动态规划

一、基本概念

阶段：将所求问题恰当地分成若干个相互联系的过程。

状态：每个阶段所面临的条件，同一阶段可能会有不同的状态。

决策(转移)：对于一个阶段的某个状态，从该状态演变到下一阶段的某个状态的选择。

边界：决策过程中的初始情况。

策略：由每个阶段所做的所有决策组成的序列称为策略。所有可行策略中使得目标达到最佳情况的策略称为最优策略。

而动态规划则是求解一类多阶段决策最优化(最优策略)问题的数学方法。DP 的思想很大程度上源于递归。将原问题分解成许多小的子问题求解并进行决策的转化策略。利用搜索递归求解问题并用记忆化数组进行优化。搜索时用到的参数即为状态，搜索中的分支便是转移。找出重叠子问题后，将搜索递归改为利用循环并按一定顺序进行递推决策，这种实现形式通常便称为 DP。状态的递推式称为状态转移方程。利用递归改写的方程一般为倒推，通过改变状态描述可以写成顺推

DP 的基本思想：建立子问题（也就是状态的描述），找出状态间的转移关系，再根据具体问题选择记忆化搜索或是递推的形式实现。不同的状态设计或是定义描述，对应的转移方程也往往不同，而状态转移方程需要注意递推的顺序与边界。

并不是所有状态都能正确求解，通常需要满足一些条件：

最优化原理：将原问题转化为规模更小的子问题时，原问题最优当且仅当子问题最优。

最优子结构：若最优化原理在一个问题中成立，则称这个问题具有最优子结构。

无后效性：决策只取决于当前状态中的因素，而与到达此状态的方式无关。

例子：若背包问题不记录当前已使用的容量，则状态有后效性，不能正确求解。

往往可以通过增加状态的维数，记录更多的关键信息来满足无后效性与最优子结构，但维数的增加会导致重叠子问题减少而影响时间效率。

复杂度 = 状态数 × 决策数目 × 转移费用

复杂度 = 状态数 + 总转移费用

无后效性与最优子结构保证了 DP 正确性，而重叠子问题则是效率的关键。同样有状态、转移方程、无后效性等特点但无最优性决策过程的递推往往也纳入 DP，如计算方案数。

相比于严谨的定义，通过解决各类问题与模型能更加深入理解。

背包 DP

一、01 背包

有 n 个物品，每个物品有其重量 $w[i]$ 与价值 $v[i]$，求在总重量不超过 W 的情况下，能选出的物品的最大价值和。

$f[i][j]$ 表示前 i 个物品中，选出的重量总和为 j 的情况下，能得到的最大价值和。$O(nW)$ 的时间复杂度转移即可。

$$f[i][j]=\max\{f[i-1][j-w[i]]+v[i],f[i-1][j]\}$$

$f[i][j]$ 表示前 i 个物品中，选出的价值总和为 j 的情况下，所需要的最小重量。$O(n\sum v_i)$ 的时间复杂度转移即可。

两种状态表示根据题目范围来选择。若倒序枚举第二维，则第一维的 i 可以省略不记录。

二、完全背包

有 n 种物品，每种物品有其重量 $w[i]$ 与价值 $v[i]$，且每种物品数量无限，求在总重量不超过 W 的情况下，能选出的物品的最大价值和。

与 01 背包类似，$f[i][j]$ 表示前 i 个物品中，选出的重量总和为 j 的情况下，能得到的最大价值和。$O(nW)$ 的时间复杂度转移即可，

$$f[i][j]=\max\{f[i][j-w[i]]+v_i,f[i-1][j]\}$$

由于物品数量无限，所以若要省略第一维，则第二维需要顺序枚举。

三、多重背包

有 n 种物品，每种物品有其重量 $w[i]$ 与价值 $v[i]$ 以及这种物品的个数 $c[i]$，求在总重量不超过 W 的情况下，能选出的物品的最大价值和。

$f[i][j]$ 表示前 i 个物品中，选出的重量总和为 j 的情况下，能得到的最大价值和。$O(\sum c_iW)$ 的时间复杂度转移即可。

$$f[i][j]=\max\{f[i-1][j-k\cdot w[i]]+k\cdot v[i]\mid 0\le k\le c_i\}$$

后面我们会进一步说明如何对多重背包进行优化。

树形 DP

一、简介

顾名思义，树型动态规划就是在“树”的数据结构上的动态规划，平时作的动态规划都是线性的或者是建立在图上的，线性的动态规划有二种方向即向前和向后，相应的线性的动态规划有二种方法既顺推与逆推，而树型动态规划是建立在树上的，树中的父子关系天然就是个递归（子问题）结构，所以也相应的有二个方向：

1.叶—>根，即根的子节点传递有用的信息给根，之后由根得出最优解的过程。这种方式 DP 的题目应用比较多。

2.根—>叶：这种方式相对于上面那种较少出现，一般是在树形 DP“换根”中常用，即需要取所有点作为一次根节点进行求值，此时父亲得到了整棵树的信息，只需将其中这个儿子的 DP 值的影响去除，然后再转移给这个儿子，这样就能达到根->叶的顺序。

共性总结

1）动态规划的顺序:一般按照后序遍历的顺序，即处理完儿子再处理当前节点，才符合树的子结构的性质。

2）多叉树转换为二叉树:由于要分配附加维到各个节点，而分配附加维是个划分问题，若还是按当前节点到各个儿子节点分配，则成了一个整数划分问题,O(n^2)。所以要把多叉树转换为二叉树，这样才能按动态规划的方式只决策当前点的分配问题，O(n)。

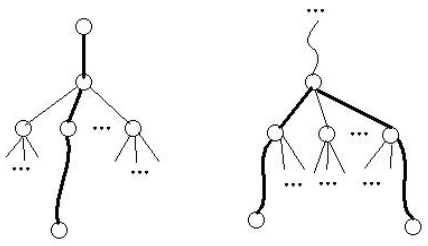
3）加当前点的选或不选的常数维:加此维解决的是后效性问题。

4)复杂度:树型动态规划复杂度基本上是 O(n)；若有附加维 m，则是 O(nm)。

二、经典问题

最长链问题：给定一棵 n 个点的边带权树，求它的最长链，n<=10^5.

一棵有根树的最长链，可能出现如下图的两种情况



设 $dep[i]$ 表示以节点 i 为根的子树的最大深度（到叶子的最远距离）。

$f[i]$ 表示以节点 i 为根的子树中，包含节点 i 的最长链长度

我们有 $dep[i]=\max\{dep[j]+w[i][j]\}$，其中 j 是 i 的子节点。

以及：$f[i]=\max\{dep[i],dep[j]+w[i][j]+dep[k]+w[i][k]\}$ 其中 j,k 是 i 的子节点，且 $j\neq k$。

不难发现，我们的状态转移方程是按照从下至上的顺序计算的。

做一遍 DFS 遍历，在回朔的时候分别计算 dep 和 f 的值

关于 f 值的计算：由于节点 j 和 k 之间没有关联，所以我们只需要选择 $dep[j]+w[i][j]$ 最大的两个子节点进行累加即可。

树形 DP 考虑链（路径）时，常常在一个点上，考虑在这个点子树内、过这个点的路径，也就是一条路径会在它路径中深度最浅的那个节点上被考虑。

整个算法的复杂度为 O(|V|+|E|)=O(n)

树的中心问题：给出一棵边带权的树，求树中的点，使得此点到树中的其他结点的最远距离最近。

第一次从儿子节点转移给父亲：$f[i]=\max\{f[son]+w[i][son]\}$

第二次从父亲节点转移给儿子：$f[i]=\max\{f[i],f[fa[i]]+w[fa[i]][i]\}$。

但是存在一个问题就是如果 $f[fa[i]]$ 的值是从 i 那里得到的，这样计算显然就错了。不要放弃，在实际操作过程中，f 需要记下两个值，一个是最优值，一个是次优值，这两个值必须由不用的子结点得到。这样当最优值发生矛盾的时候，次优值一定不会矛盾。问题就解决了。复杂度 O(N)十分的理想。

这就是常见的需要“换根”的树形 DP。这里换根时为了消除自己影响而记录了次大值。

普通树形 DP：给定一棵树，现在要从中选出最少的节点，使得所有边至少有一个端点，这个端点是被选出的。

按照要求构建一棵树。对于这类最值问题，向来是用动态规划求解的。

点的取舍可以看作一种决策，那么状态就是在某个点取的时候或者不取的时候，以他为

根的子树的最小代价。分别可以用 $f[i][1]$ 和 $f[i][0]$ 表示。

当这个点不取的时候，他的所有儿子都要取，所以 $f[i][0]=\sum_{j\in son[i]} f[j][1]$

取的时候，他的所有儿子取不取无所谓，不过当然应该取最优的一种情况。所以

$$f[i][1]=(\sum_{j\in son[i]}\min\{f[j][1],f[j][0]\})+1$$

普通的树形 DP 中，常常会采用叶->根的转移形式，根据父亲的状态，来确定子节点的状态，若子节点有多个，则需要枚举过去，将子节点（子树）的 DP 值合并。

树形 DP 还有一个重要拓展是与各类树形数据结构进行结合。例如 Trie 上的 DP、AC 自动机上的 DP、后缀自动机上的 DP 等。

有时我们的图可以不简单限制于树，在树的基础上进行简单扩展，也可以得到一些能用 DP 解决的例子，例如环+外向树（在有根树的基础上，添加了一条某节点指向根的边的图）上的 DP、仙人掌（每条边至多存在于一个简单环上）上的 DP 等。

三、例题

骑士 （BZOJ 1040）

题目大意：

给定 n 个人的价值以及每个人最痛恨的人（每个人都有且仅有一个最痛恨的人），现在要从 n 个人中选出若干人，不能同时选择一个人与他最痛恨的人，求所有合法方案中，选出的人的最大价值和。n <= 10^6

分析：

若每个人由他最痛恨的人向他连边，那么题目就转为给出了一棵基环外向树。若原题是一棵树，那么这就是基础的树形 DP（父亲和儿子不可同时选），而现在这棵树中会出现一个唯一的环，那么常用的解决方法是选择环中一条边断开，将图变为有根树，选择这条边某个端点当根且强制选根进行 DP。由于这样强制了某个点不选，所以我们还需让那个点当根也进行一次 DP。（相当于选了相邻的某条边并断开这个环）。

代码：

```
1.  #include <algorithm>
```

```cpp
#include <iostream>
#include <cstring>
#include <cstdio>
#include <cmath>
using namespace std;

typedef long long ll;
const int N = 1e6 + 3;
int n, val[N], fa[N], cnt[N];
int tot, info[N], nxt[N], go[N];
bool vis[N];
ll ans, f[N][2];

inline void SetE(const int &x, const int &y) {
    nxt[++tot] = info[x]; info[x] = tot; go[tot] = y;
    fa[y] = x;
}

inline void Dfs(const int &x) {
    int y; vis[x] = true;
    f[x][1] = val[x];
    for (int k = info[x]; y = go[k], k; k = nxt[k])
        if (!vis[y]) {
            Dfs(y);
            f[x][0] += max(f[y][0], f[y][1]);
            f[x][1] += f[y][0];
        }
}

inline void Dp(int x) {
    int rt, y;
    for (rt = x; cnt[rt] != x; rt = fa[rt])
        cnt[rt] = x;
    Dfs(rt);

    x = fa[rt];
    f[x][1] = f[x][0];
    for (x = fa[x]; x != rt; x = fa[x]) {
        f[x][0] = 0; f[x][1] = val[x];
        for (int k = info[x]; y = go[k], k; k = nxt[k]) {
            f[x][0] += max(f[y][0], f[y][1]);
            f[x][1] += f[y][0];
        }
    }
```

```
    f[rt][1] = val[rt];
    for (int k = info[rt]; y = go[k], k; k = nxt[k])
        f[rt][1] += f[y][0];
    ans += max(f[rt][0], f[rt][1]);
}

char ch;
inline int read() {
    while (ch = getchar(), ch < '0' || ch > '9');
    int res = ch - 48;
    while (ch = getchar(), ch >= '0' && ch <= '9')
       res = res * 10 + ch - 48;
    return res;
}

int main() {
    n = read();
    for (int i = 1; i <= n; ++i) {
        val[i] = read();
        SetE(read(), i);
    }
    for (int i = 1; i <= n; ++i)
        if (!vis[i]) Dp(i);
    printf("%lld\n", ans);
    return 0;
}
```

区间 DP

一、线性模型

问题：给定 n 个数 $a[i]$，将这些数分成若干组，每组大小不小于 k，每组代价为 C + (组内极差)^2，C 是常量，求最小代价，n <= 2000.

