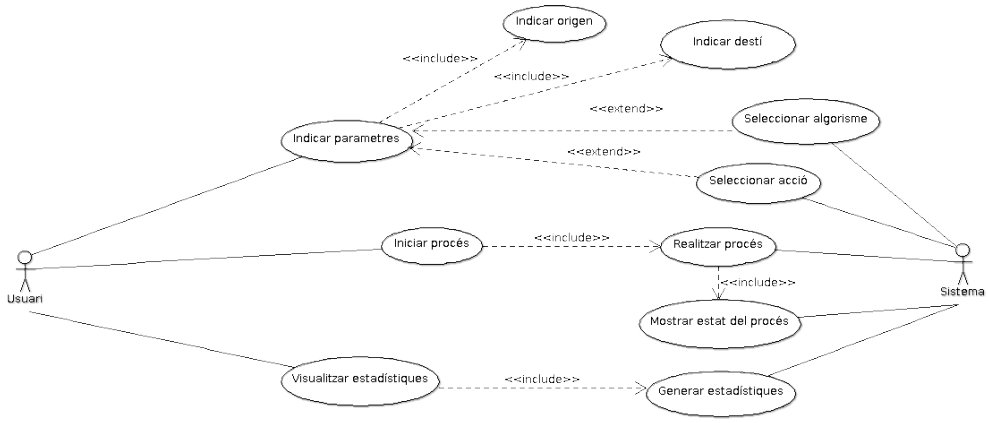
PROP – Primera Entrega – Grup 1.5

Marc Simó Guzmán
Daniel Boté Mayer
Marc Jiménez Gimeno
Aleix Pérez Vidal

Índex

Casos d'Ús – Esquema

L'esquema de casos d'ús del nostre projecte és el següent:



Adjuntem imatge en la carpeta DOCS

Casos d'Ús – Documentació

Títol:	Indicar paràmetres
Descripció:	L'usuari indica els paràmetres que vol utilitzar en el procés a realitzar
Actor primari:	Usuari
Precondicions:	-
Postcondicions:	S'han guardat els paràmetres indicats
Escenari principal d'èxit	L'usuari ha indicat una serie de paràmetres necessaris per executar el procés i s'han guardat
Freqüència d'ús:	Cada vegada que es vulgui canviar els paràmetres del procés

Títol:	Indicar origen
Descripció:	S'indica d'on s'ha de llegir l'objecte sobre el que és realitzarà l'acció
Actor primari:	Usuari
Precondicions:	-
Postcondicions:	S'ha guardat la direcció de l'objecte d'entrada
Escenari principal d'èxit	L'usuari indica la direcció de l'objecte sobre el que és realitzarà l'acció i aquesta és guarda
Freqüència d'ús:	Cada vegada que es vulgui canviar l'objecte d'entrada

Títol:	Indicar destí
Descripció:	S'indica on s'ha de guardar l'objecte resultant de realitzar l'acció sobre l'objecte d'entrada
Actor primari:	Usuari
Precondicions:	-
Postcondicions:	S'ha guardat la direcció de sortida
Escenari principal d'èxit	L'usuari indica la direcció on es guardarà l'objecte resultant de realitzar l'acció sobre l'objecte d'entrada i aquesta és guarda
Freqüència d'ús:	Cada vegada que es vulgui canviar la direcció de sortida

Títol:	Seleccionar algorisme
Descripció:	S'indica quin algorisme és vol utilitzar per dur a terme l'acció
Actor primari:	Usuari/Sistema
Precondicions:	-
Postcondicions:	S'ha guardat l'algorisme a utilitzar
Escenari principal d'èxit	El sistema indica segons el tipus d'entrada quin algorisme seria més convenient utilitzar, i posteriorment, l'usuari pot canviar aquesta selecció i escollir l'algorisme que vulgui.
Freqüència d'ús:	Cada vegada que es vulgui canviar l'algorisme a utilitzar

