Fast Matrix Exponentiation

With Applications

Nurseiit Abdimomyn

UNIST

October 27, 2018

Matrix Multiplication

Consider two matrices:

- Matrix A is $n * k$ dimensional.
- Matrix B is $k * m$ dimensional.

Notice that A's columns and B's rows number are identical!

Then we define matrix $C = A * B$ as:

$$\begin{bmatrix} c_{11} & c_{12} & \dots & c_{1m} \\ c_{21} & c_{22} & \dots & c_{2m} \\ \dots & \dots & \dots & \dots \\ c_{n1} & c_{n2} & \dots & c_{nm} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1k} \\ a_{21} & a_{22} & \dots & a_{2k} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nk} \end{bmatrix} * \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1m} \\ b_{21} & b_{22} & \dots & b_{2m} \\ \dots & \dots & \dots & \dots \\ b_{k1} & b_{k2} & \dots & b_{km} \end{bmatrix}$$

Thus, C is an $n * m$ dimensional matrix.

Which is calculated as: $c_{ij} = \sum_{r=1}^{k} a_{ir} * b_{rj}$

Couple of things to notice

- Matrix C has $n * m$ elements and each element is computed in k steps with given formula.
  Thus, we can obtain C in $O(n * m * k)$, given A and B.
- If $n = m = k$ (i.e. both A and B have n rows and n columns), then C has n rows and n columns, and can be computed in $O(n^3)$.

Some useful properties of matrix multiplication

- It is *not commutative*: $A * B \neq B * A$ in general case.
- It is *associative*:
  $A * B * C = (A * B) * C = A * (B * C)$ in case $A * B * C$ exists;
- If you have a matrix with n rows and n columns,
  then multiplying it by I_n gives the same matrix.
  i. e. $I_n * A = A * I_n = A$.
  Where I_n is a matrix with n rows and n columns of such form:

$$I_n = \begin{bmatrix} 1 & 0 & ... & 0 \\ 0 & 1 & ... & 0 \\ ... & ... & ... & ... \\ 0 & 0 & ... & 1 \end{bmatrix}$$

Matrix exponentiation

Suppose we need to find some $n * n$ dimensional square matrix A to the power p or A^p

We can do so via:

```
function matpow_naive(A, p):
  result = I_n
  for i = 1..p:
    result = result * A
  return result
```

Which will run in $O(n^3 * p)$

Fast Matrix exponentiation

Can we do it any faster?
Yes, we can apply the *BinPower* algorithm here:

```
function matBinPow(A, p):
  result = I_n
  while p > 0:
    if p % 2 == 1:
      result = result * A
    A = A * A
    p = p / 2
  return result
```

Which will run in $O(n^3 * \log p)$

Finding Nth Fibonacci number

Fibonacci numbers, F_n are defined as:

- $F_0 = F_1 = 1$
- $F_i = F_{i-1} + F_{i-2}$ for $i > 1$.

We want to calculate $F_n \ mod \ M$, where $n < 10^{18}$ and $M = 10^9 + 7$.

Finding Nth Fibonacci number

Suppose we have a vector (matrix with one row and several columns) 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) as a result.
Let's call this matrix M:
$(F_{i-2}, F_{i-1}) * M = (F_{i-1}, F_i)$

Two questions we should answer arise immediately:

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

Finding Nth Fibonacci number

We can answer them using the definition of matrix multiplication:

- The *size*.
  We multiply (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 L columns, we get a matrix with N rows and L columns.

  Therefore, matrix M has $K = 2$ rows and $L = 2$ columns.

Finding Nth Fibonacci number

We can answer them using the definition of matrix multiplication:

- The *values*.
  We now know that M has 2 rows and 2 columns, 4 values overall.
  Lets denote them by letters, as we usually do with unknown variables:

$$M = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

  We want to find a, b, c and d.

