Ingeniería Matemática e Inteligencia Artificial

Memoria Aritmética Modular

Práctica 1

Matemática Discreta

Sergio Herreros Pérez
Daniel Sánchez Sánchez

INTERFAZ

Comenzamos describiendo nuestro proyecto con los ficheros que lo componen: imatlab.py, modular.py, commands.py y exceptions.py

- **imatlab.py:**
  Este fichero contiene: un main de ejecución, una clase Imatlab y la función run_commands.
  - El main de ejecución decide que hacer en función de los argumentos de entrada (sys.argv). Si no se incluye ninguno, ejecuta la interfaz de usuario. Si se incluyen dos, ejecuta los ficheros en modo batch. Es necesario incluir el nombre del fichero de entrada y el de fichero de salida, ambos con la extensión de archivo incluida.
  - La clase Imatlab se encarga de controlar la interfaz con modular.py y el usuario. Tiene dos atributos: un diccionario con objetos Command que ayudaran a simplificar tareas como el control de errores, conversión de entrada y salida, ejecución de funciones de modular.py, etc. y un pattern de regex pre-compilado en el __init__ que matchea solo entradas validas, capturando a su vez el nombre y los argumentos.
    Contiene cuatro funciones: una función parse_command para parsear el texto crudo de entrada usando el pattern regex, una función start_interface que inicia una interfaz de usuario interactiva, y una función run_batch, que ejecuta comandos por lotes desde un fichero de entrada a un fichero de salida. Se crea una instancia de la clase de manera persistente para cargar toda la clase una vez se importe imatlab.py.
  - La función run_commands es llamada por imatlab_benchamark.py, y recibe dos ficheros abiertos. Usando la instancia de Imatlab: imatlab_interface, se parsea y ejecuta cada línea del fichero de entrada, y se escribe el output en el fichero de salida.

- **modular.py**
  Este fichero contiene todas las funciones requeridas por la práctica, sin embargo, contiene también una clase Modulo con toda la funcionalidad compactada. A parte de las funciones obligatorias, tiene varias funciones extra necesarias para las mejorar las obligatorias, como "pollard_rho_module" que busca un divisor de n dada una semilla. Tiene también una "memoria cache", que se inicia al cargar la clase, en la cual podemos ir almacenando información temporal que puede acelerar ciertas funciones.
  Dicha clase además tiene una instancia de la clase Fp2, con la funcionalidad necesaria para operar en F_{p^2} y en particular, sacar la raíz $mod\ p$ de n en F_p.

- **commands.py**
  En este fichero se guardan las clases Command. Hay una primera clase Command, la clase padre, donde se definen las funciones principales de cada comando por defecto, que son parse_check_args, parse_output y execute, y las variables propias de cada comando, la función o funciones de modular.py a la/s que llamar y los últimos args que le han entrado. A partir de dicha clase, creamos una clase hija para cada comando.

- En parse_check_args, se parsea el string de argumentos. Se llama parse _check ya que está pensada para que, si finaliza sin ningún error, los argumentos son correctos, y si no, levante una excepción.
- En parse_output, se parsea la salida de la función de modular.py para cumplir los requisitos de interfaz de la practica para cada función.
- En execute_command, se llama a parsear_output capturando las excepciones, que serán de tipo NOP, después se ejecuta la función de modular.py con los argumentos parseados, capturando las exceptciones, que serán NE, y finalmente, devuelve la salida parseada.

- **exceptions.py**
  En este fichero únicamente se encuentran varias excepciones definidas por nosotros, específicas de este proyecto, como InvalidCommand o NoSolution. Cada una tiene un atributo "type", que puede ser NOP o NE, para distinguir entre los dos tipos de excepciones al ejecutar ficheros. También tienen un mensaje, para imprimirlo en modo interactivo.