Títol:	Seleccionar acció
Descripció:	S'indica quina acció és vol dur a terme (comprimir/descomprimir)
Actor primari:	Usuari/Sistema
Precondicions:	-
Postcondicions:	S'ha guardat l'acció a realitzar
Escenari principal d'èxit	El sistema indica segons el tipus d'entrada quina acció seria convenient realitzar, i posteriorment, l'usuari pot canviar aquesta selecció.
Freqüència d'ús:	Cada vegada que es vulgui canviar l'acció a realitzar

Títol:	Iniciar procés
Descripció:	L'usuari indica que comenci la realització del procés determinat pels paràmetres indicats
Actor primari:	Usuari
Precondicions:	Haver seleccionat destí, origen, algorisme i acció, i que siguin valors vàlids.
Postcondicions:	El sistema comença la realització del procés
Escenari principal d'èxit	Els paràmetres indicats són correctes i estan complets i es pot començar la realització del procés.
Freqüència d'ús:	Sempre que es vulgui comprimir o descomprimir.

Títol:	Realitzar procés
Descripció:	Es realitza el procés determinat pels paràmetres indicats per l'usuari
Actor primari:	Sistema
Precondicions:	Haver seleccionat destí, origen, algorisme i acció, i que siguin valors vàlids. I que l'usuari hagi indicat que vol que s'inicie el procés.
Postcondicions:	És genera un nou objecte
Escenari principal d'èxit:	S'ha llegit l'objecte indicat en destí i mitjançant l'algorisme indicat s'ha realitzat l'acció seleccionada, guardant l'objecte resultant en la destinació indicada.
Freqüència d'ús:	Sempre que es vulgui comprimir o descomprimir.

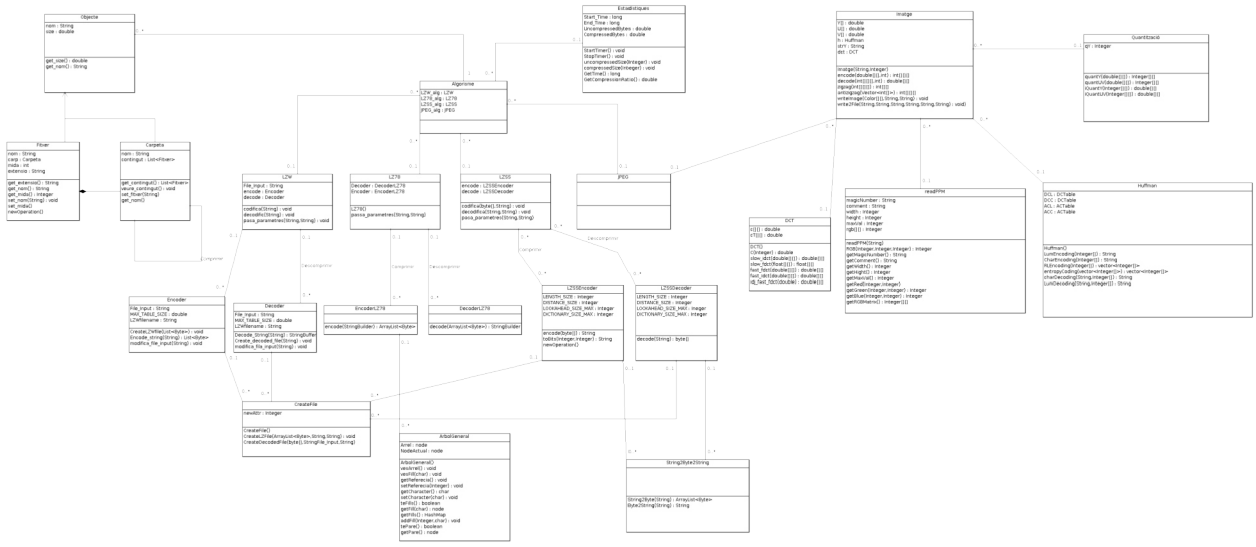
Títol:	Mostrar estat del procés
Descripció:	Es visualitza gràficament i en viu, l'estat en que es troba el procés durant la seva execució.
Actor primari:	Sistema
Precondicions:	Que s'estigui executant el procés.
Postcondicions:	S'ha visualitzat amb èxit el procés.
Escenari principal d'èxit	Durant l'execució del procés s'ha estat visualitzant en tot moment l'estat en que es trobava i el percentatge de l'estat que ja estava complet.
Freqüència d'ús:	Sempre que es vulgui comprimir o descomprimir.

