Lesson3

> **计数DP**
>
> **整数划分**
>
> 特点
>
> - 求方案数时，由题意，并不考虑数与数之间的顺序。例如112、121、211都是同一种方案
>
> 解决方法
>
> - 转换为完全背包问题，背包容量是n，且有n个物品，物品体积分别为1,2,3...n。目标是求使背包容量恰好装满的方案数。
>
>   **分析思路1（完全背包）**
>
>   - 状态表示：f[i, j]
>   - 集合：**从1~i个物品中选，总体积恰好是j的选法集合**
>   - 属性：该集合中元素的**数量**
>   - 状态计算：集合划分
>   - 根据第i个物品选择的数量进行集合的划分。则对于选择k个第i个物品的子集，可以用**f[i-1, j-k*i]**进行表示，k为满足k * i <= j 的最大正整数值。因此朴素做法时间复杂度为O((n^2)*logn) 其中，logn为计算次数的调和级数求和 。进一步，利用完全背包的优化方法进行优化。
>
>   递推方程：f[i, j] = f[i-1, j] + f[i-1, j-i] + f[i-1, j-2i] + ... + f[i-1, j-k*i]
>
>   f[i, j-i] = f[i-1, j-i] + f[i-1, j-2i] + ... + f[i-1, j-k*i]
>
>   因此，**f[i, j] = f[i-1, j] + f[i, j-i]**，进一步降维可得 **f[j] = f[j] + f[j-1]** （循环从小到大进行）
>
> 具体实现1：完全背包变形，时间复杂度O(n^2)

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

using namespace std;

const int N = 1010, mod = 1e9+7;

int n;
int f[N];    //f[i][j]降去第一维
             //从1~i个物品中选，总体积恰好是j的选法集合中元素的数量

int main(void) {
    scanf("%d", &n);

    f[0] = 1;    //当容量为0时，只有一种选法即全都不选，

    //初始时，当j!=0时，f[j]=0，在没有选任何数的情况下不可能装满j

    //类似完全背包求解
```

```
    for (int i=1; i<=n; i++)
        for (int j=i; j<=n; j++)    //注意循环起点位置
            f[j] = (f[j]+f[j-i]) % mod;     //f[i][j] = f[i-1][j]+f[i]
[j-i];

    cout << f[n] << endl;

    return 0;
}

```

- **分析思路2**
  - 状态表示：f[i, j]
    - 集合：**所有总和是i，并且恰好表示成j个数的和的方案集合**
    - 属性：集合中的数量
  - 状态计算：集合划分
    - 分成两大类，第一类代表方案中**最小值是1**的子集，第二类代表方案中**最小值大于1**的子集。

      对于第一类子集，去掉该子集中每个方案的最后一个1，则该类别可以用f[i-1, j-1]（总和是i-1，且是j个数的和，和原子集中每个方案一一对应）表示。对于第二类子集中的每一个方案，由于其中每一个数都严格大于1，因此可以将方案里每一个数都减去1，且减去1前后仍一一对应，所以第二类子集可以用f[i-j, j]进行表示。

      递推方程：**f[i, j] = f[i-1, j-1] + f[i-j, j]**

      且最终答案为：**ans = f[n, 1] + f[n, 2] + ... + f[n, n]**

  具体实现

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

using namespace std;

const int N = 1010, mod = 1e9+7;

int n;
int f[N][N];    //f[i][j]表示总和是i，且恰好表示为j个数和的方案集合中元素数量

int main(void) {
    scanf("%d", &n);

    f[0][0] = 1;    //总和是0，由0个数构成，方案数是1

    for (int i=1; i<=n; i++)
        for (int j=1; j<=i; j++)
            //状态转移方程
            f[i][j] = (f[i-1][j-1]+f[i-j][j]) % mod;

    //枚举求和得答案
    int res = 0;
    for (int i=1; i<=n; i++) res = (res+f[n][i]) % mod;