显然数字排序后，每一组一定是连续的一段，设 $f[i]$ 表示前 i 个数分组后的最小代价。

$$f[i]=\max\{f[j-1]+(a[i]-a[j])^2+C\mid i-j+1\geq k\}$$

即处理一维空间上按顺序进行连续一段分组的问题常用 DP。

但有时我们的问题虽然也是连续一段进行分组，但更加复杂。

二、问题引入

给定长为 n 的序列 $a[i]$，每次可以将连续一段回文序列消去，消去后左右两边会接到一起，求最少消几次能消完整个序列，$n\leq 500$

与线性模型不同，这里消去的顺序是任意的，且消完后左右会接起来。但我们发现，不管消去的顺序是什么，每个时刻被消去的位置总是一段段连续区间。

考虑消去区间 $[l,r]$ 时，若 $a[l],a[r]$ 不在一起消去，则总能找到一个分界点 k，使得我们能先消完 $[l,k]$ 再去消 $[k+1,r]$。注意这里是只考虑消 $[l,r]$，因此不考虑和外面一起消去。

若 $a[l],a[r]$ 在一起消去，则它们只要接在 $[l+1,r-1]$ 这段区间中最后一次消除时那个回文序列的左右即可。

$f[l][r]$ 表示消去区间 $[l,r]$ 需要的最少次数。

容易发现这样问题就具有了最优子问题结构且状态满足无后效性。

$$f[i][j]=\min\{f[i][k]+f[k+1][j]\mid i\leq k\leq j\}$$

若 $a[l]=a[r]$，则还有 $f[i][j]=\max\{f[i][j],f[i+1][j-1]\}$

这里实际上是以区间长度为阶段的，这种 DP 我们通常称为区间 DP。

区间 DP 的做法较为固定，即枚举区间长度，再枚举左端点，之后枚举区间的断点进行转移。一般使用 DP 时的特征比较明显，就像上述例子中的，每次可以消一个区间，之后左

右两边会合并起来。有时问题不像上述例子那样，消去整个区间可以拆分成每次消去一对位置对称的元素，那么转移时，$f[l][r]$ 的初值则就需要进行提前计算，赋为一次消去它的值。

三、例题

压缩（BZOJ 1068）

题目大意：

Description

给一个由小写字母组成的字符串，我们可以用一种简单的方法来压缩其中的重复信息。压缩后的字符串除了小写字母外还可以（但不必）包含大写字母R与M，其中M标记重复串的开始，R重复从上一个M（如果当前位置左边没有M，则从串的开始算起）开始的解压结果（称为缓冲串）。 bcdcdcdcd可以压缩为bMcdRR，下面是解压缩的过程：

已经解压的部分	解压结果	缓冲串
b	b	b
bM	b	
bMc	bc	c
bMcd	bcd	cd
bMcdR	bcdcd	cdcd
bMcdRR	bcdcdcdcd	cdcdcdcd

另一个例子是abcabcdabcabcdxyxyz可以被压缩为abcRdRMxyRz。

Input

输入仅一行，包含待压缩字符串，仅包含小写字母，长度为n。

Output

输出仅一行，即压缩后字符串的最短长度。

N <= 50

分析：

区间 DP，转移时考虑当前区间的压缩方式即可。（分两段进行压缩或将两段合并压缩）。

代码：

```
#include <algorithm>
#include <iostream>
#include <cstring>
#include <cstdio>
#include <cmath>
using namespace std;

const int N = 54;
```

```
int n, f[N][N][2];
char s[N];

inline bool check(const int &i, const int &j, const int &k) {
    for (int l = 0; l < k; ++l)
        if (s[i + l] != s[j + l]) return false;
    return true;
}

inline void upt(int &x, const int &y) {
    if (x > y) x = y;
}

int main() {
    scanf("%s", s + 1);
    n = strlen(s + 1);
    memset(f, 127 / 3, sizeof(f));
    for (int i = 1; i <= n; ++i) f[i][i][0] = f[i][i][1] = 1;
    for (int l = 1; l <= n; ++l)
        for (int i = 1; i <= n; ++i) {
            int j = i + l;
            if (j > n) break;
            upt(f[i][j][0], l + 1);
            upt(f[i][j][1], l + 1);
            for (int k = i; k < j; ++k) {
                upt(f[i][j][0], f[i][k][0] + j - k);
                upt(f[i][j][1], f[i][k][1] + j - k);
                upt(f[i][j][1], f[i][k][1] + f[k + 1][j][1] + 1);
            }
            if (l & 1) {
                int k = i + j >> 1, t = l + 1 >> 1;
                if (check(i, k + 1, t))
                    upt(f[i][j][1], f[i][k][0] + 1), upt(f[i][j][0], f[i][k][0] + 1);
            }
        }
    printf("%d\n", min(f[1][n][0], f[1][n][1]));
    return 0;
}
```

状压 DP

一、简介

状压 DP，即基于状态压缩的动态规划，又叫集合动态规划。顾名思义，这是一类以集合信息为状态的特殊的动态规划问题。主要有传统集合动态规划和基于连通性状态压缩的动态规划两种。下面先来介绍一下传统的状压 DP。

一般的动态规划往往着眼于整体，提取出两三个关键信息，并依此划分阶段使得问题具备无后效性及最优子结构性质，然后根据已知信息依次对各个阶段进行决策。然而有时候一般的状态描述难以满足无后效性原则，或者保存的信息不足够进行决策，许多元素的状态都直接影响到决策，都需要被考虑到。为每一个元素的状态都开一维数组来存储显然行不通，于是就需要一种便于识别和操作的方式记录下各个元素的状态，即对多个元素的状态进行压缩存储，于是基于状态压缩的动态规划应运而生。

基于状态压缩的动态规划是一种以集合内的元素信息作为状态，状态总数为指数级别的动态规划，它有以下两个特点：1、具备动态规划的两个基本性质：最优性原理和无后效性；2、数据规模的某一维或者几维非常小。

下面将通过一个简单的例子来引出基于状态压缩的动态规划的基本思路和做法。

二、问题引入

售货员的难题

某乡有 n 个村庄(1<n<15)，有一个售货员，他要到各个村庄去售货，各村庄之间的路程 s(0<s<1000)是已知的，且 A 村到 B 村与 B 村到 A 村的路大多不同。为了提高效率，他从商店出发到每个村庄一次，然后返回商店所在的村，假设商店所在的村庄为 1 号村庄，请你帮他选择一条最短的路。

这是一个非常经典的问题——求最小哈密尔顿回路，即 TSP（旅行商问题），已被证明是 NP 完全问题，无多项式时间复杂度算法。那是否意味着只能通过搜索算法来解决呢？显然不是。$O(n!)$ 的搜索复杂度实在难以承受 n≤15 的数据范围。仔细思考不难发现任何时候我们只需要知道哪些点已经被遍历过而遍历点的具体顺序对以后的决策是没有影响的，因此不妨以当前所在的位置 i，遍历过的点的集合 S 为状态作动态规划：

$f(i,S)$ 表示现在已经访问的顶点的集合（起点出发后当作还未访问）为 S，当前所在的顶点为 i，从 i 出发访问剩余所有的顶点，最终回到起点的路径的长度最小值。初值为：$f[0][\{0\}]=0$ 转移方程即为：（倒推形式）

$$f(i,S)=\min\{f(j,S-\{i\})+dist(j,i)\,|\,i,j\in S且i\neq j\}$$

那么集合 S 应该如何表示呢？

这个递推式中，这个下标是集合而不是普通整数因此需要稍加处理。即我们可以考虑用某种方式将其编码成一个整数。

考虑到每个点有两种状态：已访问过和未访问过，可分别用 1 和 0 表示，那么所有的 n 个点的访问情况可以用一个长度为 n 的 01 串来表示，而每一个 01 串都唯一地对应着一个二进制数。于是我们就想到用一个二进制数 $mask$ 来表示集合 S，其中 i 点在 S 中当且仅当 $mask$ 中的第 i 位为 1。这里的 $mask$ 实际上就是一个压缩了 n 个点的信息的状态，所以这样的 DP 称为状态压缩 DP。

将状态压成二进制数后，我们可以利用位运算大大方便计算与维护。

对于不是整数的下标，有时难以确定一个合适的递推顺序，往往需要利用递归搜索来实现。但用二进制数压缩状态时，对于两个整数 i,j 若它们对应的集合 $S(i)\leq S(j)$ 则也有 $i\leq j$，所以我们可以利用循环简单枚举进行 DP。

状态数为 $O(n2^n)$，转移开销为 n，整个动态规划的时间复杂度为 $O(n^2 2^n)$，虽然为指数级算法，但是对于 n = 15 的数据规模来说已经比朴素的的搜索算法高效很多了。至此，问题已得到圆满解答。

三、位运算

首先是对于 1 位二进制数，取值只有 01，它们间常用的位运算有：

与运算 and（C++中&）：0 & 0 = 0，0 & 1 = 0，1 & 0 = 0，1 & 1 = 1

或运算 or(C++中|)：0 | 0 = 0，0 | 1 = 1，1 | 0 = 1，1 | 1 = 1

异或运算 xor(C++中^)：0 ^ 0 = 0，0 ^ 1 = 1，1 ^ 0 = 1，1 ^ 1 = 0

可以类比逻辑运算来记忆结果。（与就是都为 1 才为 1，或就是有一个 1 就是 1，异或则是不一样则是 1）。

而对于多位二进制数（也就是普通整数），上述操作则均为按位操作，即二进制数中一

位位处理，每一位的结果单独计算出来，这也是常用的技巧：上述三种位运算每位结果独立。

此外，我们还有取反操作 not（C++中~）：按位取反，即每位 0 变 1，1 变 0.需要注意的是它会将有符号整数的符号位也取反。

左移操作 lsh(C++中<<)：mask << k，其中 mask 是个二进制数，k 是个整数，这个操作是将二进制数 mask 的每一位向左（高位）移动 k 位，即结果的第 x 位是原来的第 x-k 位。而低位的最后 k 位补 0.

右移操作 rsh(C++中>>)：mask >> k，其中 mask 是个二进制数，k 是个整数，这个操作是将二进制数 mask 的每一位向右（低位）移动 k 位，即结果的第 x 位是原来的第 x+k 位。而高位的起始 k 位补 0.

容易发现，若把 num << k 中的 num 看做十进制数，那么结果=num * 2^k 而 num >> k 的结果则是=num / 2^k（取下整）。

用位运算，我们可以快速的进行一些对二进制数中某些位的操作，而在状压 DP 中，就是对集合中的元素的状态进行操作：（下述描述中我们默认位从 0 开始）

1.取出二进制数 x 的第 k 位，存储在 y 中：

```
y = 1 << k & x  或是  y = x >> k & 1
```

2.取出二进制数 x 的最后一个 1（也就是 lowbit），存储在 y 中：

```
y = x & -x
```

3.将二进制数 x 的第 k 位设为 1：

```
x = x | 1 << k
```

4.将二进制数 x 的第 k 位设为 0：

```
x = x & ~(1 << k)
```

5.将二进制数 x 的第 k 位取反：

```
x = x ^ 1 << k
```

6.取出二进制数 x 的最后 k 位，存储在 y 中：

```
y = x & (1 << k) - 1
```

7.将二进制数 x 低位连续的 1 变为 0：

```
x = x & x + 1
```

8.将二进制数 x 低位连续的 0 变为 1：

```
x = x | x - 1
```

9.取出二进制数 x 低位连续的 1，存储在 y 中：

y = (x ^ x + 1) >> 1

10. 快速枚举出集合 S（用二进制数 mask 来表示）的所有子集。

```
for (int subset = mask; subset != 0; subset = (subset - 1) & mask);
```

使用二进制时，可能会因为不熟悉二进制运算符与普通运算符的优先级，而导致代码的执行顺序与自己所想不一，因此这里再介绍一下 C++中各类运算符的操作优先级。

优先级	操作符	描述	例子	结合性

1	()
	[]
	->
	.
	::
	++
	--
		调节优先级的括号操作符
		数组下标访问操作符
		通过指向对象的指针访问成员的操作符
		通过对象本身访问成员的操作符
		作用域操作符
		后置自增操作符
		后置自减操作符
			(a + b) / 4;
			array[4] = 2;
			ptr->age = 34;
			obj.age = 34;
			Class::age = 2;
			for(i = 0; i < 10; i++) ...
			for(i = 10; i > 0; i--) ...
				从左到右

2	!
	~
	++
	--
	-
	+
	*
	&
	(type)
	sizeof
		逻辑取反操作符
		按位取反(按位取补)
		前置自增操作符
		前置自减操作符
		一元取负操作符
		一元取正操作符
		解引用操作符
		取地址操作符
		类型转换操作符
		返回对象占用的字节数操作符
			if(!done) ...
			flags = ~flags;
			for(i = 0; i < 10; ++i) ...
			for(i = 10; i > 0; --i) ...
			int i = -1;
			int i = +1;
			data = *ptr;
			address = &obj;
			int i = (int) floatNum;
			int size = sizeof(floatNum);
				从右到左

3	->*	在指针上通过指向成员的指针	ptr->*var = 24;	从左

	.* 	访问成员的操作符 在对象上通过指向成员的指针访问成员的操作符	obj.*var = 24;	到右

4	*
	/
	%
		乘法操作符
		除法操作符
		取余数操作符
			int i = 2 * 4;
			float f = 10 / 3;
			int rem = 4 % 3;
				从左到右

5	+ -	加法操作符 减法操作符	int i = 2 + 3; int i = 5 - 1;	从左到右
6	<< >>	按位左移操作符 按位右移操作符	int flags = 33 << 1; int flags = 33 >> 1;	从左到右

7	<
	<=
	>
	>=
		小于比较操作符
		小于或等于比较操作符
		大于比较操作符
		大于或等于比较操作符
			if(i < 42) ...
			if(i <= 42) ...
			if(i > 42) ...
			if(i >= 42) ...
				从左到右

8	== !=	等于比较操作符 不等于比较操作符	if(i == 42) ... if(i != 42) ...	从左到右
9	&	按位与操作符	flags = flags & 42;	从左到右
10	^	按位异或操作符	flags = flags ^ 42;	从左到右
11	|	按位或操作符	flags = flags | 42;	从左到右
12	&&	逻辑与操作符	if(conditionA && conditionB) ...	从左到右
13	||	逻辑或操作符	if(conditionA || conditionB) ...	从左到右
14	? :	三元条件操作符	int i = (a > b) ? a : b;	从右

				到左

15	=
	+=
	-=
	*=
	/=
	%=
	&=
	^=
	|=
	<<=
	>>=
		赋值操作符
		复合赋值操作符(加法)
		复合赋值操作符(减法)
		复合赋值操作符(乘法)
		复合赋值操作符(除法)
		复合赋值操作符(取余)
		复合赋值操作符(按位与)
		复合赋值操作符(按位异或)
		复合赋值操作符(按位或)
		复合赋值操作符(按位左移)
		复合赋值操作符(按位右移)
			int a = b;
			a += 3;
			b -= 4;
			a *= 5;
			a /= 2;
			a %= 3;
			flags &= new_flags;
			flags ^= new_flags;
			flags |= new_flags;
			flags <<= 2;
			flags >>= 2;
				从右
				到左

16	,	逗号操作符	for(i = 0, j = 0; i < 10; i++, j++) ...	从左 到右

结合性有两种，一种是自左至右，另一种是自右至左，大部分运算符的结合性是自左至右，只有单目运算符、三目运算符的赋值运算符的结合性自右至左。

优先级有 15 种。记忆方法如下：

记住一个最高的：构造类型的元素或成员以及小括号。

记住一个最低的：逗号运算符。

剩余的是**一、二、三、赋值**。

意思是**单目、双目、三目和赋值运算符**。

在诸多运算符中，又分为：

算术、关系、逻辑。

两种位操作运算符中，移位运算符在算术运算符后边，逻辑位运算符在逻辑运算符的前面。再细分如下：

算术运算符分 *，/，%高于+，-。

关系运算符中，〉，〉=，<，<=高于==，！=。

逻辑运算符中，除了逻辑求反（！）是单目外，逻辑与（&&）高于逻辑或（||）。

逻辑位运算符中，除了逻辑按位求反（~）外，按位与（&)高于按位半加（^)，高于按

位或（|）。

这样就将 15 种优先级都记住了，再将记忆方法总结如下：

去掉一个最高的，去掉一个最低的，剩下的是一、二、三、赋值。双目运算符中，顺序为算术、关系和逻辑，移位和逻辑位插入其中。

四、问题 2

下面回到状压 DP 中，再来看看状态压缩在覆盖模型问题上的应用。

多米诺游戏

题目大意：有一个 m 行 n 列的矩阵，用 1*2 的骨牌(可横放或竖放)完全覆盖，骨牌不能重叠，有多少种不同的覆盖的方法？只需求出覆盖方法总数 mod p 的值即可。m<=5 ，p<=10000 ， n<=10000.

