dp Tip：倒序遍历可以避免一个物品被多次选择的情况

一背包问题

.01背包问题（一个物品只能选择一次）/完全背包问题，一个物品可以重复使用

```python
def knapsack(weights, values, capacity):
    n = len(weights)
    dp = [0] * (capacity + 1)  # 用一维数组来存储每个背包容量下的最大价值

    # 遍历每个物品
    for i in range(n):
        # 从大到小遍历背包容量
        for w in range(capacity, weights[i] - 1, -1):
            # 更新 dp[w]，即考虑是否选择第 i 个物品
            dp[w] = max(dp[w], dp[w - weights[i]] + values[i])

    return dp[capacity]  # 返回最大背包容量下的最大价值
```

```python
def complete_knapsack(weights, values, capacity):
    n = len(weights)
    dp = [0] * (capacity + 1)  # 初始化 dp 数组，dp[w] 表示容量 w 时的最大价值

    # 遍历每个物品
    for i in range(n):
        for w in range(weights[i], capacity + 1):  # 遍历每个可能的背包容量
            # 如果能放入物品 i，则选择最大值（放或者不放）
            dp[w] = max(dp[w], dp[w - weights[i]] + values[i])

    return dp[capacity]  # 返回背包容量为 capacity 时的最大价值

# 示例
weights = [2, 3, 4, 5]  # 物品的重量
values = [3, 4, 5, 6]   # 物品的价值
capacity = 5            # 背包的容量

print(complete_knapsack(weights, values, capacity))  # 输出最大价值
```

2.多重背包问题

```python
def bounded_knapsack(weights, values, num, capacity):
    n = len(weights)
    dp = [0] * (capacity + 1)  # dp[w] 表示容量 w 的最大价值

    # 对每个物品
    for i in range(n):
        # 对每个背包容量，从大到小遍历
        for w in range(capacity, -1, -1):
            # 对当前物品的可选次数进行枚举
            max_count = min(num[i], w // weights[i])  # 当前物品最多能选多少次
            for k in range(1, max_count + 1):
                dp[w] = max(dp[w], dp[w - k * weights[i]] + k * values[i])

    return dp[capacity]  # 返回最大价值
```

二.字符串问题

1.最长公共子序列：给定两个字符串 `X` 和 `Y`，求它们的最长公共子序列（LCS）。子序列是从原序列中删除一些元素（可以不删除任何元素），但不改变剩下元素的顺序。我们的目标是找到这两个字符串的最长公共子序列的长度。

**子序列的定义**：公共子序列中的元素顺序要保持一致，但可以不连续。每次比较 `X[i-1]` 和 `Y[j-1]` 来决定是否更新公共子序列的长度。

```python
def longestCommonSubsequence(X: str, Y: str) -> int:
    m = len(X)
    n = len(Y)

    # 创建一个 (m+1) x (n+1) 的二维 dp 数组
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    # 填充 dp 数组
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if X[i - 1] == Y[j - 1]:  # 如果字符相等
                dp[i][j] = dp[i - 1][j - 1] + 1
            else:  # 如果字符不相等
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

    # 最长公共子序列的长度是 dp[m][n]
    return dp[m][n]
```

2.（拦截导弹）最长递增子序列问题：给定一个整数数组 `nums`，返回其中最长递增子序列的长度。递增子序列是一个子序列，子序列的元素是严格递增的。

通过维护一个 `dp` 数组，其中 `dp[i]` 表示以 `nums[i]` 为结尾的最长递增子序列的长度。

对于每一个 `nums[i]`，检查前面所有的 `nums[j]`，如果 `nums[i] > nums[j]`，则 `dp[i]` `max(dp[i], dp[j] + 1)`。最终答案为 `dp` 数组中的最大值。

```python
def lengthOfLIS(nums):
    if not nums:
        return 0

    # 初始化 dp 数组，每个位置的 dp[i] 至少为 1（每个元素本身就是一个递增子序列）
    dp = [1] * len(nums)

    # 填充 dp 数组
    for i in range(1, len(nums)):
        for j in range(i):
            if nums[i] > nums[j]:
                dp[i] = max(dp[i], dp[j] + 1)

    # 返回 dp 数组的最大值，即为最长递增子序列的长度
    return max(dp)

# 示例
nums = [10, 9, 2, 5, 3, 7, 101, 18]
print(lengthOfLIS(nums))  # 输出 4
```

