动态规划 B

经典问题

杨乐

blog.csdn.net/yangle61
@897982078
yangle61@163.com

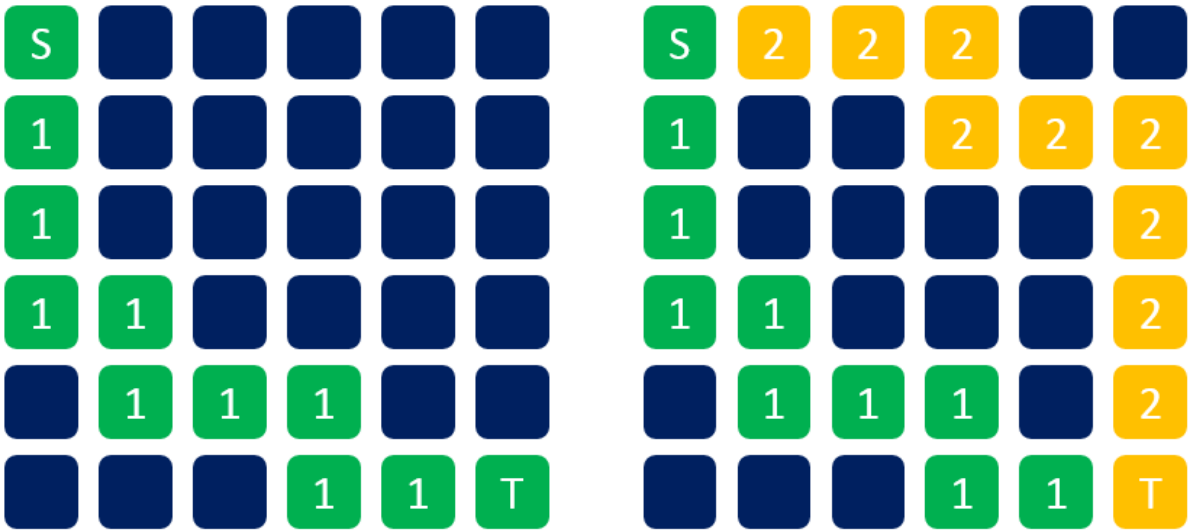
2017 年 7 月 4 日

问题引入 - 路径行走问题

* Luogu 1004 : 方格取数
* 设有 *N*N* 的方格图 *(N<=9)*，我们将其中的某些方格中填入正整数，而其他的方格中则放入数字 *0*。
* 某人从图的左上角的 *A* 点出发，可以向下行走，也可以向右走，直到到达右下角的 *B* 点。在走过的路上，他可以取走方格中的数（取走后的方格中将变为数字 *0*）。
* 此人从 *A* 点到 *B* 点共走两次，试找出 *2* 条这样的路径，使得取得的数之和为最大。

- 与数字金字塔很类似？如果只走一次呢？

问题引入 - 路径行走问题



图：左图：只走一次的情形；右图：两条路径

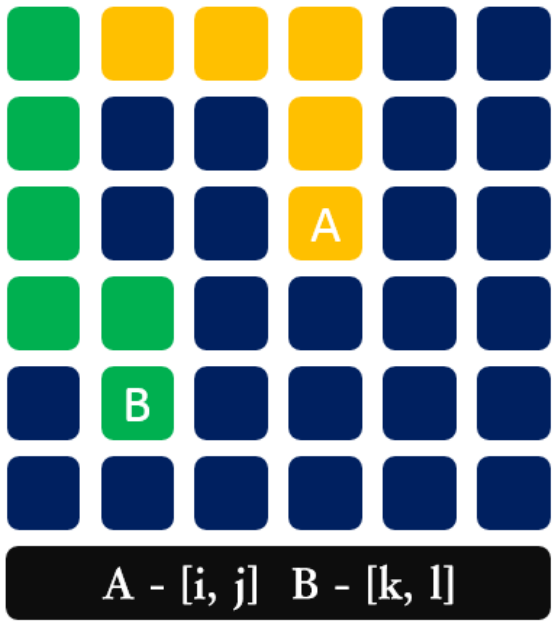
* 只走一次：(仿照数字金字塔)记录 `F[i][j]` 为走到第 `i` 行第 `j` 列的最大值。

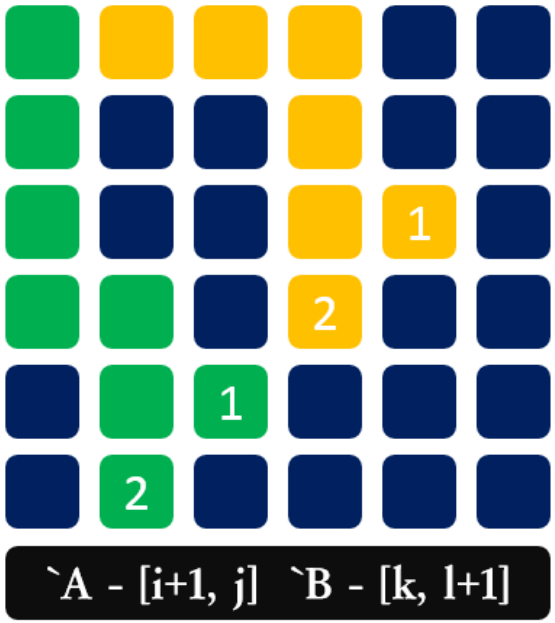
- 思考：转移的顺序？转移的方程？

问题引入 - 路径行走问题 - 状态设计

* 问题：在这道题目当中我们不能直接套用走一次的方法；一个方格只能被取走一次。

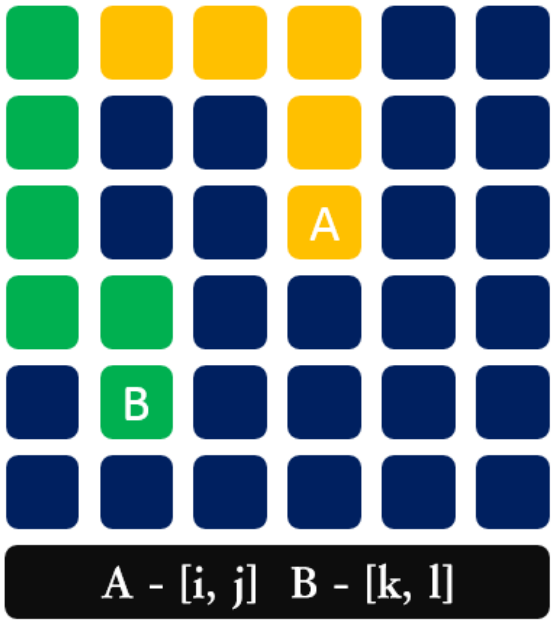
- 考虑两条道路同时进行：状态 `F[i][j][k][l]` 来记录第一条路径走到 `(i,j)`，而第二条路径走到 `(k,l)` 的最大值。

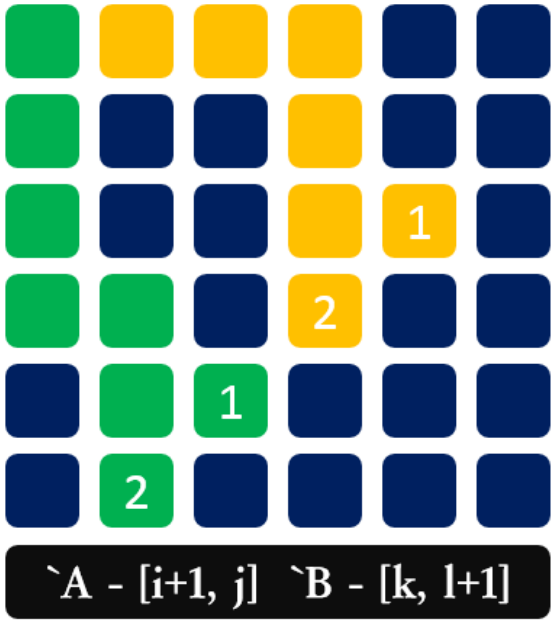




问题引入 - 路径行走问题 - 转移方程

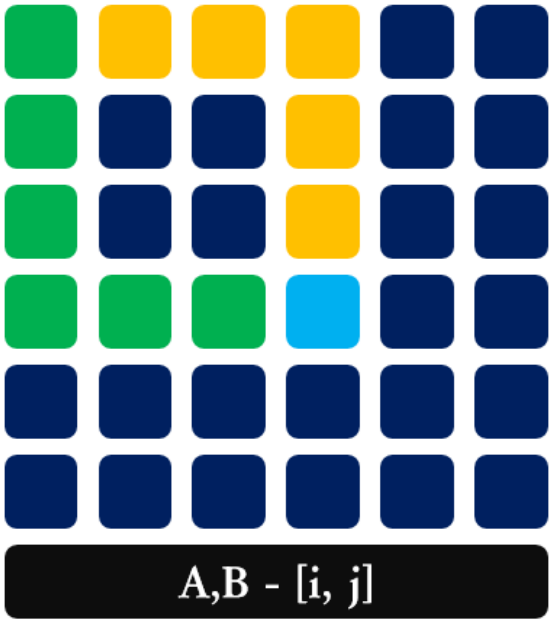
* **转移方程**：考虑逆推(我可能是由哪些状态得到的)。
- 每个点可以往下走或者往右走；一共有 4 种可能性(时刻注意边界情况！)

