动态规划的套路：
从入门到精通到弃坑

wisdompeak

[Youtube Recording]

Disclaimer

今天，我们只讲套路。

不按套路出牌的DP问题，请不要问我（问我也不会），

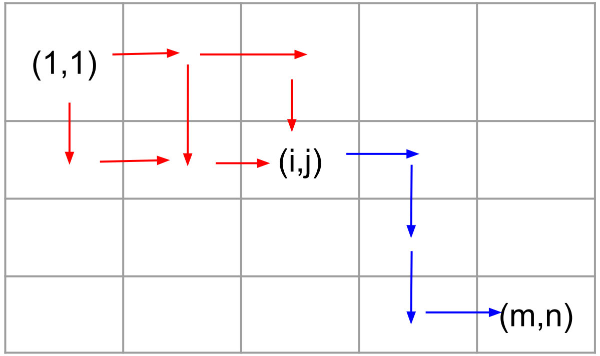
出门右转找崔日零一寒don苏。

残酷群积分比我高的同学可以下课了。

动态规划的思考艺术

有一个m x n大小的矩阵迷宫，每次移动只能向右或者向下，问从左上角到右下角共有多少种不同的走法？

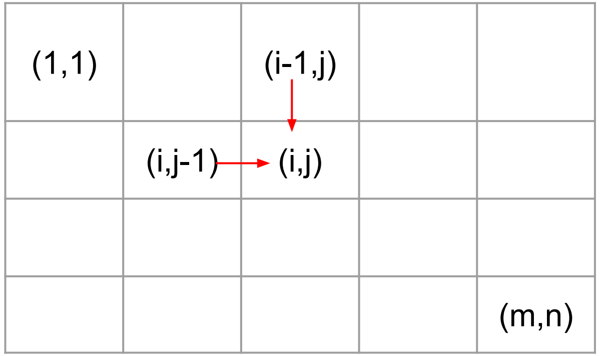
- **暴力枚举来计数为什么不优秀**
  - 从(1,1)到(m,n)的不同路径中有大量的重复。比如(1,1)->(i,j)有k条不同路径，那么 对于任何一条固定的(i,j)->(m,n)的路径，都需要走k遍来模拟。
  - 但是我们并不关心具体的走法。我 们只关心“状态”，也就是走法的数目。
  - 同理，如果我 们还知道(i,j)->(m,n)有t条不同路径，那么(1,1)->(i,j)->(m,n)的不同路径 总数就是k*s，而不需要真正了解怎么走的。



动态规划的思考艺术

有一个m x n大小的矩阵迷宫，每次移动只能向右或者向下，问从左上角到右下角共有多少种不同的走法？

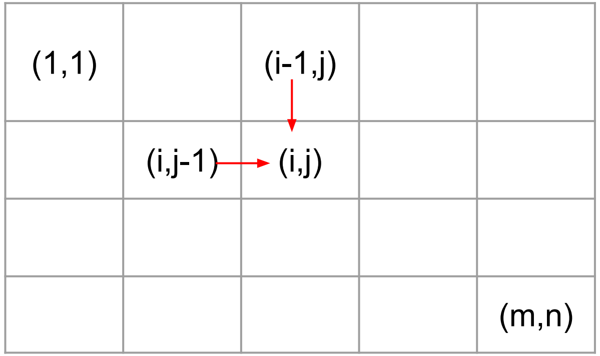
- **动态规划的美**
  - 令$f(i,j)$表示从$(1,1)$->(i,j)的不同路径数目。则$f(i,j)=f(i-1,j)+f(i,j-1)$
  - 我们发现想求出$f(i,j)$，只需要知道几个参数更小的 $f(i',j')$。我们将求解$f(i',j')$称作求解$f(i,j)$的“子问题”。
  - 我们舍弃了冗余信息（具体的走法）。我 们只记录了对解决问题有帮助的信息$f(i,j)$.



动态规划的思考艺术

动态规划的两大特点（适用前提）

- **无后效性：**
  - 一旦f(i,j)确定，就不用关心“我们如何计算出f(i,j)”。
  - 想要确定f(i,j)，只需要知道f(i-1,j)和f(i,j-1)的值，而至于它们是如何算出来的，对当前或之后的任何子问题都没有影响。
  - “过去不依赖将来，将来不影响过去”——智巅语录
- **最优子结构：**
  - f(i,j)的定义就已经蕴含了“最优”。
  - 大问题的最优解可以由若干个小问题的最优解推出。(max, min, sum...)

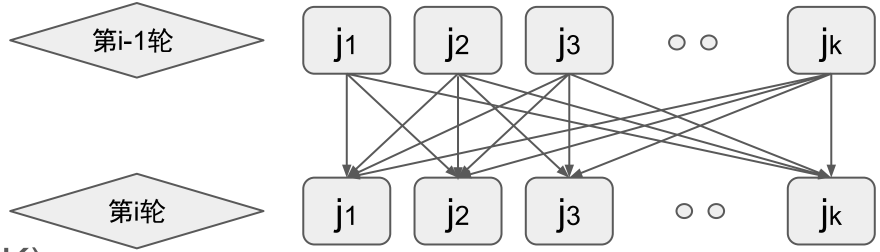


DP能适用的问题：能将大问题拆成几个小问题，且满足无后效性、最优子结构性质。

DP套路(I): 第I类基本型（“时间序列”型）

给出一个序列（数组/字符串），其中每一个元素可以认为“一天”，并且“**今天**”的状态只取决于“**昨天**”的状态。

- House Robber
- Best Time to Buy and Sell Stocks
- ...

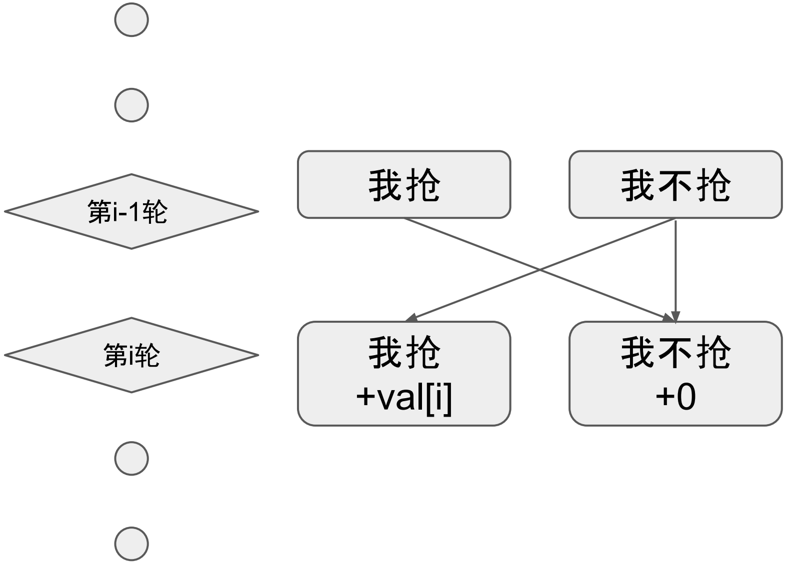


套路：

- 定义dp[i][j]：表示第i-th轮的第j种状态 (j=1,2,...,K)
- 千方百计将dp[i][j]与前一轮的状态dp[i-1][j]产生关系(j=1,2,...,K)
- 最终的结果是dp[last][j]中的某种aggregation (sum, max, min …)

