动态规划 C

进阶问题/ 树形 DP / 状态压缩 DP

杨乐

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

2017 年 7 月 17 日

问题引入 - 序列型状态划分

* Luogu 1018 ： 乘积最大
* 设有一个长度为 *N* 的数字串，要求选手使用 *K* 个乘号将它分成 *K+1* 个部分，找出一种分法，使得这 *K+1* 个部分的乘积能够为最大。
* 例如，有一个数字串：*312*，当 *N=3*，*K=1* 时会有以下两种分法：

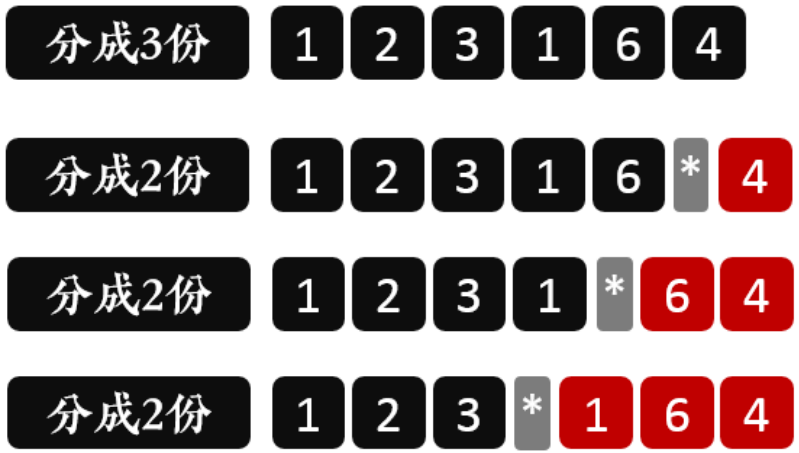
1 *3×12=36*

2 *31×2=62*

* 符合题目要求的结果是：*31×2=62*
* 现在，请你帮助你的好朋友 *XZ* 设计一个程序，求得正确的答案。

乘积最大 - 序列型状态划分

* 题目中要求把整个序列**有序地**切分成 `K+1` 个部分。
* **状态设计**：设我们把前 `F[i][j]` 为前 `i` 个数字切成 `j` 部分所能得到的最大乘积。
* 很显然这时候我们只需要通过**枚举**最后一部分的数字有多大，就能得到结果乘积了。*(满足最优性)*



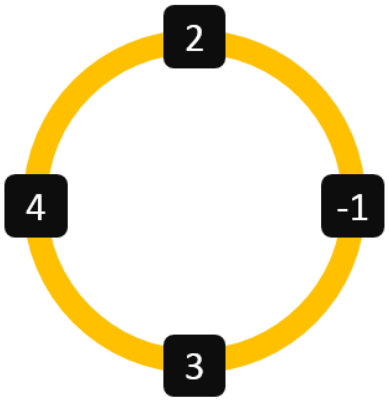
乘积最大 - 序列型状态划分

```
// T20 : 乘积最大（DP）

  ++ k; // K+1

  f[0][0] = 1; // 初值

  for(int i = 1; i <= n; ++ i) // 枚举长度
     for(int j = 1; j <= k; ++ j) // 枚举切割部分
        for(int l = 0; l < i; ++ l) // 枚举前一块的最后位置
           f[i][j] = max(f[i][j], f[l][j-1] * val(l+1, i));

  // 答案
  cout << f[n][k] << endl;
```

问题：为什么不设计状态为 `F[i][j][k]`，表示把 `i` 到 `j` 的数字切成 `k` 块？

问题引入 - 序列型状态划分

* Luogu 1043 : 数字游戏
* 丁丁最近沉迷于一个数字游戏之中。这个游戏看似简单，但丁丁在研究了许多天之后却发觉原来在简单的规则下想要赢得这个游戏并不那么容易。游戏是这样的，在你面前有一圈整数（一共 n 个），你要按顺序将其分为 m 个部分，各部分内的数字相加，相加所得的 m 个结果对 10 取模后再相乘，最终得到一个数 k。游戏的要求是使你所得的 k 最大或者最小。
* 例如，对于下面这圈数字（$n=4$，$m=2$）：
* 要求最小值时，$((2-1) \bmod 10)\times((4+3) \bmod 10)=1\times7=7$，要求最大值时，为 $((2+4+3) \bmod 10)\times(-1 \bmod 10)=9\times9=81$。特别值得注意的是，无论是负数还是正数，对 10 取模的结果均为非负值。



最小值 (2-1) * (3+4) = 7

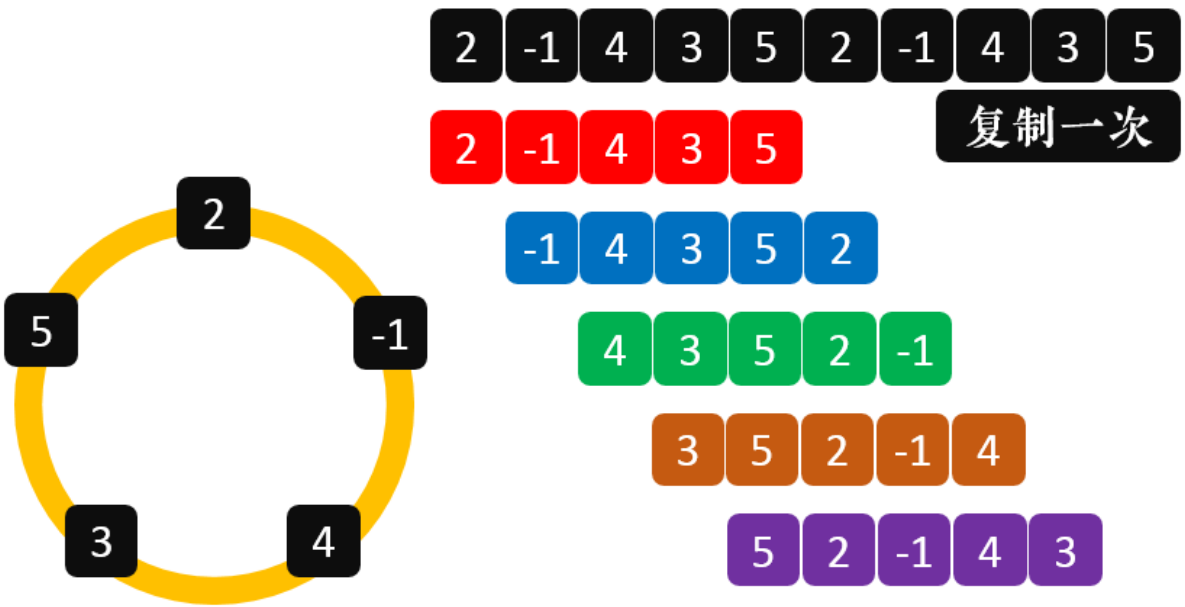
最大值 (-1) * (3+4+2) = 81

数字游戏 - 序列型状态划分

* 题目中要求把整个**环形**序列切分成 m 个部分；先不考虑环形的问题，假若是对于一个**线形**序列，如何做？
* **状态设计**：参考前一道题目的做法，设我们把前 F[i][j] 为前 i 个数字切成 j 部分所能得到的最大乘积。
* 很显然这时候我们只需要通过**枚举**最后一部分的数字有多大，就能得到结果乘积了。*(满足最优性)*

数字游戏 - 序列型状态划分

* **环形序列**：第一个位置不一定是开头，有可能位于序列的中间。
* **解决方法**：枚举每一个位置，把它当作开头算一遍，得到的结果取最大值即为答案。



图：复制序列一次，以每个数为开头算一次

数字游戏 - 序列型状态划分

```
  // T21 : 数字游戏 (DP)
void work(int *a)
{
    s[0] = 0; // 预处理前缀和
    for(int i = 1; i <= n; ++ i) s[i] = s[i-1] + a[i];

    for(int i = 0; i <= n; ++ i)
        for(int j = 0; j <= m; ++ j)
            f[i][j] = 0, g[i][j] = INF; // 初值

    f[0][0] = g[0][0] = 1; // 初值

    ... // DP 过程
```