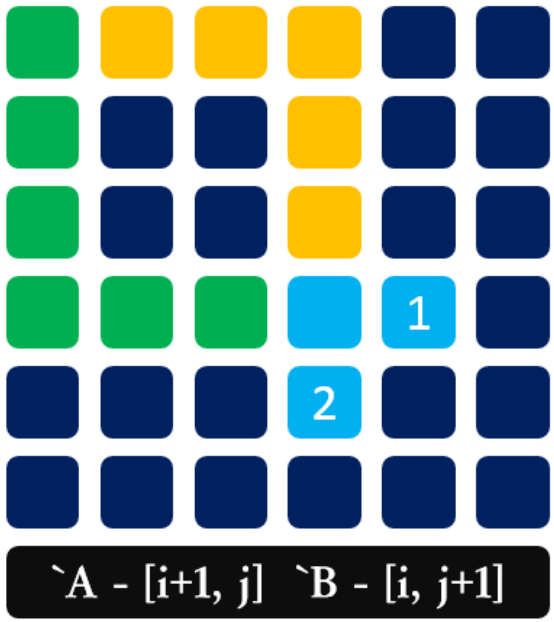




问题引入 - 路径行走问题 - 转移方程

* **转移方程**：考虑逆推(我可能是由哪些状态得到的)。
- **注意！**：如果两个点同时走到一个地方呢？权值只能被加一次。





路径行走问题 - 逆推

```
// T11 : 方格取数（DP / 逆推）

  for(int i = 1; i <= n; ++ i)
     for(int j = 1; j <= n; ++ j)
        for(int k = 1; k <= n; ++ k)
           for(int l = 1; l <= n; ++ l)
           {
              // 注意，如果走到了一起，只加一次
              int cost = a[i][j] + a[k][l] - a[i][j] * (i == k && j == l);

              // 四种可能性；考虑：为什么不加边界情况的判断？
              f[i][j][k][l] = max
              (
                 max(f[i-1][j][k-1][l], f[i-1][j][k][l-1]),
                 max(f[i][j-1][k-1][l], f[i][j-1][k][l-1])
              ) + cost;
           }
```

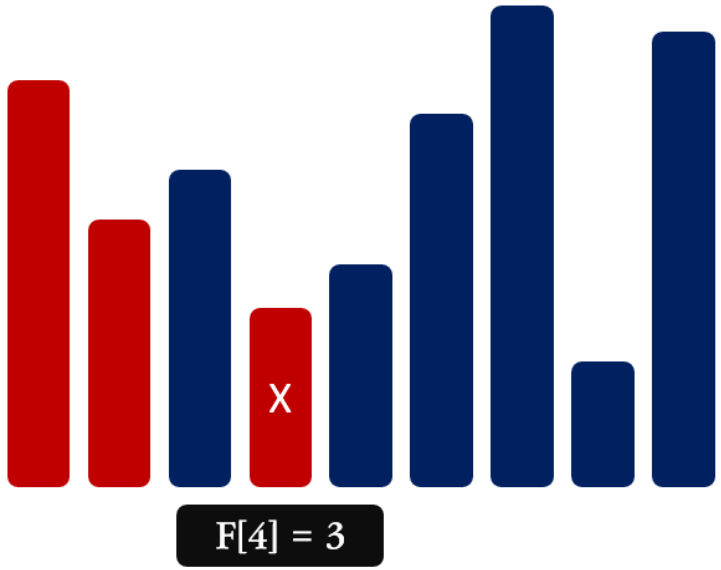
问题引入 - 最长不下降子序列问题

* Luogu 1020 : 导弹拦截
* 某国为了防御敌国的导弹袭击，发展出一种导弹拦截系统。但是这种导弹拦截系统有一个缺陷：虽然它的第一发炮弹能够到达任意的高度，但是以后每一发炮弹都不能高于前一发的高度。某天，雷达捕捉到敌国的导弹来袭。由于该系统还在试用阶段，所以只有一套系统，因此有可能不能拦截所有的导弹。
* 输入导弹依次飞来的高度（雷达给出的高度数据是不大于 *30000* 的正整数），计算这套系统最多能拦截多少导弹，如果要拦截所有导弹最少要配备多少套这种导弹拦截系统。

- *389 207 155 300 299 170 158 65*
- *最多能够拦截：6；最少要配备：2*

问题引入 - LIS - 状态设计

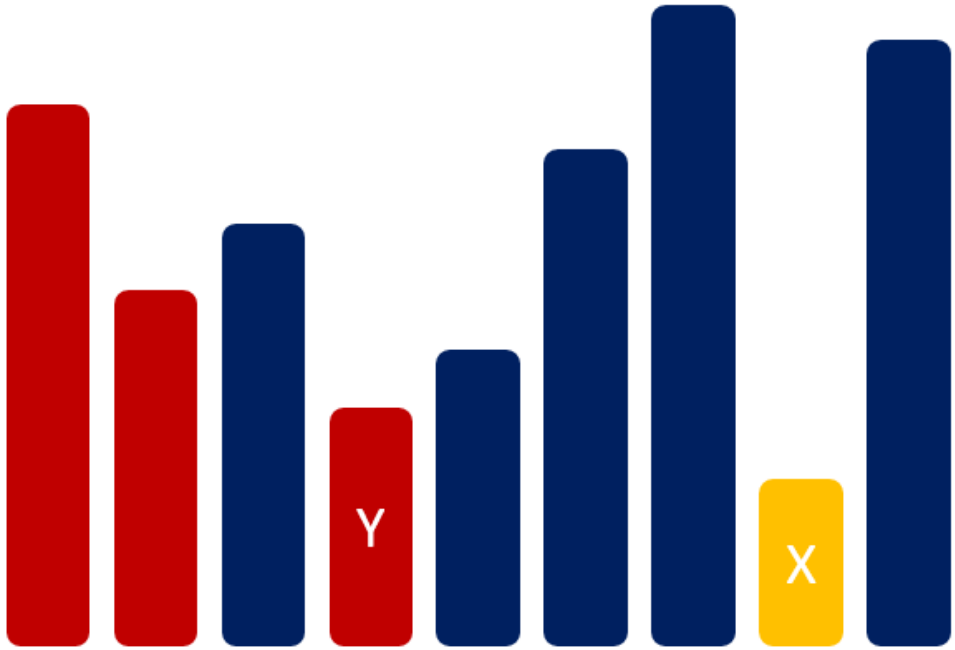
* **LIS**：一个序列当中一段不下降的子序列。
* 这道题目中**第一问**要求我们找到一段最长的**单调下降的子序列**。(无论是上升还是下降，可以使用类似的算法解决)
* **状态**：我们用 `F[i]` 代表，以**i 位置为结尾**的一段，**最长的下降子序列**。
! **最优性**：如果某段 $[q_1 q_2 q_3 \dots q_n]$ 是以 q_n 结尾的**最长下降子序列**；那么去掉最后一个的序列 $[q_1 q_2 q_3 \dots q_{n-1}]$，依然是以 q_{n-1} 结尾的**最长下降子序列**。



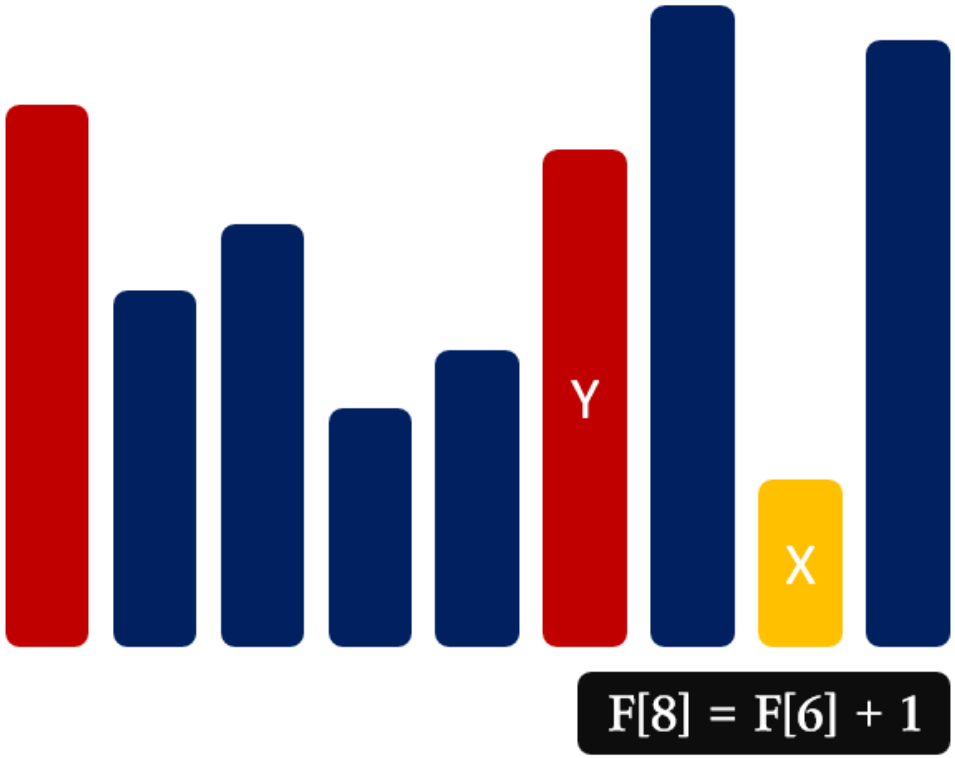
问题引入 - LIS - 转移方程

* **逆推**：假设我们需要求以 X 结尾的**最长下降子序列 F[X]**；
* 由**最优性**可得，我们除去最后一个位置*(也就是 X)*，还是一段**最长下降子序列**。
* 那我们可以枚举这个子序列的结尾 Y，最优值就是 F[Y]。
! 但需要注意的是，必须保证**A[X] < A[Y]**，X 比 Y 要低，才满足下降的要求。
* 我们从所有枚举的结果中找到一个**最大**的即可。

