Misc. matematicas

November 1, 2019

1 Más teoría de números

Existe otra manera de hacer tests de primalidad:

- El método isProbablePrime de Java BigInteger!
- Algoritmo randomizado (**Miller-Rabin**) con propriedad de **completud probabilística**.
- Depende de un **factor de certeza** c.
- Si el output es 'false', seguro el número no es primo. Si es 'true', la probabilidad de que es un primo es:

$$p = 1 - (\frac{1}{2})^c.$$

- Lo más grande c, lo más confianza en el resultado, **pero** lo más tardío la respuesta.
- Tipicamente $c = 10$ es suficiente ($p > 0.999$) con tiempo razonable.

1.0.1 El pequeno teorema de Fermat

Teorema: si p es un número primo, entonces para **cualquier entero** x,

$$x^p \equiv x \mod p$$

Consequencia: si x no es divisible por p, $x(x^{p-1} - 1) = kp$, y eso implica:

$$x^{p-1} \equiv 1 \mod p.$$

(pero **la implicación sólo es en ese sentido**).

[https://www.geeksforgeeks.org/sum-of-each-element-raised-to-prime-1-prime/]

Given an array $arr[]$ and a positive integer P where P is prime and none of the elements of array are divisible by P. Find the sum of all the elements of the array raised to the power P˘1 i.e. $arr[0]^{P\breve{}1} + arr[1]^{P\breve{}1} + \ldots + arr[n\breve{}1]^{P\breve{}1}$ and print the result modulo P.

Input: An array $arr[]$ and a positive integer P where P is prime and none of the elements of array are divisible by P.

Output: Sum of the elements at the power $P - 1$.

Observaciones:

- Solución trivial en aplicación del pequeno teorema de Fermat.

Idea para un test de no-primalidad: si se muestra la existencia de un x no divisible por p tal que

$$x^{p-1} \not\equiv 1 \mod p.$$

entonces p **no es primo** (x se llama **testigo de Fermat** para p). Si repites el proceso y que no se encuentra un x con esa propriedad, entonces p es **probablemente primo**.

Repetir k veces:

- Muestrear x aleatoriamente en $[2, p-2]$.
- Si $x\%p \neq 0$ y $x^{p-1} \not\equiv 1 \mod p$, return false

return (probably) true

Observaciones y problemas:

- Requiere una implementación eficiente de la **potencia modular**.
- Para un x dado, existe una infinidad de números **no-primos** p tales que $x^{p-1} \equiv 1 \mod p$.
- Los números de Carmichael n son números **compuestos** (**no primos**) que satisfacen la misma propriedad que la enunciada en el pequeno teorema de Fermat: para cualquier x tal que $gcd(x, n) = 1$, $x^{n-1} \equiv 1 \mod n$.
- Con el test arriba, sería **imposible probar que esos números no son primos**. Lo bueno es que no son muchos (el primero es 561).

1.0.2 Algoritmo de Miller-Rabin

Extension del pequeno teorema de Fermat:

Si p es un número primo > 2, y $p\check{}1 = 2^s d$, con $s > 0$ y d impar, entonces para cualquier x primo relativo con p, o $x^d \equiv 1 \mod p$, o existe $t \in [0, s-1]$ tal que $x^{2^t d} \equiv -1 \mod p$.

El algoritmo usa la idea de muestrear x como desscrito arriba, e intentar encontrar un contra-ejemplo a esa propriedad para mostrar la no-primalidad:

- Descomponer $p = 2^s d + 1$, con d impar.
- Muestrear x en $[2, p-2]$.
- Evaluar $b = x^d \mod p$.
- Si $b = 1$ o $b = -1$, la prueba es inconclusiva.
- Si $b \neq 1$ y $b \neq -1$, tomar b al cuadrado modulo p, hasta d veces, hasta encontrar un valor -1.
- Si el valor -1 no ha sido obtenido, entonces p no es primo.

Si no encontramos x invalidando la propriedad, p **puede ser primo**.
Importante:

