一、 子数组问题（子数组的和满足某种条件）：

T209：nums 数组元素都是正整数，方法：滑动窗口 （类似题：T1658）

T862：nums 数组元素有正有负，方法：滑动窗口+单调队列+前缀和 （非常综合好题）

（单调队列与滑动窗口经常结合，特别是当数组元素有正有负时，在滑动窗口的过程中要高效地取出窗口内的最值，就要配合单调队列）

T560：和为 k 的子数组：给你一个整数数组 nums 和一个整数 k ，请你统计并返回 *该数组中和为 k 的子数组的个数* 。（重点题）

方法：前缀和+哈希表 （数组元素有正有负，不能用滑动窗口，因为不知道什么时候扩大窗口，什么时候收缩窗口）

T523：连续的子数组和

T918：环形子数组最大和：前缀和+单调队列+滑动窗口，关键在于维护一个滑动窗口（单调队列实现的），以便根据窗口中的最小值计算最大子数组和（前缀和减去窗口最小值得到的就是最大和子数组），之所以要用单调队列而不用优先级队列是因为除了有优先级以外还想保持先进先出的特性，特别是对于数组的长度有要求的题目，如长度不超过 k 之类的，本题因为是环形数组，而且我们通过遍历两倍原数组长度的方式实现，因此需要维护窗口内的元素个数最多只能为 nums.size()。这样的话维护先进先出就很必要了，当滑动窗口元素等于 nums.size()时，就要先 pop_front()才能继续 push_back()元素了(均按照单调队列的算法来实现 pop_front()和 push_back())。（**子数组**最多只能包含固定缓冲区 nums 中的每个元素一次。形式上，对于子数组 nums[i], nums[i + 1], ..., nums[j]，不存在 i <= k1, k2 <= j 其中 k1 % n == k2 % n）

【经验】一般对于这种问题，用单调队列实现的滑动窗口里面最好不要放值，而是放索引，这样可以通过 window.front()获得队头元素的索引，从而进行一些窗口长度的判断（当前索引 i 与队头元素索引相减即得窗口长度，即可判断窗口长度是否符合题目要求，不符合则要 pop_front()），具体看 T918

T53：最大子数组和 （三种解法）

T974：**和可被 K 整除的子数组** ：给定一个整数数组 nums 和一个整数 k，返回其中元素之和可被 k 整除的（连续、非空）**子数组**的数目。 前缀和+哈希表

关键在于问题的转化：

sum(nums[i..j]) % k == 0

=> (sum(nums[0..j]) - sum(nums[0..i])) % k == 0

=> sum(nums[0..i]) % k == sum(nums[0..j]) % k

所以只需要用一个哈希表记录前缀和除以 k 的余数，构造好 preSum 数组后，遍历 preSum 数组时通过查看哈希表中是否已经存在相同的余数即可知道式子满不满足，也就知道了题目条件满不满足了。（实际上只需遍历一次，在遍历计算 preSum[i]时，就可以顺便计算余数，存入哈希表，比较是否满足条件了）

注意：涉及到和为 xxx 的子数组，就是要考察前缀和技巧和哈希表的结合使用了。

T525：连续数组，方法：前缀和+哈希表，哈希表记录 valToIndex，即前缀和的值到该前缀和的索引的映射，问的是最长的子数组，那就是先记录下第一次出现的某前缀和的索引 i，这就是该前缀和最小的索引，之后遍历过程中如果再出现了同样的前缀和，那么用此时的前缀和索引减去之前记录下的同样的前缀和值第一次出现时的索引（最小的索引），即得最长的子数组。

T1124：给你一份工作时间表 hours，上面记录着某一位员工每天的工作小时数。

我们认为当员工一天中的工作小时数大于 8 小时的时候，那么这一天就是「**劳累的一天**」。所谓「表现良好的时间段」，意味在这段时间内，「劳累的天数」是严格 **大于**「不劳累的天数」。请你返回「表现良好时间段」的最大长度。

技巧：问题的转换，归一化，题目的界限是>8 或<=8，那么就可以二分化，>8 的记为 1，<=8 的记为-1（preSum[i]=preSum[i-1]+(hours[i-1]>8 ? 1 : -1)），这样就可以转化为求子数组和大于 0 的最长子数组问题了。（最长子数组问题的 base 版本是 T525）

T238：前缀积问题，方法：定义两个数组：

1. *从左到右的前缀积，prefix[i] 是 nums[0..i] 的元素积*
2. *从右到左的前缀积，suffix[i] 是 nums[i..n-1] 的元素积*

（类似题：T713，注意滑动窗口收集答案时，考虑所有合法的子数组，都要加到可行的答案中去）

二、 BFS 算法

1. T934：BFS+岛屿问题 （学习思路，很重要）

类似题：T547 （注意这题和 T934 在图的结点表示上的不同，对于题目 695，图的表示方法是，每个位置代表一个节点，每个节点与上下左右四个节点相邻。而在 T547 里面，每一

行（列）表示一个节点，它的每列（行）表示是否存在一个相邻节点）

T934：

给你一个大小为 `n x n` 的二元矩阵 `grid` ，其中 `1` 表示陆地，`0` 表示水域。

岛 是由四面相连的 `1` 形成的一个最大组，即不会与非组内的任何其他 `1` 相连。`grid` 中 恰好存在两座岛 。

你可以将任意数量的 `0` 变为 `1` ，以使两座岛连接起来，变成 一座岛 。

返回必须翻转的 `0` 的最小数目。

方法一：广度优先搜索

题目中求最少的翻转 0 的数目等价于求矩阵中两个岛的最短距离，因此我们可以广度优先搜索来找到矩阵中两个块的最短距离。首先找到其中一座岛，然后将其不断向外延伸一圈，直到到达了另一座岛，延伸的圈数即为最短距离。广度优先搜索时，我们可以将已经遍历过的位置标记为 -1，实际计算过程如下：

- 我们通过遍历找到数组 $grid$ 中的 1 后进行广度优先搜索，此时可以得到第一座岛的位置集合，记为 $island$，并将其位置全部标记为 -1。
- 随后我们从 $island$ 中的所有位置开始进行广度优先搜索，当它们到达了任意的 1 时，即表示搜索到了第二个岛，搜索的层数就是答案。

2. T417：DFS 问题，反向思考：

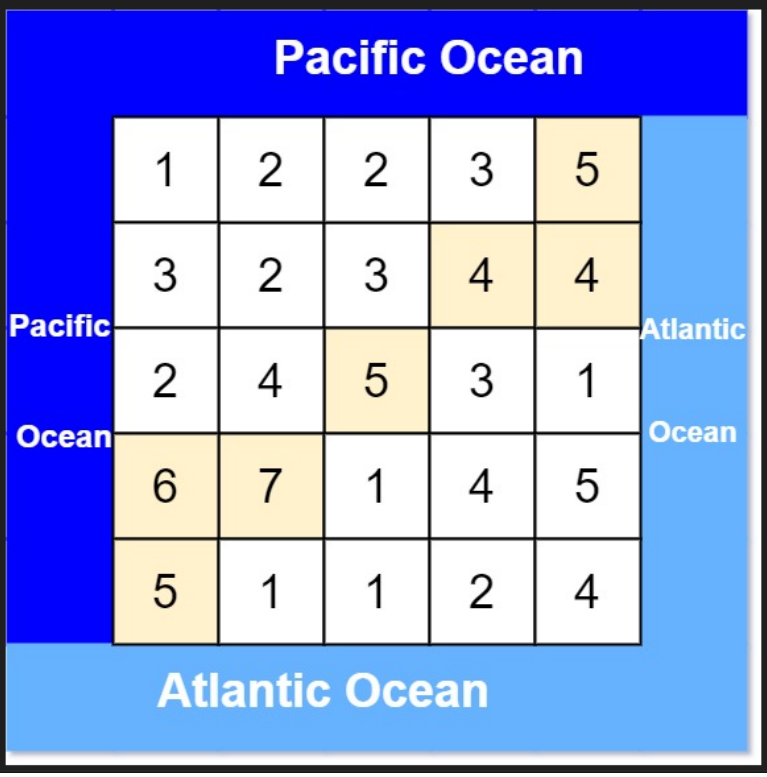
有一个 `m x n` 的矩形岛屿，与 **太平洋** 和 **大西洋** 相邻。 **“太平洋”** 处于大陆的左边界和上边界，而 **“大西洋”** 处于大陆的右边界和下边界。