- **Posibles mejoras:**
  - Añadir un comando help, que reciba el nombre de un comando e imprima información sobre él, o en su defecto imprima los posibles comandos.
  - Hacer un regex pattern para cada comando. Uno general que saque el nombre y el resto que cada uno compruebe y saque los argumentos con uno especifico.
  - Mejorar el control de excepciones añadiendo más funcionalidades y errores más personalizados y específicos.
  - Usar multiprocessing de manera adecuada para poder particionar la ejecución de ficheros en distintos procesos.
  - Añadir más funciones al módulo.
  - Explotar el uso de "memoria cache", pre-computación y más cosas que tiene sentido hacer en una clase modulo.

FUNCIONES

- **Es primo**:
  Hemos usado la variante determinista del algoritmo Miller-Rabin, el Miller's Test para comprobar si un número es primo o no.
  A continuación, describimos el algoritmo:
  Vamos a usar el pequeño teorema de Fermat, y el hecho que las únicas raíces de 1 módulo p son 1 y -1. Escribiendo n como $2^s \cdot d + 1$, podemos generar la secuencia, $a^{2^s \cdot d}$, $a^{2^{s-1} \cdot d}$, … , a^d . Definimos ahora un pseudoprimo fuerte base a como un n que cumple que o dicha secuencia es todo 1 o el primer número distinto de 1 es -1. Si esto no se cumple, n es compuesto.
  Asumiendo la Hipótesis de Riemann, bastaría con probar que un numero sea un pseudoprimo fuerte para cada base $a \in [2, 2 * log(n)^2]$. Sin embargo, podemos ahorrarnos bastantes bases para los números de primosTest:
  Hasta el 2047 usamos solo el 2 como base, hasta el 1373653 el 2 y el 3 y para cualquier otro número hasta 14386156093, usamos 2, 3 y 5, comprobando antes los ocho

números menores que 14386156093 que son pseudoprimos fuertes para esas tres bases. Si es mayor, usamos la prueba de Miller en su forma general. De esta manera, conseguimos un algoritmo determinista para cualquier n, algo más rápido para el tamaño de números de las pruebas.

- **Lista de primos:**

  Usamos la criba de Eratóstenes modificada para cualquier rango $[a, b)$. Si $a^2 < b$ sacamos los primos en $[2, \sqrt{b})$ con una primera criba, y con ellos cribamos los no primos en $[a, b)$, obteniendo la lista de primos en dicho rango. En caso contrario, hacemos la criba normal y filtramos los primos que estén dentro del rango.

- **Factorizar:**

  Para empezar, reducimos el número usando los primos hasta el 37. Después, si el número es menor a 10^{10}, factorizamos comprobando si los números del 41 a $\sqrt{n}$, haciendo paso 2 y luego 4, lo dividen. En caso de que alguno lo divida, dividimos n por ese número tantas veces como se pueda y guardamos el número de veces que lo ha dividido como el exponente del número. Al acabar esto, o n no ha cambiado (es primo) o es 1, y ya lo tendremos factorizado.

  Si el número es mayor a 10^{10} , usamos el algoritmo rho de Pollard . Este algoritmo puede encontrar un divisor de n de manera eficiente, así que es especialmente útil en los casos en los que $n = p_1 p_2$, con p_1 bastante menor que p_2. El algoritmo rho de Pollard funciona de la siguiente manera:

  Primero de todo, definimos una función, la cual aplicaremos repetidamente módulo n. En nuestro caso, usaremos $f(x) = x^2 + 1$, pues es la más frecuente y la que experimentalmente aporta mejores resultados. Al aplicar esta función repetidamente, encontraremos un ciclo que se repita en el campo F_n, pues hay un número finito de opciones, y el valor en cualquier momento del ciclo determina el siguiente. Además, también se encontrará uno en el campo F_{d_1}, siendo d_1 un divisor de n, y éste seguramente se dé antes que el ciclo en F_n, al haber menos valores posibles. Si encontramos el ciclo encontraremos d_1, pues si en $\{x_k \bmod d_1\}$ hay un $x_{k_1} \neq x_{k_2} \mid x_{k_1} \equiv x_{k_2} (\bmod d_1)$, $\gcd(x_{k_1} - x_{k_2}, n)$ será un divisor de n.