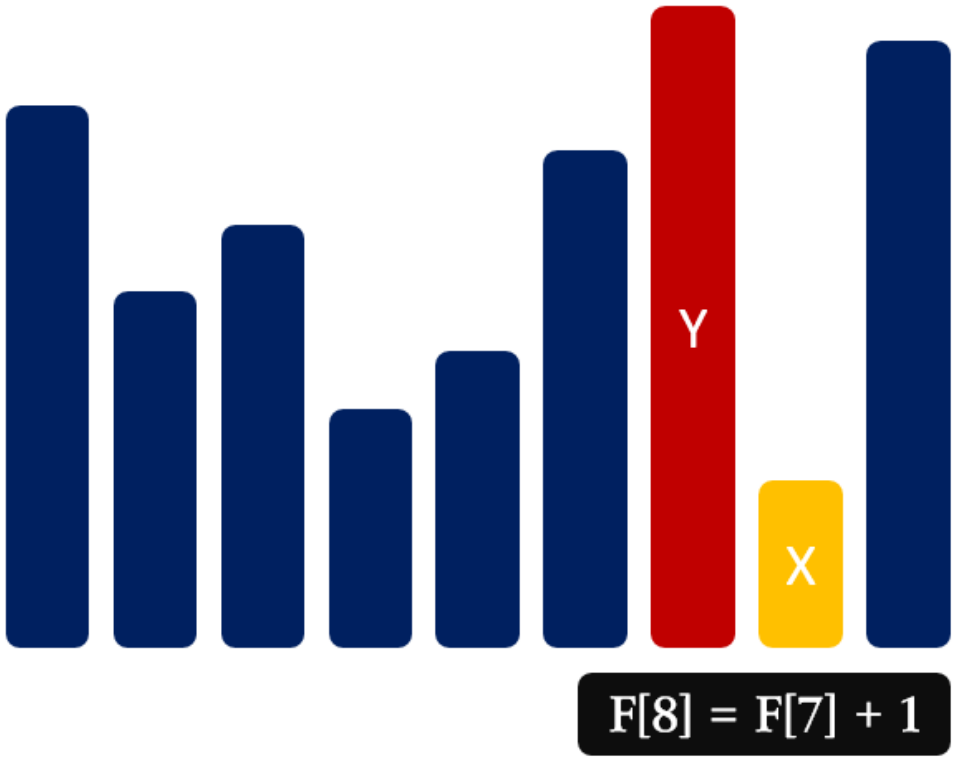
问题引入 - LIS - 转移方程



问题引入 - LIS - 转移方程

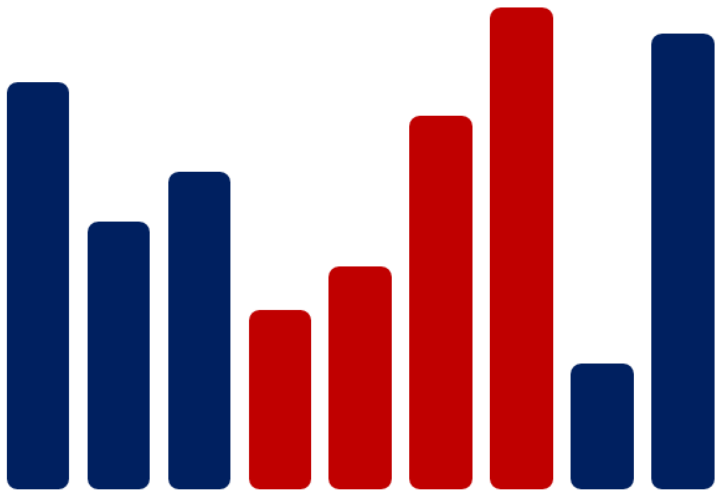


问题引入 - LIS - 转移方程



问题引入 - LIS - 状态设计

* 注意到题目还需要计算“如果要拦截所有导弹最少要配备多少套这种导弹拦截系统。”
* 可以直接观察得到，所求的答案至少为原题的最长不下降子序列。
  - 因为它们当中，任意两个都不可能被同一个导弹打中。
  - 事实可以证明，这就是答案。



图：图中的 4 个导弹都单独需要一个系统。

LIS - 逆推

```
// T12 : 导弹拦截（DP / LIS / 逆推）

  int ansf = 0, ansg = 0; // 记录所有的 f(g) 中的最优值

  // f 计算下降子序列，g 计算不下降子序列
  for(int i = 1; i <= n; ++ i)
  {
     for(int j = 1; j < i; ++ j)
        if(a[j] > a[i]) f[i] = max(f[i], f[j]);
        else g[i] = max(g[i], g[j]);

     ++ f[i], ++ g[i]; // 加上自己的一个

     ansf = max(ansf, f[i]);
     ansg = max(ansg, g[i]);
  }

  cout << ansf << endl << ansg << endl; // 输出答案
```

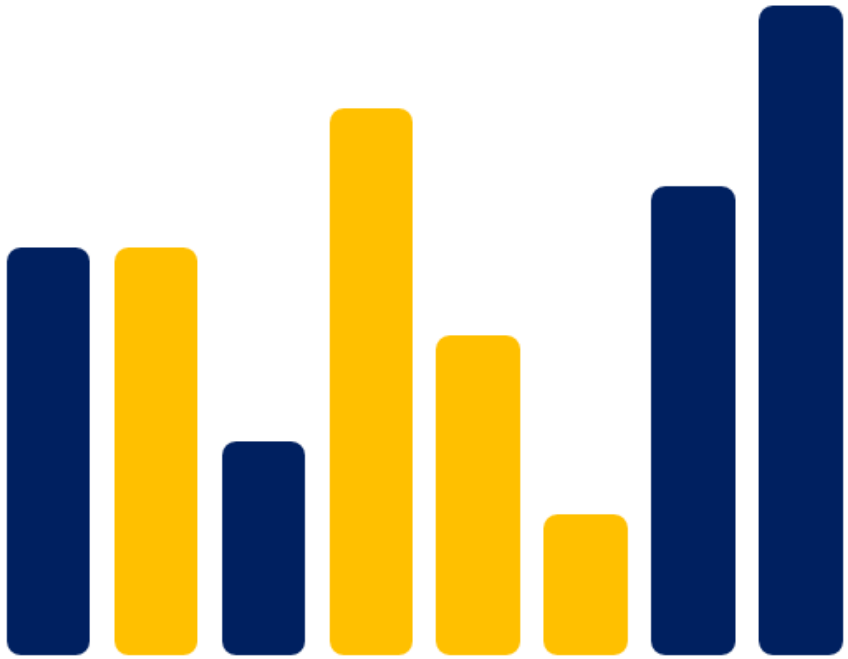
问题变形 - 合唱队形

* Luogu 1091 : 合唱队形
* N 位同学站成一排，音乐老师要请其中的 $(N-K)$ 位同学出列，使得剩下的 K 位同学排成合唱队形。
* 合唱队形是指这样的一种队形：设 K 位同学从左到右依次编号为 $1, 2 \cdots, K$，他们的身高分别为 $T_1, T_2, \ldots, T_K$，则他们的身高满足 $T_1 < \ldots < T_i > T_{i+1} > \ldots > T_K (1 \leq i \leq K)$。
* 你的任务是，已知所有 N 位同学的身高，计算最少需要几位同学出列，可以使得剩下的同学排成合唱队形。

- *186 186 150 200 160 130 197 220*
- 最少需要 *4* 位同学出列

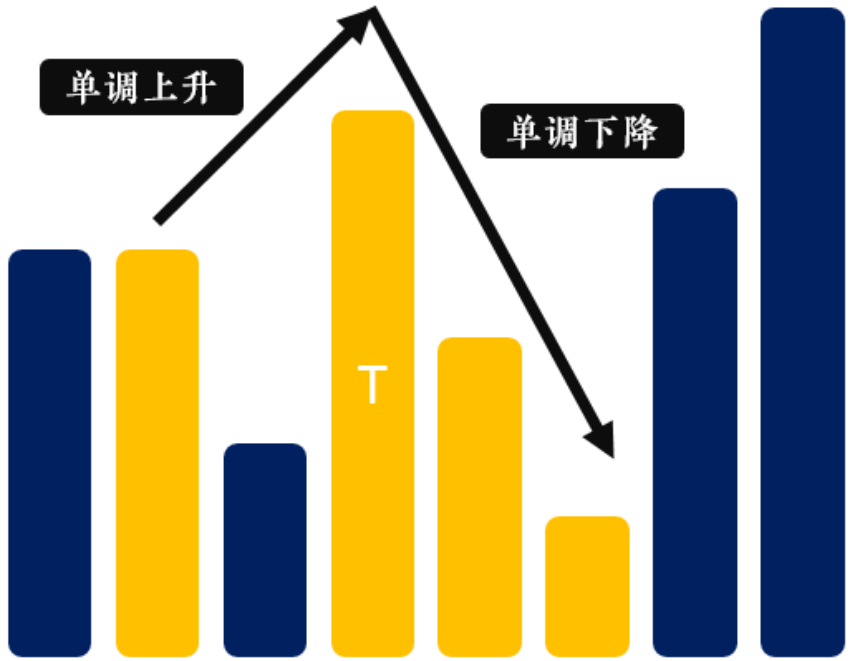
问题变形 - 合唱队形

* **合唱队形**：类似于山峰的形状，有一个顶尖。



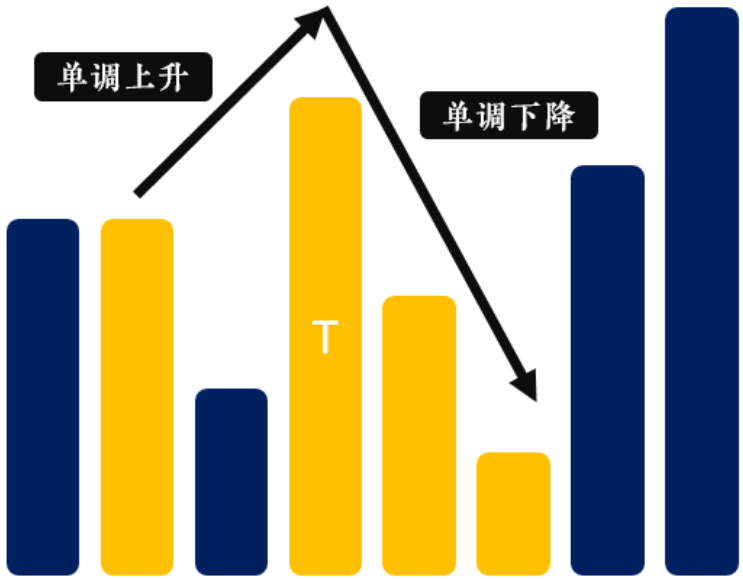
问题变形 - 合唱队形

* 问题转化：最少的同学出列 -> 尽量多的同学留在队列
* 与 LIS 的联系：如果确定了中间的“顶尖”，两侧就是“单调上升”和“单调下降”的。



问题变形 - 合唱队形

* 状态设计：F[i] 与 G[i]
  - F[i]：以 i 为端点，左侧的最长的上升子序列长度。
  - G[i]：以 i 为端点，右侧的最长的下降子序列长度。
* 则计算出 F[i] 与 G[i] 后，以 i 为“顶尖”的最长“合唱队形”为是多少？