数字游戏 - 序列型状态划分

```
    // T21 : 数字游戏 (DP)
void work(int *a) {
    ... // 初值，预处理

    for(int i = 1; i <= n; ++ i)
        for(int j = 1; j <= m; ++ j)
            for(int k = 0; k < i; ++ k)
            { // 分别计算最大值和最小值

                // 问题：为什么 f 不用判断而 g 需要判断
                f[i][j] = max(f[i][j], f[k][j-1] * (((s[i] - s[k]) % 10 + 10) %
                    10));
                if(g[k][j-1] != INF) // g[k][j-1] = INF 代表什么
                    g[i][j] = min(g[i][j], g[k][j-1] * (((s[i] - s[k]) % 10 + 10) %
                        10));
            }

    amax = max(amax, f[n][m]);
    amin = min(amin, g[n][m]);
}
```

数字游戏 - 序列型状态划分

```
// T21 : 数字游戏 (DP)

  // 主程序
  cin >> n >> m;

  for(int i = 1; i <= n; ++ i)
      cin >> a[i], a[n+i] = a[i]; // 复制一遍

  amax = 0;
  amin = INF; // 初值

  // 以每个位置开始计算
  for(int i = 1; i <= n; ++ i) work(a + i - 1);

  cout << amin << endl << amax << endl;
```

问题引入 - 序列型状态划分

* Luogu 1063 : 能量项链
* 在 Mars 星球上，每个 Mars 人都随身佩带着一串能量项链。在项链上有 N 颗能量珠。能量珠是一颗有头标记与尾标记的珠子，这些标记对应着某个正整数。并且，对于相邻的两颗珠子，前一颗珠子的尾标记一定等于后一颗珠子的头标记。因为只有这样，通过吸盘（吸盘是 Mars 人吸收能量的一种器官）的作用，这两颗珠子才能聚合成一颗珠子，同时释放出可以被吸盘吸收的能量。如果前一颗能量珠的头标记为 m，尾标记为 r，后一颗能量珠的头标记为 r，尾标记为 n，则聚合后释放的能量为 m*r*n（Mars 单位），新产生的珠子的头标记为 m，尾标记为 n。
* 需要时，Mars 人就用吸盘夹住相邻的两颗珠子，通过聚合得到能量，直到项链上只剩下一颗珠子为止。显然，不同的聚合顺序得到的总能量是不同的，请你设计一个聚合顺序，使一串项链释放出的总能量最大。

问题引入 - 序列型状态划分

* Luogu 1063 ： 能量项链
* 例如：设 *N=4*，*4* 颗珠子的头标记与尾标记依次为 *(2，3) (3，5) (5，10) (10，2)*。我们用记号 ⊕ 表示两颗珠子的聚合操作，*(j⊕k)* 表示第 *j*，*k* 两颗珠子聚合后所释放的能量。则第 *4*、*1* 两颗珠子聚合后释放的能量为：

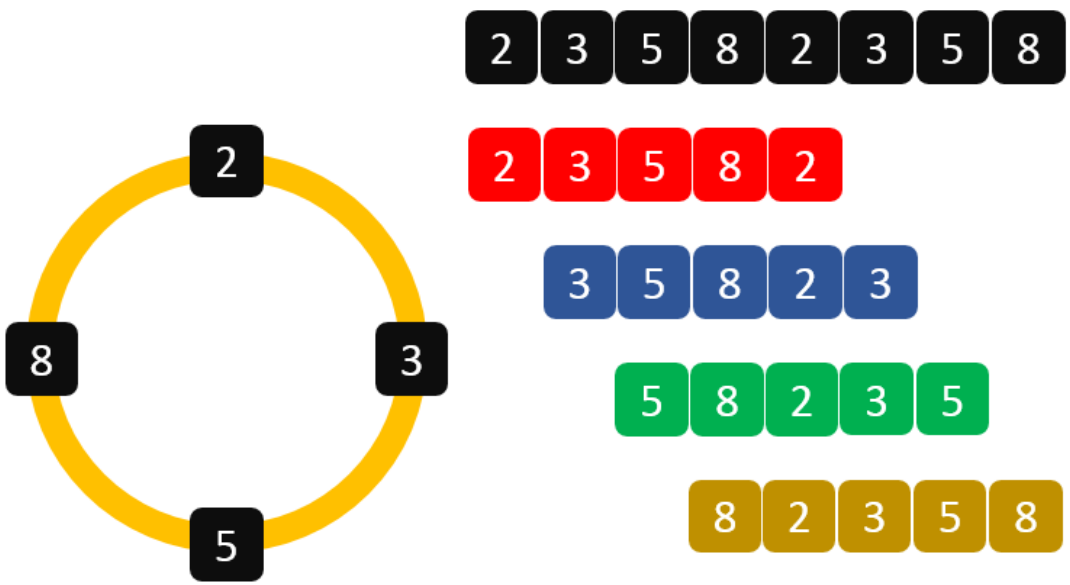
- (4⊕1)=10*2*3=60

* 这一串项链可以得到最优值的一个聚合顺序所释放的总能量为

- ((4⊕1)⊕2)⊕3）=10*2*3+10*3*5+10*5*10=710

能量项链 - 序列型状态划分

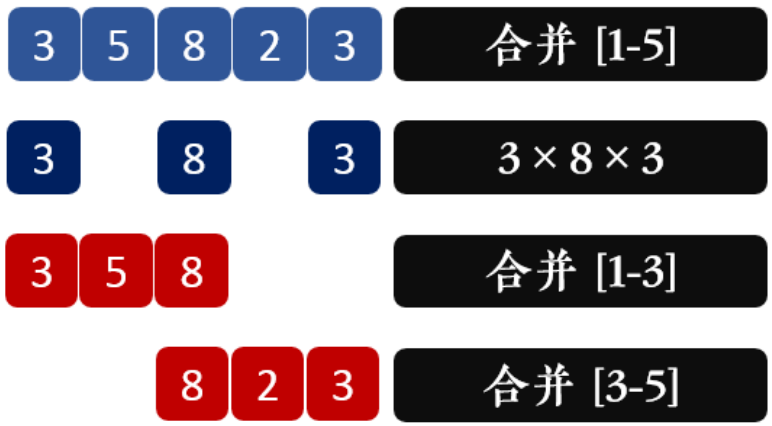
* **环形序列**：枚举每一个位置，把它当作开头算一遍，得到的结果取最大值即为答案。
* 那么我们只需要考虑**线性序列**的问题了.



图：复制序列一次，以每个数为开头算一次

能量项链 - 序列型状态划分

* **状态设计**：设 `F[i][j]` 为把 `i` 到 `j` 之间的珠子合并起来，所能释放的**最大能量**。
* **问题**：为什么不能只设 `F[i]` 代表**前 `i` 个珠子**合并的最大能量。
- 和合并的方式有关。
* **状态转移**：枚举最后一颗和 `i`，`j` 一同合并的珠子 `k`。



图：枚举中间的一颗，最后和 `i`，`j` 一起合并，释放能量

能量项链 - 序列型状态划分

```
// T22 : 能量项链 (DP)

int work(int *a) // 对一段线性序列进行 DP
{
    for(int i = 0; i <= n; ++ i)
        for(int j = 0; j <= n; ++ j) f[i][j] = 0;

    for(int s = 2; s <= n; ++ s) // 动态规划
        for(int i = 0; i + s <= n; ++ i)
        {
            int j = i + s;
            for(int k = i + 1; k < j; ++ k) // 枚举中间的一颗
                f[i][j] = max(f[i][j],
                    f[i][k] + f[k][j] + a[i] * a[k] * a[j]);
        }

    return f[0][n]; // 答案
}
```

问题引入 - 序列型状态划分

* Luogu 1077 : 摆花
* 小明的花店新开张，为了吸引顾客，他想在花店的门口摆上一排花，共 m 盆。通过调查顾客的喜好，小明列出了顾客最喜欢的 n 种花，从 1 到 n 标号。为了在门口展出更多种花，规定第 i 种花不能超过 ai 盆，摆花时同一种花放在一起，且不同种类的花需按标号的从小到大的顺序依次摆列。
* 试编程计算，一共有多少种不同的摆花方案。

- 2 种花，要摆 4 盆
- 第一种花不超过 3 盆，第二种花不超过 2 盆
- 答案：2

摆花 - 序列型状态划分

* **状态设计**：设 `F[i][j]` 为摆到第 `i` 种花，共摆了 `j` 盆，的总方案数
* **转移方程**：?
* **初值/ 边界情况**：?

