



Instituto Politécnico Nacional
Escuela Superior de Cómputo



Práctica 3: Codificación de Huffman



Integrantes:

Martinez Partida Jair Fabian

Martinez Rodriguez Alejandro

Monteros Cervantes Miguel Angel

Ramirez Cotonieto Luis Fernando

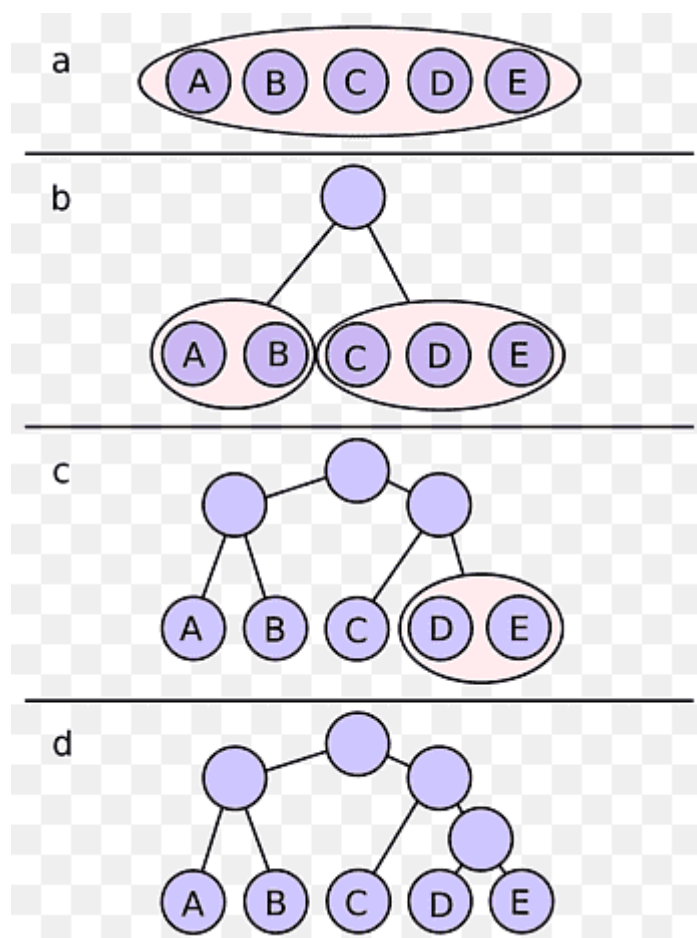
3CM13

01-Junio-2021

Algoritmo de Huffman.

Es un algoritmo utilizado para el cifrado de datos por medio de la frecuencia de aparición de caracteres y su clasificación en un árbol binario.

A través de la frecuencia de aparición de caracteres en determinada cadena, el algoritmo de Huffman calcula los pesos del conjunto de símbolos y los coloca en un árbol binario. El ordenamiento de los caracteres se balancea de forma que los caracteres más solicitados, son los que contienen menor cantidad de caracteres binarios.



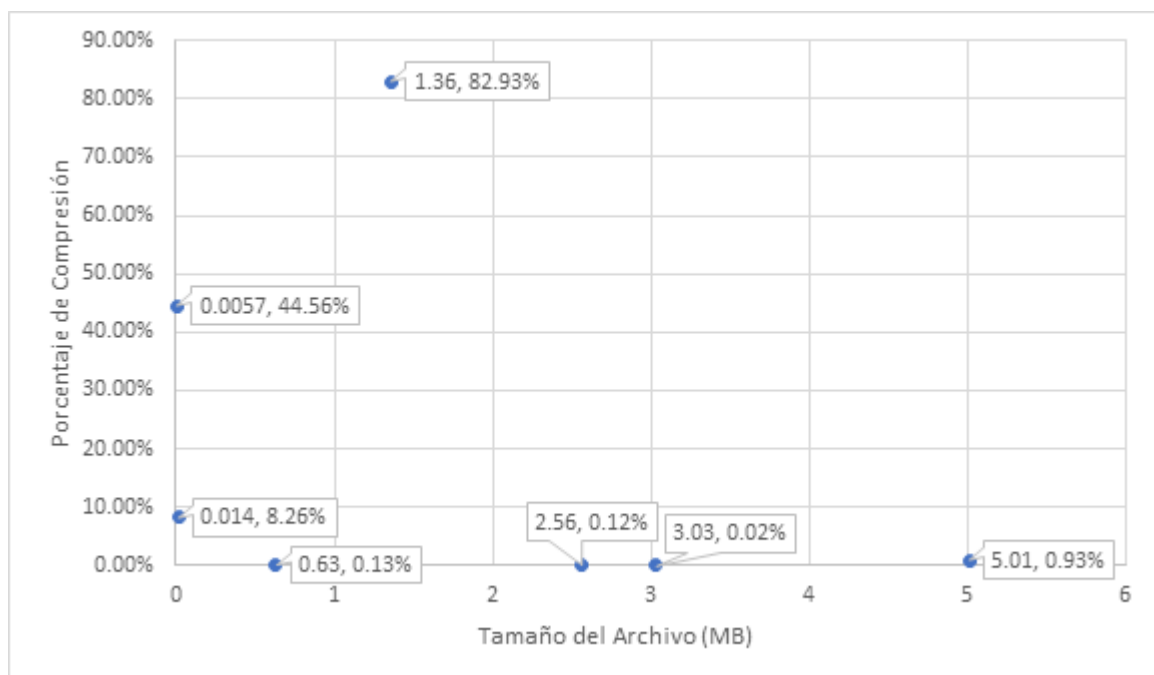
La complejidad del algoritmo de Huffman es: $O(n \log(n))$

Desarrollo de la Práctica.

