ALEJANDRO_SANTORUM-labot-ex3-2015(EJERCICIO)

March 4, 2018

CALIFICACIÓN:

1) Por favor, empieza cambiando *nombre.apellido* en el nombre de esta hoja por los tuyos, tal como aparecen en tu dirección de correo eletrónico de la UAM, y deja el resto del nombre del archivo como está.

2) Recuerda que debes entregar esta hoja de SAGE, con tus respuestas, en la carpeta $ENTREGA\_examen\_nombre.apellido - labot - ex3$ que está en el escritorio de la cuenta que utilizas para el examen. Para guardar la hoja, una vez terminado el examen, utiliza el menú "FILE>Save worksheet to a File...." dentro de la hoja. Comprueba que la hoja que has guardado en la carpeta de entrega se llama "nombre.apellido-labot-ex3.sws", con nombre.apellido los tuyos.

3) En principio es posible resolver un apartado sin haber resuelto apartados previos de los que depende. Quizá no podrás comprobar que el código funciona bien, pero cada apartado se valorará por sí mismo.

4) Para elevar un entero a a un exponente n módulo otro entero p conviene usar la función de SAGE $power\_mod(a, n, p)$, que realiza el cálculo de manera eficiente.

Ejercicio 1

Sea p un número primo. Las clases de restos no nulas módulo p se pueden multiplicar entre ellas y decimos que $g \in \{1, 2, \ldots, p-1\}$ es un generador si todas las potencias de g, módulo p, con exponente en $srange(1, p)$ son distintas entre sí. Podríamos buscar un generador mediante fuerza bruta probando con los posibles generadores hasta que encontráramos uno que cumpliera la condición. Sin embargo, hay una manera de reducir esta búsqueda:

Factoriza el entero $p-1$, y sean $p_1, p_2, \ldots, p_k$ sus factores primos.

Para cada posible generador, $a \in \{2, 3, \ldots, p-1\}$ calcula las potencias de a módulo p con exponentes $(p-1)/p_i$ mientras no se obtiene una potencia igual a 1. Si se obtiene una potencia igual a 1 el elemento a no es un generador y hay que probar con otro valor de a.

Si se obtienen k potencias todas diferentes de 1 se demuestra (no hay que demostrarlo) que a es un generador.

(A) (1 punto) Programa,usando la búsqueda reducida, una función $generador0(p)$ que devuelva el generador módulo p más pequeño.

(B) (1 punto) Modifica la función anterior, para obtener otra a la que debes llamar $generador(p)$, que devuelva un generador encontrado aleatoriamente.

(C) (1 punto) Escribe una función $comprobar(g, p)$ que devuelva $True $ si g verifica la definición de generador módulo p, y $False$ si no la verifica. Utiliza esta función para comprobar que el entero que devuelve $generador(nth\_prime(33333))$ es realmente un generador módulo $nth\_prime(33333)$.

```
In [20]: def generador0(p):
             if not is_prime(p): #comprobacion de errrores
```

```
        print 'Error, p no es primo'
        return

    if p==2: #caso base especial
        return 1

    LF = list(factor(p-1)) #resto del algoritmo
    l = len(LF)
    flag = 1
    for a in xsrange(2, p):
        flag = 1
        for k in xsrange(0, l):
            ex = ((p-1)//LF[k][0])
            if power_mod(a, ex, p) == 1:
                flag=0
                break
        if flag == 1:
            return a
    return -1
```

In [21]:
```
generador0(6)
```

```
Error, p no es primo
```

In [22]:
```
for k in xsrange(2, 50):
    if is_prime(k):
        print (k, generador0(k))
```

```
(2, 1)
(3, 2)
(5, 2)
(7, 3)
(11, 2)
(13, 2)
(17, 3)
(19, 2)
(23, 5)
(29, 2)
(31, 3)
(37, 2)
(41, 6)
(43, 3)
(47, 5)
```