摆花 - 序列型状态划分

```
// T24 : 摆花 (DP)

  f[0][0] = 1;

  for(int i = 1; i <= n; ++ i)
     for(int j = 0; j <= m; ++ j)
        for(int k = 0; k <= j && k <= a[i]; ++ k)
           f[i][j] = (f[i][j] + f[i-1][j-k]) % MOD;

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

问题引入 - 序列型状态划分

* Luogu 1103 : 书本整理
* *Frank* 是一个非常喜爱整洁的人。他有一大堆书和一个书架，想要把书放在书架上。书架可以放下所有的书，所以 *Frank* 首先将书按高度顺序排列在书架上。但是 *Frank* 发现，由于很多书的宽度不同，所以书看起来还是非常不整齐。于是他决定从中拿掉 *k* 本书，使得书架可以看起来整齐一点。
* 书架的不整齐度是这样定义的：每两本书宽度的差的绝对值的和。例如有 *4* 本书：

- *1x2 5x3 2x4 3x1* 那么 *Frank* 将其排列整齐后是：
- *1x2 2x4 3x1 5x3* 不整齐度就是 *2+3+2=7*
- 已知每本书的高度都不一样，请你求出去掉 *k* 本书后的最小的不整齐度。

书本整理 - 序列型状态划分

* **第一步**：先把书本按照高度排序！
* **状态设计**：从 `N` 本书选出 `N-k` 本书；设 `F[i][j]` 为从前 `i` 本书选出 `j` 本书的**最小的不整齐度**。
* **状态转移**：讨论第 `i` 本书选不选？

书本整理 - 序列型状态划分

* **上一种方法行不通！**
- 为什么？我们需要计算选出相邻两本书之间的**宽度的差的绝对值**。
* **状态设计**：从 `N` 本书选出 `N-k` 本书；设 `F[i][j]` 为从前 `i` 本书选出 `j` 本书的**最小的不整齐度**，并且第 `i` 本书必须要选。
* **状态转移**：上一本书选的是什么？

书本整理 - 序列型状态划分

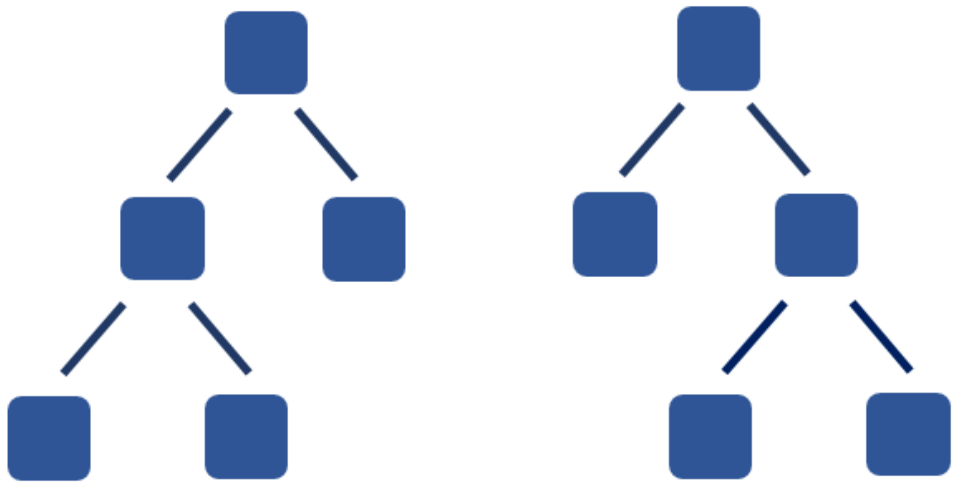
```
// T23 : 书本整理 (DP)

  int m = n - k; // 共选出 m 本书

  for(int i = 1; i <= n; ++ i)
     for(int j = 1; j <= m; ++ j)
        f[i][j] = INF; // 初值

  for(int i = 1; i <= n; ++ i) f[i][1] = 0;
  // 初值，第一本书没有前一本，为 0

  for(int i = 2; i <= n; ++ i)
     for(int j = 2; j <= m; ++ j)
        for(int l = 1; l < i; ++ l) // l 为上一本书
           f[i][j] = min(f[i][j], f[l][j-1] + abs(a[i] - a[l]));