从左往右按列转移，我们发现，每一列可能的摆放情况都受到前一列每一个格子覆盖情况的影响，即前一列每个格子的情况都应被考虑到。同例一，我们同样可以用一个 01 序列表示整列格子的覆盖情况：1 表示已被覆盖，0 表示未被覆盖。又注意到数据范围 m<=5，为我们用一个二进制数表示整列格子状态创造了条件。如下图，可用 $(10101)_2$ 即 21 来表示该状态。

这样，我们就可以用一个二维数组 $f[i][S]$ 其中 $i \le n, S < 2^m$ 来记录前 $i-1$ 列全部覆盖，第 i 列覆盖情况为 S 的方案数。

转移：考虑到 1 个骨牌有两种铺法——横与竖。横铺要求前一列改行未被覆盖，竖铺则同时覆盖该列相邻的两个。竖着放的可以通过预处理列举出所有情况

t[8] = {0,3,6,12,24,15,30,27};

对于任意一个S，若 t[k] and S = t[k]，则S中的$t[k]$状态可以由竖放的骨牌完成，剩余的覆盖部分 S xor t[k]则由横置的骨牌完成,其对应的上一列状态就应该为(2^m - 1)xor(S xor t[k])。所以就有：

$$f[i][S]=\sum_{1\le k\le 8}^{t[k]and\ S=t[k]} f[i-1][(2^m-1)\,xor\,(S\,xor\,t[k])]$$

状态数$O(n\times 2^m)$,转移代价为 8，时间复杂度为$O(8n\times 2^m)$.

状态压缩的动态规划的本质和一般的动态规划是相同的，而它的最大特点就是它的状态不再是孤立的一个元素,而是多个元素的集合体。这样就能为接下来的决策提供更多信息。另外我们要注意到这种动态规划的状态大多都是指数级别的。所以我们常常需要对状态进行合理的优化。

五、例题

互不侵犯（BZOJ 1087）

题目大意：

在 N×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。

1 <= N <= 9 , 0 <= K <= N*N

分析：

注意到 N 很小，所以我们若一行一行放入国王，则上一行国王的状态以及这一行国王的状态我们都是能利用二进制数枚举出来，而由于国王只影响周围八格，这给我们利用位运算判断两个状态是否矛盾带来了有利条件：

((x << 1) & y) == 0 且 ((x >> 1) & y) == 0) 且 (x & y == 0)

$f[i][j][S]$表示前i行放入了j个国王且第i行摆放情况为S的方案数。

枚举下一行的状态进行转移即可。

代码：

```
#include <algorithm>
#include <iostream>
```

```cpp
#include <cstring>
#include <cstdio>
#include <cmath>
using namespace std;

typedef long long ll;
const int N = 10, M = 26, P = 512;
int n, m, t, g[P];
ll ans, f[N][P][M];

int main() {
  scanf("%d%d", &n, &m);
  if (m > 25 || m >= n * n) puts("0");
  else {
    t = 1 << n; f[0][0][0] = 1; g[0] = 0;
    for (int i = 1; i < t; ++i) g[i] = g[i >> 1] + (i & 1);
    for (int i = 1; i <= n; ++i)
      for (int j = 0; j < t; ++j)
        if (g[j] <= m && !(j & j >> 1))
          for (int k = 0; k < t; ++k)
            if (g[k] <= m && !(k & k >> 1) && !(k & j) && !(j & k >> 1) && !(j & k << 1))
              for (int l = g[j] + g[k]; l <= m; ++l)
                f[i][j][l] += f[i - 1][k][l - g[j]];
    for (int i = 0; i < t; ++i)
      ans += f[n][i][m];
    cout << ans << endl;
  }
  return 0;
}
```

数位 DP

一、简介

在信息学竞赛中，有一类与数位有关的区间统计问题，这类问题往往具有比较浓厚的数学味道，但用简单的组合数学知识不足以简单解决，更无法暴力求解。此时我们需要借用 DP 的思想，以数位为阶段，在数位上进行递推，这即是数位 DP。

如这样一类问题：求给定区间中，满足给定条件的某个 D 进制数或此类数的数量。所求的限定条件往往与数位有关，例如数位之和、指定数码个数、数的大小顺序分组等等。题目给定的区间往往很大，无法采用朴素的方法求解。此时，我们就需要利用数位的性质，设计 log 级别复杂度的算法。解决这类问题最基本的思想就是“逐位确定”的方法。有时求解这类问题时还需要使用预处理，而预处理的过程也可以视作一个数位 DP。

二、问题引入

求给定区间$[L,R]$中满足下列条件的整数个数：这个数恰好等于K个互不相等的B的整数次幂之和。$1\le L\le R<2^{31}-1,1\le K\le 20,2\le B\le 10$

例如，设 L=15，R=20，K=2，B=2，则有且仅有下列三个数满足题意：

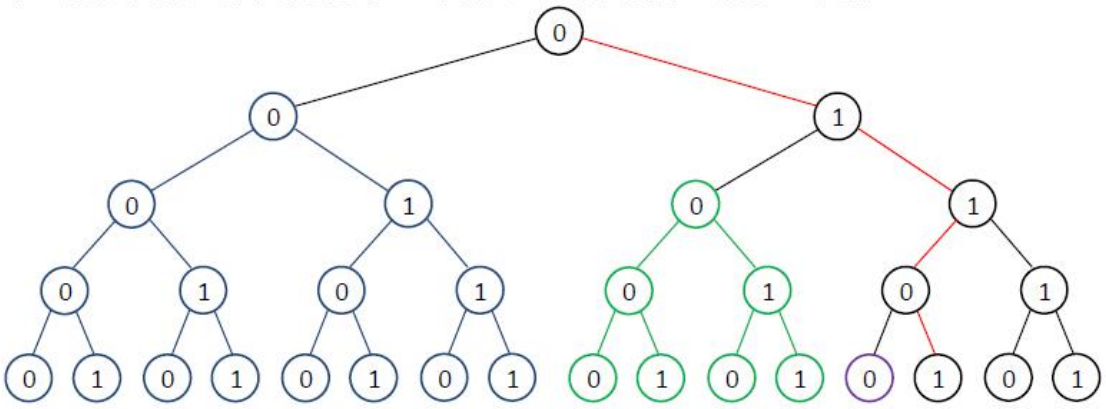
17 = 2^4 + 2^0 , 18 = 2^4 + 2^1 , 20 = 2^4 + 2^2

所求的数为互不相等的幂之和，亦即其 B 进制表示的各位数字都只能是 0 和 1。因此，我们只需讨论二进制的情况，其他进制都可以转化为二进制求解。

很显然，数据范围较大，不可能采用枚举法。我们发现二进制下数位只有 log 级别，因此我们考虑从数位上下手。

实际上这类问题求解$[L,R]$的答案，而这个值是满足区间减法的，即我们可以求$[0,R]$的答案$ansR$，再求出$[0,L-1]$的答案$ansL$，则$ansR-ansL$即为我们所求的答案。所以我们现在只需要考虑数的上界的限制，这也是数位 DP 中最常用的技巧。

假设 n=13，其二进制表示为 1101，K=3。我们的目标是求出 0 到 13 中二进制表示含 3 个 1 的数的个数。为了方便思考，让我们画出一棵高度为 4 的完全二叉树：



为了方便起见，树的根用 0 表示。这样，这棵高度为 4 的完全二叉树就可以表示所有 4 位二进制数$[0,2^4-1]$，每一个叶子节点代表一个数。其中，红色路径表示 n。所有小于 n 的数组成了三棵子树，分别用蓝色、绿色、紫色表示。因此，统计小于 13 的数，就只需统计这三棵完整的完全二叉树：统计蓝子树内含 3 个 1 的数的个数、统计绿子树内含 2 个 1 的数的个数（因为从根到此处的路径上已经有 1 个 1），以及统计紫子树内含 1 个 1 的数的个数。注意到，只要是高度相同的子树统计结果一定相同。而需要统计的子树都是“右转”时遇到的。当然，我们不能忘记统计 n 本身。实际上，在算法最初时将 n 自加 1，可以避免讨论 n 本身，但是需要注意防止上溢。

剩下的问题就是，如何统计一棵高度为i的完全二叉树内二进制表示中恰好含有 j 个 1 的数的个数。这很容易用递推求出：设$f[i][j]$表示所求，则分别统计左右子树内符合条件数的个数，有 $f[i][j]=f[i-1][j]+f[i-1][j-1]$。

这样，我们就得出了询问的算法：首先预处理 f，然后对于输入n，我们在假想的完全二叉树中，从根走到 n 所在的叶子，每次向右转时统计左子树内数的个数。

最后的问题就是如何处理非二进制。对于询问 n，我们需要求出不超过 n 的最大 B 进制数，表示只含 0、1 的数：找到 n 的左起第一位非 0、1 的数位，将它变为 1，并将右面所有数位设为 1。将得到的 B 进制表示视为二进制进行询问即可。

预处理递推 f 的时间复杂度为$O(\log^2 n)$，共有$O(\log n)$次查询，因此总时间复杂度为$O(\log^2 n)$。

实际上，最终的代码并不涉及树的操作，我们只是利用图形的方式来方便思考。因此也可以只从数位的角度考虑：对于询问 n，我们找到一个等于 1 的数位，将它赋为 0，则它右面的数位可以任意取，我们需要统计其中恰好含有 K-tot 个 1 的数的个数（其中 tot 表

示这一位左边的 1 的个数），则可以利用组合数公式求解。逐位枚举所有”1”进行统计即可。我们发现，之前推出的 f 正是组合数。同样是采用“逐位确定”的方法，两种方法异曲同工。当你觉得单纯从数位的角度较难思考时，不妨画出图形以方便思考。

三、通用状态

实际上求解问题时不一定要把数都转化为二进制，以十进制为例，我们可以有如下的通用流程：

对于一个小于 n 的数，它从高到低位肯定会出现某一位，这位上的数值小于 n 所对应的这位上的数值。例如 n=58,49 在十位上小于 58，51 在个位上小于 58.

有了这个性质后，我们只需要从高到低位枚举我们所求解的数字第一次小于 n 的那一位，则此时，之前的位都已经确定了（它们都与 n 相等，不会大于或小于 n 对应位），那么之后的位数不受限制，即是类似[0,99..99]这样的区间进行统计，此时这其中的答案可以预处理，询问时直接统计即可。

这类问题的预处理状态通常为 $f[i][S]$，表示处理到第 i 位，而此时这 i 位的状态为 S 时数字的个数， S 根据具体题目要求来定。例如题目要求各个数位和等于某个数，则 S 就记录和；若要求数字模某个数字要为多少，则 S 就记录模后的值。

更一般的，我们可以采取不处理的方式，直接对 n 进行 DP，此时的状态需要加入一维，来记录我们当前所填出的数字与 n 的大小关系：$f[i][S][0/1]$ 表示填完了前 i 位，数字当前的状态为 S，前 i 位与 n 的前 i 位的大小关系为 0（小等于）或是 1（大于）的数字个数，转移时枚举下一位所填数字进行转移即可。要注意从低位开始填与从高位开始填的转移区别。

数位 DP 解决问题的核心思想就是“逐位确定”，利用枚举数位将求解问题划分好阶段，再利用子问题进行求解。同时要注意数字前导零对于问题的影响。

四、例题

windy 数 （BZOJ 1026）

题目大意：

windy 定义了一种 windy 数。不含前导零且相邻两个数字之差至少为 2 的正整数被称为 windy 数。 windy 想知道，在 A 和 B 之间，包括 A 和 B，总共有多少个 windy 数？

1 <= A <= B <= 2 * 10^9

分析：

数位 DP 模板题。状态中记录填数时上一位数字的大小即可。

代码：

```
#include <algorithm>
#include <iostream>
#include <cstring>
#include <cstdio>
#include <cmath>
using namespace std;

const int N = 11;
int A, B, f[N][N][2], c[N][N];

inline int Dp(int x) {
  static int a[N]; int n = 0;
  while (x) a[++n] = x % 10, x /= 10;
  if (n == 0) a[++n] = 0;
  memset(f, 0, sizeof(f));
  for (int i = 0; i <= 9; ++i)
    if (i <= a[1]) f[1][i][0] = 1;
    else  f[1][i][1] = 1;
  for (int i = 2; i <= n; ++i)
    for (int j = 0; j <= 9; ++j)
      for (int k = 0; k <= 9; ++k)
        if (c[j][k] >= 2) {
          if (j < a[i])
            f[i][j][0] += f[i - 1][k][0] + f[i - 1][k][1];
          else if (j == a[i])
            f[i][j][0] += f[i - 1][k][0], f[i][j][1] += f[i - 1][k][1];
          else  f[i][j][1] += f[i - 1][k][0] + f[i - 1][k][1];
        }
  int res = 0;
  for (int i = 1; i <= 9; ++i) {
    if (i < a[n]) res += f[n][i][0] + f[n][i][1];
    else if (i == a[n]) res += f[n][i][0];
  }
  for (int i = n - 1; i; --i)
    for (int j = 1; j <= 9; ++j)
      res += f[i][j][0] + f[i][j][1];
  return res;
}
```

```

inline void Init() {
  for (int i = 0; i <= 9; ++i)
    for (int j = i; j <= 9; ++j)
      c[i][j] = c[j][i] = j - i;
}

int main() {
  Init();
  scanf("%d%d", &A, &B);
  printf("%d\n", Dp(B) - Dp(A - 1));
  return 0;
}
```

插头 DP

由于 CDQ 论文已经完整地进行了研究，下面就直接贴出她的原论文：

基于连通性状态压缩的动态规划问题

长沙市雅礼中学 陈丹琦

【摘要】

基于状态压缩的动态规划问题是一类以集合信息为状态且状态总数为指数级的特殊的动态规划问题. 在状态压缩的基础上，有一类问题的状态中必须要记录若干个元素的连通情况，我们称这样的问题为基于连通性状态压缩的动态规划问题，本文着重对这类问题的解法及优化进行探讨和研究.

本文主要从动态规划的几个步骤——划分阶段，确立状态，状态转移以及程序实现来介绍这类问题的一般解法，会特别针对到目前为止信息学竞赛中涌现出来的几类题型的解法作一个探讨. 结合例题，本文还会介绍作者在减少状态总数和降低转移开销两个方面对这类问题优化的一些心得.

