GRUPO: 12

INTEGRANTES:

Andrés Martínez Fuentes
Pablo García Molina

DECISIONES Y ESTRATEGIAS GENERALES:

- Hemos optado por un modelo de estructura del programa con un orden secuencial: librerías, clases, funciones y bloque de código principal (main)

- Queríamos un procesado limpio, rápido y eficiente, pero que nos permitiera ir añadiendo contenido a medida que desarrollábamos la librería modular. De esta forma, utilizamos ciertas constantes y definimos el código de forma que, si cambiáramos algo en las constantes, se añadiera directamente y se tuviera en cuenta sin necesidad de incluirlo a mano en las funciones internas del programa.

IMAT-LAB:

LIBRERÍAS

En primer lugar el programa IMATLAB importa las librerías: “io”, “os”, “sys” y “modular”. Esta programa se basa en el uso de dos clases principales, una clase denominada “Interfaz”, que contiene la estructura general del mismo, y otra que sirve para gestionar uno de los tipos de errores, “ErrorNOP”.

CONSTANTES

Al principio se definen las constantes necesarias para el correcto funcionamiento. Un par de constantes de código de strings formateados con escapes (\) que nos servirá para agregar carácter al texto por terminal.

FLAGS es un diccionario que contiene las flag que se deberán introducir por teclado, relacionadas a un identificador en forma de string que será lo que procese el programa, dando así la posibilidad de que existan varias flag para un mismo modo de ejecución.

NOP_TYPES y NOP_MESSAGES son dos diccionarios que contienen (el primero) integrales relacionados a los denominados como tipos de error y el segundo integrales relacionados a mensajes de error, así el ErrorNOP puede presentarse con varios tipos cada uno asociado a un string que lo define y con un mensaje que lo explica.

Por último tenemos el diccionario de COMANDOS. Este último tiene una estructura constante que relaciona una llave, que corresponde con la función que el usuario introduce por teclado, con una lista que contiene: la función de “modular.py” correspondiente, el tipo de objeto que se introduce en ella como parámetro, y el número de estos parámetros.

CLASES

La clase ErrorNOP sólo contiene una forma de inicialización, en la que se indicaría el tipo de ErrorNOP, una forma de impresión por pantalla (‘NOP’) y otra forma de impresión para especificar el tipo de error e imprimir su mensaje asociado.

Interfaz() representa la forma de inicialización de IMAT-LAB, de forma que para que todo funcione debe iniciarse una variable que almacene el objeto y después ejecutar el método open(). En este método se encapsula todo el proceso de generación de la interfaz. Consiste en procesar los argumentos, introducidos con el comando que abre el compilador y el fichero, y después, en función del modo de operación, iniciar las funciones de “batch” o de “iterative”. Hay un caso extra, una especie de control de errores, ya sea por defecto de no especificar el modo o introducir la flag del modo ‘help’, aparecerá un mensaje que te indicará que debes usar una flag el propio mensaje de ayuda del modo ‘help’ que especifica las posibles funcionalidades de la interfaz.

Si se inicia la interfaz con el modo ‘iterative’, se utilizará la función run_iterative() que se incluyen cómo método de la interfaz pero podría haber sido definida cómo independiente al igual que las del modo ‘batch’. Esta es la función que presenta la forma de interfaz con la que interactúa el usuario, pues la de ‘batch’ no la necesita. Imprime un título, una leyenda e inicia el bucle que: recoge la entrada del usuario; procesa la función y los parámetros y da error si son incorrectos; y devuelve una respuesta llamando a la función correspondiente de modular. Da la opción al usuario de borrar los resultados obtenidos hasta el momento, limpiando la interfaz, o de salir de forma segura usando EXIT.

De forma especial, y específicamente para el modo ‘batch’, se incluyen dos métodos encargados de abrir y cerrar los ficheros que se incluirían como argumentos del comando de inicialización.

FUNCIONES

Hay tres funciones principales en el programa. La primera, __getparams__(), dada la entrada de parámetros en forma de string del usuario, comprueba y almacena estos en una tupla ya transformados al tipo correspondiente. En esta función entra en acción el tipo de número que guarda entre otras cosas el diccionario de COMANDOS, que permite utilizar una u otra técnica para leer los parámetros dependiendo de la función.

La función function() se encarga de tomar la entrada del usuario, llamar a __getparams__() y ejecutar la función de modular.py correspondiente. También levanta errores si hay problemas con el comando o los parámetros.

Por último, la función run_comands() lee línea por línea el fichero de entrada que recibe y escribe el resultado de llamar a function() en el fichero de salida. En este punto se puede apreciar el escalonamiento entre estas tres funciones clave: run_commands() se apoya en function() que a su vez necesita de __getparams()__.

