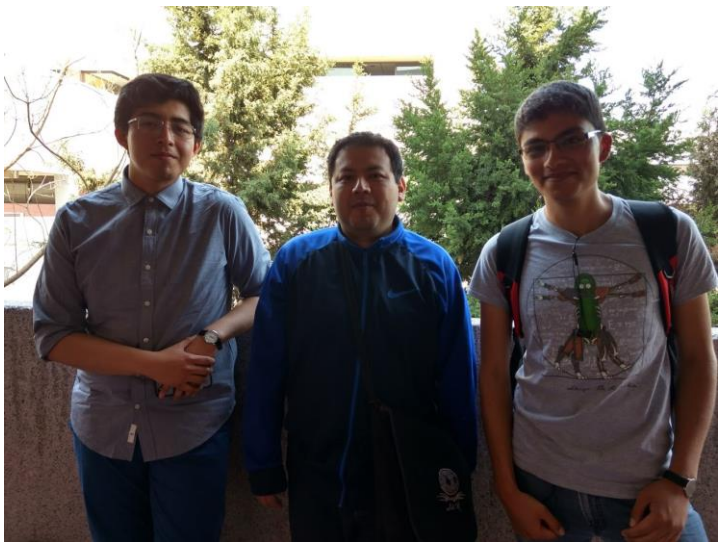
Instituto Politécnico Nacional

Escuela Superior de Cómputo

CODIFICACIÓN VORAZ DE HUFFMAN

Práctica 03



Análisis de Algoritmos

M. en C. Edgardo Adrián Franco Martínez
Grupo: 3CM3
Fecha: 18/Junio/2018

Equipo: Git Gud (Equipo Arbol)

- Calva Hernández José Manuel 2017630201
- Meza Madrid Raúl Damián 2017631051
- Montaño Ayala Alan Israel 2016630260

Índice

Introducción

Árbol Binario

El árbol es una estructura de datos fundamental en informática, muy utilizada en todos sus campos, porque se adapta a la representación natural de informaciones homogéneas organizadas y de una gran comodidad y rapidez de manipulación.

Esta estructura se encuentra en todos los dominios (campos) de la informática, desde la pura algorítmica (métodos de clasificación y búsqueda...) a la compilación (árboles sintácticos para representar las expresiones o producciones posibles de un lenguaje) o incluso los dominios de la inteligencia artificial (árboles de juegos, árboles de decisiones, de resolución, etc.).

Las estructuras tipo árbol se usan principalmente para representar datos con una relación jerárquica entre sus elementos, como son árboles genealógicos, tablas, etc.

Un árbol A es un conjunto finito de uno o más nodos, tales que:

1. Existe un nodo especial denominado RAIZ(v1) del árbol.
2. Los nodos restantes (v2, v3, ..., vn) se dividen en m >= 0 conjuntos disjuntos denominado A1, A2, ..., Am, cada uno de los cuales es, a su vez, un árbol. Estos árboles se llaman subárboles del RAIZ.

La definición de árbol implica una estructura recursiva. Esto es, la definición del árbol se refiere a otros árboles. Un árbol con ningún nodo es un árbol nulo; no tiene raíz.

Existe un tipo de árbol denominado árbol binario que puede ser implementado fácilmente en una computadora.

Un árbol binario es un conjunto finito de cero o más nodos, tales que:

- Existe un nodo denominado raíz del árbol.
- Cada nodo puede tener 0, 1 o 2 subárboles, conocidos como subárbol izquierdo y subárbol derecho. [1]

Algoritmo de Huffman

El algoritmo de Huffman es un algoritmo para la construcción de códigos de Huffman, desarrollado por David A. Huffman en 1952 y descrito en A Method for the Construction of MinimumRedundancy Codes.

Este algoritmo toma un alfabeto de n símbolos, junto con sus frecuencias de aparición asociadas, y produce un código de Huffman para ese alfabeto y esas frecuencias.

El algoritmo de Huffman es un algoritmo para la construcción de códigos de Huffman, desarrollado por David A. Huffman en 1952 y descrito en A Method for the Construction of Minimum-Redundancy Codes.

Este algoritmo toma un alfabeto de n símbolos, junto con sus frecuencias de aparición asociadas, y produce un código de Huffman para ese alfabeto y esas frecuencias.

Es un algoritmo usado para compresión de datos. El termino se requiere al uso de una tabla de códigos de longitud variable para codificar un determinado símbolo (como puede ser un carácter en un archivo), donde la tabla ha sido rellenada de una manera específica basándose en la probabilidad estimada de aparición de cada posible valor de dicho símbolo.

Huffman propuso un algoritmo voraz que obtiene una codificación prefijo-óptima.

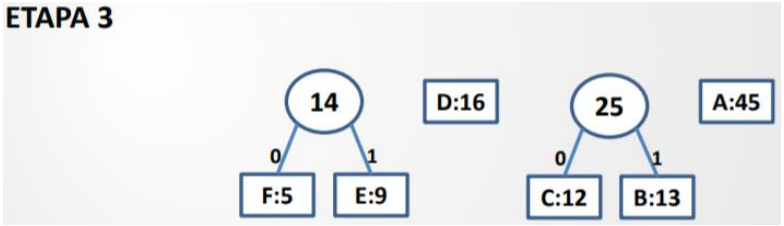
Para ello construye un árbol binario de códigos de longitud variable de manera ascendente.

