Diseño y Análisis de Algoritmos

Práctica 3

Merkle-Hellman | Programación dinámica.
Cifrado. Dando cambio. Subsecuencias. BST óptimos.

Andrián Navas Ajenjo
Gloria del Valle Cano
adrian.navas@estudiante.uam.es & gloria.valle@estudiante.uam.es

Grado en Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid

Índice general

1. Cuestiones

Cuestiones sobre el cifrado Merkle-Hellmann

Cuestión 1

Dado el posible tamaño de los términos de la sucesión supercreciente, es necesario trabajar con enteros de tamaño adecuado. Averiguar el tamaño máximo de un entero en Python.
En la versión 3 de Python *[Link]*, se eliminó la limitación máxima para números enteros, y a efectos prácticos, esta no existe. En la versión 2 de Python, este límite máximo sí que existía, y se podía obtener ejecutando:

```
python
>>> sys.maxint
9223372036854775807
```

Y como se puede observar, este máximo era 9223372036854775807. Esta función ha sido deshabilitada en la versión 3, pero en caso de necesitar este número por cuestiones de compatibilidad entre versiones o migraciones a la versión 3, se puede obtener con el comando:

```
python
>>> sys.maxsize
9223372036854775807
```

Cuestión 2

Un elemento importante en el algoritmo Merkle–Hellman es la longitud de las sucesiones empleadas, lo que a su vez influye en el valor máximo de sucesión supercreciente y el módulo. Si dicha sucesión tiene N terminos, estimar los valores mínimos del ultimo término de una sucesión supercreciente de N terminos y del módulo. Sugerencia: considerar el ejemplo de la sucesion
s_n `= 2`^{n}`, n = 0, 1, 2,...`
Contando con que la sucesión sigue una distribución 2^N para un N arbitrario, el límite mínimo del último término obviamente será $2^{(N-1)}$. Estudiando cada uno de los números, podemos observar que cada número, al ser potencia de 2 representa un bit nuevo en una representación binaria (ejemplo: `N=3` $\longrightarrow$ S_n `= [b1, b10, b100]`). Teniendo esto en cuenta, el módulo, cuyo mínimo debería ser la suma de todos los números existentes en la lista, lo que tendrá como mínimo un número cuya representación binaria tiene *N números uno + 1*, lo cual es igual a 2^N.

Cuestión 3

A la vista de las dos cuestiones previas, discutir cual puede ser la longitud máxima razonable de la sucesión supercreciente.

Considerando la capacidad de cómputo actual, en la cual las recomendaciones para algoritmos de clave pública como RSA son utilizar claves de 4096 bits, la longitud de lista recomendada es la cual hace que el módulo tenga 4096 bits. Dicha longitud es 12 (lo que es igual a `log2(4096)`) ya que de esa forma el módulo tendrá como mínimo 4096 bits.

Cuestión 4

Un enfoque trivial para la función `inverse(p, mod)` es probar con enteros de manera iterada hasta encontrar un q tal que `p * q% mod == 1`. Sin embargo, esto es muy costoso computacionalmente y se puede mejorar mediante una variante del algoritmo de Euclides. Describir aquí dicha variante y estimar su coste computacional. En nuestro caso, hemos aplicado el algoritmo de Euclides extendido para obtener el inverso multiplicativo en la práctica. Este algoritmo utiliza el algoritmo de Euclides de forma secuencial con un número entero y un módulo hasta tener la siguiente fórmula:

$$1 = u * mod + v * p\,\%(mod) \tag{1.1}$$

Donde v es el inverso multiplicativo de p en *Zmod*. Teniendo en cuenta que cada iteración tiene un coste de O(1) y que calculamos tantas iteraciones como O(log N) (porque en cada iteración se le aplica el módulo al número siguiente), entonces el coste computacional del algoritmo de Euclides es de O(log N).

Cuestiones sobre Programación Dinámica

Cuestión 1

Estimar en detalle el coste computacional del algoritmo usado en la funcion `optimal_order(l_probs)`.:
Observando el código del algoritmo desarrollado observamos que el bucle principal del algoritmo (obviando la inicialización de la diagonal) es un bucle el cual recorre para cada nodo, todos los nodos por lo que el coste computacional de optimal order es O($N * N$) lo que es igual a O(N^2). Pero en la práctica, y si cogemos la función $m[j, j + i] = get\_min\_tree(m, j, j + i)$ como la función base de nuestro algoritmo, podemos observar que el coste real del algoritmo es de O($(N^2)/2$) ya que solo necesitamos rellenar la parte superior de la matriz para obtener el resultado final.

Cuestión 2

El problema de encontrar la maxima subsecuencia común (no consecutiva) a veces se confunde con el de encontrar la máxima subcadena (consecutiva) común. Ver por ejemplo la entrada Longest common substring problem en Wikipedia. Describir un algoritmo de programación dinámica para encontrar dicha subcadena común máxima y aplicarlo "a mano" a las cadenas de los primeros apellidos de los miembros de tu pareja de practicas.

Hemos visto que existe una confusión habitual con el problema de la subcadena y el de la subsecuencia. Entendemos que la subcadena es aquella que se encuentra entre dos strings con caracteres consecutivos, mientras que para el caso de la subsecuencia éstos no tienen por qué estar consecutivos. Una vez obtenido el problema de la subsecuencia, nos ha resultado más intuitivo modificar el código para detectar el problema de la subcadena. La lógica que sigue el algoritmo sigue partiendo de la misma matriz formada por las dos cadenas. Seguimos la siguiente lógica:

- Cada vez que hay una coincidencia, se suma 1 en la posición de la matriz correspondiente.
- Si los últimos caracteres coinciden, reducimos ambas longitudes en 1.
- Si los últimos caracteres no coinciden, el resultado es 0.
- El sufijo común más largo es la subcadena común más larga, es decir, m(str_1, str_2) = max(m(str_1, str_2)) donde i>0 y j>0.

Finalmente, esta sería la lógica seguida para implementarlo en Python.

```
#Cuestión 2: Longest common substring problem
def max_matrix_common_substring(str_1, str_2):
    l1, l2 = len(str_1), len(str_2)
    m = [[0 for k in range(l2+1)] for l in range(l1+1)]
    length = 0
    for i in range(l1+1):
        for j in range(l2+1):
            if (i == 0 or j == 0):
                m[i][j] = 0
            elif (str_1[i-1] == str_2[j-1]):
                m[i][j] = m[i-1][j-1] + 1
                length = max(length, m[i][j])
```

```
            else:
                m[i][j] = 0
    return m, length

def max_length_common_substring(str_1, str_2):
    i, j = len(str_1), len(str_2)
    m, length = max_matrix_common_substring(str_1, str_2)
    print(m)
    return length

def find_max_common_substring(str_1, str_2):
    j, i = len(str_2), len(str_1)
    m, length = max_matrix_common_substring(str_1, str_2)
    ind = max_length_common_substring(str_1, str_2)
    cad = [""]*ind
    while (m[i][j]!=0):
        length -= 1
        cad[length] = str_1[i-1]
        i -= 1
        j -= 1
    print(cad)
```

La salida de nuestro algoritmo es la esperada:

```
>> find_max_common_substring("navasajenjo", "delvallecano")
[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
[0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0],
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 1, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]]
>> ['v', 'a']
```

Ese 2 es el máximo de la matriz, que viene de la diagonal y que se ha acumulado por estar consecutivo.