- Si p no es primo, la probabilidad del test de ser conclusivo es superior a $\frac{1}{4}$, o sea la probabilidad de no determinar que no es primo es inferior a $\frac{3}{4}$.
- Entonces, la probabilidad de tener k tests inconclusivos consecutivos es inferior a $(\frac{3}{4})^k$.
- Jugando con k, se puede disminuir la probabilidad de no-detección de los no-primos.

```
// Computes b^e mod md
llong modPow(llong b,llong e,llong md) {
    llong rm = 1;
    llong bb = b%md;
    while (e) {
        if (e%2)
            rm = (rm * bb)%md;
        e >>= 1;
        bb = (bb*bb)%md;
    }
    return rm;
}

bool isProbablePrime(llong n, int c) {
    if (n < 2 || (n%2==0))
        return false;
    if (n == 2)
        return true;

    // Writes n-1 = 2^s d with d odd
    llong d = n-1;
    int s = 0;
    while (d%2==0) {
        d >>= 1;
        s++;
    }

    llong b;
    // Performs c tests
    for (int k = 0; k< c; k++) {
        // Random x in [2,n-2]
        llong x = 2 + rand()%(n-3);
        b = modPow(x, d, n);
        // Inconclusive test
        if ((b == 1) || (b == n-1))
            continue;
        for (int i = 1; i < s; i++) {
            b = (b*b)%n;
            if (b == n-1)
                // Another inconclusive test
```

```
                break;
        }
        // If we did not find -1 in the loop, we return false
        if (b != n-1)
            return false;
    }
    // Probably prime (all inconclusive tests)
    return true;
}

cout << isProbablePrime(17,5) << endl;
cout << isProbablePrime(561,5) << endl;
cout << isProbablePrime(65537,5) << endl;
```

2 Probabilidades

Dos tipos de resoluciones posibles: * Buscar una solución explícita con forma cerrada. * Contar el cardenal del evento solicitado para evaluar una probabilidad (cuidado con los ordenes de magnitud). Eso lleva a problemas de combinatoria.

UVA 10056

Probability has always been an integrated part of computer algorithms. Where the deterministic algorithms have failed to solve a problem in short time, probabilistic algorithms have come to the rescue. In this problem we are not dealing with any probabilistic algorithm. We will just try to determine the winning probability of a certain player. A game is played by throwing a dice like thing (it should not be assumed that it has six sides like an ordinary dice). If a certain event occurs when a player throws the dice (such as getting a 3, getting green side on top or whatever) he is declared the winner. There can be N such player. So the first player will throw the dice, then the second and at last the N-th player and again the first player and so on. When a player gets the desired event he or she is declared winner and playing stops. You will have to determine the winning probability of one (The I-th) of these players.

Input: Input will contain an integer S ($S \leq 1000$) at first, which indicates how many sets of inputs are there. The next S lines will contain S sets of inputs. Each line contain an integer N ($N \leq 1000$) which denotes the number players, a floating point number p which indicates the probability of the happening of a successful event in a single throw (If success means getting 3 then p is the probability of getting 3 in a single throw. For a normal dice the probability of getting 3 is 1/6), and I ($I \leq N$) the serial of the player whose winning probability is to be determined (Serial no varies from 1 to N). You can assume that no invalid probability (p) value will be given as input.

Output: For each set of input, output in a single line the probability of the I-th player to win. The output floating point number will always have four digits after the decimal point as shown in the sample output.

La formula es relativamente simple: para que gane i, se considera todos los eventos en que gana i y de duración (en ciclo) k. Para que gane i en la iteración $k+1$, todos tienen que haber no ganado hasta esa iteración:

$$(1-p)^{kN}(1-p)^{i-1}p$$

así que la respuesta es:

$$p(1-p)^{i-1}\sum_{k=0}^{\infty}(1-p)^{kN} = p(1-p)^{i-1}\frac{1}{1-(1-p)^N}$$

Iteraciones del RANSAC.

RANSAC es un algoritmo muy usado en visión por computadora. Sirve entre otras cosas a estimar objetos geométricos a partir de n datos, entre los cuales unos pueden ser completamente incorrectos ("outliers"). La idea es iterar un cierto número de veces: elegir al azar p datos; calcular el objeto buscado; evaluarlo con los n datos; guardar l p-uplo de mayor consenso. Se supone:

- que con p datos se puede proponer un candidato del objeto buscado.
- que hay una proporción π de outliers.

Por ejemplo: regresión lineal con outliers.

Cuántas veces tenemos que iterar para estar con probabilidad $> 1-\epsilon$ de encontrar al menos una vez un "buen objeto"?

Tenemos:

$$1-\epsilon = 1-(1-(1-\pi)^p)^N$$

y se deduce:

$$N = \frac{\log\epsilon}{\log(1-(1-\pi)^p)}$$

UVA 10238

In the pictures below, you can see several types of Dices. It is very easy to assume that they are used in many different types of games. In this problem, the face of the dices will be like those on the first figure, that means each will have some dots it each face, which will indicate the value of that particular face. But the looks of the dices will be like those on the second figure, that means each dice may have 2, 3, 4, 5 up to 50 different faces. A dice with p faces have face values 1, 2, 3... p.