这个岛被分割成一个由若干方形单元格组成的网格。给定一个 `m x n` 的整数矩阵 `heights` ， `heights[r][c]` 表示坐标 `(r, c)` 上单元格 **高于海平面的高度** 。

岛上雨水较多，如果相邻单元格的高度 **小于或等于** 当前单元格的高度，雨水可以直接向北、南、东、西流向相邻单元格。水可以从海洋附近的任何单元格流入海洋。

返回网格坐标 `result` 的 **2D 列表** ，其中 `result[i] = [ri, ci]` 表示雨水从单元格 `(ri, ci)` 流动 **既可流向太平洋也可流向大西洋** 。

示例 1:



题解

虽然题目要求的是满足向下流能到达两个大洋的位置，如果我们对所有的位置进行搜索，那么在不剪枝的情况下复杂度会很高。因此我们可以反过来想，从两个大洋开始向上流，这样我们只需要对矩形四条边进行搜索。搜索完成后，只需遍历一遍矩阵，满足条件的位置即为两个大洋向上流都能到达的位置。

3. T542：01 矩阵

方法：从 0 出发进行 BFS，遍历到的 1 离 0 的距离就是 cur 离 0 的距离再加上 1

三、 动态规划问题

归纳：动态规划有自顶向下的递归思路和自底向上的迭代思路

递归的特点是未知推已知，从未知的大问题逐步分解成小问题直到 base case，然后所有问题都解决了，利用递归思路的话，dp 函数的定义就应该是：dp(i)表示 s[i..]的最小/最大/xxx 的方法数 xxx；

而迭代的特点是已知推未知，从小问题逐步推到最终的大问题，利用迭代思路的话，dp 数组的定义应该是：dp[i]表示 s[0..i]（或者不一定从 0 开始，表示以 i 结束的子字符串怎么怎么样）

背包问题总结：

https://www.programmercarl.com/%E8%83%8C%E5%8C%85%E6%80%BB%E7%BB%93%E7%AF%87.html#%E8%83%8C%E5%8C%85%E9%80%92%E6%8E%A8%E5%85%AC%E5%BC%8F

问能否能装满背包（或者最多装多少）：dp[j] = max(dp[j], dp[j - nums[i]] + nums[i]);，对应题目如下：

- 动态规划：416.分割等和子集
- 动态规划：1049.最后一块石头的重量 II

问装满背包有几种方法：dp[j] += dp[j - nums[i]]，对应题目如下：

- 动态规划：494.目标和
- 动态规划：518. 零钱兑换 II
- 动态规划：377.组合总和IV
- 动态规划：70. 爬楼梯进阶版（完全背包）

问背包装满最大价值：dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);，对应题目如下：

- 动态规划：474.一和零

问装满背包所有物品的最小个数：dp[j] = min(dp[j - coins[i]] + 1, dp[j]);，对应题目如下：

- 动态规划：322.零钱兑换
- 动态规划：279.完全平方数

遍历顺序

01背包

在动态规划：关于01背包问题，你该了解这些！ 中我们讲解二维dp数组01背包先遍历物品还是先遍历背包都是可以的，且第二层for循环是从小到大遍历。

和动态规划：关于01背包问题，你该了解这些！（滚动数组） 中，我们讲解一维dp数组01背包只能先遍历物品再遍历背包容量，且第二层for循环是从大到小遍历。

一维dp数组的背包在遍历顺序上和二维dp数组实现的01背包其实是有很大差异的，大家需要注意！

完全背包

说完01背包，再看看完全背包。

在动态规划：关于完全背包，你该了解这些！ 中，讲解了纯完全背包的一维dp数组实现，先遍历物品还是先遍历背包都是可以的，且第二层for循环是从小到大遍历。

但是仅仅是纯完全背包的遍历顺序是这样的，题目稍有变化，两个for循环的先后顺序就不一样了。

如果求组合数就是外层for循环遍历物品，内层for遍历背包。

如果求排列数就是外层for遍历背包，内层for循环遍历物品。

相关题目如下：

- 求组合数：动态规划：518.零钱兑换II
- 求排列数：动态规划：377. 组合总和 Ⅳ 、动态规划：70. 爬楼梯进阶版（完全背包）

如果求最小数，那么两层for循环的先后顺序就无所谓了，相关题目如下：

- 求最小数：动态规划：322. 零钱兑换 、动态规划：279.完全平方数

对于背包问题，其实递推公式算是容易的，难是难在遍历顺序上，如果把遍历顺序搞透，才算是真正理解了。

1. T322：凑零钱问题（最经典的动态规划问题）

```cpp
int coinChange(vector<int>& coins, int amount) {
    vector<int> dp(amount+1,INT_MAX);

    dp[0]=0;
    for(int i=0;i<coins.size();i++)  //物品
    {
        for(int j=coins[i];j<=amount;j++)  //背包
        {
            if(dp[j-coins[i]]!=INT_MAX) // 如果dp[j
                dp[j]=min(dp[j],dp[j-coins[i]]+1);
        }
    }
    return dp[amount]==INT_MAX ? -1 : dp[amount];
}
```

问装满背包最少需要多少个物品，物品是零钱，背包容量是 amount

（类似思想的题目：T279：完全平方数）

T279：（也是属于完全背包问题，顺序无关紧要故先遍历物品或者先遍历背包都可以）

```
int numSquares(int n) {
    // 定义：和为 i 的平方数的最小数量是 dp[i]
    vector<int> dp(n+1,INT_MAX);
    dp[0]=0;

    for(int i=1;i*i<=n;i++)  //物品
    {
        for(int j=i*i;j<=n;j++)  //背包
        {
            // j-i*i 只要再加一个平方数i*i即可凑出j
            dp[j]=min(dp[j],dp[j-i*i]+1);
        }
    }
    return dp[n];
}
```

本题的物品就是<=n 的完全平方数，背包容量就是 n，问装满背包最少需要多少个物品

T279：完全平方数 的类似问题：T139 单词拆分：

题解

类似于完全平方数分割问题，这道题的分割条件由集合内的字符串决定，因此在考虑每个分割位置时，需要遍历字符串集合，以确定当前位置是否可以成功分割。注意对于位置 0，需要初始化值为真。

本题的“词典”就相当于物品，每个物品可以用无数次，因此属于完全背包问题，要用词典里的物品凑满背包（目标字符串），但是是有顺序的，按照顺序拼接到一起才能形成目标字符串，因此是排列类完全背包问题。

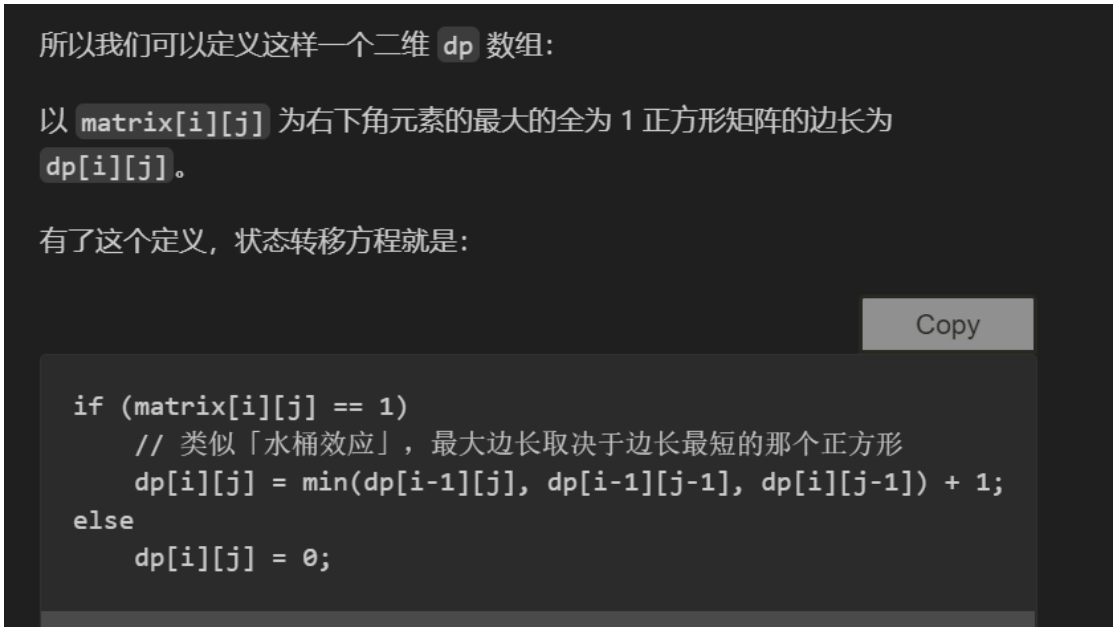
2. T413 等差数列划分

题解

这道题略微特殊，因为要求是等差数列，可以很自然的想到子数组必定满足 num[i] - num[i-1] = num[i-1] - num[i-2]。然而由于我们对于 dp 数组的定义通常为以 i 结尾的，满足某些条件的子数组数量，而等差子数组可以在任意一个位置终结，因此此题在最后需要对 dp 数组求和。