Archivos usados.








Extensión de Archivo	Peso en MB
TXT	0.0057
MP3	5.01
BMP	1.36
JPG	0.63
PDF	2.56
DOCX	0.014
PNG	3.03

Gráfica de Porcentaje de Compresión.



Como podemos ver el tamaño del archivo no es influyente en que tanto se va a comprimir nuestro archivo, esto depende de que tan óptimo sea ese archivo un archivo más óptimo tendrá una compresión menor a uno que no lo es.

Tiempos de ejecución para la codificación y decodificación de diferentes archivos procesados por el programa.

Archivo	Codificación(segundos)	Decodificación(segundos)
	0.008s	0.002s
	5.571s	0.857s
	0.515s	0.025s
	0.682s	0.110s
	2.793s	0.385s
	0.051s	0.022s
	3.220s	0.452s

Complejidad de las funciones implementadas en el programa.

- ★ Complejidad de la función `main()`: $O(n \log n)$
- ★ Complejidad de la función `imprimeTablaFrecuencias`: $O(n)$
- ★ Complejidad de la función `genera frecuencias()`: $O(n^2)$
- ★ Complejidad de la función: `insertaFrecuencia`: $O(n)$
- ★ Complejidad de la función `vectorFrecuencia()`: $O(n^2)$
- ★ Complejidad de la función `ordenaFrecuenciasPorAparicion`: $O(n^2)$
- ★ Complejidad de la función `crearArbolCodificacion()`: $O(n)$
- ★ Complejidad de la función `generaUnicoArbol()`: $O(n^2)$
- ★ Complejidad de la función `reordenaArboles()`: $O(2 \log(n)) = O(\log n)$
- ★ Complejidad de la función `obtieneCaminoR()`: $O(1)$
- ★ Complejidad de la función `ordenaVectorFrecuenciasPorCaracter()`: $O(n^2)$
- ★ Complejidad de la función `escribeBits()`: $O(n)$
- ★ Complejidad de la función `asignaCodificacion()`: $O(n)$
- ★ Complejidad de la función `buscaFrecuencias()`: $O(n)$
- ★ Complejidad de la función `decodificaBits`: $O(n^2)$
- ★ Complejidad de la función `iniciaCola()`: $O(1)$
- ★ Complejidad de la función `formaCodificacion()`: $O(1)$
- ★ Complejidad de la función `formalInicioCodificacion()`: $O(1)$
- ★ Complejidad de la función `atiendeCola()`: $O(n)$
- ★ Complejidad de la función `atiendeTodos()`: $O(n)$
- ★ Complejidad de la función `pon1()`: $O(1)$
- ★ Complejidad de la función `pon0()`: $O(1)$
- ★ Complejidad de la función `valorBits()`: $O(1)$

Preguntas.

i. ¿Los niveles de codificación de archivos proporcionan una ventaja respecto al tamaño del archivo original en el promedio de los casos?

Si, en la mayoría de los casos, sí

ii. ¿Los tiempos de codificación o decodificación del archivo son muy grandes?

Realmente no, pero todo depende del tamaño del archivo que estemos ingresando

iii. ¿Ocurrieron pérdidas de la información al codificar los archivos?

Aparentemente no se encontró ninguna pérdida de información en los archivos

iv. ¿El comportamiento experimental del algoritmos era el esperado?

Completamente el algoritmo cumplió con lo esperado.

¿Por qué?

Pues logró comprimir en distintos rangos, la información que era ingresada al programa

v. ¿Qué características debería tener una imagen BMP para codificarse en menor espacio?

La calidad o definición de los pixeles de la imagen sea de una calidad media o promedio.

vi. ¿Qué características debería tener un archivo de texto para tener una codificación en menor espacio?

El tipo de archivo que es, si tiene imagenes o si es un texto plano, etc

vii. De 3 aplicaciones posibles en problemas de la vida real a la codificación de Huffman.

La codificación de Huffman hoy en día se usa a menudo como un "back-end" para algún otro método de compresión. DEFLATE (algoritmo de PKZIP) y códecs multimedia como JPEG y MP3 tienen un modelo front-end y cuantificación seguido de codificación Huffman.

viii. ¿Existió un entorno controlado para realizar las pruebas experimentales? ¿Cuál fue?

Si, se realizó en una máquina virtual de una computadora con 16 Ram, Ryzen 2600 y GPU Radeon 5500xt.

ix. ¿Qué recomendaciones darían a nuevos equipos para realizar esta práctica?

Que sean pacientes, es un código bastante largo y que necesita bastante atención al momento de codificarse, pero no es imposible

Anexos.