When a dice is thrown all the faces are equally likely to come on top. For example an ordinary dice in Figure 1 has face values 1, 2, 3, 4, 5, 6 and the probability that face with value 1 will be on top is 1/6. The same is the probabilityfor the other faces coming on top.

The situation becomes complicated when you throw a dice more than once dice. For example when you throw a dice twice, for any arbitrary incident the summation of the top two faces can be within the range (inclusive) 2 . . . 12. But their probability is not equally likely or 1/12 . The probability that the sum will be two is 1/36 , but the probability that some will be three is 2/36 and so on. Given the description of a dice throw, that's how many faces it have, how many throws will be made and value for the sum, you will have to find the probability of obtaining that sum.

Input: The input will contain several lines of input. Each line contains three integers F (The number of faces of the dice thrown, $1 \leq F \leq 50$), N (The number of throws and $0 \leq N \leq 50$) and S (The summation value for which you to find out the probability, $0 \leq S \leq 4000$).

Output: For each line of input you will have to produce one line of output which will be of the form 'a/b' (Here b = F/N). Here a/b means the probability value. Note that we don't need the floating point value of probability.

- El orden de magnitud del número total eventos es muy grande (F^N , hasta 50^{50}, entre 50 y 100 dígitos). Usar BigInteger.
- Usar programación dinámica:

$$n(N,S) = \sum_{k=1}^{F} n(N-1, S-k)$$

```
class Main {
    public static void main(String[] args) {
        BigInteger[][] vals = new BigInteger[51][4001];
        Scanner reader = new Scanner(System.in);
        while (reader.hasNext()) {
            int F = reader.nextInt();
            int N = reader.nextInt();
            int S = reader.nextInt();
            BigInteger fn = BigInteger.valueOf(F);
            fn = fn.pow(N);
            //System.out.println(F+" "+N+" "+S);
            if (N==0) {
                System.out.println(0+"/"+fn.toString());
                continue;
            }
            for (int i=1;i<=F;i++)
                vals[1][i]=BigInteger.ONE;
            for (int i=F+1;i<=4000;i++)
                vals[1][i]=BigInteger.ZERO;
            for (int i=2;i<=N;i++)
                for (int j=1;j<=S;j++) {
                    vals[i][j] = BigInteger.ZERO;
                    for (int k=1;k<=F;k++) if (j-k>=1) {
                        vals[i][j]=vals[i][j].add(vals[i-1][j-k]);
                    }
                }
                System.out.println(vals[N][S].toString()+"/"+fn.toString());
        }
    }
}
```

UVA 10328

Toss is an important part of any event. When everything becomes equal toss is the ultimate decider. Normally a fair coin is used for Toss. A coin has two sides head(H) and tail(T). Superstition may work in case of choosing head or tail. If anyone becomes winner choosing head he always wants to choose head. Nobody believes that his winning chance is 50-50. However in this problem we will deal with a fair coin and n times tossing of such a coin. The result of such a tossing can be represented by a string.

Such as if 3 times tossing is used then there are possible 8 outcomes: HHH HHT HTH HTT THH THT TTH TTT

As the coin is fair we can consider that the probability of each outcome is also equal. For simplicity we can consider that if the same thing is repeated 8 times we can expect to get each possible sequence once. In the above example we see 1 sequence has 3 consecutive H, 3 sequence has 2 consecutive H and 7 sequence has at least single H. You have to generalize it. Suppose a coin is tossed n times. And the same process is repeated 2^n times. How many sequence you will get which contains a sequence of H of length at least k.

Input: The input will start with two positive integer, n and k ($1 \leq k \leq n \leq 100$). Input is terminated by EOF.

Output: For each test case show the result in a line as specified in the problem statement.

Programación dinámica! Evaluar el cardenal del conjunto complemento, es decir el número de secuencias de N resultados con no k aguilas consecutivas.

- Sea n el id de la primera jugada que hacemos, y sea h el **número de aguilas consecutivas que tenemos hasta esa jugada**. Llamaremos $s[n][h]$ el número de secuencias sin k aguilas consecutivas (necesariamente $h \leq k-1$) empezando con la jugada n precedida de h aguilas.
- Observar:

$$s[N][h] = 1 \; \forall h \leq k-1$$

  y

$$\begin{aligned} s[n][h] &= s[n+1][h+1] + s[n+1][0] \text{ si } h+1 < k \\ s[n][h] &= s[n+1][0] \text{ sino.} \end{aligned}$$

