Pseudocódigo de los algoritmos

Estructuras de datos comunes

Dado que el diagrama de clases que hemos desarrollado ha estado pensado desde el primer momento para que nuestro código sea altamente reusable y cambiable, en el caso de las clases que implementan los algoritmos nos encontramos que estas se aprovechan de las bondades del polimorfismo. Esto es así porque al fin y al cabo, aún siendo algoritmos distintos, comparten características comunes, como la manera de recibir o devolver los datos de compresión o descompresión. Es por esto que todos ellos hacen uso de las mismas estructuras de datos para justamente este caso.

Byte array

Con esta estructura pasamos al algoritmo de compresión y descompresión de cualquiera de las cuatro clases los datos contenidos por el fichero de input. También es la estructura usada para devolver el resultado del algoritmo. Nos permite leer y escribir con coste constante, y su coste en memoria es igual al tamaño del fichero.

Lista de bytes

En los algoritmos se utiliza esta estructura de datos principalmente para almacenar el resultado a medida que el código lo genera. Como no sabemos con antelación el tamaño del resultado del proceso, debemos estar provistos de una estructura de datos con tamaño dinámico, la cual nos permita a su vez también escribir secuencialmente. Es por esto que una lista es ideal para estos casos. Nos es más conveniente usar una lista que por ejemplo arrays de bytes. Si quisiéramos usar array de bytes tendríamos que hacer un nuevo array al escribir cada byte en el resultado, siendo esto altamente ineficiente tanto en tiempo como en memoria.

OutputAlgoritmo

Con esta nueva clase devolvemos la información del tiempo que ha tardado la función del algoritmo de compresión y descompresión, y el resultado de la codificación del fichero input. Esta nueva clase nos permite crear las estadísticas y posteriormente guardar en un fichero output la codificación de cada algoritmo.

JPEG

El algoritmo JPEG es un algoritmo de compresión de imágenes el cual es lossy, es decir, la compresión provoca pérdida de información. Subdivide la imagen en cuadrados de 8x8 píxeles, a los cuales se les aplica downsampling de la crominancia, una transformación cosenoidal discreta, cuantización, codificación Run-Length, y codificación de Huffman. Con la computación de estos cálculos se logra una compresión de hasta un 90% respecto al tamaño original del archivo por comprimir.
Para descomprimir, se hace el mismo proceso anterior, pero a la inversa. Esto es, el primer paso es el último de la compresión, y además el paso en sí también se hace a la inversa para lograr la información previa. Como resultado de la descompresión, obtenemos una aproximación muy fidedigna a la original. Cuanta mayor sea la calidad de compresión elegida, menos comprimirá, pero más semejante será al archivo original.
Las dos funciones más relevantes del fichero JPEG.java son la compresion y descompresión. Estas serán las funciones que se comentarán a continuación. En azul están comentadas las estructuras de datos utilizadas con una pequeña explicación que razona la elección de estas.

Atributos privados

- **CalidadHeader**: número de 1 al 7 que determina la calidad de las imágenes a la hora de comprimir y descomprimirlas, siendo 1 la calidad más baja y 7 la más alta. Este atributo se puede cambiar al comprimir la imagen, pero al descomprimirla, este está escrito en el fichero de manera que el algoritmo ya sepa con qué calidad ha sido comprimida la imagen. De esta manera sabrá con que calidad se tiene que descomprimir y que muestre un resultado correcto. (se usará en el paso 3 de compresión y en el paso 1 de descompresión).
- **Calidad**: escalar calculado en función de CalidadHeader que se multiplica por las tablas de cuantización de luminosidad y crominancia (se usará en el paso 4 y en el paso 2 de descompresión).
- **LuminanceQuantizationTable**: tabla de cuantización (array 2D) de la luminosidad (Y) para cuadrados de 8x8 valores (se usará en el paso 4 de compresión y en el paso 2 de descompresión).
- **ChrominanceQuantizationTable**: tabla de cuantización (array 2D) de la crominancia (Cb y Cr) para cuadrados de 8x8 valores (se usará en el paso 4 de compresión y en el paso 2 de descompresión).

Comprimir

Paso 1: Se inicia contador de tiempo. Inicializar lista result para almacenar el resultado. Si los datos de entrada tienen un tamaño inferior al mínimo posible o hay menos o más datos que los que debería haber con las dimensiones provistas, se activa una excepción de error de formato.
Lectura del header del fichero .ppm, que contiene la siguiente información:

- Magic Number: el cual tiene que ser **P6**, tal y como se especificó desde el inicio del proyecto por parte del profesorado. En caso contrario se activa una excepción de error de formato.
- Dimensiones: las cuales deben estar formadas por **dos cifras con valor positivo**. En caso contrario se activa una excepción de error de formato.

