**Algoritmos de compresión-descompresión**: además de empequeñecer la información, representan a su vez una capa de seguridad. Dos pájaros de un tiro.

Por ejemplo, Telegram utiliza un algoritmo de compresión de textos que omite las vocales. ¿Qué tipo de algoritmo de compresión existen?

1. Compresión sin pérdida: no se pierden datos al recuperar la información. Mantiene la calidad, pero no se reduce tanto el tamaño.
2. Compresión con pérdida: el archivo descomprimido es más pequeño que el archivo original. Por lo tanto, existe pérdida de información. No obstante, esta pérdida no debe ser obvia para el usuario final.

Por ejemplo, la compresión con pérdida se utiliza en ficheros tipos mp3 o jpeg.

**Tipos de algoritmos de compresión-descompresión**

**Codificación Huffman**: el algoritmo consiste en la creación de un árbol binario que tiene cada uno de los símbolos por hoja.   
Se parte de un texto guardado en un archivo: ***abbacabbcd***

El archivo de texto se guarda utilizando un estándar de codificación, por ejemplo, ASCII.   
ASCII codifica cada carácter en una secuencia de **8 bits**.

a🡪01100001

b🡪01100010

c🡪01100011

d🡪01100100

Así, si se parte de una cadena de 10 caracteres, al codificarla con ASCII resultaría una secuencia de 80 dígitos. Como esto no resulta muy eficiente, se codifica.

***abbacabbcd:* tiene cuatro caracteres únicos🡪abcd. Bastaría con utilizar únicamente dos bits para codificar cada uno de ellos.**

a🡪00

b🡪01

c🡪10

d🡪11

Esto nos estaría ahorrando seis dígitos por cada carácter respecto a la codificación en ASCII.

Además, se puede reducir incluso más:

a🡪0

b🡪1

c🡪10

d🡪11

Con esta compresión, el texto que debe decodificar el receptor es:

0 1 1 0 10 0 1 1 10 11

¿Cómo se decodificaría el 5º carácter? 10 puede ser interpretado como ba o c.

El problema surge cuando la forma codificada de un carácter se encuentra al inicio de la otra. Para solucionarlo, **decodificación Huffman**, que evita que la forma codificada de un carácter se encuentre al inicio de la otra:

1. Se crea un nodo para cada carácter presente en el texto, incluyendo su frecuencia de aparición: ***abbacabbcd***

a(3), b(4), c(2), d(1)

1. Se ordena con una cola de prioridad, de menor frecuencia a mayor frecuencia.
2. Se cogen las dos con prioridad más pequeña (que serán los dos de la izquiera), creando un nodo padre cuyo valor será la suma de las dos frecuencias:

Forma, Círculo

Descripción generada automáticamente

1. Se inserta el nodo padre creado (raíz) de nuevo en la lista.

Forma

Descripción generada automáticamente

1. Se repite el mismo procedimiento:

Diagrama, Forma

Descripción generada automáticamente

La estructura resultante es el **árbol de Huffman,** a partir del cual se generan los códigos que se necesitan.

Cada vez que se baje a la izquierda, se pone un cero. Cada vez que se baje a la derecha, se pone un uno.

Imagen de la pantalla de un celular con la imagen de una caricatura

Descripción generada automáticamente con confianza baja

**a🡪10**

**b🡪0**

**c🡪111**

**d🡪110**

***Nota:*** la forma codificada de a, b, c o d no se encuentra al inicio de otra. Esto es, 10 no es el inicio de ninguna forma codificada. Tampoco el 0, el 111 ni el 110.

**Por lo tanto; *abbacabbcd 🡪 10 0 0 10 111 10 0 0 111 110***

**La forma comprimida de la cadena inicial queda finalmente con 19 bits.**

Esto es, se ha pasado de una hipotética reducción de 80 bits a 19 bits.

**RLE (run-length encoding):** Es una forma muy simple de compresión de datos en la que secuencias de datos con el mismo valor consecutivas son almacenadas como un único valor más su recuento. Por ejemplo, considera una pantalla que contiene texto en negro sobre un fondo blanco. Supongamos una única línea, con N representando las zonas en negro y B las de blanco: BBBBNNBBBBBBBBNNNNBNNBBBB

El resultado en RLE sería la siguiente: 4B2N8B4N1B2N4B.

RLE es capaz de reducir la cadena binaria considerablemente. Toda la información es recuperable.

**Zip:** Es un formato de compresión sin pérdida, muy utilizado para la compresión de datos de documentos, imágenes o programas.

**Rar:** Es un formato de archivo privado, con un algoritmo de compresión sin pérdida utilizado para la compresión de datos y archivado. Utiliza un algoritmo de compresión basado en el LZSS que, a su vez, se basaba en el LZ77.

**Navegadores**: software, por norma general gratuito, utilizado para representar gráficamente la World Wide Web. Un navegador permite visualizar textos, imágenes y vídeos, pero también enlaces y otras [funciones de los sitios web](https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/que-pasa-cuando-se-abre-una-pagina-web/).

Con la introducción del **hipertexto**, se añadieron **referencias cruzadas** en la navegación, los llamados [hipervínculos](https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/hipervinculos-que-son-realmente-y-para-que-se-utilizan/). Desde entonces, las funcionalidades de los navegadores han crecido enormemente. Además de imágenes y vídeos, pueden integrar y dotar de funciones a gráficos interactivos, archivos de audio, PDF y otros recursos.

**Man in the Middle:** Un ataque Man in the Middle (MitM) o ataque de intermediario es el método por el cual un hacker interviene en el tráfico de datos de dos partes vinculadas entre sí en una comunicación haciéndose pasar por cualquiera de ellas, haciéndoles creer que se están comunicando entre ellos cuando en realidad es el intermediario quien recibe la comunicación. Tipo de ataques MitM:

* Ataques basados en servidores DHCP (DHCP sproofing)
* ARP cache poisoning
* Ataques basados en servidores DNS
* Simulación de un punto de acceso inalámbrico
* Ataque Man in the Browser

**sys.getsizeof():**

**https://code.tutsplus.com/tutorials/understand-how-much-memory-your-python-objects-use--cms-25609**

sys.getsizeof([])

72

sys.getsizeof([1])

88

sys.getsizeof([1, 2, 3, 4])

104

sys.getsizeof(['a long longlong string'])

An empty list takes 72 bytes, but each additional int adds just 8 bytes, where the size of an int is 24 bytes. A list that contains a long string takes just 80 bytes.

The list doesn’t contain the int objects themselves. It just contains an 8-byte (on 64-bit versions of CPython) pointer to the actual int object. What that means is that the getsizeof() function doesn’t return the actual memory of the list and all the objects it contains, but only the memory of the list and the pointers to its objects. In the next section I’ll introduce the deep\_getsizeof() function that addresses this issue.

sys.getsizeof() can only tell you how much memory a primitive object takes. For example, called on a list, it would not include the memory taken by the list's elements).

$ py -m pip install pympler