【关键词】

状态压缩　连通性　括号表示法　轮廓线　插头　棋盘模型

【目录】

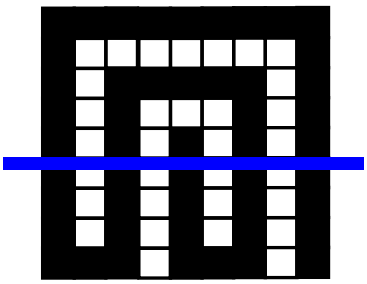
【序言】

先看一个非常经典的问题——旅行商问题(即 TSP 问题，Traveling Salesman Problem)：一个 n(≤15)个点的带权完全图，求权和最小的经过每个点恰好一次的封闭回路．这个问题已经被证明是 NP 完全问题，那么对于这样一类无多项式算法的问题，搜索算法是不是解决问题的唯一途径呢？答案是否定的．不难发现任何时候我们只需要知道哪些点已经被遍历过而遍历点的具体顺序对以后的决策是没有影响的，因此不妨以当前所在的位置 i，遍历过的点的集合 S 为状态作动态规划：

$$f(i,S)=\min\{f(j,S-\{i\})+dist(j,i)\}$$ ，其中 $j<>i$，i，j in S.

动态规划的时间复杂度为 $O(2^n*n^2)$，虽然为指数级算法，但是对于 n = 15 的数据规模来说已经比朴素的 $O(n!)$ 的搜索算法高效很多了．我们通常把这样一类以一个集合内的元素信息作为状态且状态总数为指数级别的动态规划称为**基于状态压缩的动态规划或集合动态规划**．基于状态压缩的动态规划问题通常具有以下两个特点：1．数据规模的某一维或几维非常小；2．它需要具备动态规划问题的两个基本性质：**最优性原理**和**无后效性**．

一般的状态压缩问题，压缩的是一个小范围内每个元素的决策，状态中元素的信息相对独立．而有些问题，仅仅记录每个元素的决策是不够的，不妨再看一个例子：给你一个 m * n (m, n≤9) 的矩阵，每个格子有一个价值 $V_{i,j}$，要求找一个连通块使得该连通块内所有格子的价值之和最大．按从上到下的顺序依次考虑每个格子选还是不选，下图为一个极端情况，其中黑色的格子为所选的连通块．只考虑前 5 行的时候，所有的黑色格子形成了三个连通块，而最后所有的黑色格子形成一个连通块．如果状态中只单纯地记录前一行或前几行的格子选还是不选，是无法准确描述这个状态的，因此压缩的状态中我们需要增加一维，记录若干个格子之间的连通情况．我们把这一类必须要在状态中记录若干个元素之间的连通信息的问题称为**基于连通性状态压缩的动态规划问题**．本文着重对这类问题进行研究．



连通是图论中一个非常重要的概念，在一个无向图中，如果两个顶点之间存在一条路径，则称这两个点连通．而基于连通性状态压缩的动态规划问题与图论模型有着密切的关联，比如后文涉及到的哈密尔顿回路、生成树等等．通常这类问题的本身与连通性有关或者隐藏着连通信息．

全文共有六个章节.

第一章，问题的一般解法，介绍解决基于连通性状态压缩的动态规划问题的一般思路和解题技巧；

第二章，一类简单路径问题，介绍一类基于棋盘模型的简单路径问题的状态表示的改进——括号表示法以及提出广义的括号表示法；

第三章，一类棋盘染色问题，介绍解决一类棋盘染色问题的一般思路；

第四章，一类基于非棋盘模型的问题，介绍解决一类非棋盘模型的连通性状态压缩问题的一般思路；

第五章，一类最优性问题的剪枝技巧，本章的重点是优化，探讨如何通过剪枝来减少扩展的状态的总数从而提高算法的效率；

第六章，总结，回顾前文，总结解题方法.

【正文】

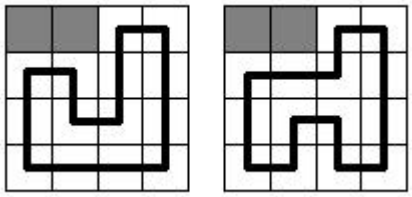
一. 问题的一般解法

基于连通性状态压缩的动态规划问题通常具有一个比较固定的模式，几乎所有的题目都是在这个模式的基础上变形和扩展的. 本章选取了一个有代表性的例题来介绍这一类问题的一般解法.

【例 1】Formula 1[1]

问题描述

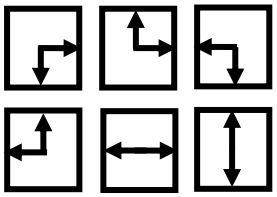
给你一个 *m* * *n* 的棋盘，有的格子是障碍，问共有多少条回路使得经过每个非障碍格子恰好一次. *m*, *n* ≤ 12.



如图，*m* = *n* = 4，(1, 1), (1, 2)是障碍，共有 2 条满足要求的回路.

算法分析

【划分阶段】 这是一个典型的基于**棋盘模型**的问题，棋盘模型的特殊结构，使得它成为连通性状态压缩动态规划问题最常见的“舞台”. 通常来说，棋盘模型有三种划分阶段的方法：逐行，逐列，逐格. 顾名思义，逐行即从上到下或从下到上依次考虑每一行的状态，并转移到下一行；逐列即从左到右或从右到左依次考虑每一列的状态，并转移到下一列；逐格即按一定的顺序(如从上到下，从左到右)依次考虑每一格的状态，并转移到下一个格子.



对于本题来说，逐行递推和逐列递推基本类似[2]，接下来我们会对逐行递推和逐格递推的状态确立，状态转移以及程序实现一一介绍.

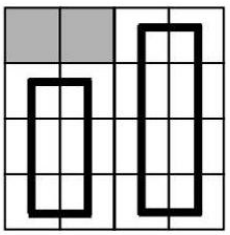
---

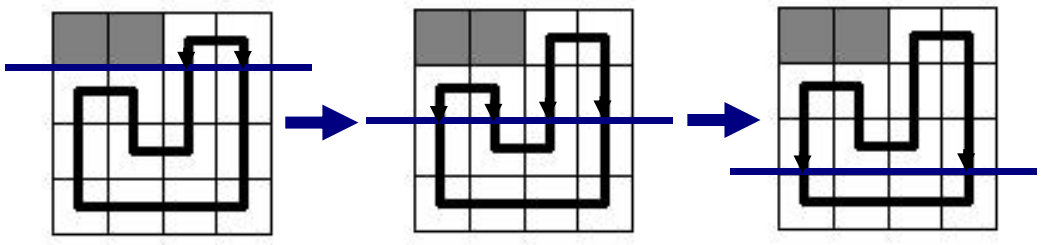
[1] Ural1519, Timus Top Coders : Third Challenge

[2] 有的题目，逐行递推和逐列递推的状态表示有较大的区别，比如本文后面会讲到的 Rocket Mania 一题

【确立状态】 先提出一个非常重要的概念——“插头”．对于一个 4 连通的问题来说，它通常有上下左右 4 个插头，一个方向的插头存在表示这个格子在这个方向可以与外面相连．本题要求回路的个数，观察可以发现所有的非障碍格子一定是从一个格子进来，另一个格子出去，即 4 个插头恰好有 2 个插头存在，共 6 种情况．

逐行递推 不妨按照从上到下的顺序依次考虑每一行．分析第 *i* 行的哪些信息对第 *i* + 1 行有影响：我们需要记录第 *i* 行的每个格子是否有下插头，这决定了第 *i*+1 行的每个格子是否有上插头．仅仅记录插头是否存在是不够的，可能导致出现多个回路 (如右图)，而本题要求一个回路，也就隐含着最后所有的非障碍格子通过插头连接成了一个**连通块**，因此还需要记录第 *i* 行的 *n* 个格子的**连通情况**.





插头：0011　　插头：1111　　插头：1001[3]

连通性：(3,4)　　连通性：(1,2) (3,4)　　连通性：(1,2,3,4)[4]

我们称图中的蓝线为**轮廓线**，任何时候只有轮廓线上方与其直接相连的格子和插头才会对轮廓线以下的格子产生直接的影响．通过上面的分析，可以写出动态规划的状态：$f(i,S_0,S_1)$ 表示前 *i* 行，第 *i* 行的 *n* 个格子是否具有下插头的一个 *n* 位的二进制数为 S_0，第 *i* 行的 *n* 个格子之间的连通性为 S_1 的方案总数．

如何表示 *n* 个格子的连通性呢? 通常给每一个格子标记一个正数，属于同一个的连通块的格子标记相同的数．比如{1,1,2,2}和{2,2,1,1}都表示第 1,2 个格子属于一个连通块，第 3,4 个格子属于一个连通块．为了避免出现同一个连通信息有不同的表示，一般会使用**最小表示法**.

一种最小表示法为：所有的障碍格子标记为 0，第一个非障碍格子以及与它连通的所有格子标记为 1，然后再找第一个未标记的非障碍格子以及与它连通的格子标记为 2，……，重复这个过程，直到所有的格子都标记完毕．比如连通信息((1,2,5),(3,6),(4))表示为{1,1,2,3,1,2}．还有一种最小表示法，即一个连通块内所有的格子都标记成该连通块最左边格子的列编号，比如上面这个例子，我们表示

---

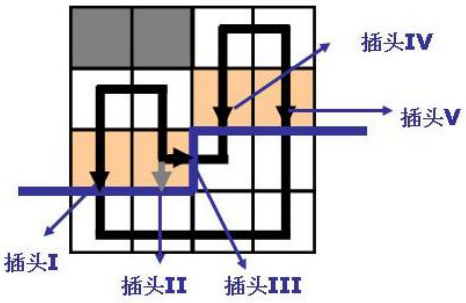
[3] 从左到右, 0 表示无插头, 1 表示有插头

[4] 括号内的数表示的是格子的列编号, 一个括号内的格子属于一个连通块

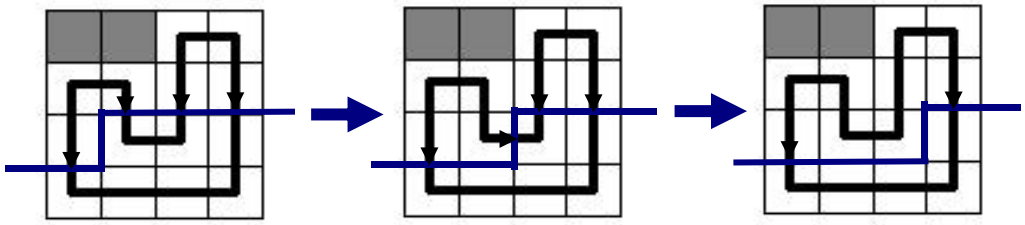
为{1,1,3,4,1,3}．两种表示方法在转移的时候略有不同，本文后面将会提到[5]．如上图三个状态我们可以依次表示为 $f(1,(0011)_2,\{0,0,1,1\})$，$f(2,(1111)_2,\{1,1,2,2\})$，$f(3,(1001)_2,\{1,1,1,1\})$．

状态表示的优化 通过观察可以发现如果轮廓线上方的 *n* 个格子中某个格子没有下插头，那么它就不会再与轮廓线以下的格子直接相连，它的连通性对轮廓线以下的格子不会再有影响，也就成为了“冗余”信息．不妨将记录格子的连通性改成记录插头的连通性，如果这个插头存在，那么就标记这个插头对应的格子的连通标号，如果这个插头不存在，那么标记为 0．这样状态就从 $f(i,S_0,S_1)$ 精简为 $f(i,S)$，上图三个状态表示为 $f(1,\{0,0,1,1\})$，$f(2,\{1,1,2,2\})$，$f(3,\{1,0,0,1\})$．优化后不仅状态表示更加简单，而且状态总数将会大大减少．

逐格递推 按照从上到下，从左到右的顺序依次考虑每一格．分析转移完(*i*, *j*)这个格子后哪些信息对后面的决策有影响：同样我们可以刻画出轮廓线，即轮廓线上方是已决策格子，下方是未决策格子．由图可知与轮廓线直接相连的格子有 *n* 个，直接相连的插头有 *n*+1 个，包括 *n* 个格子的下插头以及(*i*, *j*)的右插头．为了保持轮廓线的“连贯性”，不妨从左到右依次给 *n* 个格子标号，*n*+1 个插头标号．类似地，我们需要记录与轮廓线直接相连的 *n*+1 个插头是否存在以及 *n* 个格子的连通情况．



通过上面的分析，很容易写出动态规划的状态：$f(i,j,S_0,S_1)$ 表示当前转移完(*i*, *j*)这个格子，*n*+1 个插头是否存在表示成一个 *n*+1 位的二进制数 S_0，以及 *n* 个格子的连通性为 S_1 的方案总数．



$f(3,1,(10111)_2,\{1,1,2,2\})$　$f(3,2,(10111)_2,\{1,1,2,2\})$　$f(3,3,(10001)_2,\{1,1,1,1\})$

逐行递推的时候我们提到了状态的优化，同样地，我们也可以把格子的连通性记录在插头上，新的状态为 $f(i,j,S)$，上图 3 个状态依次为 $f(3,1,\{1,0,1,2,2\})$，$f(3,2,\{1,0,1,2,2\})$，$f(3,3,\{1,0,0,0,1\})$．

[5]因为第一种表示法更加直观，本文如果不作特殊说明，默认使用第一种最小表示法

【转移状态】

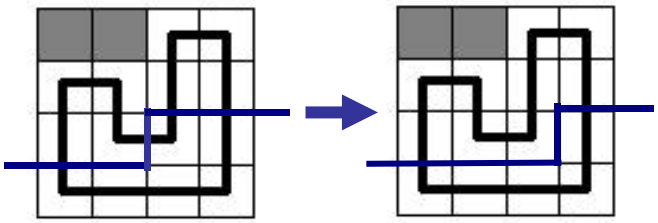
状态的转移开销主要包含两个方面：每个状态转移的状态数，计算新的状态的时间.

逐行递推　假设从第 *i* 行转移到第 *i*+1 行，我们需要枚举第 *i*+1 行的每个格子的状态(共 6 种情况)，对于任何一个非障碍格子，它是否有上插头和左插头已知，因此最多只有 2 种情况，状态的转移数≤2^n.

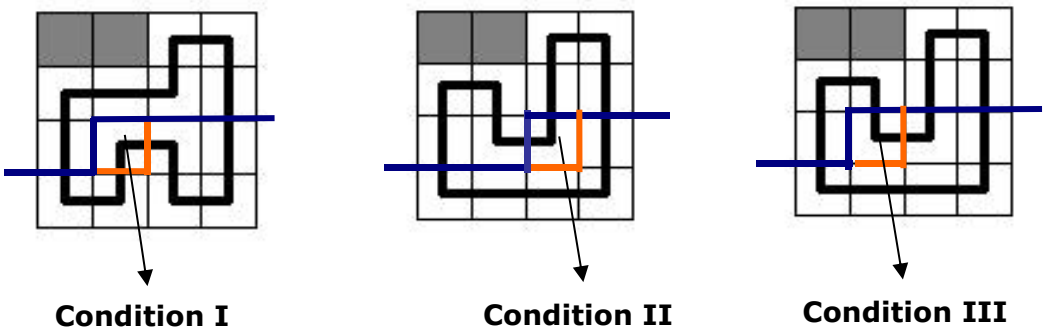
枚举完第 *i*+1 行每个格子的状态后，需要计算第 *i*+1 行 *n* 个格子之间的连通性的最小表示，通常可以使用并查集的 Father 数组对其重新标号或者重新执行一次 BFS/DFS，时间复杂度为 *O*(*n*)，最后将格子的连通性转移到插头的连通性上.