3. T221：最大正方形

学习：对于二维矩阵场景，如何定义 dp 数组的含义



4. T91： 解码方法 关键在于找准状态转移方程

5. T300：LIS（最长递增子序列）

6. T1143：LCS（最长公共子序列）

题解

对于子序列问题，第二种动态规划方法是，定义一个 dp 数组，其中 dp[i] 表示到位置 i 为止的子序列的性质，并不必须以 i 结尾。这样 dp 数组的最后一位结果即为题目所求，不需要再对每个位置进行统计。

在本题中，我们可以建立一个二维数组 dp，其中 dp[i][j] 表示到第一个字符串位置 i 为止、到第二个字符串位置 j 为止、最长的公共子序列长度。这样一来我们就可以很方便地分情况讨论这两个位置对应的字母相同与不同的情况了。

7. T343：整数拆分

思路：关键在于既可以拆分成两部分，又可以拆分成多个部分，在比较最大乘积结果时需要考虑两种情况，拆成两部分的直接乘，拆成多个部分的，一部分是当前遍历到的结果，另一部是存在 dp 数组里的结果，都要拿来取 max 值。

8. T96：不同的二叉搜索树

思路：dp[i]：从 1 到 i 共 i 个结点能有 dp[i]个不同的二叉搜索树，每次让 j 做根节

点，dp[j-1]左子树有几种情况，dp[i-j]右子树有几种情况。

打家劫舍问题

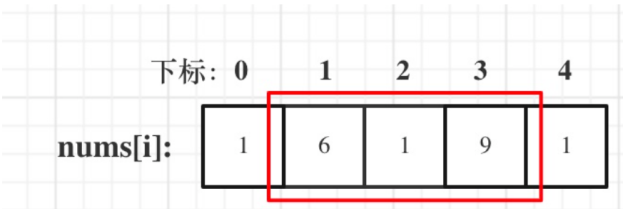
1. T198：打家劫舍

思路：dp[i]表示：nums[0..i]所能偷到的最多金币数，当前状态决定偷 or 不偷，若偷则获得当前金币，同时上一家不能偷，只能加上 dp[i-2]的价值；若不偷，则上一家可以偷，dp[i-1]，偷与不偷取 max 即可。

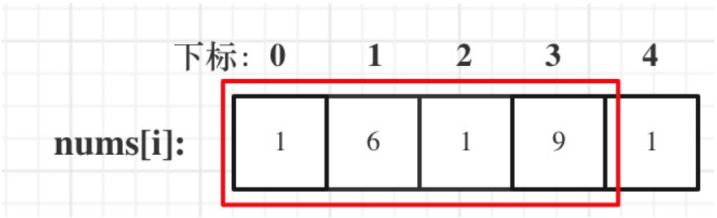
2. T213：打家劫舍 II

思路：对于成环问题，解决方法就是把环打开，化为线性来进行考虑，因为首尾相接故首和尾只能选一个来偷，要不就都不偷，无非就是 3 种情况：

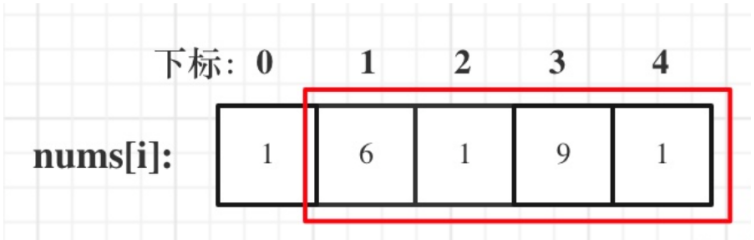
- 情况一：考虑不包含首尾元素



- 情况二：考虑包含首元素，不包含尾元素



- 情况三：考虑包含尾元素，不包含首元素



而情况 1 含在了情况 2 和 3 中，情况 2 和 3 考虑的范围包括了 1，故其实只要情况 2 和 3 比较一下那种情况获得的金币最多即可。

3. T337：打家劫舍 III （好题）

思路：利用二叉树后序遍历，在后序位置可以获得左子树偷或不偷所能获得的最大价

值和右子树偷或不偷所能获得的最大价值。然后分别算出 root 偷或不偷的价值进行返回即可。left_dp[0]表示左孩子不偷，left_dp[1]表示左孩子偷。不偷 root，左右孩子可偷可不偷，取个最大值，把左右孩子的最大价值加在一起；偷 root，左右孩子都不能偷。

股票买卖问题

1. T121：买卖股票的最佳时机 （最多可以买卖一次）

思路：注意 dp 数组如何定义：dp[i][0]：第 i 天不持有股票；dp[i][1]：第 i 天持有股票。要表示第 i 天“持有”或者“不持有”，而不是记录第 i 天是否存在买卖行为。第 i-1 天持有股票而第 i 天空仓则说明在第 i 天卖出了股票；第 i-1 天空仓，第 i 天持有股票则说明第 i 天买入了股票。

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

    //今天买入股票则之前一定是空仓且还没有买过的，因为题目要求只能买卖一次
    dp[i][1]=max(dp[i-1][1],-prices[i]);
}
return dp[n-1][0];
```

2. T122：买卖股票的最佳时机 II （可以买卖多次） （因为可以多次买卖故也可以用贪心算法解决本题）

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

return dp[n-1][0];
```

与上一题唯一区别就在于上一题只能买卖一次，因此第 i 天如果买入了股票那么前面一定是没有买卖过的，就是 0-prices[i]；而本题可以进行多次买卖，因此是 dp[i-1][0]-prices[i]

3. T123：买卖股票的最佳时机 III （可以买卖 2 次） （好题）

思路：关键在于把两次买卖的 5 个状态都表示出来，即不操作，第一次持有/不持有，第二次持有/不持有，最后肯定是不持有获得的价值最大，并且第二次不持有包含了第一次不持有的情况的最大价值，即使第一次不持有已经获得了最大价值，也可以在当天再进行第二次买卖，这样一来第二次不持有的价值和第一次不持有的价值一样，故第二次不持有包含

了第一次不持有的最大价值，最后返回第 n-1 天第二次不持有的价值即可。

```
int maxProfit(vector<int>& prices) {
    int n=prices.size();

    //dp[i][0]: 不操作
    //dp[i][1]: 第i天第一次持有
    //dp[i][2]: 第i天第一次不持有
    //dp[i][3]: 第i天第二次持有
    //dp[i][4]: 第i天第二次不持有
    vector<vector<int>> dp(n,vector<int>(5));
    dp[0][0]=0;
    dp[0][1]=-prices[0];
    dp[0][2]=0;
    dp[0][3]=-prices[0];
    dp[0][4]=0;      //可以理解为第一天买了又卖了，故第二次不持有

    for(int i=1;i<n;i++)
    {
        dp[i][1]=max(dp[i-1][1],dp[i-1][0]-prices[i]);
        dp[i][2]=max(dp[i-1][2],dp[i-1][1]+prices[i]);
        dp[i][3]=max(dp[i-1][3],dp[i-1][2]-prices[i]);
        dp[i][4]=max(dp[i-1][4],dp[i-1][3]+prices[i]);
    }
    return dp[n-1][4];
}
```

4. T188：买卖股票的最佳时机 IV （可以买卖 k 次）

思路：其实就是把买卖股票的最佳时机 III 中的 dp 数组的第二个维度 j 抽象出来进行循环，表示第 k 次买卖行为即可。

规律：对于第 i 天而言，如果 i-1 天和第 i 天的状态一样（即 dp 数组的第二个维度一样），那么不需要任何操作，表示第 i 天没有进行买卖操作；如果 i-1 天的状态（即 dp 数组的第二个维度）为第 i 天的 dp 数组的第二个维度减去 1，那么说明第 i 天进行了买卖，按照 dp[i][1]表示持有，dp[i][2]表示不持有，第二个维度 j 两步两步更新（j+=2），分析第 i 天是否是持有状态，若是持有状态，而又进行了买卖，那么第 i-1 天一定是不持有的，那么 dp[i-1][j]-prices[i]表示第 i 天买入即可；第 i 天不持有的情况也类似分析。