Títol:	Visualitzar estadístiques
Descripció:	Es visualitzen les estadístiques que genera el sistema sobre els processos realitzats.
Actor primari:	Usuari
Precondicions:	-
Postcondicions:	S'han mostrat per pantalla les estadístiques obtingudes dels processos realitzats.
Escenari principal d'èxit	S'obtenen les estadístiques a representar i és mostren per pantalla.
Freqüència d'ús:	Sempre que es vulguin veure les estadístiques

Títol:	Generar estadístiques
Descripció:	A partir de les dades recavades dels processos realitzats es generen estadístiques
Actor primari:	Usuari
Precondicions:	L'usuari ha indicat que vol visualitzar les estadístiques
Postcondicions:	S'han generat les estadístiques.
Escenari principal d'èxit	A partir de les dades recavades dels processos realitzats el sistema realitza una serie d'operacions per obtindre les estadístiques corresponents a aquestes dades.
Freqüència d'ús:	Sempre que es vulguin veure les estadístiques

Diagrama de Classes UML – Esquema

El diagrama de classes UML del nostre projecte és el següent:



Adjuntem imatge en la carpeta DOCS

Diagrama de Classes UML – Explicació Classes

LZW:
La classe LZW codifica o descodifica segons els paràmetres que se li passin.

Encoder:
Codifica el fitxer d'entrada i crea un nou fitxer codificat amb extensió LZW.

Decoder:
Descodifica el fitxer en format LZW i crea un de nou amb els valors descodificats del fitxer. El fitxer nou està amb extensió .txt.

LZ78:
Classe de l'algorisme LZ78 que rep el nom d'un fitxer sobre el que realitzar l'acció i si aquesta és codificar o descodificar amb l'algorisme LZ78 segons el valor del paràmetre acció. Llavors realitza l'acció (cridant a EncoderLZ78 o DecoderLZ78) i ho guarda en un nou fitxer amb extensió .lz78 o l'extensió original segons corresponga a compressió o descompressió.

EncoderLZ78:
Classe que codifica el text d'entrada d'acord amb l'algorisme LZ78 i retorna la codificació.

DecoderLZ78:
Classe que descodifica el text d'entrada codificat amb l'algorisme LZ78 i retorna la descodificació.

ArbolGeneral:
Classe que representa un arbre N-ari i que utilitzem per realitzar la codificació amb LZ78 de forma eficient. Els nodes tenen com atributs un enter referencia, un char caràcter, el node pare i un HashMap amb els fills del node identificats pel seu caràcter.

LZSS:
Classe de l'algorisme LZSS que rep un fitxer sobre el que realitzar l'acció, essent aquesta codificar (creant un fitxer amb extensio .lzss) o decodificar (creant un fitxer _decodificat.txt).

LZSSEncoder:
Classe que codifica el fitxer d'entrada d'acord amb l'algorisme LZSS i retorna un String amb la codificació.

LZSSDecoder:
Classe que descodifica un fitxer d'entrada (donat en Bytes) codificat amb l'algorisme LZSS i retorna la descodificació.

CreateFile:
Classe que donats el nom fitxer, el resultat de comprimir o descomprimir, implementa les funcionalitats de crear un fitxer comprimit amb nom fitxer i l'extensió que li es indicada, o crear un fitxer descomprimit amb nomFitxer_decoded_AlgorismeEmpleat.txt.

String2Byte2String:
Classe per millorar la presentació de l'algorisme LZSS, el que fa es transforma un String codificat en binari de 0 i 1s, a un ArrayList<Bytes> per escriure a un fitxer o al reves, llegint de un fitxer de Bytes, genera un String de 0 i 1s per descodificar.