特别需要注意的是在转移的过程中，为了避免出现多个连通块，除了最后一行，任何时候一个连通分量内至少有一个格子有下插头.

逐格递推　仔细观察下面这个图，当 $f(i,j-1,S)$ 转移到 $f(i,j,S')$ 时，轮廓



线上 *n* 个格子只有(*i*-1, j)被改成(*i*, *j*)，*n*+1 个插头只有 2 个插头被改动，即(*i*, *j*-1)的右插头修改成(*i*, *j*)的下插头和(*i*-1，*j*)的下插头修改成(*i*, *j*)的右插头. 转移的时候枚举(*i*, *j*)的状态分情况讨论. 一般棋盘模型的逐格递推转移有 3 类情况：新建一个连通分量，合并两个连通分量，以及保持原来的连通分量. 下面针对本题进行分析：



情况 1　新建一个连通分量，这种情况出现在(*i*, *j*)有右插头和下插头. 新建的两个插头连通且不与其它插头连通，这种情况下需要将这两个插头连通分量标号标记成一个未标记过的正数，重新 *O*(*n*)扫描保证新的状态满足最小表示.

情况 2　合并两个连通分量，这种情况出现在(*i*, *j*)有上插头和左插头．如果两个插头不连通，那么将两个插头所处的连通分量合并，标记相同的连通块标号，*O*(*n*)扫描保证最小表示；如果已经连通，相当于出现了一个回路，这种情况只能出现在最后一个非障碍格子．

情况 3　保持原来的连通分量，这种情况出现在(*i*, *j*)的上插头和左插头恰好有一个，下插头和右插头也恰好有一个．下插头或右插头相当于是左插头或上插头的延续，连通块标号相同，并且不会影响到其他的插头的连通块标号，计算新的状态的时间为 *O*(1)．

注意当从一行的最后一个格子转移到下一行的第一个格子的时候，轮廓线需要特殊处理．值得一提的是，上面三种情况计算新的状态的时间分别为 *O*(*n*), *O*(*n*), *O*(1)，如果使用前面提到的第二种最小表示方法，情况 1 只需要 *O*(1)，但是情况 3 可能需要 *O*(*n*)重新扫描．

比较一下逐行递推和逐格递推的状态的转移，逐行递推的每一个转移的状态总数为指数级，而逐格递推为 *O*(1)，每次计算新的状态的时间两者最坏情况都为 *O*(*n*)，但是逐行递推的常数要比逐格递推大，从转移开销这个角度来看，逐格递推的优势是毋庸置疑的．

【程序实现】

逐行递推和逐格递推的程序实现基本一致，下面以逐格递推为例来说明．首先必须解决的一个问题是，对于像 $f(3,2,\{1,0,1,2,2\})$ 这样的一个状态我们该如何存储，可以开一个长度为 *n*+1 的数组来存取 *n*+1 个插头的连通性，但是数组判重并不方便，而且空间较大．不妨将 *n*+1 个元素进行**编码**，用一个或几个整数来存储，当我们需要取一个状态出来对它进行修改的时候再进行**解码**．

编码最简单的方法就是表示成一个 *n*+1 位的 *p* 进制数，*p* 可以取能够达到的最大的连通块标号加 1[6]，对本题来说，最多出现 $\lfloor n/2 \rfloor \le 6$ 个连通块，不妨取 *p* = 7．在不会超过数据类型的范围的前提下，建议将 *p* 改成 2 的幂，因为位运算比普通的运算要快很多，本题最好采用 8 进制来存储．

如需大范围修改连通块标号，最好将状态 *O*(*n*) 解码到一个数组中，修改后再 *O*(*n*)计算出新的 *p* 进制数，而对于只需要局部修改几个标号的情况下，可以直接用(*x* div p^{i-1}) mod *p* 来获取第 *i* 位的状态，用 $\pm k * p^{i-1}$ 直接对第 *i* 位进行修改．

最后我们探讨一下实现的方法，一般有两种方法：

1．对所有可能出现的状态进行编码，枚举编码方式：预处理将所有可能的连通性状态搜索出来，依次编号 1, 2, 3, …,*Tot*，那么状态为 $f(i,j,k)$ 表示转移完

[6] 因为还要把 0 留出来存没有插头的情况

(*i*, *j*)后轮廓线状态编号为 *k* 的方案总数．将所有状态存入 Hash 表中，使得每个状态与编号一一对应，程序框架如下：

```
For  i ← 1  to  m
  For  j ←1  to  n
    For  k ← 1  to  Tot
      For  x ← (i, j, State[k])  的所有转移后的状态
        k' ←  状态 x 的编号
        f(i', j', k') ← f(i', j', k') + f(i, j, k) ，  (i', j') 为 (i, j) 的后继格子.
      End For
```

2．记忆化宽度优先搜索：将初始状态放入队列中，每次取队首元素进行扩展，并用 Hash 对扩展出来的新的状态判重．程序框架如下：

```
Queue.Push(所有初始状态)
While not Empty(Queue)
    p ← Queue.Pop()
    For  x ← p 的所有转移后的状态
      If  x 之前扩展过  Then
          Sum [x] ← Sum[x] + Sum[p]
      Else
          Queue.Push(x)
          Sum[x] ← Sum[p]
      End If
    End For
End While
```

比较上述两种实现方法，直接编码的方法实现简单，结构清晰，但是有一个很大的缺点：无效状态可能很多，导致了很多次空循环，而大大影响了程序的效率．下面是一组实验的比较数据：

表 1．直接编码与宽度优先搜索扩展状态总数比较

测试数据	宽度优先搜索 扩展状态总数	直接编码 *Tot*	*Tot* * *m* * *n*	无效状态比率
m = 9, *n* = 9 (1,1)为障碍	30930	2188	177228	82.5%
m = 10, *n* = 10 无障碍	134011	5798	579800	76.8%
m = 11, *n* = 11 (1,1)为障碍	333264	15511	1876831	82.2%
m = 12, *n* = 12 无障碍	1333113	41835	6024240	77.9%

可以看出直接编码扩展的无效状态的比率非常高，对于障碍较多的棋盘其对比更加明显，因此通常来说宽度优先搜索扩展比直接编码实现效率要高.

Hash 判重的优化：使用一个 HashSize 较小的 Hash 表，每转移一个(i, j)清空一次，每次判断状态 x 是否扩展过的程序效率比用一个 HashSize 较大的 Hash 表每次判断状态(i, j, x)高很多．类似地，在不需要记录路径的情况下，也可以使用滚动的扩展队列来代替一个大的扩展队列.

最后我们比较一下，不同的实现方法对程序效率的影响[7]：

Program 1 ： 8-Based，枚举编码方式.

Program 2 ： 8-Based，队列扩展，HashSize = 3999997.

Program 3 ： 8-Based，队列扩展，HashSize = 4001，Hash 表每次清空.

Program 4 ： 7-Based，队列扩展，HashSize = 4001，Hash 表每次清空.

表 2．不同的实现方法的程序效率的比较

测试数据	Program 1	Program 2	Program 3	Program 4
m = 10, n = 10 无障碍棋盘	46ms	31ms	15ms	31ms
m = 11, n = 11 (1,1)为障碍	140ms	499ms	109ms	187ms
m = 12, n = 12 无障碍	624ms	1840ms	499ms	873ms

小结

本章从划分阶段，确立状态，状态转移以及程序实现四个方面介绍了基于连通性状态压缩动态规划问题的一般解法，并在每个方面归纳了一些不同的方法，最后对不同的算法的效率进行比较. 在平时的解题过程中我们要学会针对题目的特点和数据规模“对症下药”，选择最合适的方法而达到最好的效果.

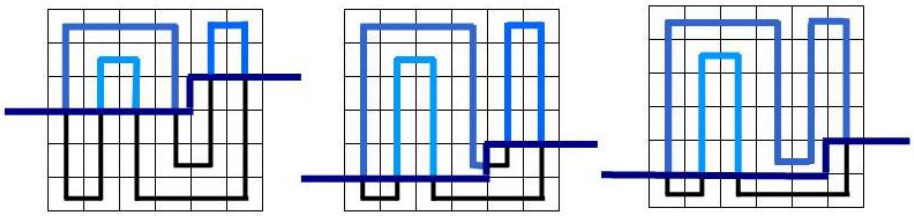
由于逐格递推的转移开销比逐行递推小很多，下文如果不作特殊说明，我们都采用逐格的阶段划分.

[7] 测试环境: Intel Core2 Duo T7100, 1.8GHz, 1G 内存

二. 一类简单路径问题

这一章我们会针对一类基于棋盘模型的简单回路和简单路径问题的解法作一个探讨. 简单路径，即除了起点和终点可能相同外，其余顶点均不相同的路径，而简单回路为起点和终点相同的简单路径. Formula 1 是一个典型的棋盘模型的简单回路问题，这一章我们继续以这个题为例来说明.

首先我们分析一下简单回路问题有什么特点：



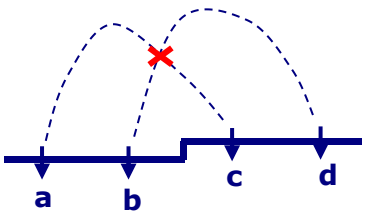
仔细观察上面的图，可以发现轮廓线上方是由若干条互不相交的路径构成的，而每条路径的两个端口恰好对应了轮廓线上的两个**插头**! 一条路径上的所有格子对应的是一个连通块，而每条路径的两个端口对应的两个插头是连通的而且不与其他任何一个插头连通.

在上一章我们提到了逐格递推转移的时候的三种情况：新建一个连通分量，合并两个连通分量，保持原来的连通分量，它们分别等价于两个插头成为了一条新的路径的两端，两条路径的两个端口连接起来形成一条更长的路径或一条路径的两个端口连接起来形成一个回路以及延长原来的路径.

通过上面的分析我们知道了简单回路问题一定满足任何时候轮廓线上每一个连通分量恰好有 2 个插头，那么这些插头之间有什么性质呢?

【性质】轮廓线上从左到右 4 个插头 a, b, c, d，如果 a, c 连通，并且与 b 不连通，那么 b, d 一定不连通.

证明：反证法，如果 a, c 连通，b, d 连通，那么轮廓线上方一定至少存在一条 a 到 c 的路径和一条 b 到 d 的路径. 如图，两条路径一定会有交点，不妨设两条路径相交于格子 P，那么 P 既与 a, c 连通，又与 b, d 连通，可以推出 a, c 与 b, d 连通，矛盾，得证.



这个性质对所有的棋盘模型的问题都适用.

“两两匹配”，“不会交叉”这样的性质，我们很容易联想到**括号匹配**. 将轮廓线上每一个连通分量中左边那个插头标记为左括号，右边那个插头标记为右括号，由于插头之间不会交叉，那么左括号一定可以与右括号一一对应. 这样我

们就可以使用 3 进制——0 表示无插头，1 表示左括号插头，2 表示右括号插头记录下所有的轮廓线信息．不妨用#表示无插头，那么上面的三幅图分别对应的是(())#()，(()#)()，(()###)，即 $(1122012)_3,(1120212)_3,(1120002)_3$，我们称这种状态的表示方法为**括号表示法**.

依然分三类情况来讨论状态的转移：

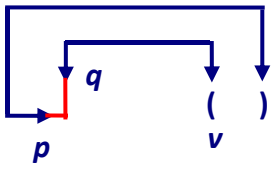
为了叙述方便，不妨称(*i*,*j*-1)的右插头为 *p*，(*i*-1, *j*)的下插头为 *q*，(*i*, *j*)的下插头为 p'，右插头为 q'，那么每次转移相当于轮廓线上插头 *p* 的信息修改成 p' 的信息，插头 *q* 的信息修改成 q' 的信息，设 *W*(*x*) = 0, 1, 2 表示插头 *x* 的状态.

情况 1　新建一个连通分量，这种情况下 *W*(*p*) = 0，*W*(*q*) = 0，p'，q' 两个插头构建了一条新的路径，相当于 p' 为左括号，q' 为右括号，即 $W(p')\leftarrow 1$，$W(q')\leftarrow 2$，计算新的状态的时间为 *O*(1).

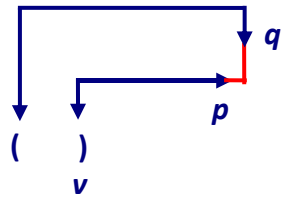
情况 2　合并两个连通分量，这种情况下 *W*(*p*) > 0，*W*(*q*) > 0，$W(p')\leftarrow 0$，$W(q')\leftarrow 0$，根据 *p*, *q* 为左括号还是右括号分四类情况讨论：

情况 2.1　*W*(*p*) = 1，*W*(*q*) = 1．那么需要将 *q* 这个左括号与之对应的右括号 *v* 修改成左括号，即 *W*(*v*) ← 1.

情况 2.2　*W*(*p*) = 2，*W*(*q*) = 2．那么需要将 *p* 这个右括号与之对应的左括号 *v* 修改成右括号，即 *W*(*v*)← 2.



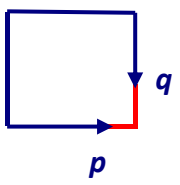
情况 2.1 图



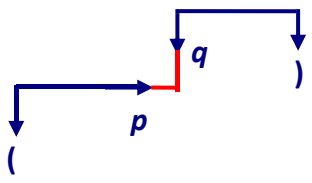
情况 2.2 图

情况 2.3　*W*(*p*) = 1，*W*(*q*) = 2，那么 *p* 和 *q* 是相对应的左括号和右括号，连接 *p*, *q* 相当于将一条路径的两端连接起来形成一个回路，这种情况下只能出现在最后一个非障碍格子.

情况 2.4　*W*(*p*) = 2，*W*(*q*) = 1，那么 *p* 和 *q* 连接起来后，*p* 对应的左括号和 *q* 对应的右括号恰好匹配，不需要修改其他的插头的状态.