除了 0 以外，偶数就是卖出，奇数就是买入。dp[0][j]当 j 为奇数的时候都初始化为 -prices[0]

```cpp
int maxProfit(int k, vector<int>& prices) {
    int n=prices.size();

    //dp[i][0]：不操作
    //dp[i][j+1]：第i天持有
    //dp[i][j+2]：第i天不持有
    vector<vector<int>> dp(n,vector<int>(2*k+1,0));
    dp[0][0]=0;
    for(int j=1;j<2*k;j+=2)
        dp[0][j]=-prices[0];    //初始化：第0天不持有就是0，持有就是-prices[0]

    for(int i=1;i<n;i++)
    {
        for(int j=0;j<2*k;j+=2)    //j+=2表示进入下一次买卖
        {
            dp[i][j+1]=max(dp[i-1][j+1],dp[i-1][j]-prices[i]);  //持有
            dp[i][j+2]=max(dp[i-1][j+2],dp[i-1][j+1]+prices[i]);  //不持有
        }
    }
    return dp[n-1][2*k];
}
```

5. T309：[买卖股票的最佳时机含冷冻期](#)（可以买卖无限次）

思路：本题因为有冷冻期的存在，因此相比于之前的题目需要把“不持有股票”的状态进一步拆解成为两个状态：一直保持卖出的状态、卖出股票的这个操作

然后冷冻期当天也是一个状态，这样 4 个状态进行转移即可。

```cpp
int maxProfit(vector<int>& prices) {
    int n=prices.size();
    //dp[i][0]：持股
    //dp[i][1]：保持卖出状态
    //dp[i][2]：卖出股票操作
    //dp[i][3]：冷冻期(指刚好是冷冻期那一天)
    vector<vector<int>> dp(n,vector<int>(4));
    dp[0][0]=-prices[0];
    dp[0][1]=0,dp[0][2]=0,dp[0][3]=0;
    for(int i=1;i<n;i++)
    {
        dp[i][0]=max({dp[i-1][0],dp[i-1][1]-prices[i],dp[i-1][3]-prices[i]});
        dp[i][1]=max(dp[i-1][1],dp[i-1][3]);
        dp[i][2]=dp[i-1][0]+prices[i];
        dp[i][3]=dp[i-1][2];
    }
    return max({dp[n-1][1],dp[n-1][2],dp[n-1][3]});
}
```

当天持股：max{前一天也持股、前一天保持着卖出状态，当天买入、前一天是冷冻期，当天买入}

当天保持卖出状态：max{前一天也保持着卖出状态、前一天是冷冻期}

当天卖出股票操作：前一天持有股票，当天卖出

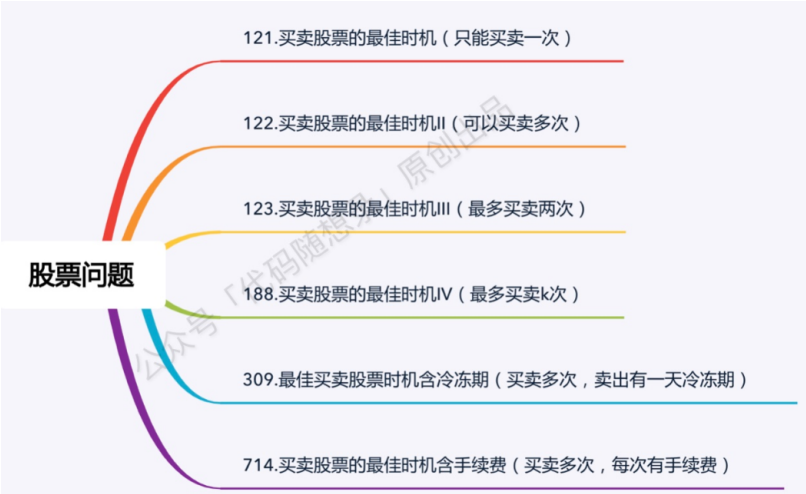
当天为冷冻期：前一天做了卖出了股票操作

6. T714：买卖股票的最佳时机含手续费 （可以买卖无限次）

思路：只需要把 fee 算在买股票的那个时候就行，相当于买股票变贵了，多了一个手续费，其他代码都和买卖股票的最佳时机 II 基本一样。

股票问题总结：

https://www.programmercarl.com/%E5%8A%A8%E6%80%81%E8%A7%84%E5%88%92-%E8%82%A1%E7%A5%A8%E9%97%AE%E9%A2%98%E6%80%BB%E7%BB%93%E7%AF%87.html#%E5%8D%96%E8%82%A1%E7%A5%A8%E7%9A%84%E6%9C%80%E4%BD%B3%E6%97%B6%E6%9C%BA



子序列问题（dp）

1. T300：最长递增子序列

定义：dp[i] 表示以 nums[i] 这个数结尾的最长递增子序列的长度。用双指针——快慢指针进行遍历，内层循环的指针 j（j 遍历 0~i-1 的元素）如果发现 nums[j] < nums[i]，那么可以加上 i 作为更长的递增子序列，即 dp[j]+1，dp[i]=max(dp[i],dp[j]+1)即可。注意最后还要遍历一遍 dp 数组找到其中记录的最大的元素，因为 dp 数组定义的是以这个数结尾的最长递增子序列的长度，但是整个数组最长递增子序列不一定是以 dp[n-1]结尾的子序列。

类似题：T646：最长数对链

思路：本题要点在于：构建最长递增子序列的 dp 数组时，前面的 j 位置的元素的第 1 个关键字与后面的 i 位置的第 0 个关键字进行比较：

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

}
```

类似题：T674：最长连续递增序列 （这题是“子数组”，不是“子序列”，要下标连续的才行）此时状态转移就只是一层循环：

定义：dp[i] 表示以 nums[i] 这个数结尾的最长递增子序列的长度

