UNIVERSIDAD POLITÉCNICA DE MADRID

Proyecto de Fin de Grado
Grado en Ingeniería de Software

compresión de ficheros de texto plano utilizando códigos de fibonacci generalizados

Author: Ricardo Hortelano Sánchez

Supervisors:
Luis Miguel Pozo Coronado

escuela tecnica superior de ingenieria de sistemas informaticos

Acknowledges

A mi tutor,
a mi familia,
a mis amigos de la facultad.

Abstract

Compression systems today use improved techniques over the years. Because of this, the study and research of certain more classical techniques has been relegated to the background. The Fibonacci succession completeness property allows to build a universal compressor code of variable length based on these numbers. A Zeckendorf Theorem ensures the completeness of the generalized Fibonacci number sequences, so these sequences would also give rise to compressor codes. In this work we build, implement and evaluate compression algorithms based on Fibonacci, Lucas and generalized Fibonacci successions (which we have called Zeckendorf codes). These algorithms have been applied to the compression of plain text, as they seem more adapted to them.

Resumen

En la actualidad los sistemas de compresión utilizan técnicas mejoradas a lo largo de los años. Debido a esto, el estudio e investigación de ciertas técnicas más clásicas ha quedado relegado a un segundo plano. La propiedad de completitud de la sucesión de Fibonacci permite construir un código universal compresor de longitud variable basado en estos números. Un Teorema de Zeckendorf asegura la completitud de las sucesiones de números de Fibonacci generalizados, por lo que estas sucesiones también darían lugar a códigos compresores. En este trabajo de construye, implementa y evalúa algoritmos de compresión basados en la sucesión de Fibonacci, Lucas y sucesiones de Fibonacci generalizadas (a las cuales hemos llamado códigos de Zeckendorf). Estos algoritmos se han aplicado a la compresion de texto plano, ya que parecen más adaptados a ellos.

Índice

Índice de Tablas

Índice de Figuras

Capítulo 1

Introducción

La nueva era de la información en la que vivimos nos ha dado el privilegio de tener a nuestra disposición enormes cantidades de datos. Es por ello que también es responsabilidad nuestra el manejar y almacenar esos datos de una manera eficiente y útil. Uno de los problemas de tener tal cantidad de datos es el de disponer de un lugar donde almacenarlos, y lo que es más importante, guardarlos ahorrando el mayor espacio posible.

En el mundo de la computación el ahorro de espacio es un tema recurrente desde sus comienzos, esto es, codificar la información de un modo estricto, conciso y recuperable de modo que el espacio de almacenamiento resultante sea lo menor posible.

Los algoritmos de compresión actuales utilizan multitud de técnicas distintas y complejas para lograr la mayor compresión posible, de modo que su análisis y/o desarrollo queda reducido a una pequeña porción de ingenieros y matemáticos expertos en la materia.

En este proyecto se va a investigar y desarrollar un nuevo algoritmo de compresión de archivos centrado en disminuir el tamaño final de un tipo de archivo muy determinado, de texto plano, usando para ello técnicas poco convencionales si se comparan con las usadas actualmente tanto por programas de código abierto como privativos de compresión.

1.1 Objetivos

El objetivo principal del proyecto es el de estudiar e implementar algoritmos de compresión basados en códigos de longitud variable universal, en particular el código de Fibonacci. Para ello se estudiarán y aprovecharán las propiedades matemáticas de la sucesión de Fibonacci así como se analizará e implementará un código de compresión basado en la generalización de la sucesión (teorema de Zeckendorf). Por ultimo se comparará el rendimiento de estos algoritmos con algunos de los programas comerciales de compresión más utilizados en la actualidad.

1.1.1 Objetivos Específicos

Nos proponemos los siguientes objetivos a cumplir para el proyecto.

1. Analizar los métodos actuales de compresión y ubicar el algoritmo a desarrollar dentro de estos.
2. Investigar y definir teóricamente la técnica de compresión a utilizar.
3. Diseñar el algoritmo de compresión
4. Implementar el algoritmo de compresión.
5. Realizar un estudio comparativo entre los resultados obtenidos de utilizar el algoritmo propio y los programas de compresión comerciales.

1.2 Introducción Teórica

Según la Teoría de la Información, la compresión de datos, a la que nos referiremos a partir de ahora como concisión de los datos, se consigue eliminando la redundancia. Como contrapartida, este método da como resultado una información menos segura y más propensa a errores, lo que se denotará como integridad de los datos.

La forma de lograr una información más segura y menos propensa a errores, esto es, lograr una mayor integridad, se consigue añadiendo bits de comprobación, lo que incrementa el tamaño resultante. De este modo es lógico darse cuenta de que la concisión de datos y su nivel de integridad son dos conceptos opuestos e inversos. A más garantía de integridad, menos concisión, y viceversa.

Es interesante saber que el concepto de concisión, pese a ser una rama de la computación reciente, lleva existiendo desde antes de la invención de las computadoras, como es el caso del código Braille, desarrollado por Louis Braille y utilizado para hacer posible la lectura a personas ciegas. Este código consiste en grupos (o celdas) de 3x2 puntos cada una, marcadas en una gruesa hoja de papel. Cada punto puede ser plano o elevado, así que la información contenida en cada celda es equivalente a 6 bits, dando como resultado 64 posibles grupos. Como el número de letras y signos de puntuación no requieren del uso de todos los grupos, los grupos resultantes se utilizan para codificar las palabras comunes más utilizadas – como and, for, y of – o para cadenas de caracteres – como wh, ch, y th. (Figura 1.1)

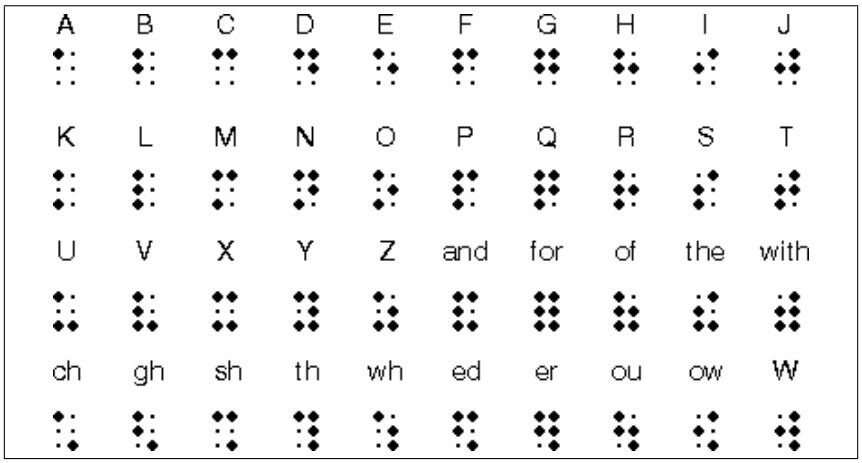


Figura 1.1: Letras y cadenas del código Braile

La cantidad de compresión lograda por el código Braille es pequeña pero importante, ya que el tamaño de los libros escritos usando este sistema suele ser bastante grande (un solo grupo ocupa el espacio de diez letras escritas). Aun con todo, está pequeña compresión acarrea un precio. Si un libro de Braille se trata incorrectamente o envejece y algunos puntos se vuelven planos, podrían surgir problemas serios de lectura.

1.2.1 Códigos con perdida

En algunos casos muy específicos es posible comprimir la información desechando algunas partes de esta que, o bien no son necesarias al ser imperceptibles o bien la

perdida de información es mínima comparado con la ganancia en tamaño ahorrado. Esta "compresión" se conoce como "compresión irreversible", "compactación" o "compresión con perdida". Un ejemplo de esta técnica podría ser la sustitución de varios espacios seguidos por uno solo en un fichero de texto, la restricción del rango de colores posibles en el sistema #RGB o la acotación superior e inferior de la frecuencia en un fichero de audio para que se adecue a el umbral de audición humano. Pese a que él estudio de estas técnicas abarcan tanto como los códigos sin perdida, su uso y conocimiento no ha sido necesario para la realización de este trabajo.

1.2.2 Técnicas Básicas

Dentro de las técnicas básicas o *clasicas* de la compresión sin perdida de archivos podemos encontrar multitud de ellas, desde las más primitivas como usar una tabla de códigos distinta, código de Baudot [MacMillan and Krandall, 2010], hasta técnicas diseñadas específicamente para un sistema, como es el caso del formato BinHex 4.0 para Macintosh [Lewis, 1991]. Para este trabajo solo se detallarán dos técnicas por la relación que guardan con la técnica a desarrollar.

RLE

Evolucionando el concepto de compresión surgen los Run Length Encoding (RLE). Estos métodos podrían resumirse de modo que: Si se sabe que un carácter[1] d ocurre n consecutivas veces en un flujo, entonces se puede sustituir ese flujo de d caracteres por un simple par nd. n consecutivas ocurrencias de un carácter es llamado un run length de n, y este uso en la compresión de datos es a lo que se llama RLE.

Este tipo de compresión suele necesitar de un carácter especial (o de escape) el cual solo se utilizaría para indicar que lo que viene a continuación es un flujo compreso. RLE no suele ser efectivo en textos, ya que la aparición de varias letras iguales de forma consecutiva es extraña tanto en castellano como en inglés y sería necesaria la aparición de más de tres caracteres consecutivos para que pudiera existir una mínima compresión.

[1] Entendiendo carácter como unidad mínima de información en un contexto.

Ejemplo 1 *La siguiente cadena de texto:*

El peeerro del hortelanooooo noooooo me deja comer.

podría ser compresa (usando RLE y @ como carácter de escape) del siguiente modo

El p@3erro del hortelan@5o n@6o me deja comer.

reduciendo su tamaño inicial de 51 caracteres a 46.

Aun con todo el uso de RLE es muy efectivo en compresión de imágenes y específicamente el protocolo MNP Clase 5 hace uso de este método para la compresión de datos en modems de forma eficaz.

Move-To-Front

La idea principal de esta codificación es la de mantener en una lista mutable los elementos de un alfabeto A de modo que los elementos más utilizados ocupan los primeros puestos de esa lista. De este modo, el método aprovecha la naturaleza del lenguaje y permite una codificación con números relativamente bajos (Figura 1.2)

El algoritmo funciona del siguiente modo; primero, se comienza con un array que contenga un alfabeto ordenado, después por cada aparición de un elemento a comprimir se apunta en otro array objetivo la posición que ocupa dicho elemento en el alfabeto original y a continuación se mueve el elemento del alfabeto a la primera posición del array, alterando de este modo el alfabeto original. Este proceso se repetirá del mismo modo pero utilizando el array del alfabeto modificado hasta haber recorrido todo el texto a comprimir. Una vez terminado el proceso, si tenemos el array objetivo y la ultima modificación del array alfabeto podremos revertir el proceso y descomprimir.

De modo análogo, esta codificación puede ser utilizada usando una lista de palabras en lugar de elementos del alfabeto. Si bien esta variante también puede ser utilizada con una lista que contiene todas las palabras posibles lo usual es que la lista se vaya formando con las palabras que aparecen en el texto de un modo dinámico.

Iteration	Sequence	List
bananaaa	1	(abcdefghijklmnopqrstuvwxyz)
ba**n**anaaa	1,1	(bacdefghijklmnopqrstuvwxyz)
ban**a**naaa	1,1,13	(abcdefghijklmnopqrstuvwxyz)
bana**n**aaa	1,1,13,1	(nabcdefghijklmopqrstuvwxyz)
banan**a**aa	1,1,13,1,1	(anbcdefghijklmopqrstuvwxyz)
banana**a**a	1,1,13,1,1,1	(nabcdefghijklmopqrstuvwxyz)
bananaa**a**	1,1,13,1,1,1,0	(anbcdefghijklmopqrstuvwxyz)
bananaa**a**	1,1,13,1,1,1,0,0	(anbcdefghijklmopqrstuvwxyz)
Final	1,1,13,1,1,1,0,0	(anbcdefghijklmopqrstuvwxyz)

Figura 1.2: Codificación usando Move-To-Front.

Es fácil ver que la transformación es reversible. Simplemente manteniendo la misma lista y decodificando de modo que se reemplace cada índice en la secuencia codificada con la letra de ese índice de la lista. Se puede ver una clara diferencia entre este método y el método de codificación: el índice en la lista se usa directamente en lugar de buscar cada valor para su índice.