LC 198. House Robber

给一排房子，相邻的房子不能都抢。问最多能抢的价值。



0:这轮我抢的最大收益
1:这轮我不抢的最大收益

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

Ans = max (dp[N][0], dp[N][1])
```

LC 213. House Robber II

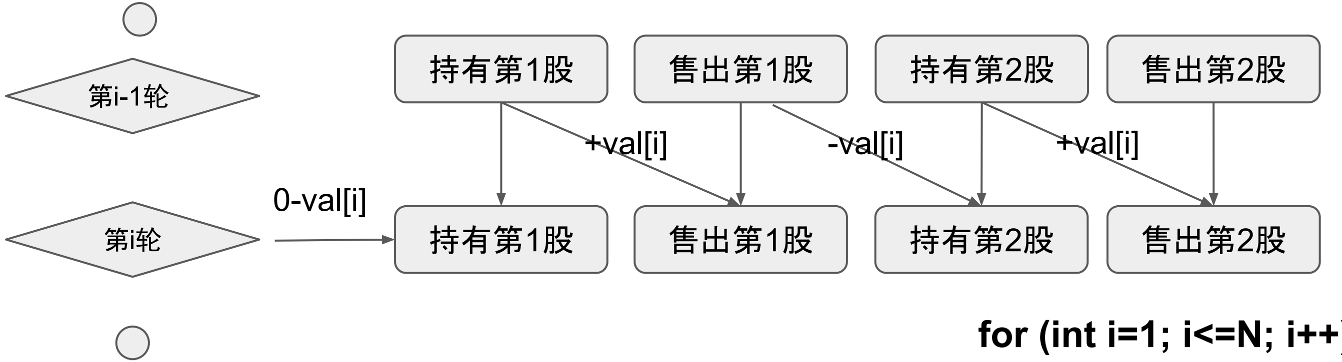
给一圈“首尾相连”的房子，相邻的房子不能都抢。问最多能抢的价值。

Trick：首位和末位不能同时抢，这说明至少有一个不能抢。

1. 考虑首位的房子我不抢，那么对于house[1]~house[last]就是一个基本的House Robber问题。
2. 考虑末位的房子我不抢，那么对于house[0]~house[last-1]就是一个基本的House Robber问题。

LC 123.Best Time to Buy and Sell Stock III

给一系列每日股票的价格。每日只能买入或卖出或不操作。最多交易两次。问最大的收益。



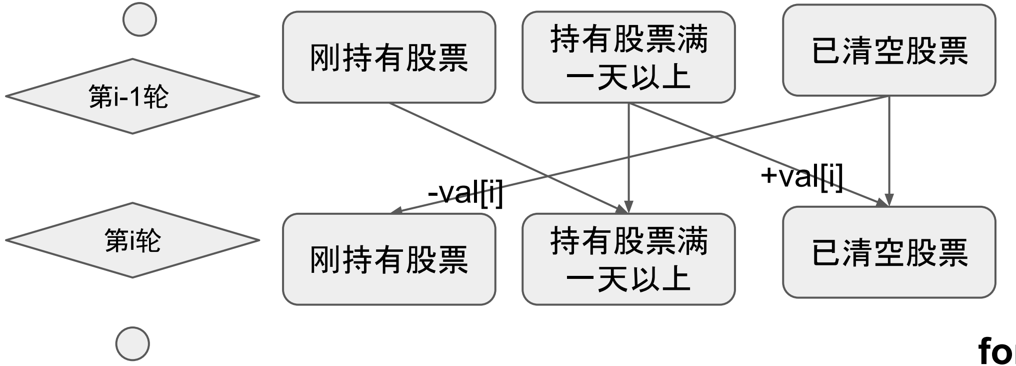
0:这轮我已持有第1股的最大收益
1:这轮我已售出第1股的最大收益
2:这轮我已持有第2股的最大收益
3:这轮我已售出第2股的最大收益

```
for (int i=1; i<=N; i++)
  {
      dp[i][0] = max(dp[i-1][0], -val[i]);
      dp[i][1] = max(dp[i-1][1], dp[i-1][0]+val[i])
      dp[i][2] = max(dp[i-1][2], dp[i-1][1]-val[i])
      dp[i][3] = max(dp[i-1][3], dp[i-1][2]+val[i])
  }
```

Ans = max {dp[N][i]} (i=0,1,2,3)

LC 309.Best Time to Buy and Sell Stock with Cooldown

给一系列每日股票的价格。每日只能买入或卖出或不操作，买入后要隔一天才能卖出。无总交易次数限制。问最大的收益。



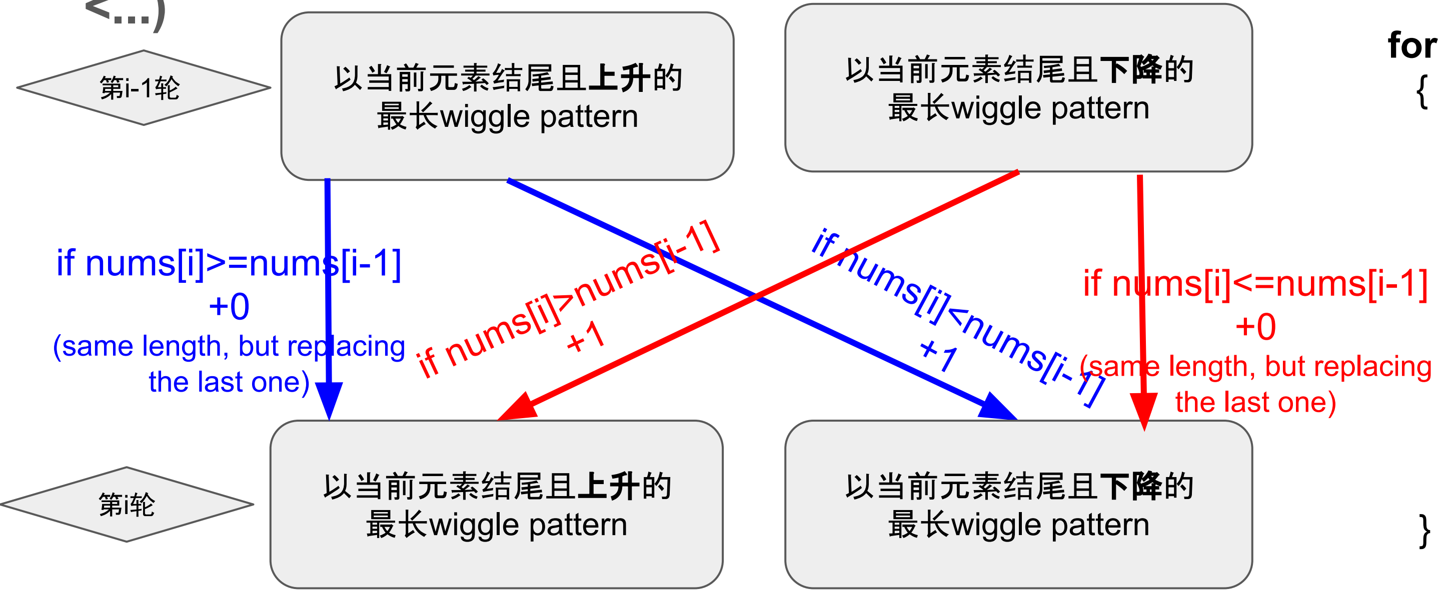
0:这轮我刚持有股票的最大收益
1:这轮我已持有股票一天以上的最大收益
2:这轮我已清空股票的最大收益

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

Ans = max {dp[N][i]} (i=0,1,2,3)

LC 376.Wiggle Subsequence

给一个序列s，求其中最长的wiggle pattern subsequence. (...> s[i] < s[j] > s[k] <...)



0:以当前元素结尾且**上升**
1:以当前元素结尾且**下降**

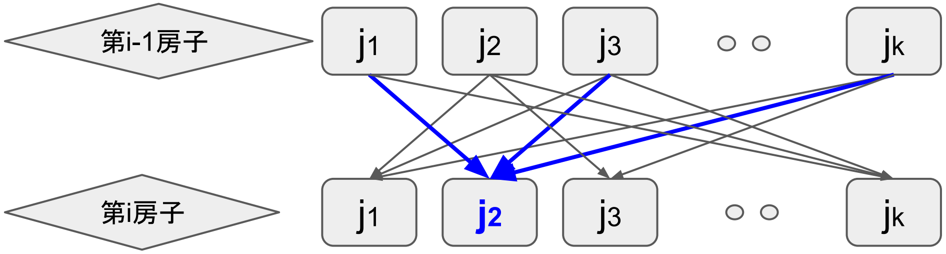
```
for (int i=1; i<=N; i++)
  {
     if (nums[i] < nums[i-1])
       dp[i][0] = dp[i-1][1] + 1;
      else
        dp[i][0] = dp[i-1][0];

     if (nums[i] < nums[i-1])
       dp[i][1] = dp[i-1][0] + 1;
     else
        dp[i][1] = dp[i-1][1];
  }
```

Ans = max {dp[N][i]} (i=0,1)

LC 276. Paint Fence

给出cost[i][j]表示第i个房子喷涂第j种颜色的价格。相邻的房子不能涂同一种颜色。求喷涂所有房子的最小价格。



dp[i][j]:第i间房子喷涂第j种颜色的代价