Compilación:

Para la compilación del código se implementó en el archivo "huffman.sh" una serie de scripts para que de esta manera sea mas rapida la ejecución del programa para cada archivo de entrada que se quiera codificar así como una misma serie de script para llevar a cabo la descompresión. El comando a ejecutar es mediante la terminal de linux, accediendo a la ruta donde se encuentre el archivo "huffman.sh" posteriormente, se ejecutara con el comando "./huffman.sh", despues de la ejecucion observaremos como se ha generado un archivo de texto "registro.txt" en donde se guardo toda la información de los tiempos de compresión y descompresión así como el peso de cada archivo

Script de ejecución:

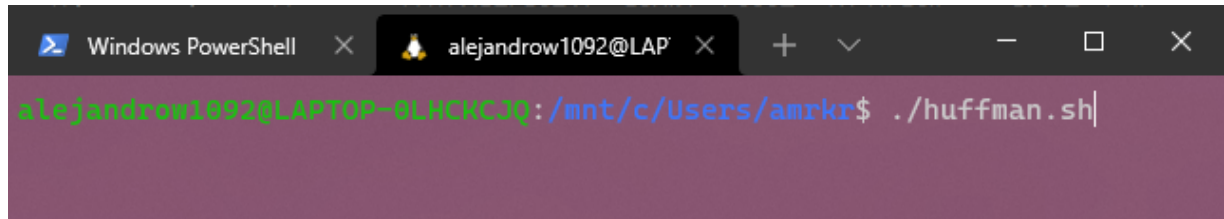
```
#!/bin/bash
gcc 'practica3.c' -o 'practica3' -w

./practica3 0 texto.txt texto_comp.txt caracteres1.txt >registro.txt
./practica3 0 cancion.mp3 cancion_comp.txt caracteres2.txt
>>registro.txt
./practica3 0 imagen1.bmp imagen1_comp.txt caracteres3.txt
>>registro.txt
./practica3 0 imagen2.jpg imagen2_comp.txt caracteres4.txt
>>registro.txt
./practica3 0 pdf.pdf pdf_comp.txt caracteres5.txt >>registro.txt
./practica3 0 word.docx word_comp.txt caracteres6.txt >>registro.txt
./practica3 0 imagen3.png imagen3_comp.txt caracteres7.txt
>>registro.txt

./practica3 1 caracteres1.txt texto_comp.txt texto_desc.txt
>>registro.txt
./practica3 1 caracteres2.txt cancion_comp.txt cancion_desc.mp3
>>registro.txt
./practica3 1 caracteres3.txt imagen1_comp.txt imagen1_desc.bmp
>>registro.txt
./practica3 1 caracteres4.txt imagen2_comp.txt imagen2_desc.jpg
>>registro.txt
./practica3 1 caracteres5.txt pdf_comp.txt pdf_desc.pdf >>registro.txt
./practica3 1 caracteres6.txt word_comp.txt word_desc.docx
>>registro.txt
```

```
./practica3 1 caracteres7.txt imagen3_comp.txt imagen3_desc.png  
>>registro.txt
```

Comando de ejecución del script en linux bash:

A screenshot of a Windows PowerShell terminal window. The title bar shows 'Windows PowerShell' and the user 'alejandrow1092@LAP'. The terminal prompt is 'alejandrow1092@LAPTOP-0LHCKCJQ:/mnt/c/Users/amrkr\$' and the command being executed is './huffman.sh'.

Código

```
#include <stdio.h>
#include <stdlib.h>
#include "practica3.h"

int main(int argc, char** argv) {
    char * salida;
    char * codificado;
    int leidos;
    float leidosTotal = 0;
    float bytesTotal = 0;
    int bitsEscritosReal = 0;
    Frecuencia * v;

    char opc1 = argv[1][0];

    if(opc1 == '1') // Decompresión
    {

        uswtime(&usertime11, &systime11, &walltime11);

        //Abrimos archivos
        tablaFrecuenciasArchivo = fopen(argv[2], "r");
        entrada = fopen(argv[3], "rb");
        out = fopen(argv[4], "wb");

        if(out == NULL || tablaFrecuenciasArchivo == NULL || entrada ==
        NULL) printf("Error\n");
```



```

    // Variables auxiliares para la descompresión
    long size;
    char character;
    int apariciones;
    int padding;
    int tamanoOriginal;
    int numeroCaracteres;
    char basura;

    // Obtenemos el tamaño de nuestro archivo codificado
    fseek (entrada,0,SEEK_END);
    size = ftell(entrada);
    rewind(entrada);

    // Leemos información importante sobre el archivo de salida
    fscanf(tablaFrecuenciasArchivo,"L:%d",&numeroCaracteres); // El
número de caracteres distintos en el archivo
    fscanf(tablaFrecuenciasArchivo,"%c",&basura); // Salto de línea
    fscanf(tablaFrecuenciasArchivo,"P:%d",&padding); // El número
de bits "basura"
    fscanf(tablaFrecuenciasArchivo,"%c",&basura); // Salto de línea
    fscanf(tablaFrecuenciasArchivo,"B:%d",&tamanoOriginal); // El
tamaño original del archivo comprimido

    salida = (char *)malloc(sizeof(char)*tamanoOriginal);
    codificado = (char *)malloc(sizeof(char)*size);

    // Reservamos el espacio para el arreglo de frecuencias
    v = (Frecuencia *)malloc(sizeof(Frecuencia) *
numeroCaracteres);

    // Lectura del archivo de la tabla de frecuencias para generar
un arreglo
    for(leidos = 0; leidos < numeroCaracteres; leidos++){
        fscanf(tablaFrecuenciasArchivo,"%c",&basura); // Salto de
línea
        // Obtenemos los caracteres y su número de apariciones,
para que a partir de ellos
        // se cree el árbol de decodificación

fscanf(tablaFrecuenciasArchivo,"%c|%d",&character,&apariciones);
        Frecuencia f;