情况 2.3 图



情况 2.4 图

情况 2.1, 2.2 需要计算某个左括号或右括号与之匹配的括号，这个时候需要对三进制状态解码，利用类似模拟栈的方法．因此情况 2.1, 2.2 计算新的状态的时间复杂度为 $O(n)$，2.3, 2.4 时间复杂度为 $O(1)$．

情况 3　保持原来的连通分量，$W(p)$，$W(q)$中恰好一个为 0，$W(p')$，$W(q')$中也恰好一个为 0．那么无论 p'，q' 中哪个插头存在，都相当于是 p, q 中那个存在的插头的延续，括号性质一样，因此 $W(p') \leftarrow W(p) + W(q)$，$W(q') \leftarrow 0$ 或者 $W(q') \leftarrow W(p) + W(q)$，$W(p') \leftarrow 0$．计算新的状态的时间复杂度为 $O(1)$．

通过上面的分析可以看出，括号表示法利用了简单回路问题的“一个连通分量内只有 2 个插头”的特殊性质巧妙地用 3 进制状态存储下完整的连通信息，插头的连通性标号相对独立，不再需要通过 $O(n)$扫描大范围修改连通性标号．实现的时候，我们可以用 4 进制代替 3 进制而提高程序运算效率，下面对最小表示法与括号表示法的程序效率进行比较：

表 3．不同的状态表示的程序效率的比较

测试数据	最小表示法 7Based	最小表示法 8Based	括号表示法 3Based	括号表示法 4Based
$m = 10, n = 10$ 无障碍棋盘	31ms	15ms	0ms	0ms
$m = 11, n = 11$ (1,1)为障碍	187ms	109ms	46ms	31ms
$m = 12, n = 12$ 无障碍	873ms	499ms	265ms	140ms

可以看出，括号表示法的优势非常明显，加上它的思路清晰自然，实现也更加简单，因此对于解决这样一类简单回路问题是非常有价值的．

类似的问题还有：NWERC 2004 Pipes，Hnoi2004 Postman，Hnoi2007 Park，还有一类非回路问题也可以通过棋盘改造后用简单回路问题的方法解决，比如 POJ 1739 Tony’s Tour：给一个 $m * n$ 棋盘，有的格子是障碍，要求从左下角走到右下角，每个格子恰好经过一次，问方案总数．(m, n ≤ 8)

只需要将棋盘改造一下，问题就等价于 Formula 1 了．

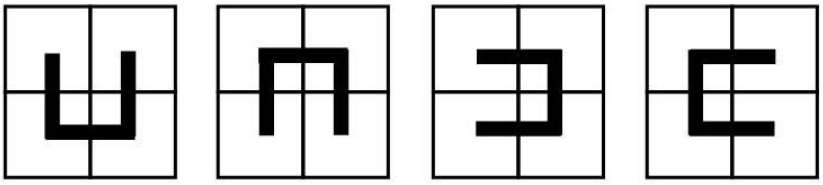
```
                .......
#..   改造成    .#####.
...             .#..#.
                .....
```

介绍完简单回路问题的解法，那么一般的简单路径问题又如何解决呢?

【例 2】Formula 2[8]

问题描述

给你一个 *m* * *n* 的棋盘，有的格子是障碍，要求从一个非障碍格子出发经过每个非障碍格子恰好一次，问方案总数．*m*, *n* ≤ 10．

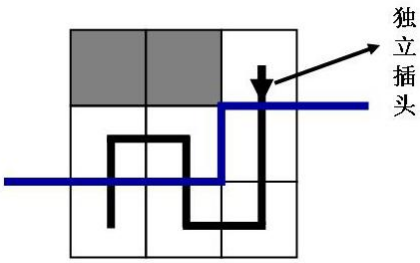


如图，一个 2 * 2 的无障碍棋盘，共有 4 条满足要求的路径．

算法分析

确立状态：按照从上到下，从左到右依次考虑每一个格子，设 $f(i,j,S)$ 表示转移完(*i*, *j*)这个格子，轮廓线状态为 *S* 的方案总数．如果用一般的最小表示法，不仅需要记录每个插头的连通情况，还需要额外记录每个插头是否连接了路径的一端，状态表示相当复杂．依然从括号表示法这个角度来思考如何来存储轮廓线的状态：

这个问题跟简单回路问题最大的区别为：不是所有的插头都两两匹配，有的插头连接的路径的另一端不是一个插头而是整条路径的一端，我们称这样的插头为**独立插头**．不妨将原来的 3 进制状态修改成 4 进制——0 表示无插头，1 表示左括号插头，2 表示右括号插头，3 表示独立插头，这样我们就可以用 4 进制完整地记录下轮廓线的信息，图中状态表示为$(1203)_4$．



状态转移：依然设(*i*, *j*-1)的右插头为 *p*，(*i*-1, *j*)的下插头为 *q*，(*i*, *j*)的下插头为 p'，右插头为 q'．部分转移同简单回路问题完全一样，这里不再赘述，下面分三类情况讨论与独立插头有关的转移：

情况 1　*W*(*p*) = 0，*W*(*q*) = 0．当前格子可能成为路径的一端，即右插头或下插头是独立插头，因此 $W(p')\leftarrow 3$，$W(q')\leftarrow 0$ 或者 $W(q')\leftarrow 3$，$W(p')\leftarrow 0$．

情况 2　*W*(*p*) > 0，*W*(*q*) > 0，那么 $W(p')\leftarrow 0$，$W(q')\leftarrow 0$

情况 2.1　*W*(*p*) =3，*W*(*q*) = 3，将插头 *p* 和 *q* 连接起来就相当于形成了

[8] 改编自 Formula 1

一条完整的路径，这种情况只能出现在最后一个非障碍格子.

情况 2.2 $W(p)$ ，$W(q)$ 中有一个为 3，如果 p 为独立插头，那么无论 q 是左括号插头还是右括号插头，与 q 相匹配的插头 v 成为了独立插头，因此，$W(v) \leftarrow 3$. 如果 q 为独立插头，类似处理.

情况 3 $W(p)$ ，$W(q)$ 中有一个>0，即 p, q 中有一个插头存在.

情况 3.1 如果这个插头为独立插头，若在最后一个非障碍格子，这个插头可以成为路径的一端，否则可以用右插头或下插头来延续这个独立插头.

情况 3.2 如果这个插头是左括号或右括号，那么我们以将这个插头“封住”，使它成为路径的一端，需要将这个插头所匹配的另一个插头的状态修改成为独立插头.

情况 2.2, 3.2 需要计算某个左括号或右括号与之匹配的括号，计算新的状态的时间复杂度为 $O(n)$，其余情况计算新的状态的时间复杂度为 $O(1)$.

特别需要注意，任何时候轮廓线上独立插头的个数不可以超过 2 个. 至此问题完整解决，$m = n = 10$ 的无障碍棋盘，扩展的状态总数为 3493315，完全可以承受.

上面两类题目我们用括号表示法取得了很不错的效果，但是它存在一定的局限性，即插头必须满足两两匹配. 那么对于更加一般的问题，一个连通分量内出现大于 2 个插头，上述的括号表示方法显得束手无策. 下面将介绍一种括号表示法的变形，它可以适用于出现连通块内大于 2 个插头的问题，我们称之为**广义的括号表示法：**

假设一个连通分量从左到右有多个插头，不妨将最左边的插头标记为“**(**”，最右边的插头标记为“**)**”，中间的插头全部标记为“**)(**”，那么能够匹配的括号对应的插头连通. 如果问题中可能出现一个连通分量只有一个插头，那么这个插头标记为“**( )**”，这样插头之间的连通性可用括号序列完整地记录下来，比如对于一个连通性状态为{1,2,2,3,4,3,2,1}，我们可以用**(-(-)(-(-()-)-)-)**记录.

这种广义的括号表示方法需要用 4 进制甚至 5 进制存储状态，而且直接对状态连通性进行修改情况非常多，最好还是将状态进行解码，修改后再重新编码. 下文我们将会运用广义的括号表示法解决一些具体的问题.

小结

本章针对一类简单路径问题，提出了一种新的状态表示方法——括号表示法，最后提出了广义的括号表示方法. 相比普通的最小表示法，括号表示法巧妙地把连通块与括号匹配一一对应，使得状态更加简单明了，虽然不会减少扩展的状态总数，但是转移开销的常数要小很多，是一个不错的方法.

三. 一类棋盘染色问题

有一类这样的问题——给你一个 *m* * *n* 的棋盘，要求给每个格子染上一种颜色(共 *k* 种颜色)，每种颜色的格子相互连通 (4 连通)．本章主要对这类问题的解法进行探讨，我们从一个例题说起：

【例 3】Black & White[9]

问题描述

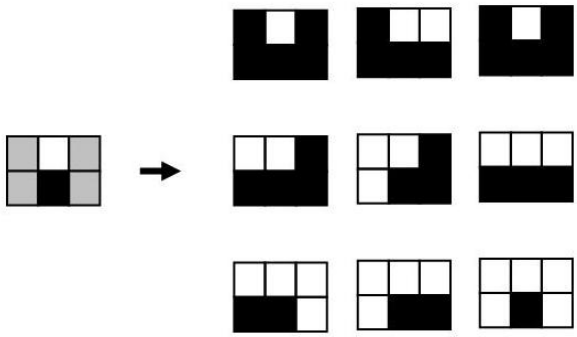
一个 *m* * *n* 的棋盘，有的格子已经染上黑色或白色，现在要求将所有的未染色格子染上黑色或白色，使得满足以下 2 个限制：

1) 所有的黑色的格子是连通的，所有的白色格子也是连通的．

2) 不会有一个 2 * 2 的子矩阵的 4 个格子的颜色全部相同．

问方案总数．($m, n \le 8$)

如下图，$m = 2$，$n = 3$，灰色格子为未染色格子，共有 9 种染色方案．



算法分析

这是一个典型的棋盘染色问题，着色规则有：

1) 只有黑白两种颜色，即 $k = 2$，并且同色的格子互相连通．

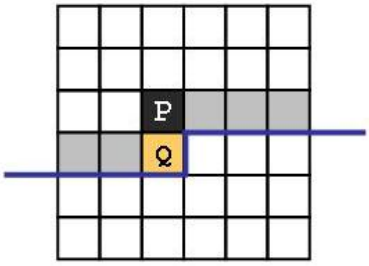
2) 没有同色的 2 * 2 的格子．

对于简单路径问题来说，相邻的格子是否连通取决于它们之间的插头是否存在，状态记录轮廓线上每个插头是否存在以及插头之间的连通性；而棋盘染色问题相邻的格子是否连通取决于它们的颜色是否相同，这就需要记录轮廓线上方 *n* 个格子的颜色以及格子之间的连通性．

确立状态　设当前转移完 $Q(i, j)$这个格子，对以后的决策产生影响的信息有：

[9] Source : Uva10572

轮廓线上方 n 个格子的染色情况以及它们的连通性，由第 2 条着色规则“没有同色的 2 * 2 的格子”可知 $P(i-1, j)$的颜色会影响到$(i, j+1)$着色，因此我们还需要额外记录格子 P 的颜色．动态规划的状态为：$f(i, j, S_0, S_1, cp)$ 表示转移完(i, j)，轮廓线上从左到右 n 个格子的染色情况为 S_0 $(0 \le S_0 < 2^n)$，连通性状态为S_1，格子 P 的颜色为 cp(0 或 1)的方案总数．



状态的精简　如果相邻的 2 个格子不属于同一个连通块，那么它们必然不同色，因此只需要记录$(i, 1)$和$(i-1, j+1)$两个格子的颜色，利用 S_1 就可以推出 n 个格子的颜色．这个精简不会减少状态的总数，仍然需要一个变量来记录两个格子的颜色，因此意义并不大，这里只是提一下．

状态转移　枚举当前格子(i, j)的颜色，计算新的状态：S_0 和 cp 都很容易 $O(1)$ 计算出来．考虑计算 S_1：轮廓线的变化相当于将记录$(i-1, j)$的连通性改成记录(i, j)的连通性．根据当前格子与上面的格子和左边的格子是否同色分四类情况讨论．应当注意的是如果(i, j)和$(i-1, j)$不同色，并且$(i-1, j)$在轮廓线上为一个单独的一个连通块，那么$(i-1, j)$以后都不可能与其他格子连通，即剩余的格子都必须染上与$(i-1, j)$相反的颜色，需要特殊判断．转移的时间复杂度为 $O(n)$．计算新状态的 S_1 程序框架如下：

```
将前一个状态的 S1 解码，连通性存入 c[1],c[2],…,c[n].
If  (i, j) 与 (i-1, j) 不同色并且 (i-1, j) 为一个单独的连通块 Then
    特殊判断
Else
    If  (i, j) 与 (i-1, j) 和 (i, j-1) 均同色 Then
        For k ← 1 to n
            If c[k] = c[j] Then
            c[k] ← c[j-1]   // 合并两个连通块
            EndIf
    Else
        If  (i, j) 与 (i-1, j) 和 (i, j-1) 均不同色 Then
            c[j] ← 最大可能出现的连通块标号 // (i, j) 新建一个连通块.
        Else
            If  (i, j) 与 (i, j-1) 同色与 (i-1, j) 不同色 Then
                c[j] ← c[j-1]   // (i, j) 的连通性标号跟 (i, j-1)相同.
        EndIf
    EndIf
EndIf
对 c[] O(n)扫描，修改成最小表示，利用 c[]编码计算出新的 S1.
```

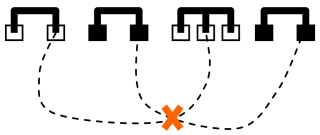
对于 $m = n = 8$ 的一个全部未染色的棋盘，扩展出来的状态总数为 122395，

转移需要时间为 $O(n)$，因此总的时间复杂度为 $O(TotalState * n) = 979160$，运行时间<0.1*s*．至此问题完整解决．类似可以解决的问题还有 2007 年重庆市选拔赛 Rect 和 IPSC 2007 Delicious Cake．

扩展 上面提到的是 4 连通问题，如果要求 8 连通呢?