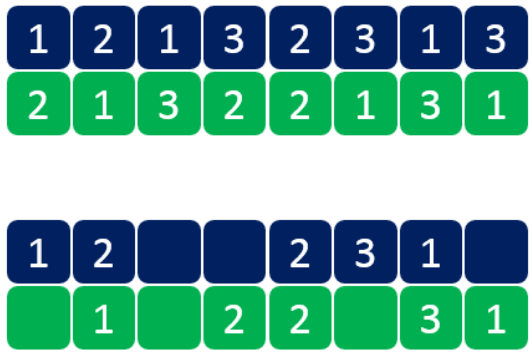
合唱队形 - 逆推

```
// T13 : 合唱队形（DP / LIS / 逆推）

  for(int i = 1; i <= n; ++ i) cin >> a[i], f[i] = g[i] = 1;

  for(int i = 1; i <= n; ++ i)
     for(int j = 1; j < i; ++ j)
        if(a[i] > a[j]) f[i] = max(f[i], f[j] + 1);

  for(int i = n; i; -- i) // g 的计算从反方向进行枚举
     for(int j = n; j > i; -- j)
        if(a[i] > a[j]) g[i] = max(g[i], g[j] + 1);

  int ans = 0;
  for(int i = 1; i <= n; ++ i) // 枚举每一个顶点为顶尖
     ans = max(ans, f[i] + g[i] - 1);
```

问题引入 - 最长公共子序列

* Luogu 1439 : 排列 LCS 问题
* 给出 *1-n* 的两个排列 *P1* 和 *P2*，求它们的最长公共子序列。
* 公共子序列：既是 P1 的子序列，也是 P2 的子序列。

- *3 2 1 4 5*
- *1 2 3 4 5*
- 最长公共子序列 *(LCS)*: *3([1 4 5])*



图：例子中，*[1 2 2 3 1]* 即为序列的公共子序列

问题引入 - LCS - 状态设计

* **LCS**：两个序列的**最长公共子序列**。
* **状态**：我们用 `F[i][j]` 代表，前一个序列以 `i` 位置为结尾，后一个序列以 `j` 位置为结尾，它们的**最长公共子序列**。
! **最优性**：如果某段 $[q_1 q_2 q_3 \dots q_n]$ 是分别以 i, j 结尾的**最长公共子序列**；那么去掉最后一个的序列 $[q_1 q_2 q_3 \dots q_{n-1}]$，依然是以 $i-1, j-1$ 结尾的**最长公共子序列**。

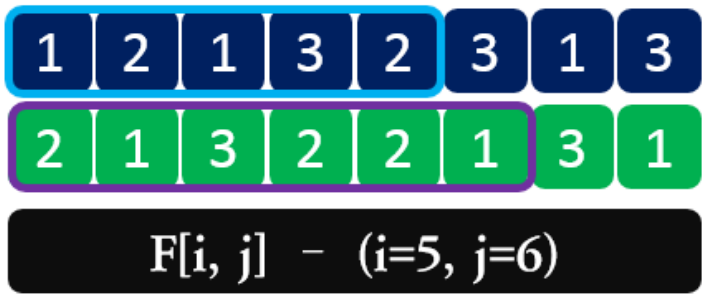
图：例子中，去掉结尾的 5 后，[1 4] 仍是最长公共子序列

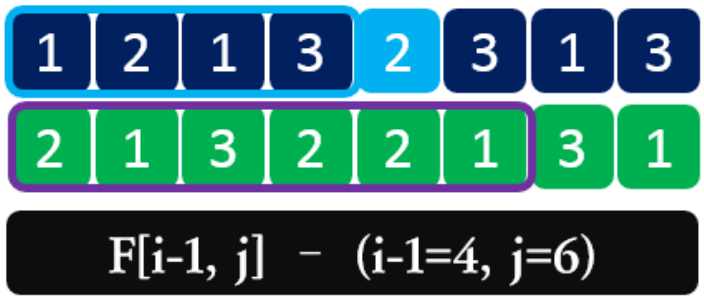
问题引入 - LCS - 转移方程

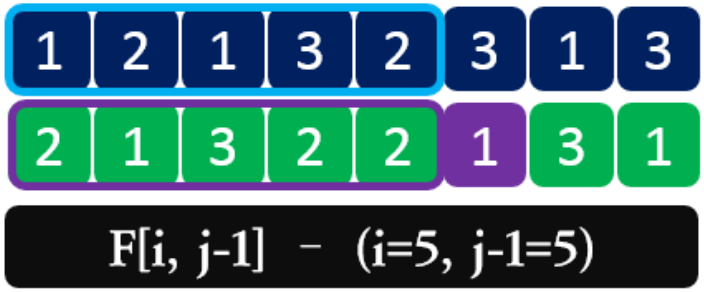
* **逆推**：假设我们需要求两个序列分别以 `i,j` 结尾的**最长公共子序列** `F[i][j]`，接下来我们可以分几种情况讨论：
  - `A[i]` 不在**公共子序列**中，那么长度则等于 `F[i-1][j]`；
  - `B[j]` 不在**公共子序列**中，那么长度则等于 `F[i][j-1]`；
  - `A[i]` 与 `B[j]` 都在**子序列**中，并且两者匹配，那么长度等于 `F[i-1][j-1]+1`；
* 我们从所有枚举的结果中找到一个**最大**的即可。

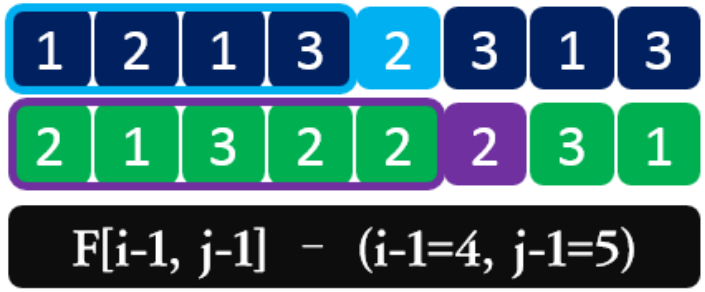
问题引入 - LCS - 转移方程

* 逆推：假设我们需要求两个序列分别以 `i,j` 结尾的最长公共子序列 `F[i][j]`，可能的三种情况：









LCS - 逆推

```
// T16 : 排列 LCS 问题（DP / LCS / 50 分数据规模限制）

  for(int i = 1; i <= n; ++ i)
     for(int j = 1; j <= n; ++ j)
     {
        // 分三种情况进行讨论
        f[i][j] = max(f[i-1][j], f[i][j-1]);
        if(p[i] == q[j]) f[i][j] = max(f[i][j], f[i-1][j-1] + 1);
     }

  int ans = f[n][n];
```

问题引入 - LCS 到 LIS

* 在这道题目里，**LCS**问题是可以转化为**LIS**的：
* 假定某一个序列为 [1 2 3 ... N]，那么答案则是另一个序列的 LIS；

- *3 2 1 4 5*
- *1 2 3 4 5*

* 但如果两个序列都不是 [1 2 3 ... N] 呢？通过转化使一个序列变成它，而答案不变。

- *5 3 4 1 2* -> *1 2 3 4 5*
- *3 5 1 2 4* -> *2 1 4 5 3*

问题变形 - 字串距离

* Luogu 1279 ： 字串距离
* 设有字符串 X，我们称在 X 的头尾及中间插入任意多个空格后构成的新字符串为 X 的扩展串，如字符串 X 为”$abcbcd$”，则字符串“$abcb\square cd$”，“$\square a\square bcbcd\square$”和“$abcb\square cd\square$”都是 X 的扩展串，这里“$\square$”代表空格字符。
* 如果 $A1$ 是字符串 A 的扩展串，$B1$ 是字符串 B 的扩展串，$A1$ 与 $B1$ 具有相同的长度，那么我们定义字符串 $A1$ 与 $B1$ 的距离为相应位置上的字符的距离总和，而两个非空格字符的距离定义为它们的 $ASCII$ 码的差的绝对值，而空格字符与其他任意字符之间的距离为已知的定值 K，空格字符与空格字符的距离为 0。在字符串 A、B 的所有扩展串中，必定存在两个等长的扩展串 $A1$、$B1$，使得 $A1$ 与 $B1$ 之间的距离达到最小，我们将这一距离定义为字符串 A、B 的距离。
* 请你写一个程序，求出字符串 A、B 的距离。

