] = min(dp[j], dp[j + 1]) + triangle[i][j];
            }
        }
        return dp[0];
    }
};
```

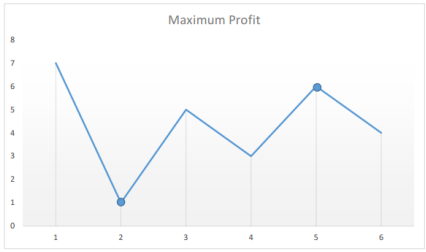
  - 复杂度
    - 时间复杂度：$O(n^2)$

- 空间复杂度：$O(n)$

121. 买卖股票的最佳时机

> 给定一个数组 prices ，它的第 i 个元素 prices[i] 表示一支给定股票第 i 天的价格。
>
> 你只能选择 某一天 买入这只股票，并选择在 未来的某一个不同的日子 卖出该股票。设计一个算法来计算你所能获取的最大利润。
>
> 返回你可以从这笔交易中获取的最大利润。如果你不能获取任何利润，返回 0 。
>
> 示例 1：
>
> 输入：[7,1,5,3,6,4]
> 输出：5
> 解释：在第 2 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，最大利润 = 6-1 = 5 。
> 注意利润不能是 7-1 = 6, 因为卖出价格需要大于买入价格；同时，你不能在买入前卖出股票。
> 示例 2：
>
> 输入：prices = [7,6,4,3,1]
> 输出：0
> 解释：在这种情况下, 没有交易完成, 所以最大利润为 0。

- 一次遍历
  - 思路
    - 假设给定的数组为 `[7, 1, 5, 3, 6, 4]`，绘制图表如下



    - 直观思路就是，如果在历史最低点购买股票，在高点卖出，就可获得最大收益
  - 算法

    遍历价格数组，记录历史最低点，然后再每一天都考虑是否可卖出，收益是多少？遍历完得到最大值即可
  - 代码

```cpp
// https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock/solution/121-mai-mai-gu-piao-de-zui-jia-shi-ji-by-leetcode-/
// 一次遍历
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int minprice = INT_MAX, maxprofit = 0;
        for (int price : prices) {
            if (price < minprice) {
                minprice = price;
            } else {
                maxprofit = max(maxprofit, price - minprice);
            }
        }
        return maxprofit;
    }
};
```

  - 复杂度
    - 时间复杂度：$O(n)$
    - 空间复杂度：$O(1)$

131. 分割回文串

> 给你一个字符串 s，请你将 s 分割成一些子串，使每个子串都是 回文串 。返回 s 所有可能的分割方案。
>
> 回文串 是正着读和反着读都一样的字符串。
>
> 示例 1：
>
> 输入：s = "aab"
> 输出：[["a","a","b"],["aa","b"]]
> 示例 2：
>
> 输入：s = "a"
> 输出：[["a"]]

139. 单词拆分

> 给定一个非空字符串 s 和一个包含非空单词的列表 wordDict，判定 s 是否可以被空格拆分为一个或多个在字典中出现的单词。
>
> 说明：
>
> 拆分时可以重复使用字典中的单词。
> 你可以假设字典中没有重复的单词。
> 示例 1：
>
> 输入: s = "leetcode", wordDict = ["leet", "code"]
> 输出: true
> 解释: 返回 true 因为 "leetcode" 可以被拆分成 "leet code"。
> 示例 2：
>
> 输入: s = "applepenapple", wordDict = ["apple", "pen"]
> 输出: true
> 解释: 返回 true 因为 "applepenapple" 可以被拆分成 "apple pen apple"。
> 注意你可以重复使用字典中的单词。
> 示例 3：
>
> 输入: s = "catsandog", wordDict = ["cats", "dog", "sand", "and", "cat"]
> 输出: false

- 动态规划
  - 思路
    - **状态定义**：`dp[i]` 表示字符串前 i 个字符组成的字符串 `s[0 : i - 1]` 是否可由字典中单词表示
    - **状态转移方程**：遍历 `s[0 : i - 1]` 的每个子串 `s[j : i - 1]`，若 `dp[j]` 为 `true`，且子串 `s[j : i - 1]` 存在于字典中，则 `dp[i]` 为 `true`，`check(str)` 表示子串 `str` 是否出现在字典中。公式如下

$$dp[i] = \{dp[j] \&\& check(s[j..i-1]), \quad j \in [0, i)$$

    - **初始值**：默认 `dp[0]=true`，表示空串
  - 代码

```cpp
// https://leetcode-cn.com/problems/word-break/solution/dan-ci-chai-fen-by-leetcode-solution/
// 动态规划
class Solution {
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        // dp[i] 表示s[0, i]是否可由wordDict中单词表示
        vector<bool> dp(s.size() + 1, false);
        unordered_set<string> lookup;
        for (auto word : wordDict) {
            lookup.insert(word);
        }

        // 表示空串
        dp[0] = true;

        for (int i = 1; i <= s.size(); ++i) {
            for (int j = 0; j < i; ++j) {
                if (dp[j] && lookup.find(s.substr(j, i - j)) != lookup.end()) {
                    dp[i] = true;
                    break;
                }
            }
        }

        return dp[s.size()];
    }
};
```

  - 复杂度
    - 时间复杂度：$O(n^2)$，其中 n 为字符串 s 的长度。我们一共有 O(n)个状态需要计算，每次计算需要枚举 O(n) 个分割点，哈希表判断一个字符串是否出现在给定的字符串列表需要 O(1)的时间，因此总时间复杂度为 $O(n^2)$
    - 空间复杂度：$O(n)$，其中 n 为字符串 s 的长度。我们需要 O(n) 的空间存放 $\textit{dp}$ 值以及哈希表亦需要 O(n)的空间复杂度，因此总空间复杂度为 O(n)

152. 乘积最大子数组

> 给你一个整数数组 nums，请你找出数组中乘积最大的连续子数组（该子数组中至少包含一个数字），并返回该子数组所对应的乘积。
>
> 示例 1：

> 输入: [2,3,-2,4]
> 输出: 6
> 解释: 子数组 [2,3] 有最大乘积 6。
> 示例 2:
>
> 输入: [-2,0,-1]
> 输出: 0
> 解释: 结果不能为 2, 因为 [-2,-1] 不是子数组。

- 动态规划
  - 思路
    - 错误思路
      - **状态定义**：$f_{max}(i)$表示以第`i`个元素结尾的乘积最大子数组的乘积
      - **状态转移方程**：$f*{max}(i)* = \max{i=1}^{n}{f(i - 1) \times a*{i}, a{i}}$，假设`a`表示数组。则方程表示以第`i`个元素结尾的成绩最大子数组的乘积是从 前面$f_{max}(i - 1)$对应的一段，或者单独成为一段 的两个值的最大值。
      - **但这样思路是错误的**：因为这的定义不满足`最优子结构`，即假如`a = {5, 6, -3, 4, -3}`，那么此时$f*{max}*$对应的序列为`{5, 30, -3, 4, -3}`，故答案为`30`。但实际上，答案应该是数组全体数字的乘积，问题出在哪里呢？ **问题在最后一个*-3*对应的$f*{max}$的值不是`-3`，而是`5 * 30 * (-3) * 4 * (-3)`，故结论是当前位置的最优解未必是由前一个位置的最优解转移得到**
    - 正确思路
      - **根据数字正负性分类讨论**：
        - 考虑若当前位置是**负数**，则希望前一个位置的乘积也是负数，这样负负得正，并且前一段乘积负数尽可能小，此位置相乘后乘积才会尽可能大。
        - 考虑若当前位置是**正数**，则希望前一个位置的成绩也是正数，并且尽可能地大，这样相乘后才会更大
      - **状态定义**
        - $f_{max}(i)$表示以第`i`个元素结尾的乘积最大子数组的乘积
        - $f_{min}{(i)}$表示以第`i`个元素结尾的乘积最小子数组的乘积
      - **状态转移方程**

$$f_{max}(i) = max_{i=1}^{n}(f_{max}(i-1) * a_i, f_{min}(i-1) * a_i, a_i)$$
$$f_{min}(i) = min_{i=1}^{n}(f_{max}(i-1) * a_i, f_{min}(i-1) * a_i, a_i)$$

      - **初始值**：$f*{max}(0) = nums[0], f*{min}(0) = nums[0]$
      - **返回值**：返回乘积最大子数组中的最大值，$max(f_{max})$，
  - 代码

```
// https://leetcode-cn.com/problems/maximum-product-subarray/solution/cheng-ji-zui-da-zi-shu-zu-by-leetcode-solution/
// 动态规划
class Solution {
public:
    int maxProduct(vector<int>& nums) {
        vector<int> maxF(nums), minF(nums);
        for (int i = 1; i < nums.size(); ++i) {
            maxF[i] = max(maxF[i - 1] * nums[i], max(minF[i - 1] * nums[i], nums[i]));
            minF[i] = min(minF[i - 1] * nums[i], min(maxF[i - 1] * nums[i], nums[i]));
        }
        return *max_element(maxF.begin(), maxF.end());
    }
};
```

  - 复杂度
    - 时间复杂度：$O(n)$
    - 空间复杂度：$O(n)$
- 动态规划-空间优化
  - 思路：由于第`i`个状态只和第`i-1`个状态相关，根据 **滚动数组**思想，可以只用两个变量来维护`i-1`时刻的状态$f*{max}, f*{min}$
  - 代码