```

```
    cout << res;

    return 0;
}
```

数位统计DP

计数问题

思路

- 首先实现函数count(n, x)，count(n, x)，1~n中x出现的次数

  所以[a, b]之间x出现的次数为：count(b, x) - count(a-1, x)（前缀和思想）

  因此问题可转化为求1~n中x出现的次数。

- 如何求出**1~n**中x出现的次数，分情况讨论

  分别求出**x在每一位上**出现的次数

  例如求1在第四位上出现的次数，则1 <= xxx1yyy <= abcdefg， **（1）若xxx范围为[000,abc-1]**，则后三位yyy可随意取，即yyy = 0~999，一共是abc*1000选法； **（2）当xxx=abc**， **（2.1）若d < 1**，则abc1yyy > abc0efg，方案数是0， **（2.2）若d=1**，yyy=[000, efg]，方案数为efg+1， **（2.3）若d>1**, yyy=[000, 999]，方案数为999+1=1000。根据实际情况将对应方案数相加，则是1出现在第四位上的次数。进而可以求出x在每一位上出现的次数，求和即可得在1~n中x出现的次数。

- 边界情况
  - 当x出现在最高位，第（1）种情况不存在
  - 当枚举数字0时（1）中不能取000，xxx范围为[**1**, abc-1]，即一共是（abc-1）*1000

具体实现

```
#include <iostream>
#include <algorithm>
#include <cstring>
#include <vector>

using namespace std;

//返回n中从l(高位)~r(低位)构成的数
int get(vector<int> num, int l, int r) {
    int res = 0;     // res 从0开始
    for (int i=l; i>=r; i--) res = res*10+num[i];
    return res;
}

//返回10^x
int power10(int x) {
    int res = 1;
    while (x--) res *= 10;
    return res;
}

//返回1~n中，x出现的次数
int count(int n, int x) {
```

```
    if (!n) return 0;    //当n=0，返回0

    //将n的每一位抠出来
    vector<int> num;
    while (n) {
        num.push_back(n % 10);
        n /= 10;
    }

    n = num.size();      //n更新为n的位数

    int res = 0;     //存储结果
     //从n的最高位开始统计x出现的次数，若x=0，则从次高位开始统计
    for (int i=n-1-!x; ~i; i--) {
        //情况1
        if (i < n-1) {
            res += get(num, n-1, i+1)*power10(i);    //当x不为0，方案数为abc*10^i
            if (!x) res -= 1*power10(i);     //当x=0时，为(abc-1)*10^i
        }

        //情况2.2
        if (num[i] == x) res += get(num, i-1, 0)+1;
        //情况2.3
        else if (num[i] > x) res += power10(i);
    }

    return res;
}

int main(void) {
    int a, b;

    //输入技巧
    while (scanf("%d%d", &a, &b), a || b) {
        if (a > b) swap(a, b);

        for (int i=0; i<=9; i++)
            cout << count(b, i)-count(a-1, i) << " ";
        puts("");
    }

    return 0;
}
```

状压DP

用二进制数来表示状态，每一位是0是1用于代表不同的状态

N的最大值能取到20，2^20 ~ 10^6，枚举状态就已经百万级别

蒙德里安的梦想

分析思路：转换为求解横向方块的放置方案数目

- 状态表示：f[i, j]

    - 集合：摆放第i列，上一列延伸到第i列的行的状态（用二进制表示）(j的范围是0~2^11)是j的方案集合
    - 属性：数量
- 状态计算：集合划分（二进制位运算思想）
    - **(j&k) == 0**（k表示所有上一列的状态j），两数相与不能发生冲突，保证横向方块的放置不发生冲突
    - 且所有连续的空行个数必须是偶数（用纵向方块填，每一个纵向方块纵向长度是2），即