问题变形 - 字串距离

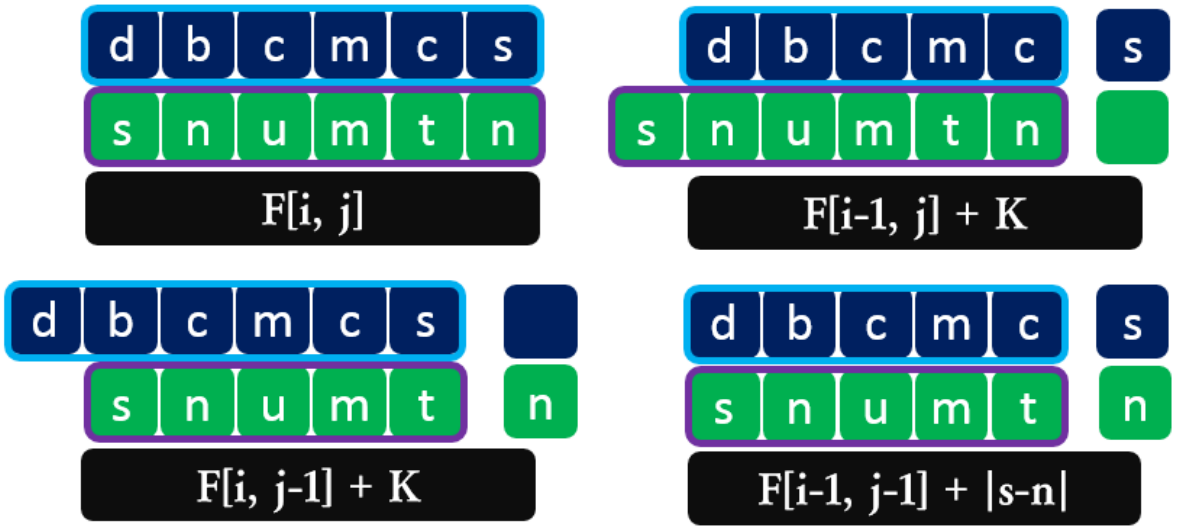
- *cmc*
- *snmn*
- *2*
- 距离为：*10*

图：样例解释，空白位置即为空格

问题变形 - 字串距离

* 状态设计：仿照最长公共子序列，我们设计状态 `F[i][j]` 为前一个序列以 `i` 结尾，后一个序列以 `j` 结尾的最小距离；同样也有以下三种情况

- `A[i]` 与空格匹配：距离为 `F[i-1][j] + K`；*(K 为到空格的距离)*
- `B[j]` 与空格匹配：距离为 `F[i][j-1] + K`；
- `A[i]` 与 `B[j]` 匹配：距离为 `F[i-1][j-1] + |A[i] - B[j]|`；*(ASCII 码的差的绝对值)*



字串距离 - 逆推

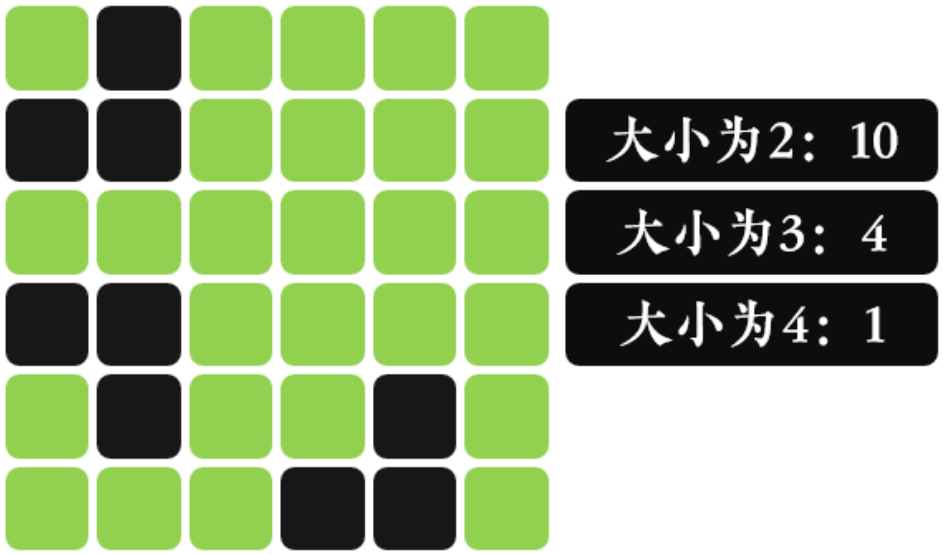
```
// T17 : 字串距离（DP / LCS）

  // 初值：在遇到最小值的问题，一定要小心初值的处理
  f[0][0] = 0;
  for(int i = 1; i <= n; ++ i) f[i][0] = i * k;
  for(int j = 1; j <= m; ++ j) f[0][j] = j * k;

  for(int i = 1; i <= n; ++ i) // n 为第一个串的长度
     for(int j = 1; j <= m; ++ j) // m 为第二个串的长度
     {
        // 三种情况
        f[i][j] = min(f[i-1][j], f[i][j-1]) + k;
        f[i][j] = min(f[i][j], f[i-1][j-1] + abs(p[i-1] - q[j-1]));
     }

  int ans = f[n][m];
```

问题引入 - 二维平面问题

* Luogu 2733 ： 家的范围
* 农民约翰在一片边长是 *N (2 <= N <= 250)* 英里的正方形牧场上放牧他的奶牛。(因为一些原因，他的奶牛只在正方形的牧场上吃草。) 遗憾的是，他的奶牛已经毁坏一些土地。(一些 *1* 平方英里的正方形)
* 农民约翰需要统计那些可以放牧奶牛的正方形牧场 (至少是 *2x2* 的，在这些较大的正方形中没有一个点是被破坏的，也就是说，所有的点都是“*1*”)。
* 你的工作要在被供应的数据组里面统计所有不同的正方形放牧区域 *(>=2x2)* 的个数。当然，放牧区域可能是重叠。

问题引入 - 二维平面问题

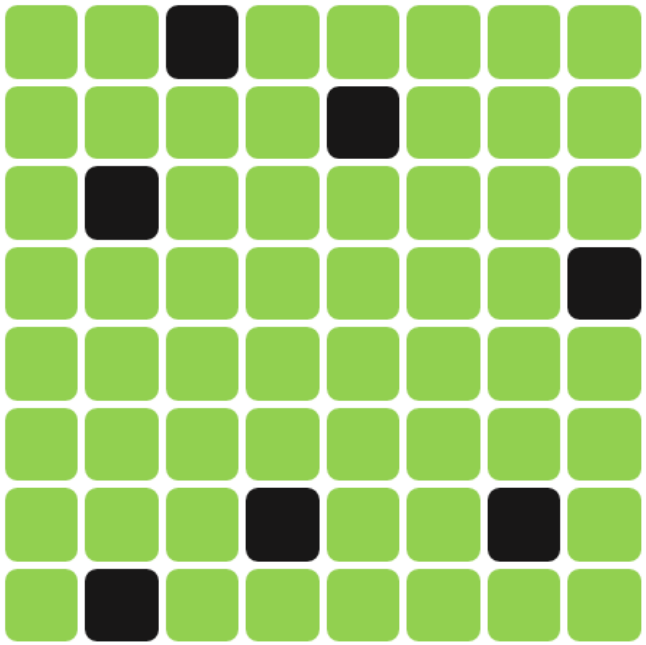
* Luogu 2733 ： 家的范围
* 你的工作要在被供应的数据组里面统计所有不同的正方形放牧区域 (>=2x2) 的个数。当然，放牧区域可能是重叠。

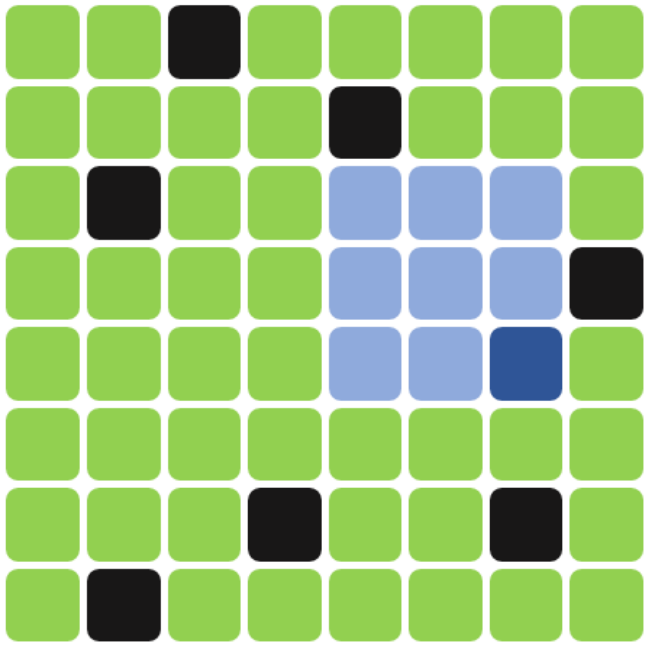


问题引入 - 二维平面问题 - 状态设计

* 考虑一个简单的问题：最大的一个正方形区域是多大？
* **状态设计**：设状态 `F[i][j]` 为以 `(i,j)` 位置为右下角，最大的一个正方形区域。

! 考虑如何进行**状态转移**？





问题引入 - 二维平面问题 - 状态转移

* 考虑位置 F[i-1][j] 与 F[i][j-1];
- 则有 F[i][j] <= F[i-1][j] + 1; F[i][j] <= F[i][j-1] + 1;