Classe Imatge:
Conté matrius 2d que representen els eixos Y,U i V referents a la imatge llegida. A l'hora de comprimir, per cada eix, l'organitza en blocs de 8x8, s'hi apliquen a cada un d'ells la DCT i la quantització adequada. Finalment es codifica cada eix per Huffman. Per realitzar la descompressió es llegeixen d'un fitxer els tres eixos codificats, s'hi aplica la quantització i DCT inversa i es genera una imatge.

Classe readPPM:
Donada una imatge en format PPM s'encarrega de llegir-la correctament en una matriu RGB.

Classe DCT:
És on s´hi troben els mètodes per realitzar la DCT (discrete cosine transform) i la seva inversa.

Classe Huffman:
És on s´hi troben els mètodes per poder codificar i descodificar per Huffman. Es consulten diferents documents de text per poder realitzar les correspondències necessàries per a una codificació correcta.

Classe Quantitzacio.
En aquesta classe s'hi quantitzen i desquantitzen matrius 2D multiplicant-les per uns coeficients qY (luminància) i qC(crominància).

Estructures de Dades utilitzades

Motiu utilització estructures de dades:

- StringBuilder: Perquè és més ràpida que StringBuffer al no fer sincronització de threads, que nosaltres no en fem,i és molt més ràpida que String, a més de permetre llegir sense problemes textos de qualsevol tamany.

- Vector<>: Pel seu tamany variable i la possibilitat d'accedir a qualsevol element del vector mitjançant l'index.

- Pair<>: Per retornar dues objectes diferents en funcions.

- Triplet<>: Per retornar tres objectes diferents en funcions.

- String: Per emmagatzemar conjunts reduïts de caràcter i operar d'una forma més dinàmica, després s'escriuran en l'StringBuilder.

- Map<,>: Estructura que emmagatzema valors i claus i permet trobar els valors amb la clau d'identificació. En el nostre cas ens ha servit de diccionari.

- HashMap<>: Estructura que emmagatzema valors i claus i permet trobar els valors amb la clau d'identificació. En el nostre cas ens ha servit de diccionari. L'hem preferit en la utilització perquè és més eficient i té més funcions que el map normal i t'ofereix temps constant per les operacions de get i set que són les que utilitzem.

- List: La llista és una variable que permeten emmagatzemar grans quantitats de dades. A més, no cal especificar-li una mida concreta per utilitzar-la. Era necessària per certes funcions i com no li cal una mida concreta i a mesura que es vagin afegint elements es fiquen a l'ultima posició de la fila, és molt útil.

- ArrayList<> : Perquè no té una mida fixa i és eficient a l'hora de realitzar molts accessos i afegir nous elements al final. L'hem preferida al array normal degut a la seua millor eficiència i facilitat d'ús.

- byte[]: Encara que és més ineficient i bàsic que l'ArrayList, és necessari per certes operacions i funcions.

- Iterator<>: L'iterador recorre tots els elements d'un array. En aquest cas ha servit per recórrer totes les posicions de la llista. El seu ús ve lligat de l'ús de la classe llista.

- BinaryIn: Per llegir a nivell de byte quan és necessari.

- BinaryOut: Per escriure a nivell de byte quan és necessari.

- ByteArrayOutputStream: Com a priori no sabem exactament el tamany del nostre fitxer de text, aquesta estructura de dades es bastant eficient sense necessitat d'especificar un tamany determinat, i es va modificant la mida a mesura que inserim Bytes.

- ByteBuffer: L'utilitzem per passar d'enter a vector de bytes, i perquè ens permet saber quan quedarà buit amb hasRemaining(), i alhora ens permet extreure elements actualitzant la posició.

- Matrius cuadrades de 2 i 3 dimensions: Perquè són necessàries per realitzar de forma eficient les transformacions requerides pel JPEG.

- FileOutputStream: Usem aquesta estructura de dades ja que, juntament amb BinaryOut, per escriure un array de bytes en un fitxer de text es dels mes eficients, i amb altres estructures que havíem provat no s'escrivien els bytes correctament a la sortida.