1.2.3 Métodos Estadísticos

Una característica que comparten todas las variantes de RLE y parecidos es que todos hacen uso de códigos con tamaño prefijado sobre los que actúan. En contraste, los métodos estadísticos hacen uso de códigos de tamaño variable, de modo que a los elementos que más aparecen en el fichero a comprimir se les asignan los códigos más pequeños. Esta nueva metodología acarrea dos nuevos problemas:

1. Asignar códigos que pueden ser decodificados sin ambigüedad.
2. Asignar códigos con el tamaño medio más bajo.

Del mismo modo, uno de los aspectos más importantes de la ciencia de la codificación es la Teoría de la Información [Shannon, 1948].

1.2.4 Teoría de la Información

De forma intuitiva parece ser que todo el mundo entiende lo que es la información. Constantemente se recibe y se envía información en forma de texto, voz e imágenes. También se tiene la sensación de que es un concepto no-matemático que no se puede medir con precisión. De hecho, la definición que da la RAE es:

> (5). f. Comunicación o adquisición de conocimientos que permiten ampliar o precisar los que se poseen sobre una materia determinada.

Posiblemente, una de las definiciones más acertadas teóricamente hablando sobre ¿Qué es la información? Es la dada por la teoría de la información de Shannon. Según [Salomon, 1997]

> La importancia de la Teoría de la Información es que consigue cuantificar la información. Muestra cómo medir la información de modo que se puede responder a la pregunta "¿Cuánta cantidad de información hay incluida en este fragmento de datos?" con un número preciso utilizando sistemas probabilísticos. La cuantificación de la información se basa en la cantidad de *sorpresa* que contiene un mensaje. De modo que si se comunica algo que ya se sabía con anterioridad la cantidad de información enviada es nula. La inversa de esta *sorpresa* se le denomina redundancia de modo que cuanta menos redundancia se comunique más información (sorpresa) se estará transmitiendo.

Como ya se dijo en la introducción, el objetivo de la compresión de datos es eliminar esa redundancia, de modo que los datos contengan el mayor número posible de información sin repetir.

Otro de los aspectos clave de la teoría de la información es la función matemática que expresa la información y esta es logarítmica. Para poder explicar de forma simplificada el porqué de este hecho se puede tomar como premisa un sistema similar al que utiliza el popular juego 20Q, que trata de contar el número de preguntas con respuesta Si/No que son necesarias para llegar a la respuesta final. Cada respuesta Si/No puede ser codificada con un bit.

Pregunta	Respuesta	Codificación
¿Es un animal?	Si	1
¿Le gusta correr?	Si	1
¿Vale mucho dinero?	No	0
¿Puede gruñir?	Si	1
¿Es domestico?	Si	1
¿Es un perro?	Si	1

Tabla 1.1: Preguntas de 20Q para la respuesta perro.

La premisa muestra que cuanto más complicada sea la respuesta final más preguntas van a ser necesarias para llegar a está, es decir, mientras más información es necesaria mayor será el tamaño de los datos. Como puede verse en la Tabla 1.1, cada pregunta trata de evitar al máximo ser redundante, de modo que cada pregunta trata de maximizar al máximo la información que aporta.

Usando este sistema se puede tratar de encontrar un numero entre 1-64 con preguntas del mismo tipo. La forma óptima es usar la búsqueda binaria, de modo que primero se preguntaría si el número está contenido entre 1-32, si la respuesta es No, se sabe que está entre 33 y 64 y la siguiente pregunta sería si está contenida entre 33-48, etc. De esta forma se puede ver que el número máximo de preguntas necesarias para saber el número buscado es seis. Esto es porque 6 es el número de veces que 64 puede ser divido entre la mitad o de forma matemática $6 = \log_2 64$. Esto nos puede ayudar a entender porque el logaritmo en base 2 es la función matemática que expresa la información medida en bits.

1.2.5 Entropía

La información enviada es redundante, esto es, los contenidos transmitidos normalmente contienen más datos que información real. Para conseguir una mayor concisión, y por tanto la compresión de los datos enviados o almacenados, en necesario extraer la información y desechar el excedente de datos. La teoría de la información cuantifica la

información, de modo que nos permite saber, mediante métodos probabilísticos, como de buena es una compresión y cuál es el límite teórico para comprimir. De forma simplificada se puede definir la cantidad de información de un carácter a_i como el inverso de la probabilidad de que aparezca en un mensaje, es decir:

$$a_i = \frac{1}{P(a_i)} = \frac{1}{P_i} \tag{1.1}$$

Más información se recibe de un carácter mientras menos probable sea que ese carácter nos aparezca, de modo que los caracteres que más aparecen aportan menos información y son más redundantes, con lo cual son más propensos a ser compresos. Todo este concepto es definido como entropía de un mensaje.

La definición formal de este hecho se puede hacer del siguiente modo:

Definición 1 *Se define H como la cantidad de información transmitida en cada unidad mínima de información n en un periodo de tiempo, esto es,*

$$H = s \log_2 n \tag{1.2}$$

donde s es la cantidad de unidades mínimas de información transmitidas en el periodo de tiempo.

El siguiente paso es expresar H como la probabilidad de ocurrencia de un símbolo n. Se asume que el símbolo a_i con una probabilidad P_i y por supuesto $P_1+P_2+P_3+\cdots+P_n = 1$. En el caso especial de que todas las n probabilidades sean iguales $Pi = P$, se tiene que $1 = \sum P_i = nP$ lo que implica que $P = 1/n$, con lo cual nos queda que $H = s \log_2 n = s \log_2 (\frac{1}{P}) = -s \log_2 P$. Ahora bien, en general, las probabilidades son diferentes y lo que se quiere es expresar H en términos de probabilidad de todos ellos en la forma general. Como el símbolo a_i ocurre con P_i probabilidad significa que en promedio ocurre sP_i veces por periodo de tiempo, así que tenemos $H = -sP_i \log_2 P_i$. La suma de todos los n símbolos arroja que $H = -s \sum_1^n P_i \log_2 P_i$. Como recordatorio y aclaración, se define pues, que la entropía de los datos es la cantidad mínima de bits necesarios, en promedio, para representar los símbolos en una codificación determinada.

La entropía de los datos depende en las probabilidades individuales de P_i y es menor cuando más se aproximan las probabilidades de n. Este hecho se utiliza para

definir la redundancia en los datos. Es definida como la diferencia entre la entropía y la menor entropía posible.

$$R = \left(-\sum_{1}^{n} P_i \log_2 P_i \right) - \log_2 \left(\frac{1}{P} \right) = -\sum_{1}^{n} P_i \log_2 P_i + \log_2 n \tag{1.3}$$

Podemos observar que una compresión perfecta (sin redundancia) seria de la siguiente forma $\sum_1^n P_i \log_2 P_i = \log_2 n$ lo que coincide con la función que define la teoría de la información.

1.2.6 Códigos de longitud variable

Aprovechando esta teoría se han desarrollado diferentes métodos de compresión donde el carácter del código compreso resultante es de tamaño variable, al contrario de lo que ocurre en RLE. Probablemente el código más famoso de este estilo sea el Código Morse, donde en su variable en ingles usa un solo carácter para codificar la letra E, dos para la I y A, cinco para los dígitos, etc.

Es importante recalcar que el código Morse es un código ternario, es decir, tiene tres caracteres con lo que su función de información H sería $s \log_3 n$.

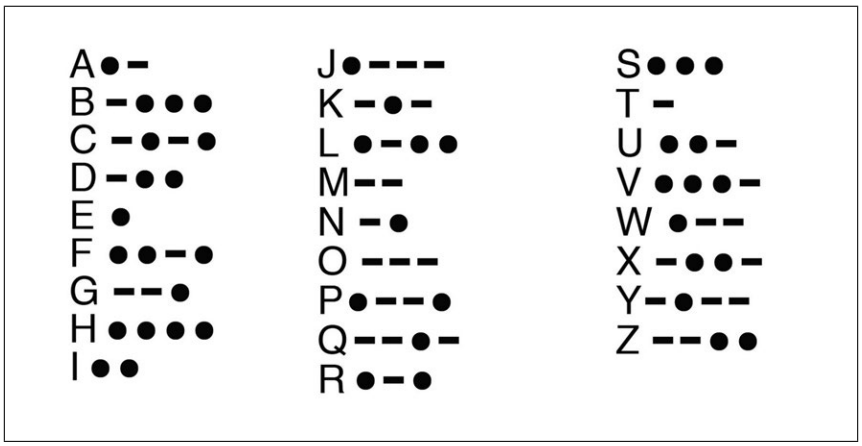


Figura 1.3: Código Morse.

Desarrollando estos métodos surgen muchos otros los cuales podemos explicarlos de forma genérica a través del siguiente ejemplo:

Ejemplo 2 *Se consideran cuatro símbolos;* a_1,a_2,a_3,a_4. *Si estos símbolos aparecen en el flujo de datos con igual probabilidad* $(= 0.25)$ *entonces la entropía de los datos es* $-4(0.25 \log_2 0.25) = 2$. *Dos es el mínimo numero de bits necesarios, en promedio, para representar cualquiera de los cuatro símbolos, así que podemos asignar el código de 2-bit* 00,01,10,11.

Ahora se considera que la probabilidad de ocurrencia de cada símbolo es distinta, $(0.49, 0.25, 0.25, 0.01)$ *si calculamos su entropía tenemos que es* $= 1.57$. *Así que tenemos que el número mínimo de bits necesarios, en promedio, para representar cada símbolo es* 1.57.

Símbolo	Prob.	Código1
a_1	.49	1
a_2	.25	01
a_3	.25	010
a_4	.01	001

Tabla 1.2: Códigos de longitud variable.

Si se vuelve a asignar a los símbolos el código de 2-bit se tiene que la redundancia sería igual a $R = -1.57 + \log_2 4 = 0.43$. *Si por el contrario asignamos el Código1 a los símbolos, el cual su entropía es* 1.77, *nos quedaría una entropía muy próxima al mínimo. En este caso su redundancia seria* $R = 1.77 - 1.57 = 0.2$ *bits por símbolo. Esto arroja como resultado, que con un flujo de datos lo suficientemente largo, el uso del Código1 sería menos redundante frente al código 2-bit y por lo tanto los datos estarían más compresos.*

Propiedad Prefijo

El uso de códigos de longitud variable conlleva varias desventajas. Una de las principales es la ambigüedad a la hora de descomprimir el código, es decir, que no haya duda posible sobre cuál es el siguiente símbolo a descodificar.

Como se puede observar en la Tabla 1.3, en el Código2 no está claro si en el mensaje

Símbolo	Código2	Código3
a_1	1	1
a_2	01	01
a_3	100	000
a_4	010	001

Tabla 1.3: Códigos ambiguos y no ambiguos.

a descodificar 1011000101 el resultado es 1|01|100|010|1 o en cambio 1|01|100|01|01, en el Código3 ese problema no ocurre ya que el mensaje es 1|01|1|000|1|01. Esto es debido a que el Código3 cumple la propiedad prefijo. Esta propiedad postula que una vez un determinado patrón de bits es identificado, ningún otro patrón puede comenzar de la misma forma. Por ejemplo, cuando un símbolo es asignado con 1 ningún otro símbolo podrá comenzar con 1 y en su lugar deberá hacerlo con 01. Como puede observarse, el Código2 no cumple esta propiedad mientras que el Código3 si lo hace. De este modo, al asignar un código de longitud variable, se deben seguir dos principios:

1. Asignar los códigos más cortos posibles a los símbolos de más frecuencia.
2. Cumplir la propiedad prefijo.

1.2.7 Codificación Huffman

Después de identificar cuáles son los principios a seguir en una codificación de longitud variable el problema que surge es encontrar un algoritmo que encuentre la mejor codificación posible. El algoritmo más comúnmente usado es Huffman [Huffman, 1952]. De forma general Huffman produce mejores códigos frente a otros algoritmos como Shannon-Fanon. Huffman producirá mejores códigos cuando la ocurrencia de los símbolos se aproxime lo máximo a potencias negativas de dos.

El algoritmo comienza construyendo una lista del alfabeto de símbolos de forma descendente en función de la probabilidad de ocurrencia. Después construye un árbol con cada símbolo descendente en las hojas desde el final al principio. Esto se realiza por pasos, donde en cada paso los símbolos con menor probabilidad son seleccionados,

añadidos a la parte alta del árbol, eliminados de la lista y remplazados por un símbolo auxiliar que los representa en la lista. Cuando la lista es reducida a un solo símbolo auxiliar (que representa a todo el alfabeto) significa que el árbol está completo. Después el árbol es recorrido para determinar el código de los símbolos asignando los códigos más cortos a las hojas con menos ramas.