MODULAR:

La única librería implementada en el archivo es “math”, que se utiliza para el cálculo de raíces cuadradas de enteros. Antes de entrar en materia con las distintas funciones, debemos explicar las dos clases propias. ErrorNE tiene un funcionamiento similar a la anterior clase para gestión de errores, y sólo sirve para ser devuelta cómo resultado de una función en caso de que no exista solución. Pero lo más relevante es el objeto Primos. Al iniciarse el documento (al ser importado por imatlab.py) la constante PRIMOS, que inicialmente tenía un valor de None, adquiere un valor de Primos(). Esta clase posee dos atributos, una lista que se inicia con los primos 2 y 3, y una variable que almacena su longitud. Lo importante es su método self.__p__(n: int), el cual, dado un valor entero n, actualiza de lista de primos del objeto, añadiéndole primos hasta ese valor n. Se puede deducir el valor de iniciar este objeto, que servirá como memoria de los primos ya calculados hasta el momento, acelerando exponencialmente la velocidad de ejecución del código.

es_primo()

Actualiza la lista de primos hasta la raíz cuadrada del número (Teorema de la cota para divisores primos), después la recorre comprobando si el número es divisible entre alguno de ellos. Devuelve ‘No’ si es divisible entre alguno, devuelve ‘Si’ si no lo es.

lista_primos()

Devuelve ErrorNE si b <= a. Actualiza la lista de primos hasta b. mediante dos bucles y dos variables busca los índices de la lista de primos entre los que se encuentran a y b y devuelve esa parte de la lista.

factorizar()

Actualiza la lista de primos hasta la raíz del número a factorizar. Mediante un bucle se recorre la lista hasta que se llegue como máximo la raíz del número, un bucle en su interior, si el número es divisible entre el primo de la lista, divide el número entre dicho primo actualizando un contador. El contador actualizado es el exponente y el primo la base y el resultado se guarda en un diccionario que después se devuelve cómo resultado.

bezout()

Utiliza el mismo principio visto en clase y crea una especie de base con los números dados a la función y calcula los coeficientes necesarios de esa base para llegar al mcd. Obteniendo así el bezout.

mcd()

La única función recursiva de todo el código, implementada de esta forma por su elegancia y su sencillez. Basada en el algoritmo de euclides tradicional se llama a sí misma alternando el orden de los parámetros de entrada y dando el módulo del primero entre el segundo en vez de el primero hasta que el segundo sea 0.

coprimos()

Primera vez en el código que se llama a otra de las funciones. Devuelve ‘Si’ si el mcd() de los dos enteros es 1, sino devuelve ‘No’.

potencia_mod_p()

Habíamos realizado una primera versión sobre esta función, siguiendo los pasos que conocíamos de hacer el módulo de la base, … . Sin embargo, esto resultó en una función poco efectiva, lenta y sobre todo fuera de la competición con la tabla de tiempos. La solución, tras un poco de búsqueda, fue un algoritmo llamado exponenciación modular, que

encontramos en stackoverflow[1]. A este último tuvimos que añadirle bezout() para que funcionara correctamente, y decidimos integrarlo dentro de la función en vez de llamarla, lo que resultó en una ejecución mucho más rápida.

inversa_mod_p()

La función utiliza bezout para calcular la inversa módulo p del número dado. Con bezout se obtiene un número que multiplicado por el dado, por la manera en la que está construida Bezout, que es congruente con 1 módulo p.

euler()

La función factoriza el número y aplica las propiedades de euler vistas en clase para calcularlo teniendo en cuenta que el euler de ab es igual al euler de a por el euler de b y que el euler de un número primo elevado a k es igual a (p-1) multiplicado por p elevado a (k-1)

legendre()

La función calcula el símbolo de legendre realiza la potencia modulo p de n, el número introducido, elevado a la división de p-1 entre 2. Esto funciona ya que si un cierto x al cuadrado deja como residuo a el número n, entonces ese x elevado a p-1 es congruente con 1, y a su vez congruente con a elevado a p-1 entre 2. De la misma forma para los que no son residuos cuadráticos funciona puesto que serán las soluciones de x elevado a p-1 entre 2 todo ello menos 1. Esto funciona bien gracias a la propiedad de euler del símbolo de legendre.

resolver_sistema_congruencias()

En primer lugar se realiza una comprobación para ver si son coprimos todos los módulos. Se devuelve ErrorNE en caso negativo.

Un bucle cuenta el número de ecuaciones de congruencias, y almacena en m el resultado de multiplicar todos los primos.

Las ecuaciones en congruencias son de la forma: $ax \equiv b \pmod{p}$. Por tanto, podemos asumir, para cada ecuación, un c tal que: c = inversa_mod_p(a,p)*b. De tal forma que la almacenamos, siguiendo el Teorema Chino del Resto, el sumatorio de los c*inversa_mod_p(n,p)*n, con n=m/p, para cada c y p de cada ecuación, y el resultado será ese sumatorio módulo m.

raiz()

Habíamos intentado realizar la función mediante el algoritmo de Cipolla pero finalmente debido a una mayor sencillez a la hora de comprender el algoritmo de Tonelli y

Shanks nos decantamos por esa opción. Este algoritmo lo encontramos buscando alternativas para la obtención de una raíz módulo p, en una pagina web, Algoritmo de Tonelli-Shanks Ideas centralesyEl algoritmo (hmong.es)[2].

ecuacion_cuadratica()

Despejando de la fórmula de la ecuación cuadrática se obtiene una raíz módulo p que se calcula con la función anterior y a la solución hay que restarle b/2a siendo a el parámetro que multiplica a x^2 y b el parámetro que multiplica a x.

BIBLIOGRAFÍA:

[1] https://stackoverflow.com/questions/57668289/implement-the-function-fast-modular-exponentiation

[2] https://hmong.es/wiki/Tonelli–Shanks_algorithm