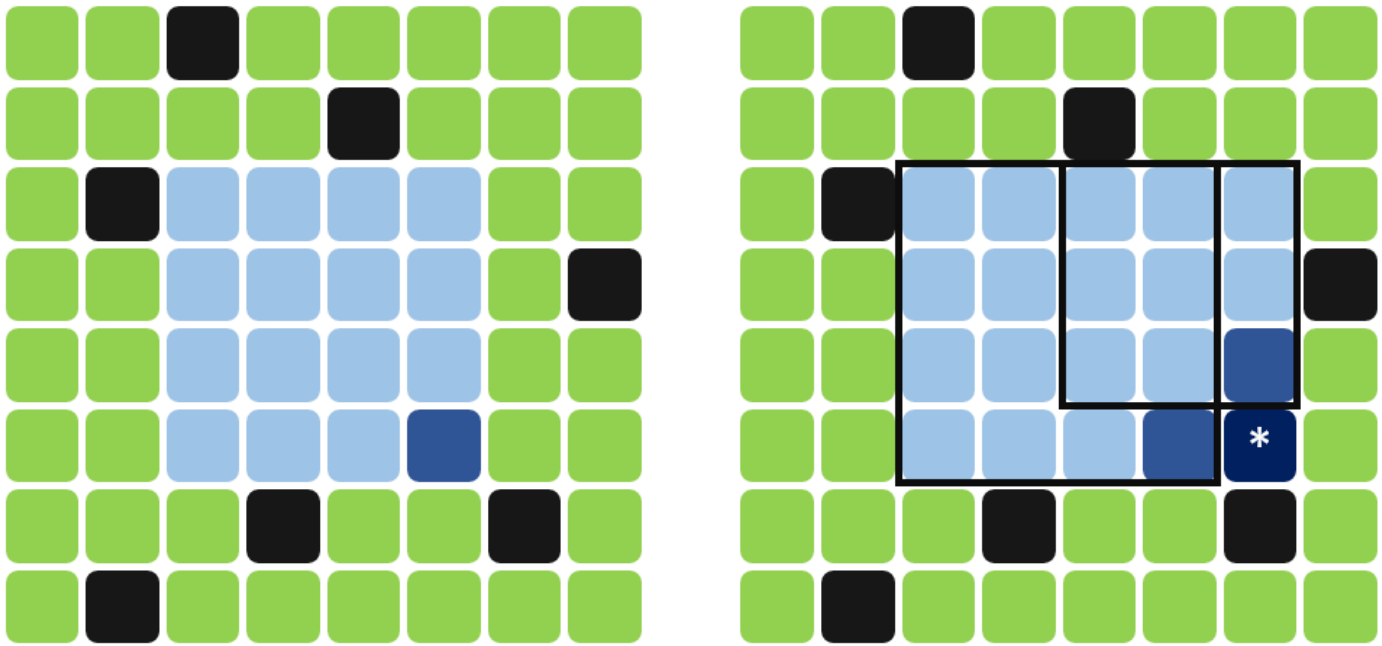
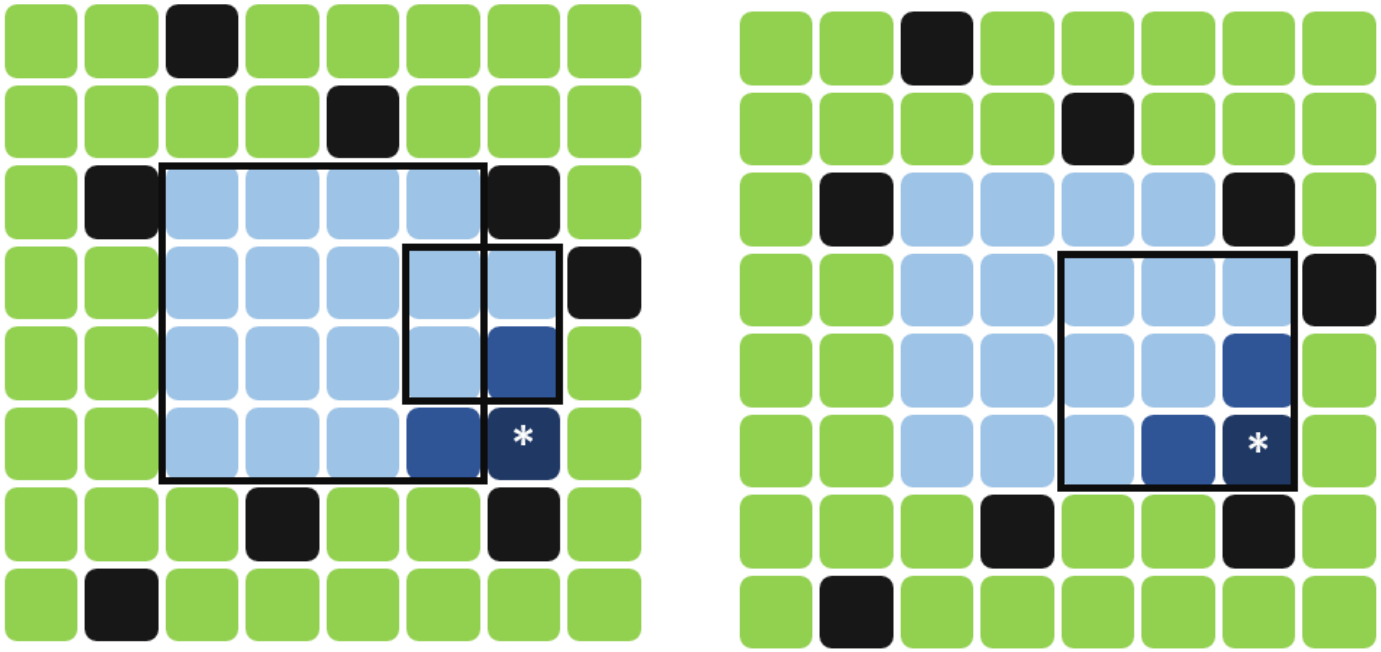


图: 考虑 * 号位置的最大区域，不可能超过 4

问题引入 - 二维平面问题 - 状态转移

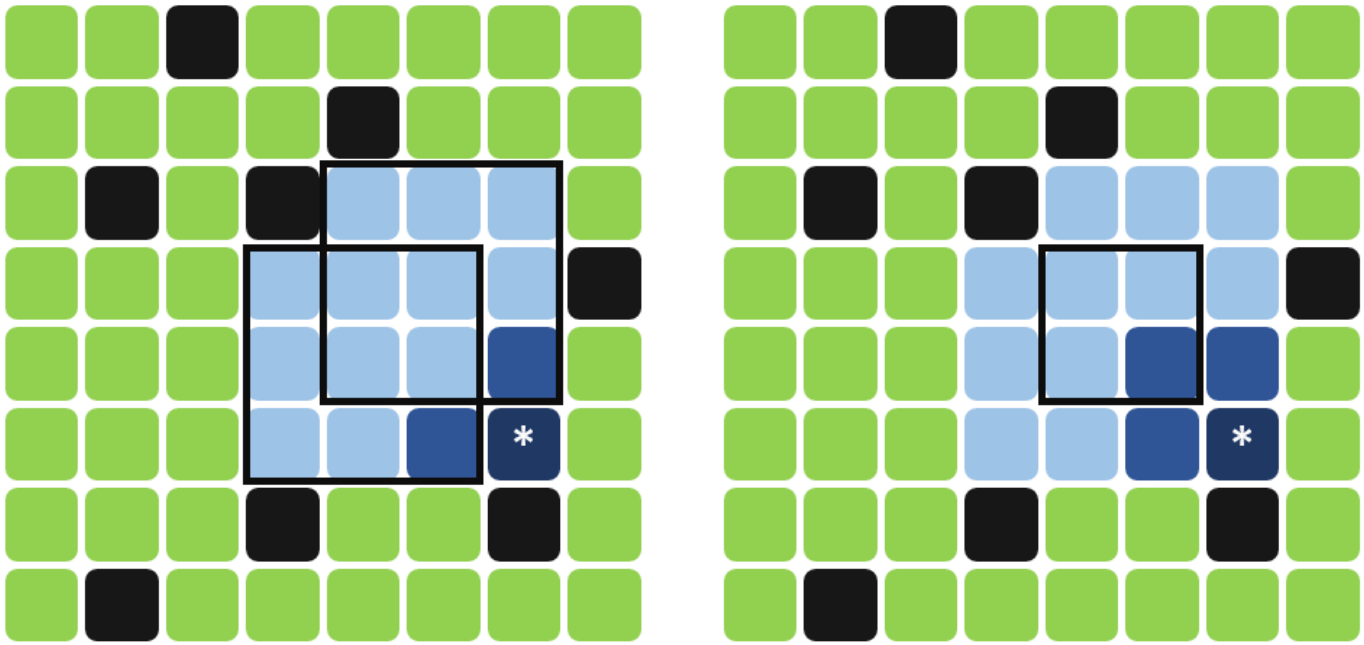
* 很明显，`F[i][j-1]` 与 `F[i-1][j]` 中较小值限制了最大区域的边长。



图：考虑 * 号位置的最大区域，不可能超过 3

问题引入 - 二维平面问题 - 状态转移

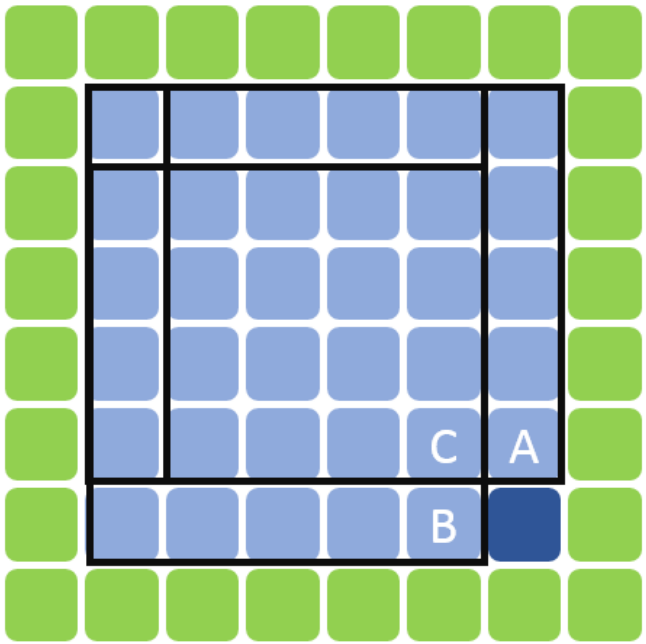
* 但在某些情况下，仅考虑 `F[i][j-1]` 与 `F[i-1][j]` 显然不全面。
* 还需要考虑 `F[i-1][j-1]`。



图：在两个区域中，左上边界的还没有考虑入内

问题引入 - 二维平面问题 - 状态转移

* 所以我们可以得到最终的表达式：`F[i][j] = MIN (F[i-1][j], F[i][j-1], F[i-1][j-1]) + 1` *(当 (i,j) 不为障碍时)*