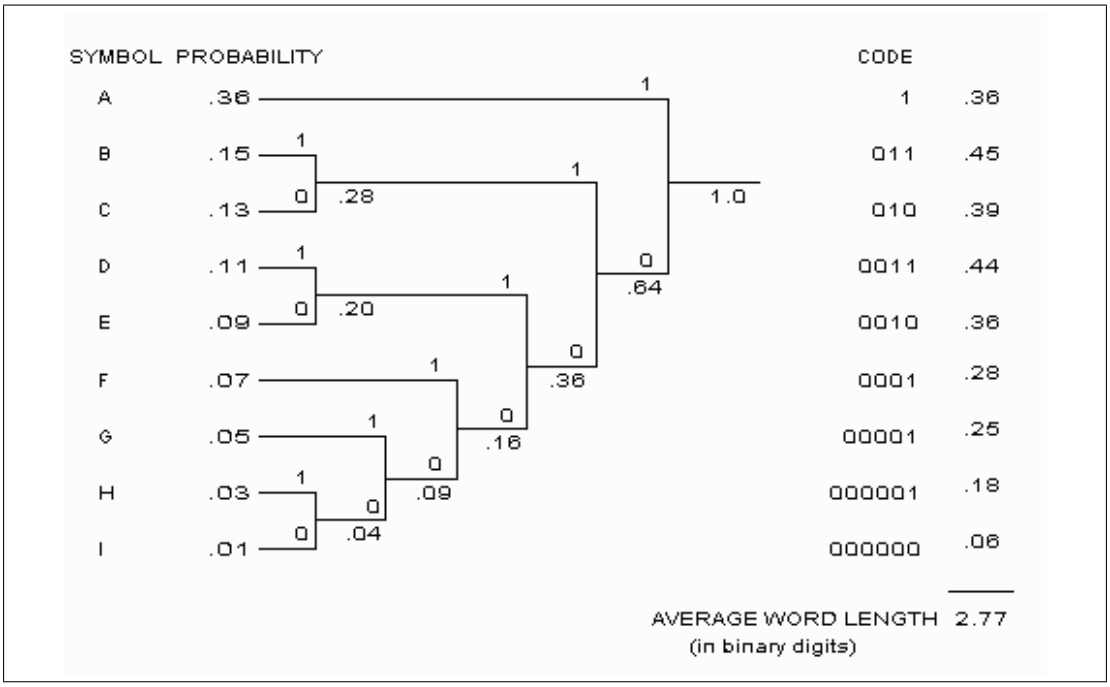


Figura 1.4: Ejemplo de un código Huffman y su árbol de construcción. Fuente: CCM.net

Pese a que la codificación Huffman garantiza el código más corto posible no siempre es utilizado ya que su uso acarrea otros costos, como el tiempo necesario para construir el árbol o la necesidad de conocer de antemano la distribución de los datos del fichero a tratar. Son estos impedimentos los que en algunas ocasiones Huffman no compensa frente a otros métodos, que bien pese a no encontrar el código más corto si se le aproxima.

Código Huffman Adaptativo

Una de las posibles soluciones a los problemas que tiene la codificación Huffman es el método adaptativo de este último [Knuth, 1985] [Vitter, 1987], por el cual no es necesario conocer de antemano la distribución de los símbolos a codificar.

Para el proceso, el árbol de símbolos estará vacío. Cada vez que un nuevo símbolo aparezca ocurrirán dos cosas. Primero, se escribirá en el stream de salida el símbolo en su forma sin comprimir, después, se añadirá al árbol y se le asignará un código. En cada iteración el árbol comprobará que sigue siendo un árbol de Huffman, en caso de no serlo, cambiará los códigos de los símbolos encontrados y continuará con el proceso. La próxima vez que encuentre un símbolo que esté en el árbol lo escribirá en el stream de salida con el código correspondiente del árbol. De este modo, es importante recalcar, que un mismo símbolo no tiene por que tener el mismo código asignado durante todo el proceso.

El proceso de descompresión funciona igual pero a la inversa. Comenzará con el árbol de símbolos vacío. Cada vez que encuentre un símbolo en su forma sin comprimir lo añadirá al árbol, aumentará en uno su ocurrencia en el fichero y le asignará un código. La próxima vez que se encuentre este código se descomprimirá. En cada iteración, el árbol sigue comprobando que sigue siendo un árbol de Huffman.

1.2.8 Códigos universales

Como alternativa a Huffman se pueden usar métodos en los cuales no es necesario conocer a priori la distribución de los datos, como los códigos universales. Un código universal es un código prefijo en el cual la probabilidad de aparición de un símbolo i es siempre mayor que cualquiera de sus siguientes, $P(i) \geq P(i+1)$. En general, todos los códigos universales asignan los códigos mas cortos a las símbolos de mayor ocurrencia.

Código de Golomb

El código de Golomb es un código universal óptimo para distribuciones de probabilidad geométricas [Golomb, 1966]. Se construye en función de dos cantidades (q, r) y un

numero arbitrario (b).

$$q = \left\lfloor \frac{n-1}{b} \right\rfloor, \qquad r = n - qb - 1$$

El código se divide en dos, la primera parte es el valor $q+1$ codificado en binario, la segunda es el valor r en binario con máximo numero de bits $\log_2 b$.

n:	1	2	3	4	5	6	7
$b=3$	0|0	0|10	0|11	10|0	10|10	10|11	110|0
$b=5$	0|00	0|01	0|100	0|101	10|110	10|00	10|01

Tabla 1.4: Códigos de Golomb para $b=3$ y $b=5$.

Código de Fibonacci

La codificación de Fibonacci [Fraenkel and Kleinb, 1996], es un código universal, prefijo que se basa en la bien conocida sucesión de Fibonnaci.

$$\begin{cases} f_0 = 0 \\ f_1 = 1 \\ f_n = f_{n-1} + f_{n-2} \end{cases} \tag{1.4}$$

El código aprovecha la propiedad de que la sucesión de Fibonacci es "completa", es decir, que cualquier natural $\mathbb{N}$ puede ser expresado como la suma de diferentes elementos de la sucesión de Fibonacci. Además también utiliza el teorema de Zeckendorf que postula que cualquier numero positivo puede ser expresado de forma única como la suma de uno o más números de Fibonacci de modo que la suma no incluya dos números de Fibonacci seguidos [Zeckendorf, 1972].

La codificación utiliza un bit a uno en una posición i para indicar que el elemento i-esimo+2 de la sucesión está presente en la suma final.

Ejemplo 3 *Tomando los elementos más a la derecha como los de menor cardinalidad, tendremos que el código*

0101

n:	Codificación
1	1
2	01
3	001
4	101
5	0001
6	1001
7	0101

Tabla 1.5: Primeros siete códigos de Fibonacci.

será descodificado de la siguiente manera:

$$f_5 \times 0 + f_4 \times 1 + f_3 \times 0 + f_2 \times 1 = \\ = f_4 + f_2 = 5 + 2 = 7$$

Aprovechando esta característica se puede usar un simple bit a 1 para indicar el final de símbolo ya que ningún símbolo va a contener nunca dos bits a 1 seguidos. Es importante destacar que esto solo ocurre cuando se colocan los bits de más peso a la izquierda. Colocando los bits de menor peso a la derecha tendremos que siempre el ultimo bit del símbolo es 1 y podremos por tanto usar otro bit a 1 para indicar su finalización.

Código de Lucas

La codificación de Lucas, es también un código universal, prefijo que se basa en una ligera modificación de la sucesión de Fibonacci.

$$\begin{cases} l_0 = 2 \\ l_1 = 1 \\ l_n = l_{n-1} + l_{n-2} \end{cases} \tag{1.5}$$

Como puede observarse, la definición de la sucesión es igual que la sucesión de Fibonacci (1.4) salvo por el valor del primer elemento, que es 2 en lugar de 0. Esto

hace que la sucesión tenga valores distintos pero siga teniendo las mismas propiedades, es decir, la sucesión de Lucas es "completa" y cumple el teorema de Zeckendorf, con lo cual se puede generar un código de similar funcionamiento al de Fibonacci pero que comprime usando valores distintos tal y como se ve en la tabla 1.6

n:	Codificación
1	01
2	1
3	001
4	0001
5	0101
6	1001
7	00001

Tabla 1.6: Primeros siete códigos de Lucas.

Teorema de Zeckendorf generalizado - Códigos Zeckendorf

Viendo que tanto el código de Fibonacci y el de Lucas mantienen las mismas propiedades y surgen de series casi idénticas, el siguiente paso lógico es pensar que existe alguna regla que defina una serie de sucesiones que cumplan estás características. Esta generalización del teorema de Zeckendorf ya fue demostrada [Hoggatt, 1972] y postula lo siguiente:

Definición 2 *Si se tiene una sucesión del tipo,*

$$\begin{cases} z_0 = 0 \\ z_1 = 1 \\ z_n = kz_{n-1} + z_{n-2} \end{cases} \tag{1.6}$$

entonces todo posible entero positivo N tiene una única representación de la forma

$$N = \epsilon_1 z_1 + \epsilon_2 z_2 + \cdots + \epsilon_n z_n$$

donde

$$\epsilon_1 = 0, 1, 2, 3, \ldots, \ o \ k-1$$

o alternativamente

$$\left.\begin{array}{l} \epsilon_1 = 0, 1, 2, 3, \ldots, \text{ o } k \\ \text{si } \epsilon_i = k, \text{ entonces } \epsilon_{i-1} = 0 \end{array}\right\} i \geq 2$$

Todas las sucesiones que sigan este tipo cumplirán la propiedad de Zeckendorf y por tanto podrán ser usadas para generar un código que pueda ser usado para comprimir. De hecho, la sucesión de Fibonacci es la primera de estas series de sucesiones, ya que es generada cuando $k = 1$.

Códigos Zeckendorf

El ultimo paso es definir un tipo de codificación general que satisfaga el teorema de Zeckendorf generalizado. Para ello hay que tener una serie de consideraciones; primero es necesario saber a priori el valor de k, ahora cada valor compreso no será representado necesariamente por un solo bit y por ultimo, el carácter de fin de palabra no será necesariamente un bit a 1.

Para la primera consideración, conocer el valor de antemano de k, parece razonable incluirlo en la extensión del archivo, ya que un valor de k distinto resultará en un código totalmente distinto. De este modo, un fichero llamado "texto.gzk6" sabremos que ha sido compreso utilizando la codificación de Zeckendorf con $k = 6$.

En cuanto a la segunda consideración, esta es, saber cuantos bits ocupa un valor compreso en el código de Zeckendorf. Pueden ser fácilmente calculados al tomar la parte entera del logaritmo en base dos del valor k mas uno, es decir $\lceil \log_2 k + 1 \rceil$.

Por ultimo, el carácter fin de palabra será una palabra de longitud $\lceil \log_2 k + 1 \rceil$ con todos sus valores a uno.

Como punto extra, para optimizar el código de Zeckendorf se ha ajustado la generalización para aprovechar al máximo sus propiedades quedando esta de la siguiente forma:

$$\begin{cases} z_0 = 1 \\ z_1 = k + z_0 \\ z_n = k z_{n-1} + z_{n-2} \end{cases} \tag{1.7}$$

De este modo se aprovecha el primer valor ya que en la codificación no se codifica el valor 0. Esta peculiaridad será explicada en detalle más adelante.

Ejemplo 4 *Usando el código de Zeckendorf con* $k = 5$ *tendremos un tamaño de palabra* $\lceil \log_2 6 \rceil = 3$ *con lo cual el código*

$$100|001$$

será descodificado de la siguiente manera:

$$z5_0 \times 100|z5_1 \times 001 = 1 \times 100|6 \times 001 =$$
$$1 \times 4 + 6 \times 1 =$$
$$= 4 + 6 = 10$$

Ejemplo 5 *Usando el código de Zeckendorf con* $k = 6$ *tendremos un tamaño de palabra* $\lceil \log_2 7 \rceil = 3$ *con lo cual el código*

$$001|011|001$$

será descodificado de la siguiente manera:

$$z6_0 \times 001|z6_1 \times 011|z6_2 \times 001 = z6_0 \times 1|z6_1 \times 3|z6_2 \times 1 =$$
$$1 \times 1 + 7 \times 3 + 43 \times 1 =$$
$$= 1 + 21 + 43 = 65$$

Por último, destacar, que para el caso de la compresión los códigos de Zeckendorf óptimos serán aquellos que aprovechen al máximo el ancho de banda de los valores disponibles, con lo cual los códigos óptimos de Zeckendorf serán aquellos que cumplan que $k = 2^a - 1$ con $a \in \mathbb{N}$.