```
for (int i=1; i<=N; i++)
  {
      dp[i][j] = min { dp[i-1][j’] } + cost[i][j];
                 (j’=1,2,3..,K but j’!=j)
  }
```

Ans = min {dp[N][j]} (i=0,1,2,...,K)

LC 1289. Minimum Falling Path Sum II 一毛一样

思考题

给N个房子，涂白色和涂黑色的花费分别是a,b。要求不能有连续三间房子涂同一种颜色。求喷涂所有房子的最小价格。

思考题

给N个房子，涂白色和涂黑色的花费分别是a,b。要求不能有连续三间房子涂同一种颜色。求喷涂所有房子的最小价格。

0: 结尾有连续1间为黑色 <= 2,3 + black

1: 结尾有连续2间为黑色 <= 0 + black

2: 结尾有连续1间为白色 <= 0,1 + white

3: 结尾有连续2间为白色 <= 2 + white

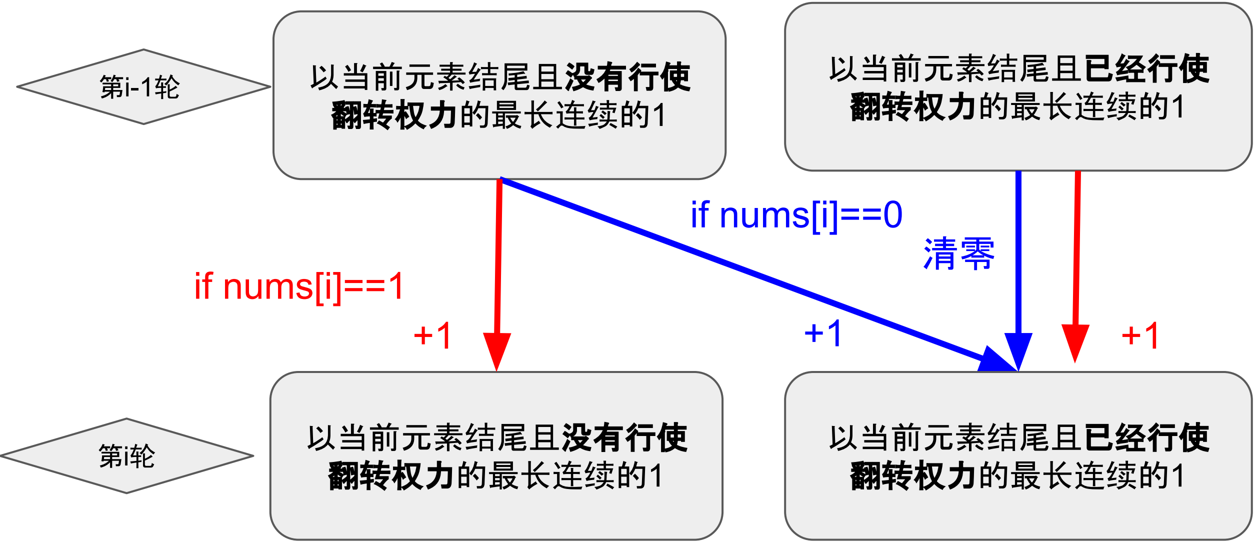
To Do or Not To Do

很多不是那么套路的DP题，DP状态可能比较难设计。不过还是有套路可循。

某些题目给你一次“**行使某种策略的权力**”。联想到买卖股票系列的题，我们常会设计的两个状态就是“**行使了权力**”和“**没有行使权力**”分别对应的价值。

LC 487. Max Consecutive Ones II

给你一个0/1数组。有最多一次从0翻转到1的权力。问最多可以多少连续的1？



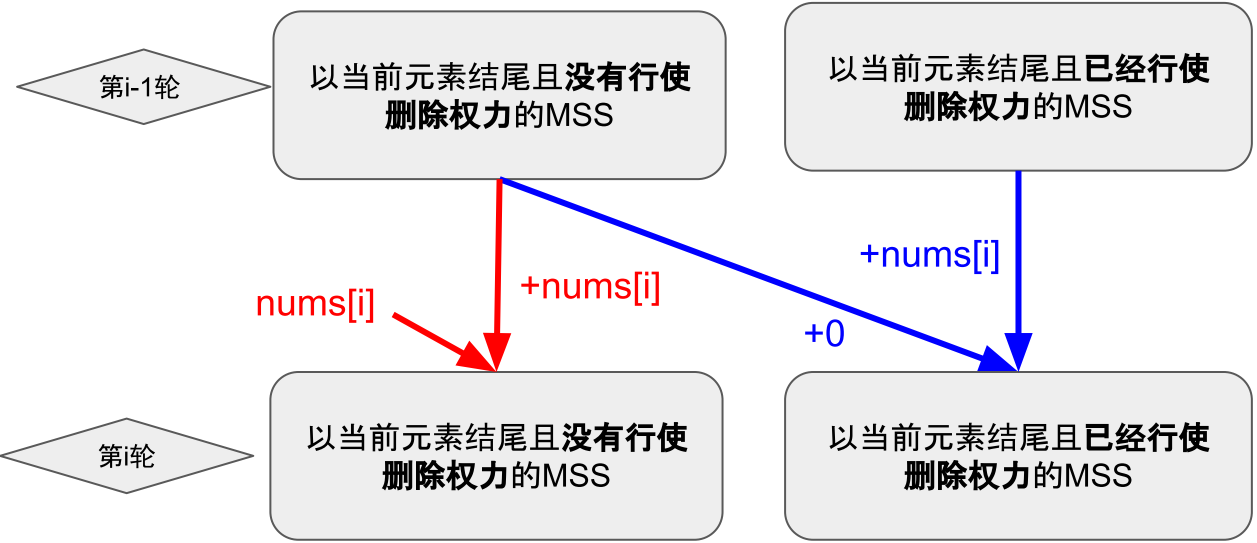
0:以当前元素结尾且**没有行使翻转权力**的最长连续的1
1:以当前元素结尾且**已经行使翻转权力**的最长连续的1