3.编辑距离：给定两个字符串，求将一个字符串转换成另一个字符串所需的最小操作次数。操作包括插入、删除和替换。

插入操作：`dp[i][j-1] + 1` 删除操作：`dp[i-1][j] + 1` 替换操作：`dp[i-1][j-1] + 1` 如果字符相同：`dp[i-1][j-1]`

```python
def minDistance(word1, word2):
    m, n = len(word1), len(word2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]
    for i in range(m + 1):
        for j in range(n + 1):
            if i == 0:
                dp[i][j] = j
            elif j == 0:
                dp[i][j] = i
            elif word1[i - 1] == word2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1]
            else:
                dp[i][j] = min(dp[i - 1][j], dp[i][j - 1], dp[i - 1][j - 1]) + 1
    return dp[m][n]
```

4.最大子数组和 5.土豪购物（难，构造了两个数组，一个表示连续子数组最大和，一个表示允许放回一个商品的连续子数组和）

### 5. 最大子数组和（Maximum Subarray Sum）

- **描述**：给定一个整数数组，求出其连续子数组的最大和。
- **核心思路**：使用 DP 数组 `dp[i]` 表示以 `i` 为结尾的最大子数组和。
  - 状态转移：`dp[i] = max(dp[i - 1] + arr[i], arr[i])`
- **时间复杂度**：O(n)

```python
def maxSubArray(nums):
    if not nums:
        return 0
    dp = [0] * len(nums)
    dp[0] = nums[0]
    for i in range(1, len(nums)):
        dp[i] = max(dp[i - 1] + nums[i], nums[i])
    return max(dp)
```

1. `dp1[i]` 表示 **以第 i 个商品结尾的连续子数组最大和**，不考虑放回商品。
   - 状态转移公式：`dp1[i] = max(dp1[i - 1] + a[i], a[i])`
   - 解释：当前商品 `a[i]` 要么加入之前的子数组，要么单独成为一个新的子数组。
2. `dp2[i]` 表示 **以第 i 个商品结尾的连续子数组最大和**，允许放回其中一个商品。
   - 状态转移公式：`dp2[i] = max(dp1[i - 1], dp2[i - 1] + a[i], a[i])`
   - 解释：
     - `dp1[i - 1]` 表示选择前面的子数组，但不加当前商品。
     - `dp2[i - 1] + a[i]` 表示当前商品加入到之前可能已经放回一个商品的子数组。
     - `a[i]` 表示单独选择当前商品。

6.最大切割数

```python
n, a, b, c = map(int, input().split())
# 因为绳子的数量不可能为负数，所以将不可达数据定义为-1
# 随着dp[n]从0开始不断往上推，当遍历到n时，n前面的情况其实已经遍历过了，
# 如果三个if条件都无法满足的话，-1的不可达状态就未发生改变
dp = [-1] * (n + 1)
# 以dp【0】为初始化数据状态，确定了底层数据情况
dp[0] = 0
# 从底层往上遍历
for i in range(n + 1):
    if i >= a and dp[i - a] != -1:
        dp[i] = max(dp[i - a] + 1, dp[i])
    if i >= b and dp[i - b] != -1:
        dp[i] = max(dp[i - b] + 1, dp[i])
    if i >= c and dp[i - c] != -1:
        dp[i] = max(dp[i - c] + 1, dp[i])

print(dp[n])
```

7.将DFS与dp结合起来，用dp储存某点处的最优值

```python
# 滑雪
#返回值型DFS
# 创造一个矩阵表示某个点最多产生的路径数
dir = [(1, 0), (-1, 0), (0, 1), (0, -1)]
m, n = map(int, input().split())
maze = []
dp=[[0 for _ in range(n)] for _ in range(m)]
for _ in range(m):
    maze.append(list(map(int, input().split())))


def is_valid(x, y):  1 个用法
    if 0 <= x < m and 0 <= y < n:
        return True
    return False


def dfs(x, y):  2 用法
    if dp[x][y]!=0:
        return dp[x][y]
    dp[x][y]=1
    for dx, dy in dir:
        if is_valid(x + dx, y + dy):
            if maze[x + dx][y + dy] < maze[x][y]:
                dp[x][y]= max(1+dfs(x+dx,y+dy),dp[x][y])
    return dp[x][y]

zuichang=0
for i in range(m):
    for j in range(n):
        zuichang=max(dfs(i, j),zuichang)
print(zuichang)
```