Capítulo 2

Técnicas y Metodología

En este capítulo se desarrollará el grueso de proyecto, la definición del problema, el plan de trabajo, su implementación y los inconvenientes derivados de las decisiones tomadas así como se analizarán algunos de los resultados obtenidos que influyan en algunos de los aspectos anteriores.

2.1 Definición del problema

Tras haber asentado los aspectos teóricos necesarios ya es posible abordar el primer objetivo del proyecto, ubicar el algoritmo de compresión en las técnicas existentes. Se recuerda que el algoritmo a desarrollar se centra en comprimir exitosamente ficheros de texto plano en alfabeto latino. Llegados a este punto, es necesario analizar la estructura de estos ficheros. Existen diversos tipos de codificación que se usan actualmente para texto, los más utilizados son ASCII y Unicode, además de sus variantes.

1. ASCII: Incluido en 1963 en la lista de hitos de la IEEE es, desde entonces, el estándar de facto para todo fichero de texto. Usa 7 bits para codificar 128 caracteres de los cuales solo 95 son imprimibles.

2. ASCII Extendido: Pese a que el ASCII convencional era suficiente para la comunicación en inglés, la necesidad de incluir otros símbolos (sobretodo en los lenguajes latinos) hizo que tras la ISO/IEC 8859 se desarrolla este nuevo

código, que usa 8 bits para codificar y añade otros 96 caracteres imprimibles. Esta codificación es la que usan por defecto casi todos los editores de texto.

3. Unicode: Tras ISO/IEC 10646 se estandariza Unicode con sus variantes UTF-8,16 y 32 que usan respectivamente 8, 16 y 32 bits para codificar los símbolos, añadiendo con ellos un gran abanico de símbolos para prácticamente todos los idiomas. Es importante que destacar que UTF-8 y ASCII convencional tienen exactamente el mismo mapeado de símbolos.

Teniendo en cuenta que ASCII Extendido es usada por defecto en casi todos los editores de texto, es lógico pensar que la mayoría de ficheros usarán esta codificación, por tanto, se ha elegido ASCII Extendido como la codificación objetivo a comprimir para este proyecto.

Para tener a disposición una amplia cantidad de textos de prueba se ha utilizado la herramienta online Lipsum Generator (http://www.lipsum.com) que genera textos Lorem Ipsum[1] que utilizan exclusivamente caracteres latinos. De este modo podemos tener una cantidad ilimitada de ficheros de texto que siguen las pautas que queremos.

2.1.1 Características

Ahora que ya se ha definido la naturaleza del problema, es momento de analizar las características de este. Quizás la mas importante sea la distribución de los datos. Tal y como se puede ver en la Figura 2.2 la inmensa mayoría de los símbolos se localizan en una pequeña franja de valores, además, la frecuencia de aparición de los símbolos no es equitativa, tal y como se ve en la Figura 2.1.

Conociendo esta peculiaridad del problema y con lo expuesto en la introducción teórica parece, en primera instancia, que un acercamiento utilizando la codificación de Fibonacci es la más adecuada para obtener una compresión exitosa, ya que con un desplazamiento simple de los valores para acercarlos al cero la distribución pasaría a ser muy parecida a la óptima para el código.

[1] *Wikipedia: Lorem ipsum es el texto que se usa habitualmente en demostraciones de tipografías. El texto en sí no tiene sentido, aunque no es completamente aleatorio, sino que deriva de un texto de Cicerón en lengua latina, a cuyas palabras se les han eliminado sílabas o letras.*

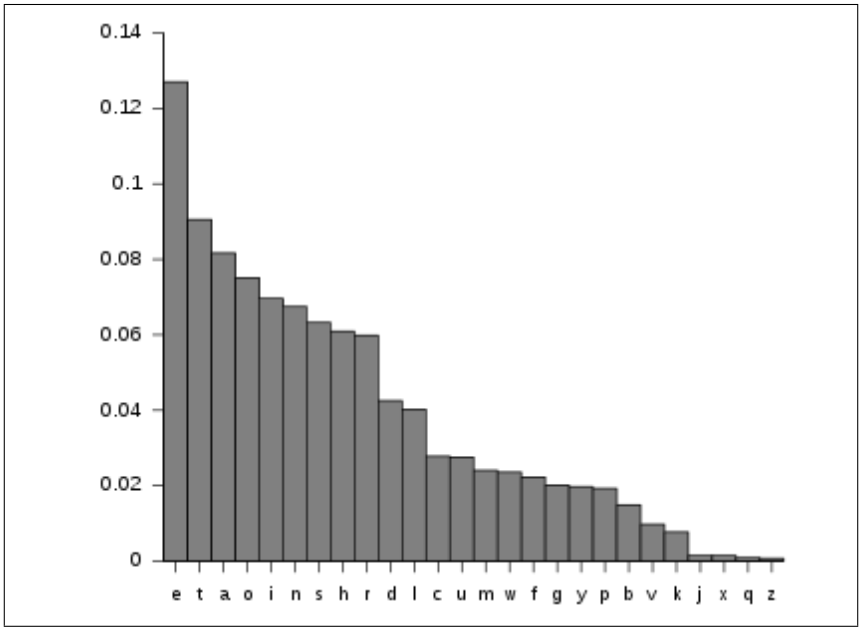


Figura 2.1: Frecuencia de aparición de las letras para un texto en inglés.

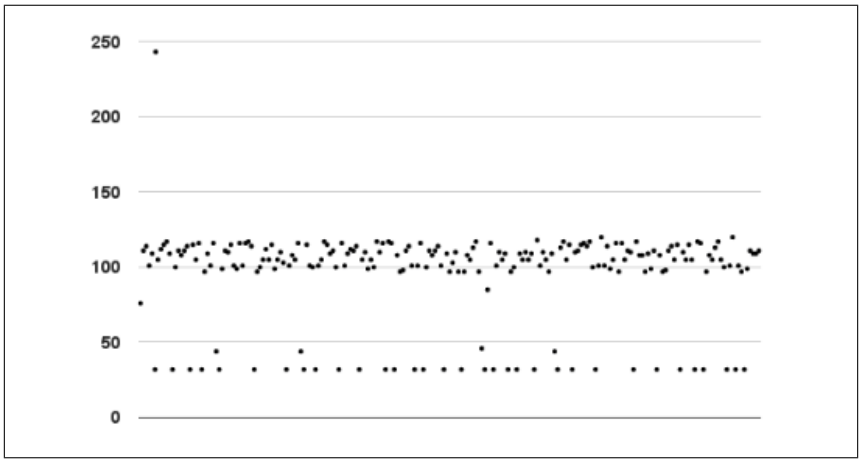


Figura 2.2: Distribución de caracteres para un texto Lorem Ipsum.

2.2 Plan de Trabajo

Para la correcta y óptima realización del trabajo es necesaria una buena planificación de este así como la mejor elección de tecnologías disponibles para el alumno.

2.2.1 Tecnologías

Sistema Operativo

Para comenzar se eligió GNU/Linux como el OS para el desarrollo, en concreto su distribución Debian 9 (Stretch), ya que el uso diario junto con el extenso conocimiento que el alumno tenia previamente del sistema permitía realizar el desarrollo de la forma mas rápida posible. Además en caso de la ocurrencia de cualquier problema, la extensa documentación disponible gratuitamente en internet hace de GNU/Linux, y en particular Debian, la opción más adecuada para el desarrollo de software para cualquier ingeniero.

Como añadido importante, la libre modificación de ficheros sin ningún tipo de restricción que las herramientas de GNU ofrece hace que para este proyecto en particular la elección de GNU/Linux sea la más idónea.

Lenguaje de Programación

Con la plataforma decidida se barajaron varios lenguajes de programación, todos conocidos de ante mano por el alumno. Como el fin ultimo del proyecto era desarrollar un algoritmo de compresión eficaz, se eligió Python 2.7 como lenguaje, ya que al ser interpretado, totalmente portable, contener su propia shell, no estar condicionado a usar un determinado IDE y poder ser ejecutado desde la shell predeterminada de Linux lo hacia perfecto para desarrollar un pequeño framework importable en otros programas de modo que el prototipado de los algoritmos y su modificación se hacia de la manera mas óptima y desacoplada posible para el alumno.

Aprovechando las características que la combinación que Python junto a la shell de Linux ofrecían, las pruebas de compresión se han realizado utilizando pequeños scripts escritos tanto en python como en bash.

IDE y control de versiones

Como IDE de desarrollo se ha utilizado el editor de textos VIM 8, y para el control de versiones una instancia privada de GitLab localizada en un sistema Raspberry propio.

2.3 Desarrollo del Proyecto

2.3.1 Hoja de ruta

La planificación del proyecto se realizó definiendo los hitos principales de manera secuencial. Cada hito principal contiene sub-hitos que no necesitan ser completados para poder continuar con la realización de otros hitos principales.

Hitos

1. Análisis y acercamiento teórico del problema.
   (a) Definición formal del algoritmo para la codificación de Fibonacci.
   (b) Definición formal del algoritmo para la codificación generalizada de Fibonacci.
2. Implementación del algoritmo 1.a)
   (a) Realización de pruebas.
   (b) Análisis de resultados.
3. Implementación del algoritmo 1.b)
   (a) Realización de pruebas.
   (b) Análisis de resultados.
4. Comparación de los resultados 2.b) y 3.b)
   (a) Analizar posibles mejoras básicas de los algoritmos e implementarlas.

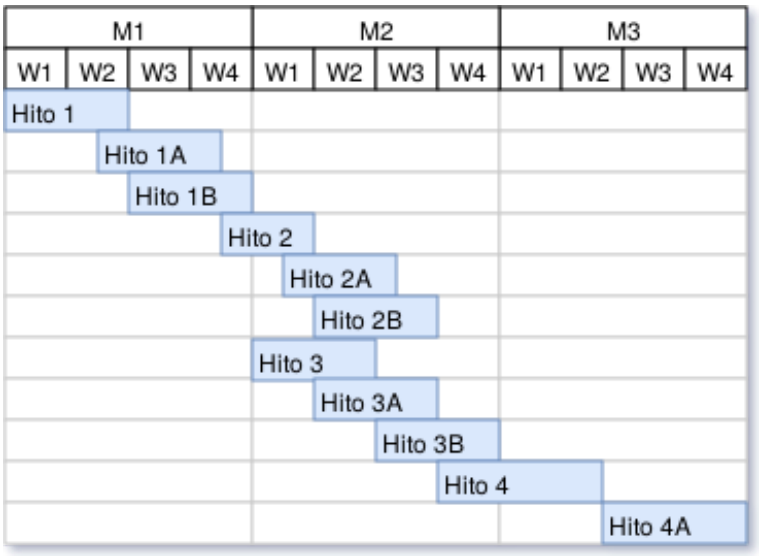


Figura 2.3: Hoja de ruta del PFG.

2.3.2 Definición de los Algoritmos

Siguiendo con lo establecido en la hoja de ruta, el Hito 1 ya ha sido explicado en la introducción en su totalidad, de modo que ahora se procederá a la explicación de los hitos 1A y 1B.

Definición del algoritmo para la codificación de Fibonacci

Para el primer algoritmo será necesario el uso de dos funciones elementales, NFibonacci(n) que retorna el numero n en términos de Fibonacci y compressNumAsFibo(n) que retorna el valor compreso en código de Fibonacci.

La primera instancia del algoritmo calcula y guarda en memoria los números de Fibonacci menores o iguales de n. Después,se irá recorriendo la lista retornando los indices de los valores de la representación de Zeckendorf que corresponde al valor n dado.

Algorithm 1 NFibonacci(n)

```
1: descomposition := fibonacci(n)
2: indexes := ∅
3: for f in descomposition do
4:     if f ≤ n then
5:         n := n − f
6:         indexes.add(descomposition.index(f))
7:     end if
8: end for
9: return indexes
```

Algorithm 2 compressNumAsFibo(n)

```
1: indexes := NFibonacci(n)
2: compressedByte := ∅
3: for i in range(0, indexes[−1] + 1) do
4:     if i in indexes then
5:         compressedByte.add(1)
6:     else
7:         compressedByte.add(0)
8:     end if
9: end for
10: return compressedByte
```

Una vez que se tiene a disposición el algoritmo 1 la definición del algoritmo de generación del código final es trivial. Como puede verse en el algoritmo 2 simplemente hay que recorrer la lista de índices e ir escribiendo 0 o 1 en función de la aparición del elemento en la lista para más tarde retornar el numero ya codificado.