```
for(int i=1;i<n;i++)
{
    if(nums[i-1]<nums[i])
        dp[i]=dp[i-1]+1;
}
```

2. **T718：最长重复子数组** （最长公共子数组问题）

注意：dp[i][j]定义：nums1 中以 i-1 结尾的子数组与 nums2 中以 j-1 结尾的子数组的最长公共子数组长度。之所以要有这个索引偏移，是因为想把第 0 行和第 0 列都初始化为 0，然后再正常进行状态转移。

如果定义 dp[i][j]为以下标 i 为结尾的 nums1，和以下标 j 为结尾的 nums2，那么第一行和第一列毕竟要进行初始化，如果 nums1[i] 与 nums2[0] 相同的话，对应的 dp[i][0] 就要初始为 1， 因为此时最长重复子数组为 1。 nums2[j] 与 nums1[0]相同的话，同理。因此这样定义 dp 数组的话初始化就会比较麻烦，所以还是有索引偏移（dp[i][j]表示 nums1[..i]和 nums2[..j]的最长公共子数组长度）更好，很多类似的题目也是这样去定义 dp 数组的，也是有索引偏移更好。

dp 数组定义方法归纳：

1. 将 dp[i]定义为以 nums 数组中下标 i 为结尾的 xxx

这种定义的话最后还要遍历一遍 dp 数组找到其中的最大值

T300：最长递增子序列　（一维 dp）

T674：最长连续递增序列　（最长递增子数组）　（一维 dp）

T718：最长重复子数组　（二维 dp）

T53：最大子数组和　（一维 dp）

2. 将 dp[i]定义为 nums[0..i]的 xxx

这种定义的话最后 dp 数组的最后一个元素，即 dp[nums.size()-1]就是最后的答案

T1143：LCS（最长公共子序列）　（二维 dp）

T1035：不相交的线　（二维 dp）　本题就是最长公共子序列问题套了另外一个情景，代码与 T1143 完全相同。

T392：判断子序列　（二维 dp）　本题需要转个弯，要判断 s 是否为 t 的子序列，只需要求出 s 和 t 的最长公共子序列，如果这个最长公共子序列长度等于 s 的长度，那么就是 true 了，否则为 false，所以本质还是 T1143 的最长公共子序列问题。

（本题也可以用双指针思路实现，时间复杂度更低）

T72：编辑距离　（二维 dp）

总结：对于要考虑 nums[i]这个元素本身的贡献时，dp[i]要定义为“以 i 为结尾的 xxx”，最后要遍历 dp 数组取最大值作为答案，例如递增子序列，要考虑当前 nums[i]符不符合递增条件才能进行状态转移，如果定义的是 nums[0..i]的 xxx，即前 i 个元素的最长递增子序列，则很难进行递推公式状态转移。

而对于不需要考虑 nums[i]这个元素本身的贡献时，dp[i]定义为 “nums[0..i]的 xxx”，例如最长公共子序列问题，只需要知道前 i 个元素的最长公共子序列是多少然后比较当前元素是否相同就可以确定最长公共子序列是否可以变长。

但是要注意的是 T718：最长重复子数组，这题是最长公共子数组问题，由于子数组必须是连续的，因此必须考虑 nums[i]元素本身，如果相同才可以推出 dp[i]，因此 dp[i]要定义为“以 i 为结尾的 xxx”。

子数组子序列问题总结文章：

https://juejin.cn/post/7247354308257382456

3. T115：不同的子序列

思路：关键在于把问题看成是删除 s 中的元素使得最后和 t 串一样，有几种不同的删除方法？先比较当前 s[i-1]和 t[j-1]是否相等，如果相等，则 dp[i][j]=dp[i-1][j-1]+dp[i-1][j]，即两个串同时往前走或者 s 往前走（因为是在求 s 的子串如何变成 t），例如：s：bagg；t：bag，最后一个 g 相同，则 s 可以往前走一步，发现也是跟 t 想同的，因此要把这种情况算进来，dp[i-1][j]就相当于“编辑距离”里面删除 s 中下标为 i-1 的元素的操作。

如果 s[i-1]和 t[j-1]不相等，则 dp[i][j]=dp[i-1][j]，就直接把 s 中下标为 i-1 的元素删除了，s 的指针往前走。

4. T583：两个字符串的删除操作

思路 1：求出两个字符串的最长公共子序列，这个就是两个字符串分别完成删除后最后剩下的相同的子串了。设最长公共子序列长度为 L，则总的操作步数为(s1.length()-L)+(s2.length()-L)

思路 2：按照最小编辑距离的解法来：

若 word1[i-1]==word2[j-1]，则 dp[i][j]=dp[i-1][j-1];

否则要么删除 i-1 这个位置的字符：dp[i-1][j]+1　　（+1 是一次删除操作步数）

要么删除 j-1 这个位置的字符：dp[i][j-1]+1

要么都删除：dp[i-1][j-1]+2

这 3 种情况取个 min 即可。

5. T647：回文子串

思路：双指针，左指针 i，右指针 j，如果 s[i]==s[j]，则去看 dp[i+1][j-1]是否为 true，若是，则 res++，dp[i][j]定义为：s 中[i,j]闭区间是否为回文字符串。注意状态转移遍历顺序：由于 dp[i][j]由 dp[i+1][j-1]转移而来，因此从下往上遍历，从左往右遍历，即外层 i 倒序遍历，内层 j 正序遍历。

6. T516：最长回文子序列

思路：**dp 数组定义：在子串 s[i..j] 中，最长回文子序列的长度为 dp[i][j]，跟**

上一题 T647 类似，如果 s[i]==s[j]则最长回文子序列长度为 dp[i+1][j-1]+2，否则为 max(dp[i+1][j],dp[i][j-1])，看看 s[i+1..j] 和 s[i..j-1] 谁的回文子序列更长。

7. T650：两个键的键盘

题解

不同于以往通过加减实现的动态规划，这里需要乘除法来计算位置，因为粘贴操作是倍数增加的。我们使用一个一维数组 dp，其中位置 i 表示延展到长度 i 的最少操作次数。对于每个位置 j，如果 j 可以被 i 整除，那么长度 i 就可以由长度 j 操作得到，其操作次数等价于把一个长度为 1 的 A 延展到长度为 i/j。因此我们可以得到递推公式 dp[i] = dp[j] + dp[i/j]。

思路：本题难点在于递推公式，想到如果 i 可以整除以 j，那么只需要通过 dp[i/j] 步就可以从 j 长度扩展到 i 长度了，从初态扩展到 j 长度的最少操作步数为 dp[j]，因此 dp[j]+dp[i/j]即得到 dp[i].

8. T10：正则表达式匹配 （难）

要分多种情况讨论，具体分析见：

https://labuladong.online/algo/dynamic-programming/regular-expression-matching/#%E4%BA%8C%E3%80%81%E5%8A%A8%E6%80%81%E8%A7%84%E5%88%92%E8%A7%A3%E6%B3%95

9. T44：通配符匹配

类似于 T10，这里的 ？通配符就是第 10 题的 .，这里的 * 就是第 10 题的 .？组合

动态规划最终总结文章：

https://www.programmercarl.com/%E5%8A%A8%E6%80%81%E8%A7%84%E5%88%92%E6%80%BB%E7%BB%93%E7%AF%87.html#%E5%8A%A8%E8%A7%84%E7%BB%93%E6%9D%9F%E8%AF%AD

01 背包问题

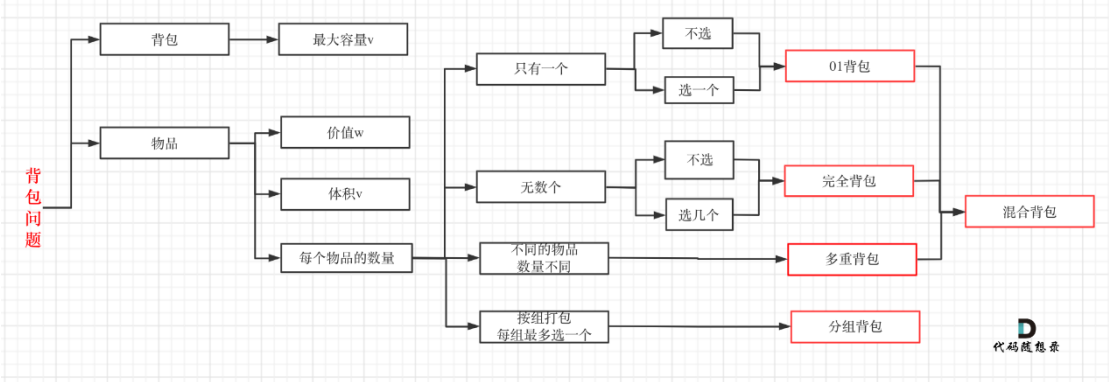
1. （01 背包问题）T416：分割等和子集 （重点）

思路：抽象成 01 背包问题：整个数组和的一半就是背包的总容量，可选物品就是数组

中的每一个数字，物品的重量就是数组中每个数字的值，不需要考虑物体的价值，因为本题不是像传统的 01 背包问题那样要求在给定容量范围内使得装入物品价值最大，而是只要满足装入物品的重量（数组中每个数字的值）之和是否能恰好等于整个数组和的一半即可。

一维 dp 数组解 01 背包的核心代码模板：

```
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]);
    }
}
```



2. （01 背包问题）T1049：[最后一块石头的重量 II](#)　（好题）

思路：根据题意抽象出：只要将原数组分为和大致相等的两堆即可，如 21 可以分为 10 和 11，这样粉碎后最终剩 1。这就转化为了上面的分割为等和子集问题，只不过不需要完全相等，近似相等（差 1）也可以。要注意的是这里的物品重量和物品价值其实都指的是数字值本身，因为没有明显的“价值”意义存在，所以目标是尽可能装满重量为 sum/2 的背包，然后每装入一个石头，它的价值就可以认为是它的重量，因为不一定能恰好装满 sum/2 的，那么只需要求在容量为 sum/2 的要求下的最大价值即可（就是 01 背包的标准型），最大价值也就是最大重量（如 sum/2=10 时，可能最多只能装重量为 9 的东西，再装任意一个就超过 10 了，那么此时求到的 9 就是最大的价值，也是存在 dp 数组里的值）

3. （01 背包问题）T494：[目标和](#)　（难）

https://www.programmercarl.com/0494.%E7%9B%AE%E6%A0%87%E5%92%8C.html#%E6%80%9D%E8%B7%AF

如何转化为01背包问题呢。

假设加法的总和为x，那么减法对应的总和就是sum - x。

所以我们要求的是 x - (sum - x) = target

x = (target + sum) / 2

此时问题就转化为，装满容量为x的背包，有几种方法。

这里的x，就是bagSize，也就是我们后面要求的背包容量。

这次和之前遇到的背包问题不一样了，之前都是求容量为 j 的背包，最多能装多少。本题则是装满有几种方法。其实这就是一个组合问题了。dp[j] 表示：填满 j（包括 j）这么大容积的包，有 dp[j]种方法。

只要搞到nums[i]，凑成dp[j]就有dp[j - nums[i]] 种方法。

例如：dp[j]，j 为5，

- 已经有一个1（nums[i]） 的话，有 dp[4]种方法 凑成 容量为5的背包。
- 已经有一个2（nums[i]） 的话，有 dp[3]种方法 凑成 容量为5的背包。
- 已经有一个3（nums[i]） 的话，有 dp[2]中方法 凑成 容量为5的背包
- 已经有一个4（nums[i]） 的话，有 dp[1]中方法 凑成 容量为5的背包
- 已经有一个5 （nums[i]）的话，有 dp[0]中方法 凑成 容量为5的背包

那么凑整dp[5]有多少方法呢，也就是把 所有的 dp[j - nums[i]] 累加起来。

所以求组合类问题的公式，都是类似这种：