  int ans = INF;
  for(int i = 1; i <= n; ++ i) ans = min(ans, f[i][m]);
  // 答案：最后一本书可以是任意一本
```

问题引入 - 树形 DP

* Luogu 1472 : 奶牛家谱
* 农民约翰准备购买一群新奶牛。在这个新的奶牛群中，每一个母亲奶牛都生两个小奶牛。这些奶牛间的关系可以用二叉树来表示。这些二叉树总共有 N 个节点 $(3 <= N < 200)$。这些二叉树有如下性质：
* 每一个节点的度是 0 或 2。度是这个节点的孩子的数目。
* 树的高度等于 $K(1 < K < 100)$。高度是从根到最远的那个叶子所需要经过的结点数；叶子是指没有孩子的节点。
* 有多少不同的家谱结构？ 如果一个家谱的树结构不同于另一个的，那么这两个家谱就是不同的。输出可能的家谱树的个数除以 9901 的余数。

问题引入 - 树形 DP

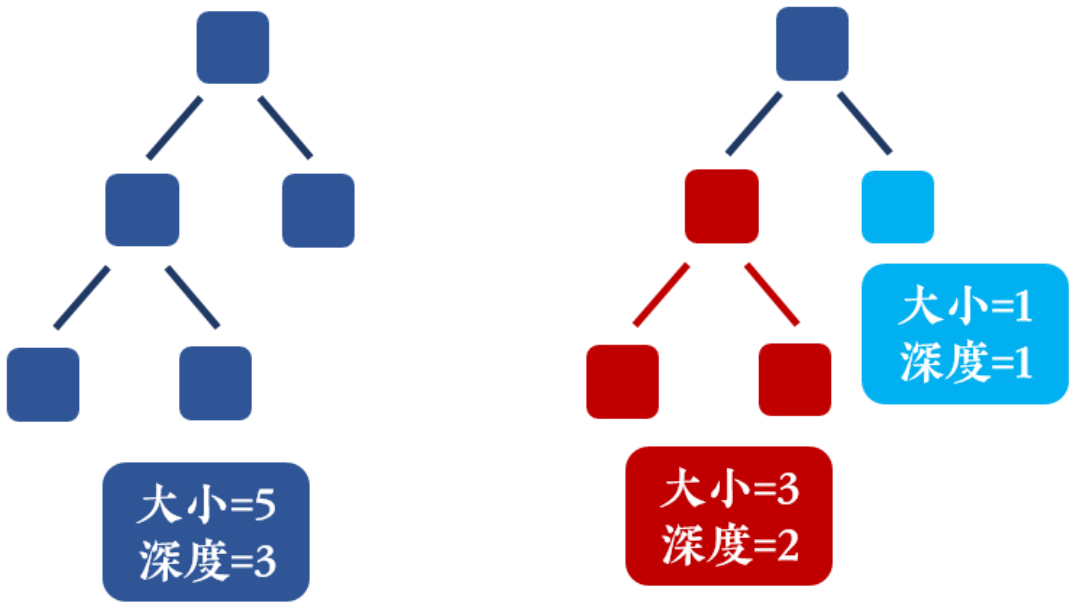
* Luogu 1472 ： 奶牛家谱
  - 5 个节点，高度为 3
  - 答案：可能的家谱树个数为 2



图： 可能的两种树的形态

奶牛家谱 - 树形 DP

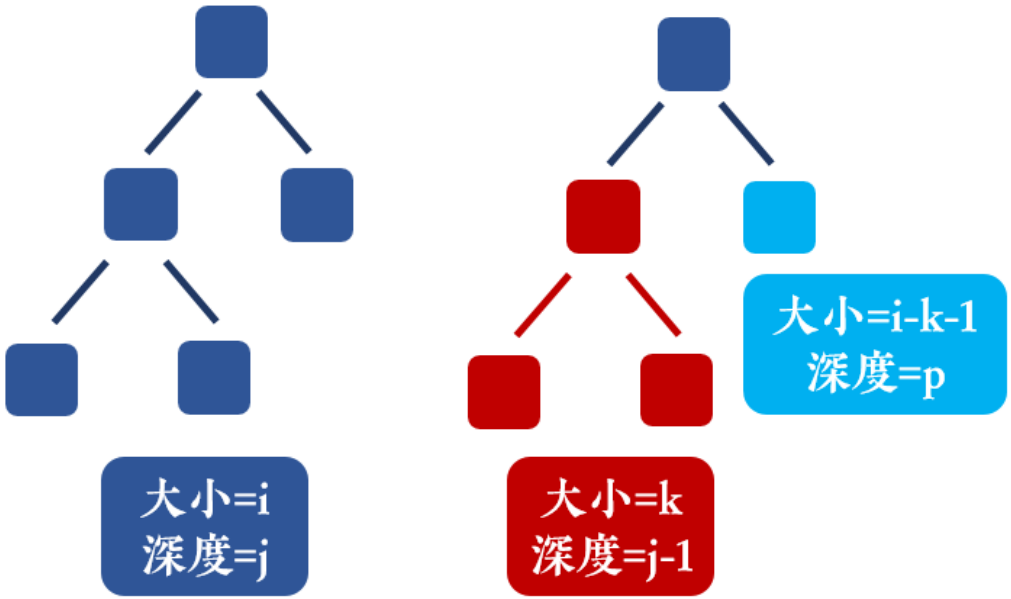
* **状态设计**：看题说话（十分简单），设 `F[i][j]` 为 `i` 个节点高度为 `j` 的树，一共有多少种方案。
- **问题**：如何进行状态转移？
- **分成左右两棵子树**。



图：去掉根后，分成两棵子树

奶牛家谱 - 树形 DP

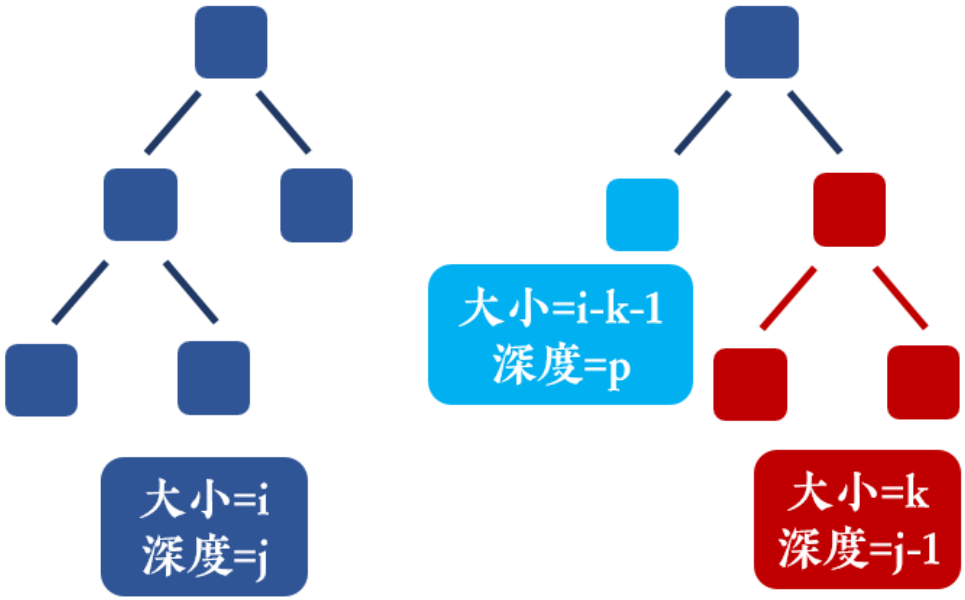
* **状态转移**：枚举子树的状态
- **限制**：注意要满足深度的限制



图：如图所示，必须保证一棵子树的深度为 $j-1$

奶牛家谱 - 树形 DP

* **状态转移**：枚举子树的状态
  - **限制**：注意要满足深度的限制



图：如图所示，右子树也是可以的

奶牛家谱 - 树形 DP

* **状态转移**：F[i][j] = $\sum$ (*k* 为左子树大小，*l* 为右子树深度)

  + F[k][j-1] * F[i-k-1][l] * 2 (l < j-1) (左右子树只有一棵深度为 *j-1*，直接翻倍)

  + F[k][j-1] * F[i-k-1][j-1] (左右子树深度均为 *j-1*，不重复计算)

奶牛家谱 - 树形 DP

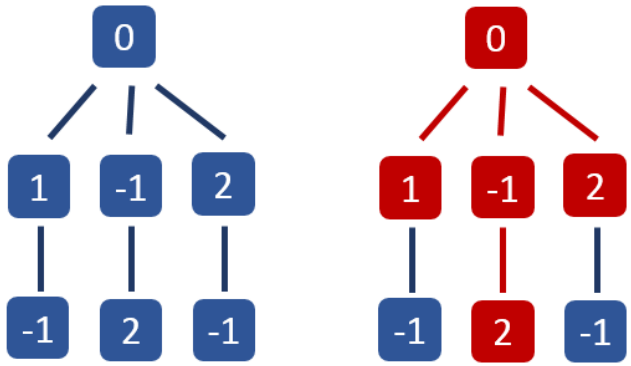
```
// T28 : 奶牛家谱 (DP)

  f[1][1] = 1; // 初值，深度为 1 的子树只有一种情况

  for(int i = 3; i <= n; ++ i)
     for(int j = 2; j <= m; ++ j)
        for(int k = 1; k < i; ++ k)
        {
           for(int l = 1; l < j - 1; ++ l) // l < j-1, 结果乘 2
              f[i][j] = (f[i][j] + f[k][j-1] * f[i-k-1][l] * 2 % MOD) % MOD;

           // 左右子树深度均为 j-1
           f[i][j] = (f[i][j] + f[k][j-1] * f[i-k-1][j-1] % MOD) % MOD;
        }

  int ans = f[n][m]; // 答案
```

奶牛家谱 - 树形 DP - 前缀和优化

```
// T28 : 奶牛家谱 (DP / 前缀和优化)

  f[1][1] = g[1][1] = 1;

  for(int j = 2; j <= m; ++ j) g[1][j] = 1; // 前缀和初始化

  for(int i = 3; i <= n; ++ i)
     for(int j = 2; j <= m; ++ j)
     {
        for(int k = 1; k < i; ++ k)
        {
           // 注意到把深度小于 j-1 的方案全部加起来利用前缀和可以略去枚举过程
           f[i][j] = (f[i][j] + f[k][j-1] * g[i-k-1][j-2] * 2 % MOD) % MOD;
           f[i][j] = (f[i][j] + f[k][j-1] * f[i-k-1][j-1] % MOD) % MOD;
        }
        g[i][j] = (g[i][j-1] + f[i][j]) % MOD; // 前缀和计算
     }

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

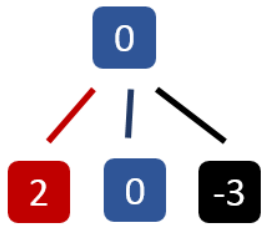
问题引入 - 树形 DP

* Luogu 1122 ： 最大子树和
* 一株奇怪的花卉，上面共连有 *N* 朵花，共有 *N-1* 条枝干将花儿连在一起，并且未修剪时每朵花都不是孤立的。每朵花都有一个“美丽指数”，该数越大说明这朵花越漂亮，也有“美丽指数”为负数的，说明这朵花看着都让人恶心。所谓“修剪”，意为：去掉其中的一条枝条，这样一株花就成了两株，扔掉其中一株。经过一系列“修剪“之后，还剩下最后一株花（也可能是一朵）。老师的任务就是：通过一系列“修剪”（也可以什么“修剪”都不进行），使剩下的那株（那朵）花卉上所有花朵的“美丽指数”之和最大。