- FileInputStream: Utilitzem aquesta estructura de dades ja que, com BinaryIn, ens serveix per llegir un File escrit en Bytes sense errors i de manera eficient Byte per Byte.

- BufferedInputStream: Per llegir del FileInputStream.

- FileReader: Utilitzem aquesta estructura de dades per tal de llegir correctament els documents externs de crominància i luminància.

- BufferedReader: Per llegir del FileReader.

- BufferedInputStream i FileInputStream: Per a la lectura de la imatge inicial en format PPM.

Estructures de dades utilitzades en cada classe:

Classe LZ78:
- ArrayList<Byte> : Perquè no té una mida fixa i és eficient a l'hora de realitzar molts accessos i quan s'afegeixen nous elements al final. L'utilitzem per guardar la codificació en conjunt de bytes.

- StringBuilder: Perquè és més ràpida que StringBuffer al no fer sincronització de threads, que nosaltres no en fem,i és molt més ràpida que String. L'utilitzem per llegir i escriure texts llargs.

- BinaryIn: Per llegir a nivell de byte o char quan és necessari.

- BinaryOut: Per escriure a nivell de byte o char quan és necessari.

Classe LZ78Encoder:
- ArrayList<Byte>: Perquè no té una mida fixa i és eficient a l'hora de realitzar molts accessos i quan s'afegeixen nous elements al final. L'utilitzem per guardar la codificació en conjunt de bytes.

- StringBuilder: Perquè és més ràpida que StringBuffer al no fer sincronització de threads, que nosaltres no en fem,i és molt més ràpida que String. L'utilitzem per llegir i escriure texts llargs.

- ByteBuffer: Per passar d'enter a vector de bytes.

- Vector<>: Pel seu tamany variable i la possibilitat d'accedir a qualsevol element del vector mitjançant l'index. L'utilitzem quan volem poder accedir a qualsevol element del vector.

- ArbolGeneral: Classe creada per Marc Simó que representa un arbre N-ari, i que s'utilitza per dur a terme la compressió en LZ78 de forma eficient, en quant a la búsqueda de frases existents en el diccionari.

- Pair<>: Per retornar dues objectes diferents en funcions.

- Triplet<>: Per retornar tres objectes diferents en funcions.

Classe LZ78Decoder:
- ArrayList<Byte>: Perquè no té una mida fixa i és eficient a l'hora de realitzar molts accessos i quan s'afegeixen nous elements al final. L'utilitzem per guardar la codificació en conjunt de bytes.

- StringBuilder: Perquè és més ràpida que StringBuffer al no fer sincronització de threads, que nosaltres no en fem,i és molt més ràpida que String. L'utilitzem per llegir i escriure texts llargs.

- String: Per emmagatzemar conjunts reduïts de caràcter i operar d'una forma més dinàmica, després s'escriuran en l'StringBuilder.

- ByteBuffer: Per passar d'enter a vector de bytes.

- Vector<>: - Vector<>: Pel seu tamany variable i la possibilitat d'accedir a qualsevol element del vector mitjançant l'index. L'utilitzem quan volem poder accedir a qualsevol element del vector.

- Pair<>: Per retornar dues objectes diferents en funcions.

Classe ArbolGeneral:
- node: Classe creada per Marc Simó que conté els atributs que és necessari guardar en cada node d'un arbre N-ari, i un HashMap on conté els nodes dels fills i el caràcter que els identifica.

Classe node:
- HashMap<>: Perquè t'ofereix temps constant per les operacions de get i set que són les utilitzades. Conté els nodes dels fills i el caràcter que els identifica.

Classe LZW:
- List: La llista és una variable que permeten emmagatzemar grans quantitats de dades. A més, no cal especificar-li una mida concreta per utilitzar-la.

