快速幂和矩阵快速幂

快速幂

计算 x^n 通常需要 n 次乘法, 时间复杂度为 $O(n)$, 当 n 非常大的时候, 运算效率很低.

快速幂是通过把n 转化为二进制来实现的. 例如: 计算 x^{14}, 14 可以用二进制表示为:

$$14 = (1110)_2 = 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

那么对应的乘法可以表示为:

$$x^{14} = x^{2^3} \times x^{2^2} \times x^{2^1}$$

转换后乘法运算次数减少, 每次计算 x^{2^n} , 再决定是否将这个数字加入到最终结果里面去. 代码如下:

```python
def fpowx(x, n):
    res = 1
    while n:
        if n & 1:
            res = res * x
        # compute x^2 x^4 x^8
        x *= x
        n >>= 1
    return res
```

乘法防止溢出

注: 对于 python 没有任何帮助, python整数直接相乘取模会快10倍

```
f_multi: 0.030360s
s_multi: 0.003781s
```

防止溢出的乘法和快速幂类似, 出现的原因是, 想两个数直接相乘发生溢出时, 改为相加运算, 并且可以直接取模. 这样保证了数据的正确性.

例如 $x \times 14$ 可以转化为:

$$x \times 14 = 8 \times x + 4 \times x + 2 \times x$$

```python
def fmulti(m, n, mod=10 ** 9 + 7):
    res = 0
    while n:
        if n & 1:
            res += m
        m = (m + m) % mod
        res %= mod
        n >>= 1
    return res
```

矩阵快速幂

矩阵乘法

对于 $A_{m\times k} \times B_{k\times n} = C_{m\times n}$

其计算公式为

$$C_{ij} = \sum_{p=1}^{k} A_{ip} \cdot B_{pj}$$

Python 代码如下:

```python
def matrix_multiply(matrix_a, matrix_b):
    n_row = len(matrix_a)
    n_col = len(matrix_b[0])
    n_tmp = len(matrix_a[0])
    matrix_c = [[0 for _ in range(n_col)] for _ in range(n_row)]
    for i in range(n_row):
        for j in range(n_col):
            for k in range(n_tmp):
                matrix_c[i][j] += matrix_a[i][k] * matrix_b[k][j]
    return matrix_c
```

快速幂

对于一个方阵 A, 也可以使用快速幂计算. 例如:

$$A^{14} = A^8 \times A^4 \times A^2 \times I$$

Python 代码如下:

```python
def get_unit_matrix(n):
    # matrix I 生成单位矩阵
    unit_matrix = [[0 for _ in range(n)] for _ in range(n)]
    for _ in range(n):
        unit_matrix[_][_] = 1
    return unit_matrix


def quick_matrix_pow(matrix_a, n):
    # A ^ n
    l = len(matrix_a)
    res = get_unit_matrix(l)
    while n:
        if n & 1:
            # 调用矩阵乘法
            res = matrix_multiply(res, matrix_a)
        a = matrix_multiply(a, a)
        n >>= 1
    return res
```

斐波那契数列的第N项

这个笔记主要源于我下午遇到的一道题，简介如下:

> *疫情爆发，第一天x个病人，第二天y个病人，病人在两天后有传染性，所以第三天 x+y，求第N天有多少个病人，结果需要对10^9 +7 取模*

这个题N可以取很大，$N \leq 10^{15}$，一直出错，下面两个标题分别是一般的思路和出问题的原因

带备忘录递归(爆栈)

开始的思路是用备忘录+递归. 爆栈了

大致的代码如下

```
from functools import lru_cache

@lru_cache(None)
def fibonacci(x, y, n):
  if n == 1:
    return x
  if n == 2:
    return y
  return fibonacci(x, y, n - 1) + fibonacci(x, y, n - 2)
```

思考通项公式

然后的思路是，斐波那契是有通项公式的，如果使用通项公式来计算，会不会快一点，但需要注意的是，这个题只是类似斐波那契数列，首先需要求解通项公式.

通项公式计算(OverflowError)

> *参考了知乎一名用户的解答: 斐波那契数列通项公式是怎样推导出来的？ - 土家族大酋长的回答 - 知乎*

他指出了一个结论，对于任意形如 $a_{n+2} = pa_{n+1} + qa_n$ 的数列求通项，可以用 $a_n = z^n$ 简化为 $z^2 = pz + q$

然后反解z，对于这个问题，显然有 $a_{n+2} = a_{n+1} + a_n$ 即$z^2 = z + 1$

可以求解 $z_1 = \frac{1+\sqrt{5}}{2}, z_2 = \frac{1-\sqrt(5)}{2}$

又指出通项可以写作 $a_n = Az_1^n + Bz_2^n$

可以通过a_1, a_2计算出A, B后直接得到这个题的通项

Python 代码如下

```python
def faboci(i, j, n):
    # i, j 分别为前两天的感染人数
    ta = (1 + sqrt(5)) / 2 #z_1
    tb = (1 - sqrt(5)) / 2 #z_2
    # 计算 A B
    B = (i / ta - j / (ta ** 2)) / (tb / ta - tb ** 2 / (ta ** 2))
    A = i / ta - B * tb / ta
    # 通过通项公式直接返回
    res = A * ta ** n + B * tb ** n
    return round(res)
```

这个解法会出错原因在于,当`n`比较大的时候,浮点数会溢出. 由于存在无理数不能转化成`Decimal`解决.

矩阵快速幂(通过)

这个思路是:

$$\begin{bmatrix} F_3 \\ F_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} B \\ A \end{bmatrix}$$

注意$F_2 = B, F_1 = A$ 所以有:

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-2} \times \begin{bmatrix} B \\ A \end{bmatrix}$$

Python 代码如下,注意加法乘法减法模运算律都成立:

```python
def MatrixMultiply(matrix_a, matrix_b):
    MOD = 10 ** 9 + 7
    n_row = len(matrix_a)
    n_col = len(matrix_b[0])
    matrix_c = [[0 for j in range(n_col)] for i in range(n_row)]
    for i in range(0, n_row):
        for j in range(0, n_col):
            for k in range(0, n_row):
                # 此处进行 mod 操作
                matrix_c[i][j] += (matrix_a[i][k] * matrix_b[k][j]) % MOD
                matrix_c[i][j] %= MOD
    return matrix_c
```

```python

def get_unit_matrix(l):
    unit_matrix = [[0 for j in range(l)] for i in range(l)]
    for k in range(l):
        unit_matrix[k][k] = 1
    return unit_matrix

def QuickMatrixPow(a, n):
    res_matrix = get_unit_matrix(len(a))
    while n:
        if n & 1:
            res_matrix = MatrixMultiply(res_matrix, a)
        a = MatrixMultiply(a, a)
        n = n >> 1
    return res_matrix


def get_Fib_n(i, j, n):
    if n == 0:
        return i
    elif n == 1:
        return j
    else:
        a = [[1, 1], [1, 0]]
        base = [[j], [i]]
        Fib_n = MatrixMultiply(QuickMatrixPow(a, n - 2), base)
        return Fib_n[0][0]
```