Ejercicio Hoja de Problemas de Algoritmos de Compresión

Óscar Brizuela - 820773

EJERCICIO 3

Índice

Introducción

Para la implementación de ambos algoritmos se ha elegido el lenguaje *Python*, que permite el uso eficiente de bits mediante la librería *bitarray*.
Por simplicidad, sólo se han implementado los algoritmos de compresión, y no los de descompresión, tanto para el algoritmo *LZ78* como para el algoritmo de codificación por probabilidades *Huffman*. Esta decisión se debe a que el rendimiento de cada algoritmo (ratio de compresión) se puede calcular, como ya veremos, en función de la longitud del mensaje comprimido y de la longitud del mensaje sin comprimir, por lo que no es necesario ningún algoritmo de descompresión en la comparación.

1. Algoritmo de compresión *LZ78*

La implementación de este algoritmo de compresión se puede dividir en varias funciones:

- *diccionario_lista_lz78*(*cadena*): dada una cadena de caracteres, almacena en una lista cada una de las frases nuevas que el algoritmo encuentra a medida que va recorriendo la cadena. Siguiendo la idea principal del algoritmo *LZ78*, cada una de estas frases será una de las anteriores más un carácter. Además, guarda en un diccionario el número de frase anterior correspondiente a la frase actual más el carácter nuevo, no perteneciente a dicha frase anterior. Esta información se guarda en forma de tupla (int, char).
  Por ejemplo, si la frase actual (frase número 5) es “aab” y la frase número 2 es “aa”, la entrada número 5 del diccionario será (2, “b”).
  Cabe destacar que si la frase actual comienza por un carácter no encontrado hasta el momento, la entrada guardada en el diccionario será (0, *caracter_nuevo*), ya que no existe una frase anterior que contenga dicho carácter.
  Ambas estructuras de datos (tanto la lista de las frases como el diccionario que codifica el conjunto de frases) se van actualizando de manera simultánea para una mayor eficiencia del algoritmo.

  Ejemplo 1.1:
  - String (input): **aababaaba**
  - Frases: **a ab aba aba**
  - Lista de frases (devuelta por la función): **[a, ab, aba]**
  - Diccionario (devuelto por la función): **{1: (0, “a”), 2: (1, “b”), 3: (2, “a”)}**

A partir de aquí, podemos encontrarnos con dos casos diferentes: que la última frase de la cadena (aquella en cuyo final acaba la cadena) haya aparecido antes o que no. En el caso de haber aparecido antes, esta última frase no se guarda en la lista devuelta por la función, puesto que cada frase solo se guarda la primera vez que aparece. La razón de esto es que cada frase, excepto la última en ciertas ocasiones, solo aparecerá una vez, puesto que las posteriores lo más cerca que estarán de dicha frase estarán formadas por dicha frase más un símbolo como mínimo. En el ejemplo anterior, como la frase 4 es igual a la frase 3, la

frase 4 no aparece ni en la lista ni en el diccionario, en los cuales aparece tan solo la frase 3.

Sin embargo, si la última frase es nueva (es decir, es una frase anterior más un símbolo, o quizá un símbolo nuevo), sí aparece tanto en la lista de frases como en el diccionario.

Ejemplo 1.2:

- String (input): **aababaabaa**
- Frases: **a ab aba abaa**
- Lista de frases (devuelta por la función): **[a, ab, aba, abaa]**
- Diccionario: **{1: (0, “a”), 2: (1, “b”), 3: (2, “a”), 4: (3, “a”)}**

Por tanto, para lidiar con el primer caso (es decir, aquel en el que la última frase ha aparecido previamente en la cadena), se ha optado por crear una nueva función cuyo comportamiento se describe a continuación.

- *encontrar_diferencia_entre_cadenas*(*cadena1*, *cadena2*):
  Esta función, dadas dos cadenas de caracteres, comprueba si son iguales, devolviendo “Iguales” en ese caso. En caso contrario, encuentra el índice hasta el cual la cadena 2 incluye a la cadena 1, es decir, a partir del cual las dos cadenas se diferencian. La función devuelve dicho índice.

- *completar_diccionario*(*cadena*):
  La función anterior es necesaria para esta nueva función, cuyo objetivo es completar el diccionario en caso de que no estuviera ya completo debido a las razones explicadas en la explicación de la función *diccionario_lista_lz78()* de este apartado. No hace falta completar la lista ya que, al fin y al cabo, el algoritmo *LZ78* codificará solamente el diccionario.
  A esta función se le pasa como parámetro la cadena de caracteres a comprimir. La función llama a su vez a la función *encontrar_diferencia_entre_cadenas()* que, como ya se ha explicado, devuelve una lista con una serie de frases, la cual como mínimo contendrá todas las frases de la cadena menos la última. En este momento se comprueba si la concatenación de las frases devueltas por la función *diccionario_lista_lz78()* conforman la cadena a comprimir completa mediante una llamada a la función *encontrar_diferencia_entre_cadenas()*. Si no es así, es decir, si la última frase ya había aparecido en la cadena previamente, se crea una nueva entrada en el diccionario asignando a la tupla la cadena el número de frase que ya había aparecido y una cadena vacía.
  - Por tanto, el diccionario resultante con la cadena input del ejemplo 1.1 quedaría: **{1: (0, “a”), 2: (1, “b”), 3: (2, “a”), 4: (3, “”)}**

De esta forma, ya tenemos el diccionario completo listo para codificar mediante las funciones *codificar_diccionario()* y *comprimir_cadena_LZ78()*.

- *codificar_diccionario*(*diccionario*): codifica el diccionario en binario, utilizando log2 por arriba (número de frase) para el dato *i* de la tupla *(i, j)*, y para el símbolo *j*, depende de cuántos símbolos diferentes haya.

- *comprimir_cadena_LZ78*(*cadena*): es la función que se encarga de llamar a todas las demás para que, dado una cadena de caracteres, devuelva la compresión correspondiente a *LZ78* para dicha cadena. Por tanto, crea el diccionario a codificar llamando a la función *completar_diccionario()* (que, a su vez, llama a las funciones *diccionario_lista_lz78()* y *encontrar_diferencia_entre_cadenas*) y lo codifica llamando a la función *codificar_diccionario()*.

2. Algoritmo de compresión *Huffman*

La implementación de este algoritmo de compresión se puede dividir también en varias funciones, además de una clase.

- Clase *HuffmanNode*: esta clase representa un nodo de un árbol que sigue la codificación Huffman. Cada uno de estos nodos, cuando se instancia mediante el constructor, guarda información sobre el símbolo del nodo, la frecuencia (peso del nodo) con la que aparece dicho símbolo en la cadena a comprimir, el hijo izquierdo y el hijo derecho. Esta clase también redefine el operador "<" (less than) mediante el método *__lt__*, pues será necesario para construir el árbol.

- *construir_arbol_huffman*(*diccionario*): dado un diccionario que almacena los distintos caracteres de la cadena junto con sus correspondientes frecuencias, construye un árbol que sigue la codificación Huffman haciendo uso de la estructura de datos heap, facilitada por la librería de *Python heapq*.

- *construir_codigos_huffman*(*diccionario*): dado un árbol que sigue la codificación Huffman ya construido, devuelve un diccionario con la codificación correspondiente de cada símbolo. Siguiendo los fundamentos de este tipo de codificación, aquellos símbolos con frecuencia más alta en la cadena a comprimir serán codificados con menos bits.

- *comprimir_cadena_huffman*(*cadena*): de la misma manera que en algoritmo *LZ78*, esta función es la que se encarga de llamar a las demás para que, dada una cadena de caracteres, devuelva la codificación binaria correspondiente al código por probabilidades *Huffman*. esta función devuelve tanto el diccionario que contiene los códigos correspondientes a cada símbolo como la codificación de la cadena completa.

- *contar_simbolos*(*cadena*): función que, dada una cadena de caracteres, devuelve en un diccionario el número de veces que cada símbolo distinto aparece en dicha cadena. Esta función auxiliar es necesaria, puesto que la codificación por probabilidades Huffman utiliza directamente las frecuencias de cada símbolo para hacer una compresión símbolo a símbolo.

3. Comparación del nivel de compresión de ambos algoritmos

Cada una de las pruebas realizadas comprime una cadena de caracteres con una determinada configuración. Por cuestiones de simplicidad, las cadenas de caracteres utilizadas en las pruebas pueden constar de dos símbolos diferentes (“a” ó “b”, apartados .1 de cada una de las pruebas) o de todas las letras, en minúscula, del alfabeto inglés (26 símbolos diferentes en total, de la “a” a la “z”).
Las cadenas a comprimir generadas en cada una de las pruebas siguen configuraciones que pueden resultar interesantes para el análisis y la comparación del rendimiento de ambos algoritmos. Además, las cadenas generadas en las pruebas 1 y 2 permiten conocer, mediante algo de cálculo, el número de frases en función de un parámetro *n* a partir del cual también se podrá calcular el número de caracteres de la cadena. Esto es útil para calcular el ratio de compresión aproximado en función del número de frases del algoritmo *LZ78* en caso de que, debido a la larga longitud del mensaje a comprimir, el tiempo de ejecución de dicha compresión sea demasiado largo. Este radio de compresión aproximado para el algoritmo LZ78 se calcula de la siguiente manera:

$$\frac{|\,C(x)\,|}{|\,x\,|} = \frac{t(x)*(\,log(t(x)) + log(\alpha))}{|\,x\,|}\ ,$$

siendo *t(x)* el número de frases de la cadena y (α) el número de símbolos distintos.

Los tiempos presentes en cada tabla corresponden únicamente al proceso de compresión de cada algoritmo, no al de generación de las entradas correspondientes.

En el código, cada prueba muestra por pantalla (siempre que la ejecución no sea demasiado larga) la siguiente información:

- La longitud de la cadena, en bits, antes de ser comprimida
- La longitud aproximada de la cadena comprimida mediante el algoritmo *LZ78* (se ha utilizado el número de frases para el cálculo de dicha longitud)
- La longitud real de la cadena comprimida mediante el algoritmo *LZ78*
- El ratio de compresión aproximado del algoritmo *LZ78* (se ha utilizado el número de frases para el cálculo de dicha longitud)
- El ratio de compresión real del algoritmo *LZ78*
- El tiempo de ejecución de la compresión del algoritmo *LZ78*, en microsegundos
- La longitud de la cadena comprimida mediante el algoritmo de códigos de *Huffman*
- El ratio de compresión del algoritmo de códigos de *Huffman*
- El tiempo de ejecución de la compresión del algoritmo códigos de *Huffman*, en microsegundos

3.1. Prueba 1

Las cadenas de esta prueba están formadas de forma que cada frase sea la anterior más un símbolo aleatorio (“a” o “b” en el caso del apartado 3.1.1, cualquiera de las letras en minúscula del alfabeto inglés en el apartado 3.1.2).
En el código, el parámetro que se le pasa a esta prueba es el número de frases *n* que debe tener la cadena de caracteres a comprimir, siempre siguiendo la configuración explicada. Por ejemplo, si se llamara a la función que ejecuta la prueba 1 con *n = 6*, una cadena de caracteres con la que se podría realizar la prueba sería:

a ab aba abaa abaab abaaba

Por tanto, la longitud (en número de caracteres) del mensaje a comprimir seguiría la serie aritmética:

$$\sum_{m=1}^{n} m$$

Esta serie se puede calcular rápidamente en función del número de frases n con la siguiente fórmula:

$$longitud\ del\ mensaje\ (en\ caracteres)\ =\ \frac{n*(n+1)}{2}$$

Debido a que cada carácter se codifica con 8 bits, la longitud total del mensaje en bits será:

$$longitud\ del\ mensaje\ (en\ bits)\ =\ \frac{n*(n+1)}{2}\ *\ 8$$

De esta forma, con 100 frases la cadena a comprimir estará formada por 40400 bits (5050 caracteres).

Se ha elegido este tipo de configuración para la cadena a comprimir ya que garantiza el mínimo número posible de frases para un determinado número de caracteres por lo que, en principio, el número de entradas del diccionario generado por el algoritmo *LZ78* sería mínimo y, por tanto, su ratio de compresión sería muy bueno.

3.1.1. Con dos símbolos

Num frases	**Longitud de mensaje**	**LZ78**		**Huffman**	
		Ratio de compresión (real / aprox)	Tiempo de compresión	Ratio de compresión	Tiempo de compresión
100	40400	0.0167 / 0.0187	4945.80	0.125	3172.10
1000	4004000	0.0025 / 0.0027	2673715.20	0.125	2527657.30

10000	400040000	- / 0.0003	??	50005000 / 400040000 = 0.125	??

Debido a que esta prueba comprime una cadena formada solamente por los caracteres a y b, el algoritmo de los códigos de *Huffman* asigna un 0 al carácter “a”, y un 1 al carácter “b”. Por tanto, el ratio de compresión es constante. El hecho de que valga 0.125 se explica porque en el mensaje sin comprimir, cada carácter está codificado con 8 bits , mientras que en el comprimido cada carácter está codificado con un único bit. De esta manera obtenemos el ratio de compresión 1 / 8 = 0.125.

Sin embargo, se observa que, a medida que el tamaño de la cadena a comprimir va aumentando, el rendimiento del algoritmo *LZ78* es cada vez mejor, acercándose el ratio de compresión a 0 cada vez más. Esto confirma la teoría de que la configuración de la cadena generada para esta prueba es muy buena para el ratio de compresión (de hecho, mínima en cuanto a entradas en el diccionario a codificar).

3.1.2. Con todas las letras del alfabeto

Num frases	**Longitud de mensaje**	**LZ78**		**Huffman**	
		Ratio de compresión (real / aprox)	Tiempo de compresión	Ratio de compresión	Tiempo de compresión
100	40400	0.0167 / 0.0187	4176.80	0.5587	3860.20
1000	4004000	0.0025 / 0.0027	2337759.70	0.5910	2369193.70
10000	400040000	- / 0.0003	??	250025000 / 400040000 = 0.625	??

Observamos que los ratios de compresión del algoritmo *LZ78* son exactamente iguales a los de la realización de la prueba con solo dos símbolos. La razón de esto es que, independientemente de los símbolos utilizados, las frases en la cadena a comprimir están conformadas por la frase anterior con un símbolo concatenado al final, por lo que el número de símbolos distintos no influye en el ratio de compresión de este algoritmo.

La compresión utilizando los códigos de *Huffman* proporciona una media de unos 5 bits por símbolo en este caso (parte entera superior de log2(26)). Por tanto, a pesar de que al tratar

de comprimir 400040000 bits el tiempo de ejecución de la compresión es demasiado largo, podemos estimar el ratio de compresión de *Huffman* (además del de *LZ78*, explicado anteriormente):

$\frac{400040000\ bits}{8} = letras\ distintas$

Debido a que cada una cuenta con 5 bits de media
longitud del mensaje comprimido (en bits) = $50005000 * 5 = 250025000$ bits

Por tanto, si dividimos la longitud del mensaje comprimido y la longitud del mensaje sin comprimir obtenemos:

$ratio\ de\ compresión = \frac{250025000}{400040000} = 0.625$

Por tanto, queda claro que el número de símbolos diferentes tiene mucha influencia en cuanto al ratio de compresión del algoritmo de códigos de *Huffman*.

3.2. Prueba 2

Las cadenas de esta prueba están formadas de forma que se consideran todas las posibles combinaciones de caracteres para una frase de longitud 1 hasta longitud *n*. Así, para *n* = *3*, la cadena estará formada por:
Primero, las frases con longitud 1: *a b*
Segundo, las frases de longitud 2: *aa ab ba bb*
Tercero, las frases de longitud 3: *aaa aab aba abb baa bab bba bbb*
La cadena a comprimir será la concatenación de todas esas frases:

abaaabbabbaaaaababaabbbaababbbabbb

Por tanto, para calcular el número de frases de la cadena a comprimir en función de *n*, se puede seguir la serie geométrica:

$$a * r^0 + a * r^1 + a * r^2 + \ldots$$

siendo *a* = *2, r* = *2* y *n* es 1 más el máximo exponente del ratio *r* de la serie.

Por tanto, para *n* = *3*,el número de frases de la cadena será:

$2 * 2^0 + 2 * 2^1 + 2 * 2^2 --> 2 * 2^0 + 2 * 2^1 + 2 * 2^{n-1} = 14\ frases$

3.2.1. Con dos símbolos

Para saber la cantidad de caracteres en base a *n*, debemos conocer también el número de frases. Este es calculado con la función *calculate_sum*(), que devuelve el número de frases creadas con dos símbolos según la configuración de la cadena explicada y basándose en la serie geométrica anterior para este cálculo.

Una vez contamos con el número de frases, tenemos que habrá 2^n frases con *n* caracteres, 2^{n-1} frases con *n - 1* caracteres… y así sucesivamente.

Mirando el ejemplo, tenemos 2^3 frases con 3 caracteres, 2^2 frases con 2 caracteres y 2^1 frases con 1 carácter.

Teniendo esto en cuenta, la función *calculate_letters()* acepta como parámetros *n* y *m* (número de frases), y calcula el número de caracteres en base a estos dos parámetros. Una vez tenemos el número de caracteres en función de n, podemos calcular el tamaño de la cadena en bits en función de n. Por ejemplo, para *n = 8*, la cadena a comprimir tendrá 3586 caracteres (es decir, 28688 bits, tal y como indica en la tabla).

Se ha elegido este tipo de configuración para la cadena a comprimir ya que, a pesar de que no muestra ningún caso mejor ni peor para ninguno de los dos algoritmos, permite calcular el ratio de compresión aproximado en función del número de frases y la longitud del mensaje cuando el algoritmo *LZ78* tarda demasiado en comprimir 318767120 bits.

n	**Longitud de mensaje**	**LZ78**		**Huffman**	
		Ratio de compresión (real / aprox)	Tiempo de compresión	Ratio de compresión	Tiempo de compresión
8	28688	0.1599 / 0.1777	7942.00	0.125	9751.80
14	3407888	0.1442 / 0.1538	24934820.10	0.125	31316839.70
20	318767120	- / 0.1447	??	39845890 / 318767120 = 0.125	

Para el cálculo del ratio de compresión aproximado para *n = 20* del algoritmo *LZ78* se ha utilizado las funciones auxiliares *calculate_sum()* y *calculate_letters()* con el objetivo de obtener el número de frases de la cadena en función de n. Por tanto, para *n = 20*, el número de frases es 2097150. Con esta información podemos calcular el ratio de compresión aproximado siguiendo la fórmula del apartado …

De la misma forma que en la *prueba 1*, al utilizar solo dos símbolos en la cadena a comprimir el algoritmo de códigos de *Huffman* asigna un solo bit por símbolo, por lo que el ratio de correlación sigue siendo 1 / 8 = 0.125. Si lo queremos hacer con la longitud total de los mensajes comprimidos y sin comprimir, tendríamos igualmente:

$$ratio\ de\ compresión\ =\ \frac{39845890}{318767120}\ =\ 0.125$$

3.2.2. Con todas las letras del alfabeto

Debido a que, en este caso, se utilizan todos los símbolos del alfabeto (26), y el tiempo para todas las ejecuciones era demasiado largo(no solo por la compresión, sino también por la generación de la cadena a comprimir, puesto que tendría que contener todas las combinaciones posibles dados 26 símbolos), se ha procedido a calcular los ratios de compresión de *LZ78* en función del número de frases. Por su parte, para el cálculo aproximado del ratio de compresión del algoritmo de los códigos de Huffman que cada uno de los símbolos cuenta con 5 bits de media.

Además cabe apuntar que, aunque el tiempo hubiera sido asequible, no tendría sentido ejecutar esta prueba con n < 26, puesto que no se estarían teniendo en cuenta todas las posibles combinaciones para todas las letras del alfabeto.

n	**Longitud de mensaje**	**LZ78**		**Huffman**	
		Ratio de compresión (real / aprox)	Tiempo de compresión	Ratio de compresión	Tiempo de compresión
8	28688	- / 0.2489	??	17930 / 28688 = 0.625	??
14	3407888	- / 0.1923	??	2129930 / 3407888 = 0.625	??
20	318767120	- 0.1711	??	199229450 / 318767120 = 0.625	??

Observamos que los ratios de compresión (calculados de forma aproximada en función del número de frases) del algoritmo *LZ78* no son tan buenos como en la *prueba 1*, puesto que no se está tratando con cadenas con un número de frases muy reducido. Sin embargo, también se observa que, cuanto más aumenta la longitud del mensaje, mejor es el ratio de compresión, reforzando la teoría de que este algoritmo es muy bueno para entradas largas. Por su parte, el algoritmo de códigos de *Huffman* tiene un rendimiento mucho menor. A pesar de que los ratios calculados aparecen constantes independientemente de la longitud de la cadena a comprimir, puesto que se han asumido 5 bits para cada símbolo distinto, dichos ratios no deberían ser muy diferentes entre sí en la realidad. Además, el hecho de

que la frecuencia de aparición de cada carácter sea aleatoria y, por tanto, uniforme, es un argumento a favor para hacer esa asignación de bits a cada símbolo.

3.3. Prueba 3

Esta prueba genera cadenas de caracteres con alta entropía, en la que todos los símbolos siguen una distribución uniforme, tendiendo a aparecer el mismo número de veces. Se ha elegido esta prueba ya que, al ser las frecuencias similares, los códigos de *Huffman* asignados a cada símbolo no serán muy distintos entre sí, tanto en el caso de realizar la prueba con dos símbolos como con todos los del alfabeto.
En este caso no se puede calcular el número de frases para estimar el ratio de compresión del algoritmo *LZ78* en caso de que la ejecución tome demasiado tiempo. Esto se debe a que los caracteres se colocan de forma aleatoria, imposibilitando el cálculo del número de frases.
El parámetro utilizado en esta prueba es el número de caracteres n de la cadena a comprimir, con lo que los cálculos de la longitud del mensaje a comprimir se simplifican (simplemente hay que multiplicar el número *n* por 8 para saber el número de bits de la cadena).

3.3.1. Con dos símbolos

n	**Longitud de mensaje**	**LZ78**		**Huffman**	
		Ratio de compresión (real / aprox)	Tiempo de compresión	Ratio de compresión	Tiempo de compresión
5050	40400	0.1544 / 0.1690	23206.50	0.125	19459.40
500500	4004000	0.1414 / 0.1502	50474761.4	0.125	36401008.70
50005000	400040000	??	??	0.125	??

Observamos que, en este caso, el algoritmo de códigos de *Huffman* obtiene un mejor ratio de compresión debido a la forma en la que asigna bits a los dos símbolos que conforman la cadena a comprimir (uno para cada símbolo). El hecho de que los símbolos estén presentes de forma aleatoria hace que el algoritmo *LZ78* no tenga tan buen rendimiento en este caso, ya que las frases no siguen ningún patrón determinado que favorezca el ratio de compresión de este algoritmo.

3.3.2. Con todas las letras del alfabeto

n	**Longitud de mensaje**	**LZ78**		**Huffman**	
		Ratio de compresión (real / aprox)	Tiempo de compresión	Ratio de compresión	Tiempo de compresión
5050	40400	0.5125 / 0.5576	102423.40	0.5925	91121.40
500500	4004000	0.5375 / 0.5687	343676361.50	0.5959	275375021.50
50005000	400040000	??	??	0.60 aprox	??

Debido a la imposibilidad de obtener el número de frases dado *n* en esta prueba, no se han podido realizar los cálculos del ratio de compresión aproximado para el algoritmo LZ78 cuando la longitud del mensaje a comprimir es 400040000 bits.

Observamos que ambos algoritmos comprimen los mensajes con un ratio similar. Los valores de estos ratios se corresponden más con lo que nos podemos encontrar en la realidad, ya que la cadena generada para esta prueba es totalmente aleatoria y está formada por todas las letras del alfabeto. Los ratios de compresión anteriores no representan el rendimiento habitual de estos algoritmos de compresión, ya que la configuración de las cadenas a comprimir (debido al número de símbolos o a los patrones que seguían las frases) están generadas a propósito para comprobar los rendimientos en casos concretos.

3.4. Prueba 4

Al contrario que en la *prueba 3*, las cadenas de caracteres generadas para esta prueba constan de baja entropía. Por tanto, no existe una distribución uniforme, puesto que existen ciertos caracteres que tienden a aparecer con mucha más frecuencia que otros.
En el caso de la prueba con dos símbolos (*apartado 3.4.1*), el símbolo “a” tiende a aparecer un 90% de las veces, mientras que el símbolo “b” tan solo tiende a aparecer un 10%.
En el caso de la prueba con todos los símbolos del alfabeto, el símbolo “a” tiende a aparecer un 80% de las veces, mientras que el 20% restante se encuentra repartido de forma uniforme (o todo lo uniforme que permite la librería *random* de *Python*) entre los 25 caracteres restantes del alfabeto.

3.4.1. Con dos símbolos

n	**Longitud de mensaje**	**LZ78**		**Huffman**	
		Ratio de compresión (real / aprox)	Tiempo de compresión	Ratio de compresión	Tiempo de compresión
5050	40400	0.08589 / 0.09466	12213.40	0.125	8177.40
500500	4004000	0.0709 / 0.0754	12944001.90	0.125	12027982.30
50005000	400040000	??	??	0.125	??

El hecho de que se le haya dado una entropía tan baja hace que el número de frases para el algoritmo *LZ78* sea relativamente pequeño, ya que en muchas ocasiones la frase será la inmediatamente anterior (puesto que el carácter b aparece con mucha menor frecuencia). Esto hace que el ratio de compresión de este algoritmo sea muy bueno, y que mejore conforme el tamaño de la cadena a comprimir va aumentando.
Por su lado, el ratio de compresión del algoritmo de códigos de *Huffman* vuelve a ser 0.125, ya que la entropía no tiene ningún efecto cuando solo se trata con dos símbolos. Se vuelve a asignar un único bit para cada símbolo ya que no se pueden asignar menos, a pesar de que el carácter “a” aparezca muchas más veces que el carácter “b”.

3.4.2. Con todas las letras del alfabeto

n	**Longitud de mensaje**	**LZ78**		**Huffman**	
		Ratio de compresión (real / aprox)	Tiempo de compresión	Ratio de compresión	Tiempo de compresión
5050	40400	0.2034 / 0.2224	16851.80	0.2415	16357.80
500500	4004000	0.2010 / 0.0.2129	46605512.10	0.2428	46840616.30
50005000	400040000	??	??	0.24 aprox	??

De una forma parecida al caso anterior (*apartado 3.4.2*), la baja entropía hace que se generen casos favorables para el algoritmo *LZ78* en cuanto al número de frases. Sin embargo, por primera vez observamos un ratio de compresión bueno en el algoritmo de

códigos de *Huffman* cuando trata con todo el alfabeto. Esto se debe a que, eficientemente, asigna muy pocos bits (seguramente 1 o 2, dependiendo del caso) al carácter “a”, que aparece con mayor frecuencia, mientras que a todos los demás les asigna de 3 a 5 bits.

En conclusión, podemos determinar que el algoritmo *LZ78* es muy eficiente para entradas muy largas, además de facilitar el cálculo del ratio de compresión debido a que este se encuentra en función del número de frases de la cadena a comprimir.
Por su parte, el algoritmo de códigos de *Huffman* obtiene rendimientos peores cuando tratamos con muchos símbolos, y constantes cuando solo tratamos con dos símbolos, debido a su forma de asignar bits a cada símbolo.
Se aprecia también que, al contrario de lo que se podría pensar, el hecho de que la cadena a comprimir tenga una baja entropía no solo beneficia al algoritmo de códigos de *Huffman*, sino también al *LZ78* debido al número de frases generadas. En términos generales, por tanto, se puede concluir que el algoritmo *LZ78* es mejor en la mayoría de los casos, aunque no en todos.

Por último, hay que destacar que, a pesar de que los tiempos de compresión son muy similares entre los dos algoritmos, el algoritmo de códigos de *Huffman* es ligeramente más rápido.