```

```

        f.caracter = caracter;
        f.apariciones = apariciones;
        v[leidos] = f;
    }

    // Cerramos el flujo de la tabla de frecuencias
    fclose(tablaFrecuenciasArchivo);

    // Ordenamos el arreglo por las apariciones
    ordenaFrecuenciasPorAparicion(v, numeroCaracteres);
    // Creamos el árbol de codificación
    creaArbolCodificacion(v, numeroCaracteres);

    // Leemos el archivo codificado
    fread(codificado, sizeof(char), size, entrada);
    fclose(entrada); // Cerramos el flujo de salida
    decodificaBits(codificado, (size*8) - padding, salida);

    fwrite(salida, sizeof(char), tamanoOriginal, out); // Escribimos
    el archivo final
    fclose(out); // Cerramos el flujo

    // Concluimos la medición de tiempos
    uswtime(&usertime21, &systime21, &walltime21);

    ImprimirTiempos(usertime11, systime11, walltime11, usertime21,
    systime21, walltime21);

    // Imprimimos información
    printf("Se ha descomprimido el archivo %s con éxito, ahora es
    %s\n", argv[3], argv[4]);

}

else if(opc1 == '0') {
    uswtime(&usertime11, &systime11, &walltime11);
    ListaFrecuencia f;
    f.inicio = NULL;

    entrada = fopen(argv[2], "rb");

    if (entrada == NULL) {
        perror("No se puede abrir el fichero de entrada");
    }
}

```

```

        return -1;
    }

    uswtime(&usrtime11, &systime11, &walltime11);
    // Este primer ciclo se encarga de la lectura para generar el
árbol
    do {
        // Leemos el archivo hasta que ya no haya que leer
        // fread nos devuelve la cantidad de datos leídos
        leidos = fread(buffer, sizeof(char), BUFFER, entrada);
        leidosTotal += leidos;
        // Insertamos las frecuencias en nuestra tabla
        generaFrecuencias(&f,buffer,leidos);
        // Cuando no se hayan ocupado todos los espacios en la
lectura
        // sabemos que ya se terminó de leer el archivo
    }while(leidos == BUFFER);

    // Reservamos memoria suficiente para almacenar el archivo
codificado
    salida = (char *)malloc(sizeof(char) * leidosTotal);
    // Obtenemos el arreglo de frecuencias
    v = vectorFrecuencias(&f);
    // Generamos la codificación mediante la llamada a este método
    creaArbolCodificacion(v,f.length);
    // Regresamos al inicio del archivo
    leidos = 0;
    rewind(entrada);

    int resultadoEscritura = 0;
    // El segundo ciclo lee el archivo para generar la codificación
    do {
        intiaux = 0;
        // Leemos el archivo hasta que ya no haya que leer
        // fread nos devuelve la cantidad de datos leídos
        leidos = fread(buffer, sizeof(char), BUFFER, entrada);
        // Por cada caracter leído vamos a modificar los bits
dentro de este ciclo
        for(iaux = 0; iaux < leidos; iaux++){
            resultadoEscritura =
escribeBits(salida,leidosTotal*8,bitsEscritosReal, buffer[iaux], v,
f.length);

            if(resultadoEscritura < 0) break;

```

```

        bitsEscritosReal += resultadoEscritura;
    }
    // Cuando no se hayan ocupado todos los espacios en la
lectura
    // sabemos que ya se terminó de leer el archivo
    // Cuando no se puedan seguir escribiendo bits en el arreglo
de salida, rompemos el ciclo
    }while(leidos == BUFFER && resultadoEscritura >= 0);

    fclose(entrada); // Cerramos la entrada

    bytesTotal = (bitsEscritosReal / 8); // Determinamos cuantos
bytes ocupará el nuevo archivo
    int padding = ((int)(bytesTotal)) % 8;
    if(padding > 0)
        bytesTotal++;

    out = fopen(argv[3],"w"); // Abrimos un flujo de salida para
guardar el nuevo archivo
    fwrite(salida,sizeof(char),bytesTotal,out); // Escribimos el
arreglo de chars, el cual fue modificado para contener la información
    fclose(out); // Cerramos el flujo

    // Imprimimos la tabla de frecuencias a un archivo para
conservarla
    imprimeTablaFrecuencias(v,f.length, padding, leidosTotal
,argv[4]);

    // Concluimos la medición de tiempos
    uswtime(&usertime21, &systime21, &walltime21);

    ImprimirTiempos(usertime11, systime11, walltime11,usertime21,
systime21, walltime21);

    // Impresión de información relevante de la compresión
[Académico]
    float resta = (leidosTotal - bytesTotal)*100;
    float por ciento = resta/leidosTotal;
    if(resultadoEscritura >= 0)
        printf("Se ha comprimido %s con éxito, paso de %.0f bytes a
%.0f bytes (%.2f%), ahora se encuentra en
%s",argv[2],leidosTotal,bytesTotal,por ciento, argv[3]);