```
// https://leetcode-cn.com/problems/maximum-product-subarray/solution/cheng-ji-zui-da-zi-shu-zu-by-leetcode-solution/
// 动态规划-空间优化
class Solution {
public:
    int maxProduct(vector<int>& nums) {
        int maxF = nums[0], minF = nums[0], res = nums[0];
        for (int i = 1; i < nums.size(); ++i) {
            int preMaxF = maxF, preMinF = minF;
            maxF = max(preMaxF * nums[i], max(preMinF * nums[i], nums[i]));
            minF = min(preMinF * nums[i], min(preMaxF * nums[i], nums[i]));
            res = max(maxF, res);
        }
```

```
        return res;
    }
};
```

- 复杂度
  - 时间复杂度：$O(n)$
  - 空间复杂度：$O(1)$

198. 打家劫舍

> 你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。
>
> 给定一个代表每个房屋存放金额的非负整数数组，计算你 不触动警报装置的情况下 ，一夜之内能够偷窃到的最高金额。
>
> 示例 1：
>
> 输入：[1,2,3,1]
> 输出：4
> 解释：偷窃 1 号房屋 (金额 = 1) ，然后偷窃 3 号房屋 (金额 = 3)。
> 偷窃到的最高金额 = 1 + 3 = 4 。
> 示例 2：
>
> 输入：[2,7,9,3,1]
> 输出：12
> 解释：偷窃 1 号房屋 (金额 = 2), 偷窃 3 号房屋 (金额 = 9)，接着偷窃 5 号房屋 (金额 = 1)。
> 偷窃到的最高金额 = 2 + 9 + 1 = 12 。

- 动态规划
  - 思路
    - **状态定义**：`dp[i]`表示前`i`间房屋偷窃到的最大总金额
    - **状态转移方程**
      - 假如只有一间房屋，则偷窃该房屋，即最高总金额
      - 假如只有两间房屋，由于两间房屋相邻，不可同时偷窃，则需选择金额较大的偷窃，得到最高总金额
      - 假如房屋数目大于两间，那么对于第$k, k > 2$间房屋，则有两种偷窃方案
        - 偷窃第k间，那么就不能偷窃第$k - 1$间，偷窃总金额为前$k - 2$间的最高总金额加上第k间房屋的金额
        - 不偷窃第k间，偷窃总金额为前$k-1$间房屋的最高总金额

      则第k间房屋偷窃总金额就是上述两个方案的较大值，方程如下

      $$dp[i] = \max\left(dp[i-2] + nums[i], dp[i-1]\right)$$

    - **初始值**

      $$\begin{cases} dp[0] = nums[0], & \text{只有一间房屋，则偷窃该房屋} \\ dp[1] = max(nums[0], nums[1]), & \text{只有两间房屋，选择较大值} \end{cases}$$

    - **返回值**：最终答案就是第n间房屋的偷窃总金额，即$dp[n-1]$
  - 代码

```
// https://leetcode-cn.com/problems/house-robber/solution/da-jia-jie-she-by-leetcode-solution/
// 动态规划
class Solution {
public:
    int rob(vector<int>& nums) {
        if (nums.empty()) {
            return 0;
        }
        if (nums.size() == 1) {
            return nums[0];
        }

        vector<int> dp(nums.size(), 0);
        dp[0] = nums[0];
        dp[1] = max(nums[0], nums[1]);
        for (int i = 2; i < nums.size(); ++i) {
            dp[i] = max(dp[i-2] + nums[i], dp[i-1]);
        }
        return dp[nums.size() - 1];
    }
};
```

  - 复杂度
    - 时间复杂度：$O(n)$
    - 空间复杂度：$O(n)$

- 动态规划-空间优化
  - 思路：由于第 `i` 状态只和第 `i-1` 和第 `i-2` 个状态相关，根据 **滚动数组**思想，可以只用两个变量来维护第 `i-1` 和第 `i-2` 个房屋的偷盗总金额
  - 代码

```
// https://leetcode-cn.com/problems/house-robber/solution/da-jia-jie-she-by-leetcode-solution/
// 动态规划-空间优化
class Solution {
public:
    int rob(vector<int>& nums) {
        if (nums.empty()) {
            return 0;
        }
        if (nums.size() == 1) {
            return nums[0];
        }

        int first = nums[0], second = max(nums[0], nums[1]);
        for (int i = 2; i < nums.size(); ++i) {
            int third = max(first + nums[i], second);
            first = second;
            second = third;
        }
        return second;
    }
};
```

  - 复杂度
    - 时间复杂度：$O(n)$
    - 空间复杂度：$O(1)$

213. 打家劫舍 II

> 你是一个专业的小偷，计划偷窃沿街的房屋，每间房内都藏有一定的现金。这个地方所有的房屋都 围成一圈 ，这意味着第一个房屋和最后一个房屋是紧挨着的。同时，相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警 。
>
> 给定一个代表每个房屋存放金额的非负整数数组，计算你 在不触动警报装置的情况下 ，今晚能够偷窃到的最高金额。
>
> 示例 1：
>
> 输入：nums = [2,3,2]
> 输出：3
> 解释：你不能先偷窃 1 号房屋（金额 = 2），然后偷窃 3 号房屋（金额 = 2），因为他们是相邻的。
> 示例 2：
>
> 输入：nums = [1,2,3,1]
> 输出：4
> 解释：你可以先偷窃 1 号房屋（金额 = 1），然后偷窃 3 号房屋（金额 = 3）。
> 偷窃到的最高金额 = 1 + 3 = 4 。
> 示例 3：
>
> 输入：nums = [0]
> 输出：0

- 动态规划-空间优化版（基于198题噢）
  - 思路
    - 分析
      - 本题与上题198不同处在于：本题房屋首尾相连，故第一间房屋和最后一间房屋不能同时偷窃
      - 假设只有一间房屋，则偷窃该房屋，得到最高总金额
      - 假设只有两间房屋，则选择其中金额较高的房屋偷窃，得到最高总金额
      - 注意，以上是房屋数目 **不超过**两间时，可不考虑首尾相连问题
      - 假设房屋数目超过2间，则需要考虑首尾相连问题。第一件房屋和最后一间房屋不能同时偷窃；假设数组 `nums` 长度为 `n`
        - 如果偷窃第一间房屋，则可偷窃房屋范围是$[0, n-2]$
        - 如果偷窃最后一间房屋，则可偷窃房屋范围是$[1, n - 1]$
      - 确定了以上可偷窃房屋范围后，则可用 **198题**方法解决，对于两端下标范围，分别计算可以偷窃到的最高总金额，其中的最大值即为在 `n` 间房屋中可偷窃得到的最高总金额
    - 算法
      - 假设可偷窃房屋的下标范围是$[start, end]$
      - **状态定义**：$dp[i]$表示在下表范围内$[start, i]$内可偷窃到的最高总金额
      - **状态转移方程**：计算得到$dp[end]$即为下标范围$[start, end]$内可偷窃到的最高总金额

$$dp[i] = \max(dp[i-2] + nums[i], dp[i-1])$$

- **初始值**

$$\begin{cases} dp[start] = nums[start] & \text{只有一间房屋时，则偷窃该房屋} \\ dp[start+1] = \max(nums[start], nums[start+1]) & \text{只有两间房屋时，则偷窃金额较高的房屋} \end{cases}$$

- **返回值**：分别取$(start, end) = (0, n - 2) \text{和} (start, end) = (1, n - 1)$进行计算，得到$dp[n - 2] \text{和} dp[n - 1]$，返回较大值，为最终结果

- 代码

```
// https://leetcode-cn.com/problems/house-robber-ii/solution/da-jia-jie-she-ii-by-leetcode-solution-bwja/
// 动态规划-空间优化
class Solution {
public:
    int rob(vector<int>& nums) {
        int n = nums.size();
        if (n == 0) {
            return 0;
        }
        if (n == 1) {
            return nums[0];
        }
        if (n == 2) {
            return max(nums[0], nums[1]);
        }

        return max(robRange(nums, 0, n - 2), robRange(nums, 1, n - 1));
    }

    int robRange(vector<int> &nums, int start, int end) {
        int first = nums[start], second = max(nums[start], nums[start + 1]);
        for (int i = start + 2; i <= end; ++i) {
            int third = max(first + nums[i], second);
            first = second;
            second = third;
        }
        return second;
    }
};
```

- 复杂度
  - 时间复杂度：$O(n)$
  - 空间复杂度：$O(1)$

[337. 打家劫舍 III]

> 在上次打劫完一条街道之后和一圈房屋后，小偷又发现了一个新的可行窃的地区。这个地区只有一个入口，我们称之为“根”。 除了“根”之外，每栋房子有且只有一个“父”房子与之相连。一番侦察之后，聪明的小偷意识到“这个地方的所有房屋的排列类似于一棵二叉树”。 如果两个直接相连的房子在同一天晚上被打劫，房屋将自动报警。
>
> 计算在不触动警报的情况下，小偷一晚能够盗取的最高金额。
>
> 示例 1:
>
> 输入: [3,2,3,null,3,null,1]
> 3
> / \
> 2 3
> \ \
> 3 1
>
> 输出: 7
> 解释: 小偷一晚能够盗取的最高金额 = 3 + 3 + 1 = 7.
> 示例 2:
>
> 输入: [3,4,5,1,3,null,1]
> 3
> / \
> 4 5
> / \ \
> 1 3 1
>
> 输出: 9
> 解释: 小偷一晚能够盗取的最高金额 = 4 + 5 = 9

- 动态规划

- 思路
  - 简化下问题：一棵二叉树，树上每个节点都有对应权值，每个节点都有两种状态（选或不选），问在不能同时选中有父子关系的节点的情况下，能选中的点的最大权值和是多少？假设`l, r`表示`o`节点的左右孩子
  - `f(o)`：表示选择`o`节点的情况下，`o`节点的子树上被选择的节点的最大权值和；当`o`被选中时，其左右孩子都不能被选中，故最大权值和为`l, r`不被选中的最大权值和相加，即`f(o) = g(l) + g(r)`
  - `g(o)`：表示不选择`o`节点的情况下，`o`节点的子树上被选择的节点的最大权值和；当`o`不被选中时，`o`的左右孩子可被选中，也可不被选中。故`o`节点的最大权值和为选中其孩子节点和不选中其孩子节点的最大值，即`g(o) = max(f(l), g(l)) + max(f(r), g(r))`
  - 至此，可使用哈希表存储`f`和`g`的函数值，用深搜后序遍历二叉树，可得到每个节点的f和g，根节点的f和g就是答案
- 代码