- Máximo valor de RGB: el cual, tal y como se especifica en el formato estándar de PPM con Magic Number P6, tiene que ser **255**, ya que todos los valores RGB del pixel map tienen que ocupar un solo byte, y tal y como se acordó con el profesor de laboratorio, no tiene sentido desaprovechar bits del byte y poner un valor menor a 255. En caso contrario se activa una excepción de error de formato.
  https://my.eng.utah.edu/~cs5610/ppm.html
  En caso de que haya comentarios, se acordó con el profesor que estos sólo estarían situados en aquellos sitios considerados como estándar. Esto es, entre Magic Number y Dimensiones, y entre Dimensiones y Máximo valor de RGB. En estos casos, el algoritmo omite los comentarios y los elimina de cara a la escritura de los datos de salida.
  Se escriben estos valores a la lista result.

Paso 2: Lectura del pixel map, descomposición de las componentes RGB en YCbCr, centrado de los valores alrededor de cero (rango [-128,127]) y almacenado en sus correspondientes arrays bidimensionales (uno por componente de color). Se hace uso de arrays, ya que la medida del pixel map es fija, e interesa tener acceso constante y aleatorio a sus elementos. Su coste espacial es adecuado para las necesidades de JPEG y su coste temporal es óptimo para las necesidades del algoritmo.

```
for(cada 3 bytes del byte[] que se nos pasa por parametro){
        Lectura de las componentes RGB.
        Decomposición del RGB en YCbCr.
        Centrado de los valores alrededor de cero.
        Almacenado de YCbCr en sus correspondientes arrays.
        En caso de que las dimensiones no sean múltiples de 8, se arrastran los valores de la
        última columna y la última fila hasta que se cumpla la condición.
}
```

Paso 3: Downsampling de la crominancia (Cb y Cr) a un 25% de la imagen original, reduciendo el tamaño de la imagen hasta a un 50% del original sin pérdidas visibles de calidad y color.

```
for(cada 2 valores en el eje x y en el eje y de Cb y Cr){
        Se hace la media de los cuatro valores (2 por eje x y 2 por eje y) y se guardan en el array
        de Cb o Cr downsampled.
        Si los arrays downsampled al dividir entre 2 sus dimensiones estas pasan a no ser
        múltiples de 8, se arrastran los valores de la última columna y la última fila hasta que se
        cumpla la condición.
}
Se escribe el valor de la calidad de compresión en la lista result.
```

Paso 4: DCT-II de los arrays de Y, Cb y Cr, cuantización de los valores resultantes en función de la calidad configurada para el proceso de compresión y compresión de los datos usando Run-length y Huffman Encoding.

```
for(cada cuadrado de 8x8 de la Y){
        for(cada valor){
                Se calcula la contribución cosenoidal de las frecuencias de la luz de los valores
                del cuadrado para esa posición y se guarda el nuevo valor en un buffer temporal.
```

}

Se cuantiza y guarda en el array de Y los 64 nuevos valores calculados y almacenados en el buffer, correspondiéndoles a sus respectivas posiciones en el array de Y.

Se guardan los valores en una list en zig zag siguiendo el estándar especificado en JPEG.

Se aplica RLE a la list, reduciendo el espacio usado para escribir valores que tienen múltiples ocurrencias consecutivas a un espacio muy inferior (sobretodo la gran tira de ceros consecutivos que consigue DCT+cuantización+zig zag).

Se aplica Huffman encoding a estos datos, construyendo un árbol binario que nos permite escribir los valores en la lista result usando el mínimo de bits posible. También se escribe la tabla de valores generada para poder decodificar los datos.

}

for(cada cuadrado de 8x8 de la Cb y Cr){

for(cada valor){

Se calcula la contribución cosenoidal de las frecuencias de la luz de los valores del cuadrado para esa posición y se guarda el nuevo valor en un buffer temporal.

}

Se cuantiza y guarda en los arrays de Cb y Cr los 64 nuevos valores calculados y almacenados en el buffer, correspondiéndoles a sus respectivas posiciones en los arrays de Cb y Cr.

Se guardan los valores en una list en zig zag siguiendo el estándar especificado en JPEG.

Se aplica RLE a la list, reduciendo el espacio usado para escribir valores que tienen múltiples ocurrencias consecutivas a un espacio muy inferior (sobretodo la gran tira de ceros consecutivos que consigue DCT+cuantización+zig zag).

Se aplica Huffman encoding a estos datos, construyendo un árbol binario que nos permite escribir los valores en la lista result usando el mínimo de bits posible. También se escribe la tabla de valores generada para poder decodificar los datos.

}

Paso 5: Escritura del byte array de output, del cual se hace return, con los datos almacenados en la list result. Se finaliza contador de tiempo.