In [23]:
```
def generador(p):
    if not is_prime(p): #comprobacion de errrores
        print 'Error, p no es primo'
```

```
            return -1

        if p==2: #caso base especial
            return 1

        LF = list(factor(p-1)) #resto del algoritmo
        l = len(LF)
        flag = 1
        while(1):
            a = randint(1, p-1)
            flag = 1
            for k in xsrange(0, l):
                ex = ((p-1)//LF[k][0])
                if power_mod(a, ex, p) == 1:
                    flag=0
                    break
            if flag == 1:
                return a
        return -1
```

```
In [24]: for k in xsrange(2, 50):
             if is_prime(k):
                 print (k, generador(k))
```

```
(2, 1)
(3, 2)
(5, 3)
(7, 3)
(11, 8)
(13, 6)
(17, 14)
(19, 14)
(23, 11)
(29, 26)
(31, 17)
(37, 24)
(41, 35)
(43, 34)
(47, 19)
```

```
In [25]: def comprobar(g, p):
             L = list()
             for ex in xsrange(1, p):
                 a = power_mod(g, ex, p)
                 if a in L:
                     return False
                 else:
```

```
                L.append(a)
        return True
```

```
In [26]: for k in xsrange(1, 11):
             print(k, comprobar(k, 11))
```

```
(1, False)
(2, True)
(3, False)
(4, False)
(5, False)
(6, True)
(7, True)
(8, True)
(9, False)
(10, False)
```

```
In [27]: a = nth_prime(33333)
         a
```

```
Out[27]: 393191
```

```
In [14]: g = generador(a)
         g
```

```
Out[14]: 387373
```

```
In [ ]: comprobar(g, a)
```

No acaba en la vida porque puede estar repitiendo muchos números debido a la elección de aleatorio.

Ejercicio 2

Un sistema de intercambio de claves permite a dos usuarios A y B comunicarse claves de manera segura de forma que ambos dispongan de la misma clave. Uno de los existentes funciona de la siguiente manera:

Los usuarios eligen un primo p muy grande y un generador g módulo p . No hay ningun problema en que p y g se transmitan sin encriptar.

Cada usuario elige una clave privada (e_A es la clave privada de A, e_B es la clave privada de B), que mantendrán en secreto, y transmite al otro usuario g elevado a su clave privada módulo p. Las claves privadas son mayores que 1 y menores que $p - 1$ y se eligen aleatoriamente.

Cada usuario, al recibir el entero transmitido en el punto 2 lo eleva a su propia clave privada módulo p y el resultado K es la clave común. Es claro que

$$(g^{e_A})^{e_B} = (g^{e_B})^{e_A}$$

de forma que la clave es realmente la misma.

La seguridad del sistema reside en que conociendo g, p y g^{e_A} no es posible, en un tiempo razonable, obtener el exponente secreto e_A. Se conoce este problema como el del cálculo del logaritmo discreto, y no se conoce una solución.

(D) (2 puntos) Vamos a utilizar como primo $p = next\_prime(26^{128})$. Define una función $clavesA()$ que devuelva un g calculado usando $generador(p)$, la clave privada e_A de A y la potencia g^{e_A} módulo p. De forma similar define una función $clavesB(g)$, que devuelva la clave privada de B y la potencia g^{e_B} módulo p usando el generador g que recibe como argumento (el usuario B no calcula un generador porque debe usar el mismo que A).

(E) (1 punto) Define una función $clave()$ que, utilizando las dos anteriores, devuelva la clave común K.

```
In [1]: p = next_prime(26^128)
        p

Out[1]: 130794216386320785386099858867605235749262232604493153327801416131094487558359727671668

In [2]: def clavesA(p):
            g = generador(p)
            Ea = randint(1, p-1)
            pot = power_mod(g, Ea, p)
            return g, Ea, pot

In [3]: def clavesB(g, p):
            Eb = randint(1, p-1)
            pot = power_mod(g, Eb, p)
            return Eb, pot

In [70]: def clave(p):
             g, Ea, potA = clavesA(p)
             Eb, potB = clavesB(g, p)
             K1 = power_mod(potA, Eb, p)
             K2 = power_mod(potB, Ea, p)
             if K1 != K2:
                 print("Error. Las claves no coinciden.\n")
                 return -1
             return K1

In [67]: A = clavesA(p)
         print("g=generador de A: "+str(A[0])+"\n")
         print("Ea=clave privada de A: "+str(A[1])+"\n")
         print("potA=potencia g^Ea mod p: "+str(A[2]))

g=generador de A: 63637416322426126421731347134160044512266881218808929512494581516765720188341

Ea=clave privada de A: 77386970903720333476966626555921856714667362244534036235369429272350761

potA=potencia g^Ea mod p: 34983766664620303892687350448220703333019774592826123970959853968403


In [68]: B = clavesB(A[0], p)
         print("Eb=clave privada de B: "+str(B[0])+"\n")
         print("potB=potencia g^Eb mop p: "+str(B[1]))
```

```
Eb=clave privada de B: 51013386979274945435689479866026185075154888508942610404741377934767829

potB=potencia g^Eb mop p: 173746757220541114152671123472597775675561317021140751119801137752708
```

```
In [72]: K = clave(p) #OBSERVACION, la clave no se ha calculado con los numeros anteriores
                      # ya que en la funcion clave(p) se calculan de nuevo aleatoriamente las
         print K
```

```
161352280997167228961642124911276946009311107824380022672691423650782847136800399171225684787
```