```
// https://leetcode-cn.com/problems/house-robber-iii/solution/da-jia-jie-she-iii-by-leetcode-solution/
// 深搜
class Solution {
public:
    unordered_map<TreeNode*, int> f, g;
    void dfs(TreeNode *node) {
        if (node == nullptr) {
            return;
        }
        dfs(node->left);
        dfs(node->right);
        f[node] = node->val + g[node->left] + g[node->right];
        g[node] = max(f[node->left], g[node->left]) + max(f[node->right], g[node->right]);
    }
    int rob(TreeNode* root) {
        dfs(root);
        return max(f[root], g[root]);
    }
};
```

- 复杂度
  - 时间复杂度：$O(n)$
  - 空间复杂度：$O(n)$

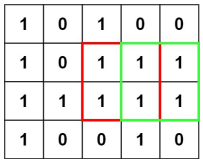
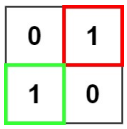
- 动态规划（优化版）
  - 思路

    仔细分析，可发现，无论是`f(o)`还是`g(o)`，都只和`f(l), g(l), f(r), f(r)`有关，所以对每个节点，都只关心其孩子节点的`f`值和`g`值。故可设计一个结构体，表示某个节点的`f`和`g`值，在每次递归返回时，将该节点的结构体返回给上一级调用，可省去哈希表空间
  - 代码

```
// https://leetcode-cn.com/problems/house-robber-iii/solution/da-jia-jie-she-iii-by-leetcode-solution/
// 动态规划优化版
struct Status {
    int selected;
    int unselected;
};
class Solution {
public:
    Status dfs(TreeNode *node) {
        if (node == nullptr) {
            return {0, 0};
        }
        auto l = dfs(node->left);
        auto r = dfs(node->right);
        int selected = node->val + l.unselected + r.unselected;
        int unselected = max(l.selected, l.unselected) + max(r.selected, r.unselected);
        return {selected, unselected};
    }

    int rob(TreeNode* root) {
        auto rs = dfs(root);
        return max(rs.selected, rs.unselected);
    }
};
```

  - 复杂度
    - 时间复杂度：$O(n)$
    - 空间复杂度：$O(n)$，系统栈空间。若不包含系统栈空间，则空间复杂度为$O(1)$

221. 最大正方形

> 在一个由 '0' 和 '1' 组成的二维矩阵内，找到只包含 '1' 的最大正方形，并返回其面积。
>
> 示例 1：
>
> 
>
> 输入：matrix = [["1","0","1","0","0"],["1","0","1","1","1"],["1","1","1","1","1"],["1","0","0","1","0"]]
> 输出：4
> 示例 2：
>
> 
>
> 输入：matrix = [["0","1"],["1","0"]]
> 输出：1
> 示例 3：
>
> 输入：matrix = [["0"]]
> 输出：0

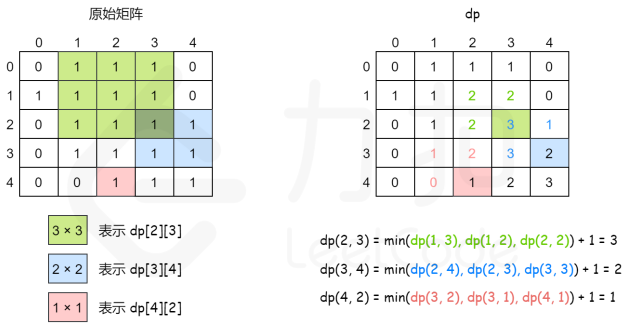
- 动态规划
  - 思路
    - **状态定义**：$dp(i, j)$表示以(i, j)为右下角的最大正方形的边长值。**注意，不是表示这个区域中最大正方形的边长值。**
    - **状态转移方程**：如何计算$dp(i, j)$呢?
      - 如果$matrix(i, j) = 0$，则$dp(i,j) = 0$，因为当前位置无法构成正方形
      - 如果$matrix(i, j) = 1$，则$dp(i, j)$取决于其 **上方、左方、左上方**的三个相邻位置的dp值。具体而言，等于其三个相邻位置的元素的最小值加1

$$dp(i,j) = \min(dp(i-1,j), dp(i-1,j-1), dp(i,j-1)) + 1$$

    - **初始值**

$$\begin{cases} dp(i,0) = 0, & i \in (0, rows)，即最左一列 \\ dp(0,j) = 0, & j \in (0, cols)，即最上一行 \end{cases}$$

    - **返回值**：$dp(i, j) ^2$，存放的是边长，返回的是面积



  - 代码

```cpp
// 参考 https://leetcode-cn.com/problems/maximal-square/solution/zui-da-zheng-fang-xing-by-leetcode-solution/
// 这个也不错 https://leetcode-cn.com/problems/maximal-square/solution/li-jie-san-zhe-qu-zui-xiao-1-by-lzhlyle/
// 动态规划
class Solution {
public:
    int maximalSquare(vector<vector<char>>& matrix) {
        if (matrix.size() == 0 || matrix[0].size() == 0) {
            return 0;
        }
        int rows = matrix.size(), cols = matrix[0].size();
        int maxSide = 0;
        vector<vector<int>> dp(rows, vector<int>(cols, 0));

        for (int i = 0; i < rows; ++i) {
```

```
            for (int j = 0; j < cols; ++j) {
                if (matrix[i][j] == '1') {
                    if (i == 0 || j == 0) {
                        dp[i][j] = 1;
                    } else {
                        dp[i][j] = min(dp[i - 1][j - 1], min(dp[i - 1][j], dp[i][j - 1])) + 1;
                    }
                    maxSide = max(maxSide, dp[i][j]);
                }
            }
        }

        return maxSide * maxSide;
    }
};

// 优化思路参考，然后方法是根据方法一优化的 https://leetcode-cn.com/problems/maximal-square/solution/li-jie-san-zhe-qu-zui-xiao-1-by-lzhlyle/
// 动态规划-空间优化
class Solution {
public:
    int maximalSquare(vector<vector<char>>& matrix) {
        if (matrix.size() == 0 || matrix[0].size() == 0) {
            return 0;
        }
        int rows = matrix.size(), cols = matrix[0].size();
        int maxSide = 0;

        // 计算某行的dp(j)时，所使用的上一行的dp值
        vector<int> dp(cols, 0);
        // 计算某行dp(j)时，所使用的该点(i, j)的左上角(i-1, j-1)值
        int lastRowLeftUpCorner = 0;

        for (int i = 0; i < rows; ++i) {
            // 每行开始时，左上角值为0
            lastRowLeftUpCorner = 0;
            for (int j = 0; j < cols; ++j) {
                // 存放当前dp[j]，用于计算dp(j+1)时的左上角值
                int lastRowLeft = dp[j];
                if (matrix[i][j] == '1') {
                    if (i == 0 || j == 0) {
                        dp[j] = 1;
                    } else {
                        // 更新当前dp[j]
                        dp[j] = min(lastRowLeftUpCorner, min(dp[j - 1], dp[j])) + 1;
                    }
                    maxSide = max(maxSide, dp[j]);
                } else {
                    dp[j] = 0;
                }
                // 将未更新前的dp[j]保存，用于计算dp[j+1]做左上角的值
                lastRowLeftUpCorner = lastRowLeft;
            }
        }

        return maxSide * maxSide;
    }
};
```

- 复杂度
  - 时间复杂度：$O(mn)$
  - 空间复杂度：$O(mn)$，第二种空间优化方法空间复杂度为$O(n)$

264. 丑数 II

> 给你一个整数 n，请你找出并返回第 n 个 丑数 。
>
> 丑数 就是只包含质因数 2、3 和/或 5 的正整数。
>
> 示例 1：
>
> 输入：n = 10
> 输出：12
> 解释：[1, 2, 3, 4, 5, 6, 8, 9, 10, 12] 是由前 10 个丑数组成的序列。
> 示例 2：
>
> 输入：n = 1
> 输出：1
> 解释：1 通常被视为丑数。

- 动态规划
  - 思路
    - **状态定义**：$dp[i]$表示第i个丑数
    - **状态转移方程**
      - 最小的丑数是1，因此$dp[1] = 1$
      - 如何得到其他的丑数呢?
        - 定义三个指针p_2, p_3, p_5，表示下一个丑数是当前指针指向的丑数乘以对应的质因数。
        - 初始时，三个指针的值都是1
        - 当$2 \leq i \leq n$时，令$dp[i] = min(dp[p_2] \times 2, dp[p_3] \times 3, dp[p_5] \times 5) $，然后分别比较$dp[i]$和$dp[p_2], dp[p_3], dp[p_5]$是否相等，如果相等则将对应的指针加$1$

$$dp[i] = \begin{cases} 1 & i = 1 \\ min(dp[p_2] \times 2, dp[p_3] \times 3, dp[p_5] \times 5) & 2 \leq i \leq n \end{cases}$$

    - **初始值**：$dp[1] = 1, p_2 = 1, p_3 = 1, p_5 = 1$
  - 代码