```
for (int i=1; i<=N; i++)
{
      if (nums[i]==0) {
         dp[i][1] = dp[i-1][0]+1;
         dp[i][0] = 0;
      }
      else {
         dp[i][0] = dp[i-1][0]+1;
         dp[i][1] = dp[i-1][1]+1;
      }
   }
```

Ans = max {dp[i][j]}
(for all possible i and j=0,1)

LC 1186. Maximum Subarray Sum with One Deletion

给你一个数组。有最多一次删除一个数的权力。问sum最大的subarray？



for (int i=1; i<=N; i++)
{
dp[i][0] = max(dp[i-1][0] + nums[i], nums[i]);

dp[i][1] = max(dp[i-1][0], dp[i-1][1] + nums[i]);
}

Ans = max {dp[i][j]}
(for all possible i and j=0,1)

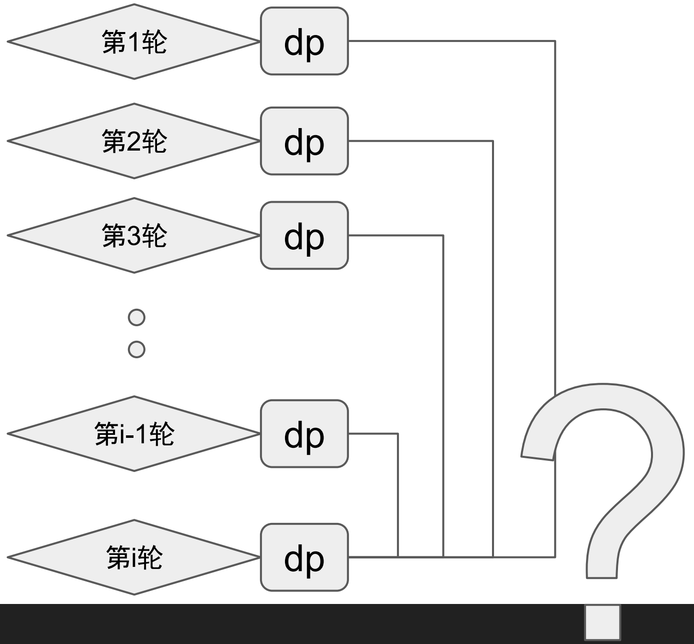
0:以当前元素结尾且**没有行使删除权力**的MSS
1:以当前元素结尾且**已经行使删除权力**的MSS

DP套路(II): 第II类基本型（“时间序列”加强版）

给出一个序列（数组/字符串），其中每一个元素可以认为“一天”：但“**今天**”的状态 和之前的“**某一天**”有关，需要挑选。

套路：

- 定义dp[i]：表示第i-th轮的状态，一般这个状态要求和元素i直接有关。
- 千方百计将dp[i]与之前的状态dp[i']产生关系(i=1,2,...,i-1) (比如sum, max, min)
    - dp[i]肯定不能与大于i的轮次有任何关系，否则违反了DP的无后效性。
- 最终的结果是dp[i]中的某一个



LC 300. Longest Increasing Subsequence

给一个数组s。求最长的递增子序列的长度。

状态定义：照抄问题 dp[i] => s[1:i]里以s[i]结尾的、最长的递增子序列的长度。

状态的转移：寻找最优的前驱状态j，将dp[i]与dp[j]产生联系。

```
for (int i=1; i<=N; i++)
{
   // i是该LIS的最大元素，搜索该LIS的第二大元素j
   for (int j=1; j<i; j++)
   {
      if (nums[j]<nums[i])
         dp[i] = max (dp[i], dp[j] + 1);
   }
}
```

Ans = max {dp[i]}, for i=1,2,...,N

<u>思考题</u>

<u>:673.Number-of-Longest-Increasing-Subsequence</u>

if (dp[j]+1==dp[i]) count[i]+=count[j];

else if (dp[j]+1>dp[i]) dp[i]=dp[j]+1, count[i] = count[j];

LC 368. Largest Divisible Subset

给一个数组s。求最大的子集，使得里面的所有元素之间都可以互相整除。

状态定义：照抄问题 dp[i] => s[1:i]里以s[i]结尾的、满足题意的最大子集的元素数目。

状态的转移：寻找最优的前驱状态j，将dp[i]与dp[j]产生联系。

```
sort(nums);
for (int i=1; i<=N; i++)
{
   // i是该集合的最大元素，搜索该子集的第二大元素j
   for (int j=1; j<i; j++)
   {
      if (nums[i]%nums[j]==0)
         dp[i] = max (dp[i], dp[j] + 1);
   }
}
```

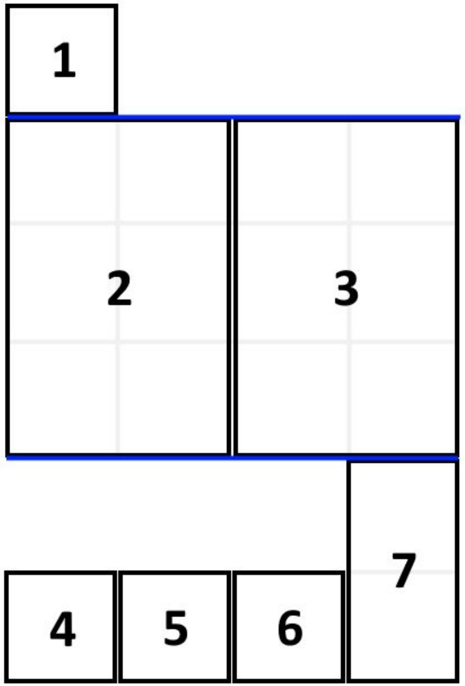
XXX j XXX i

Ans = max {dp[i]}, for i=1,2,...,N

LC 1105. Filling Bookcase Shelves

给你N本书（宽度高度各异）的序列要求按照所给顺序摆放。相邻的若干本可以放一层，但同一层的宽度不能超过W。

问这个书架最矮可以有多高？



LC 1105. Filling Bookcase Shelves

将数组S分成若干个subarray，最小化“每个subarray的最大值之和”，输出该值。

状态定义：照抄问题 dp[i] => 将数组S[1:i]分成若干个subarray，最小化“每个subarray的最大值之和”，保存该值。

状态的转移：寻找最优的前驱状态j，将dp[i]与dp[j]产生联系。

- 第i本书所在的这一层可能有多高？取决于上一层的最后一本书在哪里。

```
for (int i=1; i<=N; i++)
{
    // i是本层的最后一本，搜索上层的最后一本的位置
    for (int j=i-1; j>=1; j--)
    {
        if (totalWidth[j+1:i] <= W)
            dp[i] = min (dp[i], dp[j] + maxHeight[j+1:i] );
        else
            break;
    }
}
Ans = dp[N]
```

DP套路(III): 双序列型

给出两个序列s和t（数组/字符串），让你对它们搞事情。

- Longest Common Subsequences
- Shortest Common Supersequence
- Edit distances
- ...

套路：

- 定义dp[i][j]：表示针对s[1:i]和t[1:j]的子问题的求解。
- 千方百计将dp[i][j]往之前的状态去转移：dp[i-1][j], dp[i][j-1], dp[i-1][j-1]
- 最终的结果是dp[m][n]

LC1143: Longest Common Subsequences

Q: 求字符串s和t的length of LCS

状态定义：照抄问题 dp[i][j] => s[1:i]和t[1:j]的length of LCS

状态的转移：外面两层大循环遍历i和j；核心从s[i]与t[j]的关系作为突破口，拼命往dp[i-1][j-1], dp[i][j-1], dp[i-1][j]转移。

```
for (int i=1; i<=m; i++)
  for (int j=1; j<=n; j++)
  {
     if (s[i]==t[j])
        dp[i][j] = dp[i-1][j-1]+1;
     else
        dp[i][j] = max(dp[i-1][j],dp[i][j-1]);
  }
```

```
s: XXXXX i
t:  YYY j
```

LC1092: Shortest Common Supersequences

Q: 求字符串s和t的length of SCS

状态定义：照抄问题 dp[i][j] => s[1:i]和t[1:j]的length of SCS

状态的转移：外面两层大循环遍历i和j；核心从s[i]与t[j]的关系作为突破口，拼命往dp[i-1][j-1], dp[i][j-1], dp[i-1][j]转移。

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

s: XXXXX i
t: YYY j

LC72: Edit Distance

Q: 求字符串s和t的min Edit Distance

状态定义：照抄问题 dp[i][j] => s[1:i]和t[1:j]的Min Edit Distance

状态的转移：外面两层大循环遍历i和j；核心从s[i]与t[j]的关系作为突破口，拼命往dp[i-1][j-1], dp[i][j-1], dp[i-1][j]转移。

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

s: XXXXX i
t: YYY j

LC97: Interleaving String

Q: 求字符串s和t能否交叠组成字符串w

状态定义：照抄问题 dp[i][j] => s[1:i]和t[1:j]能否交叠组成字符串w[1:i+j]

状态的转移：外面两层大循环遍历i和j；核心从s[i], t[j] 和w[i+j]的关系作为突破口，拼命往dp[i-1][j-1], dp[i][j-1], dp[i-1][j]转移。