Descomprimir

Paso 1: Se inicia contador de tiempo. Inicializar lista result para almacenar el resultado. Si los datos de entrada tienen un tamaño inferior al mínimo posible o hay menos o más datos que los que debería haber con las dimensiones provistas, se activa una excepción de error de formato.
Lectura del header del fichero .imgc, que contiene la siguiente información:

- Magic Number: el cual tiene que ser **P6**, tal y como se especificó desde el inicio del proyecto por parte del profesorado. En caso contrario se activa una excepción de error de formato.
- Dimensiones: las cuales deben estar formadas por **dos cifras con valor positivo**. En caso contrario se activa una excepción de error de formato.
- Máximo valor de RGB: el cual, tal y como se especifica en el formato estándar de PPM con Magic Number P6, tiene que ser **255**, ya que todos los valores RGB del pixel map tienen que ocupar un solo byte, y tal y como se acordó con el profesor de laboratorio, no tiene sentido

desaprovechar bits del byte y poner un valor menor a 255. En caso contrario se activa una excepción de error de formato.
https://my.eng.utah.edu/~cs5610/ppm.html

- Calidad: calidad que se debe usar de escalar para hacer la cuantización inversa en el proceso de descompresión.

En caso de que haya comentarios, se acordó con el profesor que estos sólo estarían situados en aquellos sitios considerados como estándar. Esto es, entre Magic Number y Dimensiones, y entre Dimensiones y Máximo valor de RGB. En estos casos, el algoritmo omite los comentarios y los elimina de cara a la escritura de los datos de salida.
Se escriben estos valores a la lista result.

Paso 2: Lectura en zig zag del pixel map en cuadrados de 8x8, cuantización inversa, DCT-III y almacenado en sus correspondientes arrays bidimensionales (uno por componente de color). Se hace uso de arrays, ya que la medida del pixel map es fija, e interesa tener acceso constante y aleatorio a sus elementos. Su coste espacial es adecuado para las necesidades de JPEG y su coste temporal es óptimo para las necesidades del algoritmo.

```
for(cada cuadrado de 8x8 de la Y){
    Lectura en zig zag del cuadrado, decodificado de Huffman usando la tabla provista para cada cuadrado y decodificado de RLE interpretando las ocurrencias reducidas y volviéndolas a escribir hasta tener los 64 elementos.
    for(cada valor){
        Se hace la cuantización inversa de cada elemento.
        Se hace el cálculo inverso de DCT de los valores del cuadrado para esa posición y se guarda el nuevo valor en un buffer temporal.
    }
    Se escriben en el array de Y los valores almacenados en el buffer, correspondiéndoles a sus respectivas posiciones en el array de Y.
}
for(cada cuadrado de 8x8 de la Cb y Cr){
    Lectura en zig zag del cuadrado, decodificado de Huffman usando la tabla provista para cada cuadrado y decodificado de RLE interpretando las ocurrencias reducidas y volviéndolas a escribir hasta tener los 64 elementos.
    for(cada valor){
        Se hace el cálculo inverso de DCT de los valores del cuadrado para esa posición y se guarda el nuevo valor en un buffer temporal.
    }
    Se escriben en el array de Cb y Cr los valores almacenados en el buffer, correspondiéndoles a sus respectivas posiciones en el array de Cb y Cr.
}
```

Paso 3: Conversión de los valores de YCbCr a RGB, descentrado de los valores alrededor de cero (rango [0,255]) y supersampling de Cb y Cr para que haya un valor de YCbCr por cada pixel de la imagen

```
for(cada elemento de Y, Cb y Cr) {
```

Conversión y descentrado de los valores para que sean RGB con rango [0,255] (cada elemento de CbCr se usa dos veces en cada eje para recuperar los valores perdidos por el downsampling).
Escritura en la list result de los valores RGB de cada pixel.

}

Paso 4: Escritura del byte array de output, del cual se hace return, con los datos almacenados en la list result. Se finaliza contador de tiempo.

LZ78

El algoritmo LZ78 es un algoritmo de compresión sin pérdida. La idea principal es ir recorriendo secuencias de caracteres y reunirlas todas según su aparición en un árbol de tipo Trie, con el cual por cada secuencia recogida, solo necesitaremos saber el último carácter de la misma y la posición del anterior carácter para poder construir cada secuencia y reunirlas ordenadamente en el texto completo.

Para ir añadiendo elementos al árbol lo que buscamos es que una secuencia esté identificada por el último carácter de la misma, por tanto, será necesario que no haya ningún tipo de repetición exacta. Cada carácter tendrá dos valores asignados, su propio índice, que viene implícito según el orden en el que hemos escrito las secuencias y, por último, el índice del anterior carácter de la secuencia. De tal manera que yo sabiendo el índice de todos los últimos caracteres de las secuencias formadas puedo volver a construir el texto de nuevo.