```

```

        else
            printf("NO se ha comprimido %s con exito, paso de %.0f
bytes a %.0f bytes, ahora se encuentra en
%s",argv[2],leidosTotal,bytesTotal, argv[3]);
    }

    return (EXIT_SUCCESS);
}

/**
 * Función para imprimir la tabla de frecuencias a un archivo
especificado
 * @param frecuencias La tabla de frecuencias en el archivo
 * @param ruta La ruta en la que se debe guardar
 */
void imprimeTablaFrecuencias(Frecuencia * frecuencias, int length, int
padding, int byteOriginal,const char ruta[]) {
    tablaFrecuenciasArchivo = fopen(ruta,"w");
    if (tablaFrecuenciasArchivo == NULL) {
        perror("No se puede abrir archivo");
    }
    fprintf(tablaFrecuenciasArchivo,"L:%d\n", length);
    fprintf(tablaFrecuenciasArchivo,"P:%d\n", padding);
    fprintf(tablaFrecuenciasArchivo,"B:%d\n", byteOriginal);
    // Impresión de las frecuencias al archivo especificado
    int i;
    for (i = 0; i < length; i++) {
        fprintf(tablaFrecuenciasArchivo,"%c|%d\n"
            , (int) frecuencias[i].caracter
            , frecuencias[i].apariciones
            /*, frecuencias[i].codigo*/);
    }
    fclose(tablaFrecuenciasArchivo);
}

/**
 * Función encargada de insertar los caracteres en una lista de
frecuencias
 * @param frecuencias Un apuntador a la lista
 * @param caracteres Los caracteres a insertar
 * @param length La longitud del arreglo de caracteres a insertar
 */

```

```

void generaFrecuencias(ListaFrecuencia * frecuencias, char
caracteres[], int length) {
    // Comprobamos que la lista esté iniciada, si no lo está, la
inicializamos
    if (frecuencias->inicio == NULL) {
        frecuencias->length = 0;
    }
    int ix;
    // Recorremos el arreglo de caracteres para insertarlos en la lista
de frecuencia
    for (ix = 0; ix < length; ix++) {
        insertaFrecuencia(frecuencias, caracteres[ix]);
    }
}

/**
 * Función para insertar (o actualizar) en la lista de frecuencias
 * @param frecuencias La lista de frecuencias
 * @param caracter El caracter que queremos insertar o actualizar
 */
void insertaFrecuencia(ListaFrecuencia * frecuencias, char caracter) {
    // Buscamos si la frecuencia ya existe
    int x;
    NodoFrecuencia * f;
    NodoFrecuencia * fanterior;
    ///DEBUG
    //imprimeListaF(frecuencias);

    for(f = frecuencias->inicio, fanterior = frecuencias->inicio; f !=
NULL; f = f->siguiente)
    {
        if (f->frecuencia.caracter == caracter) {
            // En caso de que exista solo incrementamos sus ocurrencias
            f->frecuencia.apariciones++;
            int ix = 0;
            // Comprobamos que la lista siga en orden, si no, la
acomodamos
            // TODO: Hacer un ciclo para mover el elemento hasta
que quede en orden
            // actualmente solo lo mueve una posición.
            if( f->siguiente != NULL

```

```

        && f->frecuencia.apariciones >
f->siguiente->frecuencia.apariciones)
    {

        NodoFrecuencia * auxiliar = f->siguiente;
        if(f == frecuencias->inicio)
        {
            // En caso de que el elemento que se tenga que
mover esté en el inicio de la lista
            frecuencias->inicio = f->siguiente;
            f->siguiente = auxiliar->siguiente;
            frecuencias->inicio->siguiente = f;
        }
        else
        {
            // Si el elemento se encuentra en otra posición
            fanterior->siguiente = f->siguiente;
            f->siguiente = fanterior->siguiente->siguiente;
            fanterior->siguiente->siguiente = f;
        }
    }
    // Salimos de la función
    return;
}

// Movemos el auxiliar a la siguiente posición
fanterior = f;
}

// Si se llega hasta este punto significa que el nodo no existía
anteriormente en la lista
// Insertamos el nodo en la lista, como es su primera aparición,
queda al inicio
NodoFrecuencia * aux = frecuencias->inicio;
f = (NodoFrecuencia *) malloc(sizeof (NodoFrecuencia));
f->frecuencia.caracter = caracter;
f->frecuencia.apariciones = 1;
f->siguiente = aux;
frecuencias->inicio = f;
frecuencias->length++;
}

/**