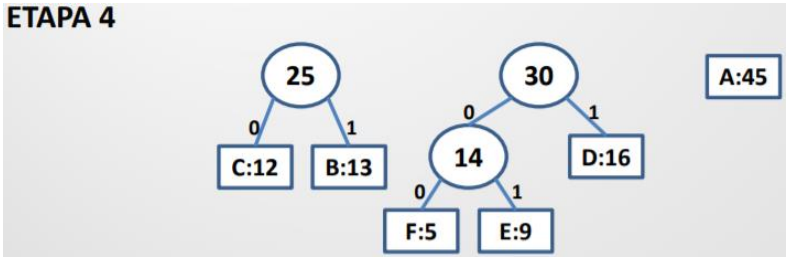
El algoritmo funciona de la siguiente manera:

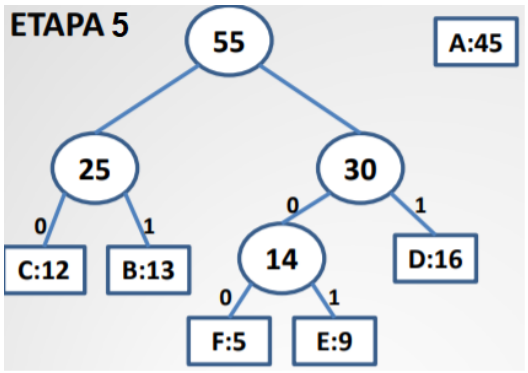
- Parte de una secuencia inicial en la que los caracteres a codificar están colocados en orden creciente de frecuencia.
    - Esta secuencia inicial se va transformando, a base de fusiones, hasta llegar a una secuencia con un único elemento que es el árbol de codificación óptimo.

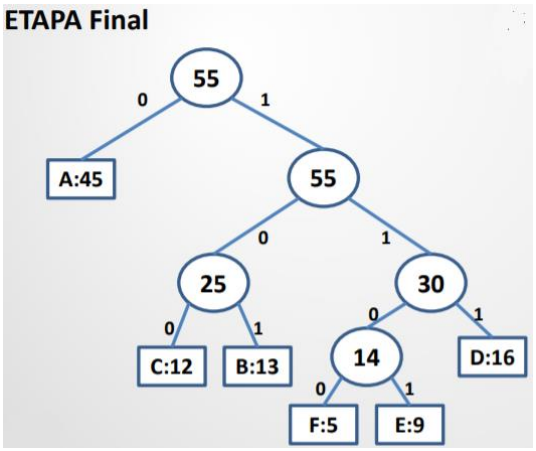
Pasos del código de Huffman

1. Se crean varios árboles, uno por cada uno de los símbolos del alfabeto, consistiendo cada uno de los árboles en un nodo sin hijos, y etiquetado cada uno con su símbolo asociado y su frecuencia de aparición.
2. Se toman los dos árboles de menor frecuencia, y se unen creando un nuevo árbol. La etiqueta de la raíz será la suma de las frecuencias de las raíces de los dos árboles que se unen, y cada uno de estos árboles será un hijo del nuevo árbol. También se etiquetan las dos ramas del nuevo árbol: con un 0 la de la izquierda, y con un 1 la de la derecha.
3. Se repite el paso 2 hasta que sólo quede un árbol.









Una vez terminado el árbol se puede ver con facilidad cuál es el código de los símbolos:

A: 0
B: 101
C: 100
D: 111
E: 1101
F: 1100

Decodificación del código de Huffman

1. A partir del árbol de codificación, comenzar a recorrer los caminos según los bits de la codificación. Al llegar a un nodo hoja se toma el valor de esta y coloca en el archivo original.
2. Se repite el paso 1 a partir del bit siguiente de la codificación comenzando un nuevo recorrido a partir de la raíz del árbol de la codificación.
3. La decodificación termina una vez se hallan recorrido todos los bits de la codificación. [2]