- La solución es $2^N - s[0][0]$.

```
import java.util.Scanner;
import java.math.BigInteger;
class Main {
    public static void main(String[] args) {
        BigInteger[][] vals = new BigInteger[101][101];
        BigInteger[] pows   = new BigInteger[101];
        pows[0] = BigInteger.ONE;
        BigInteger t = BigInteger.valueOf(2);
        for (int i=1;i<=100;i++) {
            pows[i] = pows[i-1].multiply(t);
        }
        Scanner reader = new Scanner(System.in);
        while (reader.hasNext()) {
            int N = reader.nextInt();
```

```
            int K = reader.nextInt();
            // For all i in [0,K-1], there is one sequence starting at N pr
            for (int i=0;i<K;i++)
                vals[N][i]=BigInteger.ONE;
            for (int i=N-1;i>=0;i--) {
                for (int j=0;j<=K-1;j++) {
                    vals[i][j] = BigInteger.ZERO;
                    // If the starting Heads-chain before i+1 has a length
                    if (j+1<K)
                        vals[i][j] = vals[i][j].add(vals[i+1][j+1]);
                    // Add the count of all sequences starting at i+1 prece
                    vals[i][j] = vals[i][j].add(vals[i+1][0]);
                }
            }
            System.out.println(pows[N].subtract(vals[0][0]).toString());
        }
    }
}
```

3 Búsqueda de ciclos

Para problemas donde se hace un uso de secuencias definidas por **iteraciones sucesivas de una función** f:

$$x_0$$
$$x_i = f(x_{i-1})$$

¿Habrá un μ a partir del cual, cada λ (periodo), regresamos a x_μ , i.e. buscamos λ, μ t.q.

$$x_\mu = x_{\mu+\lambda}$$

Aplicaciones: generadores de números pseudo-aleatorios. Un buen algoritmo debería de **tener λ muy grande**.

Algoritmo **ingenuo**:

- Guardar los valores pasados (x_i, i) en una estructura de consulta eficiente.
- Checar en k si el nuevo valor calculado por iteración de la función está presente y, si lo es, deducir: $\mu = i$, $\lambda = k - i$.

$O(\lambda + \mu)$ en tiempo (al menos, dependiendo de la estructura de consulta: BSTs,...) y memoria.

3.0.1 Algoritmo de Floyd

Algoritmo en $O(\lambda + \mu)$ de complejidad temporal; $O(1)$ de memoria.

Ejemplo:

1 4 5 2 3 0 7 5 2 3 0 7

Paso 1: encontrar primero un múltiple de λ, $k\lambda$, basandose en que, para cualesquieras $i \geq \mu$ y $k > 0$, tenemos

$$x_i = x_{i+k\lambda}.$$

En particular, para $i = k\lambda$, tenemos que tener, $x_i = x_{2i}$.

Entonces: al incrementar simultaneamente desde x_0 dos apuntadores por pasos de 1 (x_i) y pasos de 2 (x_{2i}) ("la tortuga y la liebre"), cuando tenemos $x_i = x_{2i}$, tendremos un múltiple del periodo $i = k\lambda$. Observar que forzosamente: $i \geq \mu$.

Ejemplo:
1 4 5 2 3 0 7 5 2 3 0 7
1 **4 5** 2 3 0 7 5 2 3 0 7
1 4 **5** 2 **3** 0 7 5 2 3 0 7
1 4 5 **2** 3 0 **7** 5 2 3 0 7
1 4 5 2 **3** 0 7 5 **2** 3 0 7
1 4 5 2 3 **0** 7 5 2 3 **0** 7
Deducimos

$$i = (k\lambda =)5$$

Complejidad: después de μ iteraciones, vamos probando los periodos de 1 en 1 (número de iteración=periodo). Entonces al iterar, habremos pasado μ y probaremos el periodo μ.

Si $\mu = k\lambda + \delta$ con $k \geq 0$, $\delta < \lambda$, entonces en $\lambda - \delta$ iteraciones alcanzaremos el test del periodo $(k+1)\lambda$.

Entonces $O(\lambda + \mu)$

Paso 2: dejar el apuntador en x_i (la tortuga), regresar el otro en x_0. Avanzando los dos apuntadores por pasos de 1, buscamos μ como el primer j tal que

$$x_j = x_{j+i}.$$

Ejemplo:
1 4 5 2 3 **0** 7 5 2 3 0 7
1 **4** 5 2 3 0 **7** 5 2 3 0 7
1 4 **5** 2 3 0 7 **5** 2 3 0 7
Entonces, $\mu = 2$.
Complejidad en $\theta(\mu)$.
Paso 3: empezar en μ e iterar de uno en uno para encontrar el periodo λ.
1 4 **5** 2 3 0 7 5 2 3 0 7
1 4 **5 2** 3 0 7 5 2 3 0 7
1 4 **5** 2 **3** 0 7 5 2 3 0 7
1 4 **5** 2 3 **0** 7 5 2 3 0 7
1 4 **5** 2 3 0 **7** 5 2 3 0 7
1 4 **5** 2 3 0 7 **5** 2 3 0 7
Entonces, $\lambda = 5$.
Complejidad en $\theta(\lambda)$.
En total: $O(\lambda + \mu)$ en tiempo; $O(1)$ en memoria.
Otro algoritmo de eficiencia simiar: el de Brent.