Classe Encoder:

- List<Byte>: Era el format de retorn de la funció Encode_String. El que fa es retornar una llista dels valors codificats. El fet de que no li calgui una mida concreta ha anat molt bé degut a que en un principi no es sap quants caràcters tindrà el codi a comprimir, en conseqüència a mesura que es vagin codificant es fiquen a l'ultima posició de la fila.

- Map<String, Byte>: Estructura que emmagatzemes valors i els pots trobar amb una clau d'identificació. En el nostre cas ens ha servit de diccionari. S'ha creat una taula on guardar tots els valors per codificar.

- HashMap<String, Byte>: Idem que el mapa.

- ArrayList<Byte>: Es un vector redimensionable. És el que permet que la llista pugui tenir mida indefinida.

- BinaryOut: S'ha utilitzat aquesta classe per tal d'escriure en un Buffer tots els valor codificats. La raó d'haver escollit aquesta classe i no un Buffer és per poder reduir el numero de dependències de la classe.

- Iterator<Byte>: L'iterador recorre tots els elements d'un array. En aquest cas ha servit per recórrer totes les posicions de la llista.

Classe Decoder:
- StringBuffer: S'ha decidit utilitzar aquest tipus de variable i no un String per tal de que no hi hagués problemes amb la mida de la paraula a decodificar.

- BinaryIn: S'ha utilitzat aquesta classe pel mateix motiu que BinaryOut. En aquest cas necessitam llegir del buffer i no escriure.

- List<Byte>: Explicat abans.

- BinaryOut: Explicat abans.

- Map<Byte, String>: Explicat abans.

- HashMap<Byte, String>: Explicat abans.

Classe CreateFile
- ArrayList<Byte>: Explicat abans.

- File_Input (String): String per que es un paràmetre que entrem per consola.

- BinaryOut: Explicat abans.

- FileOutputStream: Usem aquesta estructura de dades ja que per escriure un array de bytes en un fitxer de text es dels mes eficients, i amb altres estructures que havíem provat no s'escrivien els bytes correctament a la sortida.

Classe String2Byte2String:
- ArrayList<Byte>: Explicat abans.

- FileInputStream: Utilitzo aquesta estructura de dades ja que ens serveix per llegir un File escrit en Bytes sense errors i de manera eficient Byte per Byte, i així poder fer les transformacions pertinents per a després descodificar.

- StringBuilder: Explicat abans.

Classe LZSSDecoder:
- ByteArrayOutputStream: com a priori no sabem exactament el tamany del nostre fitxer de text, aquesta estructura de dades es bastant eficient sense necessitat de especificar un tamany determinat, i es va modificant el tamany a mesura que inserim Bytes.

- byte[]: utilitzem un byte array com a finestra lliscant, ja que sabem específicament la mida que ha de tenir, i hi ha funcions que optimitzen el temps que es triga en "desplaçar els elements" que realment es copien, com la funció arraycopy.

- String: Usem un string en aquest cas per tenir el codi codificat en 0 i 1s, ja que així ens permet anar cridant la funció substring, i anar reduint la mida d'aquest, en funció que els anem descodificant.

Classe LZSSEncoder:
- ByteBuffer: L'utilitzem per que ens permet saber quan quedarà buit amb hasRemaining(), i alhora ens permet extraure elements actualitzant la posició amb position(novaPosicio).

Classe Imatge:
- Matrius: Moltes matrius 2D per poder gestionar correctament cada píxel de la imatge, i quan és necessari un vector de matrius 8x8 per poder operar emprem una matriu 3D per poder emmagatzemar-les òptimament.

- BufferedInputStream i FileInputStream: Per a la lectura de la imatge comprimida.

Classe readPPM:
- BufferedInputStream i FileInputStream: Per a la lectura de la imatge inicial en format PPM.

Classe Huffman:
- FileReader i BufferedReader: Per a llegir correctament els documents externs de crominància i luminància.

