Matrix Exponentiation

Suppose you have a matrix A with n rows and n columns. We can define matrix exponentiation as:

$A^x = A * A * A * \ldots * A$ (x times)
with special case of x = 0:
$A^0 = I_n$
Here x is non-negative integer (i.e. 0, 1, 2, 3, …). Let's now analyse how fast we can compute A^x, given A and x.

The brute-force solution would be (written in pseudo code):

```
matrix_power_naive(A, x)
{
        result = I_n
        for i = 1..x:
                result = result * A
        return result
}
```

It runs in $\Theta(n^3 * x)$: we do x matrix multiplications on square matrices of size n, and each multiplication runs in $\Theta(n^3)$.

We can do better! Let's choose **x = 75** as an example. Write down binary representation of x:
$75_{10} = 1001011_2 \Rightarrow 75 = 2^0 + 2^1 + 2^3 + 2^6 = 1 + 2 + 8 + 64$.
Now we can rewrite A^x as
$A^{75} = A^1 * A^2 * A^8 * A^{64}$. There are a lot less multiplications. We had 75 of them, now we've got **only 4**, which is a number of 1's in binary representation of our x.
And it turns out, we can find A^{2r} really fast for any r:
$A^{2^\wedge 0} = A^1 = A$ (zero steps)
$A^{2^\wedge 1} = A^{2^\wedge 0 * 2} = A^{2^\wedge 0 + 2^\wedge 0} = A^{2^\wedge 0} * A^{2^\wedge 0}$ (one multiplication - n^3 steps)
$A^{2^\wedge 2} = A^{2^\wedge 1 * 2} = A^{2^\wedge 1 + 2^\wedge 1} = A^{2^\wedge 1} * A^{2^\wedge 1}$ (one multiplication - $2 * n^3$ steps)
…..
$A^{2^\wedge r} = A^{2^\wedge (r-1) * 2} = A^{2^\wedge (r-1) + 2^\wedge (r-1)} = A^{2^\wedge (r-1)} * A^{2^\wedge (r-1)}$ (one multiplication - $r * n^3$ steps)
Therefore, we can find $A^{2^\wedge r}$ in **$O(r * n^3)$** time, given A and r. Notice, however, that r = O(log(x)), because we compute $A^{2^\wedge r}$ only when there is r'th bit is set to 1 in binary representation of x, and length of binary representation of x is not more than $\log_2(x) + 1$.

So, for x = 75 we compute $A^2, A^4, A^8, A^{16}, A^{32}, A^{64}$ (that's the fastest way we can get A^{64}) in $6 * n^3$ steps, then perform 4 multiplications ($I_n * A^1 * A^2 * A^8 * A^{64}$) in $4 * n^3$ steps. In total, this gets us to **$10 * n^3$ steps for computing A^{75}**, instead of $75 * n^3$ steps with brute force.

We could implement this new idea in general (for any x) in the following way:

```
matrix_power_smart(A, x)
{
  result = I_n
  r = 0
  cur_a = A
  while 2^r <= x:
    if r'th bit is set in x:
      result = result * cur_a
    r += 1
    cur_a = cur_a * cur_a
  return result
}
```

Here, on every step of the while loop, cur_a = A^{2^r}.

Applications of Matrix Exponentiation.

Finding Nth Fibonacci number.

Fibonacci numbers $\mathbf{F_n}$ are defined as follows:

1. $F_0 = F_1 = 1$;
2. $F_i = F_{i-1} + F_{i-2}$ for $i \geq 2$.

We want to find F_N modulo 1000000007, where **N** can be up to 10^{18}. We know the recursive and iterative approaches which has **O(N)** running time complexity.. This can work in reasonable time for N up to $10^7 - 10^8$. If we want N up to 10^{18}, we have to switch to a faster approach.

Here is when matrices are helpful. Suppose we have a **vector** of (F_{i-2}, F_{i-1}) and we want to multiply it by some matrix, so that we get (F_{i-1}, F_i). Let's call this matrix **M**:

$$\begin{pmatrix} F_{i-2} & F_{i-1} \end{pmatrix} * M = \begin{pmatrix} F_{i-1} & F_i \end{pmatrix}$$

Two questions arise immediately:

1. What are the dimensions of M?
2. What are exact values in M?

We can answer them, using the definition of matrix multiplication:

1. **The size**. We multiply the (F_{i-2}, F_{i-1}), which has 1 row and 2 columns, by M. The result is (F_{i-1}, F_i), which has 1 row and 2 columns.
   By definition, if we multiply a matrix with n rows and k columns by a matrix with k rows and m columns, we get a matrix with n rows and m columns.
   In our case, n = 1, k = 2 (number of rows and columns of (F_{i-2}, F_{i-1})), and m = 2 (number of columns in the resulting (F_{i-1}, F_i)).
   Therefore, M has **k = 2 rows and m = 2 columns**.