```
dp[j] += dp[j - nums[i]]
```

这个公式在后面在讲解背包解决排列组合问题的时候还会用到！

4. （01 背包问题）T474：一和零

思路：物品的重量有 2 个维度：x 个 0 和 y 个 1，dp[i][j]：容量为 i 个 0，j 个 1 的背

包最多能装多少个物品，使用的还是一维 dp 解决 01 背包的思路，只是这里物品重量有 2 个维度所以要用 2 维 dp 数组

注意与 T1049 最后一块石头的重量 II 的区别，T1049 中 dp 数组记录的是容量为 j 的背包能装下的最大价值是多少（这里的价值具体指的就是石头的重量）；而本题 T474 的 dp 数组记录的是容量为 j 的背包能装下多少个物品，是物品的个数，而不是物品的重量或是价值。

（01 背包问题）归纳：层层递进，01 背包的变体：

此时我们讲解了0-1背包的多种应用，

- 纯 0 - 1 背包 是求 给定背包容量 装满背包 的最大价值是多少。
- 416. 分割等和子集 是求 给定背包容量，能不能装满这个背包。
- 1049. 最后一块石头的重量 II 是求 给定背包容量，尽可能装，最多能装多少
- 494. 目标和 是求 给定背包容量，装满背包有多少种方法。
- 本题是求 给定背包容量，装满背包最多有多少个物品。

完全背包问题

1. T518：零钱兑换 II

由于零钱有无数个，因此是完全背包问题（01 背包：每个物品只有一个；完全背包：每个物品有无数个）

完全背包与 01 背包问题的解法区别就在于遍历顺序：对于使用一维 dp 数组的方法而言：01 背包是先遍历物品再遍历背包，背包要倒序遍历；

纯 01 背包问题（给定容量问装入的最大价值是多少）：先遍历物品或先遍历背包都可以

完全背包：对于问有几种组合的数量：先遍历物品再遍历背包，背包要正序遍历（保证每个物品可以装入多次，即物品数量无限）　（如本题 T518）

注意注意注意：

对于问有几种排列的数量，则要反过来，先遍历背包再遍历物品，背包要正序遍历（保证每个物品可以装入多次，即物品数量无限）　（如题 T377：组合总和 Ⅳ）

如果求组合数就是外层for循环遍历物品，内层for遍历背包。

如果求排列数就是外层for遍历背包，内层for循环遍历物品。

求组合数：动态规划：518.零钱兑换II 求排列数：动态规划：377. 组合总和 IV 、动态规划：70. 爬楼梯进阶版（完全背包） 求最小数：动态规划：322. 零钱兑换 、动态规划：279.完全平方数

dp[j]定义为：容量为 j 的背包能装下零钱的方法数

dp[0]一定要为 1，dp[0] = 1 是 递归公式的基础。如果 dp[0] = 0 的话，后面所有推导出来的值都是 0 了。

类似于 T494 目标和，都是问“方法数”的问题，这类问题的递推公式就是：

```
dp[j] += dp[j - nums[i]];
```

其实是 dp[j]=dp[j]+dp[j-nums[i]]，dp[j]表示不装入，方法数就还是那么多；dp[j-nums[i]]表示装入，那就要看扣除这个重量 nums[i]之后有多少种方法，再把 i 物体装入，装或不装，两种情况种类数目的相加。

注意不要+1，因为同属于一个方法，方法数并没有多一个，这跟“装多少个物品”不同

2. 爬楼梯问题进阶版：一次可以爬 m 层(1<=m<=n)，问爬上 n 层的阶梯有多少种不同的方法

本题也是完全背包的排列类问题，因为先爬一层，再爬两层和先爬两层再爬一层是不同的方法，有顺序性。

看：

https://www.programmercarl.com/0070.%E7%88%AC%E6%A5%BC%E6%A2%AF%E5%AE%8C%E5%85%A8%E8%83%8C%E5%8C%85%E7%89%88%E6%9C%AC.html#%E6%80%9D%E8%B7%AF

3. T322：零钱兑换

属于完全背包问题，问的是装满背包需要的最少物品件数

由于硬币的顺序不影响其面值总和，也不影响需要的最少个数，故可视为排列问题或组合问题都可以，即先遍历背包或先遍历物品都可以。注意要求最少个数，要用 min，因此 dp 数组初始化应该为 INT_MAX

四、 Dijkstra 算法

2023 百度之星 BD202301 公园

今天是六一节，小度去公园玩，公园一共 N 个景点，正巧看到朋友圈度度熊也在这个公园玩，于是他们约定好一块去景点 N 。 小度当前所在景点编号为 T ，从一个景点到附近的景点需要消耗的体力是 TE ，而度度熊所在景点编号为 F ，移动消耗为 FE 。 好朋友在一块，赶路都会开心很多，所以如果小度和度度熊一块移动(即在相同位置向相同方向移动)，每一步他俩的总消耗将会减少 S 。
求他俩到景点 N 时，所需要的总消耗最少是多少？

格式

输入格式： 第一行三个数值， TE, FE, S ，分别代表小度移动消耗值，度度熊移动消耗值，一起移动的消耗减少值。 $1 \leq TE, FE, S \leq 40000, S \leq TE + FE$ ；
第二行四个数值， T, F, N, M ，分别代表小度出发点，度度熊出发点，目标节点，总路径数。 $1 \leq T, F, N, M \leq 40000$ ；
接下来 M 行，每行两个整数 X, Y ，代表连通的两个景点。 $1 \leq X, Y \leq N$ 。

输出格式： 一个整数，即总消耗最小值。如果不能到达 N ， 输出-1。

样例 1

输入： 复制
```
4 4 3
1 2 8 8
1 4
2 3
3 4
4 7
2 5
5 6
6 8
7 8
```
输出： 22 复制

本题相关知识点： 图论：BFS

思路：遍历每一个除了 T、F、N 这三个结点的点，用 dijkstra 算法求出这些结点到其他结点的最短距离，每个除了 T、F、N 这三个结点的点都可以当作中转结点，设中转节点为 a，总的 cost 就是从 T 到 a 的 cost（单步 cost=TE）加上 F 到 a 的 cost（单步 cost=FE）再加上会合后从 a 到终点 N 的 cost（单步 cost=TE+FE-S），然后通过打擂台的方式比较出以哪个结点为中转结点时总的 cost 最小就行了。

```
#include <bits/stdc++.h>
using namespace std;

vector<int> dijkstra(const vector<vector<pair<int, int>>>& graph, int start) {
    int n = graph.size();
    vector<int> distTo(n, INT_MAX);
    distTo[start] = 0;
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> pq;
```

```
    pq.push({0, start});

    while (!pq.empty()) {
        auto [cur_dist, cur_id] = pq.top();
        pq.pop();

        if (cur_dist > distTo[cur_id])
            continue;

        for (auto [neigh_id, weight] : graph[cur_id]) {
            int distTonext = cur_dist + weight;
            if (distTonext < distTo[neigh_id]) {
                distTo[neigh_id] = distTonext;
                pq.push({distTonext, neigh_id});
            }
        }
    }

    return distTo;
}

int main() {
    int TE, FE, S;
    cin >> TE >> FE >> S;
    int T, F, N, M;
    cin >> T >> F >> N >> M;

    vector<vector<pair<int, int>>> graph(N + 1);
    for (int i = 0; i < M; ++i) {
        int X, Y;
        cin >> X >> Y;
        graph[X].push_back({Y, 1});
        graph[Y].push_back({X, 1});
    }

    auto distT = dijkstra(graph, T);
    auto distF = dijkstra(graph, F);
    auto distN = dijkstra(graph, N);

    if (distT[N] == INT_MAX || distF[N] == INT_MAX) {
        cout << -1 << endl;
        return 0;
    }

```

