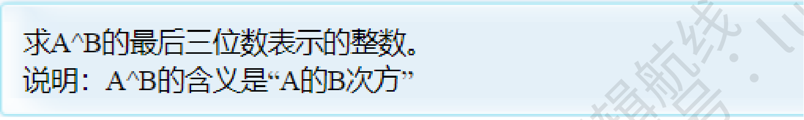
快速幂


求A^B的最后三位数表示的整数。
说明：A^B的含义是“A的B次方”

同余原理

1. $(a + b) \% p = (a \% p + b \% p) \% p$
2. $(a - b) \% p = (a \% p - b \% p) \% p$
3. $(a * b) \% p = (a \% p * b \% p) \% p$

```
#include <bits/stdc++.h>
using namespace std;

/**
 * 普通的求幂函数
 * @param base 底数
 * @param power  指数
 * @return  求幂结果的最后3位数表示的整数
 */
long long normalPower(long long base, long long power) {
    long long result = 1;
    for (int i = 1; i <= power; i++) {
        result = result * base;
        result = result % 1000;
    }
    return result % 1000;
}

int main() {
    clock_t start, finish;
    //clock_t为CPU时钟计时单元数
    long long base, power;
    cin >> base >> power;
    start = clock();
    //clock()函数返回此时CPU时钟计时单元数
    cout << normalPower(base, power) << endl;
    finish = clock();
    //clock()函数返回此时CPU时钟计时单元数
    cout << "the time cost is" << double(finish - start) / CLOCKS_PER_SEC;
    //finish与start的差值即为程序运行花费的CPU时钟单元数量，再除每秒CPU有多少个时钟单元，即为程序耗时
    return 0;

}
```

快速幂算法能帮我们算出指数非常大的幂，传统的求幂算法之所以时间复杂度非常高（为 O(指数 n)），就是因为当指数 n 非常大的时候，需要执行的循环操作次数也非常大。所以我们快速幂算法的核心思想就是每一步都把指数分成两半，而相应的底数做平方运算。这样不仅能把非常大的指数给不断变小，所需要执行的循环次数也变小，而最后表示的结果却一直不会变。让我们先来看一个简单的例子：

3^10=3*3*3*3*3*3*3*3*3*3

//尽量想办法把指数变小来，这里的指数为 10

3^10=(3*3)*(3*3)*(3*3)*(3*3)*(3*3)

3^10=(3*3)^5

3^10=9^5

//此时指数由 10 缩减一半变成了 5，而底数变成了原来的平方，求 3^10 原本需要执行 10 次循环操作，求 9^5 却只需要执行 5 次循环操作，但是 3^10 却等于 9^5,我们用一次（底数做平方操作）的操作减少了原本一半的循环量，特别是在幂特别大的时候效果非常好，例如 2^10000=4^5000,底数只是做了一个小小的平方操作，而指数就从 10000 变成了 5000，减少了 5000 次的循环操作。

//现在我们的问题是如何把指数 5 变成原来的一半，5 是一个奇数，5 的一半是 2.5，但是我们知道，指数不能为小数，因此我们不能这么简单粗暴的直接执行 5/2，然而，这里还有另一种方法能表示 9^5

9^5=（9^4）*（9^1）

//此时我们抽出了一个底数的一次方，这里即为 9^1，这个 9^1 我们先单独移出来,剩下的 9^4 又能够在执行“缩指数”操作了，把指数缩小一半，底数执行平方操作

9^5=（81^2）*(9^1)

//把指数缩小一半，底数执行平方操作

9^5=（6561^1）*(9^1)

//此时，我们发现指数又变成了一个奇数 1，按照上面对指数为奇数的操作方法，应该抽出了一个底数的一次方，这里即为 6561^1，这个 6561^1 我们先单独移出来，但是此时指数却变成了 0，也就意味着我们无法再进行“缩指数”操作了。

9^5=（6561^0）*(9^1)*(6561^1)=1*(9^1)*(6561^1)=(9^1)*(6561^1)=9*6561=59049

我们能够发现，最后的结果是 9*6561，而 9 是怎么产生的？是不是当指数为奇数 5 时，此时底数为 9。那 6561 又是怎么产生的呢？是不是当指数为奇数 1 时，此时的底数为 6561。所以我们能发现一个规律：最后求出的幂结果实际上就是在变化过程中所有当指数为奇数时底数的乘积。

```
long long fastPower(long long base, long long power) {
    long long result = 1;
    while (power > 0) {
        if (power % 2 == 0) {
            //如果指数为偶数
            power = power / 2;//把指数缩小为一半
            base = base * base % 1000;//底数变大成原来的平方
        } else {
            //如果指数为奇数
            power = power - 1;//把指数减去 1，使其变成一个偶数
            result = result * base % 1000;//此时记得要把指数为奇数时分离出来的底数的一次方收集好
            power = power / 2;//此时指数为偶数，可以继续执行操作
            base = base * base % 1000;
        }
    }
    return result;
}
```

虽然上面的快速幂算法效率已经很高了，但是我们仍然能够再一次的对其进行“压榨级别”的优化。我们上面的代码看起来仍然有些地方可以再进一步地进行简化，例如在 if 和 else 代码块中仍然有重复性的代码：

```
            power = power / 2;
            base = base * base % 1000;

            power = power - 1;//把指数减去 1，使其变成一个偶数
            power = power / 2;
```

可以压缩成一句：

```
power = power / 2;
```

因为 power 是一个整数，例如当 power 是奇数 5 时,power-1=4,power/2=2；而如果我们直接用 power/2=5/2=2。在整型运算中得到的结果是一样的，因此，我们的代码可以压缩成下面这样：

```
long long fastPower(long long base, long long power) {
    long long result = 1;
    while (power > 0) {
        if (power % 2 == 1) {
            result = result * base % 1000;
        }
        power = power / 2;
        base = (base * base) % 1000;
    }
    return result;
```

```
}
```