2.3.3 Implementación

Ahora que ya disponemos de la definición formal de los algoritmos para compresión, ya podemos implementarlos en Python 2.7, aprovechando todas las herramientas y

características que este nos ofrece. La primera de todas es el paquete itertools, del que utilizaremos la función *takewhile* para codificar la función $fibonacci()$ necesaria para el algoritmo 1.

```
from itertools import takewhile
import sys

def __fibonacci(first, second):
    a, b = first, second
    while True:
        yield b
        a, b = b, a + b

#get the given number n in terms of fibonacci numbers indexs
def NFibonacci(n):
    descomposition = list(reversed(list(takewhile(lambda f: f <= n, __fibonacci(0,1)))))
    for f in descomposition:
        if f <= n:
            n -= f
            yield list(reversed(descomposition)).index(f)
```

Para ahorrar tiempo de computo la implementación utiliza otra característica de Python, los generadores. De este modo se puede codificar un método que devuelve un objeto generador, fácilmente transformable en lista, en lugar de un solo elemento por ejecución, lo que permite en el futuro ahorrar mucho tiempo de computo y memoria al programa.

```
#get the given number as a byte compressed representation
#the compress matrix is [1, 2, 3, 5, 8, ...]
def compressNumAsFibo(n):
    indexes = list(reversed(map(lambda x: x - 1 if x>1 else x, NFibonacci(n))))
    compressedByte = ""
    for i in range(indexes[-1]+1):
        if i in indexes:
            compressedByte += '1'
        else:
            compressedByte += '0'
    return compressedByte
```

Tal y como puede verse en la implementación del algoritmo 2, el uso de generadores permite muy fácilmente modificar el objeto que devuelve NFibonacci(n) y adaptarlo para que siga el funcionamiento esperado del código de Fibonacci, tal como se muestra en el ejemplo 5.

Al realizar estos cambios incluimos ciertas mejoras al código, la primera, simplificamos la sucesión de Fibonacci al hacer que comience por 1 y 2, en otras palabras, retiramos la redundancia de los primeros elementos. La segunda mejora viene dada en parte por la primera, al haber reducido los valores iniciales que se repetían en la sucesión se puede aprovechar mejor los siguientes valores, con lo cual el tamaño resultante del fichero compreso será incluso menor que si no usáramos estos cambios.

Implementación de la codificación de un archivo

El siguiente paso es implementar un método que comprima un determinado archivo. En este paso no es necesario mostrar la codificación entera, pero si lo es tratar algunos temas concernientes, en cualquier caso, el código completo está disponible en el anexo de este documento.

El primer aspecto a mencionar es la limitación que tiene python 2.7 para tratar y operar con estructuras de datos basadas en bytes. Esta versión del lenguaje no ofrece tratamiento nativo de bytes con lo cual no se le puede ordenar a python que escriba en disco un valor concreto. Con está limitación en mente se ha optado por una solución alternativa. Esta solución pasa por realizar todas las operaciones aritméticas de forma ordinal, es decir, tratar los valores en modulo 10, y más tarde, cuando esos valores decimales vayan a ser escritos en disco ordenar a la librería de python que escriba en disco el carácter ASCII-8 correspondiente a ese valor. De este modo, la escritura en disco se realiza correctamente al guardar el valor ASCII-8 que corresponde con el valor decimal que queremos guardar.

Otro de los aspectos importantes es el valor máximo a comprimir que se incluye en la implementación del método a propósito, así como la unidad que se aumenta siempre en todo valor al ser compreso.

```
[...]
compressedBytes += compressNumAsFibo(((ord(byte)-valueFit)%maxValueToCompress)+1)
[...]
```

Tal y como se ve en la linea de código de encima, a todo valor a comprimir se le aplica la operación modulo $maxValueToCompress$. El valor por defecto de esta variable es 255, esto es incluido por varias razones. La principal y mas importante es para evitar símbolos indeseados que se hayan podido incluir en el fichero a comprimir por error. Ya que en un principio el algoritmo está pensado para comprimir códigos ASCII-8 no tiene sentido encontrarse ningún símbolo que se codifique con un valor mayor. Al incluir esta simple operación estamos asegurándonos de que en ningún caso el código compreso resultante pueda ser corrompido por ningún valor indeseado. La otra razón de peso para incluir esta operación de modulo es para, en primer lugar, aumentar la velocidad de descompresión al no tener que operar con códigos más largos de lo normal, y la segunda, acotar la longitud máxima de un símbolo al comprimirse. La unidad que se aumenta a todo valor es consecuencia de los cambios realizados antes. Al haber reducido los primeros elementos de la sucesión de Fibonacci hemos eliminado con ellos el valor cero, con lo cual, este no puede ser compreso. Esta limitación que en principio podría parecer muy grave es fácilmente solucionable aumentando en uno todos los valores a comprimir.

Por ultimo, es muy importante mencionar el uso de la variable *valueFit* que se encarga de desplazar toda la distribución de valores del fichero a comprimir. Si se observa la figura 2.2 podemos observar que la mayoría de los símbolos empiezan a aparecer a partir del valor 100. Esta característica no es nada favorable a la codificación que queremos usar, con lo que una primera y rápida solución podría ser restarle a todos los valores 99 unidades o 100 unidades. Sin embargo, y tras muchas pruebas se ha llegado a la conclusión experimentalmente de que el valor optimo para este tipo de ajuste es 97 tal y como puede observarse en la tabla 2.1. Sin embargo, este valor es solo válido para los textos Lorem Ipsum. Para cualquier otro idioma que tenga una distribución distinta de sus caracteres este *valueFit* tendrá presumiblemente un valor distinto.

valueFit	size compressed (bytes)	size uncompressed (bytes)
97	28295	31404
96	29182	31404
99	29723	31404
95	29876	31404
98	30142	31404
94	30453	31404
100	30562	31404

Tabla 2.1: Relación de compresión en función del valor de valueFit.

Implementación de la decodificación de un archivo

Por ultimo, queda pendiente la implementación del método que nos devuelva el fichero compreso a su estado inicial. El proceso a seguir es el inverso al realizado en la compresión. Para ello, y debido a las limitaciones de python 2.7, el método toma todos los valores en ASCII-8 del fichero por orden y los transforma de nuevo a bytes, almacenando en un buffer todo el fichero compreso. Una vez ya se ha cargado en memoria todo el fichero, se comienza a analizar los valores. Se guardarán en una variable todos los bits leídos de forma consecutiva hasta que aparezcan dos bits a 1 seguidos. En ese momento el valor almacenado en la variable se descomprimirá obteniendo de él el valor ordinal. Después este valor será utilizado para guardar en otra variable final el valor en ASCII-8 correspondiente. El proceso se repite hasta que todo el buffer es descodificado. Por ultimo, se escriben en disco los valores ASCII-8 obtenidos.

```
[...]
decompressedFile+=chr(((decompressedNum+valueFit-1) % maxValueToDecompress))
[...]
```

Es importante destacar, tal como se ve en el extracto del código, que para que se pueda descomprimir el archivo es necesario conocer de antemano el valueFit utilizado para comprimir. Para no tener que almacenarlo dentro del fichero compreso se ha

decidido guardarlo como parte de la extensión del fichero, en caso de no aparecer en la extensión se asume que usa el valor por defecto 97.

Implementación para el código de Lucas

Debido a la similitud con el código de Fibonacci, la implementación del código de Lucas es muy parecida. Por esto mismo solo se mencionará los pequeños detalles importantes en los que difieren ambas implementaciones.

El más destacable de todos es el relativo a la generación de los valores de la sucesión. Como la sucesión de Fibonacci se ha recortado en sus primeros valores para ganar rendimiento, la reutilización del mismo código para el código de Lucas acarrea un coste extra, el primer valor nunca es incluido en la generación de los valores de la sucesión. Debido a esto, siempre se tiene que añadir explícitamente el primer valor 2 a la sucesión (línea 3).

```
1  def NLucas(n):
2      descomposition = list(reversed(list(takewhile(lambda f: f <= n, __fibonacci(2,1)))))
3      descomposition.append(2)
4      for f in descomposition:
5          if f <= n:
6              if n == 2 :
7                  n-=f
8                  yield 0
9              else:
10                 n -= f
11                 yield list(reversed(descomposition)).index(f)
```

Además, es necesario añadir estructuras de control extra para gestionar la lógica de este primer valor, tal y como se ve en las líneas 6-8 del método de encima.

El otro aspecto destacable de la implementación es el relativo al valueFit tanto de compresión como de descompresión en los códigos de Lucas. De forma análoga al código de Fibonacci, se ha demostrado que el valor óptimo de ajuste en el código de Lucas para ficheros de texto en ingles codificados en ASCII-8 es 96. Es una característica a destacar ya que las diferencias de pendiente en los valores consecutivos de cada sucesión

(Tabla 2.2) hacen que una distribución media centrada una unidad menos respecto a la correspondiente para el código de Fibonacci, sea la más óptima para estos casos.

n:	0	1	2	3	4	5	6	7	8	9	10	11	12
Lucas	2	1	3	4	7	11	18	29	47	76	123	199	322
Fibonacci	1	1	2	3	5	8	13	21	34	55	89	144	233

Tabla 2.2: Primeros valores de Lucas y Fibonacci.

Implementación para el código de Zeckendorf

Si bien a simple vista, la codificación para el código de Zeckendorf parece mucho más compleja que la de Lucas o Fibonacci, lo cierto es que con un poco de análisis es fácil ver que es muy similar a la codificación de Fibonacci,salvo en que en esta ocasión se ha tenido que generalizar prácticamente todas las operaciones añadiéndolas un factor extra. De modo que si se compara esta codificación con la de Fibonnacci se puede ver que siguen el mismo patrón.

Por este motivo, y pese a que los recorridos por los arrays si son algo más engorrosos, no me parece adecuado repetir la misma explicación para este apartado ya que sigue la misma lógica que el apartado para Fibonnacci. Por poner algún ejemplo:

```
1  def compressNumAsGZeck(k,n):
2      indexes = list(reversed(list(__GZeck(k,n))))
3      compressedByte = ""
4      numBitsPerIndex = int(math.ceil(math.log(k+1,2)))
5      for i in range(indexes[-1][0]+1):
6          indexesIndex = list(indexes[x][0] for x in range(len(indexes)))
7          if i in indexesIndex:
8              compressedByte += format(indexes[indexesIndex.index(i)][1],
9                                       '0'+str(numBitsPerIndex)+'b')
10         else:
11             compressedByte += format(0,'0'+str(numBitsPerIndex)+'b')
12     return compressedByte
```

```
def compressNumAsFibo(n):
    indexes = list(reversed(map(lambda x: x - 1 if x>1 else x, __NFibonacci(n))))
    compressedByte = ""
    for i in range(indexes[-1]+1):
        if i in indexes:
            compressedByte += '1'
        else:
            compressedByte += '0'
    return compressedByte
```

En el anexo puede analizarse linea por linea ambas codificaciones enteras y por tanto ver su similitud.

Capítulo 3

Test de compresión y resultados

En este ultimo capítulo se explicará la metodología seguida para poder testear los algoritmos desarrollados así como la discusión y análisis de los resultados obtenidos.

3.1 Análisis y definición de los test

Para poder probar correctamente, de una manera minuciosa y a la vez poder comprender las limitaciones de las distintas codificaciones implementadas se han recogido una serie de ficheros de texto en diferentes lenguajes tomados de los principales periódicos digitales de países donde se hablan de forma nativa estas lenguas.

De este modo, se puede asegurar de forma empírica que la distribución que siguen los diferentes ficheros es la correspondiente a la dada en sus respectivos lenguajes, tal y como puede observarse en las figuras 3.1. Las características de los ficheros usados pueden observarse en la tabla 3.1.

3.1.1 Análisis de las distribuciones

Como puede verse, la distribución de los caracteres varía en función del lenguaje en el que sean escritos, así, por ejemplo, tanto la distribución del inglés como la del Lorem Ipsum es prácticamente igual, mientras que tanto el francés, el español y el alemán contienen mas caracteres dispersos en torno a la franja de 160-200 y 60-95.

Esto parece lógico ya que estos lenguajes contienen caracteres especiales que solo

usan ellos, como las vocales con tildes o el carácter scharfes (ß) del alemán, aunque no se usan con la misma frecuencia en todos los idiomas, lo que nos deja con que aunque los textos en estos lenguajes si contienen valores en esas franjas la funciones de densidad varían de un lenguaje a otro. También es importante destacar que tanto inglés, francés, alemán y español al ser lenguajes de origen similar, el grueso del uso de sus caracteres (las zonas donde su función de densidad se maximiza) es el mismo siempre, es decir, en la franja de valores de 100-127.