Para poder conseguir que no se repitan secuencias exactas, iremos formando secuencias de tamaño variable, empezando por el tamaño de 1 carácter. Si la secuencia ya se encuentra en el árbol añadiremos un carácter más a la secuencia, así hasta tener una secuencia que no esté añadida y que, en conclusión, coincida con otra anterior pero sin el último carácter, que sería el que identificaría a toda la secuencia. Ese último carácter tendrá su propio índice asignado y se le asignará también el índice de su anterior carácter para permitir reconstruir la secuencia. Este proceso se repite hasta que se han clasificado todos los caracteres en secuencias y de esta manera ya tenemos una lista de paquetes de 3 datos (carácter, índice y anterior índice) que nos permiten reconstruir de nuevo todo el texto.

Para reconstruir todo el texto es tan fácil como consultar cada paquete de la lista e ir reconstruyendo las secuencias. Teniendo en cuenta que al estar la lista ordenada según aparición de la secuencia, se observa que todos los caracteres que forman la secuencia buscada ya los habremos recorrido y por tanto podremos encontrarlos fácilmente gracias a su índice y el índice del anterior carácter que le corresponde.

Estructuras de Datos

Trie

El objetivo principal como ya he comentado es construir un árbol, en el cual añadiremos todas las secuencias de caracteres formadas. El árbol consiste de nodos con un índice y un byte que representa el carácter. De esos nodos salen ramas a otros nodos con los siguiente posibles caracteres de la secuencia. Una secuencia de caracteres se forma desde el nodo raíz, el cual se encuentra vacío y sirve como puente para enlazar a todos los posibles nodos que inicien una secuencia de caracteres. El nodo raíz tiene como índice el 0, para poder saber fácilmente cuándo un carácter es el primero de la secuencia.

Esta estructura de datos es comúnmente llamada como Trie, la cual como ya he comentado está formada por nodos que representan caracteres y sus ramas con peso que nos permiten acceder a otros caracteres. Las ramas, al tener como peso el carácter del nodo hijo, facilitan la navegación por el árbol.

Los nodos están compuestos de solo 2 datos de los 3 que se mencionan en el paquete. Es decir, el índice específico de cada carácter es implícito según el orden de añadido de las secuencias en el árbol.

El uso de esta estructura de datos es debido a su facilidad de utilización y a que la navegación por el mismo resulta muy rápida, esto representa una clara mejora de velocidad respecto a Listas o Mapas. Esta estructura solo se utiliza en la compresión, donde se construyen palabras según letras. En la descompresión dado que solo necesitamos los índices para construir las palabras, bastará con utilizar una Lista para ubicar cada elemento.

Pair
Esta clase es un struct, la cual permite guardar elementos en paquetes de 2 en la lista de diccionario generada en descompresión. El struct está compuesto por un índice y un byte. De esta manera tenemos siempre el byte específico junto al índice del byte anterior.

Comprimir

Paso 1: Inicialización de variables.
- Se inicia contador de tiempo.
- Creamos una instancia vacía de un árbol Trie llamado *dictionary* que nos servirá para tener un seguimiento sencillo de todas las secuencias de bytes con coincidencias.

Paso 2: Tratamiento de cada byte de la secuencia.

por(cada byte dentro de la secuencia de bytes *datosInput*) {
Si el árbol está lleno reiniciamos el árbol.

Paso 3: Creamos una nueva Lista de Bytes *word* que utilizaremos para acumular bytes mientras haya coincidencias de la secuencia dentro del *dictionary*.

Añadimos el byte tratado en *word*.
Mientras (haya bytes por tratar y *word* ya exista en *dictionary*) {
Añadimos el siguiente byte por tratar a *word*.
}

Añadimos *word* al *dictionary* y conservamos el *índice* del penúltimo *byte* introducido.

Paso 4: Se prepara el *índice* para ser guardado adecuadamente como conjunto de bytes.

El *índice*, al tratarse de un integer, está compuesto de 4 bytes. Dado que no queremos guardar 3 bytes a 0 si el valor del índice realmente ocupa 1 byte, reservaremos un par de bits en el índice escrito para indicar la cantidad de bytes que ocupa el índice. En este caso con 2 bits al principio del primer byte nos sirve para definir el tamaño del índice desde 1 hasta 4 bytes.

En el *índice* también queremos guardar de alguna manera que el *dictionary* ha sido reiniciado, para que al descomprimir sepamos que hay que crear otro árbol nuevo. Por tanto, destinamos el valor máximo de los 2 bits reservados a indicar un reinicio y limitaremos el tamaño máximo del índice a 3 bytes. De estos 3 bytes reservamos igualmente los 2 bits destinados como flags.

En otras palabras, el *índice* será un máximo de 0x3FFFFF en hexadecimal o 4194303 en decimal.