      j | k 不能存在连续奇数个0，即保证剩下的空格能用纵向方块填满
    - 状态转移方程：**f[i, j] += f(i-1, k)**，k需要满足上述两种条件
    - 时间复杂度：O((11 * 2^11) * 2^11) ~ 4*10^7，一秒内能过
    - 答案为**f[m, 0]**，即**第m-1列没有延伸到第m列时，此时0~m-1列放置横向方块的方案才是合法方案**

```cpp
#include <iostream>
#include <cstring>
#include <algorithm>

using namespace std;

typedef long long LL;

const int N = 12, M = 1<<N;

int n, m;
LL f[N][M];     //f[i][j]表示摆放第i列，上一列延伸到第i列的行的状态是j的方案数量
bool st[M];      //预处理每个行状态是否具有连续偶数个0

int main(void) {
    while (scanf("%d%d", &n, &m), n || m) {

        memset(f, 0, sizeof f);     //对每组数据清空f

        //预处理每个行状态是否具有连续偶数个0
        for (int i=0; i<1<<n; i++) {
            st[i] = true;    //初始化为true
            int cnt = 0;     //存储连续偶数个0的个数
            for (int j=0; j<n; j++)      //对每一个状态枚举每一行
                if (i >> j & 1) {        //当前行是1
                    if (cnt & 1) st[i] = false;      //奇数个1
                    cnt = 0;     //cnt清零
                }
                else cnt++;

            if (cnt & 1) st[i] = false;
        }

        //DP过程
        f[0][0] = 1;     //摆放第0列，没有上一列，所以行状态是0的方案数为1
        //其余f[0][0~1<<n] = 0

        for (int i=1; i<=m; i++)
            for (int j=0; j<1<<n; j++)
```

```
                for (int k=0; k<1<<n; k++)
                    if ((j&k) == 0 && st[j | k])
                        f[i][j] += f[i-1][k];

        cout << f[m][0] << endl;     //f[m][0]中存储的才是答案
    }

    return 0;
}
```

最短Hamilton路径

分析思路

- 状态表示：f[i, j]
  - 集合：所有从0号点走到j号点，走过的所有点是状态i（二进制存储走过的路径）的所有路径

    例如i=(10001111)，所有1对应的结点已经走过，0对应的结点未走过。
  - 属性：最小值
- 状态计算：集合划分，分情况讨论
  - 用走过的倒数第二个点进行分类。所以有0~n-1-1(除去j点)类。

    递推方程：**f[i, j] = min(f[i-{j}, k]+a[k, j])**，其中i-{j}表示i状态存储的路径中除去j这个点。a[k, j]为从第k个点走到第j个点的距离

    时间复杂度**O(n * 2^n * n)**

    答案表示为：**f[(1<<n) - 1, n-1]**;

具体实现

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

using namespace std;

const int N = 21, M = 1<<N;

int n;
int w[N][N];
int f[M][N];     //f[i][j]表示所有从0号点走到j号点，走过的所有点是状态i的路径最小值

int main(void) {
    scanf("%d", &n);

    for (int i=0; i<n; i++)
        for (int j=0; j<n; j++)
            scanf("%d", &w[i][j]);

    memset(f, 0x3f, sizeof f);
    f[1][0] = 0;     //从0号点走到0号点，状态是0..01，故初始化为f[1][0]=0;

```

```
    for (int i=0; i<1<<n; i++)  //从0开始枚举所有二进制状态
        for (int j=0; j<n; j++) //具体考虑每一个状态的各个二进制位
            if (i>>j&1)     //当前状态第j个点已经走过
                for (int k=0; k<n; k++)  //枚举第j个点前一个走过的点，即倒数第二个走过的点
                    if ((i - (1<<j)) >> k & 1)  //从i中出去j，并且第k个点走过
                        f[i][j] = min(f[i][j], f[i-(1<<j)][k] + w[k][j]);