Por último, es importante destacar que el ruso no sigue ninguna de las pautas comentadas anteriormente ya que como se puede ver a simple vista, al usar el cirílico como alfabeto la distribución de los caracteres y su función de densidad es totalmente distinta, lo que nos lleva a pensar que estas características deberían afectar de algún modo reseñable a la compresión.

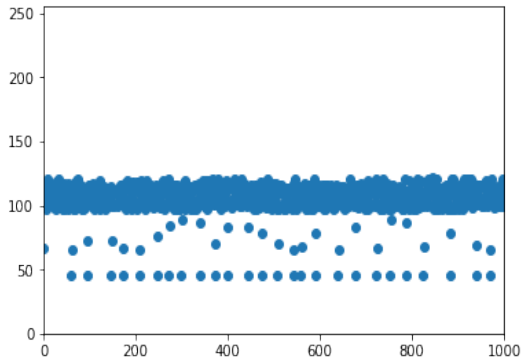
3.1.2 Automatización de las pruebas

Con el fin de agilizar y sistematizar el comprobación de resultados y evitar así posibles errores humanos, se ha escrito un script que comprime todos los archivos de texto en la codificación de; Fibonacci, Lucas, Zeckendorf Gen. con $k = 2$, $k = 3$, $k = 5$, $k = 7$ y $k = 15$ así como en formato GNU/Zip y el formato de código privativo winRAR. De este modo se intenta conseguir una muestra representativa de las capacidades compresoras de los algoritmos implementados y a la vez compararlos con los dos sistemas de compresión más utilizados en la actualidad.

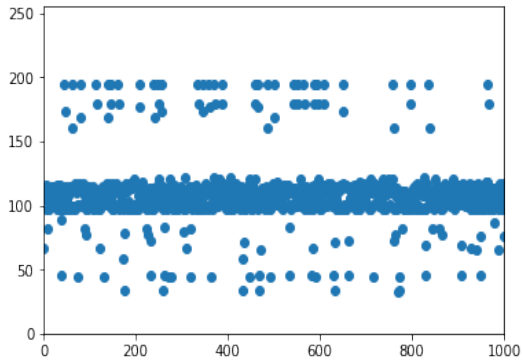
Para la codificación de Fibonacci, Lucas y Zeckendorf Gen. se ha utilizado siempre el mismo valor de *valuefit*. Esta decisión que a priori podría parecer no muy importante, en realidad nos arroja unos resultados muy esclarecedores sobre las limitaciones y bondades de estos tipos de códigos, que pasaremos a discutir a continuación.

3.2 Análisis de los resultados por idioma

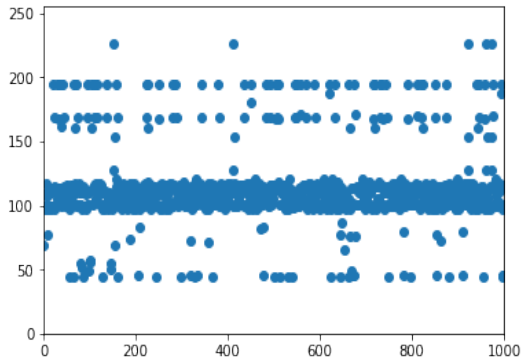
Como hemos visto en la sección anterior, cada lenguaje es distinto. Debido a esto la compresión en cada lenguaje será distinta, por ello pasaremos a discutir cada lenguaje por separado. Es de destacar antes de nada, que en muy pocos casos la compresión



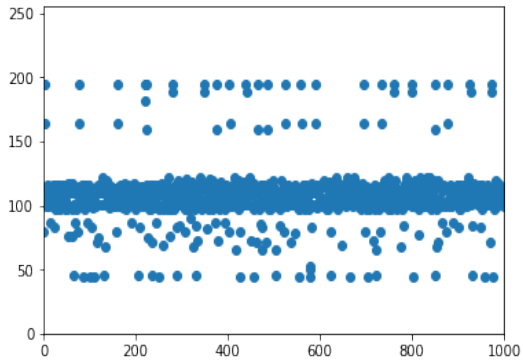
(a) Inglés



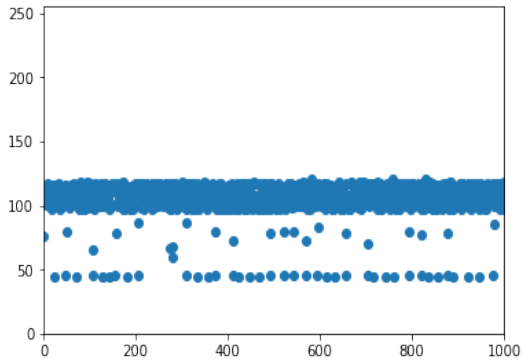
(b) Español



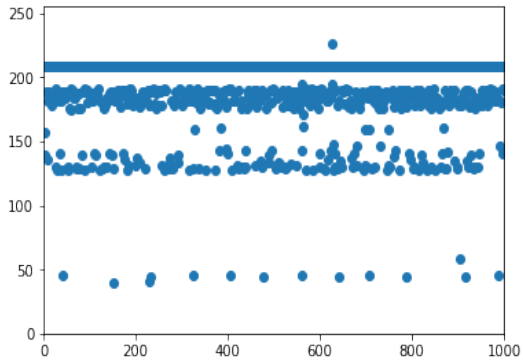
(c) Francés



(d) Alemán



(e) Lorem Ipsum



(f) Ruso

Figura 3.1: Distribuciones de caracteres de los diferentes ficheros de prueba.

Idioma	ID Fichero	#caracteres	#letras	Tamaño en disco
	i1	3334	2787	3334 bytes
	i2	4097	3423	4097 bytes
	i3	3893	3281	3893 bytes
Inglés	i4	33972	28465	33972 bytes
	e1	3973	3293	3973 bytes
	e2	3583	2995	3583 bytes
	e3	4828	4037	4828 bytes
Español	e4	37152	30967	37152 bytes
	f1	2367	1968	2367 bytes
	f2	2914	2466	2914 bytes
	f3	2279	1935	2279 bytes
Francés	f4	22680	19099	22680 bytes
	a1	3097	2664	3097 bytes
	a2	1895	1638	1895 bytes
	a3	2163	1843	2163 bytes
Alemán	a4	21465	18427	21465 bytes
	r1	9956	9186	9956 bytes
Ruso	r2	78000	71120	78000 bytes

Tabla 3.1: Características de los ficheros de test.

de Zeckendorf ha conseguido mejora alguna en la compresión llegando a aumentar el tamaño del fichero resultante en casi todos los casos o incluso a doblar el tamaño original en casos como el ruso. La razón de este motivo y la propuesta de una posible solución es presentada en las conclusiones.

3.2.1 Inglés

La lengua de Shakespeare es la lengua menos compleja en cuanto a distribución de valores de las lenguas analizadas, es por ello que observamos el mayor ratio de

compresión de todos, en torno al 10% con Fibonacci y 6% con Lucas. En cuanto a Zeckendorf generalizado siempre hay perdida de espacio excepto en $k = 3$. Pese a ello, la variación del valor k hace que los códigos generados sean algunos más óptimos que otros, quedando en este orden de mejor a peor $k3 > k7 > k2 > k5 > k15$. En cuanto a los formatos de compresión gzip y winRAR obtienen valores de compresión muy parecidos, con ratios en torno al 45% de mejora en ficheros pequeños y 85% en cuanto a ficheros más grandes. Es importante mencionar que pese a que los algoritmos de compresión mejoran su rendimiento mientras más cantidad de datos comprimen, el ratio de compresión de los algoritmos desarrollados se mantienen siempre constantes.

Id Fichero	Original	Fibonacci	Lucas	Gzeck2	Gzeck3	Gzeck5	Gzeck7	Gzeck15	gzip	winRAR
i1	3334	2991 (+0.103)	3133 (+0.060)	3820 (-0.146)	3316 (+0.005)	4110 (-0.233)	3695 (-0.108)	4195 (-0.258)	1757 (+0.473)	3700 (-0.110)
i2	4097	3698 (+0.097)	3873 (+0.055)	4711 (-0.150)	4094 (+0.001)	5049 (-0.232)	4557 (-0.112)	5185 (-0.266)	2136 (+0.479)	2254 (+0.450)
i3	3893	3469 (+0.109)	3639 (+0.065)	4434 (-0.139)	3849 (+0.011)	4771 (-0.226)	4279 (-0.099)	4892 (-0.257)	2012 (+0.483)	2127 (+0.454)
i4	33972	30471 (+0.103)	31933 (+0.060)	38893 (-0.145)	33775 (+0.006)	41786 (-0.230)	37591 (-0.107)	42815 (-0.260)	5507 (+0.838)	5478 (+0.839)

Tabla 3.2: Tamaño (en bytes) de los ficheros en inglés compresos. Entre paréntesis la ganancia de espacio.

3.2.2 Español

La lengua de Cervantes comienza a ser la primera que tratamos que contiene una distribución de caracteres mas compleja, añade seis nuevos caracteres (á, é, í, ó, ú, ñ) lo que degrada un poco la compresión, pasando esta a ser en torno al 9% con Fibonacci y 5% con Lucas. En cuanto a Zeckendorf generalizado se mantiene la calidad de los códigos con $k3 > k7 > k2 > k5 > k15$. Gzip y winRAR siguen manteniendo las mismas características y ratios.

Parece destacable el hecho de que pese a contener caracteres extra que a priori podrían hacer pensar que el ratio de compresión se vería afectado de una forma significativa esto no es así. La explicación se da en el valor que adquieren estos caracteres, en la franja de 225-250. Ya que estos caracteres están por encima del *valuefit* usado (97) su repercusión no será muy notable.

Id Fichero	Original	Fibonacci	Lucas	Gzeck2	Gzeck3	Gzeck5	Gzeck7	Gzeck15	gzip	winRAR
e1	3973	3590 (+0.096)	3760 (+0.054)	4576 (-0.152)	4002 (-0.007)	4916 (-0.237)	4439 (-0.117)	5062 (-0.274)	1924 (+0.516)	2032 (+0.489)
e2	3583	3244 (+0.095)	3398 (+0.052)	4133 (-0.154)	3608 (-0.007)	4445 (-0.241)	4011 (-0.119)	4559 (-0.272)	1764 (+0.508)	1874 (+0.477)
e3	4828	4368 (+0.095)	4567 (+0.054)	5558 (-0.151)	4856 (-0.006)	5975 (-0.238)	5399 (-0.118)	6130 (-0.270)	2109 (+0.563)	2215 (+0.541)
e4	37152	33605 (+0.095)	35175 (+0.053)	42799 (-0.152)	37396 (-0.007)	46004 (-0.238)	41544 (-0.118)	47250 (-0.272)	5661 (+0.848)	5587 (+0.850)

Tabla 3.3: Tamaño (en bytes) de los ficheros en español compresos. Entre paréntesis la ganancia de espacio.

3.2.3 Francés

La lengua de Victor Hugo empieza a complicar aun más la compresión al añadir dieciséis caracteres al abanico original (à, â, æ, ç, é, è, ê, ë, î, ï, ô, œ, ù, û, ü, ÿ) lo que degrada significativamente la compresión dejando los valores de ratio en torno al 5%para Fibonacci y un 0.4% para Lucas. Zeckendorf generalizado sigue aumentado el tamaño original y mantiene el orden en la calidad de los códigos generados $k3 > k7 > k2 > k5 > k15$. En cuanto a Gzip y winRAR el ratio de compresión se ve significativamente menguado en archivos pequeños (45% gzip y 40% winRAR) pero manteniéndose alto cuando se dispone de mayor cantidad de información.

Esta disminución tan significativa para la codificación de Fibonnaci y Lucas es producida por la misma razón que el español, los caracteres extra están incluidos en la franja 224-255 lo que junto al hecho de que estos caracteres se usan más a menudo que en el español desemboque en una perdida mayor respecto a este lenguaje.

Id Fichero	Original	Fibonacci	Lucas	Gzeck2	Gzeck3	Gzeck5	Gzeck7	Gzeck15	gzip	winRAR
f1	2367	2259 (+0.046)	2358 (+0.004)	2857 (-0.207)	2461 (-0.040)	3037 (-0.283)	2717 (-0.148)	3061 (-0.293)	1331 (+0.438)	1439 (+0.392)
f2	2914	2759 (+0.053)	2887 (+0.009)	3496 (-0.200)	3006 (-0.032)	3704 (-0.271)	3349 (-0.149)	3758 (-0.290)	1444 (+0.504)	1553 (+0.467)
f3	2279	2183 (+0.042)	2274 (+0.002)	2756 (-0.209)	2369 (-0.039)	2910 (-0.277)	2627 (-0.153)	2946 (-0.293)	1188 (+0.479)	1304 (+0.428)
f4	22680	21597 (+0.048)	22551 (+0.006)	27322 (-0.205)	23507 (-0.036)	28947 (-0.276)	26077 (-0.150)	29292 (-0.292)	3816 (+0.832)	3771 (+0.834)

Tabla 3.4: Tamaño (en bytes) de los ficheros en francés compresos. Entre paréntesis la ganancia de espacio.