Con esta información en mente ahora se procede a dividir el *índice* en un máximo de 3 partes y añadirlo a una Lista de ints *indexPart*, que nos permite añadir partes del *índice* sin definir el tamaño de la misma.

Añadimos a *indexPart* el *byte* más a la derecha del *índice*.
*Si (*índice >= 0x3F*)* {
Añadimos a *indexPart* añadimos el segundo byte más a la derecha del *índice* teniendo en mente que no se conservaban los 2 últimos bits del byte por tanto se hace un shift right de 6 bits y se aplica una máscara para 1 byte (0xFF).

Si(índice >= 0x3FFF) {
Añadimos a *indexPart* el tercer byte más a la derecha del índice teniendo en mente que no se conservaban los 2 últimos bits del byte por tanto se hace un shift right de 14 bits y se aplica una máscara para 1 byte (0xFF).
}
}

Por(cada parte de *indexPart*) {
Si(es el primer elemento de *indexPart*) {
En los dos primeros bits se le añade una mascára según 4 casos diferentes ya explicados anteriormente:
00 - Índice ocupa 1 byte
01 - Índice ocupa 2 bytes
10 - Índice ocupa 3 bytes
11 - El diccionario se ha reiniciado
}
}

Paso 5: Se añade cada uno de los bytes de *indexPart* y el byte del último carácter de la secuencia en la Lista de Bytes resultado.
}

Paso 6: Se prepara el resultado.
- Se convierte la Lista de Bytes a un Byte Array para fijar el tamaño del resultado.
- Se finaliza el contador de tiempo.
- Se devuelve una estructura de datos con el Byte Array resultante y el tiempo invertido.

Descomprimir

Paso 1: Inicialización de variables.
- Se inicia contador de tiempo.
- Se inicializa una Lista de Bytes a la que iremos agregando el texto descomprimido.
- Se inicializa una Lista de Bytes llamada diccionario que usaremos como referencia para buscar elementos según su índice. Se le añade un valor inicial null para el primer valor ya que el índice pueda saber cuando se ha termin<do de reconstruir una secuencia.

Paso 2: Tratamiento de cada byte de la secuencia.

por(cada byte dentro de la secuencia de bytes *datosInput*) {

Paso 3: Se trata del flag del índice para poder leer correctamente todo el índice o reiniciar el diccionario si se da el caso.

Lee el siguiente byte y lo asigna al index.
Flag = indice & 0xC0 (máscara de 2 bits).
index = index & 0x3F (máscara que elimina los 2 bits de flag)
if(flag == 11) Reinicia el diccionario.
if(flag == 01) Se lee otro byte, se le hace un shift left de 6 bits y se suma a index.
if(flag == 10) {
    Se lee otro byte, se le hace un shift left de 6 bits y se suma a index.
    Se lee otro byte, se le hace un shift left de 14 bits y se suma a index.
}

Paso 4: Se lee el siguiente byte que representa el carácter que va con el índice.

Paso 5: Añadimos al diccionario un Pair con el index y el carácter. La posición del mismo en el diccionario indicará cuál es su índice.

Paso 6: Reconstruimos la secuencia de caracteres a partir del Pair tratado.
Se inicializa la secuencia de caracteres, la cual se construirá desde el último carácter hasta el primero. Se añade el carácter leído a la secuencia.
Mientras (index != 0) {
    Leer del diccionario el pair con posición index.
    Index = pair.index.
    Se añade a la secuencia el pair.caracter.
}

Paso 7: Se añade la secuencia de caracteres de manera inversa a la lista de Byte para escribir ordenadamente como debería ser la secuencia.

}

Paso 8: Se prepara el resultado.

- Se convierte la Lista de Bytes a un Byte Array para fijar el tamaño del resultado.
- Se finaliza el contador de tiempo.
- Se devuelve una estructura de datos con el Byte Array resultante y el tiempo invertido.

LZSS

El algoritmo LZSS es un algoritmo compresor sin pérdida. La idea principal de este algoritmo es que se va recorriendo el archivo y se buscan coincidencias significativas con lo que se encuentra en la parte ya recorrida del archivo. Se habla de coincidencias significativas porque codificar una coincidencia tiene un coste de dos bytes y un bit, por esta razón codificar una coincidencia de un solo byte no sería eficiente ya que resultaría en un byte extra de penalización.

El funcionamiento del algoritmo de compresión es el siguiente:
Por cada posición del array de bytes que se pasa como input, se recorre el searchbuffer (las posiciones anteriores a la actual limitadas por un número máximo o lo que se haya recorrido del archivo si este número no se alcanza) en busca una coincidencia con el byte actual.