```
for (int i=1; i<=m; i++)
  for (int j=1; j<=n; j++)
  {
     if (s[i]==w[i+j] && dp[i-1][j]==true)
        dp[i][j] = true;
     else if (t[j]==w[i+j] && dp[i][j-1]==true)
        dp[i][j] =1;
     else
        dp[i][j] = false;
  }
```

s: XXXXX i
t: YYY j

LC115. Distinct Subsequences

Q: 求字符串s有多少不同的子序列等于字符串t

状态定义：照抄问题 dp[i][j] => s[1:i]有多少不同的子序列等于t[1:j]

状态的转移：外面两层大循环遍历i和j；核心从s[i]与t[j] 的关系作为突破口，拼命往dp[i-1][j-1], dp[i][j-1], dp[i-1][j]转移。

```
for (int i=1; i<=m; i++)
  for (int j=1; j<=n; j++)
  {
     if (s[i]==t[j])
        dp[i][j] = dp[i-1][j-1] + dp[i-1][j];
     else
        dp[i][j] = dp[i-1][j];
  }
```

```
s: XXXXX i
t:  YYY j
```

LC 727.Minimum Window Subsequence

Q: 求字符串s里最短的、并且包含t的substring的长度。(t是这个substring的子序列)

状态定义:照抄问题 dp[i][j] => s[1:i]里最短的、并且包含t[1:j]的substring的长度(这个substring必须要求以s[i]结尾)

状态的转移:外面两层大循环遍历i和j;核心从s[i]与t[j] 的关系作为突破口, 拼命往dp[i-1][j-1], dp[i][j-1], dp[i-1][j]转移。

```
for (int i=1; i<=m; i++)
  for (int j=1; j<=n; j++)
  {
     if (s[i]==t[j])
       dp[i][j]=dp[i-1][j-1]+1;
     else
       dp[i][j]=dp[i-1][j]+1;
  }
```

s: XXXXX i t: YYY j

注意:ans = min dp[i][n] (i=1,2,...m)

LCS/SCS的变种：换汤不换药

- LC 583. Delete Operation for Two Strings

问：从字符串s和t中总共最少删除多少个字符能使得它们相等。

- LC 712. Minimum ASCII Delete Sum for Two Strings

问：从字符串s和t中总共最少删除多少ASCII码值的字符能使得它们相等。

- LC 1035. Uncrossed Lines

两个数组s和t之间相等的数字可以连线。连线不能交叉。问最多可以有几条连线。

LCS/SCS的变种：换汤不换药

- LC 1216. Valid Palindrome III

问一个字符串s最少删除多少个字符能变成回文串。

- LC 1312. Minimum Insertion Steps to Make a String Palindrome

问一个字符串s最少需要添加多少个字符能变成回文串。

T = S[:-1]

S

如何重构出DP计算的最优方案？

Solution: 根据状态转移的正过程“回溯”逆推。

```
   ZZZZZZZZZZZ?
s: XXXXX i
t:  YYY j
```

- LC1092: Shortest Common Supersequences

```
for (int i=1; i<=m; i++)
  for (int j=1; j<=n; j++)
  {
     if (s[i]==t[j])
        dp[i][j] = dp[i-1][j-1]+1;
        // 说明了我们往SCS上添加了s[i] / t[j]
     else
        dp[i][j] = max(dp[i-1][j]+1, dp[i][j-1]+1);
       // 前者说明了我们往SCS上添加了s[i]
       // 后者说明了我们往SCS上添加了t[j]
  }
```

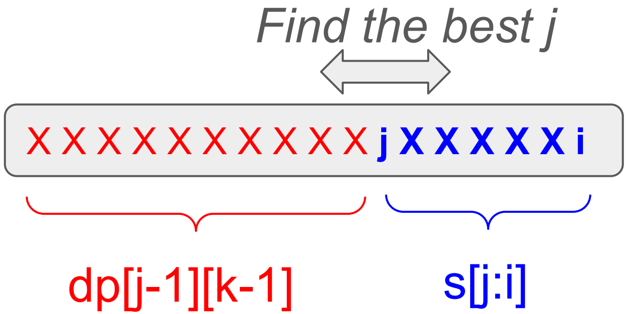
```
int i=M;
int j=N;
string result;
while (i>0 && j>0)
{
   if (str1[i]==str2[j])
   {
      result.push_back(s[i]);
      i--; j--;
   }
   else if (dp[i][j]==dp[i-1][j]+1)
   {
      result.push_back(s[i]);
      i--;
   }
   else {
      result.push_back(t[j]);
      j--;
   }
}
```

DP套路(IV): 第I类区间型DP

给出一个序列，明确要求**分割成K个连续区间**，要你计算这些区间的某个最优性质。

套路：

- 状态定义：dp[i][k]表示针对s[1:i]分成k个区间，此时能够得到的最优解
- 搜寻**最后一个区间的起始位置j**，将dp[i][k]分割成dp[j-1][k-1]和s[j:i]两部分。
- 最终的结果是dp[N][K]



LC 1278. Palindrome Partitioning III

求最小的字符变动，使得字符串S能够恰能分成K个子串，且每串都是回文串。

状态定义：照抄问题 dp[i][k] => 最小的字符变动，使得字符串S[1:i]能够恰能分成k个子串，且每串都是回文串。

状态的转移：

- 第一层循环遍历i
- 第二层循环遍历k
- 第三层循环寻找最优的位置j作为最后一个分区的起始位置。
- 将dp[i][k]分割成dp[j-1][k-1]和s[j:i]求解。

```
for (int i=1; i<=n; i++)
    for (int k=1; k<=min(i,K); k++)
    {
       for (int j=i; j>=k; j--)
       {
         dp[i][k] = min(dp[i][k],
                        dp[j-1][k-1] + count[j:i]);
       }
    }
```

k不能太大，否则不够i个元素分

j不能太小，否则分不够k-1组

X X X X X X X X X X j X X X X X i

dp[j-1][k-1]　　s[j:i]

Ans = dp[N][K]

注意边界条件：
dp[x][0], dp[0][0]

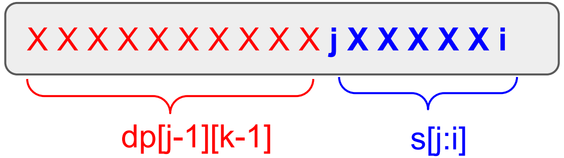
LC 813. Largest Sum of Averages

将数组S分成最多K个subarray，最大化“每个subarray平均数的sum”，输出该值。

状态定义：照抄问题 dp[i][k] => 将数组S[1:i]分成k个subarray，最大化“每个subarray平均数的和”，保存该值。

状态的转移：

- 第一层循环遍历i
- 第二层循环遍历k
- 第三层循环寻找最优的位置j作为最后一个分区的起始位置。
- 将dp[i][k]分割成dp[j-1][k-1]和s[j:i]求解。




```
for (int i=1; i<=n; i++)
    for (int k=1; k<=min(i,K); k++)
    {
      for (int j=i; j>=k; j--)
      {
        dp[i][k] = max (dp[i][k],
                        dp[j-1][k-1] + avg[j:i]));
      }
    }
```

Ans = max {dp[N][k]},
for k=1,2,...,K

注意边界条件：
dp[x][0], dp[0][0]

LC 410. Split Array Largest Sum

将数组S分成K个subarray，最小化“其中最大的subarray sum”，输出该值。

状态定义：照抄问题 dp[i][k] => 将数组S[1:i]分成k个subarray，最小化“其中最大的subarray sum”，保存该值。

状态的转移：

- 第一层循环遍历i
- 第二层循环遍历k
- 第三层循环寻找最优的位置j作为最后一个分区的起始位置。
- 将dp[i][k]分割成dp[j-1][k-1]和s[j:i]求解。