  Por lo tanto, solo nos queda encontrar dicho ciclo, y para ello usaremos el algoritmo de detección de ciclos de Floyd . Lo que haremos será guardar una variable x que en cada iteración coja el valor inmediatamente posterior en la secuencia, y una variable y que aumente de dos en dos. Además, comprobaremos el mcd en cada iteración, para ver si ya hemos encontrado el ciclo.

  Este algoritmo puede fallar, si el divisor que encuentra es n, y para ello lo probamos con distintas semillas. Si encuentra un divisor d, se factorizan d y n/d por separado. Si falla para todas las semillas, se factoriza con el método usado para números menores a 10^{10}.

- **MCD:**

  Usamos el algoritmo de Euclides mejorado con Least Absolute Remainder : al restar todos los a que quepan en b (hacer el módulo), en lugar de siempre quedarnos una por debajo, en algunos casos restamos a una vez más. Esto es, lo hacemos siempre y cuando $b\%a > a/2$, o $|r_1| > |r_2|$, siendo $r_1 = b\%a$ y $r_2 = b\%a + a$, para que el

número que quede al restar a una vez más sea menos que el que quede al hacer el módulo. De esta manera se mejora ligeramente el algoritmo. Este es más rápido que el simple para números bastante grandes.

- **Bezout:**
  Usamos el algoritmo extendido de Euclides simple, aunque con arrays de numpy, pues permiten hacer operaciones elemento a elemento con mayor facilidad.

- **MCD n:**
  Hacemos mcd repetidamente a cada número con el siguiente. Si el mcd es 1 en algún momento, lo será siempre, así que devolvemos 1 directamente.

- **Bezout n:**
  Parecido a Bezout normal, pero con arrays n dimensionales. Ahora cogemos cada vez un número de la lista y hacemos módulo al resto, actualizando la combinación lineal que nos queda. Además, en lugar de parar cuando uno de los dos números sea 0, paramos cuando todos menos uno sea 0, y ese será el mcd.

- **Coprimos:**
  Primero comprobamos si ambos son pares con operaciones de bits, y si no lo son hacemos el mcd, comprobando si es 1. Usamos el mcd simple ya que es más rápido para el tamaño de números del testPrimos.

- **Potencia módulo p:**
  Para esto, lo primero que haremos será pasar a representación binaria la potencia. A continuación, iteraremos sobre esta representación, y cada vez que encontremos un 1 multiplicaremos la "solución" (variable que iniciamos en 1) por la base elevada a la posición en la que estemos módulo p. Funciona porque $a^k = a^{2^{r_1}+2^{r_2}+\cdots+2^{r_n}} = a^{2^{r_1}} a^{2^{r_2}} \dots a^{2^{r_n}}$.
  El número de multiplicaciones es $n + k$ siendo n el número de bits del exponente y k el número de unos del exponente.

- **Inverso módulo p:**
  Lo hacemos usando Bezout de la siguiente manera: Estamos buscando un k que cumpla que $n \cdot k \equiv 1(mod\ p)$, y esto es equivalente a buscar un k que cumpla que $n \cdot k + p \cdot l \equiv 1(mod\ p)$, y esto se puede hallar usando el teorema de Bezout (n y p son coprimos pues p es primo). No hace falta guardar la l, pues solo nos interesa la k, ahorrándonos la mitad de las operaciones.

- **Euler:**
  Factorizamos n con el algoritmo simple hasta $\sqrt{n}$, y cada vez que encontramos un n primo hacemos la operación $n = n - \frac{n}{p}$. En caso de que no se haya encontrado ningún primo que lo divida, n es primo, así que devolvemos -1.
  Función ϕ de Euler

- **Legendre:**

  Usamos el criterio de Euler , $n^{\frac{p-1}{2}}(modp)$, junto con la función potencia modulo p.