4 连通问题是两个格子至少有一条边重合为连通，而 8 连通问题是两个格子至少有一个顶点重合为连通，因此需要记录所有至少有一个顶点在轮廓线上的格子的连通和染色情况，即包括$(i-1, j)$在内的 $n+1$ 个格子．

一个优化的方向 扩展的状态中无效状态的总数很大程度上决定了算法的效率．比如 Black & White 中如果出现右图的状态，那么无论之后如何决策，都不可能满足同色的格子互相连通的性质，因此它是一个无效状态．对于任何一个 k 染色棋盘问题，如果从左到右有 4 个相互不嵌套[10]的连通块 a,b,c,d，a, c 同色，b, d 同色且与 a, c 不同色，那么这个状态为无效状态．



小结

本章介绍了解决一类棋盘染色问题的一般思路．无论染色规则多么复杂，我们只要在基本状态即“轮廓线上方与其相连的格子的连通性以及染色情况”的基础上，根据题目的需要在状态中增加对以后的决策可能产生影响的信息，问题都可以迎刃而解了．

四．一类基于非棋盘模型的问题

本章将会介绍一类基于非棋盘模型的连通性状态压缩动态规划问题，它虽然不具有棋盘模型的特殊结构，但是解法的核心思想又跟棋盘模型的问题有着异曲同工之处．

[10] “嵌套”的概念可以用广义的括号匹配的表示方法来理解

【例 4】生成树计数[11]

问题描述

给你一个 n 个点的无向连通图，其边集为：任何两个不同的点 $i, j(1 \le i, j \le n)$，如果 $|i - j| \le k$，那么有一条无向边 $<i, j>$．已知 n 和 k，求这个图的生成树个数．

$n \le 10^{15}$，$2 \le k \le 5$．

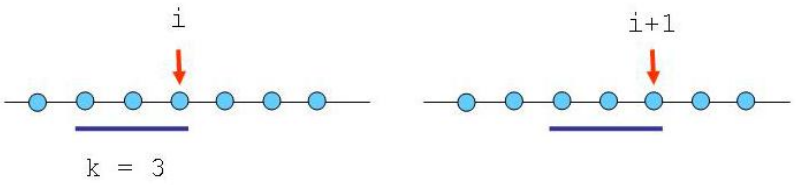
算法分析

这个题给我们的第一印象是：n 非常大，k 却非常小．

生成树最重要的两个性质：无环，连通．那么如果按照 1,2,…,n 的顺序依次考虑每一个点与前面的哪些点相连，并且保证任何时候都不会出现环，最后统计所有的点全部在一个连通分量内的方案总数即为最终的答案．

在棋盘模型的问题中，我们提出了轮廓线这个概念，任何时候只有轮廓线上方与其直接相连的格子对以后的决策会产生影响．类似地我们分析一下这个问题，当我们确定了 1~i 的所有点的连边情况后，哪些信息对以后的决策会产生影响：1~i–k 这些点与 i 之后的点一定没有边相连，那么对 i 以后的点的决策不会产生直接的影响，因此我们需要记录的仅仅是 i-k+1~i 这 k 个点的连通信息!

如下图，我们不妨也称蓝线为轮廓线，因为只有轮廓线上的点的信息会对轮廓线右边的点的决策产生直接的影响．这样我们就很容易**确立状态**：



设 $f(i, S)$ 表示考虑完前 i 个点的连边情况后，i-k+1 .. i 这 k 个点的连通情况为 S.

转移状态：$O(2^k)$依次枚举点 i 与 i-1，…，i-k 这 k 个点是否相连．转移的时候需要注意：i-1, …, i-k 这 k 个点，任何一个连通块，i 最多只能与其中的一个点相连，这样可以避免环的出现．如果 i-k 在轮廓线上为一个单独的连通块，那么 i 必然与 i-k 相连，这样可以避免出现孤立的连通块．比如对于一个 $k = 5$ 的状态 $f(i-1,\{1,2,2,1,3\})$ 来说，如果点 i 与 i-2 和 i-1 相连，那么新的状态为 $f(i,\{1,1,2,2,2\})$．这样我们就可以在 $O(2^k * k)$的时间复杂度内完成状态的转移．

---

[11] Source : Noi2007 Day2 生成树计数, Count

算法实现：设 T_k 表示 k 个点的本质不同的连通情况的个数，搜索可知 T_5=52．动态规划的时间复杂度为 $O(n * T_k * 2^k * k)$，依然太大．可以发现当 $i \geq k$，状态 $f(i,S)$ 是否可以转移到 $f(i+1,S')$ 只与 S,S' 有关，这样我们就可以用矩阵乘法实现动态规划加速，由于这不是本文的重点，这里不再详细介绍．最终的时间复杂度为 $O(T_k^3 * \log_2 n)$，对于 k = 5, T_k = 52 的数据规模来说已经完全可以承受了，至此问题完整解决．

本题中的无向图非常特殊，每个点只和距离它不超过 k 的点有边相连，并且 k 非常小．对于棋盘模型的问题，可以抽象成一个特殊的无向图——$m * n$ 个点，每个点只与它上下左右四个点有边相连．那么对于一个与连通性有关的无向图问题，无向图具备怎样的特点才可以用基于状态压缩的动态规划来解决？ 分析以上几个问题，不难发现它们有一个共同点：给无向图中的点找一个序，在这个序中有边相连的两个点的距离不超过 p(p 很小)，这样我们就可以以当前决策完序中前 i 个，最后 p 个点的连通性为状态作动态规划．棋盘模型的问题中序即为从上到下，从左到右或从左到右，从上到下，p 为 m 或 n，因此棋盘模型的问题 m 和 n 中至少有一个数会非常小．

小结

本章写得比较简略，但是依然能够给我们很多的启示．处理这样的一类非棋盘模型的问题，一般的思路是寻找某一个序依次考虑每个点的决策，并分析哪些信息对以后的决策会产生影响，找到问题中的“轮廓线”，以轮廓线的信息来确立动态规划的状态．通常来说，轮廓线上的信息比较少，这也是能够作状态压缩动态规划的基础，像本题中 $k \leq 5$ 这样的条件往往能成为解决问题的突破口．

五．一类最优性问题的剪枝技巧

基于连通性状态压缩的动态规划问题的算法的效率主要取决于状态的总数和转移的开销，减少状态总数和降低转移开销成为了优化的核心内容．前面的章节我们提到了一些优化的技巧，这一章我们选取了一个非常有趣的题目 Rocket Mania 来介绍针对这样的一类最优性问题，如何通过剪枝使状态总数大大减少而提高算法效率．

【例 5】Rocket Mania[12]

问题描述

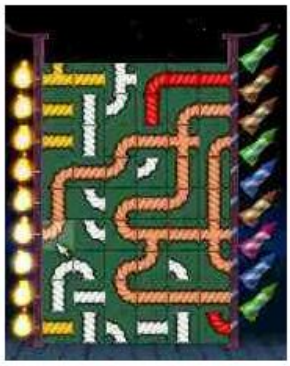
这个题目的背景是幻想游戏的“中国烟花”：

给你一个 9 * 6 的棋盘，棋盘的左边有 9 根火柴，右边有 9 个火箭．棋盘中的每一个格子可能是一个空格子也可能是一段管道，管道的类型有 4 种：

 L 型  —型  T 型  十型

一个火箭能够被发射当且仅当存在一条由管道组成的从一根点燃的火柴到这个火箭的路径．

给你棋盘的初始状态以及 *X*，你的目标是旋转每个格子内的管道 0，90，180 或 270 度，使得当点燃左边第 *X* 根火柴后，被发射的火箭个数尽可能多．

算法分析

确立状态：按照从左到右，从上到下的顺序依次考虑每一个格子，我们需要记录每个插头是否已经点燃以及它们之间的连通情况．因此状态为 $f(i,j,S,fired)$ 表示转移完(i,j)，轮廓线上 10 个插头的连通性为 *S*(把每个插头是否存在记录在 *S* 中), 10 个插头是否被点燃的 2 进制数 *fired* 的状态能否达到．

那么最后的答案为所有可以达到的状态 $f(9,6,S,fired)$ 中 *Ones*[*fired*]的最大值，其中 *Ones*[*x*]表示二进制数 *x* 的 1 的个数．

状态转移：依次枚举每一个格子的旋转方式(最多 4 种)，根据当前格子是否可以与上面的格子和左边的格子通过插头连接起来分情况讨论，*O(m)*扫描计算出新的状态．前面的题目我们已经很详细地介绍过棋盘模型的问题的转移方法，这

[12] Source : Zju 2125, Online Contest of Fantasy Game

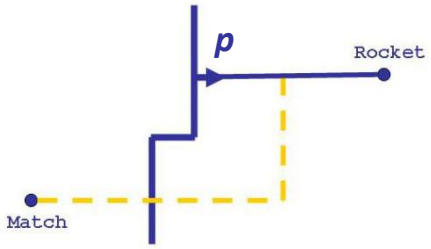
里不再赘述.

如果直接按照上面的思路作动态规划，Sample 也需要运行> 60s，实在令人无法满意．优化，势在必行．如何通过剪枝优化来减少扩展的状态总数，尽可能舍去无效状态成为了现在所面临的问题：

剪枝通常可以分为两类：一．可行性剪枝，即将无论之后如何决策，都不可能满足题目要求的状态剪掉；二．最优性剪枝，即对于最优性问题，将不可能成为最优解的状态给剪掉．我们从这两个角度入手来考虑问题：

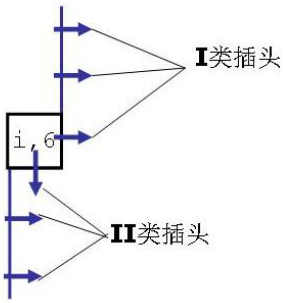
剪枝一：如果轮廓线上所有的插头全部都未被点燃，那么最后所有的火箭都不可能发射，所以这个状态可以舍去．这个剪枝看上去非常显然，对于大部分数据却可以剪掉近乎一半的状态．

剪枝二：如果轮廓线上有一个插头 *p*，它没有被火柴点燃且没有其它的插头与它连通，那么这个插头可以认为是“无效”插头．因为即使这个插头所在的路径以后会被点燃而可以发射某个火箭，那么一定存在另一条路径可以不经过这个插头而发射火箭，如图．这种情况下将插头设置为不存在．这是最重要的一个剪枝，大部分数据的状态总数可以缩小七八倍，甚至十几倍．



剪枝三：这是一个最优性问题，我们考虑**最优性剪枝**：对于一个格子(*i*, *j*)的两个状态 $(S_0, Fired_0)$，$(S_1, Fired_1)$，如果第一个状态的每一个存在的插头在第二个状态中不仅存在而且都被点燃，那么无论以后如何决策，第二个状态点燃的火箭个数不会少于第一个状态，这样我们就可以果断地舍去第一个状态．对于每一个(*i*, *j*)，选择 *Ones*[*Fired*]最多的一个状态 *Best*，如果一个状态一定不比 *Best* 好，就可以舍去．

剪枝四：从**边界情况**入手，边界状态非常特殊，也非常容易导致产生无效状态．分析一下，转移完最后一列的某个格子(*i*, 6)后，如果 I 类插头中某个插头 *p* 没有被点燃，并且 II 类插头中没有插头与它连通，那么这个插头就成了“无效”插头．



比较以上四种剪枝的效果，由于不同的棋盘初始状态扩展的状态总数差异较大，因此选取 10 组不同的棋盘初始状态来测试扩展状态的总数．10 组数据大致分布如下：Test 1~4 依次为全部“—”，“L”，“T”，“+”，Test 5 为奇数行“L”，偶数行为空，Test 6 为“L”，“T”交替．Test 7~10 为随机数据，“L”，“T”

分布较多，Test 10 的“—”较多[13].

表 4．四种剪枝扩展的状态总数的比较

Test	无剪枝	剪枝 1	剪枝 1,2	剪枝 1,2,3	剪枝 1,2,3,4
1	105	55	55	55	55
2	304250	121585	8758	8758	7620
3	18193954	13870994	1133036	11451	11451
4	55	55	55	55	55
5	958	2	2	2	2
6	4409663	2914156	438714	155049	153665
7	1662550	865193	137771	67473	67086
8	1697106	1075240	156778	7741	7741
9	557590	379853	75449	24856	13370
10	210290	43759	6084	6079	2957

由上表可以看出，优化后扩展的状态总数已经非常少了，剪枝的效果非常明显．我们从可行性和最优性两个角度，从一般情况和边界情况入手提出了 4 种剪枝方法，虽然有的剪枝看上去微不足道，但是它产生的效果确是惊人的．当然剪枝方法远远不止这 4 种，只要抓住问题的特征不断分析，就可以提出更多更好的剪枝方法．

值得一提的是 *S* 的状态表示，如果用普通的最小表示法，需要用 10 进制存储状态．由剪枝 2 可知如果一个插头属于一个单独的连通块，那么它一定被火柴点燃．如果使用广义的括号表示法，可以将无插头状态和单独的连通块插头都有“**( )**”表示，利用 *fired* 来区别，这样就可以用“**(**”，“**)**”，“**)(**”，“**( )**”——4 进制完整记录下 *n* 个格子的连通性，相比 10 进制有一定的常数优势．

至此，问题完整解决，60 多组测试数据运行时间<1.5s，实际效果确实不错．

这个问题还有一个加强版[14]：跟本题唯一不同的是，左边所有的火柴全部点燃，那么只要把初始状态中 9 个右插头全部设置为点燃，且为一个连通块即可．

小结

本章我们以 RocketMania 一题为例介绍了解决一类最优性的连通性状态压缩动态规划问题的剪枝技巧，从可行性和最优性这两个角度出发而达到减少状态总数的目的．在解题的过程中，要抓住问题的主要特征，多思考，多尝试，才能做的越来越好，**优化是无止境的**．

---

[13] 具体测试数据请参加附录

[14] Zju 2126 Rocket Mania Plus

六. 总结

本文立足于基于连通性状态压缩动态规划问题的解法和优化两个方面.

全文介绍了基于连通性状态压缩的动态规划问题的一般解法及其相关概念；针对一类特殊的问题——简单回路和简单路径问题，提出了括号表示法以及括号表示法的改进，最后从特殊问题回归到一般问题，提出了广义的括号表示法，这是文章的核心内容；接着对于一类棋盘染色问题和基于非棋盘模型的问题的解法作一个探讨；最后我们把重点放在了剪枝优化上，结合一个非常有趣的例题谈针对这类动态规划问题剪枝的重要性.