最大子树和 - 树形 DP

* **问题转化**：在这棵树中取出若干个连通的节点，使得权值之和最大。
* **观察**：如下图，根节点为 0。假如我们必须要取根节点 0；同时它有三个儿子，权值分别为 2，0，-3；则我们能取得的最大的权值是多少？
  - 贪心地，我们只取不小于 0 的节点。
* **算法思想**：贪心的只取不小于 0 的儿子。
* **状态设计**：设 `F[i]` 为只考虑以 `i` 为根的这棵子树，并且必定要取 `i` 这个点，可能达到的最大权值是多少。
* **状态转移**：把儿子中 `F[x]` 大于 `0` 的加起来即可。



最大子树和 - 树形 DP

```cpp
// T29 : 最大子树和 (DP)

void dfs(int x, int y = 0) // y 代表父亲节点
{
    f[x] = a[x]; // x 节点必取

    for(unsigned i = 0; i < e[x].size(); ++ i)
    {
        int u = e[x][i];
        if(u == y) continue; // 当 u 不为父亲节点

        dfs(u, x); // 递归求解儿子节点的 f 值

        // 当儿子权值大于 0, 则加上
        if(f[u] >= 0) f[x] += f[u];
    }
}
```

最大子树和 - 树形 DP

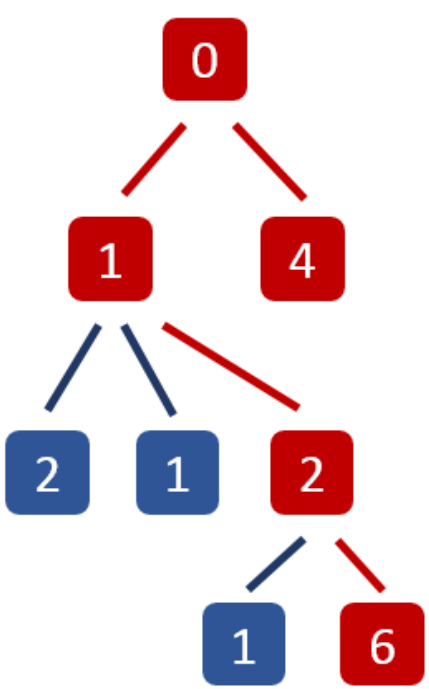
```
// T29 : 最大子树和 (DP)

  // 主程序
  cin >> n;
  for(int i = 1; i <= n; ++ i) cin >> a[i];
  for(int i = 1, x, y; i < n; ++ i)
  {
      cin >> x >> y; // 树边
      e[x].push_back(y); // 增加树边
      e[y].push_back(x); // 增加树边
  }

  dfs(1); // 递归求解 f 值

  int ans = a[1];
  for(int i = 1; i <= n; ++ i) ans = max(ans, f[i]); // 答案取最大的一个
  cout << ans << endl;
```

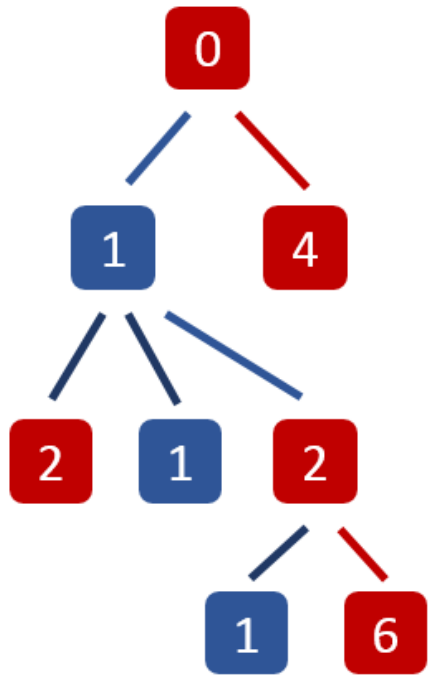
问题引入 - 树形 DP

* Luogu 2014 ： 选课
* 在大学里每个学生，为了达到一定的学分，必须从很多课程里选择一些课程来学习，在课程里有些课程必须在某些课程之前学习，如高等数学总是在其它课程之前学习。现在有 N 门功课，每门课有个学分，每门课有一门或没有直接先修课（若课程 a 是课程 b 的先修课即只有学完了课程 a，才能学习课程 b）。一个学生要从这些课程里选择 M 门课程学习，问他能获得的最大学分是多少？

问题引入 - 树形 DP

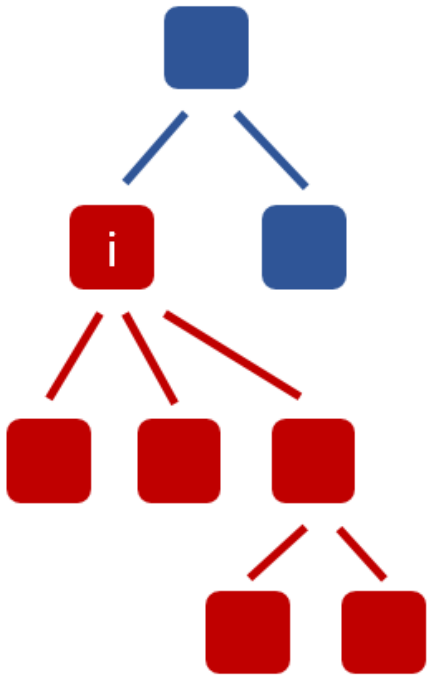
* Luogu 2014 : 选课
* 下图所示：每个节点代表一节课；父亲代表他的先修课；红颜色代表选上的课。左边是一种合法的选课方式；而右边则是一种不合法的选课方式，*2* 的先修课 *1* 没有选上。





问题引入 - 树形 DP

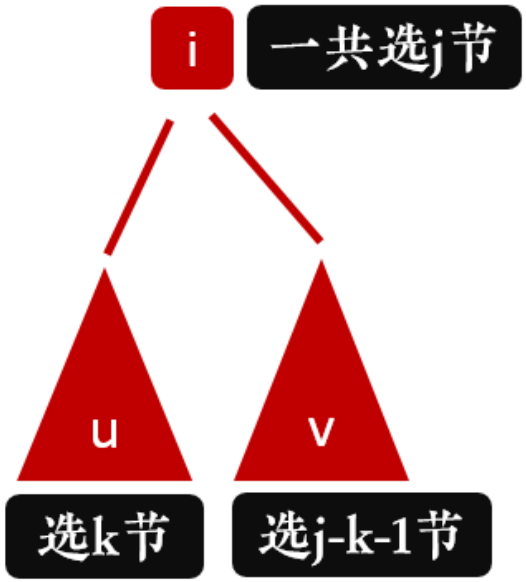
* **问题观察**：我们可以观察得到，根节点的课一定是要选的；并且选的节点是和根节点联通的。*(否则不符合选课规则)*
* **状态设计**：设 `F[i][j]` 为，对于每节点 `i`，只考虑自己的子树，一共选了 `j` 节课，所能得到的最大权值是多少。



问题引入 - 树形 DP

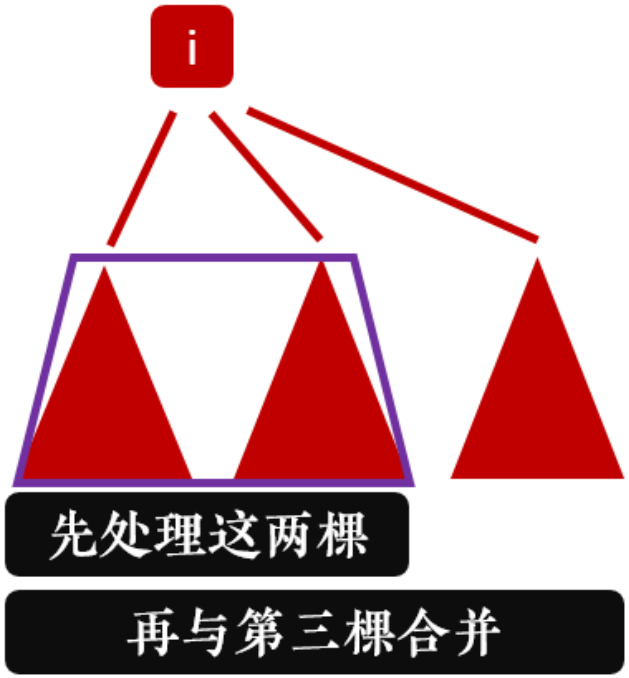
* **状态转移**：假设 i 只有两棵子树，那么我们可以枚举在其中一棵子树中，我们一共选了几门课：