- Empezando en x_0, la idea es encontrar la primera iteración potencia de 2, 2^k, tal que esta sea a la vez superior a μ y a λ, y encontrar λ por la misma ocasión.
- En cada iteración, mueves la tortuga en x_{2^i-1} y sigues moviendo la liebre (sólo una llamada a función en lugar de tres!) hasta $x_{2^{i+1}}$.

- Necesariamente encontrarás λ...
- Posicionas la tórtuga y el liebre en x_0 y $x_{\lambda-1}$ y buscas la μ por pasos de 1 paralelos.
- Misma tendencia de complejidad $O(\lambda + \mu)$ pero en práctica más eficiente según el autor.

UVA 11053

Flavius Josephus once was trapped in a cave together with his comrade soldiers surrounded by Romans. All of Josephus' fellow soldiers preferred not to surrender but to commit suicide. So they all formed a circle and agreed on a number k. Every k-th person in the circle would then commit suicide. However, Josephus had different priorities and didn't want to die just yet. According to the legend he managed to find the safe spot in the circle where he would be the last one to commit suicide. He surrendered to the Romans and became a citizen of Rome a few years later.

It is a lesser known fact that the souls of Josephus and his comrades were all born again in modern times. Obviously Josephus and his reborn fellow soldiers wanted to avoid a similar fiasco in the future. Thus they asked a consulting company to work out a better decision scheme. The company came up with the following scheme:

- For the sake of tradition all soldiers should stand in a circle. This way a number between 0 and N $-$ 1 is assigned to each soldier, where N is the number of soldiers.
- As changing numbers in the old scheme turned out to be horribly inefficient, the number assigned to a soldier will not change throughout the game.
- The consulting company will provide two numbers a and b which will be used to calculate the number of the next soldier as follows: Let x be the number of the current soldier, then the number of the next soldier is the remainder of ax 2 + b mod N .
- We start with the soldier with number 0 and each soldier calculates the number of the next soldier according to the formula above.
- As everyone deserves a second chance a soldier will commit suicide once his number is calculated for the second time.
- In the event that the number of a soldier is calculated for the third time the game will end and all remaining soldiers will surrender.

You are to write a program that given the number of soldiers N and the constants a and b determines the number of survivors.

Input: The input consists of several test cases. Each test case consists of a single line containing the three integers N ($2 \leq$ N ≤ 109), a and b ($0 \leq$ a, b < N) separated by white space. You may safely assume that the first soldier dies after no more than one million (106) steps. The input is terminated by a single number 0 which should not be processed.

Output For each test case output a single line containing the number of soldiers that survive.

- Después de μ, periodo de λ. A partir de $\mu + \lambda$, empezamos a revisitar cada persona.
- A partir de $\mu + 2\lambda$, empezamos a matar... hasta $\mu + 3\lambda$.
- Entonces habrá λ matados, y $N - \lambda$ sobrevivientes.

```
#include <iostream>
#include <vector>
```

```
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    int n,a,b;
    while (cin >> n && n>0) {
        cin >> a >> b;
        unsigned long long x = 0;
        unsigned long long klambda = 0;
        unsigned long long klambda2= 0;
        // Determinartion of klambda
        while (true) {
            klambda = (klambda*klambda)%n;
            klambda = (a*klambda+b)%n;
            klambda2= (klambda2*klambda2)%n;
            klambda2= (a*klambda2+b)%n;
            klambda2= (klambda2*klambda2)%n;
            klambda2= (a*klambda2+b)%n;
            if (klambda==klambda2) break;
        }
        klambda2 = 0;
        // Determination of mu
        while (true) {
            klambda = (klambda*klambda)%n;
            klambda = (a*klambda+b)%n;
            klambda2= (klambda2*klambda2)%n;
            klambda2= (a*klambda2+b)%n;
            if (klambda==klambda2) break;
        }
        // Determination of lambda
        unsigned long long save= klambda2;
        unsigned long long lambda = 0;
        while (true) {
            klambda2= (klambda2*klambda2)%n;
            klambda2= (a*klambda2+b)%n;
            lambda++;
            if (save==klambda2) break;
        }
        cout << n-lambda << endl;
    }
    return 0;
}
```