En caso de éxito, se tiene una posición en el searchbuffer donde hay un byte que coincide con el de la posición del recorrido principal. Entonces se avanzan estas posiciones simultáneamente para comprobar si se sigue dando una coincidencia. Este "recorrido paralelo" termina porque los bytes de las posiciones ya no coinciden o porque se ha llegado a la medida máxima de coincidencia. En este momento se consulta si la longitud de la coincidencia es significativa, en caso de que no lo sea se considera que no hay coincidencia, si es significativa se pasa a codificar el match. Para hacerlo se pone un flag a uno en un bitset que nos indica los matches del archivo y se codifican en dos bytes el offset y la longitud del match encontrado. Esta codificación se guarda en el resultado de la compresión.

En caso de que no se haya encontrado ningún match significativo en todo el searchbuffer, se deja a 0 el flag del bitset de la posición correspondiente a la actual. El byte se escribe en el output de la compresión directamente porque no se ha encontrado coincidencia.

Al terminar el recorrido principal, se forma el array final que contiene tres componentes principales. En primer lugar 4 bytes que codifican la longitud del bitset de flags. A continuación se codifica en bytes el bitset y para terminar se añade el resultado del recorrido de codificación.

Para reconstruir todo el texto es más fácil y rápido que la compresión. Se trata de, en primer lugar volver a formar el biset a partir de los bytes que lo contienen. Una vez listo se debe recorrer a la vez los datos comprimidos y el bitset. Al hacer este recorrido nos indicará el bitset si lo que hay en la posición actual de los datos es una match, o un byte sin comprimir que se puede guardar directamente en la lista resultante. En el caso de que sea un match se decodifica el offset y la longitud del match, con estos datos se obtienen los bytes correspondientes a la coincidencia de la lista resultado.

Para entender mejor el funcionamiento de este algoritmo, a continuación se comenta el **pseudocódigo** de las dos funcionalidades principales del fichero LZSS.java que son la compresion y descompresión.

Estructuras de datos

Bitset:

En la descompresión se utiliza un bitset que se ha creado al comprimir el archivo, por las razones anterior expuestas en la compresión.

Se utilizan también listas de bytes y array de bytes por las razones que se comentan al inicio del documento.

Comprimir

Paso 1: Inicializar las estructuras de datos principales que utilizará el algoritmo, estas estructuras son las siguientes:

Se inicia contador de tiempo.
Inicializar lista result para almacenar el resultado
Inicializar Bitset vacío
Inicializar las medidas del search_buffer y el look_ahead_buffer que determinarán en esencia el % de compresión.

Paso 2: Recorrer el array de bytes con los datos originales, al hacer esto se buscarán coincidencias.

for(cada byte del byte[] que se nos pasa por parametro){
Determinar el searchbuffer máximo, entre el límite y lo que queda a la izquierda de la posición actual

Paso 3: Para cada posición del search buffer se buscan coincidencias como se explica a continuación:

for(para cada posición del searchbuffer){
Determinar longitud máxima del lookaheadbuffer
for(para cada posición del lookaheadbuffer){
Ir comprobando por cada posición si coincide el byte en el lookaheadbuffer y el searchbuffer, salir del for en cuanto dejen de coincidir
}
En el caso que se haya encontrado un match más largo que en las anteriores posiciones del searchbuffer, se guarda esta nueva longitud y el offset al que se encuentra respecto a la posición actual del byte[] inicial
}
if(se ha encontrado un match>=3 en el searchbuffer){
Marcar en el bitset la posición actual
Guardar en dos bytes 12 bits de offset i 4 bits de longitud de match mediante desplazamiento de bits.
Añadimos a la lista result estos dos bytes
posact_enbyte[]=posact_enbyte[]+lengthmatch-1 para que lo que la siguiente iteración del bucle ya lea un byte que no se ha procesado
}
Else{
Marcamos en el bitset como 0 la posición actual
Añadimos a la lista result el byte de la posición actual
}
posenbitset++

}

Paso 4: Se almacena todo el resultado en un gran byte array, en este se almacenarán tres partes importantes, el tamaño del bitset que hemos inicializado en el paso uno (se guarda en 4 bytes), el bitset en sí codificado en los bytes necesarios y por último el contenido de la lista resultado que hemos ido llenando.

Pasar la lista result a un array de bytes
Pasar a 4 bytes la longitud del bitset
Pasar a bytes el bitset
Almacenar en un gran byte array las tres partes importantes del resultado que son:
for(cada byte de los 4 que componen la medida del bitset){
    Añadir en el granbytearray
}
for(cada byte de los que componen la el bitset pasado a bytes){
    Añadir en el granbytearray
}
for(cada byte de los que componen el byte[result]){
    Añadir en el granbytearray
}

Paso 5: se finaliza el tiempo del proceso y se rellena el OutputAlgorimtmo que retornará el proceso.