- `F[i][j] = MAX (F[u][k] + F[v][j-k-1]) + A[i]`



问题引入 - 树形 DP

* **状态转移**：假设 `i` 有超过两棵子树，我们可以使用逐次合并的方法：先合并前两棵子树，然后将其**视作一棵**，再与余下的子树继续合并。



选课 - 树形 DP

```
  // T30 : 选课 (DP)
void dfs(int x) // 处理根节点为 x 的子树
{
    int *dp = f[x]; // 小技巧，用 dp 来代替 f[x] 数组

    for(unsigned i = 0; i < e[x].size(); ++ i)
    {
        int u = e[x][i];
        dfs(u); // 处理儿子节点的子树

        // 合并操作
        // 将已经合并的子树信息存放到 dp 数组当中

        for(int j = 0; j <= m; ++ j) tp[j] = 0; // tp: 临时数组
        for(int j = 0; j <= m; ++ j) // 从已经合并的选 j 门
            for(int k = 0; j + k <= m; ++ k) // 从 u 子树中选 k 门
                tp[j + k] = max(tp[j + k], dp[j] + f[u][k]);
        for(int j = 0; j <= m; ++ j) dp[j] = tp[j]; // 复制过来
    }
    ...
```

选课 - 树形 DP

```
// T30 : 选课 (DP)

void dfs(int x) // 处理根节点为 x 的子树
{
    ...

    // 必须要选根节点这一门
    for(int j = m; j; -- j) dp[j] = dp[j-1] + a[x];
    dp[0] = 0;
}
```

选课 - 树形 DP

```
// T30 : 选课 (DP)

  // 主程序
  cin >> n >> m, ++ m;

  for(int i = 1, fa; i <= n; ++ i)
      cin >> fa >> a[i], e[fa].push_back(i);

  // 我们设所有没有先修课的父亲为 0 号
  // 这样结果是一样的，而且必须要选 0 号课程

  dfs(0); // 根节点出发，递归

  cout << f[0][m] << endl; // 在根节点多取 m 门课
```

问题引入 - 状态压缩 DP

* **状态压缩 DP**：利用**位运算**来记录状态，并实现动态规划。
* **问题特点**：数据规模较小；不能使用简单的算法解决。

位运算 - 二进制表示

* 大家应该了解**二进制**与**十进制**之间的转换；

- $(23)_{10} = (10111)_2$
- $(64)_{10} = (1000000)_2$
- $(55)_{10} =?$
- $(10101)_2 =?$

位运算 - 布尔运算

* 给定两个**布尔变量**a,b，他们之间可进行布尔运算：
* **与运算 and**：当 a,b 均为真的时候，**a 与 b**为真；
* **或运算 or**：当 a,b 均为假的时候，**a 或 b**为假；
* **非运算 not**：当 a 均为真的时候，**非 a**为假；
* **异或运算 xor**：当 a,b 不是同时真，或者同时假的时候，**a 异或 b**为真。

and	0	1
0	假	假
1	假	真

or	0	1
0	假	真
1	真	真

xor	0	1
0	假	真
1	真	假

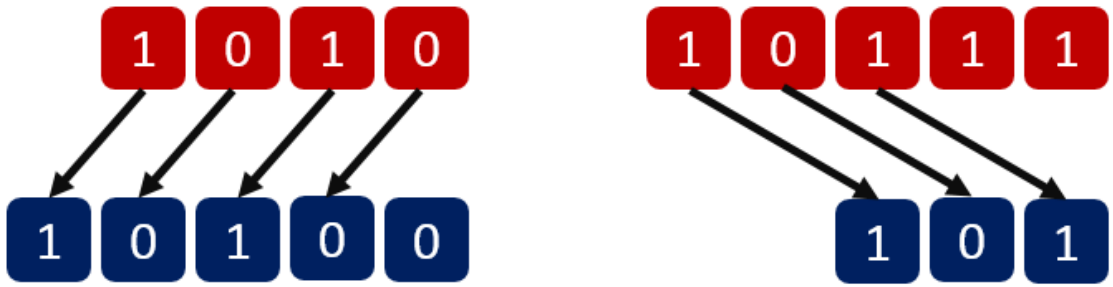
位运算 - 整数运算

* 我们把 `0` 视作布尔的假，`1` 视作布尔的真，则整数亦存在二进制的运算，而运算的结果则是二进制里对应的位作**相应的布尔操作**。
* $(23)_{10} = (10111)_2, (10)_{10} = (1010)_2$；
* **与运算 and**：23 **and** $10 = (10)_2 = 2$
* **或运算 or**：23 **or** $10 = (11111)_2 = 31$
* **异或运算 xor**：23 **xor** $10 = (11101)_2 = 29$

位运算 - 位移

* 另外，在整数位运算中还有**位移**操作：
* $(23)_{10} = (10111)_2, (10)_{10} = (1010)_2$；
* **左移 «**：将整个二进制向左移动若干位，并用 `0` 补充；
- $10 << 1 = (10100)_2 = 20$
* **右移 »**：将整个二进制向右移动若干位(实际最右边的几位就消失了)；

  $23 >> 2 = (101)_2 = 5$



问题引入 - 状态压缩 DP

* **状态压缩 DP**：利用**二进制位**来记录状态，并实现动态规划。
* **问题特点**：数据规模较小；不能使用简单的算法解决。

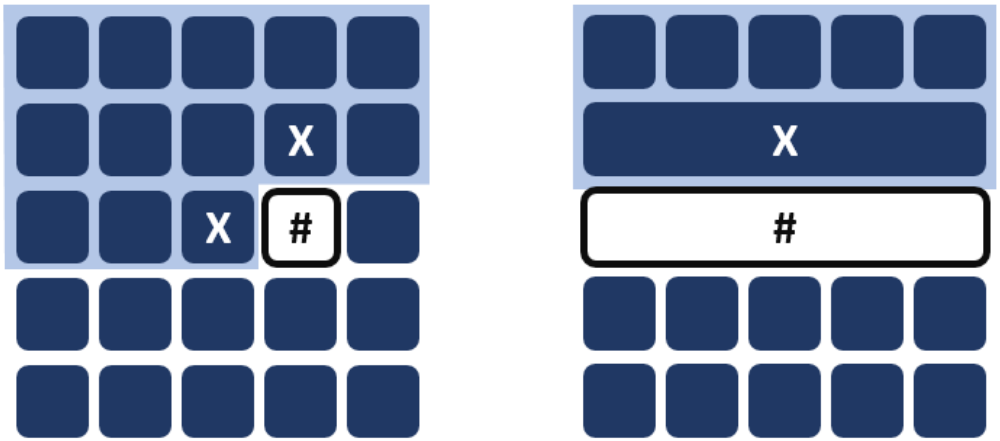
问题引入 - 状态压缩 DP

* Luogu 1879 ： 玉米田
* 农场主 *John* 新买了一块长方形的新牧场，这块牧场被划分成 *M* 行 *N* 列 *(1 ≤ M ≤ 12; 1 ≤ N ≤ 12)*，每一格都是一块正方形的土地。*John* 打算在牧场上的某几格里种上美味的草，供他的奶牛们享用。
* 遗憾的是，有些土地相当贫瘠，不能用来种草。并且，奶牛们喜欢独占一块草地的感觉，于是 *John* 不会选择两块相邻的土地，也就是说，没有哪两块草地有公共边。
* *John* 想知道，如果不考虑草地的总块数，那么，一共有多少种种植方案可供他选择？（当然，把新牧场完全荒废也是一种方案）