```
// https://leetcode-cn.com/problems/ugly-number-ii/solution/chou-shu-ii-by-leetcode-solution-uoqd/
// 动态规划
class Solution {
public:
    int nthUglyNumber(int n) {
        vector<int> dp(n + 1);
        dp[1] = 1;
        int p2 = 1, p3 = 1, p5 = 1;

        for (int i = 2; i <= n; ++i) {
            int num2 = dp[p2] * 2, num3 = dp[p3] * 3, num5 = dp[p5] * 5;
            dp[i] = min(num2, min(num3, num5));
            // 注意后面的比较，必须3个都比较，不能使用if else if else，要保证dp丑数数组中没有重复丑数，因为可能存在num2 == num3 == num5，即算出来的3个值或两个值相等的情况
            if (dp[i] == num2) {
                ++p2;
            }
            if (dp[i] == num3) {
                ++p3;
            }
            if (dp[i] == num5) {
                ++p5;
            }
        }
        return dp[n];
    }
};
```

    - 复杂度
      - 时间复杂度：$O(n)$
      - 空间复杂度：$O(n)$

279. 完全平方数

> 给定正整数 n，找到若干个完全平方数（比如 1, 4, 9, 16, ...）使得它们的和等于 n。你需要让组成和的完全平方数的个数最少。
>
> 给你一个整数 n ，返回和为 n 的完全平方数的 最少数量 。
>
> 完全平方数 是一个整数，其值等于另一个整数的平方；换句话说，其值等于一个整数自乘的积。例如，1、4、9 和 16 都是完全平方数，而 3 和 11 不是。
>
> 示例 1：
>
> 输入：n = 12
> 输出：3
> 解释：12 = 4 + 4 + 4
> 示例 2：
>
> 输入：n = 13
> 输出：2
> 解释：13 = 4 + 9

- 动态规划
  - 思路
    - **状态定义**：$dp[i]$表示组成正整数i所需的最少的完全平方数的个数
    - **状态转移方程**：遍历数组，每次都将$dp[i] = i$，表示最差情况都由1组成。然后对小于i的每个数j，将$dp[i]$由$dp[i - j * j]$来转化而来

$$dp[i] = \min(dp[i], dp[i - j * j] + 1), 0 \leq i \leq n, 0 \leq j \leq i$$

- **初始值**：$dp[i] = i, 0 \leq i \leq n$
- **返回值**：$dp[n]$

- 代码

```
// https://leetcode-cn.com/problems/perfect-squares/solution/hua-jie-suan-fa-279-wan-quan-ping-fang-shu-by-guan/
// 动态规划
class Solution {
public:
    int numSquares(int n) {
        vector<int> dp(n + 1);
        dp[0] = 0;
        for (int i = 1; i <= n; ++i) {
            // 最坏情况是都是1组成的
            dp[i] = i;
            for (int j = 1; j * j <= i; ++j) {
                dp[i] = min(dp[i], dp[i - j * j] + 1);
            }
        }

        return dp[n];
    }
};
```

- 复杂度
  - 时间复杂度：$O(n \sqrt n)$
  - 空间复杂度：$O(n)$

300. 最长递增子序列

> 给你一个整数数组 nums ，找到其中最长严格递增子序列的长度。
>
> 子序列是由数组派生而来的序列，删除（或不删除）数组中的元素而不改变其余元素的顺序。例如，[3,6,2,7] 是数组 [0,3,1,6,2,2,7] 的子序列。
>
> 示例 1：
>
> 输入：nums = [10,9,2,5,3,7,101,18]
> 输出：4
> 解释：最长递增子序列是 [2,3,7,101]，因此长度为 4 。
> 示例 2：
>
> 输入：nums = [0,1,0,3,2,3]
> 输出：4
> 示例 3：
>
> 输入：nums = [7,7,7,7,7,7,7]
> 输出：1
>
> 提示：
>
> 1 <= nums.length <= 2500
> -104 <= nums[i] <= 104

- 动态规划
  - 思路
    - **状态定义**：$dp[i]$表示以$nums[i]$为尾的最长子序列长度
    - **状态转移方程**：设$j \in [0, i]$，考虑每轮计算新$dp[i]$时，遍历$[0, i)$列表区间，做如下操作
      - 如果$nums[i] > nums[j]$，表示递增，则$dp[i] = dp[j] + 1$
      - 如果$nums[i] < nums[j]$，表示非递增，则$dp[i] = 1$

      $$dp[i] = max(dp[i], dp[j] + 1) j \in [0, i)$$

    - **初始状态**：$dp[i] = 1$，表示每个元素都至少是一个递增子序列，长度为1
    - **返回值**：返回dp列表最大值，即是全局最长上升子序列长度
  - 代码

```
// https://leetcode-cn.com/problems/longest-increasing-subsequence/solution/zui-chang-shang-sheng-zi-xu-lie-dong-tai-gui-hua-2/
// 动态规划
class Solution {
public:
    int lengthOfLIS(vector<int>& nums) {
        int n = nums.size();
```

```
        int res = 0;
        vector<int> dp(n, 1);
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < i; ++j) {
                if (nums[j] < nums[i]) {
                    dp[i] = max(dp[i], dp[j] + 1);
                }
            }
            res = max(res, dp[i]);
        }
        return res;
    }
};
```

- 复杂度
  - 时间复杂度：$O(n^2)$
  - 空间复杂度：$O(n)$

- 动态规划+二分查找
  - 思路
    - **降低复杂度切入点**：解法一中，遍历计算dp列表需要$O(n)$，计算每个$dp[i]$又需要$O(n)$
      - 动态规划中，通过线性遍历来计算dp的复杂度无法降低
      - 每轮计算中，需要通过线性遍历$[0, i)$区间元素来得到$dp[i]$。那么，是否可以重新设计 **状态定义**，使整个dp变为一个 **排序列表**，这样再计算每个$dp[i]$时，就可以通过二分法遍历$[0, k)$区间元素，将此部分复杂度由$O(n)$变为$O(n \log n)$
    - **状态定义**
      - 假设存在常量数字N，和随机数字x，那么很明显，当N越小时，$N < x$的几率越大。比如$N = 0$肯定比$N = 1000$，更可能满足$N < x$
      - 遍历计算$dp[k]$，k表示最长上升子序列的长度为k，$dp[k]$表示长度为k的最长上升子序列的结尾元素值。遍历计算且更新$dp[k]$，始终保持结尾元素值最小。比如数组$[1, 5, 3]$，遍历到元素5时，长度为2的子序列尾部元素值为5；当遍历到元素3时，尾部元素值更新为3
  - 算法
    - **状态定义**：$dp[k]$表示最长上升子序列长度为k时的序列结尾元素值
    - **状态转移方程**：设res表示dp数组当前航都，表示当前最长上升子序列的长度。设$j \in [0, res)$，考虑每遍历$nums[k]$时，通过二分法比那里$[0, res)$区间，找出$nums[k]$的大小分界点，会出现两种情况
      - **区间中存在dp[i] > nums[k]**：对第一个满足$dp[i] > nums[k]$执行$dp[i] = nums[k]$，因为更小的$nums[k]$后更有利于使得上升子序列的长度更长
      - **区间中不存在dp[i] > nums[k]**：表示最长上升子序列长度增加$res + 1$，$nums[k]$成为长度为$res+1$的子序列的结尾元素
    - **初始状态**：$dp[] = 0$
    - **返回值**：res，即最长上升子序列长度
  - 代码

```
// https://leetcode-cn.com/problems/longest-increasing-subsequence/solution/zui-chang-shang-sheng-zi-xu-lie-dong-tai-gui-hua-2/
// 动态规划+二分查找
class Solution {
public:
    int lengthOfLIS(vector<int>& nums) {
        int n = nums.size();
        int res = 0;
        vector<int> dp(n, 1);
        for (int num : nums) {
            int i = 0, j = res;
            while (i < j) {
                int m = (i + j) / 2;
                if (dp[m] < num) {
                    i = m + 1;
                } else {
                    j = m;
                }
            }
            dp[i] = num;
            if (res == j) {
                ++res;
            }
        }
        return res;
    }
};
```

  - 复杂度

    - 时间复杂度：$O(n \log n)$
    - 空间复杂度：$O(n)$

303. 区域和检索 - 数组不可变

> 给定一个整数数组 nums，求出数组从索引 i 到 j（i ≤ j）范围内元素的总和，包含 i、j 两点。
>
> 实现 NumArray 类：
>
> NumArray(int[] nums) 使用数组 nums 初始化对象
> int sumRange(int i, int j) 返回数组 nums 从索引 i 到 j（i ≤ j）范围内元素的总和，包含 i、j 两点（也就是 sum(nums[i], nums[i + 1], ... , nums[j])）
>
> 示例：
>
> 输入：
> ["NumArray", "sumRange", "sumRange", "sumRange"]
> [[[-2, 0, 3, -5, 2, -1]], [0, 2], [2, 5], [0, 5]]
> 输出：
> [null, 1, -1, -3]
>
> 解释：
> NumArray numArray = new NumArray([-2, 0, 3, -5, 2, -1]);
> numArray.sumRange(0, 2); // return 1 ((-2) + 0 + 3)
> numArray.sumRange(2, 5); // return -1 (3 + (-5) + 2 + (-1))
> numArray.sumRange(0, 5); // return -3 ((-2) + 0 + 3 + (-5) + 2 + (-1))

- 前缀和
    - 思路
        - 朴素想法：存储数组$nums$值，每次循环计算(i, j)范围内的元素和
        - 降低复杂度：因为会多次检索，每次检索都是$O(n)$；那么最理想的就是降低检索复杂度为$O(1)$
        - 方法：则可以在初始化的时候，预处理，计算得到 **前缀和prefixSum**，注意$sumRange(i, j)$计算如下

$$
\begin{aligned}
sumRange(i, j) &= \sum_{k=i}^{j} nums[k] \\
&= \sum_{k=i}^{j} nums[k] - \sum_{k=i}^{j} nums[k] \\
&= prefixSum[j+1] - prefixSum[i]
\end{aligned}
$$

    - 代码

