Fast Modular Exponentiation

(Arup Guha, University of Central Florida)

The first recursive version of exponentiation shown works fine, but is very slow for very large exponents. It turns out that one prevalent method for encryption of data (such as credit card numbers) involves modular exponentiation, with very big exponents. Using the original recursive algorithm with current computation speeds, it would take thousands of years just to do a single calculation. Luckily, with one very simple observation and tweak, the algorithm can take a second or two with these large numbers.

The key idea is that IF the exponent is even, we can exploit the following mathematical formula:

$b^e = (b^{e/2}) \times (b^{e/2})$.

The key here is that we calculate $b^{e/2}$ only ONCE and can reuse the value that we get to do the multiplication.

But, even in this situation, the problem is that the sheer size of $b^{e/2}$ for very large e would make that one multiplication very slow.

But, consider the situation, where instead of calculating b^e, we were calculating b^e % n, for some relatively large value of n, maybe 20-100 digits. In this situation, the answer and any intermediate answer that is necessary, never exceeds n^2, which is relatively few digits.

In this case, reusing the value of $b^{e/2}$ % n accrues a HUGE benefit.

Note: When we test the following function (with mod) in C, it's important to choose a base that is smaller than 2^{15} to avoid overflow errors. The exponent may be any positive allowable int.

```
int modPow(int base, int exp, int n) {

    base = base % n;

    if (exp == 0)
        return 1;

    else if (exp == 1)
        return base;

    else if (exp % 2 == 0)
        return modPow((base * base) % n, exp / 2, n);

    else
        return (base * modPow(base, exp - 1, n)) % n;
}
```

If the exponent passed to the algorithm is odd, the next recursive call will contain an even exponent. Any call to an even exponent divides it by 2. Thus, for every two recursive calls, we divide the exponent by two. This, given the exponent, the number of steps the algorithm takes is O(log exp). Thus, even if exp = 10^{30}, this would take at most about 200 recursive calls total, which is much, much better than calculating this using a for loop that runs 10^{30} times.

This idea of “repeated squaring” or “dealing with even exponents by dividing by 2”, can be replicated in many places.

One place is matrix exponentiation. If we have a matrix we wish to raise to a high power (usually in these cases we might want the entries mod some value), then we can utilize this same exact concept!

Your multiplication would have to be a function and the recursive code would look a great deal like what is shown on the previous page, except for * would be replaced by the multiply function.

If you are clever, you can build matrices whose entries are answers to particular questions. Consider the following:

Let’s say I wanted to add up ($1 + a + a^2 + \ldots + a^n$) mod p. I could calculate the following:

$$\begin{bmatrix} a & 1 \\ 0 & 1 \end{bmatrix}^n \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Notice that for n = 1, the result is $\begin{bmatrix} a+1 \\ 1 \end{bmatrix}$.
For n = 2, the result is $\begin{bmatrix} a^2+a+1 \\ 1 \end{bmatrix}$. We can prove via induction that the result, in general is:

$$\begin{bmatrix} a & 1 \\ 0 & 1 \end{bmatrix}^n \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} \sum_{k=0}^{n} a^k \\ 1 \end{bmatrix}$$

Thus, if we want to find that desired sum, we simply set up the fast modular matrix exponentiation described above, multiplying the result with the column matrix 1, 1.

In general, a very high term of any linear recurrence relation mod a value can be calculated using this technique. Basically, you set up your matrix to store the coefficients of the recurrence relation and the last column vector will store the “base cases” of the recurrence, so that in successive multiplications, the resulting column vector will store the last few values of the recurrence relation needed to build the next value of the recurrence relation.

Example Problem: Too Many Rabbits!

Imagine that we start with one pair of new rabbits on month 1. In one month, the pair matures. From that point on, each subsequent month, they give birth to three pairs of rabbits. This is true of all pairs of rabbits. Determine how many pairs of rabbits there are after n ($1 \le n \le 10^9$) months. Since this number may be large, calculate it mod $10^9 + 7$.

After carefully working out a list of how many rabbits there are after n months as a recurrence, we find that if we let r(n) = the number of pairs of rabbits after n months, the function r(n) satisfies the recurrence:

r(n) = r(n-1) + 3r(n-2)

Normally, we could write a for loop to run up to n, but with n being as large as 10^9, we will not have time to do this. Instead, we can rewrite this recurrence to be embedded into a matrix multiplication operation and then use fast matrix exponentiation to obtain our solution quickly. Note that we can rewrite this recurrence as follows:

$$\begin{bmatrix} r(n) \\ r(n-1) \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} r(n-1) \\ r(n-2) \end{bmatrix}$$

Since the two column matrices are of the same form, just off by one, we can plug into this formula many times (iterating it), and obtain the following:

$$\begin{bmatrix} r(n) \\ r(n-1) \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} r(1) \\ r(0) \end{bmatrix}$$

Thus, we have reduced our original problem to solving a matrix exponentiation problem, which we can solve in O(lg n) matrix multiplications, which will easily (in this case), run in time.

<u>Number of Paths in an Unweighted Graph</u>

Consider the problem of counting the number of paths in a directed unweighted graph of length k between two given vertices u and v. An adjacency matrix stores the answers for k = 1, by definition. Now consider trying to build the answers for k = 2.