Ejercicio 3

Una permutación de la lista $[0,1,2,\ldots,25]$ es una reordenación de los elementos de la lista en otra con los mismos elementos en distinto orden. Si llamamos L a la lista reordenada, podemos ver la permutación como una función biyectiva σ definida por

$$\sigma(i) := L[i].$$

Dos usuarios, A y B, se intercambian claves según el procedimiento del ejercicio anterior y las van a usar de la siguiente manera:

Dada la clave común K la escriben en el sistema de numeración de base 26 en forma de una lista L de enteros ≥ 0 y ≤ 25. Manteniendo el orden suprime las repeticiones en L , de forma que ontengas una lista $L1$ sin repeticiones.

Si la longitud de la nueva lista $L1$, una vez suprimidas las repeticiones, es 26, la lista es una permutación σ de $srange(26)$ y la podemos usar como clave en el sistema (cifrado de permutación) en que encriptamos la letra que ocupa el lugar i-ésimo en el alfabeto mediante la letra que ocupa el lugar $\sigma(i)$ en el alfabeto.

Si la longitud de la nueva lista $L1$, una vez suprimidas las repeticiones, es menor que 26 (esto es muy poco probable), no hemos obtenido una permutación y volvemos a generar una clave común.

(F) (2 puntos) Define una función $claveperm(K)$ que implemente el sistema descrito en los puntos 1), 2) y 3) de este ejercicio.

(G) (1 punto) Encripta el texto suministrado más abajo usando una clave común K generada usando el Ejercicio 2 y la función $claveperm(K)$.

(H) (1 punto) Desencripta el texto encriptado para obtener otra vez el texto original.

```
In [8]: alfb = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

```
In [9]: L_alfb = list(alfb)
```

```
In [10]: texto='THROUGHTHEUSEOFABSTRACTIONANDLOGICALREASONINGMATHEMATICSDEVELOPED\
         FROMCOUNTINGCALCULATIONMEASUREMENTANDTHESYSTEMATICSTUDYOFTHESHAPESANDMOT\
         IONSOFPHYSICALOBJECTSPRACTICALMATHEMATICSHASBEENAHUMANACTIVITYFORASFARBA\
         CKASWRITTENRECORDSEXISTRIGOROUSARGUMENTSFIRSTAPPEAREDINGREEKMATHEMATICSM\
         OSTNOTABLYINEUCLIDSELEMENTSMATHEMATICSDEVELOPEDATARELATIVELYSLOWPACEUNTI\
         LTHERENAISSANCEWHENMATHEMATICALINNOVATIONSINTERACTINGWITHNEWSCIENTIFICDI\
         SCOVERIESLEDTOARAPIDINCREASEINTHERATEOFMATHEMATICALDISCOVERYTHATCONTINUE\
         STOTHEPRESENTDAY'
```

In [11]:
```
def ord2(c):
    return L_alfb.index(c)
```

In [12]:
```
def chr2(n):
    return L_alfb[n]
```

In [74]:
```
K = clave(p)
K
```

Out[74]: 97907963308266624416432139228883744654469043662453785158173696839406036287194906218804

In [89]:
```
def clavePerm(K):
    L = list(K.digits(base=26))
    while(1):
        L1 = []
        for elem in L:
            if elem in L1:
                continue
            else:
                L1.append(elem)
        if len(L1) == 26:
            return L1
```

In [90]:
```
S = clavePerm(K)
print S
```

```
[24, 5, 13, 19, 10, 1, 8, 22, 12, 17, 14, 6, 20, 18, 3, 15, 2, 0, 7, 25, 23, 16, 9, 21, 11, 4]
```

In [93]:
```
def encriptacion(texto, S):
    LT = list(texto)
    for j in xsrange(0, len(LT)):
        indice = ord2(LT[j])
        nuevoIndice = S[indice]
        LT[j] = chr2(nuevoIndice)
    cad = "".join(LT)
    return cad
```

In [95]:
```
cad = encriptacion(texto, S)
```

In [104]:
```
def desencriptacion(texto, S):
    LT = list(texto)
    for j in xsrange(0, len(LT)):
        indice = ord2(LT[j])
        nuevoIndice = S.index(indice)
        LT[j] = chr2(nuevoIndice)
    cad = "".join(LT)
    return cad
```

In [105]:
```
desencriptacion(cad, S)
```

Out[105]: 'THROUGHTHEUSEOFABSTRACTIONANDLOGICALREASONINGMATHEMATICSDEVELOPEDFROMCOUNTINGCALCUL

In [ ]: