Trabajo Integridad: Búsqueda de colisiones

Silviu Valentin Manolescu – MUCC

Contenido

1. Tablas Arco Iris

1.1 Implementación base

La implementación base de Arco Iris se ha realizado empleando la función CRC32 a través de la librería de Python zlib. Esta función devuelve un entero que bien se puede trabajar con el directamente o en ciertas situaciones transformarlo en hexadecimal, pero para la mayoría de las veces se ha empleado la primera alternativa.

En cuanto al espacio de caracteres, como más adelante se mostrará, se irá variando para ver el impacto que este tiene en los resultados. Principalmente se trabajará con un espacio compuesto de números de 0 a 9 o con letras minúsculas para las pruebas extendidas.

Por otra parte, para las dimensiones de la tabla arco iris creada también se irá variando con tal ver su impacto en los resultados del porcentaje de éxitos en cada prueba.

Finalmente, en cuanto a la implementación del código, que se encuentra en el apéndice, se han seguido las indicaciones de las diapositivas a la hora de establecer la estructura general del programa:

- Función **hash** "**h**": devuelve el resumen CRC32 en formato entero.
- Función **recodificante** "**r**": devuelve un nuevo entero (caso del espacio de caracteres numéricos) o cadena String (caso del espacio de caracteres alfabéticos). Por lo tanto, esta función contará con varias versiones según el apartado.
- Función "**generateRainbowTable**": se encarga de generar toda la tabla arco iris y al final la guarda en un archivo con formato. pkl propio de la librería Pickle de Python, ideal para diccionarios.
- Función "**searchColision**": dado el diccionario de la función anterior busca la colisión devolviendo si existe la contraseña alternativa equivalente.
- Función "**mainArcoIris**": se trata de una función auxiliar para hacer varias llamadas del algoritmo Arco Iris y generar archivos de output que guardan estadísticas del proceso. Se emplea para facilitar el análisis.

Para programar todo este código anterior se ha escogido Python como lenguaje y Visual Studio Code como editor.

1.1.1 Experimentación base

Las especificaciones del equipo empleado en las pruebas se corresponden con un Portátil Lenovo Legión que cuenta con las siguientes características:

Procesador: AMD Ryzen 7 5800H (8 núcleos, N.º de subprocesos 16, Frecuencia Base 3.2 GHz)

Tarjeta Gráfica: Nvidia RTX 3060

Almacenamiento SSD: WDC PC SN730 SDBPNTY-1T00-1101 (1 Tb)

RAM: 16 GB DDR4 (3200 MHz)

Sistema Operativo: Windows 11

Para la experimentación base se va a trabajar con un espacio de caracteres numéricos de 0 al 9 con una longitud de 6. El código se encuentra en el apéndice.

Estas primeras pruebas se definen 5 contraseñas en una lista ['523824', '941167', '782749', '481939', '736342'] de forma aleatoria manualmente. Esta lista después se le aplica la función hash para convertir cada contraseña en p0 y simular el estado inicial del problema, donde lo único que se conoce de las contraseñas reales es su hash.

En este caso las dimensiones de la tabla se han fijado en 5000 filas y 200 columnas. El motivo de esta elección es que se cubre todo el dominio de posibilidades debido a que al tener 10 posibles caracteres y con una longitud de contraseña de 6 se tienen 1 000 000 de combinaciones posibles. Gracias a esto se tiene una matriz que es capaz de guardar todas esas posibles combinaciones. Otras elecciones posibles de tamaños pueden ser 10 000 filas con 100 columnas, 2 500 filas con 400 filas, 20 000 filas con 50 columnas… Sin embargo, a pesar de que se cubra todo el espacio no significa que vayan a dar resultados similares. Este aspecto se tratará en la experimentación extendida.

Respecto a la función resumen empleada en el caso de los caracteres numéricos se ha usado el módulo de 1 000 000 aplicado al resumen proporcionado por el CRC32 para convertirlo en 6 números únicamente. En el último paso se convierte a cadena String y se devuelve como resultado. Se ha intentado emplear una función recodificante alternativa agregando un índice que dependía de la columna en la que se encontraba y multiplicando así el valor de CRC32, sin embargo, no se ha obtenido ningún resultado satisfactorio. (ninguna equivalencia)

Los resultados obtenidos son los siguientes:

% Colisiones	% Éxito	Tiempo medio de búsqueda (s)
100%	20%	0.49851

A partir de la tabla anterior se puede observar que se ha obtenido una colisión para todas las 5 contraseñas, pero solo 1 de ellas tiene una equivalencia. Los resultados han sido constantes, es decir para las 10 ejecuciones se han mantenido estos valores, sin embargo, en realidad el proceso que ha llevado al resultado no ha sido el mismo. Para verlo de forma más clara se introduce esta nueva tabla con las 3 primeras ejecuciones del programa:

Número	Real	Equivalente	Colisión Inicial
Ejecución 1	523824		254615
	941167		967851
	782749		974013
	481939		163220
	736342	736342	610063
Ejecución 2	523824		159629
	941167		965141
	782749		232989
	481939		899873
	736342	736342	703768
Ejecución 3	523824		580996
	941167		134339
	782749		523619
	481939		468769
	736342	736342	055741

La contraseña real “736342” en todas las ejecuciones ha sido descubierta, no obstante, nunca por el mismo camino. La colisión inicial o punto de anclaje en la tabla arco iris ha ido cambiando no solo para esta contraseña, sino también para todas las demás que a pesar de tener otro valor no se ha podido proporcionar una equivalencia, fijando la tasa de éxito al 20%. El motivo de este comportamiento se debe a que la tabla se crea mediante un conjunto fijo de contraseñas y una función hash específica que no van a cambiar a menos de que se modifiquen los parámetros.

Del código empleado también podría resultar interesante cambiar el último loop while en su condición de parada. Es decir, una vez se produce una colisión se entra dentro de un bucle que no va a acabar a menos de que se cumpla: h(pwd) != p0 . Sin embargo, esto no tiene que ocurrir siempre y en caso de que sea así se podría permanecer un tiempo muy grande dentro de este loop, como en el caso de la contraseña “523824”. Por lo tanto, **¿cuánto tiempo se debería seguir calculando r(h(pwd))?** Realmente, este suceso no se puede saber, podría ser en 3 iteraciones como en 9999999 o más… Desde luego hay que tener en cuenta que cuanto antes se sale del bucle más tiempo se puede ahorrar de cálculo. Existen varias alternativas como condición adicional de cara a este problema, estas son algunas de ellas:

- **Iteración actual < Número Máximo Iteraciones:** en este caso para las pruebas se ha empleado n*t como número máximo de iteraciones arrojando un tiempo medio de 0.49851 segundos. Se han hecho otras pruebas con 5*t, proporcionando un tiempo medio 0.0006 de búsqueda en la tabla arco iris. En ambos casos se han obtenido los mismos resultados de porcentaje de éxito. Incluso poniendo solamente t se acaba teniendo el mismo resultado de éxito y porcentaje de colisiones. De ahora en adelante gracias a este estudio se va a dejar en 5*t, ya que se mantiene una leve esperanza de que pueda ocurrir en ese rango y tampoco compromete mucho el tiempo total.

- **Contador de Tiempo:** otras alternativas que se han intentado ha sido elegir 5 segundos para cada bucle, dejando el tiempo medio en 4 segundos. Este valor se debe a que de las 5 contraseñas la única que tiene equivalencia sale mucho antes del bucle al cumplirse la condición h(pwd) != p0, reduciendo así la media. No se han conseguido mejores resultados, el tiempo total de ejecución por su parte ha empeorado mucho.

Finalmente, el último tema a tratar de este capítulo es el tiempo de generación de las tablas arco iris. Cabe destacar que este hecho está intrínsecamente relacionado con el tamaño de la tabla, la función recodificante y espacio de caracteres:

Tamaño	Tiempo (s)
1 000 x 100	0.036
1 000 x 300	0.123
3 000 x 100	0.124
3 000 x 300	0.530
5 000 x 100	0.232
5 000 x 300	1.700
7 200 x 300	66,552
20 000 x 300	¿?
100 000 x 2	8.435

Obviamente se puede notar que cuanto más grande es una tabla más tiempo tarda en generarse. Sin embargo, ocurre un hecho interesante al llegar a tamaños mayores como el de 7200x300 de tabla con bastantes columnas, el tiempo crece drásticamente. Este hecho se debe a que en la tabla arco iris cuando se quiere introducir una nueva entrada tiene una colisión con otro valor que tiene el mismo hash y no crece en tamaño. La función recodificante también influye en este hecho, si se cubre poco dominio se

acabarán generando las mismas cadenas más a menudo que acaban teniendo el mismo hash. Por ejemplo, para el valor 20 000x300 se tardaría mucho tiempo en poder llegar a completarlo, ya que acabaría teniendo muchas colisiones de hashes al hacer tantas recodificaciones. Por otra parte, si el espacio de caracteres es reducido, no hay muchas cadenas que la función recodificante pueda generar y se termina sin poder aumentar mucho el tamaño. Por estos motivos se concluye que mantener un número más bajo de columnas es mejor a la hora de generar la tabla, al igual que hay que tener en cuenta la cantidad de posibles combinaciones del espacio de caracteres según la longitud.

A través de esta conclusión puede surgir una nueva pregunta: **¿Entonces si pongo muy muy pocas columnas sería mucho mejor, ya que se tarda menos tiempo en generar la tabla, cierto?** Se trata de un error, puesto que lo único que se provocaría es que la tabla acabase ocupando mucha memoria. Se eliminaría así la principal ventaja de las tablas arco iris, donde se guarda solo el principio y el final, pero no las operaciones intermedias, ocupando de esta forma gran cantidad de memoria. En este caso el espacio de caracteres y la longitud es reducida, no teniendo gran impacto, pero para las contraseñas que cuentan con mayor seguridad sería contraproducente hacer una tabla tan profunda. Debe haber un balance entre filas y columnas, pero este balance es diferente para cada problema siendo el programador quien deba averiguarlo para proteger la memoria sin comprometer tampoco el tiempo de cómputo de la búsqueda. (a mayor número de columnas mayor es el tiempo de búsqueda)

1.2 Experimentación extendida

Para la experimentación extendida de este apartado se va a trabajar con el espacio de caracteres alfabéticos de longitud 5: “abcdefghijklmnopqrstuvwxyz”. El código se encuentra en el apéndice.

Con tal de seguir con este capítulo se van a cambiar algunas cosas del programa. En primer lugar, ahora se pasa a trabajar con una base de datos de 100 contraseñas que se encuentra en una lista y se calcula directamente su resumen.Estas son las contraseñas originales:

['qigvh', 'tipsh', 'buvmx', 'qvydt', 'pkvjb', 'fiulf', 'wunvo', 'cmycf', 'yaltu', 'jufbh', 'nuloa', 'kqbxr', 'hwnrv', 'odvfd', 'zrypq', 'nxidl', 'uckmh', 'tqrjk', 'figfk', 'ewhqd', 'boufa', 'yvxeq', 'jexrh', 'wkkme', 'ulliq', 'lzrmp', 'zebhi', 'jezia', 'pjscs', 'qgatm', 'hsmwd', 'zcwdo', 'dmzym', 'mnobj', 'jwtrh', 'hiewd', 'obzee', 'phhsk', 'eduin', 'eklin', 'ssdkq', 'ugsti', 'wupcl', 'vhbyu', 'quubj', 'guvgu', 'fvgzm', 'exzys', 'rovcn', 'xossw', 'bxydy', 'xodbb', 'tzkro', 'oukdc', 'jcwdj', 'aqjcq', 'zntny', 'rzdrl', 'owgbu', 'nbkao', 'ygncc', 'kjvnl', 'pddhc', 'ilxfp', 'vqtva', 'fhvjd', 'whpdu', 'ibrvo', 'efqid', 'vjaof', 'igpyh', 'hbyzp', 'qrwah', 'uzued', 'ghuli', 'axawj', 'xdmlq', 'tudjd', 'bkovu', 'waree', 'snrug', 'dxakb', 'fqztp', 'ozjgq', 'nbkym', 'gdrgu', 'glfkb', 'kldpy', 'ndvro', 'bbcpo', 'uxpne', 'crmcn', 'pzjdv', 'sdpjj', 'zgxru', 'ldtyi', 'ztbdt', 'sfyic', 'psebu', 'klhbm']

En cuanto a la función recodificante en estas primeras pruebas consiste en:

1. Coger grupos de 6 bits
    1.1. Aplicar modulo espacio de caracteres (26)
    1.2. Coger la letra en la posición del anterior cálculo
2. Unir las letras

Adicionalmente, a raíz del apartado anterior se desea medir cómo evoluciona el programa según el número de filas y columnas que se introduzcan, por ello se realizarán varios bucles para ejecutar el programa según parámetros distintos. Esta tabla recoge dicho comportamiento del % de éxito:

Porcentaje de éxito (%)[1]						
	Columnas					
Filas	20	50	100	200	300	400
1 000	0	1	2	3	3	1
2 000	1	3	3	4	2	3
3 000	1	2	2	3	4	2
5 000	0	2	4	3	3	3
10 000	2	4	5	4	4	3
20 000	4	4	6	5	3	3
30 000	6	6	6	4	4	3
50 000	7	7	6	4	4	3
60 000	7	9	6	4	-	-
70 000	7	9	6	4	-	-
80 000	9	10	6	4	-	-
90 000	6	9	6	4	-	-
100 000	7	10	6	4	-	-

En el espacio de caracteres que se está manejando hay 26 tipos diferentes para una longitud de 5 y origina casi 12 millones de posibilidades. Es comprensible el motivo por el cual por debajo de 30 000 filas apenas hay éxito, dado que no se cubren todas las posibilidades del dominio. Otro aspecto interesante de los resultados es que por encima de 200 columnas hay bastantes menos coincidencias y con todo esto queda claro que para estos cálculos lo mejor será trabajar con menos de 100. Una posible explicación de este motivo es a causa de las colisiones internas que puedan ocurrir a la hora de generar las tablas, ocasionando que se pierdan cadenas al tener demasiadas columnas.

Por otra parte, los valores de filas de 60 000 a 100 000 con solo 50 columnas se tienen los mejores resultados de porcentaje de éxito. En cuanto a uno de los mayores valores, de 100 000 x 50 se han descubierto las siguientes contraseñas:

Original	Equivalente	Colisión inicial
yaltu	yaltu	uuvqi
jexrh	jexrh	nzasp
hiewd	hiewd	gmyqh
guvgu	guvgu	ndanr
ilxfp	ilxfp	twhev
efqid	efqid	wcmby
hbyzp	hbyzp	jrnyp
xdmlq	xdmlq	dlcpc
sdpjj	sdpjj	wgkya
sfyic	sfyic	vjjqd

Asimismo, al generar las estadísticas para el porcentaje de colisiones, es decir, porcentaje de veces que se ha podido anclar a un valor de la tabla durante la búsqueda del valor equivalente se ha visto en parte las explicaciones anteriores. Si la tabla es demasiado pequeña es muy improbable llegar a un número elevado de colisiones y dificultando la posibilidad de encontrar una equivalencia, especialmente para muy pocas filas o muy pocas columnas (marcadas en rojo). Se cubre tan poco dominio que los resultados son precarios. La tabla en cuestión es:

[1] Las columnas de 300 y 400 para los valores de filas más elevados no se han llegado a calcular a causa de un gran incremento en el tiempo de procesamiento.

Porcentaje colisiones (%)						
	Columnas					
Filas	20	50	100	200	300	400
1 000	2	17	33	74	84	89
2 000	4	31	55	88	88	95
3 000	3	31	68	91	95	97
5 000	7	48	74	92	96	98
10 000	19	69	94	97	97	99
20 000	34	81	96	99	98	99
30 000	43	90	98	100	100	99
50 000	55	89	100	99	100	100
60 000	56	97	99	100	-	-
70 000	72	97	100	100	-	-
80 000	71	95	100	100	-	-
90 000	74	96	99	100	-	-
100 000	73	99	99	100	-	-

Lo que no se le ha dado importancia en este capítulo ha sido el consumo de memoria primaria y secundaria. El espacio consumido en ambos casos ha sido muy pequeño, en parte como consecuencia de no crear tablas demasiado grandes o emplear un espacio de caracteres un tanto reducido. Las tablas arco iris generadas no han ocupado más allá de varios KB en memoria secundaria. (formato .pkl)

Funciones recodificantes alternativas

A lo largo de este proyecto ha habido muchos más intentos previos de conseguir una función que brindase unos resultados mejores. Uno de los primeros intentos era aplicar puertas AND, OR y XOR. De cara a los 2 primeros tipos resultaron ser muy ineficientes debido a que generaban demasiados ceros o unos en las cadenas, haciendo imposible cubrir un largo dominio de posibilidades y provocando que las tablas no pudiesen crecer más allá de pocos millones de entradas. En cuanto al programa con la XOR en un principio prometía más que las otras opciones:

1. Convertir en hexadecimal
2. Coger grupos de 8 bits y aplicar XOR por el siguiente grupo de 8 bits.
3. Coger grupos de 6 bits
   3.1. X = Aplicar modulo espacio de caracteres (26)
   3.2. Coger la letra en la posición X del abecedario
4. Unir las letras

Los resultados obtenidos han sido mucho peores y, además, para las columnas 200 y 300 en valores grandes de filas se tardaba demasiado de generar la tabla arco iris:

Porcentaje de éxito (%)				
	Columnas			
Filas	20	50	100	200
30 000	1	1	1	1
50 000	3	2	1	1
60 000	2	1	-	-
70 000	2	1	-	-

Otra función recodificante alternativa ha sido una que hiciese exactamente lo mismo función original del capítulo anterior, pero al final se le hace un añadido:

1. Invocar la función recodificante con un parámetro adicional, un índice que indique en que columna "t" se ha producido la recodificación
2. X = Aplicar módulo 5 al índice (ya que son contraseñas de 5 letras)
3. Mover las X primeras letras al final de la contraseña

A pesar de parecer una prometedora idea en un principio esta estrategia no ha producido ninguna equivalencia de las contraseñas originales.

En último lugar se ha intentado con otra función que también incluye un índice:

1. Convertir hash a hexadecimal
2. Quedarse con los últimos 5 caracteres del hexadecimal
3. Iterar caracteres:
   3.1. Si es letra añadir directamente al resultado
   3.2. Si es número multiplicarlo por el índice de la columna t y aplicarle módulo 16 para convertirlo así en carácter hexadecimal. Añadir este carácter al resultado

La función no ha dado ningún buen resultado, incluso ha sido incapaz de generar una tabla arco iris de tamaño adecuado.

Lograr implementar un índice que posibilitase que para cada columna se hiciese de forma diferente la recodificación reduciría las colisiones internas en la generación de la tabla y no se perderían tantas cadenas, pero no se ha podido averiguar ninguna función recodificante que lograse esos resultados.

2. Algoritmo Yuval

2.1 Implementación base

Para la implementación de Yuval se ha seguido una aproximación directa con las diapositivas proporcionadas de clase. Sin embargo, antes de proceder a emplear Yuval se debe invocar a otro método para adecuar los mensajes que se van a emplear. Para ello se llama al método *messageLoad* que transforma un CSV, el cual tiene 30 filas siendo separadas en 2 columnas para que cada una de estas tenga una versión alternativa/modificada, en una lista de tuplas. De esta forma, en esta lista el primer elemento se tendría la tupla de la línea 1 [(Versión 0, Versión 1) …]. La llamada messageLoad consiste simplemente en proporcionarle la ruta del archivo CSV y devuelve la lista:

messageLoad (Ruta Archivo CSV)

Con los mensajes preparados ya se puede pasar a usar Yuval eligiendo el mensaje legítimo, ilegítimo, *N.º Caracteres Hex Resumen* y el exponente del número de mensajes que se desee:

Yuval (Mensaje Legítimo, Mensaje Ilegítimo, N.º Caracteres Hex Hash, N.º mensajes)

Para esta implementación se ha escogido MD5 que devuelve 128 bits. Debido a las limitaciones de potencia de cómputo del equipo empleado, (más adelante se especifica) a lo largo de todas las pruebas que se realizan se coge un número limitado de los bits de mayor peso.

De forma general el código de la implementación base (apéndice) realiza las siguientes operaciones:

1. Invocar *messageLoad* para preparar los CSV a emplear
2. Invocar Yuval
   - 2.1 Crear diccionario legítimo
   - 2.2 Realizar todas las posibles combinaciones para N.º Bits de mensaje legítimo
     - 2.2.1 Añadir ceros detrás de cada combinación hasta 30 bits
     - 2.2.2 Construir el mensaje legítimo
     - 2.2.3 Calcular Hash del mensaje
     - 2.2.4 Guardar en el diccionario
   - 2.3 Realizar todas las posibles combinaciones para N.º Bits de mensaje ilegítimo
     - 2.3.1 Añadir ceros detrás de cada combinación hasta 30 bits
     - 2.3.2 Construir el mensaje ilegítimo
     - 2.3.3 Calcular Hash del mensaje
     - 2.3.4 Comprobar si el hash ilegítimo se encuentra dentro del diccionario.
     - 2.3.5 Si hay colisión devolver combinación binaria legítima, ilegítima y hash coincidente.

Adicionalmente, se ha creado un método *mainYuval() (apéndice)* simplemente para gestionar las llamadas del algoritmo y determinar que mensajes ilícitos proporcionar, así como crear las tablas de resultados. Esté método es de gran ayuda en experimentación base, pero aún más para la extendida a la hora de contar colisiones, es decir es un método auxiliar para poder guardar resultados.

Para programar todo este código anterior se ha escogido Python como lenguaje y Visual Studio Code como editor.

2.1.1 Experimentación base

Las especificaciones del equipo empleado en las pruebas se corresponden con un Portátil Lenovo Legión que cuenta con las siguientes características:

Procesador: AMD Ryzen 7 5800H (8 núcleos, N.º de subprocesos 16, Frecuencia Base 3.2 GHz)

Tarjeta Gráfica: Nvidia RTX 3060

Almacenamiento SSD: WDC PC SN730 SDBPNTY-1T00-1101 (1 Tb)

RAM: 16 GB DDR4 (3200 MHz)

Sistema Operativo: Windows 11

En las primeras experimentaciones se van a realizar unas simples pruebas para ver si el programa encuentra colisiones en los textos. El mensaje legítimo e ilegítimo se encuentra en el apéndice. Se van a ejecutar 3 pruebas con un hash de 28 bits, 40 bits y 48 bits para 3 textos diferentes ilícitos frente a un solo texto lícito. Los 3 textos se van a variar simplemente cambiando el nombre de "Paco" por Daniela y Juan Pérez.

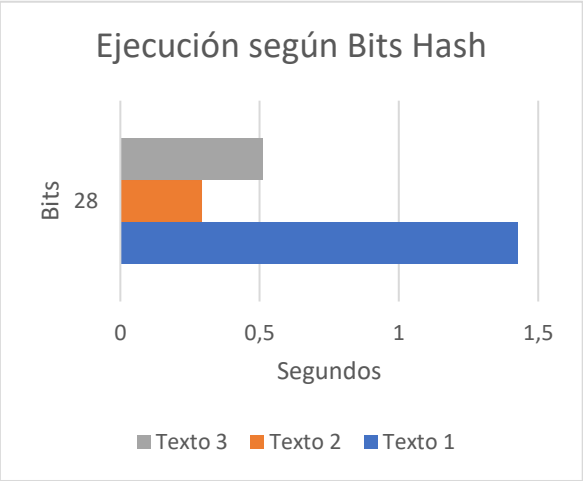
En caso de que los documentos no colisionen la estrategia a seguir en este caso que se ha elegido es aumentar en un bit el número de mensajes, empezando siempre por encima de $2^{m/2}$. Los resultados arrojados para los 3 diferentes textos ilegítimos son:

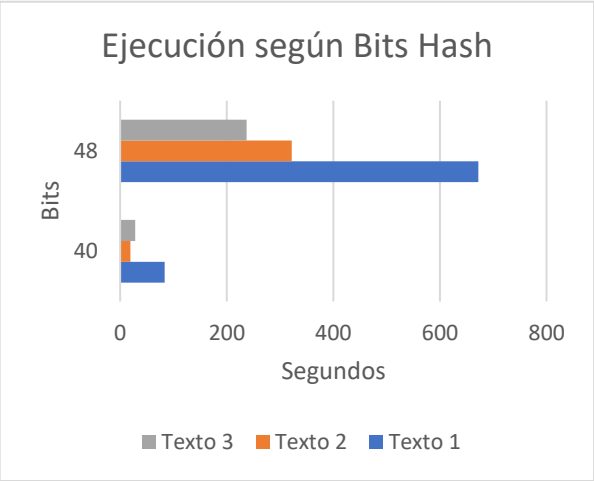
Texto	Bits Hash	N.º Mensajes	Combinación Legítima	Combinación Ilegítima	Hash	Tiempo (s)
1	28	2^{16}	10010111001001010 0000000000000	10111110110110110 0000000000000	eee455c	1,426
2		2^{14}	10110101011011000 0000000000000	10000100100011000 0000000000000	0e2d799	0,293
3		2^{15}	10110000111101100 0000000000000	10010011100001000 0000000000000	8837972	0,510
1	40	2^{22}	10010010111100010 1000100000000	10001100001111111 0001100000000	a9a01b911a	83,750
2		2^{20}	10001101111111001 1010000000000	11010011000100000 1100000000000	d0f9cd5955	18,990
3		2^{21}	10100111100010111 1011000000000	10100000101111001 0000000000000	c7d2453ccf	28,214
1	48	2^{25}	10001111111011001 1100101000000	10011011100100010 1110101100000	6dfba9b49094	672,072
2		2^{24}	11010000101000110 0000110000000	10010001001010010 1001111000000	2df4b2c8f5e1	321,668
3		2^{24}	10001001010100111 1000111000000	10011110010010111 0111100000000	99895be4ae32	237,419

Como se puede observar a partir de la tabla en muchas de estas ejecuciones no se ha podido encontrar la colisión siempre a la primera. Para el texto 2 con 28 bits de hash lo ha podido encontrar solo con 2^{14} sin necesitar aumentar, en cambio, para el texto 1 se ha visto obligado a aumentar en 2 bits logrando finalmente la colisión en 2^{16}.

Obviamente esto implica tiempos muy diferentes a pesar de buscar para un mismo tamaño de hash. El ejemplo más extremo se encuentra para 28 Bits de hash, donde se ha tardado un poco más 4

veces más para el texto 1 que para el texto 2. Para ello se puede ver dicho comportamiento en las siguientes gráficas:

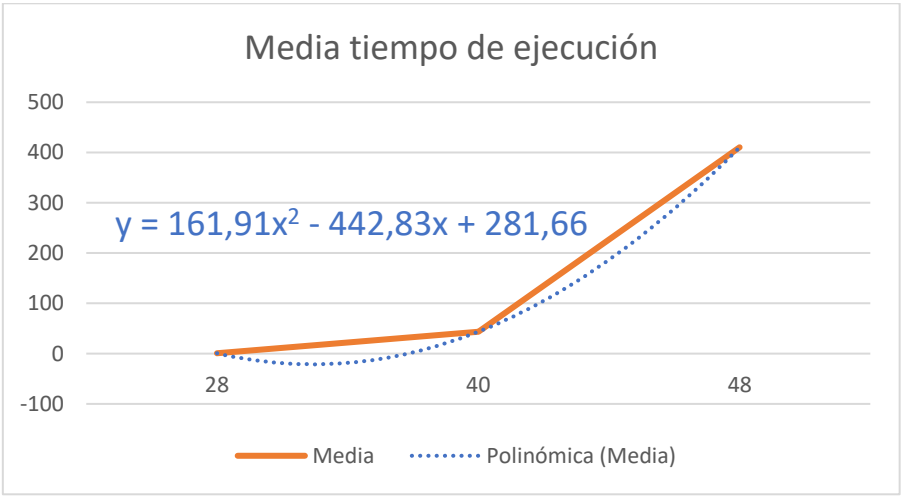




Asimismo, con los datos que ya se tienen previamente se puede calcular las medias de los tiempos para así obtener lo que se tarda en promedio según los bits de hash empleados:

Bits Hash	Media
28	0,743
40	43,6513333
48	410,386467

Con todo esto se podría realizar una predicción de **¿cuánto se tardaría para 256 bits?** Obviamente la cantidad de datos para este cálculo es realmente limitada, sin embargo, resultará interesante para más tarde compararlo con la experimentación extendida. Empleando Excel se ha agregado una línea de tendencia que intente pasar lo más cercano a los 3 puntos y por este motivo la función que más se adapta ha sido una polinómica de grado 2.



Usando la función del gráfico, el resultado para 256 bits obtenido es de **2916,07 horas o 121,5 días.** Se vuelve a incidir que este primer cálculo es muy probable que esté mal, ya que el número de puntos de datos es muy bajo, pero en el apartado de experimentación extendida se volverá a calcular con mayor exactitud. Se podría haber reflejado también en las tablas anteriores el número de iteraciones necesarias

para la colisión del hash, aunque tampoco es imprescindible a la hora de extraer la conclusión al poder hacerlo solo con el tiempo. Esto se ha hecho para reflejar que el tiempo de ejecución es muy variable y está muy ligado con la "suerte" a la hora de encontrar una colisión, donde **puede ser que ocurra en la primera iteración, la última o nunca** para los bits elegidos**.** De esta forma se explica el motivo por el cual 2 textos ilícitos de la misma longitud solo con pequeñas modificaciones puedan dar resultados tan dispares en tiempo.

Consumo memoria

Otro aspecto interesante a lo largo de las pruebas realizadas es el consumo de memoria RAM y su evolución. Se ha notado a medida que se hacían las ejecuciones que cuanto más aumenta el N.º de mensajes posibles mayor es el consumo de memoria. Este hecho es totalmente lógico ya que **crece el diccionario que guarda los hashes de los mensajes lícitos** y esta tabla lo confirma**:**

N.º mensajes	Memoria RAM (MB)
2^{12}	59,28
2^{14}	60,34
2^{16}	65,17
2^{18}	80,77
2^{20}	146,05
2^{22}	406,80
2^{24}	1450,23
2^{26}	5622,94

Estas pruebas se han parado en 2^{26} sin aumentar más el número puesto que se tardaría mucho tiempo en completar la ejecución, pero se puede intuir que para 2^{28} el ordenador portátil no tendría suficiente RAM para guardar todo el diccionario lícito. **Almacenar un diccionario puede resultar en un problema de memoria principal.**

Otro detalle interesante, es que en las primeras versiones iniciales del programa el listado de combinaciones posibles del mensaje no se generaba dinámicamente, sino que se guardaban todas las combinaciones en una lista y después se iteraba. Esto provocaba un mayor aumento del consumo de memoria RAM y por este motivo se ha optado por una implementación alternativa que no produjese esta situación:

for actualCombo in itertools.product(["0", "1"], repeat=m):

En cuanto a la memoria secundaria en esta versión del programa no influye, ya que no se guarda prácticamente ningún dato, todo permanece en memoria principal.

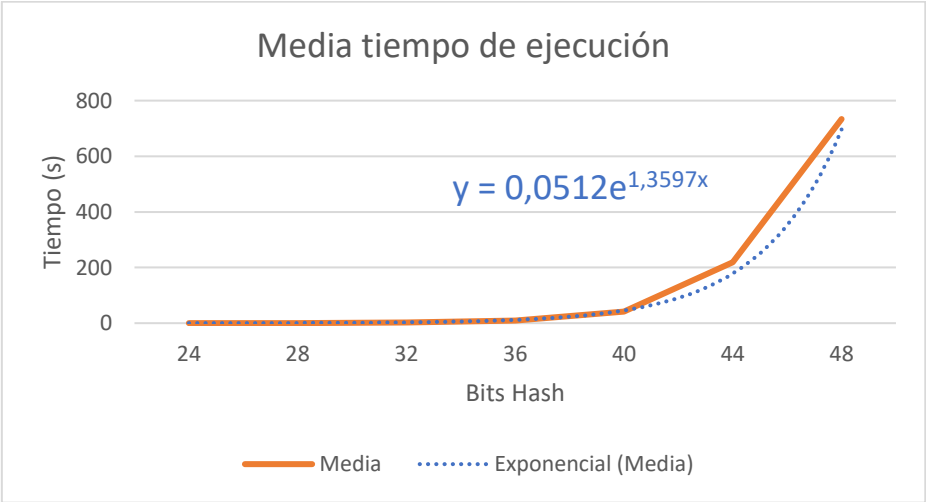
2.2 Experimentación extendida

Como se ha podido notar del anterior apartado las pruebas necesitan una mayor profundidad en cuanto a la cantidad de datos para extraer conclusiones firmes. Por este motivo, se han creado a partir del original 14 textos ilícitos más para buscar un total de 15 colisiones. Con tal de poder generarlos es tan simple como cambiar el nombre de "Paco" por uno de este listado: Daniela, Juan Pérez, María García, Carlos Rodríguez, Laura Martínez, Javier López, Andrea González, Miguel Fernández, Patricia Sánchez, Alejandro Torres, Ana Ramírez, Luis Cruz, Elena Vargas, Clara Piedra.

En este apartado es de gran importancia el método mainYuval(), que al igual que en el anterior, permite ir aumentando los bits de mensaje en caso de que no haya colisión. Pero en este caso posibilita iterar los 15 documentos ilícitos para que cada uno tenga una colisión en los diferentes tamaños de Hash. Todos estos datos que se van generando se exportan mediante la librería Pandas en un CSV externo. Se han hecho pruebas para 24, 28, 32, 36, 40, 44 y 48 bits de Hash con los resultados reflejados en el apéndice. En total la prueba ha tardado 25,3 minutos. A continuación, se procede a hacer una tabla conjunta para sacar la media de los datos:

Bits Hash	Media
24	0,2341
28	0,7978
32	2,8269
36	8,8783
40	41,9046
44	218,5007
48	733,7169

A partir de la tabla anterior se puede ver el gran incremento en el tiempo si se aumenta el número de bits, a causa de que como ya se vio en anteriores apartados es necesario aumentar el número de mensajes posibles. Si esto se refleja en una gráfica queda aún más claro:



En la gráfica también se ha incluido una nueva línea de tendencia para estimar como sería la evolución a mayores valores de Bits en el Hash. Para la situación actual la función que más se adecuaba a la tendencia de la línea ha sido la exponencial. Por lo tanto, se puede volver a plantear la pregunta **¿cuánto se tardaría para 256 bits?** Usando así la ecuación proporcionada si se sustituye se obtiene el resultado de **8, 7774 × 10^{147} días**, un valor gigantesco. Asimismo, si se opta por una ecuación polinómica de grado 3 (y

= $2{,}8968x^4 - 33{,}108x^3 + 133{,}14x^2 - 214{,}67x + 112{,}76$) que también se adapta bastante bien se tardarían igualmente **$1{,}3767 \times 10^8$ días**, el cual sigue siendo extremadamente elevado. Se destaca así que gracias al haber realizado más pruebas se puede afirmar que el valor de 121,5 días del apartado anterior de solo 3 textos estaba mal calculado al tener pocos datos de entrada.

En conclusión, a partir de todo lo visto en estos apartados, si se emplease todo el Hash que proporciona la función de MD5 y no solamente los X bits de mayor peso **sería prácticamente improbable** que alguien en un tiempo realista a través de Yuval logre una colisión (con el mismo el hardware de estas pruebas).

2.3 Implementación alternativa

Hipótesis: Si se calcula también el diccionario ilícito guardándolo en memoria en vez de solo el lícito empeoran los resultados

Esta hipótesis se espera que sea cierta después de sacar todas conclusiones del apartado anterior. El hecho de tener que calcular todo un diccionario adicional entero se puede predecir que no solo empeorará los resultados, sino que también doblará el gasto de memoria principal.

Por ello, se han obtenido 15 colisiones para los hashes de 24 a 40 bits y se han comparado frente a los resultados anteriores:

Bits Hash	Media Original	Media Nueva	Cambio %
24	0,2341	0,3202	36,78%
28	0,7978	1,1308	41,74%
32	2,8269	4,3232	52,93%
36	8,8783	12,7914	44,07%
40	41,9046	51,9711	24,02%

Las pruebas no se han continuado a mayores valores de Hash puesto que se observa que el comportamiento ya es peor con un aumento del tiempo promedio de 39,9%, aunque sorprende que el cambio porcentual en el Hash 40 es menor que en los anteriores. **Resultaría erróneo pensar que el cambio de tiempo fuese el doble exactamente**, a causa de que como bien se mencionó anteriormente la colisión en el código original bien se podría dar en las últimas iteraciones del mensaje ilegítimo, haciendo que tengan tiempos similares con esta nueva versión del programa.

En cuanto al consumo de memoria primaria al estar construyendo 2 diccionarios completamente en vez de solo 1 tiene sentido que el consumo aumente como se observa en la siguiente tabla:

N.º mensajes	Memoria RAM (MB)
2^{12}	59,81
2^{14}	69,82
2^{16}	70,85
2^{18}	102,72
2^{20}	234,86
2^{22}	754,78
2^{24}	2840,97
2^{26}	¿?

Resulta interesante como este cambio de uso de RAM no se nota entre 2^{12} y 2^{16}, muy probablemente se deba a que el diccionario es tan pequeño que ocupa muy poco espacio y no es perceptible su tamaño. Si se va aumentando el número de mensajes se puede llegar al extremo de ocupar varios gigabytes como es el caso de 2^{24} con 2,84 Gigabytes. La siguiente ejecución con 2^{26} no se ha realizado ya que tardaría mucho, pero viendo el comportamiento de las últimas ejecuciones si en la versión original esta ejecución ocupó 5,622 Gigabytes ahora podría ser alrededor de 10 Gigabytes, ya que parece estabilizarse en el **doble de almacenamiento al haber 2 diccionarios.**

A partir de esta hipótesis, que ha demostrado ser verdadera, se pueden confirmar 2 cosas:

1. Tener que procesar completamente 2 diccionarios implica peor tiempo.
2. Tener que procesar un diccionario entero ocupa mucha memoria principal. (Hecho que también se confirmó anteriormente en el apartado de "Consumo de Memoria")

Con estas conclusiones se podría conducir a una nueva hipótesis que haga hincapié en reducir el diccionario lícito que hay en memoria disminuyendo los problemas anteriormente mencionados:

Hipótesis: Si se fracciona el diccionario se resuelven los problemas de memoria principal y se acortan los tiempos

En esta hipótesis hay 2 aproximaciones diferentes:

1) **Fraccionar por cantidad de iteraciones:** esta aproximación es muy interesante porque se basa en dividir el diccionario legítimo en fragmentos, por ejemplo, por la mitad. Cuando se llegue a la mitad de las iteraciones se procedería a empezar a realizar ya la búsqueda frente a hashes ilícitos. Se podría dar el caso que si la mayoría de las veces la colisión está en la primera mitad del diccionario el tiempo se reduciría drásticamente. La contrapartida que desde luego se espera es que en caso de que esté en la segunda mitad en realidad estás computando hasta 2 veces los hashes ilegítimos, aumentado por la otra parte el tiempo. Después de realizar varias pruebas con 15 colisiones para estos hashes se han obtenido los siguientes resultados en 26,4 minutos:

Bits Hash	Media Original	Media Nueva	Cambio %
24	0,2341	0,0897	-61,68%
28	0,7978	0,3996	-49,91%
32	2,8269	1,7352	-38,62%
36	8,8783	6,786	-23,57%
40	41,9046	34,43	-17,84%

Si en vez de 2 el diccionario se parte en 4 se obtendrían estos resultados (40,3 minutos):

Bits Hash	Media Original	Media Nueva	Cambio %
24	0,2341	0,1602	-31,57%
28	0,7978	0,7487	-6,15%
32	2,8269	3,0909	9,34%
36	8,8783	11,4835	29,34%
40	41,9046	41,8287	-0,18%

Después de obtener todos los anteriores resultados quedan claras varias cosas: si la colisión resulta estar dentro de la primera fracción de diccionario, entonces el resultado es muy bueno, sin embargo, si no es así el resultado empeora bastante. Algo que sí que no reflejan las anteriores tablas es que en aquellos documentos ilícitos que no se ha encontrado colisión el tiempo de ejecución ha sido bastante superior. Si se recogen las duraciones de las pruebas se queda de esta forma:

Estrategia Yuval	Duración prueba 15 colisiones (min)
Original	25,3
Fraccionar en 2	26,4
Fraccionar en 4	40,3

Con esto se concluye que fraccionar el diccionario en diferentes partes en general no tiene por qué darte mejores resultados. Incluso cuanto más se fraccione el diccionario, como es el caso de un diccionario de 4 partes, más lento irá el algoritmo.

2) **Fraccionar por tamaño en memoria principal:** ya se sabe a partir del análisis anterior que fraccionar aumenta el tiempo. Sin embargo, soluciona un grave problema que se lleva acarreando en todo este trabajo: límite de memoria principal. Modificando la condición de fracción del código anterior se ha implementado una nueva versión que fracciona cuando el proceso de Python llegue a 5 Gb. De esta forma cuando llegue a dicho tamaño se procederá a realizar la búsqueda con hashes ilegítimos y si no se encuentra se calcula el siguiente fragmento de 5 Gb del diccionario legítimo. Se ha realizado una prueba de Yuval para Hash 52 con 2^{26} mensajes (el cual ocupaba 5,6 Gb en memoria anteriormente) ahora ha pasado a consumir como máximo 5 Gb, pero el tiempo se ha incrementado de 48,2 minutos a 65,8 minutos al encontrarse en la segunda mitad del diccionario.

En conclusión, de esta hipótesis se puede extraer que realmente los tiempos no se mejoran, pero emplear la estrategia de fraccionamiento del diccionario resuelve el problema de memoria principal para un gran número de mensajes.

Apéndice

Versión Arco Iris Numéricos

```
import statistics
import time
from zlib import crc32
import random
import pickle
import pandas as pd

longitud = 6
caracteres = '0123456789'


def h(input): #Funcion de resumen
    input = bytes(input, encoding='utf-8')
    result = crc32(input)
    return result


def r(value):
    result = str(value % 1000000) #Para que siempre devuelva 6 digitos
    return result


def generateRainbowTable(n, t):
    rainbowTable = {}
    while len(rainbowTable) < n:
        #Password al azar
        pi = ''.join(random.choice(caracteres) for _ in range(longitud))
        p = pi

        for j in range(1, t+1):
            p = r(h(p))

        rainbowTable[h(p)] = pi

    with open('tablaArcoIris.pkl', 'wb') as f:
        pickle.dump(rainbowTable, f)


def searchColision(table, p0, t):
    p = p0

    for i in range(t):
        if p in table:
            break

        p = h(r(p))

    if i == t - 1:
        return False

    else:
        pwd = table[p]
        i=0
```

```
        while (h(pwd) != p0 and i < 5*t):
            pwd = r(h(pwd))
            i+=1

        if(h(pwd) == p0):
            return [table[p], pwd]

    return [table[p], None]


realPasswords = ['523824', '941167', '782749', '481939', '736342']

def mainArcoIris():
    dfResult= pd.DataFrame()
    columns = 200
    rows = 5000

    for i in range(10):
        print(i)
        times = []
        equalPasswords = []
        colisions = 0
        passwordsGuessed = 0

        generateRainbowTable(rows, columns)
        with open('tablaArcoIris.pkl', 'rb') as f:
            tabla = pickle.load(f)

        for passw in realPasswords:
                p0 = h(passw)

                start_time = time.time()
                result = searchColision(tabla, p0, columns)
                total_time = time.time() - start_time

                times.append(total_time)

                if(result != False):
                    colisions+=1
                    if(result[1] != None):
                        equalPasswords.append((passw, result[1], result[0]))
                        passwordsGuessed += 1

        new_row = {
                    "Filas": rows,
                    "Columnas": rows,
                    "% Colisiones": (colisions/(len(realPasswords)))*100,
                    "% Éxito": (passwordsGuessed / len(realPasswords))*100,
                    "Tiempo Medio Búsqueda": statistics.mean(times),
                    "(Original / Equivalente / Colisión)": equalPasswords
                    }
        dfResult = pd.concat([dfResult, pd.DataFrame([new_row])], ignore_index=True)


    dfResult.to_csv("ArcoIris  Resultado/ArcoIrisResult Numerico.csv", index=False)

mainArcoIris()
```

Versión Arco Iris Alfabético

```
caracteres = 'abcdefghijklmnopqrstuvwxyz'
longitud = 5

def h(input): #Funcion de resumen
    input = bytes(input, encoding='utf-8')
    result = crc32(input)
    return result

def r(value, index):
    binaryResult = bin(value)[2:].zfill(32)

    #TRANSFORMAR BITS EN CARACTERES LEGIBLES
    letra1 = caracteres[int(binaryResult[0:6], 2)      %   len(caracteres)]
    letra2 = caracteres[int(binaryResult[6:12], 2)     %   len(caracteres)]
    letra3 = caracteres[int(binaryResult[12:18], 2)    %   len(caracteres)]
    letra4 = caracteres[int(binaryResult[18:24], 2)    %   len(caracteres)]
    letra5 = caracteres[int(binaryResult[24:32], 2)    %   len(caracteres)]

    result = letra5 + letra4 + letra3 + letra2 + letra1

    return result

def generateRainbowTable(n, t):
    rainbowTable = {}
    while len(rainbowTable) < n:
        #Password al azar
        pi = ''.join(random.choice(caracteres) for _ in range(longitud))
        p = pi

        for j in range(1, t+1):
            p = r(h(p), j)

        rainbowTable[h(p)] = pi
        print(len(rainbowTable))

    with open('tablaArcoIris.pkl', 'wb') as f:
        pickle.dump(rainbowTable, f)

def searchColision(table, p0, t):
    p = p0
    for i in range(t):
        if p in table:
            break
        p = h(r(p, i))

    if i == t - 1:
        return False

    else:
        pwd = table[p]
        i=0
        while (h(pwd) != p0 and i < 5*t):
            pwd = r(h(pwd), i)
            i+=1
```

```
        if(h(pwd) == p0):
            return [table[p], pwd]

    return [table[p], None]

realPasswords = ['qigvh', 'tipsh', 'buvmx', 'qvydt', 'pkvjb', 'fiulf', 'wunvo', 'cmycf', 'yaltu', 'jufbh', 'nuloa',
'kqbxr', 'hwnrv', 'odvfd', 'zrypq', 'nxidl', 'uckmh', 'tqrjk', 'figfk', 'ewhqd', 'boufa', 'yvxeq', 'jexrh', 'wkkme',
'ulliq', 'lzrmp', 'zebhi', 'jezia', 'pjscs', 'qgatm', 'hsmwd', 'zcwdo', 'dmzym', 'mnobj', 'jwtrh', 'hiewd', 'obzee',
'phhsk', 'eduin', 'eklin', 'ssdkq', 'ugsti', 'wupcl', 'vhbyu', 'quubj', 'guvgu', 'fvgzm', 'exzys', 'rovcn', 'xossw',
'bxydy', 'xodbb', 'tzkro', 'oukdc', 'jcwdj', 'aqjcq', 'zntny', 'rzdrl', 'owgbu', 'nbkao', 'ygncc', 'kjvnl', 'pddhc',
'ilxfp', 'vqtva', 'fhvjd', 'whpdu', 'ibrvo', 'efqid', 'vjaof', 'igpyh', 'hbyzp', 'qrwah', 'uzued', 'ghuli', 'axawj',
'xdmlq', 'tudjd', 'bkovu', 'waree', 'snrug', 'dxakb', 'fqztp', 'ozjgq', 'nbkym', 'gdrgu', 'glfkb', 'kldpy', 'ndvro',
'bbcpo', 'uxpne', 'crmcn', 'pzjdv', 'sdpjj', 'zgxru', 'ldtyi', 'ztbdt', 'sfyic', 'psebu', 'klhbm']

dfResult= pd.DataFrame()
dfEquals= pd.DataFrame()
columns = [20,50,100, 200]
rows = [60000, 70000, 80000, 90000, 100000]

for rws in rows:
    print(rws)
    for cols in columns:
        times = []
        equalPasswords = []
        colisions = 0
        passwordsGuessed = 0

        start_total_time = time.time()
        generateRainbowTable(rws, cols)
        with open('tablaArcoIris.pkl', 'rb') as f:
            tabla = pickle.load(f)

        for passw in realPasswords:
            p0 = h(passw)

            start_search_time = time.time()
            result = searchColision(tabla, p0, cols)
            total_search_time = time.time() - start_search_time

            times.append(total_search_time)

            if(result != False):
                colisions+=1
                if(result[1] != None):
                    equalPasswords.append((passw, result[1], result[0]))
                    passwordsGuessed += 1

        total_time = time.time() - start_total_time
        new_row = {
                "Filas": rws,
                "Columnas": cols,
                "% Colisiones": (colisions/(len(realPasswords)))*100,
                "% Éxito": (passwordsGuessed / len(realPasswords))*100,
                "Tiempo Medio Búsqueda": round(statistics.mean(times), 4),
                "Tiempo total": round(total_time, 4),
                "(Original / Equivalente / Colisión)": equalPasswords
                }
```

```
        dfResult = pd.concat([dfResult, pd.DataFrame([new_row])], ignore_index=True)

dfResult.to_csv("ArcoIris  Resultado/ArcoIrisResult.csv", index=False)
```

Funciones recodificantes alternativas

```
def r(value): #Dado el hex del hash....
   value = hex(value)[2:].zfill(8)

   #APLICAR OPERACIONES XOR
   grupo1 = bin(int(value[0:2],16) ^  int(value[2:4],16))[2:].zfill(8)
   grupo2 = bin(int(value[2:4],16) ^  int(value[4:6],16))[2:].zfill(8)
   grupo3 = bin(int(value[4:6],16) ^  int(value[6:8],16))[2:].zfill(8)
   grupo4 = bin(int(value[6:8],16) ^  int(value[0:2],16))[2:].zfill(8)

   binaryResult = grupo4 + grupo3 + grupo2 + grupo1


   #TRANSFORMAR BITS EN CARACTERES LEGIBLES
   letra1 = caracteres[int(binaryResult[0:6], 2)      %   len(caracteres)]
   letra2 = caracteres[int(binaryResult[6:12], 2)     %   len(caracteres)]
   letra3 = caracteres[int(binaryResult[12:18], 2)    %   len(caracteres)]
   letra4 = caracteres[int(binaryResult[18:24], 2)    %   len(caracteres)]
   letra5 = caracteres[int(binaryResult[24:32], 2)    %   len(caracteres)]

   result = letra5 + letra4 + letra3 + letra2 + letra1
   return result
```

```
def r(value, index):
   binaryResult = bin(value)[2:].zfill(32)

   #TRANSFORMAR BITS EN CARACTERES LEGIBLES
   letra1 = caracteres[int(binaryResult[0:6], 2)      %   len(caracteres)]
   letra2 = caracteres[int(binaryResult[6:12], 2)     %   len(caracteres)]
   letra3 = caracteres[int(binaryResult[12:18], 2)    %   len(caracteres)]
   letra4 = caracteres[int(binaryResult[18:24], 2)    %   len(caracteres)]
   letra5 = caracteres[int(binaryResult[24:32], 2)    %   len(caracteres)]

   result = letra5 + letra4 + letra3 + letra2 + letra1

   indexCaracteres = index % 5
   result = result[(indexCaracteres):] + result[:indexCaracteres]
   return result
```

```
def r(value, index):
   value = str(hex(value)[2:].zfill(8))[-5:]
   result = ""
   for charHex in value:
      if charHex.isalpha():
         result = result + charHex
      else:
         result = result + hex((int(charHex, 16)*index) % 16)[2:]
   return result
```

Mensaje Legítimo Versión 0

"Saludos estimado Silviu. El motivo de esta notificación es informarle sobre una importante modificación en las políticas de privacidad de la organización. Hemos realizado algunos ajustes menores en los términos y condiciones de la política de privacidad. Le instamos a que lea atentamente los cambios que hemos realizado en nuestra política. Estos cambios están diseñados para mejorar la transparencia y la protección de sus datos personales. Es esencial que esté totalmente informado sobre estos cambios y cómo afectan sus derechos y su privacidad. Para acceder a los detalles de los cambios, lo invitamos a examinar el documento de este correo electrónico. Encontrará una versión actualizada de la política de privacidad con todas las alteraciones resaltadas para su fácil identificación. Le solicitamos que, después de revisar las modificaciones y al estar de acuerdo con ellas, firme este mismo documento como muestra de su consentimiento y aceptación de todas las recientes políticas de privacidad. Puede hacerlo electrónicamente, si prefiere, a través del certificado electrónico para devolvérnoslos. Es importante tener en cuenta que si decide no firmar los documentos y no aceptar los cambios en nuestras políticas de privacidad, es probable que no pueda continuar utilizando los servicios de la misma forma que lo hace actualmente. Las modificaciones se han implementado para cumplir con las normativas de privacidad y mejorar la protección de su información, por lo que su consentimiento es fundamental para continuar utilizando nuestros servicios. Atentamente, Empresa Rodrigo"

Mensaje Legítimo Versión 1

"Saludos querido Silviu. La razón de este aviso es notificarle acerca de una significativa modificación en nuestras políticas de privacidad de los servicios. Hemos efectuado algunas pequeñas adaptaciones sobre términos y condiciones de nuestra política de privacidad. Le recomendamos leer cuidadosamente las alteraciones que se han efectuado en nuestra política. Estos ajustes están diseñados para aumentar la transparencia y la salvaguardia de su información personal. Es esencial que esté plenamente al tanto de estos cambios y cómo impactan en sus derechos y su privacidad. Para obtener los detalles de las modificaciones, lo invitamos a examinar el documento de este correo electrónico. Hallará una versión actualizada de nuestra política de privacidad con todas las modificaciones resaltadas para su fácil reconocimiento. Le solicitamos que, después de revisar las modificaciones, firme este documento como indicación del consentimiento y aceptación de todas las recientes políticas de privacidad. Puede hacerlo electrónicamente, si prefiere, a través del certificado electrónico para devolvérnoslos. Es relevante considerar que si decide no firmar los documentos y no aceptar las modificaciones en nuestras políticas de privacidad, es probable que no pueda continuar utilizando los servicios de la misma forma que lo hace en la actualidad. Las modificaciones se han implementado para cumplir con las normativas de privacidad y mejorar la protección de su información, por lo que su consentimiento es fundamental para continuar utilizando nuestros servicios. Cordialmente, Empresa Santiago."

Mensaje Ilegítimo Versión 0

"Saludos estimado Silviu. El motivo de esta notificación es informarle sobre una importante modificación en relación a la cesión de bienes en posesión. Hemos realizado algunos ajustes mayores en los términos y condiciones de los activos en su posesión. Le instamos a que examine detenidamente las alteraciones que hemos realizado en la gestión de dichos bienes. Estos cambios están diseñados para aclarar los términos de la cesión y el usufructo de los mismos. Es fundamental que esté plenamente informado sobre estos cambios y cómo afectan sus derechos y responsabilidades. Para obtener los detalles de las modificaciones, le invitamos a revisar el documento a este correo electrónico. Encontrará una versión actualizada de los términos y condiciones con todas las alteraciones resaltadas para su fácil identificación. Le solicitamos que, una vez haya revisado los cambios, y si está de acuerdo con ellos, firme este mismo documento como muestra de su consentimiento a entregar a Paco todos sus bienes. Puede hacerlo electrónicamente, si así lo prefiere, a través de su certificado electrónico para enviárnoslos de nuevo. Es importante tener en cuenta que si decide firmar los documentos y acepta los cambios relativos a la cesión, es seguro que no pueda seguir utilizando los bienes de la misma manera que lo hace actualmente. La notificación se ha enviado para cumplir con las normativas y comprobar sus plenas facultades, por lo que su consentimiento es fundamental para continuar utilizando nuestros servicios en la cesión. Atentamente, Empresa Rodrigo"

Mensaje Ilegítimo Versión 1

"Saludos querido Silviu. La razón de este aviso es notificarle acerca de una significativa modificación en cuanto a la transferencia de bienes bajo su custodia. Hemos efectuado algunos cambios de importancia en las condiciones y disposiciones de los activos bajo su control. Le urgimos a que revise minuciosamente las modificaciones que hemos realizado en la administración de dichos bienes. Estos ajustes están diseñados para informar los términos relativos a la cesión y el usufructo de los mismos. Es esencial que esté completamente informado sobre estos ajustes y cómo influyen en sus derechos y obligaciones. Para acceder a los detalles de los cambios, le invitamos a examinar el documento de este mismo correo. Hallará una versión actualizada de los términos y condiciones con todas las modificaciones resaltadas para su fácil reconocimiento. Le pedimos que, una vez haya revisado las modificaciones y al estar de acuerdo con ellas, firme el documento como indicación del consentimiento y aceptación de entregar a Paco todos los bienes. Puede hacerlo electrónicamente, si lo desea, a través del certificado electrónico para devolvérnoslo. Es relevante considerar que si decide firmar los documentos y acepta las modificaciones relativas a la cesión, es certero que no pueda continuar disfrutando de los bienes de la misma forma que lo hace en la actualidad. El mensaje se ha enviado para cumplir con las normativas y comprobar sus plenas facultades, por lo que su consentimiento es fundamental para continuar utilizando nuestros servicios en la cesión. Cordialmente, Empresa Santiago."

Versión Base Yuval

```
import csv
import itertools
import hashlib
import time
import statistics
import pandas as pd

def yuval(legitMessage, ilegitMessage, h, m):
   dictTextHashes = {}
   for actualCombo in itertools.product(["0", "1"], repeat=m):
      actualCombo = bin(int(''.join(map(str, actualCombo)), 2))[2:].ljust(30, '0')
      combinedText = ""
      for index, value in enumerate(actualCombo):
         combinedText = f'{combinedText} {legitMessage[index][int(value)]}'

      dictTextHashes[hashlib.md5(combinedText.encode('utf-8')).hexdigest()[:h]] = actualCombo

   for actualCombo in itertools.product(["0", "1"], repeat=m):
      actualCombo = bin(int(''.join(map(str, actualCombo)), 2))[2:].ljust(30, '0')
      combinedText = ""
      for index, value in enumerate(actualCombo):
         combinedText = f'{combinedText} {ilegitMessage[index][int(value)]}'

      ilegitHash =  hashlib.md5(combinedText.encode('utf-8')).hexdigest()[:h]

      if ilegitHash in dictTextHashes:
         legitCombo = dictTextHashes[ilegitHash]
         return [str(legitCombo), str(actualCombo), str(ilegitHash)]
```

Método auxiliar Yuval (para pruebas extensas y generar resultados)

```
def mainYuval(minHash, maxHash):
    legitMessage = messageLoad("Mensajes/mensajeLicito.csv")

    for hashValue in range(minHash, maxHash+1):
        times = []
        messageBits = hashValue*2
        colisions = 0
        df_print = pd.DataFrame()

        while colisions < 15:
            ilegitMessage = messageLoad('Mensajes/mensajeIlicito ' + str(colisions) + '.csv')
            start_time = time.time()
            result = yuval(legitMessage, ilegitMessage, hashValue, messageBits)
            total_time = time.time() - start_time
            if result != None:
                times.append(total_time)
                colisions += 1

                new_row = {
                    "Bits Mensaje": messageBits,
                    "Comb Binaria Lícita": result[0],
                    "Comb Binaria Ilícita": result[1],
                    "Hash": result[2],
                    "Tiempo (s)": round(total_time, 4)
                    }

                df_print = pd.concat([df_print, pd.DataFrame([new_row])], ignore_index=True)
                messageBits = hashValue*2

            else:
                messageBits += 1

        df_print.to_csv("Resultados Yuval/Hash " + str(hashValue) + " " + str(round(statistics.mean(times),4)) + ".csv",
index=False)
```

Método Yuval Alternativo 1: generar 2 diccionarios

```
def yuval(legitMessage, ilegitMessage, h, m):
    dictTextHashesLegit = {}
    dictTextHashesIlegit = {}

    for actualCombo in itertools.product(["0", "1"], repeat=m):
        actualCombo = bin(int(''.join(map(str, actualCombo)), 2))[2:].ljust(30, '0')
        combinedText = ""
        for index, value in enumerate(actualCombo):
            combinedText = f'{combinedText} {legitMessage[index][int(value)]}'

        dictTextHashesLegit[hashlib.md5(combinedText.encode('utf-8')).hexdigest()[:h]] = actualCombo

    for actualCombo in itertools.product(["0", "1"], repeat=m):
        actualCombo = bin(int(''.join(map(str, actualCombo)), 2))[2:].ljust(30, '0')
        combinedText = ""
        for index, value in enumerate(actualCombo):
            combinedText = f'{combinedText} {ilegitMessage[index][int(value)]}'

        dictTextHashesIlegit[hashlib.md5(combinedText.encode('utf-8')).hexdigest()[:h]] = actualCombo

    print(round(psutil.Process(os.getpid()).memory_info().rss / 1024 ** 2, 4))

    for key in dictTextHashesLegit:
        if key in dictTextHashesIlegit:
            print("para h" +  str(h))

            return [str(dictTextHashesLegit[key]), str(dictTextHashesIlegit[key]), str(key)]

    return None
```

Método Yuval Alternativo 2: Fraccionar por iteraciones

```
def yuval(legitMessage, ilegitMessage, h, m):
    dictTextHashes = {}
    totalIterations = 2**m
    actualIteration = 0

    while actualIteration < totalIterations:
        if (((actualIteration % (totalIterations / 2)) == 0)) and actualIteration != 0):
            for actualComboIlegit in itertools.product(["0", "1"], repeat=m):
                actualComboIlegit = bin(int(''.join(map(str, actualComboIlegit)), 2))[2:].ljust(30, '0')
                combinedText = ""
                for index, value in enumerate(actualComboIlegit):
                    combinedText = f'{combinedText} {ilegitMessage[index][int(value)]}'

                ilegitHash =  hashlib.md5(combinedText.encode('utf-8')).hexdigest()[:h]

                if ilegitHash in dictTextHashes:
                    legitCombo = dictTextHashes[ilegitHash]
                    return [str(legitCombo), str(actualComboIlegit), str(ilegitHash)]

            dictTextHashes = {} #Delete dict

        actualComboLegit = bin(actualIteration)[2:].zfill(30)
        combinedText = ""
        for index, value in enumerate(actualComboLegit):
            combinedText = f'{combinedText} {legitMessage[index][int(value)]}'

        dictTextHashes[hashlib.md5(combinedText.encode('utf-8')).hexdigest()[:h]] = actualComboLegit
        actualIteration += 1

    return None
```

Método Yuval Alternativo 3: Fraccionar por tamaño

```
def yuval(legitMessage, ilegitMessage, h, m):
    dictTextHashes = {}
    totalIterations = 2**m
    actualIteration = 0

    while actualIteration < totalIterations:
        if ((actualIteration % (totalIterations / 2)) == 0)) and actualIteration != 0):
            for actualComboIlegit in itertools.product(["0", "1"], repeat=m):
                actualComboIlegit = bin(int(''.join(map(str, actualComboIlegit)), 2))[2:].ljust(30, '0')
                combinedText = ""
                for index, value in enumerate(actualComboIlegit):
                    combinedText = f'{combinedText} {ilegitMessage[index][int(value)]}'

                ilegitHash =  hashlib.md5(combinedText.encode('utf-8')).hexdigest()[:h]

                if ilegitHash in dictTextHashes:
                    legitCombo = dictTextHashes[ilegitHash]
                    return [str(legitCombo), str(actualComboIlegit), str(ilegitHash)]

            dictTextHashes = {} #Delete dict

        actualComboLegit = bin(actualIteration)[2:].zfill(30)
        combinedText = ""
        for index, value in enumerate(actualComboLegit):
            combinedText = f'{combinedText} {legitMessage[index][int(value)]}'

        dictTextHashes[hashlib.md5(combinedText.encode('utf-8')).hexdigest()[:h]] = actualComboLegit
        actualIteration += 1

    return None
```

Resultados Experimentación Extendida

Hash 24 bits

Texto	N.º Mensajes	Comb Binaria Lícita	Comb Binaria Ilícita	Hash	Tiempo (s)
1	14	10110001101110000000000000000000	10110001001011000000000000000000	3bb2a3	0.3299
2	13	11110110111010000000000000000000	11011100110100000000000000000000	a45ba4	0.1393
3	13	10111001010110000000000000000000	10100101010010000000000000000000	6b193d	0.1648
4	13	10000110001110000000000000000000	11101001001010000000000000000000	a4c1ff	0.1888
5	14	10110100101001000000000000000000	10011011001000000000000000000000	e639a8	0.2114
6	14	11011101010100000000000000000000	11010000101101000000000000000000	b922f8	0.3494
7	14	10110011100001000000000000000000	11101000001110000000000000000000	0085b8	0.2873
8	14	10101111010010000000000000000000	11110101010101000000000000000000	674ac2	0.3821
9	13	10011110100100000000000000000000	10111000011110000000000000000000	6ccdd5	0.1691
10	14	10001110000011000000000000000000	10011000000100000000000000000000	a95b03	0.2318
11	14	11101101100001000000000000000000	11010110001010000000000000000000	49f1a4	0.2923
12	13	10101001010010000000000000000000	10111100100110000000000000000000	d5a211	0.1709
13	13	10100101000110000000000000000000	11101101101000000000000000000000	136169	0.1226
14	14	10010011100001000000000000000000	10110011101000000000000000000000	c027b0	0.214
15	14	11010101000001000000000000000000	10011000100110000000000000000000	66156a	0.258