2. **Values**. We now know that M has 2 rows and 2 columns, 4 values overall. Let's denote them by letters, as we usually do with unknown variables:
   $$M = \begin{pmatrix} x & y \\ z & w \end{pmatrix}$$
   We want to find x, y, z and w. Let's see what we get, if we multiply (F_{i-2}, F_{i-1}) by M by definition:
   $$\begin{pmatrix} F_{i-2} & F_{i-1} \end{pmatrix} * \begin{pmatrix} x & y \\ z & w \end{pmatrix} = \begin{pmatrix} F_{i-2} * x + F_{i-1} * z, & F_{i-2} * y + F_{i-1} * w \end{pmatrix}$$
   On the other hand, we know that the result of this multiplication must be (F_{i-1}, F_i):
   $$\begin{pmatrix} F_{i-2} * x + F_{i-1} * z, & F_{i-2} * y + F_{i-1} * w \end{pmatrix} = \begin{pmatrix} F_{i-1} & F_i \end{pmatrix}$$
   Now we can write the system of equations: $F_{i-2} * x + F_{i-1} * z = F_{i-1}$
   $F_{i-2} * y + F_{i-1} * w = F_i$
   The easiest way to satisfy the first equation is to set **x = 0, z = 1**.
   For the second equation, we look at the definition of Fibonacci numbers:
   $F_i = F_{i-1} + F_{i-2}$
   So the solution is **y = 1, w = 1**.

Now we know the size and contents of M:

$$\begin{pmatrix} F_{i-2} & F_{i-1} \end{pmatrix} * \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} F_{i-1} & F_i \end{pmatrix}$$

Initially, we have F_0 and F_1. Arrange them as a vector:

$$\begin{pmatrix} F_0 & F_1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \end{pmatrix}$$

Multiplying this vector with the matrix M will get us to (F_1, F_2) = (1, 2):

$$\begin{pmatrix} 1 & 1 \end{pmatrix} * M = \begin{pmatrix} 1 & 2 \end{pmatrix}$$

If we multiply (1, 2) with M, we get (F_2, F_3) = (2, 3):

$$\begin{pmatrix} 1 & 2 \end{pmatrix} * M = \begin{pmatrix} 2 & 3 \end{pmatrix}$$

But we could get the same result by multiplying (1, 1) by M two times:

$$\begin{pmatrix} 1 & 1 \end{pmatrix} * M * M = \begin{pmatrix} 2 & 3 \end{pmatrix}$$

In general, multiplying k times by M gives us F_k, F_{k+1}:

$$\begin{pmatrix} 1 & 1 \end{pmatrix} * M * M * \ldots * M \ (k\ times) = \begin{pmatrix} F_k & F_{k+1} \end{pmatrix}$$

Here matrix exponentiation comes into play: **multiplying k times by M is equal to multiplying by M^k**:

$$\begin{pmatrix} 1 & 1 \end{pmatrix} * M^k = \begin{pmatrix} F_k & F_{k+1} \end{pmatrix}$$

Computing M^k takes O((size of M)^3 * log(k)) time. In our problem, size of M is 2, so we can find N'th Fibonacci number in O(2^3 * log(N)) = **O(log(N))**:

```
fibonacci_exponentiation(N)
{
  if N <= 1:
    return 1
  initial = (1, 1)
  exp = matrix_power_with_modulo(M, N - 1, 1000000007) // assuming we've
defined M
  return (initial * exp)[1][2] modulo 1000000007
}
```

We multiply our initial vector (1, 1) by M^{N-1} and get initial * exp = (F_{N-1}, F_N). We must return F_N, so we take it from first row, second column of initial * exp. All indexation is done starting from 1.

There are other applications of matrix exponentiation such as:

- Finding Nth element of any linear recurrent sequence
- Finding sum of Fibonacci numbers upto N
- Computing multiple linear recurrent sequence at once and many more.

So matrix exponentiation is a very helpful technique to reduce time complexity.