# 快速幂和快速乘

## 快速幂

快速求 $a^b\%p$ 的问题，时间复杂度为 $O(\log b)$，若对于 n 组数据，那么时间复杂度就是 $O(n\times\log b)$。

引入**模运算性质**：

加法：$(a+b)\%m=[(a\%m)+(b\%m)]\%m$

减法：$(a-b)\%m=[(a\%m)-(b\%m)]\%m$

乘法：$(a\times b)\%m=[(a\%m)\times(b\%m)]\%m$

除法：（不满足类似条件，感兴趣可以看看逆元）$(a/b)\%m\neq[(a\%m)/(b\%m)]\%m$

例如：$(100/50)\%20=2$，而$[(100\%20)/(50\%20)]\%20=0$，两者结果不相等

**一、暴力解法**：$O(n\times b)$ 会 TLE

基本思路：对于 n 组数据，分别循环 b 次求出 $a^b\%p$

```
#include<iostream>
using namespace std;
int main()
{
    int n;
    scanf("%d",&n);
    while(n--){
        int a,b,p;
        long long res=1;
        scanf("%d %d %d",&a,&b,&p);
        for(int i=1;i<=b;i++)
            res=res*a%p;
        printf("%lld\n",res);
    }
    return 0;
}
```

**二、快速幂做法**: $O(n\times\log b)$

基本思路:

1. 预处理出 $a^{2^0},a^{2^1},a^{2^2},\cdots,a^{2^{\log b}}$ 这 $\log b$ 个数

2. 将 $a^b$ 用 $a^{2^0},a^{2^1},a^{2^2},\cdots,a^{2^{\log b}}$ 这 $\log b$ 个数来组合，即组合成

$a^b=a^{2^{x_1}}\times a^{2^{x_2}}\times\cdots\times a^{2^{x_t}}=a^{2^{x_1}+2^{x_2}+\cdots+2^{x_t}}$ 用二进制表示。

举例:

11 的二进制是 1011，可表示成: $11=2^0\times1+2^1\times1+2^2\times0+2^3\times1$

所以 $a^{11}=a^{2^0+2^1+2^3}=a^{2^0}\times a^{2^1}\times a^{2^3}$

$a^{2^3},a^{2^1},a^{2^0}$ 属于是**倍乘**的关系，而中间没有 $a^{2^2}$，如何跳过 $a^{2^2}$ ?

$11_{10}=(1011)_2=8+2+1$

从低位往高位处理 1011（**右移一次，就把刚处理的低位移走**）

101<u>1</u>，处理末尾的 1：计算 $a$ $\qquad$ $res=a$

10<u>1</u>1，处理第 2 个 1：计算 $a^2$ $\qquad$ $res=res\times a^2=a^{1+2}$

1<u>0</u>11，处理 0：跳过 $a^4$

<u>1</u>011，处理 1：计算 $a^8$ $\qquad$ $res=res\times a^8=a^{1+2+8}=a^{11}$

```
/*
将 b 看作是二进制，从右往左第 k 位的 1 表示因子中含有 a^(2^k)
连乘会很大(有时候需要开 long long)，考虑不断求余
*/
int fast_pow(int a,int b,int p)
{
    int res=1;
    while(b)
    {
        if(b%2==1)//b&1
            res=res*a%p;
        a=a*a%p;
        b/=2;//b>>=1;
    }
    return res;
}
```

# 快速乘

快速乘使用二进制**将乘法转化为加法**,既可以加快运算速度,又可以防止直接相乘之后溢出。

这里说的快速乘并不是计算两数的乘法，而是计算 $a\times b\%p$ ,时间复杂度为 $O(\log b)$

**相比于普通乘法而言确实增加了时间复杂度，所以快速乘法并不快。快速乘法是解决 $a\times b\%p$ 时 $a\times b$ 的结果超出 long long 的数据范围的一种方法**

快速乘和快速幂原理类似，也是将运算转换为二进制处理；

以 $a\times 11\%p$ 为例： $a\times 11=a\times 2^3+a\times 2^1+a\times 2^0$

```
int fast_mul(int a,int b,int p)
{
    int res=0;
    while(b)
    {
        if(b%2==1)//b&1
            res=(res+a)%p;
        a=(a+a)%p;
        b/=2;//b>>=1;
    }
    return res;
}
```

# 贪心

**算法介绍：**

贪心算法（又称贪婪算法）是指，在对问题求解时，总是做出在当前看来是最好的选择。也就是说，不从整体最优上加以考虑，他所做出的是在某种意义上的局部最优解。
贪心算法不是对所有问题都能得到整体最优解，关键是贪心策略的选择，选择的贪心策略必须具备无后效性，即**某个状态以前的过程不会影响以后的状态，只与当前状态有关**。

**思想：**

贪心算法的基本思路是从问题的某一个初始解出发一步一步地进行，根据某个优化测度，每一步都要确保能获得局部最优解。每一步只考虑一个数据，他的选取应该满足局部优化的条件。若下一个数据和部分最优解连在一起不再是可行解时，就不把该数据添加到部分解中，直到把所有数据枚举完，或者不能再添加算法停止。

**例题：**

假设 1 元、2 元、5 元、10 元、20 元、50 元、100 元的纸币分别有 c0, c1, c2, c3, c4, c5, c6 张。现在要用这些钱来找给顾客 K 元，怎么用数目最少的钱来找零？

**贪心准则：**在不超过要找的零钱总数的条件下，每一次都选择面值尽可能大的纸币，直到凑成的零钱总数等于要找的零钱总数。

```
#include<iostream>
using namespace std;
int main(){
    int values[] = { 1, 2, 5, 10, 20, 50, 100 };//人民币面值集合
    int counts[] = { 3, 1, 2, 1, 1, 3, 5 };//各种面值对应数量集合
    int money = 442;
    int result[100];
    int len = sizeof(values) / sizeof(values[0]);
    for (int i = len - 1; i >= 0; i--) {
        int num = 0; //当前面值纸币的数量
        num = min(money / values[i], counts[i]); //当前纸币可以找的最大数量
        money = money - num*values[i];
        result[i] = num;
    }
    for (int i = 0; i < len; i++)//输出最后结果
        if(result[i])
            cout << "需要面额为" << values[i] << "的纸币" << result[i] << "张\n";
    return 0;
}
```