8.零钱兑换

```python
n, m = map(int, input().split())
for _ in range(n):
    miane = list(map(int, input().split()))
dp = [float('inf') for _ in range(m + 1)]
dp[0] = 0
for i in range(m + 1):
    for o in miane:
        if i>=o:
            dp[i] = min(dp[i], dp[i - o] + 1)
if dp[m]==float('inf'):
    print(-1)
else:
    print(dp[m])
```

9.篮球比赛

定义两个数组 `dp1[i]` 和 `dp2[i]`，其中：

- `dp1[i]` 表示到第 `i` 个学生（即第一行的第 `i` 个学生）时，选出的学生总高度的最大值。
- `dp2[i]` 表示到第 `i` 个学生（即第二行的第 `i` 个学生）时，选出的学生总高度的最大值。

**状态转移方程：**

- 如果我们在第 `i` 个学生（第一行的学生）选择了学生，则可以从第二行的第 `i-1` 个学生转移过来，更新 `dp1[i]`：

$$dp1[i] = \max(dp2[i-1] + h1[i], dp1[i-1])$$

  其中 `dp2[i-1] + h1[i]` 表示从第二行的第 `i-1` 个学生选过来再选第 `i` 个学生的总高度，`dp1[i-1]` 表示不选择第 `i` 个学生而继续选第一行的学生。

- 同理，若选择第 `i` 个学生（第二行的学生），则可以从第一行的第 `i-1` 个学生转移过来，更新 `dp2[i]`：

$$dp2[i] = \max(dp1[i-1] + h2[i], dp2[i-1])$$

  其中 `dp1[i-1] + h2[i]` 表示从第一行的第 `i-1` 个学生选过来再选第 `i` 个学生的总高度，`dp2[i-1]` 表示不选择第 `i` 个学生而继续选第二行的学生。

```python
def main():
    n = int(input())  # 输入学生数量
    h1 = list(map(int, input().split()))  # 第一行学生的身高
    h2 = list(map(int, input().split()))  # 第二行学生的身高

    # dp1[i]表示以第i个学生（第一行）为结尾的最大总身高
    # dp2[i]表示以第i个学生（第二行）为结尾的最大总身高
    dp1 = [0] * n
    dp2 = [0] * n

    # 初始化
    dp1[0] = h1[0]
    dp2[0] = h2[0]

    # 从第2个学生开始进行动态规划
    for i in range(1, n):
        dp1[i] = max(dp2[i - 1] + h1[i], dp1[i - 1])  # 从第二行转到第一行或保持第一行
        dp2[i] = max(dp1[i - 1] + h2[i], dp2[i - 1])  # 从第一行转到第二行或保持第二行

    # 输出最后的最大值
    print(max(dp1[n - 1], dp2[n - 1]))
```

## 二.贪心问题

### 1.最大化不相交区间数量/2.机智的股民老张

```python
n=int(input())
activity=[]
for _ in range(n):
    start,end=map(int,input().split())
    activity.append((start,end))
current_time=0
activity.sort(key=lambda x:(x[1],x[0]))
count=0
for i in activity:
    if current_time<i[0]:
        count+=1
        current_time=i[1]

print(count)
```

```python
def main():  1 个用法
    a=list(map(int,input().split()))
    n=len(a)
    if n<2:
        print(0)
        return
    min_p=a[0]
    max_profit=0
    for i in range(1,n):
        max_profit=max(max_profit,a[i]-min_p)#维护最大利润
        min_p=min(min_p,a[i])#维护i之前最低的股价
    print(max_profit)
    return
main()
```

3.舞会匹配

```
n = int(input())
a = [int(i) for i in input().split()]
m = int(input())
b = [int(i) for i in input().split()]

a.sort()
b.sort()

cnt = 0
i, j = 0, 0

while i < n and j < m:
    if abs(a[i] - b[j]) <= 1:
        # 找到一个匹配，移动两个指针，并增加计数
        cnt += 1
        i += 1
        j += 1
    elif a[i] < b[j]:
        # 如果 a[i] 小于 b[j]，移动 a 的指针
        i += 1
    else:
        # 如果 a[i] 大于 b[j]，移动 b 的指针
        j += 1

print(cnt)
```