Finding Nth Fibonacci number

We have,

$$M = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

We want to find a, b, c and d.
Let's see what we get if we multiply (F_{i-2}, F_{i-1}) by M by definition:

$$(F_{i-2}, F_{i-1}) * \begin{bmatrix} a & b \\ c & d \end{bmatrix} = (a * F_{i-2} + c * F_{i-1}, b * F_{i-2} + d * F_{i-1})$$

Moreover, we know that the result of this multiplication must be (F_{i-1}, F_i):
$(a * F_{i-2} + c * F_{i-1}, b * F_{i-2} + d * F_{i-1}) = (F_{i-1}, F_i)$

Finding Nth Fibonacci number

Now we can write the system of equations:

- $a * F_{i-2} + c * F_{i-1} = F_{i-1}$
- $b * F_{i-2} + d * F_{i-1} = F_i$

The easiest solution is to set:
$a = 0, b = 1, c = 1, d = 1$
Thus, we obtain:

$$(F_{i-2}, F_{i-1}) * \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} = (F_{i-1}, F_i)$$

Finding Nth Fibonacci number

You may wonder why on Earth would we need to overcomplicate the problem, introduce matrices or else?
Hang on, you'll see in a moment.
Initially, we have F_0 and F_1.
$(F_0, F_1) = (1, 1)$
By multiplying this vector with the matrix M we get:
$(F_1, F_2) = (1, 2)$ or $(1, 1) * M = (1, 2)$.
If we multiply $(1, 2)$ by M:
$(1, 2) * M = (2, 3)$
But we could get the same result by multiplying $(1, 1)$ by M^2.
$(1, 1) * M^2 = (2, 3)$.
Now, can you see the pattern?

Finding Nth Fibonacci number

In general, we get:
$(1, 1) * M^k = (F_k, F_{k+1})$
Computing M^k takes $O(2^3 * log(k))$ time.
Thus, we can now find N-th Fibonacci number in $O(log(N))$ time.

You can find the reference code at *https://devnur.me/campunist*

General Solution for Recurrence Relations

We can generalize the Fibonacci problem's solution to solve linear recurrent sequences.
Let's take a look at more general problem than before. Suppose sequence A_i satisfies the following:

- $A_0 = a_0, A_1 = a_1, ..., A_{k-1} = a_{k-1}$ for given $a_0, a_1, ..., a_{k-1}$.
- $A_i = c_1 * A_{i-1} + c_2 * A_{i-2} + ... + c_k * A_{i-k}$ for given $c_1, c_2, ..., c_k$.

We need to find $A_N \ mod \ 10^9 + 7$, for $N \leq 10^{18}$ and $k \leq 50$.

Side note: sequence A_i is called **recurrent**, because computing A_i requires computing A_j for some $j < i$. Also, A_i is called **linear**, because it depends linearly on A_j for some $j < i$.

Notice that if we take $k = 2$, $a_0 = a_1 = 1$, $c_1 = c_2 = 1$, then this sequence will be equivalent to the *Fibonacci sequence* from the previous problem.
Let's try to solve using matrix multiplication right from the start.
If we obtain matrix M, such that:
$(A_{i-k}, A_{i-k+1}, ..., A_{i-1}) * M = (A_{i-k+1}, A_{i-k+2}, ..., A_i)$
we can get A_N as follows:
$(a_0, a_1, ..., a_{k-1}) * M^{N-k+1} = (A_{N-k+1}, A_{N-k+2}, ..., A_N)$

General Solution for Recurrence Relations

The two questions we had previously arise again:

1. What is the number of rows and columns of M?
   The reasoning is the same as with *Fibonacci numbers*:
   we multiply matrix with 1 row and k columns by M, and get matrix with 1 row and k columns.
   Therefore, M has k rows and k columns. In other words, M is a *square matrix* with size k.
2. What are the values in M? Let's denote them as x_{ij}:

$$M = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1k} \\ x_{21} & x_{22} & \dots & x_{2k} \\ \dots & \dots & \dots & \dots \\ x_{k1} & x_{k2} & \dots & x_{kk} \end{bmatrix}$$

General Solution for Recurrence Relations

Now we can write down the equations based on the definition of matrix multiplication:

$$A_{i-k} * x_{11} + A_{i-k+1} * x_{21} + ... + A_{i-1} * x_{k1} = A_{i-k+1}$$
$$A_{i-k} * x_{12} + A_{i-k+1} * x_{22} + ... + A_{i-1} * x_{k2} = A_{i-k+2}$$
$$...$$
$$A_{i-k} * x_{1k} + A_{i-k+1} * x_{2k} + ... + A_{i-1} * x_{kk} = A_{i}$$