3.2.4 Alemán

La lengua de Goethe es muy similar a la española. Solamente introduce cuatro caracteres nuevos (ä, ë, ï, ß) pero por desgracia hace uso de un mayor uso de caracteres capitales que están por debajo del *valuefit* utilizado (97) lo que repercute en una perdida respecto al español, dejando esta en un 8% para Fibonacci y un 3% para Lucas. La calidad de los códigos generalizados de Zeckendorf cambia ligeramente quedando $k3 > k7 > k2 > k15 > k5$ y tanto gzip como winRAR siguen aumentando sus ratios de compresión mientras más datos comprimen.

Id Fichero	Original	Fibonacci	Lucas	Gzeck2	Gzeck3	Gzeck5	Gzeck7	Gzeck15	gzip	winRAR
g1	3097	2827 (+0.087)	2961 (+0.044)	3597 (-0.161)	3097 (0.000)	3875 (-0.251)	3458 (-0.117)	3868 (-0.249)	1569 (+0.493)	1679 (+0.458)
g2	1895	1739 (+0.082)	1825 (+0.037)	2212 (-0.167)	1905 (-0.005)	2382 (-0.257)	2123 (-0.120)	2375 (-0.253)	1026 (+0.459)	1139 (+0.399)
g3	2163	2015 (+0.068)	2106 (+0.026)	2552 (-0.180)	2191 (-0.013)	2733 (-0.264)	2444 (-0.130)	2725 (-0.260)	1138 (+0.474)	1247 (+0.423)
g4	21465	19740 (+0.080)	20672 (+0.037)	25080 (-0.168)	21576 (-0.005)	26968 (-0.256)	24072 (-0.121)	26900 (-0.253)	3351 (+0.844)	3315 (+0.846)

Tabla 3.5: Tamaño (en bytes) de los ficheros en alemán compresos. Entre paréntesis la ganancia de espacio.

3.2.5 Ruso

La lengua de Dostoevsky es la más dispar de todas las presentadas. Al usar el cirílico ruso como alfabeto con 33 caracteres distintos estos son mapeados internamente por python para tener una representación en ascii-8. Este mapeo nos permite seguir usando la compresión utilizada hasta ahora sin ninguna limitación pero con un problema añadido, los valores están ubicados en franjas totalmente diferentes a los alfabetos latinos tal y como puede verse en la figura 3.1(f). Este mapeo tan distinto hace que todos los códigos codifiquen de un modo que su fichero resultante ocupa bastante más que el original, tal y como se ve en la tabla 3.6.

Por suerte, un pequeño ajuste en el valor del *valuefit* cambiándolo de 97 a 180 nos arroja unos resultados bastante distintos, tal y como se ve en la tabla 3.7 que nos muestra una notable mejoría en el tamaño que aumentan los ficheros de 32% en Fibonacci a solamente un 6% y de un 35% a un 14% en Lucas. Para Zeckendorf generalizado la disminución también es notoria y en cuanto a gzip y winRAR podemos

Id Fichero	Original	Fibonacci	Lucas	Gzeck2	Gzeck3	Gzeck5	Gzeck7	Gzeck15	gzip	winRAR
r1	9956	13171 (-0.323)	13481 (-0.354)	16032 (-0.610)	12405 (-0.246)	15279 (-0.535)	14400 (-0.446)	14934 (-0.500)	3544 (+0.644)	3598 (+0.639)
r2	78000	102807 (-0.318)	105344 (-0.351)	124817 (-0.600)	96614 (-0.239)	120142 (-0.540)	112215 (-0.439)	117000 (-0.500)	13318 (+0.829)	12557 (+0.839)

Tabla 3.6: Tamaño (en bytes) de los ficheros en ruso compresos usando un valueFit de 97.

Id Fichero	Original	Fibonacci	Lucas	Gzeck2	Gzeck3	Gzeck5	Gzeck7	Gzeck15	gzip	winRAR
r1	9956	10567 (-0.061)	11382 (-0.143)	13161 (-0.322)	10536 (-0.058)	12797 (-0.285)	11910 (-0.196)	14066 (-0.413)	3544 (+0.644)	3598 (+0.639)
r2	78000	83071 (-0.065)	89070 (-0.142)	103048 (-0.321)	82875 (-0.062)	100233 (-0.285)	93782 (-0.202)	110230 (-0.413)	13318 (+0.829)	12557 (+0.839)

Tabla 3.7: Tamaño (en bytes) de los ficheros en ruso compresos usando un valueFit de 180.

observar que la compresión es exactamente la misma, ya que las transformaciones que usan internamente son mucho más complejas que un simple desplazamiento de los valores.

Capítulo 4

Conclusiones, impacto social y trabajo futuro

4.1 Conclusiones

Tal y como se ha podido analizar tras la compresión de múltiples ficheros de múltiples idiomas distintos, el rendimiento de los algoritmos esta fuertemente ligado a las distribuciones y funciones de densidad de los lenguajes a comprimir. Ya que tanto el código de Fibonacci, Lucas y Zeckendorf generalizado es universal, estos no tienen en cuenta distribución alguna y sientan sus bases en una suposiciones y premisas previas. Es el algoritmo desarrollado, a través del refinamiento del valueFit lo que permite que una reducción del tamaño usado sea exitosa o no.

Por tanto, para este proyecto no se considera tan importante el hecho de conseguir una mejor o peor compresión como el hecho de que sea el ratio constante en función de la distribución de los caracteres dados. Como se ha podido observar, los formatos de compresión usados en la actualidad mejoran su eficiencia mientras más datos tienen disponibles, esto es debido a que no usan lenguajes universales de la misma clase que hemos usado nosotros.

Por ultimo, me parece importante destacar que si el objetivo final es obtener la mejor compresión posible usando la codificación de Fibonacci o Lucas es vital desarrollar un método que transponga los valores a comprimir de una forma que sea aprovechada

por estos códigos de forma optima. Del mismo modo, es importante estudiar los códigos que pueden ser generados mediante la generalización de Zeckendorf, que propiedades tienen y cual es la mejor forma de aprovecharlos ya que como se ha podido observar en las pruebas de compresión, hay indicios que indican que pueden existir códigos de Zeckendorf más óptimos que otros según el tipo de fichero a comprimir.

4.2 Impacto social y ambiental y responsabilidad ética y profesional

El impacto ambiental de este proyecto es inexistente porque la aplicación informática resultante no implica el uso de máquinas de alto consumo. Es bien sabido que el alto consumo de energía está asociado con los centros de datos. Este proyecto no los utiliza y, por lo tanto, podemos afirmar que nuestra aplicación no requiere alto consumo de energía.

No obstante, el desarrollo de nuevos algoritmos de compresión podría redundar en una disminución de recursos para almacenamiento de datos, y/o en una disminución de tiempo de computo efectivo para comprimir/descomprimir, lo cual podría traducirse en una disminución del consumo energético lo cual es un motivo suficiente como para seguir investigando esta ciencia.

Con respecto al impacto social, el desarrollo de este proyecto ha implicado una primera fase de exploración para abordar una serie de desafíos relacionados con la implementación de un algoritmo de compresión definido previamente. Dado que nuestro proyecto requiere un par de iteraciones de desarrollo de software para ser considerado un proyecto de software comercial, consideramos que el impacto social solo está relacionado con los resultados preliminares obtenidos que podrían permitir un desarrollo futuro con resultados interesantes para atacar problemas reales en varios dominios de aplicación así como un primer paso para abordar un estudio más exhaustivo de los códigos de Zeckendor.

Finalmente, con respecto a la Responsabilidad ética y profesional, afirmamos que el desarrollo de este proyecto no está relacionado con ningún factor que contradiga estos dos aspectos. Por otro lado, afirmamos que este Proyecto fue desarrollado con el más

alto respeto por el ejercicio de la profesión y, por lo tanto, se han considerado tanto Ética como Responsabilidad Profesional de una manera personal.

4.3 Análisis de los problemas encontrados

Durante el desarrollo de este proyecto se han encontrado varios problemas que, una vez terminado y con la experiencia adquirida me resulta notable mencionar, ya que de repetirse el proyecto desde el principio seria posible tomar medidas para evitarlas.

El primero de todos, y a mi parecer el más importante, es el relacionado con la imposibilidad de trabajar nativamente en Python2.7 con bytes. Debido a este problema el código escrito es a veces algo redundante en el tratamiento y realiza operaciones de 'workaround' para solventar la falla. Por supuesto, en python3 este tratamiento nativo es posible, con lo cual, si se tuviera que repetir el proyecto, esta seria la primera medida a tomar antes de comenzarlo. Como problema derivado de este, esta también el impedimento de tratar el carácter fin de fichero EOF. Por esto, si un fichero de longitud no múltiplo de 8 es compreso y descompreso usando alguna de las técnicas desarrolladas, el carácter EOF es escrito dos veces. Si bien este no es un problema grave y que es fácilmente tratable con herramientas externas, una persona que desconozca el problema llegaría a tener complicaciones al comparar ficheros con una herramienta como 'diff'.

El otro problema grave, a mi parecer, es la poca variedad (por no decir nula) de programas disponibles para comparar ficheros a nivel de bytes. Debido a esto, el proceso de codificación ha sido ralentizado al buscar posibles errores de implementación. Ya que para detectar un error en estos niveles de profundidad ha sido necesario usar las herramientas de GNU que lo permiten. Lo cual es un trabajo bastante artesano y para nada trivial.

En cuanto a la calidad de la compresión, si bien es cierto que está muy lejos de alcanzar los niveles de los programas comerciales, si que podría ser mejorada realizando algunos ajustes de mapeado de valores, lo que llevaría a una mejora bastante sustancial.

Por ultimo, también es de destacar la poca literatura pública que hay sobre codificación de ficheros en internet. Este problema es en parte entendible, ya que los algoritmos de compresión populares son en su mayoría privativos. De cualquier modo,

la investigación de los algoritmos ha sido lastrada por este motivo.

4.4 Futuras lineas de desarrollo

Después de analizar los problemas encontrados y de haber definido los limites de la implementación llevada a cabo, estos son algunos de los puntos que podrían ser desarrollados en un futuro:

1. Portar el código de python2 a python3, aprovechando las características extras de este cambio.

2. Mapeo de los valores mas utilizados para asignarlos a los códigos mas cortos.

3. Implementación de una herramienta que dado un fichero compruebe cual de las codificación usadas es la más óptima.

4. Análisis en profundidad de las propiedades de los códigos generalizados de Zeckendorf y búsqueda de una regla que relacione el tipo de fichero con su código de Zeckendorf óptimo.

Capítulo 5

Annex

5.1 Código Fuente

5.1.1 Fibonacci and Lucas Compressor

```
from itertools import takewhile
import sys

def __fibonacci(first, second):
    a, b = first, second
    while True:
        yield b
        a, b = b, a + b

#get the given number n in terms of fibonacci numbers
def Fibonacci(n,first=0,second=1):
    for f in reversed(list(takewhile(lambda f: f <= n, __fibonacci(first,second)))):
        if f <= n:
            n -= f
            yield f

#get the given number n in terms of lucas numbers, lucas matrix is [2,1,3,4,7,11...]
def Lucas(n):
    lucasList = list(reversed(list(takewhile(lambda f: f <= n, __fibonacci(2,1)))))
    lucasList.insert(-1,2)
    for f in lucasList:
```

```
22          if f <= n:
23              n -= f
24              yield f
25
26  #same as Fibonacci but the return value is the index
27  def __NFibonacci(n):
28      descomposition = list(reversed(list(takewhile(lambda f: f <= n, __fibonacci(0,1)))))
29      for f in descomposition:
30          if f <= n:
31              n -= f
32              yield list(reversed(descomposition)).index(f)
33
34  def __NLucas(n):
35      descomposition = list(reversed(list(takewhile(lambda f: f <= n, __fibonacci(2,1)))))
36      descomposition.append(2)
37      for f in descomposition:
38          if f <= n:
39              if n == 2 :
40                  n-=f
41                  yield 0
42              else:
43                  n -= f
44                  yield list(reversed(descomposition)).index(f)
45
46  #get the given number as a byte representation
47  def numToByte(n):
48      return format(n, '08b')
49
50  def byteToNum(n):
51      return int(n,2)
52
53  #get the given number as a byte compressed representation
54  #the compress matrix is [1, 2, 3, 5, 8, ...]
55  def compressNumAsFibo(n):
56      indexes = list(reversed(map(lambda x: x - 1 if x>1 else x, __NFibonacci(n))))
57      compressedByte = ""
58      for i in range(indexes[-1]+1):
59          if i in indexes:
60              compressedByte += '1'
61          else:
62              compressedByte += '0'
```

```
63      return compressedByte
64
65  #the compress matrix is [2, 1, 3, 7, 11, ...]
66  def compressNumAsLucas(n):
67      indexes = list(reversed(list(__NLucas(n))))
68      compressedByte = ""
69      for i in range(indexes[-1]+1):
70          if i in indexes:
71              compressedByte += '1'
72          else:
73              compressedByte += '0'
74      return compressedByte
75
76  def compressFileAsFibo(f, output='', maxValueToCompress=255, valueFit=97):
77      if (output == ''):
78          output=f+'.fc'
79
80      compressedBytes = ""
81
82      with open(f, "rb") as f:
83          byte = f.read(1)
84          while byte != "":
85              # Do stuff with byte.
86              compressedBytes += compressNumAsFibo(((ord(byte)-valueFit)%
87                                                    maxValueToCompress)+1)
88              compressedBytes += "1"
89              byte = f.read(1)
90
91      with open (output, "wb") as f:
92          arrayToWrite = [compressedBytes[i:i+8] for i in range(0,
93                                                    len(compressedBytes), 8)]
94          f.write(bytearray([byteToNum(i) for i in arrayToWrite]))
95
96  def compressFileAsLucas(f, output='', maxValueToCompress=255, valueFit=96):
97      if (output == ''):
98          output=f+'.lc'
99
100     compressedBytes = ""
101
102     with open(f, "rb") as f:
103         byte = f.read(1)
```

```
104         while byte != "":
105             # Do stuff with byte.
106             compressedBytes += compressNumAsLucas(((ord(byte)-valueFit)%
107                                                    maxValueToCompress)+1)
108             compressedBytes += "1"
109             byte = f.read(1)
110 
111     with open (output, "wb") as f:
112         arrayToWrite = [compressedBytes[i:i+8] for i in range(0,
113                                                    len(compressedBytes), 8)]
114         f.write(bytearray([byteToNum(i) for i in arrayToWrite]))
115 
116 def getDecompressionMatrix(n, first=0, second=1):
117     matrix = []
118     a, b = first, second
119     while b<=n:
120         matrix.append(b)
121         a, b = b, a + b
122     matrix.pop(0)
123     return matrix
124 
125 def decompressNum(n, decompressionMatrix):
126     decompressedNum=0
127     try:
128         for i in range(len(n)):
129             if n[i] == '1':
130                 decompressedNum+=decompressionMatrix[i]
131     except IndexError:
132         print "ERROR: Value out of range."
133         raise
134     return decompressedNum
135 
136 def decompressFileAsFibo(f, output="", maxValueToDecompress=255, valueFit=97):
137     if(output == ''):
138         output = f[:-3] + ".fd"
139 
140     dMatrix = getDecompressionMatrix(maxValueToDecompress)
141     compressedBytes=""
142     with open(f, "rb") as f:
143         byte = f.read(1)
144         while byte != "" :
```

```
145             compressedBytes += numToByte(ord(byte))
146             byte = f.read(1)
147 
148     compressedNum=""
149     endNumFlag = False
150     decompressedFile=""
151     for i in compressedBytes:
152         if i == '1' :
153             if endNumFlag == False:
154                 endNumFlag = True
155                 compressedNum += i
156             else:
157                 try:
158                     decompressedNum = decompressNum(compressedNum, dMatrix)
159                     decompressedFile+=chr(((decompressedNum+valueFit-1) %
160                                                         maxValueToDecompress))
161                 except:
162                     print "Ignoring char"
163                 compressedNum=""
164                 endNumFlag = False
165         else:
166             endNumFlag = False
167             compressedNum+=i
168 
169     with open(output, "wb") as f:
170         f.write(decompressedFile)
171         f.write('\n')
172 
173 def decompressFileAsLucas(f, output="", maxValueToDecompress=255, valueFit=96):
174     if(output == ''):
175         output = f[:-3] + ".ld"
176 
177     dMatrix = getDecompressionMatrix(maxValueToDecompress,first=2,second=1)
178     dMatrix.insert(0,1)
179     dMatrix.insert(0,2)
180     compressedBytes=""
181     with open(f, "rb") as f:
182         byte = f.read(1)
183         while byte != "" :
184             compressedBytes += numToByte(ord(byte))
185             byte = f.read(1)
```

```
186
187     compressedNum=""
188     endNumFlag = False
189     decompressedFile=""
190     for i in compressedBytes:
191         if i == '1' :
192             if endNumFlag == False:
193                 endNumFlag = True
194                 compressedNum += i
195             else:
196                 try:
197                     decompressedNum = decompressNum(compressedNum, dMatrix)
198                     decompressedFile+=chr(((decompressedNum+valueFit-1) %
199                                                         maxValueToDecompress))
200                 except:
201                     print "Ignoring char"
202                 compressedNum=""
203                 endNumFlag = False
204         else:
205             endNumFlag = False
206             compressedNum+=i
207
208     with open(output, "wb") as f:
209         f.write(decompressedFile)
210         f.write('\n')
```

5.1.2 Zeckendorf Compressor

```
1  from itertools import takewhile
2  import sys
3  import math
4
5  def __gzeck(k, first=1, second=1):
6      a, b = first, second
7      while True:
8          yield b
9          a, b = b, a + b*k
10
11 #return a descomposition in terms of general fibonacci's series using the zeckendorf
```

```
12 # theorem as [value, multiplier]
13 def GZeck(k,n):
14     for f in reversed(list(takewhile(lambda f: f <= n, __gzeck(k)))):
15         if f <= n:
16             for i in reversed(range(1,k+1)):
17                 if f*i <= n:
18                     n -= f*i
19                     yield [f,i]
20 
21 #same as GZeck but the return value is [index, multiplier]
22 def __GZeck(k,n):
23     if n == 1:
24         yield [0,1]
25     descomposition = list(reversed(list(takewhile(lambda f: f <= n, __gzeck(k)))))
26     for f in descomposition:
27         if f <= n:
28             for i in reversed(range(1,k+1)):
29                 if f*i <= n:
30                     n -= f*i
31                     yield [list(reversed(descomposition)).index(f), i]
32 
33 def numToByte(n):
34     return format(n, '08b')
35 
36 def byteToNum(n):
37     return int(n,2)
38 
39 #get the given number as a GZeck compressed binary representation
40 def compressNumAsGZeck(k,n):
41     indexes = list(reversed(list(__GZeck(k,n))))
42     compressedByte = ""
43     numBitsPerIndex = int(math.ceil(math.log(k+1,2)))
44     for i in range(indexes[-1][0]+1):
45         indexesIndex = list(indexes[x][0] for x in range(len(indexes)))
46         if i in indexesIndex:
47             compressedByte += format(indexes[indexesIndex.index(i)][1],'0'+
48                                                 str(numBitsPerIndex)+'b')
49         else:
50             compressedByte += format(0,'0'+str(numBitsPerIndex)+'b')
51     return compressedByte
52 
```

```
53 def compressFileAsGZeck(f, k=6, output='', valueFit=96, maxValueToCompress=255):
54     numBitsPerIndex = int(math.ceil(math.log(k+1,2)))
55     if (output == ''):
56         output=f+'.gzk' + str(k)
57 
58     compressedBytes = ""
59     finalWord=''
60     for i in range(numBitsPerIndex):
61         finalWord += '1'
62 
63     with open(f, "rb") as f:
64         byte = f.read(1)
65         while byte != "":
66             # Do stuff with byte.
67             compressedBytes += compressNumAsGZeck(k,
68                                                   ((ord(byte)-valueFit)%maxValueToCompress)+1)
69             #compressedBytes += compressNumAsGZeck(k,(ord(byte)+1))
70             compressedBytes += finalWord
71             byte = f.read(1)
72 
73     with open (output, "wb") as f:
74         while len(compressedBytes)%8 != 0:
75             compressedBytes+='0'
76         arrayToWrite = [compressedBytes[i:i+8] for i in range(0, len(compressedBytes), 8)]
77         f.write(bytearray([byteToNum(i) for i in arrayToWrite]))
78 
79 def getDecompressionMatrix(k, n, first=1, second=1):
80     matrix = []
81     a, b = first, second
82     while b<=n:
83         matrix.append(b)
84         a, b = b, a + k*b
85     return matrix
86 
87 
88 def decompressNum(k, n, decompressionMatrix):
89     numBitsPerIndex = int(math.ceil(math.log(k+1,2)))
90     decompressedNum=0
91     for i in range(len(n)/numBitsPerIndex):
92         decompressedNum+=decompressionMatrix[i]*
93                                 byteToNum(n[i*numBitsPerIndex:(i*numBitsPerIndex)
```

```
94                                                                  +numBitsPerIndex])
95      return decompressedNum
96
97  #TODO (fin de palabra si se encuentra an=k y an-1!=0 )
98  def decompressFileAsGZeck(f, k=6, output="", valueFit=96,maxValueToDecompress=255):
99      numBitsPerIndex = int(math.ceil(math.log(k+1,2)))
100     if(output == ''):
101         output = f + ".d"
102
103     dMatrix = getDecompressionMatrix(k,2**8)
104     compressedBytes=""
105     with open(f, "rb") as f:
106         byte = f.read(1)
107         while byte != "" :
108             compressedBytes += numToByte(ord(byte))
109             byte = f.read(1)
110
111     compressedNum=""
112     decompressedFile=""
113     nullWordFlag=True
114     nullWord=''
115     finalWord=''
116     for i in range(numBitsPerIndex):
117         finalWord+='1'
118         nullWord+='0'
119
120     for i in [ compressedBytes[a:a+numBitsPerIndex] for a in range(0,len(compressedBytes),
121                                                                    numBitsPerIndex) ]:
122         if i == finalWord and nullWordFlag == False:
123             decompressedNum = decompressNum(k,compressedNum, dMatrix)
124             decompressedFile+=chr(((decompressedNum+valueFit-1) % maxValueToDecompress))
125             #decompressedFile+=chr(decompressedNum-1)
126             compressedNum=""
127             nullWordFlag=True
128             continue
129         else:
130             compressedNum+=i
131
132         if i == nullWord:
133             nullWordFlag=True
134         else:
```

```
135             nullWordFlag=False
136
137     with open(output, "wb") as f:
138         f.write(decompressedFile)
139         f.write('\n')
```

Bibliografía

[Fraenkel and Kleinb, 1996] Fraenkel, A. S. and Kleinb, S. T. (1996). Robust universal complete codes for transmission and compression. *Discrete Applied Mathematics*, 64(1):31–55.

[Golomb, 1966] Golomb, S. (1966). Run-length encodings (corresp.). *IEEE transactions on information theory*, 12(3):399–401.

[Hoggatt, 1972] Hoggatt, V. E. (1972). Generalized zeckendorf theorem.

[Huffman, 1952] Huffman, D. A. (1952). A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101.

[Knuth, 1985] Knuth, D. E. (1985). Dynamic huffman coding. *Journal of algorithms*, 6(2):163–180.

[Lewis, 1991] Lewis, P. N. (1991). Binhex 4.0 definition. http://files.stairways.com/other/binhex-40-specs-info.txt. [Online a fecha 24-Noviembre-2017].

[MacMillan and Krandall, 2010] MacMillan, D. M. and Krandall, R. (2010). Codes that don't count: Some printing telegraph codes as products of their technologies (with particular attention to the teletypesetter). http://www.circuitousroot.com/artifice/telegraphy/tty/codes/. [Online a fecha 14-Diciembre-2017].

[Salomon, 1997] Salomon, D. (1997). *Data Compresion: The complete reference*. Springer-Verlag, New York, NY, USA.

[Shannon, 1948] Shannon, C. E. (1948). A mathematical theory of communication. *SIGMOBILE Mob. Comput. Commun. Rev.*, 5(1):3–55.

[Vitter, 1987] Vitter, J. S. (1987). Design and analysis of dynamic huffman codes. *Journal of the ACM (JACM)*, 34(4):825–845.

[Zeckendorf, 1972] Zeckendorf, E. (1972). Representations des nombres naturels par une somme de nombres de fibonacci on de nombres de lucas. *Bulletin de La Society Royale des Sciences de Liege*, pages 179–182.