三.递归

1汉诺塔

```
moves=[]
def hannuota(n, a, b, c,moves):  3 用法
    if n==1:
        moves.append(f'{a}->{c}')
    else:
        hannuota(n - 1, a,c,b,moves)#将前n-1个数从原柱移到B柱
        moves.append(f'{a}->{c}')#将第n个数从原柱移到目标柱
        hannuota(n - 1, b,a, c,moves)#将前n-1个柱从B柱移到目标柱



n = int(input())
print(2 ** n - 1)
hannuota(n, a: "A", b: "B", c: "C",moves)
print(*moves,sep='\n')

#递归的基本数据情况很重要
#错误笔记，因为没有将n=1的情况记录进moves导致递归出错
```

四.双指针问题

1.回文子串/2，最长无重复子字符串

**题目描述**：给定一个字符串，求该字符串中的所有回文子串的数量。

- **思路**：使用双指针，在每个字符为中心，向两边扩展，检查是否是回文。

```python
def count_substrings(s):
    def expand_around_center(left, right):
        count = 0
        while left >= 0 and right < len(s) and s[left] == s[right]:
            count += 1
            left -= 1
            right += 1
        return count

    total_count = 0
    for i in range(len(s)):
        total_count += expand_around_center(i, i)  # 奇数长度回文
        total_count += expand_around_center(i, i + 1)  # 偶数长度回文
```

```python
def length_of_longest_substring(s):
    # 用于存储窗口内的字符
    char_set = set()
    # 左指针初始位置
    left = 0
    # 记录最长无重复子串的长度
    max_len = 0

    # 右指针从0开始遍历整个字符串
    for right in range(len(s)):
        # 如果右指针指向的字符已经在窗口中，则收缩窗口
        while s[right] in char_set:
            # 移除左指针指向的字符，并将左指针右移
            char_set.remove(s[left])
            left += 1

        # 将右指针指向的字符加入窗口
        char_set.add(s[right])

        # 更新最大子串长度
        max_len = max(max_len, right - left + 1)

    return max_len
```

3.两数之和

## 2. 两数之和（Two Sum）

**题目描述**：给定一个已排序的数组和一个目标值，找出数组中两个数的和等于目标值的下标。

- **思路**：使用两个指针，`left`指向数组的开头，`right`指向数组的末尾。通过调整指针来找到符合条件的两个数。

```python
def two_sum(nums, target):
    left, right = 0, len(nums) - 1
    while left < right:
        current_sum = nums[left] + nums[right]
        if current_sum == target:
            return [left, right]
        elif current_sum < target:
            left += 1
        else:
            right -= 1
    return []
```

4.三数之和（转化为两数之和，但是要注意去除重复的第三位数）

```python
def three_sum(nums):
    # 先排序
    nums.sort()
    result = []

    # 遍历每个元素
    for i in range(len(nums) - 2):
        # 跳过重复的元素
        if i > 0 and nums[i] == nums[i - 1]:
            continue

        # 使用双指针，left指向i之后的位置，right指向数组的最后一个元素
        left, right = i + 1, len(nums) - 1

        # 双指针遍历
        while left < right:
            current_sum = nums[i] + nums[left] + nums[right]

            # 找到和为0的三元组
            if current_sum == 0:
                result.append([nums[i], nums[left], nums[right]])

                # 跳过重复的元素
                while left < right and nums[left] == nums[left + 1]:
                    left += 1
                while left < right and nums[right] == nums[right - 1]:
                    right -= 1

                # 移动指针
                left += 1
                right -= 1

            # 如果当前和小于0，说明需要增大和，left向右移动
            elif current_sum < 0:
                left += 1
            # 如果当前和大于0，说明需要减小和，right向左移动
            else:
                right -= 1

    return result

# 测试
nums = [-1, 0, 1, 2, -1, -4]
print(three_sum(nums))  # 输出 [[-1, -1, 2], [-1, 0, 1]]
```