```
    int min_cost = distT[N] * TE + distF[N] * FE;
    for (int i = 1; i <= N; ++i) {
        if (distT[i] != INT_MAX && distF[i] != INT_MAX && distN[i] != INT_MAX) {
            int combined_cost = distT[i] * TE + distF[i] * FE + distN[i] * (TE + FE - S);
            min_cost = min(min_cost, combined_cost);
        }
    }

    cout << min_cost << endl;
    return 0;
}
```

五、 贪心

1. T135：分发糖果 （好题，学习贪心思想：一次遍历只贪心一侧，第二次反向遍历贪心另一侧）

题解

做完了题目 455，你会不会认为存在比较关系的贪心策略一定需要排序或是选择？虽然这一道题也是运用贪心策略，但我们只需要简单的两次遍历即可：把所有孩子的糖果数初始化为 1；先从左往右遍历一遍，如果右边孩子的评分比左边的高，则右边孩子的糖果数更新为左边孩子的糖果数加 1；再从右往左遍历一遍，如果左边孩子的评分比右边的高，且左边孩子当前的糖果数不大于右边孩子的糖果数，则左边孩子的糖果数更新为右边孩子的糖果数加 1。通过这两次遍历，分配的糖果就可以满足题目要求了。这里的贪心策略即为，在每次遍历中，只考虑并更新相邻一侧的大小关系。

在样例中，我们初始化糖果分配为 [1,1,1]，第一次遍历更新后的结果为 [1,1,2]，第二次遍历更新后的结果为 [2,1,2]。

2. T435：区间调度问题 无重叠区间

注意贪心的方法，每次选择 end 最小的

也许我们可以每次选择可选区间中开始最早的那个？但是可能存在某些区间开始很早，但是很长，使得我们错误地错过了一些短的区间。或者我们每次选择可选区间中最短的那个？或者选择出现冲突最少的那个区间？这些方案都能很容易举出反例，不是正确的方案。

正确的思路其实很简单，可以分为以下三步：

1、从区间集合 `intvs` 中选择一个区间 `x`，这个 `x` 是在当前所有区间中**结束最早的**（`end` 最小）。

2、把所有与 `x` 区间相交的区间从区间集合 `intvs` 中删除。

3、重复步骤 1 和 2，直到 `intvs` 为空为止。之前选出的那些 `x` 就是最大不相交子集。

类似题：T452：用最少数量的箭引爆气球 （本质上也是在求无重叠区间问题，有多少个无重叠区间，就至少需要多少支箭把它们射穿）

3. T763：划分字母区间

贪心的方法：寻找起始位置 start 的字母最后出现的位置索引 last，然后搜索 start 和 last 之间的所有字母，看它们最后出现的位置，如果在 last 之后，那就更新 last 变量，直到不再更新为止，这就是一次贪心的结果；然后让 start=last+1 继续遍历直到遍历完整个字符串为止。

具体实现：利用哈希表存储每个字母在字符串中最后出现的位置。（预处理）

4. T406：根据身高重建队列

非常好题：当涉及两个变量维度时，贪心不能都考虑，一定要固定下来一个维度再考虑另一个维度的贪心，否则会顾此失彼。可以先通过排序固定下一个维度的顺序，再考虑另一维度。

5. T665：非递减数列

需要仔细思考你的贪心策略在各种情况下，是否仍然是最优解。

贪心的策略应该分情况考虑，如果 nums[i]<nums[i-1]，即出现了递减序列那么：

1. i==1 时，应该把 nums[i-1]改小，因为不知道 nums[i]之后的情况是什么样的，所以把 nums[i-1]改小就是局部的最优策略，可以改小到和 nums[i]一样

2. i>1 时，如果 nums[i]>=nums[i-2]，例如 2,5,3，那么还是应该把 nums[i-1]改小，改小到和 nums[i]一样，变成 2,3,3，这样就是局部最优了。

3. i>1 且 nums[i]<nums[i-2]时，例如 3,4,2，此时[4,2]出现了递减序列，如果把 nums[i-1]改为 nums[i]，即 3,2,2，那么[3,2]还是出现了递减，那么此时改 nums[i-1]就不是局部最优策略了，应该改 nums[i]，改为 nums[i-1]，即变为 3,4,4，就符合了。

6. T376：摆动序列 （较难）

注意分析 3 种可能的情况：1. 上下坡（如：1,3,2） 2. 上下坡有平坡（如：1,3,3,2） 3. 单调有平坡（如：1,3,3,4,5）

推荐看：

https://www.programmercarl.com/0376.%E6%91%86%E5%8A%A8%E5%BA%8F%E5%88%97.html#%E7%AE%97%E6%B3%95%E5%85%AC%E5%BC%80%E8%AF%BE

视频：

https://www.bilibili.com/video/BV17M411b7NS/?vd_source=ea93ea35bcf86634a8c6985d4fdc67aa

局部最优：删除单调坡度上的节点（不包括单调坡度两端的节点），那么这个坡度就可以有两个局部峰值。整体最优：整个序列有最多的局部峰值，从而达到最长摆动序列。其实就是让序列有尽可能多的局部峰值。

7. T53：最大子数组和（综合经典，可用贪心、滑动窗口、动态规划、前缀和 4 种方法解决）

贪心思路：如果当前连续子数组和是负数，那么就立刻抛弃此时的连续子数组和，选择下一个数字作为连续子数组的起点（因为本来就负的子数组和前缀只会对下一个数组起到减小的负担作用）；如果当前连续子数组和是正数，则无论下一个数字是正是负都照样加上下一个数字，因为此时的正数子数组和前缀对后面的数字有增大作用，每步遍历都去维护 res=max(res,cur_sum)即可得到最终最大的子数组和

贪心的思路为局部最优：当前“连续和”为负数的时候立刻放弃，从下一个元素重新计算“连续和”，因为负数加上下一个元素 “连续和”只会越来越小。从而推出全局最优：选取最大“连续和”。

8. T122：买卖股票的最佳时机 II

法 1：动态规划 O(n)

法 2：贪心算法：类似差分数组的构建思路，求出相邻两天之间的利润，只收集正利润（因为 p[3]-p[0]=p[3]-p[2]+p[2]-p[1]+p[1]-p[0]，即不断累加差值最终能得到总的差值（第 0 天买入，第 3 天卖出的利润））

9. T55：跳跃游戏

思路：不要纠结于在每一个地方要往后跳几步，而是去考虑最远能够覆盖的范围，这个覆盖范围 cover 了终点的话那么就是 true。

10. T45 跳跃游戏 II

思路：记录每一步的最大覆盖范围，不断增加覆盖范围直到覆盖终点，增加覆盖范围这个操作的次数就是最终的答案（跳跃的最小步数）

11. T134：加油站

思路关键：记录 curSum 和 totalSum，关键在于理解：当 curSum<0 时，此位置之前的所有位置都不能当做起点，起点只能从 i+1 开始继续观察。

12. T738：单调递增的数字

思路：把整数转换为 string（这是要处理整数的每一位时最常用的技巧，转为 string 之后就可以通过索引遍历或者找到改整数的某一位了），从后往前遍历，如果前一位比后一位大，那么前一位减 1，后一位赋值为 9（贪心，这样最大）

13. T968：监控二叉树

思路：本题较难，要考虑到结点的三个状态，贪心策略就在于从树底自下而上遍历树（后序遍历），在叶子结点的父节点处放摄像头，这样摄像头能覆盖上中下范围最广，能尽可能节省摄像头。

六、堆与优先级队列

1. T347：前 K 个高频元素

思路：用优先级队列维护一个最小堆，这样保证堆顶元素一直是最小的（在本题意义是数字出现的频率最小），这样只要这个优先级队列的 Size 大于 k 就 pop()，把堆顶元素（频率最小）的元素弹出去，这样在优先级队列里面剩下的就一直是前 k 个高频元素了。最后要按照频率从高到低输出到数组中，因此还需要倒着把优先级队列堆顶元素取出来放到数组里（最先取出来的是 k 个高频元素中最低频的，应放在数组后面）

类似题 T692：前 K 个高频单词 （本题还要求频率相同，按字典序排序，这一步在定义优先级队列的比较准则 cmp 时写好频率相同时按照字符串的字典序升序排序即可）

回溯算法

回溯算法能解决如下问题：

- 组合问题：N个数里面按一定规则找出k个数的集合
- 排列问题：N个数按一定规则全排列，有几种排列方式
- 切割问题：一个字符串按一定规则有几种切割方式
- 子集问题：一个N个数的集合里有多少符合条件的子集
- 棋盘问题：N皇后，解数独等等

组合问题：

1. T77：组合　　（元素无重不可复选）

思路：关键在于 backtrack()递归函数的参数设置，需要传入一个 start 作为形参，代表开始搜索的起始位置，并在递归时更新它为+1，如搜完了“1”，接下来就去搜索选择“2”的组合。

2. T216：组合总和 III　　（元素无重不可复选）

思路：T77 是要收集所有元素个数为 k 的组合，而本题要收集元素个数为 k 且和为 n 的组合，同样需要一个 start 控制搜索的起始位置，并在递归时更新它为+1。同时将 sum 作为形参传入 backtrack()递归函数，一开始 sum 为 0，在 for 循环内，sum 也要进行 “做选择” sum+=i 和“撤销选择” sum-=i

3. T17：电话号码的字母组合　　（元素无重不可复选）

思路：本题是在两个集合里面分别选出一个字母构成结果字符串，因此不需要像前两题那样用一个 start 变量控制开始搜索的起始位置。关键在于用一个 index 变量控制 digits 的访问索引，当 index==digits.size()时就可以收集此时的 track，放到 res 中了

4. T39：组合总和　　（元素无重可复选）

想解决这种类型的问题，也得回到回溯树上，**我们不妨先思考思考，标准的子集/组合问题是如何保证不重复使用元素的**？

答案在于 `backtrack` 递归时输入的参数 `start`：

java　cpp　python　go　javascript