```cpp
// https://leetcode-cn.com/problems/range-sum-query-immutable/solution/qu-yu-he-jian-suo-shu-zu-bu-ke-bian-by-l-px41/
// 前缀和
class NumArray {
public:
    vector<int> prefixSum;
    NumArray(vector<int>& nums) {
        int n = nums.size();
        prefixSum.resize(n + 1);
        for (int i = 0; i < n; ++i) {
            prefixSum[i + 1] = prefixSum[i] + nums[i];
        }
    }

    int sumRange(int left, int right) {
        return prefixSum[right + 1] - prefixSum[left];
    }
};
```

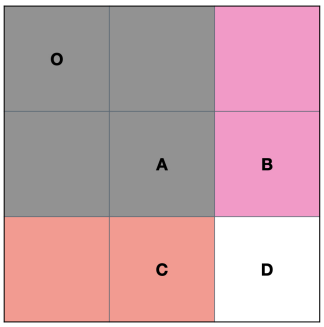
    - 复杂度
        - 时间复杂度：$O(n)$
        - 空间复杂度：$O(n)$

304. 二维区域和检索 - 矩阵不可变

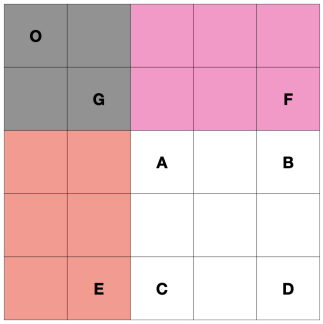
> 给定一个二维矩阵，计算其子矩形范围内元素的总和，该子矩阵的左上角为 (row1, col1) ，右下角为 (row2, col2) 。
>
> 上图子矩阵左上角 (row1, col1) = (2, 1) ，右下角(row2, col2) = (4, 3)，该子矩形内元素的总和为 8。
>
> 示例：
>
> 给定 matrix = [
> [3, 0, 1, 4, 2],
> [5, 6, 3, 2, 1],
> [1, 2, 0, 1, 5],

> [4, 1, 0, 1, 7],
> [1, 0, 3, 0, 5]
>]
>
> sumRegion(2, 1, 4, 3) -> 8
> sumRegion(1, 1, 2, 2) -> 11
> sumRegion(1, 2, 2, 4) -> 12

- 二维前缀和
  - 思路
    - **计算二维前缀和prefixSum**
      - 定义$prefixSum[i][j]$，表示从$[0, 0]$位置到$[i, j]$位置的子矩形的所有元素之和，如下图



      - $S{OD} = S{OC} + S{OB} - S{OA} + S_{D}$，转换成递推公式就是
      - $$prefisSum[i][j] = prefixSum[i-1][j] + prefixSum[i][j-1] - prefixSum[i-1][j-1] + matrix[i][j]$$
    - **根据prefixSum计算矩形面积**
      - 前面计算了从$[0, 0] \rightarrow [i j]$位置的$prefixSum$，下面就可利用$prefixSum[i][j]$快速求出任意子矩形的面积，如下图



      - 子矩形面积：

$$S_{AD} = S_{OD} - S_{OF} - S_{OE} + S_{OG} \rightarrow$$

      - 转换成递推公式：求$[row1, col1] \rightarrow [row2, col2]$的子矩形的面积

$$S = prefixSum[row2][col2] - prefixSum[row2][col1-1] - prefixSum[row1-1][j] + prefixSum[row1-1][col1$$

    - **注意，代码实现中，为了简便，使用的$prefixSum$矩阵在最上一行、最左一列补充了一行一列，为了让第0行和第一列的元素也能使用上面的递推公式。否则需要对第0行和第0列特殊判断**
  - 代码

```
// https://leetcode-cn.com/problems/range-sum-query-2d-immutable/solution/ru-he-qiu-er-wei-de-qian-zhui-he-yi-ji-y-6c21/
// 二维前缀和
class NumMatrix {
public:
    vector<vector<int>> prefixSum;
    NumMatrix(vector<vector<int>>& matrix) {
        if (matrix.size() == 0 || matrix[0].size() == 0) {
            return;
        }
        int m = matrix.size(), n = matrix[0].size();
        prefixSum.resize(m + 1, vector<int>(n + 1, 0));

        for (int i = 0; i < m; ++i) {
```

```
            for (int j = 0; j < n; ++j) {
                prefixSum[i + 1][j + 1] = prefixSum[i][j + 1] + prefixSum[i + 1][j] - prefixSum[i][j]
+ matrix[i][j];
            }
        }
    }

    int sumRegion(int row1, int col1, int row2, int col2) {
        return prefixSum[row2 + 1][col2 + 1] - prefixSum[row1][col2 + 1] - prefixSum[row2 + 1][col1]
+ prefixSum[row1][col1];
    }
};
```

- 复杂度
  - 时间复杂度：构造前缀和$prefixSum$的时间复杂度为$O(mn)$，求$sumRange$函数的时间复杂度为$O(1)$
  - 空间复杂度：$O(mn)$

309. 最佳买卖股票时机含冷冻期

> 给定一个整数数组，其中第 i 个元素代表了第 i 天的股票价格 。
>
> 设计一个算法计算出最大利润。在满足以下约束条件下，你可以尽可能地完成更多的交易（多次买卖一支股票）：
>
> 你不能同时参与多笔交易（你必须在再次购买前出售掉之前的股票）。
> 卖出股票后，你无法在第二天买入股票 (即冷冻期为 1 天)。
> 示例:
>
> 输入: [1,2,3,0,2]
> 输出: 3
> 解释: 对应的交易状态为: [买入, 卖出, 冷冻期, 买入, 卖出]

- 动态规划
  - 思路
    - **状态定义**：**$f[i]$表示第i天结束之后的累计最大收益。**根据题目，由于最多同时买入（持有）一支股票，并且卖出股票后有冷冻期限制，故有三种状态
      - **$dp[i][0]$**：表示第i天持有股票时的累计最大收益
      - **$dp[i][1]$**：表示第i天未持有股票，且是冷冻期时的累计最大收益
      - **$dp[i][2]$**：表示第i天未持有股票，并且不处于冷冻期的累计最大收益
    - **状态转移方程**：
      - 如何进行状态转移呢?
        - 在第i天时，可以在不违反规则的前提下进行 买入或 卖出操作，此时第i天的状态会从第$i - 1$天的状态转移而来。
        - 当然也可以不进行任何操作，那么此时第i天的状态就等同于第$i - 1$天的状态
      - 对于$dp[i][0]$，状态是第i天持有股票，可以由两个状态转移而来
        - 第$i - 1$天就持有股票，对应的状态即累计收益为$dp[i - 1][0]$；
        - 第$i - 1$天未持有股票，且不处于冷冻期，股票是在第i天买入的。对应状态即累计收益需要时第$i - 1$天的状态减去股票负收益，为$dp[i - 1][2] - price[i]$
      - 对于$dp[i][1]$，状态是第i天结束后不持有股票，且处于冷冻期，说明在第i天卖出了股票，即第$i - 1$天持有股票。对应状态即累计收益为第$i - 1$天持有股票的累计收益加上卖出股票的正收益，为$f[i - 1][0] + price[i]$
      - 对于$dp[i][2]$，状态是第i天结束后不持有股票，且不处于冷冻期，说明当天没有操作，即第$i - 1$天不持有股票。那第i天的状态可由第$i - 1$天状态转移而来，第$i - 1$天状态可能处于冷冻期，状态为$f[i - 1][1]$，也可能不处于冷冻期，状态位$f[i - 1][2]$。

$$
\begin{aligned}
dp[i][0] &= max(dp[i-1][0], dp[i-1][2] - price[i]) \\
dp[i][1] &= dp[i-1][0] + price[i] \\
dp[i][2] &= max(dp[i-1][1], dp[i-1][2])
\end{aligned}
$$

    - **初始状态** 可将第0天的状态作为边界条件

$$
\begin{aligned}
dp[0][0] &= -price[0] && \text{第}0\text{天，若持有股票，则只能是第}0\text{天买的} \\
dp[0][1] &= 0 && \text{第}0\text{天不持有股票，处于冷冻期(虽然不可能)，状态置为}0 \\
dp[0][2] &= 0 && \text{第}0\text{天不持有股票，不处于冷冻期，状态置为}0
\end{aligned}
$$

    - **返回值** 返回第n天的所有状态最大值，注意最后一天若仍持有股票，是没意义的，故最后比较不持有股票的两种状态即可

$$max(dp[n-1][1], dp[n-1][2])$$

  - 代码

```
// https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock-with-cooldown/solution/zui-jia-
mai-mai-gu-piao-shi-ji-han-leng-dong-qi-4/
// 动态规划
class Solution {
public:
    int maxProfit(vector<int>& prices) {
```