- **Sistema:**

  Para resolver cualquier sistema de congruencias utilizamos el siguiente algoritmo:

  1. Simplificar el sistema a la forma $x_i = r_i(modm_i)$, dividiendo primero por el $mcd(a_i, b_i, m_i)$ y multiplicando después a b_i por la inversa de a_i. Si esta simplificación no es posible, el sistema no tiene solución ($mcd(a_i, m_i) \nmid b_i$).

  2. La idea principal es ver que, si podemos combinar dos ecuaciones en una, de manera que una se cumpla si y solo si se cumplen las otras dos. Si esto pasa, podemos reducir el sistema hasta quedarnos con una sola ecuación, que será la solución.

  3. Demostración de lo anterior:
     Veamos primero que las dos ecuaciones tienen solución si y solo si $r_1 = r_2 mod(g)$, con $g = mcd(m_1, m_2)$:

     $\Rightarrow$: Si ambas ecuaciones tienen solución, entonces $m\_i \mid x_i - r_i$. En particular, $g \mid x_i - r_i$ por tanto $x_1 = r_1(modg)$ y $x_2 = r_2(modg)$ por tanto $x_1 = x_2(modg)$.

     $\Leftarrow$: Si $r_1 = r_2(modg)$, entonces podemos hacer lo siguiente. Encontramos dos enteros p y q tal que $m_1 * p + m_2 * q = g$, lo cual siempre podemos hacer por Bezout, y construimos la solución como $x = r_1 * \frac{m_2}{g} * q + a_2 * \frac{m_1}{g} * p$. Ahora es fácil de ver que dicha x es siempre $r_1(modm_1)$ y $r_2(modm_2)$.

     Hagamos un caso ya que el otro es análogo: sustituimos $m_1 * p = g - m_2 * q$ en la ecuación llegando a $x = r_2 + \frac{r_1 - r_2}{g} * m_2 * q$. Pero como $r_1 = r_2(modg)$, $\frac{r_1 - r_2}{g}$ es siempre un entero, y por tanto x es siempre $r_2(modm_2)$.

     Por tanto, por construcción, las ecuaciones tienen solución si $r_1 = r_2(modg)$, y además conocemos dicha solución.

     Podemos llevarlo un paso más, y darnos cuenta de que la mínima solución es $x\big(modMCM(m_1, m_2)\big)$.

  4. Por tanto, podemos simplificar dos ecuaciones contiguas a la nueva ecuación, e iterar sobre las ecuaciones hasta terminar o hasta que dos de ellas no se puedan simplificar, es decir que no tengan solución.

  https://forthright48.com/chinese-remainder-theorem-part-2-non-coprime-moduli/

- **Raíz mod n:**

  Calculamos $\sqrt{n}\ (mod p)$ utilizando el algoritmo de Cipolla. Este consiste en calcular un $x = \left(a + \sqrt{a^2 - n}\right)^{\frac{p-1}{2}}$, siendo $w = a^2 - n$ un elemento cuya raíz no existe en F_p. Primero, buscamos dicha a lo cual no es muy difícil ya que la mitad de los elementos de F_p cumplen la condición y usando el símbolo de Legendre podemos comprobar que no tiene raíz.
  Ahora, definimos el campo F_{p2} de manera similar a los complejos para R, y definimos la operación de multiplicación y exponenciación en dicho campo para poder calcular la x. Está demostrado que al final del proceso, la parte "compleja" del elemento de F_{p2} se cancela, y el algoritmo nos da una solución que vive en F_p.

- **Ecuación cuadrática:**

  Aplicamos la fórmula de la ecuación cuadrática, esto es, $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$, usando para la raíz Cipolla y para a^{-1} la inversa módulo p.

- **Mínimo común múltiplo:**
  Calcula el mínimo común múltiplo de n números dados, dividiendo el producto de todos ellos por su máximo común divisor.

- **Pseudo primo fuerte base a:**
  Comprueba si n es un strong pseudoprime base a con el algoritmo de Miller-Rabin.