```

```

* Función para convertir de una lista de frecuencias a un vector de
frecuencias
* @param frecuencias La lista de frecuencias a convertir
*/
Frecuencia * vectorFrecuencias(ListaFrecuencia * frecuencias){
    // Reservamos la memoria suficiente para almacenar el arreglo
    Frecuencia * vector = (Frecuencia *) malloc(sizeof(Frecuencia) *
frecuencias->length);
    int i = 0;
    NodoFrecuencia * f;
    for(f = frecuencias->inicio; f != NULL; f = f->siguiente, i++){
        vector[i] = f->frecuencia;
    }
    ordenaFrecuenciasPorAparicion(vector, frecuencias->length);

    return vector;
}

void ordenaFrecuenciasPorAparicion(Frecuencia * vector, int length){
    Frecuencia temp;
    int i, j;
    // Bubble sort para ordenar el vector de frecuencias de menor a
mayor
    // TODO: Implementar un algoritmo de ordenación más eficiente
    for(i=0; i<length-1; i++){
        {
            for(j=0; j<length-1; j++){
                {
                    if(vector[j].apariciones > vector[j+1].apariciones)
                    {
                        temp=vector[j];
                        vector[j]=vector[j+1];
                        vector[j+1]=temp;
                    }
                }
            }
        }
    }
}

/**
* Genera el árbol de codificación
* @param frecuencias Un arreglo con las frecuencias ordenadas
previamente de forma ascendente

```



```

    * @param length La longitud del arreglo de frecuencias
    */
void creaArbolCodificacion(Frecuencia * frecuencias, int length) {
    int i;
    nodos = length;
    // Reservamos memoria
    bosqueHuffman = (NodoArbol **)malloc(sizeof(NodoArbol *) * length);
    // Como las frecuencias ya estaban ordenadas previamente, basta con
    asignarlas en ese orden dentro del nuevo arreglo
    for(i = 0; i < length; i++) {
        NodoArbol * hoja = (NodoArbol *)malloc(sizeof(NodoArbol));
        hoja->frecuencia = &(frecuencias[i]); // Guardamos la
        información del caracter y sus apariciones
        hoja->peso = frecuencias[i].apariciones; // Para el paso uno el
        peso es igual a la frecuencia del caracter
        hoja->nodo0 = NULL; // Son hojas
        hoja->nodo1 = NULL; // Son hojas
        hoja->visitado = 0; // Lo marcamos como no visitado
        bosqueHuffman[i] = hoja;
    }

    // Llamado a la función encargada de realizar las combinaciones
    generaUnicoArbol();
    //ordenaVectorFrecuenciasPorCaracter(frecuencias, length);
    for(i = 0; i < length; i++) {
        // Usaremos una cola para guardar la secuencia de 1 y 0
        ColaCodificacion * cola = (ColaCodificacion
        *)malloc(sizeof(ColaCodificacion));
        iniciaCola(cola);
        // Obtenemos el camino tantas veces como caracteres tenemos
        // En bosqueHuffman[0] está el árbol de codificación.
        obtieneCaminoR(bosqueHuffman[0], cola);
        asignaCodificacion(frecuencias, length, cola);
        // Liberamos la memoria de la cola
        free(cola);
    }
}

/**
 * Función que realiza la combinación de los árboles hasta llegar a un
 * único nodo
 */
void generaUnicoArbol() {

```

```

    int i = nodos, j;

    while(bosqueHuffman[1] != NULL){ // Mientras no tengamos un solo
    árbol

        // Generamos un nuevo árbol:
        NodoArbol * nuevaHoja = (NodoArbol *)malloc(sizeof(NodoArbol));
        // Como es un árbol resultado de la combinación de otros dos,
ya no deberá tener un caracter asociado
        nuevaHoja->frecuencia = NULL;
        nuevaHoja->peso = bosqueHuffman[0]->peso +
bosqueHuffman[1]->peso; // La suma de los pesos de sus dos hijos
        nuevaHoja->nodo0 = bosqueHuffman[0]; // Hijo de menor
frecuencia
        nuevaHoja->nodo1 = bosqueHuffman[1]; // Hijo de mayor
frecuencia
        nuevaHoja->visitado = 0; // Lo marcamos como no visitado

        // Dentro de este for iremos recorriendo todos, un lugar hacia
atrás cada vez
        for(j = 2; j < i; j++) {
            bosqueHuffman[j - 1] = bosqueHuffman[j];
        }
        // Borramos el último elemento de la lista
        bosqueHuffman[j-1] = NULL;
        // Y asignamos el nuevo al inicio
        bosqueHuffman[0] = nuevaHoja;
        // Llamamos a la ordenación de los árboles para reacomodarlos
según su peso
        reordenaArboles();
    }
}

/**
 * Función encargada de la reordenación de los árboles dentro del
arreglo
 */
void reordenaArboles(){
    // Bubblesort para ordenar los árboles
    // TODO: Implementar un algoritmo de ordenación más eficiente,
    // además, sabemos que hay un solo elemento desordenado
    (bosqueHuffman[0])