Se finaliza el contador de tiempo.
Se devuelve OutputAlgorimtmo con el Byte Array resultante y el tiempo de la compresión

Estructuras de datos

bitset:

En la compresión se utiliza un bitset para poder almacenar los flags que nos indican si existe un match. La ventaja principal de esta estructura de datos es que me permite almacenar los flags usando solamente un bit por flag. Esto es mucho mejor que introducir un byte que actúe a modo de flag en antes de cada match o no match. Por esta razón hago uso del bitset y además lo almaceno al incio del código para leer con facilidad el bitset mientras descomprimo.
Se utilizan también listas de bytes y array de bytes por las razones que se comentan al inicio del documento.

Descomprimir

Paso 1: Inicializar las estructuras de datos principales que utilizará el algoritmo al descomprimir, estas estructuras son las siguientes:

Se inicia contador de tiempo.
Obtener los 4 primeros bytes del byte[] pasado por parametro
Transformar estos 4 bytes a un int para obtener el bitsetsize
Crear byte array para almacenar bitset
for(posiciones de la 4 a la 4+ bitsetsize del byte[] pasado por parametro){
    Almacenar byte en bitset byte[]
}
Inicializar un bitset con el bitset byte[]
Inicializar lista result para almacenar el resultado

Paso 2: Recorrer el array de bytes con los datos comprimidos, al hacer esto se encontraran conjuntos de offset+length y bytes sin comprimir. Para que tenga sentido se debe recorrer paralelamente el

bitset junto al byte array principa. Al hacer esto, los flags del bitset nos indican que es lo que encontremos, y lo vamos procesando como se ve a continuación.

for(cada byte a partir del (4+sizeofbitset) del byte[] que se nos pasa por parametro){
    if(la posición actual no representa un match en el bitset){
        Añadir directamente el byte de la posición act del byte[] pasado por parámetro en la lista result
    }
    Else{
        El byte de la posición actual y el siguiente codifican el offset y la longitud en 12 y 4 bits respectivamente
        Obtener un int offset y un int matchlength a partir de estos bytes mediante desplazamiento de bits
        Calcular el inicio del match a partir de la posición actual, la medida de result y la longitud del match
        Avanzar la posición en el byte[] inicial para pasar los dos bytes del offset
        for(start = posición del inicio del match; hasta (posición del inicio del match+matchlength); start++){
            Añadir el byte en la posición start de result a la lista result.
        }
    }
    Avanzamos la posición en el Bitset
}

Paso 3: Se almacena el resultado que ahora está contenido en una lista, en un byte array para devolverlo en el formato adecuado.

Crear un array result_final con el tamaño actual de la lista de bytes.
for(cada byte de los que componen la lista result){
    Añadir en el result_final
}

Paso 4: se finaliza el tiempo del proceso y se rellena el OutputAlgorimtmo que retornará el proceso.

Se finaliza el contador de tiempo.
Se devuelve OutputAlgorimtmo con el byte[] result_final y el tiempo de la descompresión

LZW

El algoritmo LZW es un algoritmo compresor sin pérdida. El beneficio principal del algoritmo LZW es que tan solo es necesario recorrer los datos de entrada una sola vez para realizar la codificación ya que partes de un diccionario precargado con los 256 caracteres (bytes) de ASCII, esto permite que el diccionario de cadenas pueda crearse sobre la marcha.

El funcionamiento del algoritmo para la codificación se basa en leer los datos de entrada, para cada byte se revisa si el conjunto de la cadena guardada con la suma del byte está en el diccionario para poder seguir añadiendo caracteres a esta cadena, ja que se busca la cadena más grande, si no se encuentra en el diccionario se guardar el conjunto anterior (cadena más larga hasta el momento) como nueva entrada de en el diccionario y se iguala la cadena con el código nuevo (último byte leído). Así sucesivamente hasta codificar toda la entrada y guardarla. Un diccionario lleno ocupa 65536 entradas de 4 bytes (cada integer), seran 256000 bytes.

Por otra lado, el funcionamiento del algoritmo para la descodificación del fichero input cuando este ya ha sido comprimido con LZW, se basa en leer la entrada como integers, una vez guardado el primer integer en una variable Old Code, para cada integer New Code se aplicará el algoritmo. También se ha precargado el diccionario con los 256 caracteres de ASCII. Si este New Code no está en el diccionario, se guarda en una cadena el conjunto de bytes traducidos de Old Code más el caràcter; si el New Code si està en el diccionario, se guarda en una cadena el conjunto de bytes traducidos de New Code. Se escribe en salida la cadena y se guarda en carácter el primer byte, ademàs de ir añadiendo conjuntos del Old Code más el carácter para completar el diccionario.

Para entender mejor el funcionamiento de este algoritmo, a continuación se comenta las estructuras de datos asociadas al algoritmo y el pseudocódigo de las dos funcionalidades principales del fichero LZW.java que son la compresión y descompresión.