```
for (int i=1; i<=n; i++)
    for (int k=1; k<=min(i,K); k++)
    {
        for (int j=i; j>=k; j--)
        {
            dp[i][k] = min (dp[i][k],
                            max(dp[j-1][k-1], sum[j:i]));
        }
    }
```

X X X X X X X X X X j X X X X X i

dp[j-1][k-1]　　s[j:i]

Ans = dp[N][K]

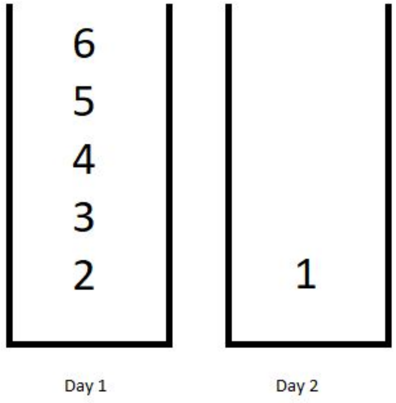
注意边界条件：
dp[x][0], dp[0][0]

LC 1335. Minimum Difficulty of a Job Schedule

有一系列task及其难度，必须顺次完成，且必须恰好分d天完成。

每天的难度定义为当天所有task难度的最大值。

求如何安排task，最小化“每天难度的总和”，输出该值。



LC 1335. Minimum Difficulty of a Job Schedule

将数组S分成K个subarray，最小化“每个subarray最大值的和”，输出该值。

状态定义：照抄问题 dp[i][k] => 将数组S[1:i]分成k个subarray，最小化“每个subarray最大值的和”，保存该值。

状态的转移：

- 第一层循环遍历i
- 第二层循环遍历k
- 第三层循环寻找最优的位置j作为最后一个分区的起始位置。
- 将dp[i][k]分割成dp[j-1][k-1]和s[j:i]求解。

```
for (int i=1; i<=n; i++)
     for (int k=1; k<=min(i,K); k++)
     {
        for (int j=i; j>=k; j--)
        {
          dp[i][k] = min (dp[i][k],
                               dp[j-1][k-1] + Max[j:i]));
         }
     }
```

$\underbrace{\text{X X X X X X X X X X}}_{\text{dp[j-1][k-1]}}\ \text{j}\ \underbrace{\text{X X X X X i}}_{\text{s[j:i]}}$

Ans = dp[N][K]

注意边界条件：
dp[x][0], dp[0][0]

DP套路(V): 第II类区间型DP

只给出一个序列S（数组/字符串），求一个针对这个序列的最优解。

适用条件：这个最优解对于序列的index而言，没有“无后效性”。即无法设计dp[i]使得dp[i]仅依赖于dp[j] (j<i). 但是大区间的最优解，可以依赖小区间的最优解。

套路：

- 定义dp[i][j]：表示针对s[i:j]的子问题的求解。
- 千方百计将大区间的dp[i][j]往小区间的dp[i’][j’]转移。
  - 第一层循环是区间大小；第二层循环是起始点。
- 最终的结果是dp[1][N]

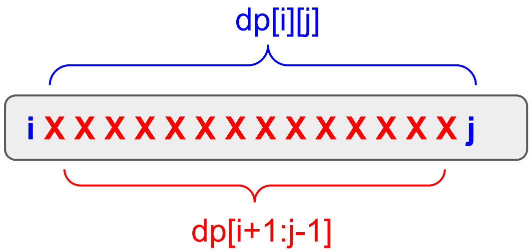
LC 516. Longest Palindromic Subsequence

给一个字符串S，求是回文串的最长subsequence的长度。

状态定义：照抄问题 dp[i][j] => 字符串S[i:j]里是回文串的最长subsequence的长度。

状态的转移：

- 第一层循环是区间大小。
- 第二层循环是起始点。
- 千方百计将大区间的dp[i][j]往小区间的dp[i’][j’]转移。




```
for (int len=1; len<=N; len++)
    for (int i=1; i+len-1<=N; i++)
    {
        int j = i+len-1;
        if (len==1)
            dp[i][j] = 1;
        else if (s[i]==s[j])
            dp[i][j] = dp[i+1][j-1]+2;
        else
            dp[i][j] = max(dp[i][j-1],dp[i+1][j]);
    }
```

Ans = dp[1][N]

注意边界条件：
dp[i][j] = 0 (i>j)

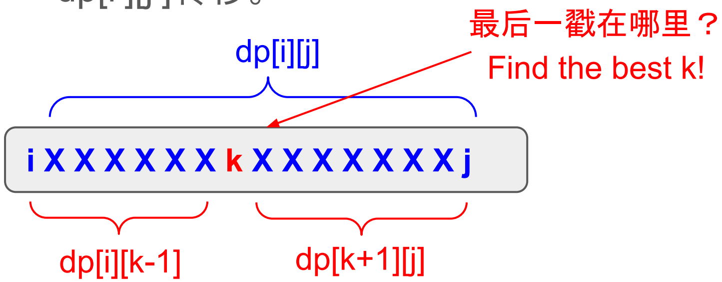
LC 312. Burst Balloons

给一排气球及其价值S。每戳爆一个气球的得分：气球本身分值*(仍存留的)左边气球分值*(仍存留的)右边气球分值。如何戳爆所有气球，最大化总得分。

状态定义：照抄问题 dp[i][j] => 戳爆S[i:j]的所有气球，最大化的总得分。

状态的转移：

- 第一层循环是区间大小。
- 第二层循环是起始点。
- 千方百计将大区间的dp[i][j]往小区间的dp[i'][j']转移。




```
for (int len=1; len<=N; len++)
    for (int i=1; i+len-1<=N; i++)
    {
        int j = i+len-1;
        for (int k=i; k<=j; k++)
            dp[i][j] = max(dp[i][j],
                    dp[i][k-1]+s[i-1]*s[k]*s[j+1]+dp[k+1][j]);
    }
```

Ans = dp[1][N]

注意边界条件：
dp[i][j] = 0 (i>j)

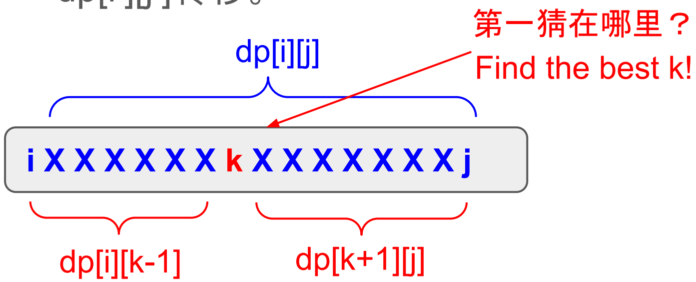
LC 375. Guess Number Higher or Lower II

从1~N的数字里猜数。如果你猜一个数x，需要付钱x块，并且会得到反馈信息x是大是小。问最少付多少钱能保证猜中。

状态定义：照抄问题 dp[i][j] => 最少付多少钱能保证猜中i~j里的数字。

状态的转移：

- 第一层循环是区间大小。
- 第二层循环是起始点。
- 千方百计将大区间的dp[i][j]往小区间的dp[i'][j']转移。




```
for (int len=1; len<=N; len++)
    for (int i=1; i+len-1<=N; i++)
    {
        int j = i+len-1;
        for (int k=i; k<=j; k++)
            dp[i][j] = min(dp[i][j],
                    k + max(dp[i][k-1],dp[k+1][j]);
    }
```

Ans = dp[1][N]

注意边界条件：
dp[i][j] = 0 (i>=j)

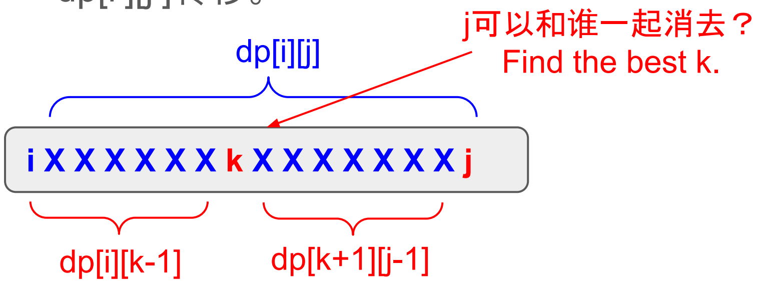
LC 1246. Palindrome Removal

给一个字符串s，每次删除其中的一个回文substring，问多少次删完？

状态定义：照抄问题 dp[i][j] => 对于字符串s[i:j]，每次删除其中的一个回文substring，问多少次删完。

状态的转移：

- 第一层循环是区间大小。
- 第二层循环是起始点。
- 千方百计将大区间的dp[i][j]往小区间的dp[i'][j']转移。