If I want to know the number of ways to travel from u to v using 2 edges, I want to find the number of intermediate vertices w such that uw and wv are edges. In some sense, I want to sum across all w, the product adj[u][w]*adj[w][v]. This sum is simply the entry adj^2[u][v], the item in row u, column v of the matrix adj squared, based on the definition of matrix multiplication.

More generally, if I want to know the number of paths of length k+1 from u to v, if I knew the number of paths of length k between all pairs of vertices, then I would want the following sum:

$$\sum_{w=1}^{n} NP[u][w] * adj[w][v]$$

Namely, if we want to travel from u to v in k+1 steps, first travel form u to w in k steps, which can be done in NP[u][w] ways, then if there is an edge from w to v, take this edge (multiply the first number by 1). If there isn't it zeros out. In this way, we add up all the different ways to travel from u to v in k+1 steps. Notice that this sum of products is PRECISELY what matrix multiplication does! (The row of the first matrix and column of the second stay the same while the "middle term" loops over all possible entries.)

Using mathematical induction, it follows that $M[i][j] = adj^k[i][j]$, the entry in row i, column j of the adjacency matrix raised to the kth power represents the number of paths of length k between vertex i and vertex j.

Example Problem: Knights (from 2014 South East Regional)

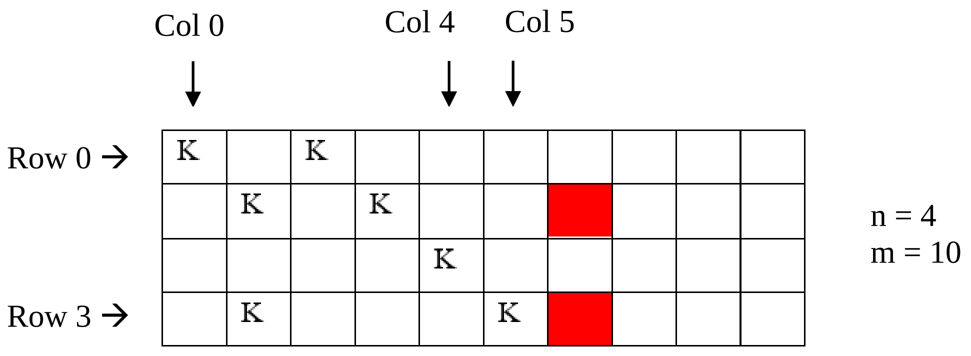
In this problem you are given an $n \times m$ grid, where $n \le 4$ but $m \le 10^9$, and the goal is to count how many ways we can place knights (the chess pieces) on the grid without any two knights attacking one another.

The strange bounds (one very small one, one very large one) are typical of fast matrix expo problems for reasons that will soon be clear.

Our strategy will be as follows:

Define a function f(x, y), where x = the number of completed columns and y = the state of the last two columns. The output of the function will be how many ways we can fill in the remaining part of the board. Notice that if we have these two pieces of information, then we can calculate the number of ways to "move to" f(x+1, y') where y' is the state of the last two columns after filling in one more column.

Consider the following example:



x = 6 completed columns

y = 01001000, knights on last two columns

filled squares can't have knights

In this picture, we have completed filling in 6 columns and the last two columns have one knight each, in rows 2 and 3 respectively. We can represent this state with an 8-bit bitmask (the concatenation of two 4-bit bitmasks), where the i^{th} bit is 1 iff there is a knight in row i. In this picture, row 0 is on the top, so this bitmask indicates that row 2 col 4 and row 3 col 5 have knights. (The least significant bit represents row 0 and so forth.)

Now consider adding knights to the 7th column, since the two red squares are banned, we can add these knights in four possible ways. In particular, the bit mask of this 7th column can be 0000, 0001, 0100 or 0101. In particular, the formula we get is as follows:

f(6, 01001000) = f(7, 10000000) + f(7, 10000001) + f(7, 10000100) + f(7, 10000101).

In fact, given any 8-bit bitmask, we can calculate the possible "next" 8-bit bitmask for ways of filling in the following column. So, for each of 2^8 possible inputs, we can calculate whether or not it's possible to move to one of the 2^8 possible outputs. But... notice that not all possible bitmasks are possible! In fact, only 81 of them (in this case) are. So, we only need to have 81 numbers to store each possible state and just remember what each of these masks are.

We precompute which 81 input states are possible, and then for each pair of input states (not necessarily distinct), we calculate if it's possible to transition between those two states, call them s_1 and s_2. If it is, we place a 1 in row s_1 and column s_2 of our transition matrix. If it isn't, we place a 0 here. In this manner we fill out an 81×81 transition matrix where each entry stores if it's possible to move from state s_1 to state s_2 or not by adding knights on a single column, based on the last two columns. Raising this matrix to the m-2 power will give us all the ways to complete the first m-2 columns of the board. To get our final answer, we must add up the expression f(m-2, mask), for all possible masks in the last two columns.

The key here is that the matrix multiplication naturally takes all possible masks representing the setting of "two columns ago" and "one column ago", and then matches them up with the masks that are "one column ago" and "the current column". When all of these 1’s get lined up, those all represent different "masks" for the middle column that can be used as a transition between the first and third of these columns. Roughly speaking, the computation shown on the previous page represents a single row being multiplied by a single column (one entry) in the matrix multiplication.