- *1 1 1*
- *0 1 0* (*1* 代表这块土地适合种草)
- 总方案数：*9*

玉米田 - 状态压缩 DP

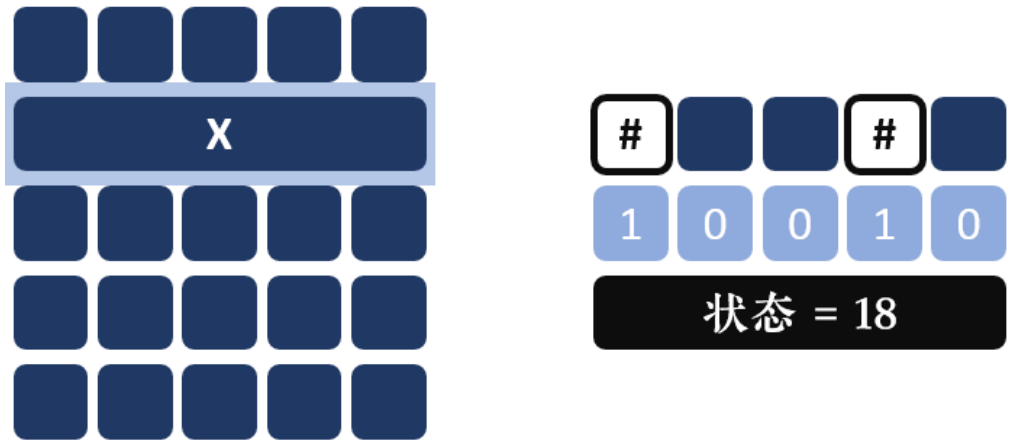
* **问题限制**：没有哪两块草地有公共边
* **考虑动态规划**：如何**划分状态**？
  - 从上到下，从左到右？
  - 记录 `F[i][j]` 为 `(i,j)` 选择在这个位置种草的方案数？
  - 问题：如何转移？我们只能限制左边的位置不能种草；不能限制上面的位置不能种草。
* **划分状态**：考虑用行来划分状态；第 `i` 行的状态只取决于第 `i-1` 行。



图：两种不同的状态划分方式

玉米田 - 状态压缩 DP

* **状态表示**：如何表示一行的种草状态？
* **二进制位**：将一整行看作是一个大的二进制数，其上为 1 表示种了草，0 表示没有种草。
* 这样，我们可以用**一个数**来表示**一行的种草状态**；



图：例如，在一行的第一个位置和第四个位置种草，则可以用 *18* 来表示这种状态

玉米田 - 状态压缩 DP

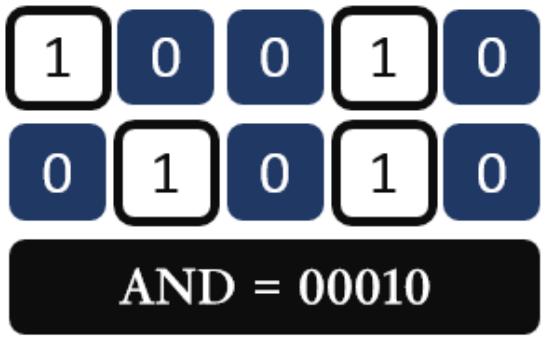
* **状态设计**：设 `F[i][j]` 为已经处理到第 `i` 行，且第 `i` 行的状态是 `j`，的总方案数。*(其中 j 为一个大的二进制数)*
* **状态转移**：枚举前一行的种草状态 `k`，如果**没有产生冲突**，则加上

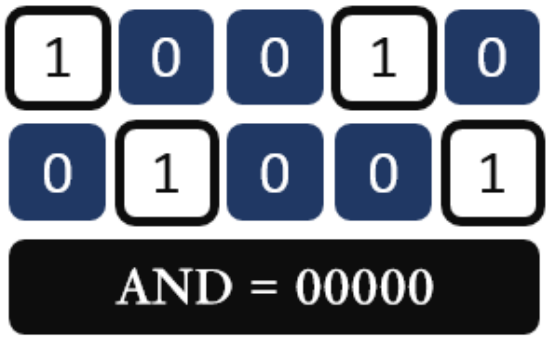
$$F[i][j] = \sum F[i-1][k]$$

* **问题**：如何判断两个状态是否冲突？如何运用二进制运算？

玉米田 - 状态压缩 DP

* **问题**：如何判断两个状态是否冲突？如何运用二进制运算？
* **解决方案**：采取**与运算**；如果两个状态是冲突的，则肯定有一位上都为 1，and 的结果肯定不为 0；否则若结果为 0，则代表状态不冲突.





* **下一个问题**：如何判断当前状态是合法的？

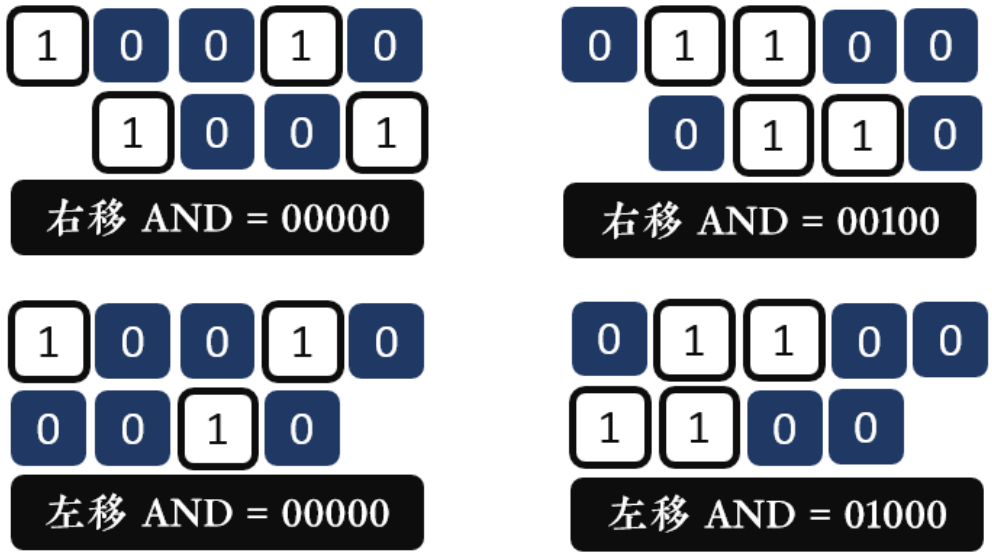
玉米田 - 状态压缩 DP

* **问题**：如何判断当前状态 `j` 是合法的？

1 **没有占有障碍物**：和判断两个状态是否冲突是类似的。

2 **左右位置判断**：与上一行判断冲突，我们只考虑到了**上下位置不可能同时种草**，但还没有考虑**左右的情况**。

- **解决方案**： `j & (j«1) = 0` 且 `j & (j»1) = 0`



玉米田 - 状态压缩 DP

```
// T31 : 玉米田（状态压缩 DP）

const int N = 13; // 边长
const int S = (1 << N) + 5; // 二进制状态数

int a[N][N]; // 棋盘数组
int s[N]; // 每一行空地的状态

int f[N][S]; // 动态规划数组
int g[S]; // 标记一个状态是否合法

// 主过程

ts = 1 << m; // 总状态数
for(int i = 0; i < ts; ++ i) // 判断左右位置
   g[i] = ((i & (i<<1)) == 0) && ((i & (i>>1)) == 0);
```

玉米田 - 状态压缩 DP

```
// T31 : 玉米田（状态压缩 DP）

  f[0][0] = 1; // 初值!

  for(int i = 1; i <= n; ++ i) // 枚举行
     for(int j = 0; j < ts; ++ j) // 枚举这行的状态
        if(g[j] && ((j & s[i]) == j)) // 状态本身合法且不占障碍物

        for(int k = 0; k < ts; ++ k) // 枚举上一行的状态
           if((k & j) == 0) // 如果不冲突
              f[i][j] = (f[i][j] + f[i-1][k]) % MOD;

  int ans = 0;
  for(int j = 0; j < ts; ++ j)
     ans = (ans + f[n][j]) % MOD; // 答案加起来
```

玉米田 - 状态压缩 DP

```
// T31 : 玉米田（状态压缩 DP）

  // 读入棋盘状态
  cin >> n >> m;
  for(int i = 1; i <= n; ++ i)
     for(int j = 1; j <= m; ++ j) cin >> a[i][j];

  // 预处理每一行的状态
  for(int i = 1; i <= n; ++ i)
     for(int j = 1; j <= m; ++ j)
        s[i] = (s[i] << 1) + a[i][j];
```