```
for (int len=1; len<=N; len++)
    for (int i=1; i+len-1<=N; i++)
    {
        int j = i+len-1;
        for (int k=i; k<=j; k++)
        {
            if (s[k] == s[j])
                dp[i][j] = min(dp[i][j],
                               dp[i][k-1] + max(1, dp[k+1][j-1]));
        }
    }
```

Ans = dp[1][N]

注意边界条件：
dp[i][j] = 0 (i>=j)

结合第I类和第II类区间型DP算法的Boss题：
LC 1000. Minimum Cost to Merge Stones

给一个数组代表N堆石头的重量。每步操作将K堆相邻的石头合并，代价是这K堆的重量和。问最少的代价将所有的石头堆合并到一起。

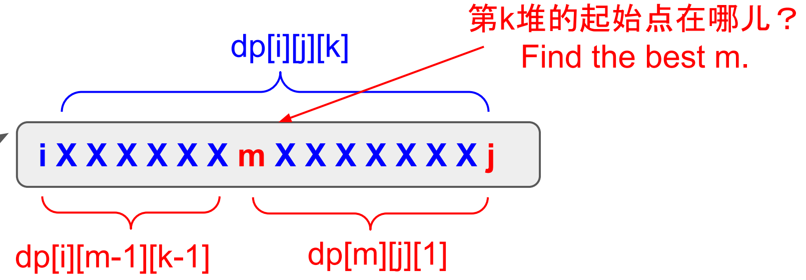
我们考虑将任意区间[i:j]归并到一起的最优解，取决于如何先最小代价地将[i:j]归并成K堆（即先分成K个subarray），然后再加sum[i:j]即可。于是提示我们需要结合两类区间型DP的套路：

dp[i][j][k]表示将区间[i:j]归并成k堆的最小代价。

结合第I类和第II类区间型DP算法的Boss题：
LC 1000. Minimum Cost to Merge Stones

dp[i][j][k]表示将区间[i:j]归并成k堆的最小代价。

```
for (int len = 1; len<=N; len++)
  for (int i=1; i+len-1<=N; i++)
  {
    int j = i+len-1;
    for (int k=2; k<=K; k++)
    {
      for (int m=i; m<=j; m++)
        dp[i][j][k] = min(dp[i][m-1][k-1]+dp[m][j][1]) ;
    }
    dp[i][j][1] = dp[i][j][K]+sum[i:j];
  }
return dp[1][N][1];
```



特别注意：当k=1时，dp[i][j][1]只能由dp[i][j][K]转化而来。

DP套路(VI): 背包入门

题型抽象：给出N件物品，**每个物品可用可不用**（或者有若干个不同的用法）。要求以某个有上限C的代价来实现最大收益。（有时候反过来，要求以某个有下限的收益来实现最小代价。）

套路：

- 定义dp[i][c]：表示考虑只从前i件物品的子集里选择、代价为c的最大收益。
  - c = 1,2,...,C
- 千方百计将dp[i][c]往dp[i-1][c’]转移：即考虑如何使用物品i，对代价/收益的影响
  - 第一层循环是物品编号i；
  - **第二层循环是遍历“代价”的所有可能值。**
- 最终的结果是 max {dp[N][c]}, for c=1,2,...,C

DP套路(VI): 背包入门

题型抽象：给出N件物品，**每个物品可用可不用**（或者有若干个不同的用法）。要求以某个有上限C的代价来实现最大收益。（有时候反过来，要求以某个有下限的收益来实现最小代价。）

背包问题的解法特点：

- 利用了物品次序的“无后效性”：我在前4件物品中做选择的最大收益，与第5件物品是啥没有关系。
    - **“过去不依赖将来，将来不影响 过去”**
- 将原本题意的解空间（代表各种物品是否使用的高维向量），替换成了代价的解空间（是一个有上限C的标量）。压缩了复杂度。
    - [0,2,0,3,1,0,0,4] => {10}

标准01背包

给一系列物品(价值为v，重量为w)，每个物品只能用一次。背包总容量上限是C。问最大能装多少价值的东西。

状态定义:dp[i][c] => 考虑仅在前i件物品的子集选择，且**代价(所选物品的总重量)恰好是c时**能得到的最大收益。

状态转移:当前的代价是c，那么前一轮的代价是多少？c 或者 c-Wi

```
for (int i=1; i<=N; i++)
    for (int c=1; c<=C; c++)
    {
        dp[i][c] = max(dp[i-1][c],
                       dp[i-1][c-wi] + vi );
    }
```

Ans = max {dp[N][c] } for c=1,2,...,C

注意边界条件:
dp[0][0] = 0
dp[0][c] = NA

有点像DP的基本款:
考虑一件物品就是“一轮”;
只不过每轮的状态以代价为单位，有C种。

LC 494. Target Sum

给你一个数组nums，可以在每个元素前添加正负号。问使得最后整体结果是S的方法有多少个？

状态定义：dp[i][s] => 考虑仅在前i个元素的子集中添加符号，所得估值恰好是s时有多少种不同的方法。

状态转移：当前的估值是s，那么前一轮的估值是多少？s-nums[i]或者s+nums[i].

```
for (int i=1; i<=N; i++)
    for (int s=-MAX_SUM; s<=MAX_SUM; s++)
    {
        dp[i][s] = dp[i-1][s-nums[i]] +
                   dp[i-1][s+nums[i]];
    }
```

Ans = dp[N][S]

需要查验 s-nums[i] and s+nums[i] 是否越界

注意边界条件：
dp[0][0] = 1
dp[0][c] = NA

LC 1049. Last Stone Weight II

给你一个数组nums，每次你可以选择相邻的两个数对消，只留下它们之差的绝对值放在原地，一直操作到只剩一个数。问这个数最小是多少？

分析：任何一种完整的消除操作方案其实对应着在所有数字前面加+/-符号。
比如[1,4] => -1+4 = 3，[1,2,3] => -1-2+3 = 0

反之虽然并不成立，但对于本题（求剩下的最小的正数）没有影响。（不做深入讨论）

本题可以转化为494. target sum的进阶版：**给你一个数组nums，可以在每个元素前添加正负号。问使得最后整体结果最小的正数是多少？**

LC 1049. Last Stone Weight II

给你一个数组nums，可以在每个元素前添加正负号。问使得最后整体结果最小的正数是多少？

状态定义：dp[i][s] => 考虑仅在前i个元素的子集中添加符号，所得估值恰好是s，是否可行。

状态转移：当前的估值是s，那么前一轮的估值是多少？s-nums[i]或者s+nums[i].

```
for (int i=1; i<=N; i++)
    for (int s=-MAX_SUM; s<=MAX_SUM; s++)
    {
        dp[i][s] = dp[i-1][s-nums[i]] || dp[i-1][s+nums[i]];
    }
```

Ans = the first positive s, s.t. dp[N][s] == true

注意边界条件：
dp[0][0] = true
dp[0][s] = false

LC 474.Ones and Zeroes

有一系列的binary strings。问你最多能挑选几个字符串并且所需要的0和1的总数不超过m和n。

状态定义: dp[i][c1][c2] => 考虑仅在前i种字符串的子集中选择，且所选的0和1数目恰好是(c1,c2)时的最优解。

状态转移: 当前的代价是(c1,c2)，那么前一轮的代价是多少？
(c1,c2) 或者 (c1-ai,c2-bi)

```
for (int i=1; i<=N; i++)
      for (int c1=1; c1<=m; c1++)
          for (int c2=1; c2<=n; c2++)
          {
               dp[i][c1][c2] = max(dp[i-1][c1][c2],
                                   dp[i-1][c1-ai][c2-bi] + 1 );
          }
```

Ans = max {dp[N][c1][c2] } for c1<=m, c2<=n

注意边界条件:
dp[0][0][0] = 0
dp[0][c1][c2] = NA

LC 879. Profitable Schemes

有一系列的任务。每个任务需要消耗人力和产生价值。问你挑选这些任务，有多少种组合方法能够满足总人力小于等于G但总价值大于等于P。

状态定义：dp[i][g][p] => 考虑仅在前i种task的子集中选择，且所选的人力为g、总价值为p时的解（即方案组合数）。

状态转移：当前的代价是(g,p)，那么前一轮的代价是多少？(g,p) 或者 (g-gi,p-pi)