```java
// 注意：cpp 代码由 chatGPT 根据我的 java 代码翻译，旨在帮助不同背景的读者理解算法逻辑。
// 本代码不保证正确性，仅供参考。如有疑惑，可以参照我写的 java 代码对比查看。

// 无重组合的回溯算法框架
void backtrack(vector<int>& nums, int start) {
    for (int i = start; i < nums.size(); i++) {
        // ...
        // 递归遍历下一层回溯树，注意参数
        backtrack(nums, i + 1);
        // ...
    }
}
```

那么反过来，如果我想让每个元素被重复使用，我只要把 `i + 1` 改成 `i` 即可：

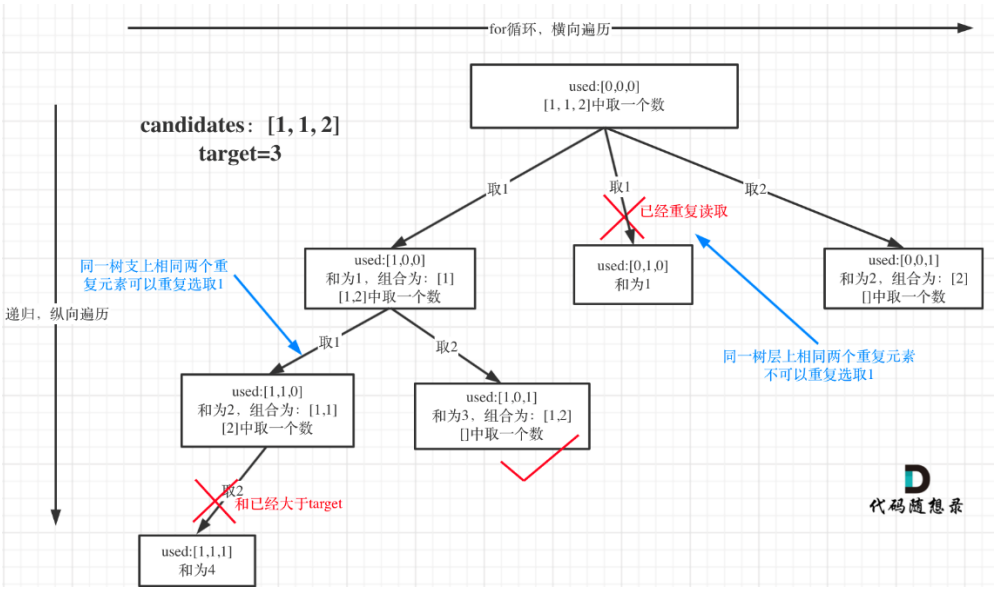
java　cpp　python　go　javascript

```java
// 注意：cpp 代码由 chatGPT 根据我的 java 代码翻译，旨在帮助不同背景的读者理解算法逻辑。
// 本代码不保证正确性，仅供参考。如有疑惑，可以参照我写的 java 代码对比查看。

// 可重组合的回溯算法框架
void backtrack(vector<int>& nums, int start) {
    for (int i = start; i < nums.size(); i++) {
        // ...
        // 递归遍历下一层回溯树，注意参数
        backtrack(nums, i);
        // ...
    }
}
```

5. T40：组合总和 II　　（元素可重不可复选）

体现在代码上，需要先进行排序，让相同的元素靠在一起，如果发现 `nums[i] == nums[i-1]`，则跳过：



6. T131：[分割回文串]

思路：子串分割问题本质上也是组合问题，要点在于确定分割线，如：aab 可以分割为 a|ab、aa|b、aab|，这个分割线所在的位置就是 start 索引处，然后 backtrack()函数中每次递归传入的参数都是 i+1，即 start 每次递归都往后移动一位，相当于分割线不断往右边移动。而 for 循环中对于每次递归固定好的分割线 start 的位置，[start,i]闭区间就是所要考虑的子串，通过一个函数判断这个子串 s[start..i]是不是回文串即可，若是，则放到 track 里面。

类似题：T93：[复原 IP 地址]

思路：本题跟上一题 T131 不同点在于，上一题是把字符串 s 切分成若干个合法的回文串，返回所有的切分方法。本题是把字符串 s 切分成 4 个合法的 IP 数字，返回所有的切分方法。因此在判断收集条件的时候需要多加一个条件：

```
if(start==n && track.size()==4)
{
    res.push_back(change(track));
    return;
}
```

先收集到 vector<string>的 track 里面，然后在通过一个函数 change 把它连成

需要的 IP 地址格式的字符串放到 res 里面。

离散化问题

1. AcWing T802 区间和

https://www.acwing.com/problem/content/804/

代码：关键在于利用二分法进行离散化，即把一个很大的区间中稀疏的一些位置值，题目是-10^9-10^9 的区间离散化成 0,1,2...的坐标，这样才有“头”，即 0 坐标处，才可以利用前后缀和之差去求区间和。

```
#include<bits/stdc++.h>
using namespace std;

//难!!!

vector<int> alls;  //存放所有待离散化的值，包含了需要操作的原始数轴上的位置，包括加数，查询操作的位置

//重点：利用二分实现离散化
int find(int x)
{
    //搜索左侧边界的二分搜索，找到第一次出现该元素的位置
    int l=0,r=alls.size()-1;
    while(l<=r)
    {
        int mid = l+(r-l)/2;
        if(alls[mid]>=x)
            r=mid-1;
        else
            l=mid+1;
    }
    return l;   //离散化后的坐标变成0,1,2,3...
}

int main()
{
    int n,m;
    cin>>n>>m;
    vector<pair<int,int>> add,query;
    vector<int> a(300010,0);
    vector<int> preSum(300010,0); //前缀和
```

```
31.
32.     for(int i=0;i<n;i++)
33.     {
34.         int x,c;
35.         cin>>x>>c;
36.         add.push_back({x,c});
37.         alls.push_back(x);
38.     }
39.
40.     //处理查询
41.     for(int i=0;i<m;i++)
42.     {
43.         int l,r;
44.         cin>>l>>r;
45.         query.push_back({l,r});
46.         alls.push_back(l);
47.         alls.push_back(r);
48.     }
49.
50.     //对alls去重（模板，背！）
51.     sort(alls.begin(),alls.end());
52.     alls.erase(unique(alls.begin(),alls.end()),alls.end());
53.
54.     for(auto item : add)
55.     {
56.         int x=find(item.first);  //找到它的离散化坐标
57.         a[x]+=item.second;   //离散化后的坐标处加上c(按题目要求)
58.     }
59.
60.     //构造前缀和数组
61.     preSum[0]=a[0];
62.     for(int i=1;i<=alls.size();i++)
63.     {
64.         preSum[i]=preSum[i-1]+a[i-1];
65.     }
66.
67.     //处理查询
68.     for(int i=0;i<query.size();i++)
69.     {
70.         int l=find(query[i].first),r=find(query[i].second);
71.         cout<<preSum[r+1]-preSum[l]<<endl;  //输出指定查询区间的区
    间和
72.     }
73.     return 0;
```

```
74.}
```