问题引入 - 状态压缩 DP

* Luogu 1896 ： 互不侵犯
* 在 $N \times N$ 的棋盘里面放 K 个国王，使他们互不攻击，共有多少种摆放方案。国王能攻击到它上下左右，以及左上左下右上右下八个方向上附近的各一个格子，共 8 个格子。

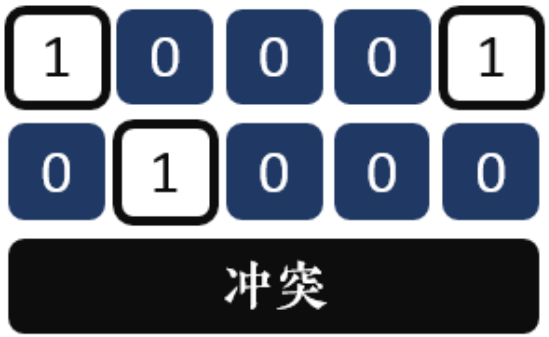
- 3 2 (在 3×3 的棋盘内放置 2 个国王)
- 总方案数：16

互不侵犯 - 状态压缩 DP

* **状态设计**：设 `F[i][j][k]` 为已经处理到第 `i` 行，且第 `i` 行的状态是 `j`，现在一共放置了 `k` 个国王的总方案数。*(其中 j 为一个大的二进制数)*
* **状态转移**：枚举前一行的放置国王状态为 `l`，如果**没有产生冲突**，则加上*(其中 x 为状态 j 的国王数)*

$$F[i][j][k] = \sum F[i-1][l][k-x]$$

* **问题**：如何判断两个状态是否冲突？如何运用二进制运算？



互不侵犯 - 状态压缩 DP

```
// T32 : 互不侵犯（状态压缩 DP）

const int N = 10;
const int S = (1 << N) + 5; // 二进制状态数

int cnt[S]; // 记录一个状态有多少个 1
int g[S]; // 记录一个状态是否合法
int h[S];

LL f[N][S][N*N]; // 动态转移状态
```

互不侵犯 - 状态压缩 DP

```
// T32 : 互不侵犯（状态压缩 DP）

  ts = 1 << n;
  for(int i = 0; i < ts; ++ i)
  {
      g[i] = ((i & (i<<1)) == 0) && ((i & (i>>1)) == 0);
      h[i] = i | (i << 1) | (i >> 1); // h: 一个状态把它左右都占据之后的状态
      cnt[i] = cnt[i>>1] + (i&1); // 计算多少个 1
  }
```

互不侵犯 - 状态压缩 DP

```
// T32 : 互不侵犯（状态压缩 DP）

  f[0][0][0] = 1; // 初值

  // 顺推
  for(int i = 0; i < n; ++ i) // 行数
     for(int j = 0; j < ts; ++ j) // 这一行状态
        for(int l = 0; l <= k; ++ l) if(f[i][j][l]) // 枚举个数
           for(int p = 0; p < ts; ++ p) // 枚举下一行
              if(g[p] && ((h[p] & j) == 0))
                 // 如果是合法状态，且不冲突
                 f[i+1][p][l + cnt[p]] += f[i][j][l];

  LL ans = 0; // 答案
  for(int j = 0; j < ts; ++ j) ans += f[n][j][k];
```

问题引入 - 状态压缩 DP

* Codeforces 453B : **Little Pony and Harmony Chest**
* 我们称一个正整数序列 B 是和谐的，当且仅当它们两两互质。
- 给出一个序列 A，求出一个和谐的序列 B，使得$\sum |A_i - B_i|$ 最小。

CF453B - 状态压缩 DP

* **问题思考**：一个和谐的序列两两互质，满足什么性质？
  - 它们分解质因数后，没有两个数拥有**相同的质因子**。
* **状态构思**：如果我已经决定完前 `i` 个数是多少，现在要决定下一个数，需要满足什么限制？——不能有和前面重复的质因子。
  - 这代表我们需要记录下，目前已经用了哪些质因子了。
* **状态设计**：设 `F[i][j]` 为，已经决定完前 `i` 个数是多少，而且**已经用了状态为 `j` 的质因子了**，现在的**最小权值差**是多少。
  - 如何状态压缩？`2,3,5,7,11,...`；一个代表一个二进制位；用了是 `1`，没有是 `0`。

CF453B - 状态压缩 DP

```
// T35 : CF453B （状态压缩 DP）

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

  // 预处理质数
  for(int i = 2; i < M; ++ i)
  {
      if(!fg[i]) p[++p[0]] = i;
      for(int j = 1; j <= p[0] && i * p[j] < M; ++ j)
          fg[i * p[j]] = 1;
  }

  // 预处理每个数的质因子所代表的状态
  for(int i = 1; i < M; ++ i)
  {
      g[i] = 0;
      for(int j = 1; j <= p[0]; ++ j)
          if(i % p[j] == 0) g[i] |= 1 << (j-1);
  }
```

CF453B - 状态压缩 DP

```
// T35 : CF453B （状态压缩 DP）
  // 动态规划主过程
  int ns = 1 << p[0];
  // 初值：最大值
  for(int i = 1; i <= n + 1; ++ i)
     for(int j = 0; j < ns; ++ j) f[i][j] = S;
  f[1][0] = 0;

  for(int i = 1; i <= n; ++ i) // 枚举位置
     for(int j = 0; j < ns; ++ j) if(f[i][j] < S) // 枚举状态
        for(int k = 1; k < M; ++ k) // 枚举这个位置的数
           if((g[k] & j) == 0) // 如果之前没有出现过
           {
              // 计算最优值
              int t = f[i][j] + absp(k - a[i]);
              if(t < f[i+1][g[k] | j]) // 更新最优值
                 f[i+1][g[k] | j] = t,
                 opt[i+1][g[k] | j] = k;
           }
```

CF453B - 状态压缩 DP

```
// T35 : CF453B （状态压缩 DP）

  // 最优值输出

  int ansp = S;
  int ansm = 0;

  for(int j = 0; j < ns; ++ j) // 记录最优值所对应的状态
     if(f[n+1][j] < ansp) ansp = f[n+1][j], ansm = j;

  for(int i = n+1; i > 1; -- i) // 依次向前查询
  {
     b[i-1] = opt[i][ansm];
     ansm ^= g[b[i-1]];
  }

  for(int i = 1; i <= n; ++ i) cout << b[i] << "␣";
  cout << endl;
```

问题引入 - 状态压缩 DP

* Luogu 2831 : 愤怒的小鸟
* *NOIP2016* 提高组
- 题目太长，请大家网上查看

愤怒的小鸟 - 状态压缩 DP

* **状态设计**：能设计形如 `F[i]`，表示前 `i` 只小猪都打下来的最小的小鸟数吗？
  - 无法这样做。我一次可以打下若干只小鸟，而且没有顺序之分
* **状态设计**：用一个大的二进制数 `s` 表示哪些小猪被打下来了*(1 表示打下来了，0 表示没有打下来)*。设 `F[s]` 为打下状态为 `s` 的小猪，**最小的小鸟数**。
* **状态转移**：采用**顺推**。考虑下一步我们可以打下状态为 `t` 的小猪，则可以这样更新：`F[s or t] = MIN (F[s or t], F[s] + 1)`
  - *`s or t` 表示我现在一共打下了 `s` 和 `t` 状态的小猪。*

愤怒的小鸟 - 状态压缩 DP

```
// T33 : 愤怒的小鸟（状态压缩 DP）

  // DP 主过程

  int ns = 1<<n;
  for(int i=1; i<ns; i++) dp[i] = n; // 初值

  for(int i=0; i<ns; i++)
  {
     int l = (ns-1) ^ i; // 异 ^ 或操作；选出没有打下的小猪
     l = mk[l & (-l)]; // l 表示第一只没被打下的小猪

     for(int j=1; j<=g[l][0]; j++) // 选择一种打的方式
        dp[i | g[l][j]] = min(dp[i | g[l][j]], dp[i] + 1);
        // g[l][j] 为能够打到的小猪的状态
  }

  printf("%d\n", dp[ns-1]); // 答案
```

总结 - 状态压缩 DP

* **状态压缩 DP**：用一个二进制数来记录状态
* **特点**：记录一些东西**是否出现过**
* **难点**：位运算判断；状态的设计
* **标志**：数据范围不大