```
        if (prices.size() == 0) { return 0; }
        int n = prices.size();
        vector<vector<int>> dp(n, vector<int>(3, 0));

        dp[0][0] = -prices[0];

        for (int i = 1; i < n; ++i) {
            dp[i][0] = max(dp[i - 1][0], dp[i - 1][2] - prices[i]);
            dp[i][1] = dp[i - 1][0] + prices[i];
            dp[i][2] = max(dp[i - 1][1], dp[i - 1][2]);
        }

        return max(dp[n - 1][1], dp[n - 1][2]);
    }
};

//// https://leetcode-cn.com/problems/best-time-to-buy-and-sell-stock-with-cooldown/solution/zui-jia-mai-mai-gu-piao-shi-ji-han-leng-dong-qi-4/
// 动态规划-空间优化：上面的转移方程中，dp[i][...]只和dp[i - 1][...]相关，与之前的状态都无关，所以只需要存储dp[i-1][...]这3个状态即可，就可以完成dp[i][...]的状态转移
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        if (prices.size() == 0) { return 0; }
        int n = prices.size();


        int dp0 = -prices[0], dp1 = 0, dp2 = 0;

        for (int i = 1; i < n; ++i) {
            dp[i][0] = max(dp[i - 1][0], dp[i - 1][2] - prices[i]);
            dp[i][1] = dp[i - 1][0] + prices[i];
            dp[i][2] = max(dp[i - 1][1], dp[i - 1][2]);
        }

        return max(dp[n - 1][1], dp[n - 1][2]);
    }
};
```

- 复杂度
  - 时间复杂度：$O(n)$
  - 空间复杂度：$O(n)$，空间优化版空间复杂度为$O(1)$

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。
>
> 你可以认为每种硬币的数量是无限的。
>
> 示例 1：
>
> 输入：coins = [1, 2, 5], amount = 11
> 输出：3
> 解释：11 = 5 + 5 + 1
> 示例 2：
>
> 输入：coins = [2], amount = 3
> 输出：-1
> 示例 3：
>
> 输入：coins = [1], amount = 0
> 输出：0
> 示例 4：
>
> 输入：coins = [1], amount = 1
> 输出：1
> 示例 5：
>
> 输入：coins = [1], amount = 2
> 输出：2

- 动态规划
  - 思路
    - **状态定义**：$dp[i]$表示组成金额i所需最少的硬币数量
    - **状态转移方程**：假设在计算$dp[i]$之前，就已经计算好$dp[0] \rightarrow dp[i - 1]$的值，那么转移方程为

$$dp[i] = \min_{j \in [0, n-1]} dp[i - c_j] + 1$$

其中c_j表示第j枚硬币面值，即枚举某枚硬币金额为c_j，那么需要从$i - c_j$这个金额的状态$dp[i - c_j]$转移而来，再加上枚举的这枚硬币。遍历枚举所有面值的硬币，取出其中最小值，加1就是本次状态

    - **初始值**：`dp[0] = 0;`
    - **返回值**：`dp[amount]`
    - **注意**：代码中，判断金额凑不出的小技巧：先初始化DP table各个元素为amount + 1（代表不可能存在的情况），在遍历时如果金额凑不出则不更新，于是若最后结果仍然是amount + 1，则表示金额凑不出
  - 代码

```
// https://leetcode-cn.com/problems/coin-change/solution/322-ling-qian-dui-huan-by-leetcode-solution/
// 动态规划
class Solution {
public:
    int coinChange(vector<int>& coins, int amount) {
        if (coins.empty()) { return -1; }

        int n = coins.size();
        // 初始值为 amount + 1，防止金额凑不出
        vector<int> dp(amount + 1, amount + 1);

        dp[0] = 0;
        for (int i = 1; i <= amount; ++i) {
            for (auto cj : coins) {
                if (cj <= i) {
                    dp[i] = min(dp[i], dp[i - cj] + 1);
                }
            }
        }

        return dp[amount] > amount ? -1 : dp[amount];
    }
};
```

  - 复杂度
    - 时间复杂度：$O(Sn)$，S表示金额大小，n表示面额个数
    - 空间复杂度：$O(S)$

337. 打家劫舍 III

> 在上次打劫完一条街道之后和一圈房屋后，小偷又发现了一个新的可行窃的地区。这个地区只有一个入口，我们称之为“根”。 除了“根”之外，每栋房子有且只有一个“父”房子与之相连。一番侦察之后，聪明的小偷意识到“这个地方的所有房屋的排列类似于一棵二叉树”。 如果两个直接相连的房子在同一天晚上被打劫，房屋将自动报警。
>
> 计算在不触动警报的情况下，小偷一晚能够盗取的最高金额。
>
> 示例 1:
>
> 输入: [3,2,3,null,3,null,1]
> 3
> / \
> 2 3
> \ \
> 3 1
>
> 输出: 7
> 解释: 小偷一晚能够盗取的最高金额 = 3 + 3 + 1 = 7.
> 示例 2:
>
> 输入: [3,4,5,1,3,null,1]
> 3
> / \
> 4 5
> / \ \
> 1 3 1
>
> 输出: 9
> 解释: 小偷一晚能够盗取的最高金额 = 4 + 5 = 9.

- 动态规划
  - 思路
    - 问题简化：一棵二叉树，树上的每个节点都有对应的权值，每个点有两种状态（选中和不选中），问在不能同时选中有父子关系的点的情况下，能选中的点的最大权值和是多少？
    - **状态定义**
      - $f(o)$表示选择o节点时，o节点的子树上被选择的节点的最大权值和
      - $g(o)$表示不选择o节点时，o节点的子树上被选择的节点的最大权值和

        - **状态转移方程**
            - 当o节点被选中时，o的左右孩子都不可被选中，故o节点被选中情况下，其子树上被选中点的最大权值和为l和r不被选中的最大权值和相加，即$f(o) = g(l) + g(r)$
            - 当o不被选中时，o的左右孩子可以被选中，也可以不被选中。对于o的某个具体的孩子x，它对o的贡献是x被选中和不被选中下权值和的较大值，故$g(o) = max{f(l), g(l)} + max{f(r), g(r)}$
        - **返回值**：返回根节点选中和不被选中时的较大值
    - 代码

```
// https://leetcode-cn.com/problems/house-robber-iii/solution/da-jia-jie-she-iii-by-leetcode-solution/
 // 动态规划
class Solution {
public:
    unordered_map<TreeNode *, int> f, g;
    int rob(TreeNode* root) {
        dfs(root);
        return max(f[root], g[root]);
    }

    void dfs(TreeNode *node) {
        if (node == nullptr) {
            return;
        }
        dfs(node->left);
        dfs(node->right);

        f[node] = node->val + g[node->left] + g[node->right];
        g[node] = max(f[node->left], g[node->left]) +max(f[node->right], g[node->right]);
    }
};
```

    - 复杂度
        - 时间复杂度：$O(n)$
        - 空间复杂度：$O(n)$
- 动态规划-优化版
    - 思路

      由上可知，无论是$f(o)$还是$g(o)$，其最终值只和$f(l), g(l), f(r), g(r)$有关系，所以对于每个节点，只需要关心它的孩子节点的f和g值。我们可设计一个节点，表示某节点的f和g值，在每次递归返回的时候，把这个点对应的f和g返回给上级调用，就可以省去哈希表的空间
    - 代码

```
// https://leetcode-cn.com/problems/house-robber-iii/solution/da-jia-jie-she-iii-by-leetcode-solution/
 // 动态规划-优化版
 struct SubtreeStatus {
     int selected;
     int notSelected;
 };

class Solution {
public:
    int rob(TreeNode* root) {
        auto rootStatus = dfs(root);
        return max(rootStatus.selected, rootStatus.notSelected);
    }

    SubtreeStatus dfs(TreeNode *node) {
        if (node == nullptr) {
            return {0, 0};
        }
        auto l = dfs(node->left);
        auto r = dfs(node->right);
        int selected = node->val + l.notSelected + r.notSelected;
        int notSelected = max(l.notSelected, l.selected) + max(r.notSelected, r.selected);
        return {selected, notSelected};
    }
};
```

    - 复杂度
        - 时间复杂度：$O(n)$
        - 空间复杂度：$O(n)$，省去了哈希表空间。但仍然是$O(n)$空间复杂度

338. 比特位计数

> 给定一个非负整数 num。对于 0 ≤ i ≤ num 范围中的每个数字 i ，计算其二进制数中的 1 的数目并将它们作为数组返回。
>
> 示例 1:
>
> 输入: 2
> 输出: [0,1,1]
> 示例 2:
>
> 输入: 5
> 输出: [0,1,1,2,1,2]
> 进阶:
>
> 给出时间复杂度为O(n*sizeof(integer))的解答非常容易。但你可以在线性时间O(n)内用一趟扫描做到吗?
> 要求算法的空间复杂度为O(n)。
> 你能进一步完善解法吗？要求在C++或任何其他语言中不使用任何内置函数（如 C++ 中的 __builtin_popcount）来执行此操作。

- 动态规划
  - 思路：对于所有的数字，只存在两类。
    - 奇数：二进制表示中，奇数一定比前面那个偶数多一个1，并且多的就是最低位的1，所以$bits[x] = bits[\lfloor{\frac{x}{2}\rfloor] + 1$

      > 举例：　　0 = 0　　1 = 1
      > 　　2 = 10　　3 = 11

    - 偶数：二进制表示中，偶数中1的个数一定和除以2之后的那个数一样多，因为最低位是0，除以2就是右移一位，也就是把最后一位0去掉而已，所以1的个数不变，所以$bit[x] = bits[\lfloor\frac{x}{2}\rfloor]$

      > 举例：
      > 2 = 10　　4 = 100　　8 = 1000
      > 3 = 11　　6 = 110　　12 = 1100

    - 上边两种情况，都可以合并为：$bits[x]$的值等于$bits[\lfloor\frac{x}{2}\rfloor]$的值加上$x$除以$2$的余数，由于$\lfloor\frac{x}{2}\rfloor$等于$x >> 1$，$x$除以$2$的余数等价于$x & 1$，因此有$bits[x] = bits[x >> 1] + (x & 1)$
    - 遍历$[1 \rightarrow num]$，计算$bits$，最终答案就是$bits$
  - 代码