    cout << f[(1<<n)-1][n-1] << endl;   //输出答案

    return 0;
}
```

树形DP

没有上司的舞会

分析思路

- 状态表示：**f[u, 0]，f[u, 1]**
    - 集合：所有从 以u为根的子树中选择的方案，并且不选u这个点的方案集合（f[u, 0]）
      所有从 以u为根的子树中选择的方案，并且选u这个点的方案集合（f[u, 1]）
    - 属性：**Max**
- 状态计算：集合划分
    - 从根结点递归往下计算。假设si是u的儿子结点，则f[u, 0]，f[u, 1]分别为

      f[u, 0] = Σmax(f[si, 0], f[si, 1])

      f[u, 1] = Σf(si, 0)

      时间复杂度O(2n) = O(n)，所有状态一共只会计算2n次，均摊下来每个状态计算1次

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

using namespace std;

const int N = 6010, M = 6010;;

int n;
int happy[N];
int h[N], e[M], ne[M], idx;     //邻接表存图
bool has_fa[N];     //用于记录第i个结点是否有父结点，树根没有父结点
int f[N][2];//f[u][0]表示所有从以u为根的子树中选择的方案，且不选u的方案集合的最大h值
            //f[u][1]表示所有从以u为根的子树中选择的方案，且选u的方案集合的最大h值

void add(int a, int b) {
    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
```

```
}

void dfs(int u) {
    f[u][1] = happy[u]; //选择u，加上u的happy值

    //枚举u的所有儿子
    for (int i=h[u]; i!=-1; i=ne[i]) {
        int j = e[i];
        dfs(j);     //递归搜索j的儿子

        //根据状态转移方程计算f[u][0]与f[u][1]
        f[u][0] += max(f[j][0], f[j][1]);
        f[u][1] += f[j][0];
    }
}

int main(void) {
    scanf("%d", &n);

    for (int i=1; i<=n; i++) scanf("%d", &happy[i]);

    memset(h, -1, sizeof h);    //初始化链表头结点

    for (int i=0; i<n-1; i++) {
        int a, b;
        scanf("%d%d", &a, &b);
        add(b, a);  //新增一条b到a的边，表示b是a的上级
        has_fa[a] = true;   //a有父结点，不是根结点
    }

    int root = 1;
    while (has_fa[root]) root++;    //找出根结点

    dfs(root);  //从根结点递归往下搜索

    printf("%d\n", max(f[root][0], f[root][1]));

    return 0;
}
```

记忆化搜索、

动态规划的另一种实现方式，采用**递归**实现。相比循环可能会更容易理解

分析思路

- 状态表示：f[i, j]
    - 属性：所有**从(i, j)开始滑**的路径集合
    - 集合：路径集合的最大长度
- 状态计算：集合划分
    - 按向哪个方向滑，将集合分为四类（不重不漏）

      f[i, j] = max(f[i, j+1]+1, f[i+1, j]+1, f[i, j-1]+1, f[i-1, j]+1)（每一类存在的点的条件是下一步的高度更小，且没有超过矩形边界）
    - 递归做法的前提是**拓扑图，不能存在环**。若存在环形依赖，则发生死锁

具体实现：递归 & 记忆化搜索

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

using namespace std;

const int N = 310;

int n, m;
int h[N][N];    //高度
int f[N][N];    //f[i][j]所有从(i, j)开始滑的路径集合中长度的最大值

int dx[4] = {-1, 0, 1, 0}, dy[4] = {0, 1, 0, -1};

//返回每个状态的值
int dp(int x, int y) {
    int &v = f[x][y];   //用引用代替f[x][y];

    //记忆化搜索剪枝
    if (v != -1) return v;

    v = 1;  //初始化f[x][y]
    //根据递推方程求出四个方向中最大值
    for (int i=0; i<4; i++) {   //注意循环次数qaq
        int a = x+dx[i], b = y+dy[i];
        //约束条件
        if (a>=1 && a<=n && b>=1 && b<=m && h[a][b] < h[x][y])
            v = max(v, dp(a, b)+1);
    }

    return v;
}

int main(void) {
    scanf("%d%d", &n, &m);

    for (int i=1; i<=n; i++)
        for (int j=1; j<=m; j++)
            scanf("%d", &h[i][j]);

    memset(f, -1, sizeof f);    //记忆化搜索初始化

    int res = 0;    //存储答案，全局最长滑雪长度
    for (int i=1; i<=n; i++)
        for (int j=1; j<=m; j++)
            res = max(res, dp(i, j));    //dp(i, j)两个作用
                                        //作用1：求出f[i][j]
                                        //作用2：递归求出路径上其他点的f[ii][jj]

    printf("%d\n", res);

```

```
    return 0;
}
```

记忆化搜索优缺点:

算法题复杂度

- 时间复杂度
- 空间复杂度
- 代码复杂度

优缺点:

- 优点：记忆化搜索在代码复杂度上相比循环实现DP有很多优势，思路简单清晰。
- 缺点：在时间复杂度上比循环差一些，慢一个常数。另外在状态比较多时，容易爆栈