Descripció dels Algoritmes

LZW:

L'algorisme LZW el que fa és que donat un document de M lletres, inicialitza una taula, en el nostre cas anomenada diccionari, assignant a cada lletra un codi de 0 a M-1.
Després inicialitza la primera lletra de la frase, serà la variable L per a l'explicació.
Si tenim una variable X què és el següent caràcter en la frase.
Si LX és una paraula del nostre diccionari, assignem a L aquesta variable LX i tornem a que X és el següent caràcter de la frase.
En cas que LX no estigui al diccionari, s'afegeix al diccionari assignant-li un codi y no utilitzat, es a dir codi de LX = y. La variable L passa a ser X i tornem a assignar a X el següent caràcter de la frase.

LZ78:

L'algorisme LZ78 és un algorisme de codificació sense perduda basat en la repetició que funciona de la següent manera:

Codificació: Utilitza un diccionari d'entrades del tipus {referencia,caràcter} per codificar, el qual es va generant a mesura que anem codificant l'entrada. Va llegint caràcters de l'entrada:

Si es llegix un caràcter i no està al diccionari s'afegix una entrada al diccionari amb la referencia 0.
Si es llegix una frase o caràcter que ja està indexat en el diccionari, és seguix llegint caràcters i afegint-los al final de la frase/caràcter fins formar una frase que no està en el diccionari.
Si es llegix una frase/caràcter que existeix i al afegir l'últim caràcter conforma una frase que no existeix és guarda en el diccionari el caràcter nou amb la referencia a l'entrada en el diccionari de l'últim caràcter de la frase que si existeix.

Totes les entrades del diccionari ({referencia,caràcter}) s'escriuen en el fitxer de sortida en ordre d'inserció de l'entrada en el diccionari i conformen el fitxer codificat.

Descodificació: Utilitza també un diccionari d'entrades que va generant a mesura que llegeix el fitxer codificat. Cada combinació {referencia,caràcter} del fitxer de sortida conforma una nova entrada del diccionari de descodificació i a mesura que les anem llegint anem descodificant el codi de la següent manera:

Si la referencia és 0, simplement escrivim el caràcter en el fitxer de sortida i afegim la combinació {referencia,caràcter} al diccionari.
Si la referencia és diferent de 0, llegirem el caràcter de la referencia i l'escriurem abans que el nostre caràcter, i si la referencia del nou caràcter tampoc és 0 anirem al caràcter referenciat per aquest i l'escriurem abans que aquest nou caràcter, i així successivament fins que la referencia sigui 0. Una vegada la referencia sigui 0, s'escriurà la frase llegida en el fitxer de sortida i afegirem la combinació {referencia, caràcter} al diccionari.

D’aquesta manera acabarem descodificant tot el text.

LZSS:

Dividim aquest algorisme en dues parts, i s'utilitzarà una o l'altra en funció dels paràmetres que li passin.

LZSSEncoder que s'encarrega de la compressió d'un fitxer de text, que guarda aquesta compressió en un nou fitxer amb el mateix nom, però amb extensió .lzss

LZSSDecoder que s'encarrega de la descompressió d'un fitxer amb extensió .lzss, que guarda aquesta descompressió en un nou fitxer amb el mateixNom_decoded_lzss.txt

La part de compressió del algorisme funciona tal que tenim un ByteBuffer sobre el que diferenciarem entre els elements que ja hem codificat i els que encara no.
El algorisme es una millora del LZ77, millorant de tal manera, que si el símbol següent que falta per veure no coincideix amb cap element dels que ja hem vist, codificarem un bit 0, i el símbol.
En canvi, si el símbol a veure te coincidència dins del ja vistos, i hi ha un nombre significatiu de elements següents a aquest símbol, que també tinguin coincidència dintre dels ja vistos (consecutius, per tant de ser eficient), codificarem un bit 1, la longitud que te la coincidència, i la distancia a la que es troba del nostre inici del element. Tot això ho codificarem de manera binaria amb 0 i 1s dintre un StringBuilder que després una altre funció dintre de CreateFile s'encarregara de escriure un fitxer de Bytes de manera eficient.