Estructuras de datos

ByteBuffer
Se utiliza la clase ByteBuffer que permite empaquetar un conjunto de bytes para poder utilizar este ByteBuffer como clave del diccionario, como problema de esta estructura en el algoritmo está que se ha de convertir a bytebuffer cada vez que modificas la lista de bytes w.

HashMap
Se utiliza un HashMap<ByteBuffer, Integer> como diccionario del algoritmo por que los elementos que inserta en el map no tendrán orden específico (cadenas de caracteres), y además no aceptan claves duplicadas ni valores nulos. Como otra opción de una búsqueda de diccionarios en java, tenemos como otra opción de maps que no se necesita la ordenación de mayor a menor del TreeMap y no se necessita LinkedHashMap que tarda en la busqueda. Además, HashMap es parecido a HashTable, però como no utilizamos aplicación multihilo, es preferible el HashMap.

Comprimir

Paso 1 : Inicializar variables necessarias

Inicializar contador del tiempo actual
Inicializar salida con un **ArrayList<byte>** donde se guardaran los datos codificados
Inicializar el diccionario con los elementos ASCII en un **HashMap<ByteBuffer,Integer>** y añadimos los 256 caràcteres (byte))
Inicializar w con el primer byte leído, se utiliza un **ArrayList<Byte>** para guardar el primer elemento de los datos de entrada en w y se crea el **ByteBuffer** de w
Inicializar el code a 256, se utiliza un **Integer** para agregar posteriormente conjuntos al **Map**.

Paso 2: Leer todos los datos de la entrada en bucle para tratar cada byte mediante el algoritmo LZW

```
MIENTRAS (haya bytes de byte [] de entrada por leer )
        Guardar byte leído en k
        Agregar k a la lista w y crear el ByteBuffer w+k
        SI (w+k está en el diccionario) ENTONCES
                w = (w+k)
        Se iguala el ByteBuffer de w a el ByteBuffer de wk
        SINO
                Añadir w en salida
                Se guarda en una lista de datos de salida los datos del ByteBuffer w
                Agregar el conjunto de (w+k, code) al diccionario
                w = k
                Se reescribe en la lista w el byte k y se guarda en el ByteBuffer w
        FIN_SI
FIN_MIENTRAS
Añadir bytes del ByteBuffer  w en el lista<byte> salida
```

Paso 3: Crear un OutputAlgoritmo para escribir el tiempo que se ha tardado en comprimir el archivo y el byte [] de los datos de la lista de bytes salida, que son los datos de entrada comprimidos, qu e finalmente se devolvera.

```
Crear array de bytes result con el tamaño de salida.
PARA(cada byte de salida){
        Añadir byte en result
}
Calcular tiempo de compresión
Devolver OutputAlgorimtmo con result y tiempo de compresión
```

Descomprimir

Paso 1: Inicializar variables necesarias

Inicializar contador del tiempo actual
Inicializar salida con un **ArrayList<byte>** donde se guardaran los datos codificados
Inicializar diccionario con elementos ASCII, se utiliza un **HashMap<Integer,ByteBuffer>** y añadimos los 256 caràcteres (byte).
Inicializar **integer** code_old a null y code_new vació
Inicializar el code a 256, se utiliza un **Integer** para agregar posteriormente conjuntos al **Map**.
Inicializar **array de bytes** c de tamaño integer - 4.

Paso 2: Leer todos los datos de la entrada en bucle para tratar cada byte mediante el algoritmo de descompresión de LZW

```
PARA (cada dato de entrada que leer)
    Meter byte leído i de la entrada en posición x del byte [] c
    SI ( x es igual a 3) ENTONCES
        Convertir byte [] c de tamaño 4 a un integer ni
        SI2 (es el primer integer que se convierte) ENTONCES
            Code_old se iguala a ni
            carácter=Traducir(code_old)
            Guardar carácter en lista<Byte> de salida
        SINO2
            Code_new se iguala a ni
            SI (code_new no está en el diccionario) ENTONCES
                letras=Traducir(code_old) + caracter
            SINO
                letras=Traducir(code_new)
            FINSI
            Se añade letras a la lista <byte> salida
            carácter=Primer byte de letras
            Agregar conjunto (CODE, Traducir(code_old)+carácter) al diccionario
            code_old=code_new
        FIN_SI2
    Igualar variable x a -1 para que en la próxima iteración empieza en 0
    FIN_SI
FIN_PARA
```

Paso 3: Crear un OutputAlgoritmo para escribir el tiempo que se ha tardado en comprimir el archivo y el byte [] de los datos de la lista de bytes salida, que son los datos de entrada comprimidos, que finalmente se devolverá.

Crear **array de bytes** result con el tamaño de salida.
PARA(cada byte de salida) {
    Añadir byte en result
}
Calcular tiempo de compresión
Devolver **OutputAlgorimtmo** con result y tiempo de compresión