```

```

        //          podríamos auxiliarnos de eso para escribir la
ordenación más eficiente.
    NodoArbol * temp;
    int i,j;
    char cambio = 0;
    for(i = 0; i < nodos && bosqueHuffman[i] != NULL; i++){
        for(j = 0; j < nodos && bosqueHuffman[j+1] != NULL; j++)
        {
            // Comprobamos el peso del árbol
            if(bosqueHuffman[j]->peso > bosqueHuffman[j+1]->peso)
            {
                temp=bosqueHuffman[j];
                bosqueHuffman[j]=bosqueHuffman[j+1];
                bosqueHuffman[j+1]=temp;
            }
        }
    }
}

/**
 * Función que arma la codificación para cada caracter de forma
recursiva,
 * almacena el resultado en una cola para posteriormente recuperarlo
 * @param arbol El árbol donde está la codificación
 * @param cola La cola en la que se almacenarán la codificación
 */
void obtieneCaminoR(NodoArbol * arbol, ColaCodificacion * cola) {
    // Comprobamos si el árbol no ha sido visitado
    if(arbol != NULL && arbol->visitado == 0){
        // Si tiene hijo 0 y no ha sido visitado, nos movemos hacia el
        if(arbol->nodo0 && arbol->nodo0->visitado == 0)
        {
            // Insertamos un 0 en la cola
            formaCodificacion(cola, '0');
            //printf("0");
            obtieneCaminoR(arbol->nodo0, cola);
        }
        // Si tiene hijo 1 y no ha sido visitado, nos movemos hacia el
        else if(arbol->nodo1 && arbol->nodo1->visitado == 0)
        {
            formaCodificacion(cola, '1');
            //printf("1");
            obtieneCaminoR(arbol->nodo1, cola);
        }
    }
}

```

```

    }
    // Si no tiene hijos o ya fueron visitados
    else
    {
        // Lo marcamos como visitado
        arbol->visitado = 1;
        if(arbol->frecuencia != NULL)
        {
            // Insertamos el caracter en la cola, pero al inicio
            formaInicioCodificacion(cola, arbol->frecuencia->caracter);
            //printf(" %c\n", arbol->frecuencia->caracter);
        }
        // Insertamos el fin de línea como referencia
        formaCodificacion(cola, '\0');
    }
    // Comprobamos que sea un nodo tallo
    if(arbol->frecuencia == NULL)
        // Si sus hijos ya fueron visitados
        if(arbol->nodo0->visitado == 1 && arbol->nodo1->visitado ==
1)
            // Marcamos el padre como visitado
            arbol->visitado = 1;
    }
}

/**
 * Función para reordenar el vector de frecuencias de acuerdo a los
caracteres que contiene
 * usando el algoritmo de ordenación por inserción
 * @param frecuencias
 * @param length
 */
void ordenaVectorFrecuenciasPorCaracter(Frecuencia * frecuencias, int
length){
    // Inicializamos una variable
    // auxiliar para recorrer el arreglo
    int ix = 1;
    for(; ix < length; ++ix) // Recorremos el arreglo
    {
        // Asignamos el valor actual a
        // una variable temporal
        Frecuencia temp = frecuencias[ix];

```

```

        // Nos ubicamos en la posición anterior a
        // la actual para ir revisando los valores
        int j = ix - 1;
        while((frecuencias[j].caracter > temp.caracter) && (j >= 0))
        {
            // Nos vamos moviendo hacia atrás
            // hasta que encontremos la nueva
            // ubicación de nuestro valor
            frecuencias[j + 1] = frecuencias[j];
            j--;
        }
        // Lo asignamos
        frecuencias[j + 1] = temp;
    }
}

/**
 * Función para modificar los bits de un arreglo de caracteres de
 * acuerdo a los parámetros recibidos
 * @param salida Es un arreglo de caracteres al cual se le van a
 * modificar los bits
 * @param salidaLength El tamaño en bits del arreglo
 * @param pointer La posición a partir de la cual se comenzar a
 * modificar los
 * @param c El caracter a codificar
 * @param frecuencias El arreglo de las frecuencias
 * @param length El tamaño del arreglo de frecuencias
 */
int escribeBits(char * salida, int salidaLength, long pointer, char c,
Frecuencia * frecuencias, int length){
    int bitsescritos;
    int indiceFrecuencia = buscaFrecuencia(frecuencias, 0, length, c);
    Frecuencia f = frecuencias[indiceFrecuencia];
    for(bitsescritos = 0; f.codigo[bitsescritos] != '\0';
bitsescritos++)
    {
        if(f.codigo[bitsescritos] == '1')
        {
            pon1(salida,pointer + bitsescritos);
        }
        else if(f.codigo[bitsescritos] == '0')
        {
            pon0(salida,pointer + bitsescritos);
        }
    }
}

```

```

    }

    }

    return bitsescritos;
}

/**
 * Función encargada de asignar la secuencia de código correspondiente
a cada caracter
 * @param frecuencias Un arreglo de frecuencias ordenado
ascendentemente por el caracter
 * @param length El tamaño del arreglo de frecuencias
 * @param cola La cola que contiene el código
 */
void asignaCodificacion(Frecuencia * frecuencias, int length,
ColaCodificacion * cola){
    char caracter = atiendeCola(colas);
    int longitudCodigo = cola->count;
    int indiceFrecuencia = buscaFrecuencia(frecuencias, 0, length,
caracter);
    int i;
    frecuencias[indiceFrecuencia].codigo = (char
*)calloc(longitudCodigo-1,sizeof(char));
    for(i = 0; i < longitudCodigo; i++){
        frecuencias[indiceFrecuencia].codigo[i] = atiendeCola(colas);
    }
}

/**
 * Función para buscar un caracter dentro del arreglo de caracteres,
utiliza búsqueda lineal
 * @param frecuencias
 * @param inicio
 * @param final
 * @param c El caracter a buscar
 */
int buscaFrecuencia(Frecuencia * frecuencias, int inicio, int final,
char c) {
    int i = -1;
    // TODO: Implementar un mejor algoritmo de búsqueda
    for(inicio = 0; inicio < final; inicio++){
        {
            if(frecuencias[inicio].caracter == c){
                i = inicio;
            }
        }
    }
}