From which we can easily obtain $x_{i,i-1} = 1$ for $i = 2, 3, ..., k$ and $x_{ij} = 0$ for $i = 1, 2, 3, ..., k$ and $j \leq k-1$, $j \neq i-1$.
The last equation looks like the definition of A_i. Based on that, we get $x_{ik} = c_{k-i+1}$:

$$M = \begin{bmatrix} 0 & 0 & ... & 0 & c_k \\ 1 & 0 & ... & 0 & c_{k-1} \\ 0 & 1 & ... & 0 & c_{k-2} \\ ... & ... & ... & ... & ... \\ 0 & 0 & ... & 1 & c_1 \end{bmatrix}$$

General Solution for Recurrence Relations

Now, when we have matrix M, there are no more obstacles.
We can implement the solution just like the original *Fibonacci problem*.

If you use *Fast Maxtrix Exponentiation* you get $O(k^3 * \log N)$ solution.

Finding the sum of Fibonacci numbers up to N

Lets go back to Fibonacci numbers:

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

Now, suppose a new sequence P_i is defined as follows:
$P_i = F_0 + F_1 + ... + F_i$
i.e the sum of first i elements of the Fibonacci Sequence.
We are to calculate $P_N \ mod \ 10^9 + 7$ for $N \leq 10^{18}$

Finding the sum of Fibonacci numbers up to N

Well, we can do that with matrices, again!
Let's imagine we have a matrix M, such that:
$(P_{i-1}, F_{i-2}, F_{i-1}) * M = (P_i, F_{i-1}, F_i)$
Then, we can get P_N as:
$(P_1, F_0, F_1) * M^{N-1} = (P_N, F_{N-1}, F_N)$
By now you must already be familiar with the method of obtaining M and determining its size. I'll show you M right away:

$$M = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

You can find sum of first N numbers of any recurrent linear sequence A_i, not just *Fibonacci*.

Solving DP problems with fixed linear transitions

Let's solve a little more complex but an interesting problem.

```
Count the number of strings of length L with
lowercase english letters, if some pairs of
letters can not appear consequently in those strings.
```

For $L \leq 10^7$ with up to 100 inputs.

The **bad** pairs are described in the input as a matrix:
$bad[i][j] = 1$ if letter j can not go after letter i in a *valid* string.
Otherwise, $bad[i][j] = 0$.

Solving DP problems with fixed linear transitions

Let's come up with a basic dynamic programming solution first.
Let $dp[n][last]$ be the number of valid strings of length n, which end with the letter $last$.
Then we can calculate dp in order of increasing n as follows:

```
dp[1][a] = dp[1][b] = ... = dp[1][z] = 1
for n = 2..L:
  for last = a..z:
    for next = a..z:
      if bad[last][next] != 1:
        dp[n+1][next] += dp[n][last]
```

This solution works in $\Theta(L * 26^2)$ time and won't pass TL of 1 second for $L = 10^7$.

Solving DP problems with fixed linear transitions

To speed things up, we can apply matrix multiplication. But how?

- Well, one can notice that transition from n to $n+1$ inside the loop happens in exactly the same way as it would for example, from $n+500$ to $n+501$.
  Because $bad[i][j]$ does not change over time!

- Also, another observation would be: transition formulas from n to $n+1$ are **linear**. To compute $dp[n+1][next]$, we sum up $dp[n][letter]$ for some $letter$.
  We do not compute $dp[n+1][next] += dp[n][letter] * dp[n][next]$ or anything alike, which *would* mean that the transition is **not linear**.

Solving DP problems with fixed linear transitions

These two observations combined provide a useful property:
there exists some matrix M, such that for any $n \geq 1$:
$(dp_n[a], dp_n[b], ..., dp_n[z]) * M = (dp_{n+1}[a], dp_{n+1}[b], ..., dp_{n+1}[z])$
To find such M we modify our initial solution a little:

```
for last = a..z:
  for next = a..z:
    dp[n+1][next] += dp[n][last] * good[last][next]
```

Where $good[i][j] = not\ bad[i][j]$ for all letters i, j.
Now, it is clear that matrix M is nothing but a matrix *good*.
The final solution is:
$(dp_1[a], dp_1[b], ..., dp_1[z]) * M^{L-1} = (dp_L[a], dp_L[b], ..., dp_L[z])$
That works in $O(26^3 * \log L)$ which is more than enough to pass the TL.

Thank you!