```
// 通俗易懂 https://leetcode-cn.com/problems/counting-bits/solution/hen-qing-xi-de-si-lu-by-duadua/
// 规范答案，代码参考https://leetcode-cn.com/problems/counting-bits/solution/bi-te-wei-ji-shu-by-leetcode-solution-0t1i/
// 动态规划
class Solution {
public:
    vector<int> countBits(int num) {
        vector<int> bits(num + 1);
        for (int i = 1; i <= num; ++i) {
            bits[i] = bits[i >> 1] + (i & 1);
        }
        return bits;
    }
};
```

  - 复杂度
    - 时间复杂度：$O(num)$
    - 空间复杂度：$O(1)$

343. 整数拆分

> 给定一个正整数 n，将其拆分为至少两个正整数的和，并使这些整数的乘积最大化。 返回你可以获得的最大乘积。
>
> 示例 1:
>
> 输入: 2
> 输出: 1
> 解释: 2 = 1 + 1, 1 × 1 = 1。
> 示例 2:
>
> 输入: 10
> 输出: 36
> 解释: 10 = 3 + 3 + 4, 3 × 3 × 4 = 36。
> 说明: 你可以假设 n 不小于 2 且不大于 58。

- 动态规划
  - 思路

- 对于正整数n，当$n \geq 2$时，可以拆分成至少两个正整数的和。令k是拆分出的第一个正整数，则剩下的部分是$n - k$，此时$n - k$可以不再拆分，也可以继续拆分成两个正整数的和。由于每个正整数对应的最大乘积取决于比它小的正整数对应的最大乘积，因此可使用动态规划求解
- **状态定义**：创建数组dp，其中$dp[i]$表示将正整数i拆分成至少两个正整数之后，这些正整数的最大乘积。
- **状态转移方程**：当$i \geq 2$时，假设正整数i拆分出的第一个正整数是$j, (1 \leq j \lt i)$，则有以下两种方案
  - 将i拆分成j和$i - j$的和，且$i - j$不再拆分成多个正整数，此时乘积为$j \times (i - j)$
  - 将i拆分成j和$i - j$的和，且$i - j$继续拆分成多个正整数，此时乘积为$j \times dp[i - j]$

  因此，当j固定时，有$dp[i] = max(j \times (i - j), j \times dp[i - j]), j \in [1, i - 1]$。故状态转移方程为

$$dp[i] = \max_{1 \le j < i} \{max(j \times (i - j), j \times dp[i - j])\}$$

- **初始值**：当$i = 0 \text{和} i = 1$时，无法拆分，故值为0

$$\begin{cases} dp[i] = 1, & i = 0 \\ dp[i] = 1, & i = 1 \end{cases}$$

- **返回值**：最终得到$dp[n]$的值就是将正整数n拆分成至少两个正整数的和之后，这些正整数的最大成绩

◦ 代码

```
// https://leetcode-cn.com/problems/integer-break/solution/zheng-shu-chai-fen-by-leetcode-solution/
// 动态规划
class Solution {
public:
    int integerBreak(int n) {
        vector<int> dp(n + 1, 1);
        for (int i = 2; i <= n; ++i) {
            for (int j = 1; j < i; ++j) {
                dp[i] = max(dp[i], max(j * (i - j), j * dp[i - j]));
            }
        }
        return dp[n];
    }
};
```

◦ 复杂度

- 时间复杂度：$O(n^2)$
- 空间复杂度：$O(n)$

- 数学方法
  ◦ 思路
  - **经过少量推算或很显然的，发现可以将该正整数拆分成尽量多的某个特定的数，可使得结果尽量大**
  - 假设要拆分的正整数为n，特定的数为x，那么可将n分为$\frac{n}{x}$项，记乘积为$f(x) = x^{\frac{n}{x}}$
  - 本题的本质是求：$\max \{ f(x), \forall n \in N_+ \}$

  ◦ 数学推导
  - 很显然，$f(x) = x^{\frac{n}{x}} = e^{\frac{n}{x} \ln x} $，由于$g(x) = e^x$单调递增，$n > 0$恒成立，故$h(x) = \frac{1}{x} \ln x$与$f(x)$有相同的单调性
  - 注意到$h'(x) = \frac{1 - \ln x}{x^2}$，令$h'(x) = 0$，得到驻点$x0 = e$。
  - 由于$x < e$时，y' > 0; x > e时，y' < 0$故$x0$为极大值点，由于函数连输，只有唯一极大值点，故$x0$为最大值点。因此$x = e$时，$f(x)$取最大值：$maxf(x) = e^{\frac{n}{e}}$
  - 由于e不是整数，故使用与e最接近的整数代替，2或者3.简单比较$f(2), f(3)$，即$h(2), h(3)$大小，考虑到$\frac{h(3)}{h(2)} = \frac{2\ln3}{3\ln2} = \frac{\ln9}{\ln8} \gt 1$，故$h(3) > h(2)$，因此$f(3) \gt f(2)$
  - 综上所述，$x = 3$

  ◦ 结论
  - 将数字n拆分为尽量多的3，可以保证乘积最大
    - 若$n \mod 3 == 0, 即n = 3k$，则拆分为k个3
    - 若$n \mod 3 == 1, 即 n = 3k + 1 = 3(k - 1) + 2 \times 2$，则拆分成$k - 1$个$3$，$2$个$2$
    - 若$n \mod 3 == 2，即n = 3k + 2$，则拆分成$k$个$3$，$1$个$2$
  - 考虑到边界情况，当$n \leq 3$时无法拆分，故直接讨论
    - 若$n = 2$，只有$2 = 1 + 1$，此时最大值为1
    - 若$n = 3$，只有$3 = 1 + 2$，此时最大值为2
    - 以上两种情形可以合并为：当$n \leq 3$时，最大值为$n - 1$
  - 综上所述

$$max = \begin{cases} n - 1, & n \le 3 \\ 3^k, & n = 3k (k \ge 2) \\ 4 \times 3^{k-1}, & n = 3k + 1 (k > 1) \\ 2 \times 3^k, & n = 3k + 2 (k \ge 1) \end{cases}$$

  ◦ 代码

```
```

```cpp
// https://leetcode-cn.com/problems/integer-break/solution/zheng-shu-chai-fen-shu-xue-fang-fa-han-wan-zheng-t/
// 数学方法
class Solution {
public:
    int integerBreak(int n) {
        if (n <= 3) {
            return n - 1;
        }
        int a = n / 3, b = n % 3;
        if (b == 0) {
            return pow(3, a);
        }
        if (b == 1) {
            return pow(3, a - 1) * 4;
        }
        return pow(3, a) * 2;
    }
};
```

- 复杂度
  - 时间复杂度：$O(1)$
  - 空间复杂度：$O(1)$

357. 计算各个位数不同的数字个数

> 给定一个非负整数 n，计算各位数字都不同的数字 x 的个数，其中 0 ≤ x < 10n 。
>
> 示例:
>
> 输入: 2
> 输出: 91
> 解释: 答案应为除去 11,22,33,44,55,66,77,88,99 外，在 [0,100) 区间内的所有数字。

- 动态规划
  - 思路
    - **状态定义**：$dp[i]$表示i个数字中有多少个数字是重复的，假设$i = 2$，那么它表示的是$10 \rightarrow 99$个数字中重复数字的数目。同理，$i = 3$时，表示的是$100 \rightarrow 999$中重复数字的数目，注意是3位数中的重复数字，不包含2位数的重复数字
    - **状态转移方程**：以根据$dp[2]$计算$dp[3]$为例
      - 首先，$dp[2]$中存在重复数字$11, 22, 33, 44, 55, 66, 77, 88, 99$，这些数字中任何一个数字在最后一位上加数字都是重复数字，因为他们本身就是重复了。最后一位加数字的选择可以使$0 - 9$，故有10中可能。并且已知$dp[2]$中有9个重复数，故可以组成$9 * 10$种3位的重复数，故$dp[i]$的一部分为： $dp[i] = dp[i - 1] * 10$
      - 其次，除了上述重复数之外，对于不属于上述重复数字的其他类型数字，比如$12, 21, 61, 59$等等，在最后一位任意加上前2位出现过的数字都会导致重复数字，比如对于数字12，在后边添加1或2都会导致重复。这样就有x中可能，$x = i - 1$。那么和第一种的重复数字不同的数字有多少呢？共有$pow(10, i - 1) - pow(10, i - 2)$种，第一部分表示$2$位数字共有100个，第二部分表示包含了$1$位数字，即$0 - 9$，需要减去。故第二部分的重复数字个数为$(pow(10, i - 1) - pow(10, i - 2)) \times (i - 1)$
      - 故最后的方程为

$$dp[i] = dp[i-1] \times 10 + (10^{i-1} - 10^{i-2}) \times (i-1)$$

  - 代码