图：一个正方形的大小需要考虑 *A,B,C* 三个位置

路径行走问题

```
// T18 : 家的范围（DP）

  // 边界初值
  for(int i = 0; i < n; ++ i) f[i][0] = (a[i][0] == '1');
  for(int j = 0; j < n; ++ j) f[0][j] = (a[0][j] == '1');

  for(int i = 1; i < n; ++ i)
     for(int j = 1; j < n; ++ j) if(a[i][j] == '1') // 如果是非障碍
     { // 计算
        f[i][j] = min( min(f[i-1][j], f[i][j-1]), f[i-1][j-1] ) + 1;
        t[ f[i][j] ] ++;
     }

  for(int i = n; i; i --) // 统计所有方形的数目
     t[i-1] += t[i];

  for(int i = 2; i <= n; ++ i) if(t[i]) // 输出结果
     cout << i << "␣" << t[i] << endl;
```

问题引入 - 区间动态规划

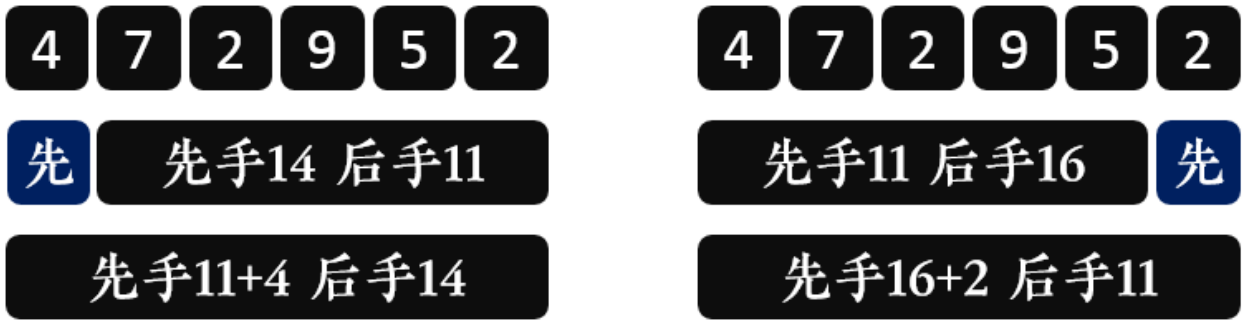
* Luogu 2734 : 游戏
* 有如下一个双人游戏:*N(2 <= N <= 100)* 个正整数的序列放在一个游戏平台上，游戏由玩家 *1* 开始，两人轮流从序列的任意一端取一个数，取数后该数字被去掉并累加到本玩家的得分中，当数取尽时，游戏结束。以最终得分多者为胜。
* 编一个执行最优策略的程序，最优策略就是使玩家在与最好的对手对弈时，能得到的在当前情况下最大的可能的总分的策略。你的程序要始终为第二位玩家执行最优策略。

图: 如图所示即为最优答案及其策略

状态设计 - 区间动态规划

* 状态设计方法为：F[i][j] 表示假若只用第 i 个数与第 j 个数之间的数进行游戏，先手能获得的最高的分为多少。
* 我们可以根据先手取左边还是取右边来决定哪种情况得分高。

! 当先手取完一件后，先后手发生交换，而问题是类似的。



- 考虑：如何决定状态转移顺序？先算哪些，后算哪些？
- 如何书写状态转移方程？

区间动态规划

```
// T19 : 游戏
  cin >> n;
  for(int i = 1; i <= n; ++ i)
     cin >> a[i],
     s[i] = s[i-1] + a[i]; // s 代表前缀和

  for(int i = 1; i <= n; ++ i) f[i][i] = a[i]; // 初值
  for(int k = 2; k <= n; ++ k) // 按照序列的长度进行枚举
     for(int i = 1, j; i + k - 1 <= n; ++ i)
     {
        j = i + k - 1;
        f[i][j] = max ( // 两者取较大值
           s[j] - s[i] - f[i+1][j] + a[i],
           s[j-1] - s[i-1] - f[i][j-1] + a[j]
        );
     }
  // 输出答案
  cout << f[1][n] << "␣" << s[n] - f[1][n] << endl;
```

问题变形 - 区间动态规划

* Luogu 1005 ： 矩阵取数游戏
* 帅帅经常跟同学玩一个矩阵取数游戏：对于一个给定的 $n*m$ 的矩阵，矩阵中的每个元素 Aij 均为非负整数。游戏规则如下：

1. 每次取数时须从每行各取走一个元素，共 n 个。m 次后取完矩阵所有元素；
2. 每次取走的各个元素只能是该元素所在行的行首或行尾；
3. 每次取数都有一个得分值，为每行取数的得分之和，每行取数的得分 = 被取走的元素值 $*2^i$，其中 i 表示第 i 次取数（从 1 开始编号）；
4. 游戏结束总得分为 m 次取数得分之和。

* 帅帅想请你帮忙写一个程序，对于任意矩阵，可以求出取数后的最大得分。

问题变形 - 区间动态规划

* **分析**：

1. 只有一行的话怎么办？如果有很多行怎么办？
2. 一行的情况时：如何设计状态，如何转移？
3. 题目还需要注意什么细节？

问题引入 - 区间动态规划

* Luogu 1040 : 加分二叉树
* 设一个 n 个节点的二叉树 $tree$ 的中序遍历为（$1,2,3,\cdots,n$），其中数字 $1,2,3,\cdots,n$ 为节点编号。每个节点都有一个分数（均为正整数），记第 i 个节点的分数为 di，$tree$ 及它的每个子树都有一个加分，任一棵子树 $subtree$（也包含 $tree$ 本身）的加分计算方法如下：

- $subtree$ 的左子树的加分 $\times$ $subtree$ 的右子树的加分 $+$ $subtree$ 的根的分数。

* 若某个子树为空，规定其加分为 1，叶子的加分就是叶节点本身的分数。不考虑它的空子树。
* 试求一棵符合中序遍历为（$1,2,3,\cdots,n$）且加分最高的二叉树 $tree$。要求输出；

1 $tree$ 的最高加分

2 $tree$ 的前序遍历

问题引入 - 区间动态规划

* Luogu 1040 : 加分二叉树
- *subtree* 的左子树的加分 × *subtree* 的右子树的加分+ *subtree* 的根的分数。
- *5 7 1 2 10*
- 答案 *1*: *145*
- 答案 *2*: *3 1 2 4 5*

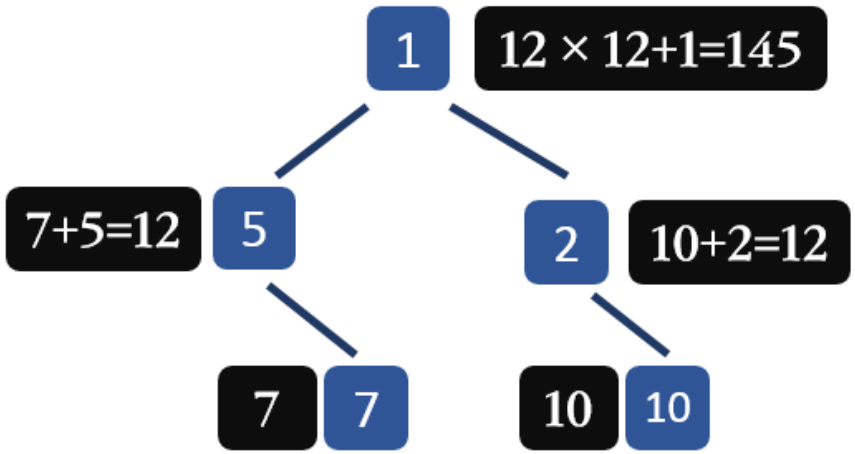
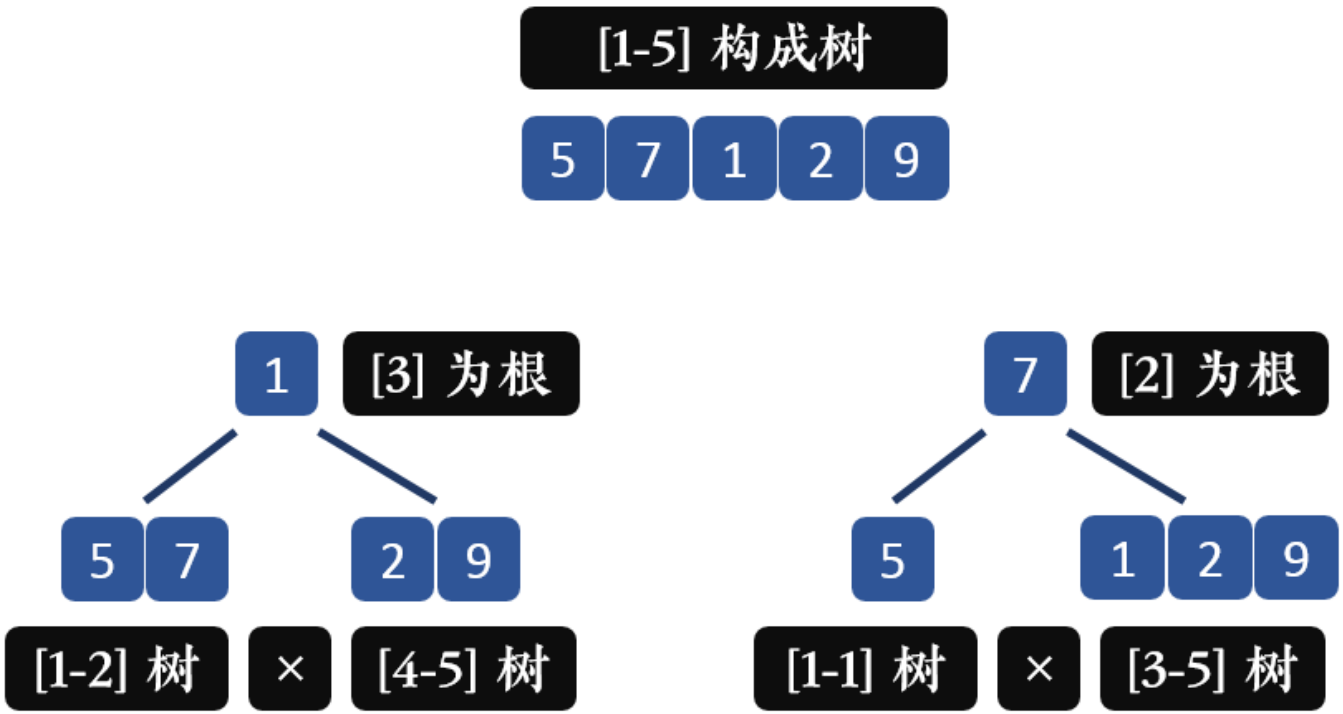