La part de descompressió del algorisme funciona tal que farem el procés contrari, tindrem un String de 0 i 1s que el col·locarem dintre de una finestra lliscant definida com a un byte[] que te dues parts, Diccionari i LookAheadBuffer, amb capacitat de TamanyDiccionari + TamanyLookAheadBuffer.
Aquest TamanyDiccionari ve definit per 2^NombreBits codifiquem la distancia.
Aquest TamanyLookAheadBuffer ve definit per 2^NombreBits codifiquem la llargària coincidència.
Anirem llegint pas a pas quin es el següent element al lookAheadBuffer. Comencem llegint si el següent element al lookAheadBuffer es un 0 o 1, depenent de quin dels dos sigui, actuarem de manera diferent.
Per un 0, llegirem 8 bits mes, que serà la codificació binaria en ASCII de el element del text original, i l'escriurem a un ByteArrayOutputStream que després escriurem pel fitxer de sortida.
Per un 1, llegirem NombreBits en que codifiquem la distancia, que serà la distancia a la que hi ha coincidència, i llegirem NombreBits en que codifiquem la llargària de la coincidència. Anirem a aquesta distancia dins del diccionari respecte al primer element del lookAheadBuffer, i escriurem els següents "llargària" elements al nostre ByteArrayOutputStream.
El ByteArrayOutputStream contindrà els elements del fitxer descodificat, que després una funció dins de CreateFile creara el fitxer descomprimit.

JPEG

L’algorisme JPEG és un algorisme amb perduda que serveix per comprimir imatges. El seu funcionament és el següent:
Primerament necessitarem passar la foto de l’espai de color RGB a YUV. Parteix les matrius Y,U i V en blocs de 8x8 i les guarda en un vector. Després aplica la DCT i quantitza per cada bloc. Seguint amb el mateix paradigma per cada bloc, es recorre cada submatriu en zigzag per poder realitzar la codificació per Huffman. Finalment tindrem al nostre fitxer comprimit 3 cadenes de caràcters diferenciades per dos separadors.

Altrament, per descomprimir aquesta imatge, haurem de seguir inversament els passos de la compressió, és a dir, descodificar Y, U, V per Huffman, recórrer les submatrius en zig-zag invers, aplicar la quantització inversa, la DCT inversa i finalment, passar de YUV a RGB per a la generació correcta d’imatges.

Relació Classes-Membre

Objecte.java → Daniel Boté
Fitxer.java → Daniel Boté
Carpeta.java → Daniel Boté
Algorisme.java → Daniel Boté
LZW.java → Daniel Boté
Encoder.java → Daniel Boté
Decoder.java → Daniel Boté
DriverEncoder.java → Daniel Boté
DriverDecoder.java → Daniel Boté
DriverLZW.java → Daniel Boté

LZ78.java → Marc Simó
EncoderLZ78.java → Marc Simó
DecoderLZ78.java → Marc Simó
ArbolGeneral.java → Marc Simó
Pair.java → Marc Simó
Triplet.java → Marc Simó
Driver_ArbolGeneral.java → Marc Simó
Driver_DecoderLZ78.java → Marc Simó
Driver_EncoderLZ78.java → Marc Simó
Driver_LZ78.java → Marc Simó

LZSS.java → Marc Jiménez
LZSSEncoder.java → Marc Jiménez
LZSSDecoder.java → Marc Jiménez
CreateFile.java → Marc Jiménez
String2Byte2String → Marc Jiménez
Estadistiques.java → Marc Jiménez
DriverLZSS.java → Marc Jiménez
DriverLZSSEncoder.java → Marc Jiménez
DriverLZSSDecoder.java → Marc Jiménez
DriverString2Byte2String → Marc Jiménez
DriverEstadistiques.java → Marc Jiménez

Imatge.java → Aleix Pérez
DCT.java → Aleix Pérez
Quantitzacio.java → Aleix Pérez
readPPM.java → Aleix Pérez
Huffman.java → Aleix Pérez
DriverImatge.java → Aleix Pérez
DriverDCT.java → Aleix Pérez
DriverQuantitzacio → Aleix Pérez
Driver_readPPM → Aleix Pérez
DriverHuffman → Aleix Pérez