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)
>
> ```java
> static int N = 1010, mod = (int)1e9+7;
>
> static int n;
> static int[] f = new int[N];
>
> public static void main(String[] args) throws Exception {
>     ins.nextToken(); n = (int)ins.nval;
>
>     f[0] = 1;
>
>     for (int i=1; i<=n; i++)
>         for (int j=i; j<=n; j++)
>             f[j] = (f[j]+f[j-i]) % mod;
>
>     out.println(f[n]);
>
>     out.flush();
> }
> ```
>
> - **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
>
> ```java
> static int N = 1010, mod = (int)1e9+7;
>
> static int n;
> static int[][] f = new int[N][N];
>
> public static void main(String[] args) throws Exception {
>     ins.nextToken(); n = (int)ins.nval;
>
>     f[0][0] = 1;
> ```

```java

    for (int i=1; i<=n; i++)
        for (int j=1; j<=i; j++)
            f[i][j] = (f[i-1][j-1]+f[i-j][j]) % mod;

    int res = 0;
    for (int i=1; i<=n; i++)
        res = (res+f[n][i]) % mod;

    out.println(res);

    out.flush();
}
```

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

```java
//返回num从[l高位, r低位]构成的数字
static int get(List<Integer> num, int l, int r) {
    int res = 0;
    for (int i=l; i>=r; i--) res = res*10+num.get(i);
    return res;
}

//返回10的i次方
static int power10(int i) {
    int res = 1;
    while (i-- > 0) res *= 10;
    return res;
}

static int count(int n, int x) {    //1-n中x出现的次数
    if (n <= 0) return 0;   //n <= 0

    List<Integer> num = new ArrayList<>();
    while (n != 0) {    //把n中每一位存到数组里
        num.add(n % 10);
        n /= 10;
    }

    n = num.size();     //n置为数组长度
    int res = 0;
    //从最高位开始统计x出现的次数, x=0则从次高位开始统计（边界条件）
    //i表示x当前处于哪一位
    for (int i=n-1-(x == 0? 1: 0); i>=0; i--) {
        //情况1
        if (i < n-1) {
            res += get(num, n-1, i+1)*power10(i);
            if (x == 0) res -= 1*power10(i);    //边界情况
        }

        //情况2.2
        if (num.get(i) == x) res += get(num, i-1, 0)+1;

        //情况2.3
        if (num.get(i) > x) res += power10(i);
    }

```

```java
    return res;
}


public static void main(String[] args) throws Exception {
    ins.nextToken(); int a = (int)ins.nval;
    ins.nextToken(); int b = (int)ins.nval;

    while (a!=0 || b!=0) {
        if (a > b) { int t = a; a = b; b = t; }

        for (int i=0; i<10; i++)
            out.print((count(b, i)-count(a-1, i))+" ");
        out.println();

        ins.nextToken(); a = (int)ins.nval;
        ins.nextToken(); b = (int)ins.nval;
    }

    out.flush();
}
```

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**

```java
static int N = 13, M = 1<<N;

static int n, m;
static long[][] f = new long[N][M]; //f[i, j]代表i-1列延伸至i列的行的状态j
static boolean[] st = new boolean[M];   //st[i]代表状态i是否具有连续偶数个0

public static void main(String[] args) throws Exception {
    ins.nextToken(); n = (int)ins.nval;
    ins.nextToken(); m = (int)ins.nval;

    while (n!=0 || m !=0) {
        //f数组清零
        for (int i=0; i<N; i++) Arrays.fill(f[i], 0);

        //预处理st[]，大循环内进行
        for (int i=0; i<1<<n; i++) {    //枚举状态
            st[i] = true;

            int cnt = 0;
            for (int j=0; j<n; j++) {
                if (((i>>j)&1) == 1) {  //当前行是1
                    if ((cnt&1) == 1) { st[i] = false; break;}  //前有奇数个0, false
                    cnt = 0;    //偶数个0, 重置
                }
                else cnt++;
            }

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

        f[0][0] = 1;    //初始化

        //DP过程
        for (int i=1; i<=m; i++)    //枚举列，注意多枚举一列，答案为f[m][0]
```

```
            for (int j=0; j<1<<n; j++)  //枚举状态
                for (int k=0; k<1<<n; k++)  //枚举上一列状态
                    if ((j&k)==0 && st[j|k])
                        f[i][j] += f[i-1][k];

        out.println(f[m][0]);   //答案为f[m][0]

        ins.nextToken(); n = (int)ins.nval;
        ins.nextToken(); m = (int)ins.nval;
    }

    out.flush();
}
```