```

```

        break;
    }
}
return i;
}

/**
 * Función encargada de convertir de un arreglo de caracteres
 * codificado a uno sin codificar empleando
 * el árbol de codificación previamente generado
 * @param codificado El arreglo de caracteres leído del archivo
 * codificado
 * @param longitudCodificado El tamaño del arreglo codificado
 * @param salida El arreglo de caracteres al que se debe escribir el
 * resultado
 */
void decodificaBits(char * codificado, long longitudCodificado, char *
salida){
    long x = 0;
    int v;
    int charSalida = 0;
    // Nos posicionamos en el la raíz de nuestro árbol
    NodoArbol * aux = bosqueHuffman[0];
    // Navegamos entre los nodos de acuerdo al valor de los bits leídos
    // hasta terminar de recorrerlos todos
    for(x= 0; x < longitudCodificado;){
        while(aux->frecuencia == NULL){
            // Obtenemos el valor del bit y de acuerdo a este
            // nos movemos hacia la hoja 0 o 1 de nuestro árbol
            v = valorBit(codificado, x++);
            if(v == 0){ aux = aux->nodo0; }
            else if (v == 1){ aux = aux->nodo1; }
        }
        // Asignamos el caracter asociado al arreglo de salida
        salida[charSalida++] = aux->frecuencia->caracter;
        // Nos colocamos nuevamente en la raíz del árbol
        aux = bosqueHuffman[0];
    }
}

void iniciaCola(ColaCodificacion * cola) {
    cola->count = 0;
}

```

```

cola->inicio = NULL;
cola->final = NULL;
}

void formaCodificacion(ColaCodificacion * cola, char valor){
    NodoCola * nuevoNodo = (NodoCola *)malloc(sizeof(NodoCola));
    nuevoNodo->valor = valor;
    nuevoNodo->siguiente = NULL;
    if(cola->final == NULL) {
        cola->final = nuevoNodo;
        cola->inicio = nuevoNodo;
    }
    else{
        cola->final->siguiente = nuevoNodo;
        cola->final = nuevoNodo;
    }
    cola->count++;
}

void formaInicioCodificacion(ColaCodificacion * cola, char valor){
    NodoCola * nuevoNodo = (NodoCola *)malloc(sizeof(NodoCola));
    nuevoNodo->valor = valor;
    nuevoNodo->siguiente = cola->inicio;
    cola->inicio = nuevoNodo;
    cola->count++;
}

char atiendeCola(ColaCodificacion * cola){
    NodoCola * aux = cola->inicio;
    char val = cola->inicio->valor;
    cola->inicio = cola->inicio->siguiente;
    cola->count--;
    free(aux);
    return val;
}

void atiendeTodos(ColaCodificacion * cola){
    while(cola->inicio != NULL){
        atiendeCola(cola);
    }
    cola->count == 0;
}

```



```

/**
 * Función encargada de poner el bit elegido en 1
 * @param array El arreglo en el que queremos modificar
 * @param pos La posición del bit que queremos cambiar
 */
void pon1(char * array, int pos){
    // Obtenemos el índice dentro del arreglo
    int ix = pos / BYTE;
    // Obtenemos la posición del bit dentro del arreglo
    int i = pos - (BYTE * ix);
    i = BYTE - i - 1;
    PONE_1(array[ix],i);
}

/**
 * Función encargada de poner el bit elegido en 0
 * @param array El arreglo en el que queremos modificar
 * @param pos La posición del bit que queremos cambiar
 */
void pon0(char * array, int pos){
    // Obtenemos el índice dentro del arreglo
    int ix = pos / BYTE;
    // Obtenemos la posición del bit dentro del arreglo
    int i = pos - (BYTE * ix);
    i = BYTE - i - 1;
    PONE_0(array[ix],i);
}

/**
 * Función encargada de obtener el valor del bit en el arreglo
 * @param array El arreglo en el que queremos conocer
 * @param pos La posición del bit que queremos obtener
 */
int valorBit(char * array, int pos){
    // Obtenemos el índice dentro del arreglo
    int ix = pos / BYTE;
    // Obtenemos la posición del bit dentro del arreglo
    int i = pos - (BYTE * ix);
    i = BYTE - i - 1;
    return COGEBIT(array[ix],i);
}

```

Referencias consultadas:

→ B. (2017, 4 septiembre). *Algoritmo de Huffman - BackStreetCode*. Medium.

<https://medium.com/@aprendizaje.maq/algoritmo-de-huffman-8523c21a1b1a>