Hash 28 bits

Texto	N.º Mensajes	Comb Binaria Lícita	Comb Binaria Ilícita	Hash	Tiempo (s)
1	16	10010111001001000000000000000000	10111110110110100000000000000000	eee455c	1.4041
2	14	10110101011011000000000000000000	10000100100011000000000000000000	0e2d799	0.2935
3	15	10110000111101100000000000000000	10010011100001000000000000000000	8837972	0.5101
4	16	10001101101011000000000000000000	10101010010011000000000000000000	f8e4d83	0.9325
5	16	11011111101101100000000000000000	11101110111100000000000000000000	ab8c950	0.8301
6	15	11110110101101000000000000000000	10000100011101100000000000000000	f5a0be0	0.5914
7	16	11011001010011000000000000000000	10110111100000000000000000000000	84ca28b	0.8105
8	16	10001110111010100000000000000000	10101011100011000000000000000000	410ce02	1.3153
9	16	10011001100101000000000000000000	10110010010101000000000000000000	a31524f	0.9261
10	15	11000110011001100000000000000000	11100111111111100000000000000000	0c14796	0.7527
11	16	11101010111110000000000000000000	11111011000101000000000000000000	f53b4f0	0.9981
12	15	11001100110101000000000000000000	11010100001000100000000000000000	c8c8962	0.7269
13	15	10010111111001000000000000000000	11011110101101100000000000000000	aa7edb0	0.7393
14	14	10111101100011000000000000000000	11000110000001000000000000000000	d5a2ead	0.3448
15	15	10100000101100000000000000000000	11111001111110100000000000000000	af483c8	0.7911

Hash 32 bits

Texto	N.º Mensajes	Comb Binaria Lícita	Comb Binaria Ilícita	Hash	Tiempo (s)
1	19	11101100101111010110000000 00000	11001011010101100000000000 00000	8992c3c4	6.9707
2	18	11110110000101001100000000 00000	10001010010111110000000000 00000	3410fc2d	3.7650
3	16	10110000111101100000000000 00000	10010011100001000000000000 00000	88379724	0.9245
4	18	10101001000111000100000000 00000	11110010011011110000000000 00000	26803ef2	4.0631
5	17	11001110100001010000000000 00000	11000000000101101000000000 00000	2914201e	2.8524
6	16	10100101111000100000000000 00000	11100110010110010000000000 00000	47579046	1.5469
7	17	10100100100101100000000000 00000	11010110010101011000000000 00000	1a5d4cc2	3.0086
8	17	11111110100101011000000000 00000	10010100100100110000000000 00000	ea76d59e	2.1162
9	17	11000010111111011000000000 00000	11111110110111011000000000 00000	476e07a0	3.2521
10	17	11001001111011001000000000 00000	11111010110111100000000000 00000	61b515dc	2.0461
11	18	11101100110100100100000000 00000	11111010111000000000000000 00000	0511ee85	3.3370
12	16	11001100110101000000000000 00000	11010100001000100000000000 00000	c8c89628	1.1622
13	18	11110101101011101100000000 00000	11001111001011101000000000 00000	71b3d312	4.6396
14	16	10111101100011000000000000 00000	11000110000001000000000000 00000	d5a2ead6	0.9702
15	17	11011101010000111000000000 00000	10001100101001000000000000 00000	ee63c36c	1.7485

Hash 36 bits

Texto	N.º Mensaje	Comb Binaria Lícita	Comb Binaria Ilícita	Hash	Tiempo (s)
1	19	11111110010000101111000000000000	10100000001010110100000000000000	3ea21cc63	8.8119
2	19	11100001100000100010000000000000	11000110101011001000000000000000	8d1918d4e	7.9983
3	20	11100100010000111001000000000000	11110101010110101100000000000000	508d58cfc	16.7699
4	19	10100011000011010010000000000000	10011000110001100000000000000000	36b84a0a0	6.8909
5	20	11010011101001010101000000000000	10100000000100011100000000000000	e2f4d7c70	15.5115
6	19	10011000010000001001000000000000	10111111110001010100000000000000	037606c2a	9.1566
7	19	11010011111111001111000000000000	10010111011110110010000000000000	e2c8320c0	10.5834
8	19	11001100110011101110000000000000	11001010010100010100000000000000	0d8009a39	9.3872
9	18	11000010111111011000000000000000	11111110110111011000000000000000	476e07a00	4.9712
10	19	10011110001001101100000000000000	10111101100000011010000000000000	01aa852fa	11.5726
11	18	10010100010101111100000000000000	11010110111010000000000000000000	0b41d738d	3.3958
12	19	11000010100110011100000000000000	11011100010011001010000000000000	fe394e0c7	12.3597
13	18	11110101101011011000000000000000	11001111001011101000000000000000	71b3d312c	4.6299
14	18	11100001101011100000000000000000	11111010011110111000000000000000	f8400723b	6.5002
15	18	10111010111010001100000000000000	11010001010000011100000000000000	a94fbbdfa	4.6354

Hash 40 bits

Texto	N.º Mensaje	Comb Binaria Lícita	Comb Binaria Ilícita	Hash	Tiempo (s)
1	22	1001001011110001010001000 00000	1000110000111111100011000 00000	a9a01b911a	84.0254
2	20	1000110111111100110100000 00000	1101001100010000011000000 00000	d0f9cd5955	18.9904
3	21	1010011110001011110110000 00000	1010000010111100100000000 00000	c7d2453ccf	28.2144
4	21	1001110010001001000000000 00000	1011000110010111010010000 00000	69f7d4fdb0	46.0255
5	21	1111001001011001000010000 00000	1100010001010110011010000 00000	cbabb7a9f1	47.4674
6	21	1000101011101100000100000 00000	1000001011001010101110000 00000	906ee9e2c7	40.6754
7	20	1010001011110000100000000 00000	1011000111010000010100000 00000	f5f38450bb	22.6799
8	22	1010011111000010110000000 00000	1001001000010110110111000 00000	650440f024	85.3774
9	21	1101000011000001100010000 00000	1001101110100111101110000 00000	d13c4b2487	43.1029
10	21	1100101111110101101010000 00000	1001111001010101000110000 00000	2bca6ab5e7	43.4731
11	21	1111101001101000110110000 00000	1111010111010100101110000 00000	f5808636af	52.4911
12	21	1100100010111110111000000 00000	1001110110110001110110000 00000	ba9d86f7f4	43.3273
13	20	1010010100111001100000000 00000	1011100010001101010100000 00000	cfa8ae7328	22.9310
14	21	1111000100111001001100000 00000	1001001001100111000100000 00000	9cb7dd79c4	34.6029
15	20	1010011001110101101100000 00000	1000100001111011010000000 00000	77f752c4fc	15.1846

Hash 44 bits

Texto	N.º Mensajes	Comb Binaria Lícita	Comb Binaria Ilícita	Hash	Tiempo (s)
1	24	11001101010010000000110010 00000	10101000010001011111111110 00000	b7a3ccccf6c	363.3904
2	23	10010111111001011000000000 00000	11110111100101100111011000 00000	348ca1aeb6d	214.0197
3	23	11111010000010101101111000 00000	11011100001001000011010000 00000	6b6f06eec17	156.0505
4	23	10100111000000000110110000 00000	11001011100101111010001000 00000	03556059254	195.0050 3
5	23	10100000001011000101101000 00000	11100110110101000111100000 00000	e1d1699384e	135.1276
6	23	11000111001001101110000000 00000	10010101111011001101111000 00000	f2467459da5	176.9130
7	23	11000011111110001100011000 00000	10010000001110011000010000 00000	099fdc8b6c9	141.5940
8	22	11010101101001111010000000 00000	10111110100001001011110000 00000	34d6dadbd3b	95.0866
9	24	11000001101101010110000010 00000	11000010100001110011101100 00000	3ad5c9e0fa8	390.8964
10	23	11000000011000110001001000 00000	10011001000101001111000000 00000	bd15e944ab1	116.7652
11	23	10000010011010111100101000 00000	11000101000111001000001000 00000	29041a3cc6c	191.5872
12	22	11011100110110110010010000 00000	11010101001001110101010000 00000	dfc1636ee1c	99.2179
13	24	11001001111101101000011100 00000	11010110001100001110011100 00000	adbb0532ded	409.2639
14	23	10111011111010010100110000 00000	10111100011100111001001000 00000	80eff60647d	197.8772
15	24	11111011111100100101010000 00000	10101110110100100000011100 00000	09a47151bcf	394.7144

Hash 48 bits

Texto	N.º Mensajes	Comb Binaria Lícita	Comb Binaria Ilícita	Hash	Tiempo (s)
1	25	1000111111101100111001010 00000	1001101110010001011101011 00000	6dfba9b49094	672.072 2
2	24	1101000010100011000001100 00000	1001000100101001010011110 00000	2df4b2c8f5e1	321.668 1
3	24	1000100101010011110001110 00000	1001111001001011011110000 00000	99895be4ae32	237.419 4
4	24	1100111011001110110111010 00000	1001111111011011100001110 00000	2feb1374bd7d	335.307 5
5	24	1111111101010001011011100 00000	1100011011101100001111110 00000	07d221b94f8e	366.054 8
6	26	1010100011001111010011010 10000	1111010101001111101100111 00000	48e4f5d06d41	1237.11 47
7	25	1111111100110110110101111 00000	1101110111111100100110111 00000	1ff6bd15aba8	772.462 6
8	26	1000100001000110010001100 00000	1100010111111010111001001 10000	70be5c46cb9f	1466.33 03
9	25	1100110011011101101110011 00000	1010110001001111110101111 00000	6afdd6b11611	689.016 86
10	26	1010110101101111001010100 10000	1100100001110110000011111 00000	a88b751e75ed	1150.76 21
11	25	1001110110100100100011111 00000	1000010000111100100010100 00000	5e38c1d74e18	465.332 4
12	25	1101011110111111110011110 00000	1001100010110101010011111 00000	55fe930dabd4	657.352 5
13	26	1111000001101101010111011 10000	1010101001000101010100010 00000	8424fd8de7cd	970.598 1
14	25	1011111011111010101100011 00000	1010100111110100110101111 00000	790d8cb12c8f	685.564 3
15	26	1011100010111100001011001 10000	1011110010001101010101010 00000	02ad42bf5633	978.697 3