HDOJ 2294 - Pendant

——by A Code Rabbit

Description

有个高富帅，要送个很装逼的吊坠给他女朋友。

他有 k 种珠子，然后要串成一个 珠子个数小于等于 n 的链子。

因为要够装逼，所以这 k 种珠子都必须要用到。

好了，输入 n 和 k。

输出他可以做出多少种不一样的项链。

Types

Maths :: Matrix

Analysis

一看又是爽歪歪的递推题。

我们来考虑如何递推出。

设 f(x, y)为前 x 个位置，搞了 y 种不同的珠子。

然后我们考虑 f(x, y)和 f(x – 1, ...)关系。

而前后的关系，就差在第 x 个位置，要放什么类型的珠子。

我们可以放**前面 x – 1 个位置已经放过**的珠子，有 y 种颜色可以选。

那么就有

f(x – 1, y) * y 种。

也可以放**前面 x – 1 个位置还没放过**的珠子，有 k- (y – 1)种颜色可以选。

那么就有

f(x – 1, y – 1) * (k – y + 1)种。

所以我们就得到递推公式——

f(x, y) = f(x – 1, y) * y + f(x – 1, y – 1) * (k – y + 1)

好了，有了递推公式就碉堡了，剩下的就是矩阵乘法和快速幂。

当然，也要注意一些小的细节。

- 我们所求的不是 f(x, y)，而是 f(1, y) ~ f(x, y)的总和（因为珠子个数是小于等于 n 嘛）。

因此我们需要在矩阵的解向量中，增加一项 sum(x – 1)，且

sum(x – 1) = sum(x – 2) + f(x - 1, y)。

- 这题对 1234567891 取余，有点那啥。

在矩阵乘法中，我们通常是每一个矩阵元素计算完后，再去取余，这样可以减少大量运算时间（如果你不是这样的，那我告诉你，真的省很多）。

而这题由于这个除数真 TMD 的大，矩阵相乘的时候，两个元素相乘然后加加加加加...。

最后你懂的，就超过 long long 的范围，溢出了，囧。

深一步讲，因为 long long = int * int，而除数过于接近 int。
导致我们计算的数接近于 int * int * order（矩阵的阶），就爆了。

好吧，怎么办？
一种办法是用 unsigned long long，的确可以 A，速度还蛮快。
当然如果感觉不够妥妥的，可以每次在计算的时候，做完 int * int 马上 mod，就不会悲剧啦。
（如果你现在不知道我在讲什么，那就等到你在写矩阵相乘函数的时候，你就懂了～）

Solution

```
// HDOJ 2294
// pendant
// by A Code Rabbit

#include <cstdio>
#include <cstring>

const int ORDER = 33;
const int DIVISOR = 1234567891;

struct Matrix {
    long long element[ORDER][ORDER];
};

int t;
int n;
int k;

Matrix mat_unit;
Matrix mat_one;
Matrix mat_ans;

void INIT();
Matrix QuickPower(Matrix mat_one, int index);
Matrix Multiply(Matrix mat_a, Matrix mat_b);

void Show(Matrix mat) {
    printf("----------------------\n");
    for (int i = 0; i < k + 1; ++i) {
        for (int j = 0; j < k + 1; ++j) {
            printf("%d ", mat.element[i][j]);
```

```
        }
        printf("\n");
    }
    printf("----------------------\n");
}

int main() {
    scanf("%d", &t);
    while (t--) {
        scanf("%d%d", &n, &k);
        INIT();
        mat_ans = QuickPower(mat_one, n);
        printf("%d\n", k * mat_ans.element[1][0] % DIVISOR);
    }

    return 0;
}

void INIT() {
    memset(mat_unit.element, 0, sizeof(mat_unit.element));
    for (int i = 0; i < k + 1; ++i) {
        mat_unit.element[i][i] = 1;
    }
    memset(mat_one.element, 0, sizeof(mat_one.element));
    mat_one.element[0][0] = 1;
    mat_one.element[k][0] = 1;
    for (int i = 1; i < k + 1; ++i) {
        mat_one.element[i][i] = i;
        if (i > 1) {
            mat_one.element[i - 1][i] = k - i + 1;
        }
    }
}

Matrix QuickPower(Matrix mat_one, int index) {
    Matrix mat_result = mat_unit;
    while (index) {
        if (index & 1) {
            mat_result = Multiply(mat_result, mat_one);
        }
        mat_one = Multiply(mat_one, mat_one);
```

```
        index >>= 1;
    }
    return mat_result;
}

Matrix Multiply(Matrix mat_a, Matrix mat_b) {
    Matrix mat_result;
    int order = k + 1;
    for (int i = 0; i < order; ++i) {
        for (int j = 0; j < order; ++j) {
            mat_result.element[i][j] = 0;
            for (int k = 0; k < order; ++k) {
                mat_result.element[i][j] += mat_a.element[i][k] * mat_b.element[k][j] %
DIVISOR;
            }
            mat_result.element[i][j] %= DIVISOR;
        }
    }
    return mat_result;
}
```