```cpp
// https://leetcode-cn.com/problems/count-numbers-with-unique-digits/solution/cdong-tai-gui-hua-xiang-jie-by-xiaohu952-9qbt/
// 动态规划
class Solution {
public:
    int countNumbersWithUniqueDigits(int n) {
        vector<int> dp(n + 1);
        for (int i = 2; i <= n; ++i) {
            dp[i] = dp[i - 1] * 10 + ((pow(10, i - 1) - pow(10, i - 2) - dp[i - 1]) * (i - 1));
        }
        int same_counts = 0;
        for (auto &x : dp) {
            same_counts += x;
        }
        return pow(10, n) - same_counts;
    }
};
```

  - 复杂度
    - 时间复杂度：$O(n)$

- 空间复杂度：$O(n)$

647. 回文子串

> 给定一个字符串，你的任务是计算这个字符串中有多少个回文子串。
>
> 具有不同开始位置或结束位置的子串，即使是由相同的字符组成，也会被视作不同的子串。
>
> 示例 1：
>
> 输入："abc"
> 输出：3
> 解释：三个回文子串: "a", "b", "c"
> 示例 2：
>
> 输入："aaa"
> 输出：6
> 解释：6个回文子串: "a", "a", "a", "aa", "aa", "aaa"

- 中心扩展法
  - 思路
    - 中心扩展法比较巧妙，思想在于确定中心点后再验证回文串，寻找到所有的中心点。比如对字符串$ababa$，选择最中间的`a`作为中心点，往两边扩散，第一次扩散发现`left`指向的是`b`，`right`指向的也是`b`,所以是回文串。继续扩展，同理`ababa`也是回文串
    - 那么中心点有多少呢？需要注意，中心点可能为`1`个，也可能为`2`个。比如奇数回文串的中心点为一个字符，偶数回文串的中心点就是两个字符。那么中心点个数呢？就是`2 * len - 1`个，分别是`len`个单字符和`len - 1`个双字符
      - 为什么有`2 * len - 1`个中心点？
        - 对于字符串`aba`，有`5`个中心点，分别是`a, b, a, ab, ba`
        - 对于字符串`abba`，有`7`个中心点，分别是`a, b, b, a, ab, bb, ba`
      - 那么什么是中心点呢？即是`left`指针和`right`指针初始化指向的地方，可能是一个或两个
      - 为什么不能是三个字符或更多呢？因为三个字符的中心点可以由一个字符扩展一次得到，四个字符的中心点可以是两个字符的中心点扩展一次得到
      - `left`指针、`right`指针和中心点的关系呢？看代码
  - 代码

```cpp
// https://leetcode-cn.com/problems/palindromic-substrings/solution/liang-dao-hui-wen-zi-chuan-de-jie-fa-xiang-jie-zho/
// 中心扩散法
class Solution {
public:
    int countSubstrings(string s) {
        int res = 0;
        for (int center = 0; center < 2 * s.size() - 1; ++center) {
            // left和right指针和中心点的关系是？
            // 首先是left，有一个很明显的2倍关系的存在，其次是right，可能和left指向同一个（偶数时），也可能往后移动一个（奇数）
            // 大致的关系出来了，可以选择带两个特殊例子进去看看是否满足。
            int left = center / 2;
            int right = left + center % 2;
            while (left >= 0 && right < s.size() && s[left] == s[right]) {
                ++res;
                --left;
                ++right;
            }
        }
        return res;
    }
};
```

  - 复杂度
    - 时间复杂度：$O(n^2)$
    - 空间复杂度：$O(1)$
- 动态规划
  - 思路
    - **状态定义**：$dp[i][j]$表示字符串s在$[i, j]$区间的子串是否是一个字符串
    - **状态转移方程**：分3种情况分析
      - 当子串只有一个字符时，比如a，自然是一个回文串
      - 当子串有两个字符时，如果两个字符相等，比如aa，也是一个回文串

        - 当子串有三个及以上字符时，比如$ababa$，把这个子串记为`串1`；然后把首尾字符都去掉，得到`串2`bab，可以看到串2是一个回文串，有$dp[i + 1][j - 1] = true$；所以状态转移方程如下

$$dp[i][j] = true, s[i] == s[j] \&\& (j - i < 2 || dp[i + 1][j - 1])$$

    - **初始值** $dp[i][i] = true$
- 代码

```
// https://leetcode-cn.com/problems/palindromic-substrings/solution/liang-dao-hui-wen-zi-chuan-de-jie-fa-xiang-jie-zho/
// 动态规划法
class Solution {
public:
    int countSubstrings(string s) {
        vector<vector<bool>> dp(s.size(), vector<bool>(s.size()));
        int res = 0;
        for (int j = 0; j < s.size(); ++j) {
            for (int i = 0; i <= j; ++i) {
                if (s[i] == s[j] && (j - i < 2 || dp[i + 1][j - 1])) {
                    dp[i][j] = true;
                    ++res;
                }
            }
        }
        return res;
    }
};
```

- 复杂度
    - 时间复杂度：$O(n^2)$
    - 空间复杂度：$O(n^2)$

[1143. 最长公共子序列]

> 给定两个字符串 text1 和 text2，返回这两个字符串的最长 公共子序列 的长度。如果不存在 公共子序列 ，返回 0 。
>
> 一个字符串的 子序列 是指这样一个新的字符串：它是由原字符串在不改变字符的相对顺序的情况下删除某些字符（也可以不删除任何字符）后组成的新字符串。
>
> 例如，"ace" 是 "abcde" 的子序列，但 "aec" 不是 "abcde" 的子序列。
> 两个字符串的 公共子序列 是这两个字符串所共同拥有的子序列。
>
> 示例 1：
>
> 输入：text1 = "abcde", text2 = "ace"
> 输出：3
> 解释：最长公共子序列是 "ace" ，它的长度为 3 。
> 示例 2：
>
> 输入：text1 = "abc", text2 = "abc"
> 输出：3
> 解释：最长公共子序列是 "abc" ，它的长度为 3 。
> 示例 3：
>
> 输入：text1 = "abc", text2 = "def"
> 输出：0
> 解释：两个字符串没有公共子序列，返回 0 。

- 动态规划
    - 思路
        - 解题思路：求两个数组或者字符串的最长公共子序列问题，肯定要用动态规划
            - 首先，区分两个概念：子序列可以不连续，子数组（子字符串）需要连续
            - 另外，动态规划套路：单个数组或字符串要用动态规划时，可以把动态规哈`dp[i]`定义为`nums[0:i]`中想要求的结果；当两个数组或者字符串要用动态规划时，可以把动态规划定义成二维的`dp[i][j]`，其含义是在`A[0:i]`和`B[0:j]`之间匹配得到的想要的结果
        - **状态定义**：比如对于本题而言，可以定义成`dp[i][j]`表示为`text1[0:i - 1]`和`text2[0:j-1]`的最长公共子序列。注意，`text[0:i-1]`表示的是`text1`的第`0`个元素和第`i-1`个元素，两端都包含。以`dp[i][j]`的定义不是`text1[0:i]`和`text2[0:j]`，是为了方便当`i = 0`或`j = 0`时，`dp[i][j]`表示的为空字符串和另外一个字符串的匹配，这样`dp[i][j]`可以初始化为0
        - **状态转移方程**
            - 当`text1[i - 1] == text2[j - 1]`时，说明两个子字符串的最后一位相等，所以最长公共子序列长度增加了1，所以`dp[i][j] = dp[i - 1][j - 1] + 1`;比如，对于`ac`和`bc`而言，他们的最长公共子序列的长度等于`a`和`b`的最长公共子序列长度`0 + 1 = 1`

- 当 `text1[i - 1] != text2[j - 1]` 时，说明两个子字符串的最后一位不相等，那么此时的状态 `dp[i][j]` 应该是 `dp[i - 1][j]` 和 `dp[i][j-1]` 的最大值。比如对于 `ace` 和 `bc` 而言，其最长公共子序列长度等于 `ace` 和 `b` 的最长公共子序列长度 `0` 与 `ac` 和 `bc` 的最长公共子序列长度 `1` 的最大值，为 `1`

$$dp[i][j] = \begin{cases} dp[i-1][j-1]+1, & text1[i-1] == text2[j-1] \\ max(dp[i-1][j], dp[i][j-1]), & text1[i-1] != text2[j-1] \end{cases}$$

- **初始化**

$$dp[i][j] = \begin{cases} 0, i = 0 \\ 0, j = 0 \end{cases}$$

- **遍历方向**：由于 `dp[i][j]` 依赖于 `dp[i - 1][j - 1]`、`dp[i - 1][j]`、`dp[i][j - 1]`，所以 `i` 和 `j` 的遍历顺序肯定是从小到大的。另外由于当 `i` 和 `j` 取值为 `0` 时，`dp[i][j] = 0`，而 `dp` 数组本身初始化就是为0，所以直接让 `i` 和 `j` 从 `1` 开始遍历，结束条件是字符串长度 `len(text1), len(text2)`
- **返回值**：`dp[len(text1)][len(text2)]`

- 代码

```
// https://leetcode-cn.com/problems/longest-common-subsequence/solution/fu-xue-ming-zhu-er-wei-dong-tai-gui-hua-r5ez6/
// 动态规划
class Solution {
public:
    int longestCommonSubsequence(string text1, string text2) {
        int M = text1.size(), N = text2.size();
        vector<vector<int>> dp(M + 1, vector<int>(N + 1, 0));
        for (int i = 1; i <= M; ++i) {
            for (int j = 1; j <= N; ++j) {
                if (text1[i - 1] == text2[j - 1]) {
                    dp[i][j] = dp[i - 1][j - 1] + 1;
                } else {
                    dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
                }
            }
        }
        return dp[M][N];
    }
};
```

- 复杂度
  - 时间复杂度：$O(MN)$
  - 空间复杂度：$O(MN)$