当然本文不可能涵盖基于连通性状态压缩动态规划问题的方方面面，因此关键是要掌握解决问题的思路，在解题的过程中抓住问题的特征，深入分析，灵活运用. 从上面的例题中可以发现，细节是不可忽略的因素，它很大程度上决定了算法的效率. 因此平时我们要养成良好的编程习惯，注意细节，注重常数优化. 做到多思考，多分析，多实验，不断优化，精益求精. **让我们做得越来越好!**

七、例题

浏览计划 （BZOJ 2595）

题目大意：

在 N*M 网格中选择一个四连通的连通块包含所有特殊点，每个方块有个选取代价，求合法方案中选取连通块所含方格的最小代价和。N,M<=8

分析：

普通的连通性 DP 模板题，在转移时保证所有特殊点必须被选即可。

代码：

```
#include <algorithm>
#include <iostream>
#include <cstring>
#include <cstdio>
#include <cmath>
using namespace std;

const int N = 11, M = 101, H = 97171, K = 1e5;
int n, m, ux, uy, lx, ly, last, ans, cont[N], mapv[N][N];
int pre[M][K];
bool p[M][K], g[N][N];

struct HashType {
    int top, info[H], nxt[K], sta[K], val[K];
    int times, vis[H];
```

```
    HashType(){}
    inline void Insert(const int &s, const int &v, const int &l, const int &prep, const bool &flag) {
        int x = s % H;
        if (vis[x] != times) vis[x] = times, info[x] = 0;
        for (int k = info[x]; k; k = nxt[k])
            if (sta[k] == s) {
                if (val[k] > v) {
                    val[k] = v;
                    pre[l][k] = prep;
                    p[l][k] = flag;
                }
                return ;
            }
        nxt[++top] = info[x]; info[x] = top;
        sta[top] = s; val[top] = v;
        pre[l][top] = prep; p[l][top] = flag;
    }
} *F, *G;

inline int encode() {
    int ret = 0, tot = 0;
    static int times, vis[N], lab[N];
    vis[0] = ++times;
    for (int i = 0; i < m; ++i) {
        if (vis[cont[i]] != times)
            lab[cont[i]] = ++tot, vis[cont[i]] = times;
        ret = ret << 3 | lab[cont[i]];
    }
    return ret;
}

inline void decode(int sta) {
    for (int i = m - 1; i >= 0; --i, sta >>= 3)
        cont[i] = sta & 7;
}

inline void Expand(const int &x, const int &y, const int &prep, const int &sta, const int &v) {
    decode(sta);
    int west = y > 0 ? cont[y - 1] : 0;
    int north = cont[y];

    bool flag;
```

```
58.     if (mapv[x][y] == 0) flag = false;
59.     else if (!north) flag = true;
60.     else {
61.         flag = false;
62.         for (int i = 0; i < m && !flag; ++i)
63.             if (i != y && cont[i] == north)
64.                 flag = true;
65.     }
66.     if (flag) {
67.         cont[y] = 0;
68.         int nxts = encode();
69.         F->Insert(nxts, v, x * m + y, prep, false);
70.         cont[y] = north;
71.     }
72.
73.     if (!west && !north) cont[y] = 8;
74.     else if (west > 0 && !north) cont[y] = west;
75.     else if (west > 0 && north > 0 && west != north) {
76.         for (int i = 0; i < m; ++i)
77.             if (cont[i] == west) cont[i] = north;
78.     }
79.     int nxts = encode();
80.     F->Insert(nxts, v + mapv[x][y], x * m + y, prep, true);
81. }
82.
83. inline void uptAns(const int &x, const int &y, const int &prep, int sta, con
    st int &val) {
84.     for (int j; sta > 0; sta >>= 3) {
85.         j = sta & 7;
86.         if (j > 1) return ;
87.     }
88.     if (val < ans) {
89.         ans = val;
90.         lx = x, ly = y;
91.         last = prep;
92.     }
93. }
94.
95. inline void record(const int &l, const int &prep) {
96.     if (l < 0) return ;
97.     int x = l / m, y = l % m;
98.     g[x][y] = p[l][prep];
99.     record(l - 1, pre[l][prep]);
100. }
```

```
101.
102. inline void Init() {
103.     F = new HashType();
104.     G = new HashType();
105.     ++F->times;
106.     F->Insert(0, 0, 100, 0, false);
107.     ans = 0x7fffffff;
108. }
109.
110. int main() {
111.     scanf("%d%d", &n, &m);
112.     for (int i = 0; i < n; ++i)
113.         for (int j = 0; j < m; ++j) {
114.             scanf("%d", &mapv[i][j]);
115.             if (mapv[i][j] == 0)
116.                 ux = i, uy = j;
117.         }
118.     Init();
119.     for (int i = 0; i < n; ++i)
120.         for (int j = 0; j < m; ++j) {
121.             swap(F, G);
122.             ++F->times; F->top = 0;
123.             for (int k = G->top; k >= 1; --k)
124.                 Expand(i, j, k, G->sta[k], G->val[k]);
125.             if (i > ux || (i == ux && j >= uy))
126.                 for (int k = F->top; k >= 1; --k)
127.                     uptAns(i, j, k, F->sta[k], F->val[k]);
128.         }
129.     record(lx * m + ly, last);
130.     printf("%d\n", ans);
131.     for (int i = 0; i < n; ++i) {
132.         for (int j = 0; j < m; ++j) {
133.             if (mapv[i][j] == 0)
134.                 putchar('x');
135.             else if (g[i][j]) putchar('o');
136.             else putchar('_');
137.         }
138.         putchar('\n');
139.     }
140.     return 0;
141. }
```

迷失游乐园（BZOJ 1187）

题目大意：

给出一个 n*m 的带权棋盘(n,m <= 8)，求这个棋盘上的一个回路，使得经过的格子点权和最大。

分析：

插头 DP 模板题。

代码：

```
#include <algorithm>
#include <iostream>
#include <cstring>
#include <cstdio>
#include <cmath>
using namespace std;

const int N = 103, M = 9;
const int H = 97171;
int n, m, ans, cont[M], mapv[N][M];

struct HashType {
    int top, info[H], nxt[H], sta[H], val[H];
    int times, vis[H];
    HashType(){}
    inline void Insert(const int &s, const int &v) {
        int x = s % H;
        if (vis[x] != times) vis[x] = times, info[x] = 0;
        for (int k = info[x]; k; k = nxt[k])
            if (sta[k] == s) {
                if (v > val[k]) val[k] = v;
                return ;
            }
        nxt[++top] = info[x]; info[x] = top;
        sta[top] = s; val[top] = v;
    }
} *F, *G;

inline int encode() {
    int ret = 0, tot = 0;
    static int times, vis[M], lab[M];
    vis[0] = ++times;
    for (int i = 0; i <= m; ++i) {
        if (vis[cont[i]] != times)
            vis[cont[i]] = times, lab[cont[i]] = ++tot;
```

```
        ret = ret << 3 | lab[cont[i]];
    }
    return ret;
}

inline void decode(int sta) {
    for (int i = m; i >= 0; --i, sta >>= 3)
        cont[i] = sta & 7;
}

inline void Expand(const int &x, const int &y, const int &S, const int &val)
{
    decode(S);
    int west = cont[y], north = cont[y + 1];
    if (!west && !north) {
        F->Insert(S, val);
        if (x == n - 1 || y == m - 1) return ;
        cont[y] = cont[y + 1] = 8;
        int nxts = encode();
        F->Insert(nxts, val + mapv[x][y]);
    }
    else if (west && !north) {
        if (x < n - 1) F->Insert(S, val + mapv[x][y]);
        if (y == m - 1) return ;
        cont[y] = 0; cont[y + 1] = west;
        int nxts = encode();
        F->Insert(nxts, val + mapv[x][y]);
    }
    else if (!west && north) {
        if (y < m - 1) F->Insert(S, val + mapv[x][y]);
        if (x == n - 1) return ;
        cont[y] = north; cont[y + 1] = 0;
        int nxts = encode();
        F->Insert(nxts, val + mapv[x][y]);
    }
    else if (west != north) {
        for (int i = 0; i <= m; ++i)
            if (cont[i] == west) cont[i] = north;
        cont[y] = cont[y + 1] = 0;
        int nxts = encode();
        F->Insert(nxts, val + mapv[x][y]);
    }
    else {
        cont[y] = cont[y + 1] = 0;
```

```
        int nxts = encode();
        if (nxts == 0 && ans < val + mapv[x][y]) ans = val + mapv[x][y];
    }
}

inline void Init() {
    F = new HashType();
    G = new HashType();
    ++F->times;
    F->Insert(0, 0);
    ans = -0x3f3f3f3f;
}

char ch;
inline int read() {
    int res, sgn = 0;
    while (ch = getchar(), ch < '0' || ch > '9') if (ch == '-') sgn = 1;
    res = ch - 48;
    while (ch = getchar(), ch >= '0' && ch <= '9')
       res = res * 10 + ch - 48;
    return sgn ? -res : res;
}

int main() {
    n = read(); m = read();
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < m; ++j)
            mapv[i][j] = read();
    Init();
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < m; ++j) {
            swap(F, G);
            ++F->times; F->top = 0;
            if (j == 0)
                for (int k = G->top; k >= 1; --k)
                    Expand(i, j, G->sta[k] >> 3, G->val[k]);
            else
                for (int k = G->top; k >= 1; --k)
                    Expand(i, j, G->sta[k], G->val[k]);
        }
    printf("%d\n", ans);
    return 0;
}
```

DP 套 DP

一、简介

DP 套 DP 简单地说就是通过一个外层的 dp 来计算使得另一个 dp 方程(子 dp)最终结果为特定值的输入数。

它的思想是一位一位确定子 dp 的输入，不妨考虑已经枚举了前 i 位了，那么注意到，由于我们只对 dp 方程的最终结果感兴趣，我们并不需要记录这前 i 位都是什么，只需要记录对这前 i 位进行转移以后，dp 方程关于每个状态的当前 DP 值就可以了，也就是说外层 dp 的状态是所有子 dp 的状态的值。

例如，假设我们有一个 DP(A)。A 读入一个序列 $a_1, a_2, \cdots, a_n$

DP(A)返回关于这个序列的一个结果。

现在我们想要知道有多少种 A 的序列可以返回给定的结果 R。

考虑第一个 DP(A)，如果我们 $a_1, a_2, \cdots, a_n$ 这么依次枚举。那么当我们处理到 a_i 的时候，有意义的只有 A 目前每个状态的值。也就是说，我们可以在状态里面记录“A 的所有状态的值”。

二、问题引入

给你一个只由 AGCT 组成的字符串 S (|S|<=15)，对于每个 0<=i<=|S|，问有多少个只由 AGCT 组成的长度为 m(1<=m<=1000)符串 T，使得 $LCS(S,T)=i$？

LCS 就是最长公共子序列。

首先让我们来考虑，给你两个串 S 和 T，我们怎么计算他们的 LCS?

显然我们可以列出方程 $f[i][j]$，表示 T 的前 i 个和 S 的前 j 个的 LCS。S 已经给定了，我们把这看成一个关于 T 的 dp，那么立刻就能发现这和刚才说的模型是一模一样的。

那么，我们一位一位枚举 T，用一个大状态记录每一个 $f[i][j]$ 的值是什么，当然这样的复杂度会很大，但是注意到对一个特定的串，$f[i][j]$ 和 $f[i][j-1]$ 的差只能是 0 或者 1，那么实际上状态最多也只有 2^15 种。

于是我们就可以枚举 T 的每一位，根据外层 DP 中的值，计算填入新位以后，新的内层

DP 值也就是 LCS 的结果，再将其作为外层 DP 下一位的状态。

通常 DP 套 DP 的流程都是一样的，找出内层 DP 的一些性质使得它能方便的进行编码，并作为外层 DP 的状态。之后进行外层 DP 时，我们枚举每一位，并根据当前状态算出内层 DP 当前的所有状态值，然后再将状态值与枚举的这一位结合，算出内层 DP 下一位时的结果，再将其编码为外层 DP 下一位时的状态。

三、例题

Hero meet devil（HDU 4899）

题目大意：

给你一个只由 AGCT 组成的字符串 S (|S|<=15)，对于每个 0<=i<=|S|，问有多少个只由 AGCT 组成的长度为 m(1<=m<=1000)符串 T，使得 $LCS(S,T)=i$？

LCS 就是最长公共子序列。

分析：

如上所述。

代码：

```
#include <algorithm>
#include <iostream>
#include <cstring>
#include <cstdio>
#include <cmath>

#define REP(i, a, b) for (int i = (a); i < (b); ++i)
#define PER(i, a, b) for (int i = (a); i > (b); --i)
#define FOR(i, a, b) for (int i = (a); i <= (b); ++i)
#define ROF(i, a, b) for (int i = (a); i >= (b); --i)

const int N = 15, M = 1003, mod = 1e9 + 7;
int T_T, n, m, nS, num1[1 << N], ans[N + 10], f[2][1 << N], a[N + 10], lcsA[N + 10], lcsB[N + 10], trans[1 << N][4];
char s[N + 10];
char chS[4] = {'A', 'C', 'G', 'T'};

inline int encode(const int *seq)
{
```

```
    int ret = 0;
    FOR(i, 1, n)
    {
        ret <<= 1;
        if (seq[i] != seq[i - 1]) ret |= 1;
    }
    return ret;
}

inline void decode(int *seq, int num)
{
    ROF(i, n, 1)
    {
        seq[i] = num & 1;
        num >>= 1;
    }
    FOR(i, 1, n) seq[i] += seq[i - 1];
}

inline void Init()
{
    nS = 1 << n;
    REP(i, 0, nS)
    {
        decode(lcsA, i);
        REP(k, 0, 4)
        {
            FOR(j, 1, n)
            {
                lcsB[j] = std::max(lcsB[j - 1], lcsA[j]);
                if (k == a[j]) lcsB[j] = std::max(lcsB[j], lcsA[j - 1] + 1);

            }
            trans[i][k] = encode(lcsB);
        }
    }
}

inline int Dp()
{
    memset(f, 0, sizeof(f));
    f[0][0] = 1;
    REP(i, 0, m)
    {
```

```
        int cur = i & 1, nxt = cur ^ 1;
        REP(j, 0, nS)
        {
            REP(k, 0, 4)
                (f[nxt][trans[j][k]] += f[cur][j]) %= mod;
            f[cur][j] = 0;
        }
    }

}

inline void Count_Ans()
{
    FOR(i, 0, n) ans[i] = 0;
    int sta = m & 1;
    REP(i, 0, nS)
        (ans[num1[i]] += f[sta][i]) %= mod;
}

inline void init_num1()
{
    REP(i, 1, 1 << N)
        num1[i] = num1[i - (i & -i)] + 1;
}

int main()
{
    init_num1();
    for (scanf("%d", &T_T); T_T > 0; --T_T)
    {
        scanf("%s%d", s + 1, &m);
        n = strlen(s + 1);
        FOR(i, 1, n)
        {
            if (s[i] == 'A') a[i] = 0;
            else if (s[i] == 'C') a[i] = 1;
            else if (s[i] == 'G') a[i] = 2;
            else a[i] = 3;
         }

         Init();
         Dp();
         Count_Ans();
         FOR(i, 0, n) printf("%d\n", ans[i]);
```

```
    }
    return 0;
}
```

练习题

树形 DP：BZOJ 1040 , BZOJ 1023 , BZOJ 1030

区间 DP：BZOJ 1068 , BZOJ 1090 , POJ 1651 , HDU 5115

数位 DP：BZOJ 1026 , HDU 2089 , HDU 3652 , CC CNTHEX

状压 DP：POJ 3254 , BZOJ 1087 , BZOJ 2064 , BZOJ 3195 , BZOJ 4067

插头 DP：BZOJ 2595 , BZOJ 1187 , Ural 1519 , BZOJ 2331

DP 套 DP：HDU 4899 , HDU 5079 , CF 578D , CF GYM 100257J

上述内容参考：

1.挑战程序设计竞赛（第二版）

2.2009 年国家集训队论文，浅谈数位类统计问题，刘聪

3.2008 年国家集训队论文，基于连通性状态压缩的动态规划问题，陈丹琦

4.WC2015 课件，计数问题选讲，陈立杰