图: 图示即为最优解

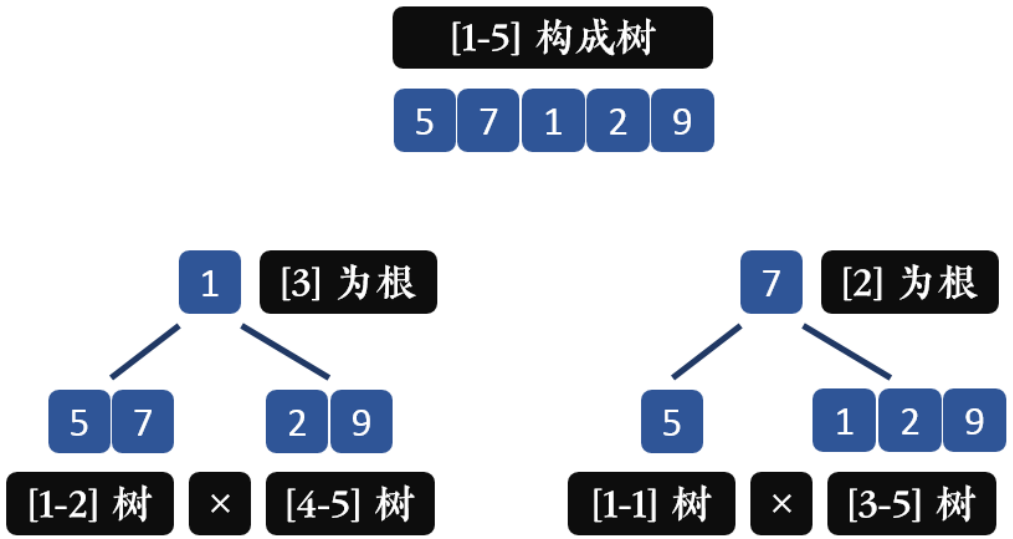
状态设计 - 区间动态规划

* 设 F[i][j] 为只用第 i 个数到第 j 个数构成的加分树的最大权值。
- 如何进行状态转移？



状态设计 - 区间动态规划

* 设 F[i][j] 为只用第 i 个数到第 j 个数构成的加分树的最大权值。
* 枚举根节点，这样就化成了左子树和右子树的问题，求最优解即可。
* F[i][j] = MAX (F[i][k-1] * F[k+1][j] + A[k])



区间动态规划

```
// T25 : 加分二叉树
  for(int i = 1; i <= n; ++ i) f[i][i] = a[i];
  for(int i = 0; i <= n; ++ i) f[i+1][i] = 1;

  for(int k = 1; k < n; ++ k)
     for(int i = 1; i + k <= n; ++ i)
     {
        int j = i + k;
        for(int l = i; l <= j; ++ l)
           f[i][j] = max(f[i][j], f[i][l-1] * f[l+1][j] + a[l]);
     }

  int ans = f[1][n];
```

状态设计 - 区间动态规划

* **问题：**如何求出树的前序遍历*(树的形态)*?
* 另外记录一个数组 `G[i][j]`，代表 `F[i][j]` 取最大值的时候，根节点是什么。
* 这样就可以通过递归来求出树的前序遍历。

区间动态规划

```
// T25 : 加分二叉树
  for(int i = 1; i <= n; ++ i) f[i][i] = a[i], g[i][i] = i; // 边界值
  for(int i = 0; i <= n; ++ i) f[i+1][i] = 1;

  for(int k = 1; k < n; ++ k)
     for(int i = 1; i + k <= n; ++ i)
     {
        int j = i + k;
        for(int l = i; l <= j; ++ l)
        {
           LL t = f[i][l-1] * f[l+1][j] + a[l];
           if(t > f[i][j]) f[i][j] = t, g[i][j] = l; // 记录最优的根
        }
     }
```

区间动态规划

```
  // T25 : 加分二叉树

  // 递归输出 x 到 y 这个树的前缀遍历
void dfs(int x, int y)
{
    if(x > y) return;

    int l = g[x][y]; // l 为根
    cout << l << "␣"; // 先输出 l
    dfs(x, l-1), dfs(l+1, y); // 再输出子树的值
}

...

// 输出答案
dfs(1, n);
```

问题引入 - 过程型状态划分

* Luogu 1057 : 传球游戏
* 上体育课的时候，小蛮的老师经常带着同学们一起做游戏。这次，老师带着同学们一起做传球游戏。
* 游戏规则是这样的：n 个同学站成一个圆圈，其中的一个同学手里拿着一个球，当老师吹哨子时开始传球，每个同学可以把球传给自己左右的两个同学中的一个（左右任意），当老师在此吹哨子时，传球停止，此时，拿着球没有传出去的那个同学就是败者，要给大家表演一个节目。
* 聪明的小蛮提出一个有趣的问题：有多少种不同的传球方法可以使得从小蛮手里开始传的球，传了 m 次以后，又回到小蛮手里。两种传球方法被视作不同的方法，当且仅当这两种方法中，接到球的同学按接球顺序组成的序列是不同的。比如有三个同学 1 号、2 号、3 号，并假设小蛮为 1 号，球传了 3 次回到小蛮手里的方式有 1->2->3->1 和 1->3->2->1，共 2 种。

问题引入 - 过程型状态划分

* Luogu 1057 : 传球游戏
* 聪明的小蛮提出一个有趣的问题：有多少种不同的传球方法可以使得从小蛮手里开始传的球，传了 m 次以后，又回到小蛮手里。两种传球方法被视作不同的方法，当且仅当这两种方法中，接到球的同学按接球顺序组成的序列是不同的。比如有三个同学 1 号、2 号、3 号，并假设小蛮为 1 号，球传了 3 次回到小蛮手里的方式有 1->2->3->1 和 1->3->2->1，共 2 种。

图：一种可能的传球方式

过程型状态划分 - 状态

* **分析**：这道题目十分容易用**搜索**解决。为什么？
* 因为题目已经明确给定了过程：传球的次数。
* 因为这个过程是一定按照**顺序进行**的，所以可以直接写出状态：
* 设状态 `F[i][j]` 为传到第 `i` 次，现在在第 `j` 个人手上的方案数。
* 很显然 `F[i]` 只和 `F[i-1]` 有关；因为题目已经规定好了传递顺序。

* 这一类的过程型问题只需要找出**事情发展的顺序**，就可以很简单的写出状态与转移方程。

过程型状态划分

```
// T26 : 传球游戏
  f[0][1] = 1; // 初值

  for(int i = 1; i <= m; ++ i)
  {
     f[i][1] = f[i-1][2] + f[i-1][n]; // 头与尾需要特殊处理
     f[i][n] = f[i-1][n-1] + f[i-1][1];

     for(int j = 2; j < n; ++ j) // 转移只和 i-1 有关系
        f[i][j] = f[i-1][j-1] + f[i-1][j+1];
  }

  int ans = f[m][1];
```

问题进阶 - 过程型状态划分

* Luogu 1541 ： 乌龟棋
* 乌龟棋的棋盘是一行 N 个格子，每个格子上一个分数（非负整数）。棋盘第 1 格是唯一的起点，第 N 格是终点，游戏要求玩家控制一个乌龟棋子从起点出发走到终点。
* 乌龟棋中 M 张爬行卡片，分成 4 种不同的类型（M 张卡片中不一定包含所有 4 种类型的卡片，见样例），每种类型的卡片上分别标有 1、2、3、4 四个数字之一，表示使用这种卡片后，乌龟棋子将向前爬行相应的格子数。游戏中，玩家每次需要从所有的爬行卡片中选择一张之前没有使用过的爬行卡片，控制乌龟棋子前进相应的格子数，每张卡片只能使用一次。
* 游戏中，乌龟棋子自动获得起点格子的分数，并且在后续的爬行中每到达一个格子，就得到该格子相应的分数。玩家最终游戏得分就是乌龟棋子从起点到终点过程中到过的所有格子的分数总和。

问题进阶 - 过程型状态划分

* Luogu 1541 ： 乌龟棋
* 很明显，用不同的爬行卡片使用顺序会使得最终游戏的得分不同，小明想要找到一种卡片使用顺序使得最终游戏得分最多。现在，告诉你棋盘上每个格子的分数和所有的爬行卡片，你能告诉小明，他最多能得到多少分吗？

- 棋盘：*6 10 14 2 8 8 18 5 17*
- 卡片：*1 3 1 2 1*
- 答案：*73 = 6+10+14+8+18+17*

问题进阶 - 过程型状态划分

* 思考：如何进行搜索？状态该如何设计？
* `dfs(x, c1, c2, c3, c4)` 为当前在第 `x` 个格子上，`ci` 代表标有数字 `i` 的卡片有多少张。
* 于是可以直接写出状态 `F[i][a][b][c][d]`，与状态是一一对应的，表示该状态下，最大的权值是多少。
* 于是，可以和 `F[i-1],F[i-2]...` 进行联系
- 思考：状态转移方程该如何写？