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

```java
static int N = 21, M = 1<<N, INF = 0x3f3f3f3f;

static int n;
static int[][] f = new int[M][N];   //f[i][j]代表从0-j个点的路径状态i的最小值
static int[][] w = new int[N][N];

public static void main(String[] args) throws Exception {
    ins.nextToken(); n = (int)ins.nval;

    for (int i=0; i<n; i++)
        for (int j=0; j<n; j++) { ins.nextToken(); w[i][j] = (int)ins.nval; }

    for (int i=0; i<M; i++) Arrays.fill(f[i], INF);     //初始化
    f[1][0] = 0;

    for (int i=0; i<1<<n; i++)  //枚举状态i
        for (int j=0; j<n; j++)
            if ((i>>j&1) == 1)    //走过j点
                for (int k=0; k<n; k++) //枚举j前一个走过的点k
                    if (((i-(1<<j))>>k&1) == 1)     //注意运算符优先级
                        f[i][j] = Math.min(f[i][j], f[i-(1<<j)][k]+w[k][j]);

    out.println(f[(1<<n)-1][n-1]);

    out.flush();
}
```

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

```java
static int N = 6010, M = N;

static int n;
static int[] happy = new int[N];
static int[] h = new int[N], e = new int[M], ne = new int[M];
static int idx;
static boolean[] has_fa = new boolean[N];
static int[][] f = new int[N][2];

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

//树形DP
static void dfs(int u) {
    f[u][1] = happy[u];     //如果选择该结点,则加上其happy值

    //对其儿子dp
    for (int i=h[u]; i!=-1; i = ne[i]) {
        int j = e[i];   //其儿子
        dfs(j);     //递归搜索其儿子

        //递归结束，回溯，更新dp数组
        f[u][0] += Math.max(f[j][0], f[j][1]);
        f[u][1] += f[j][0];
    }
}

public static void main(String[] args) throws Exception {
    ins.nextToken(); n = (int)ins.nval;

    Arrays.fill(h, -1);     //初始化邻接表头数组

    for (int i=1; i<=n; i++) { ins.nextToken(); happy[i] = (int)ins.nval; }

    for (int i=0; i<n-1; i++) {
        ins.nextToken(); int a = (int)ins.nval;
        ins.nextToken(); int b = (int)ins.nval;
        add(b, a);  //b是a的直接上司
        has_fa[a] = true;
    }

    int root = 1;
    while (has_fa[root]) root++;    //寻找树根

    dfs(root);  //从根结点开始向下dp

    out.println(Math.max(f[root][0], f[root][1]));

    out.flush();
}
```

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

```java
static int N = 310;

static int n, m;
static int[][] h = new int[N][N];
static int[][] f = new int[N][N];   //记忆化搜索

static int dp(int x, int y) {
    if (f[x][y] != -1) return f[x][y];

    f[x][y] = 1;    //初始化f[x][y]

    int[] dx= {-1, 0, 1, 0}, dy = {0, 1, 0, -1};
    for (int i=0; i<4; i++) {
        int xx = x+dx[i], yy = y+dy[i];
        if (xx>=1 && xx<=n && yy>=1 && yy<=m && h[xx][yy]<h[x][y])
            f[x][y] = Math.max(f[x][y], dp(xx, yy)+1);
    }

    return f[x][y];
}

public static void main(String[] args) throws Exception {
    ins.nextToken(); n = (int)ins.nval;
    ins.nextToken(); m = (int)ins.nval;

    for (int i=0; i<=n; i++) Arrays.fill(f[i], -1);

    for (int i=1; i<=n; i++)
        for (int j=1; j<=m; j++) { ins.nextToken(); h[i][j] = (int)ins.nval; }

    int res = 0;
    for (int i=1; i<=n; i++)
        for (int j=1; j<=m; j++)
            res = Math.max(res, dp(i, j));

    out.println(res);

    out.flush();
}
```

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