```
for (int i=1; i<=N; i++)
      for (int g=1; g<=MaxG; g++)
          for (int p=1; p<=MaxP; p++)
          {
               dp[i][g][p] = dp[i-1][g][p] + dp[i-1][g-gi][p-pi];
          }
```

Ans = sum {dp[N][g][p] } for g<=G, p>=P

LC 879. Profitable Schemes

有一系列的任务。每个任务需要消耗人力和产生价值。问你挑选这些任务，有多少种组合方法能够满足总人力小于等于G但总价值大于等于P。

状态定义: dp[i][g][p] => 考虑仅在前i种task的子集中选择，且所选的人力为g、总价值为p时的解（即方案组合数）。

状态转移: 前一轮的代价是(g,p)，那么这一轮的代价是多少？(g,p) 或者 (g+gi,p+pi)

```
for (int i=1; i<=N; i++)
      for (int g=1; g<=G; g++)
          for (int p=1; p<=P; p++)
          {
                dp[i][g][p] += dp[i-1][g][p];
                dp[i][min(g+gi,G+1)][min(p+pi,P)] += dp[i-1][g][p];
          }
```

Ans = sum {dp[N][g][P] } for g<=G

LC 956. Tallest Billboard

给你一系列棍子的长度。让你选择部分棍子拼接成两根，要求这两根拼接的棍子高度相等。问最长能拼多少？

状态定义：dp[i][l][r] => 考虑仅在前i种sticks的子集中选择，并且拼出左边长度为l、右边长度为r，是否可行（布尔型）。

状态转移：当前的状态是(l,r)，那么前一轮的代价是多少？(l,r) 或者 (l-hi,r)或者 (l,r-hi)

```
for (int i=1; i<=N; i++)
      for (int l=1; l<=MaxL; l++)
          for (int r=r; r<=MaxR; r++)
          {
               if (dp[i-1][l][r] | dp[i-1][l-hi][r] | dp[i-1][l][r-hi])
                         dp[i][l][r] = 1;
           }
```

Ans = max{l or r} for dp[N][l][r]==1 and l==r

LC 956. Tallest Billboard

给你一系列棍子的长度。让你选择部分棍子拼接成两根，要求这两根拼接的棍子高度相等。问最长能拼多少？

状态定义：dp[i][d] => 考虑仅在前i种sticks的子集中选择，并且拼出左边长度与右边长度之差为d时，对应的左边长度的最大值。

状态转移：当前的状态是d，那么前一轮的状态是多少？d或者d-hi 或者 d+hi

```
for (int i=1; i<=N; i++)
    for (int d=-MaxD; d<=MaxD; d++)
    {
        dp[i][d] = max(dp[i-1][d], dp[i-1][d+hi],
dp[i-1][d-hi]+hi)
    }
```

Ans = dp[N][0]

状态压缩

对于比较复杂的“状态”，DP经常会用到“状态压缩”的技巧。

比如：有些情况下如果想设计“状态”代表一个01向量（不超过32位），我们可以用一个整形的bit位来表示。

[1,0,1,1,0,0,1] => b1011001 => 89

LC 691. Stickers to Spell Word

给你一系列单词words。让你选择部分单词，可以利用里面的字母（无限次），尝试拼出一个新单词target。问最少需要选择多少单词？

状态定义：dp[i][set] => 考虑仅在前i种字符串的子集中选择，能够构成的字母集合是set，对应的最优解（即最少需要选择单词数）。

状态转移：前一轮的字母集合是set，那么这一轮的字母集合是多少？set 与 words[i] 的并集！

```
for (int i=1; i<=N; i++)
    for (int set=0; set<=(1<<26)-1; set++)
    {
        dp[i][set] = min(dp[i][set], dp[i-1][set]);
        int new_set = unionset (set, words[i]);
        dp[i][new_set ] = min(dp[i][new_set], dp[i-1][set]+1);
    }
```

Ans = min{dp[N][set]}
for set = 0,1,...,(1<<26)-1 and set cover target_set

LC 1125. Smallest Sufficient Team

给你N个人，每个人会一些技能（技能编号0到M）。现在有个任务需要的技能集合是target，问最少需要召唤几个人？

状态定义：dp[i][set] => 考虑仅在前i个人的子集中选择，能够构成的技能集合是set，对应的最优解（即最少需要选择多少人）。

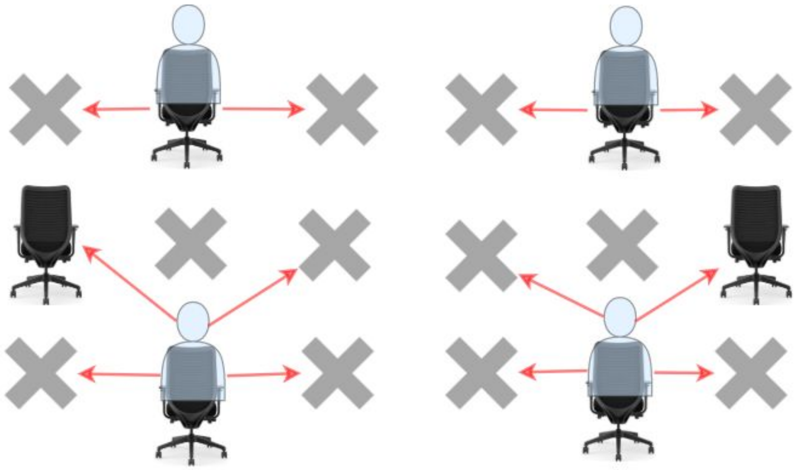
状态转移：前一轮的技能集合是set，那么这一轮的技能集合是多少？set 与 skills[i] 的并集！

```
for (int i=1; i<=N; i++)
    for (int set=0; set<=(1<<m)-1; set++)
    {
        dp[i][set] = min(dp[i][set], dp[i-1][set]);
        int new_set = unionset (set, skills[i]);
        dp[i][new_set ] = min(dp[i][new_set], dp[i-1][set]+1);
    }
```

Ans = min{dp[N][set]}
for set = 0,1,...,(1<<m)-1 and set cover target_set

LC 1349. Maximum Students Taking Exam

给你N*M的矩阵安排考生座位。要求每个人的左前、右前、左、右不能有人。问最多可以安排多少考生。



第i行考生如何安排，仅受第i-1行考生安排的制约，而与更早的状态无关。
所以这是一个“时间序列型”的基本DP。

LC 1349. Maximum Students Taking Exam

给你N*M的矩阵安排考生座位。要求每个人的左前、右前、左、右不能有人。问最多可以安排多少考生。

状态定义: dp[i][pattern] => 仅考虑前i行。当第i行考生的座位安排为pattern时的最优解（即做多可以总共安排多少人）

状态转移: 这一行的座位安排是pattern，那么前一行的座位安排模式可以是什么呢？从0到(1<<M)-1遍历检查一下。

```
for (int i=1; i<=N; i++)
   for (int pattern=0; pattern<=(1<<m)-1; pattern++)
     {
        If (!selfOK(pattern)) continue;
        for (int prev_pat=0; prev_pat<=(1<<m)-1; prev_pat++)
        {
            if (conflict(prev_pat, pattern) continue;
            dp[i][pattern] = max (dp[i][pattern],
                                  dp[i-1][prev_pat] + num(pattern);
        }
     }
```

Ans = max{dp[N][pattern]}
for pattern = 0,1,...,(1<<m)-1 and selfOK(pattern)

Homework

第I类基本型：903.Valid Permutations for DI Sequence

第II类基本型：983.Minimum Cost For Tickets

第II类区间型：546.Remove Boxes

背包型：518.Coin Change 2

状态压缩：943.Find the Shortest Superstring

弃坑型：887.Super Egg Drop, 920.Number of Music Playlists

The End

讲不动了。谢谢观看。

本次讲座的视频录像链接: https://youtu.be/FLbqgyJ-70I