矩阵乘法优化

主讲人：邓哲也

矩阵乘法优化

对于两维以上的 dp，如 f[i][j][k]，如果它的转移全部来自 f[i - 1]，那我们可以使用矩阵来加速。

也就是把 f[i] 压缩成一维数组，f[i-1] 也压缩成一维数组。

转移方程 就相当于 f[i-1] 到 f[i] 乘上的转移矩阵。

比如 f[i][t] += f[i-1][s]，那么转移矩阵中的第 s 行第 t 列的元素就要加一。

这样对于转移 n 次的 dp，我们只要计算矩阵的 n 次幂即可。

时间复杂度可以从 O(n) 减少为 O(log n)

POJ 3734 Blocks

有 N 个方格排成一列，用红、绿、蓝、黄四种颜色来涂每个方格。

每个格子都必须涂，且只能涂一次。

问使得红色和蓝色的格子的数量为偶数的方案数。

$N \leqslant 10^9$

样例：

N=1，答案：2

N=2，答案：6

POJ 3734 Blocks

简单的线性模型。

我们可以用 f[i][a][b][c][d] 来表示填了前 i 个数字，且 4 种颜色的奇偶性分别为 a，b，c，d。（0 表示偶数，1 表示奇数）

方便起见我们把 abcd 看作一个二进制数 s，范围是 0～15 记为 f[i][s]

POJ 3734 Blocks

思考转移：

f[i][s] 可以转移到哪些状态呢？

枚举第 i+1 位的颜色：一共有 0，1，2，3 四种选择。

假设选择颜色 j，那么颜色 j 的奇偶性会发生改变，也就是

得到了新的状态 s ^ (1 << j)

f[i + 1][s ^ (1 << j)] += f[i][s]

POJ 3734 Blocks

f[i + 1][s ^ (1 << j)] += f[i][s]

可以发现每次都是从 f[i] 转移到 f[i + 1]

因此我们只要让矩阵的第 s 行第 s ^ (1 << j) 列 加一即可。

然后计算矩阵的 n 次幂，乘上 f[0] 就可以得到 f[n] 了。

POJ 3734 Blocks

```
struct matrix{
      int data[20][20];
}a;

matrix mul(matrix a, matrix b){
    matrix c;
    memset(c.data, 0, sizeof(c.data));
    for (int i = 0;i < n;i ++)
        for (int j = 0;j < n;j ++)
            for(int k = 0;k < n;k ++)
                c.data[i][j] = (c.data[i][j] + 1LL *
a.data[i][k] * b.data[k][j]) % m;
    return c;
}
```

POJ 3734 Blocks

```
matrix quickpow(matrix a, int k){
    matrix c;
    memset(c.data, 0, sizeof(c.data));
    for (int i = 0;i < n;i ++)
        c.data[i][i] = 1;
    while(k){
        if (k & 1) c = mul(c, a);
        k >>= 1;
        a = mul(a, a);
    }
    return c;
}
```

POJ 3734 Blocks

```
matrix A;
for (int i = 0;i < (1 << 4);i ++)
        for(int j = 0;j < 4;j ++)
                A.data[i][i ^ (1 << j)] = 1;
A = quickpow(A, n);
for (int i = 0;i < (1 << 4);i ++)
        if ((i & 1) == 0 && (i & 2) == 0)
                ans = (ans + A.data[0][i]) % MOD;
printf(“%d\n”, ans);
```

下节课再见