Planteamiento del problema

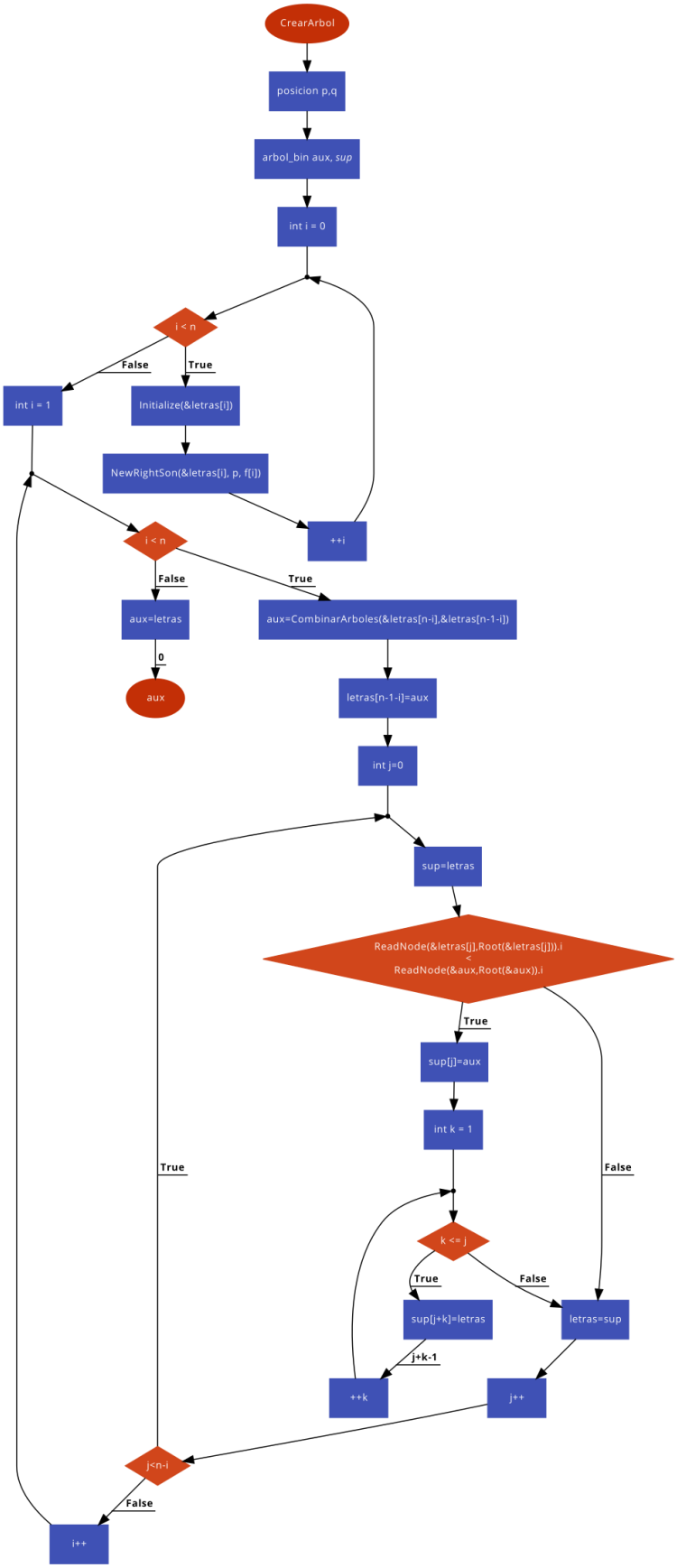
- Implementar el algoritmo de codificación de Huffman para codificar archivos de cualquier tipo bajo lenguaje C.
    - Implementar codificación voraz de Huffman
    - Implementar el algoritmo de decodificación
- Medir y comprobar las ventajas de tamaño de los archivos una vez realizadas diferentes codificaciones de archivos.
- Medir los tiempos de ejecución de las implementaciones (codificador y decodificador).

Algoritmos

In the pseudocode that follows, we assume that C is a set of n characters and that each character c∈C is an object with an attribute c:freq giving its frequency.

The algorithm builds the tree T corresponding to the optimal code in a bottom-up manner. It begins with a set of |C| leaves and performs a sequence of |C| 1 “merging” operations to create the final tree. The algorithm uses a min-priority queue Q, keyed on the freq attribute, to identify the two least-frequent objects to merge together. When we merge two objects, the result is a new object whose frequency is the sum of the frequencies of the two objects that were merged.

```
HUFFMAN.C
    1.  n← | C |
    2.  Q← | C |
    3.  for i← 1 to n 1
            1.  allocate a new node z
            2.  z.left← x← EXTRACT - MIN(Q)
            3.  z.right← y← EXTRACT - MIN(Q)
            4.  z.freq← x: freq + y: freq
            5.  INSERT(Q, z)
    4.  return EXTRACT - MIN(Q) // return the root of the tree
```


CrearArbol
posicion p,q
arbol_bin aux, sup
int i = 0
i < n
False
True
int i = 1
Initialize(&letras[i])
NewRightSon(&letras[i], p, f[i])
++i
i < n
False
True
aux=letras
aux=CombinarArboles(&letras[n-i],&letras[n-1-i])
0
aux
letras[n-1-i]=aux
int j=0
sup=letras
ReadNode(&letras[j],Root(&letras[j])).i
<
ReadNode(&aux,Root(&aux)).i
True
sup[j]=aux
int k = 1
True
False
k <= j
True
False
sup[j+k]=letras
letras=sup
j+k-1
++k
j++
j<n-i
False
i++

Implementación de los algoritmos

Codificación

```
1.  int main() {
2.      int j, k, l;
3.      char c, s[100];
4.      unsigned long i;
5.      arbol A[256];
6.      char cod[256][256];
7.      char b[256];
8.      FILE * f = NULL;
9.      FILE * f2 = NULL;
10.     lista * L;
11.     L = NULL; // Se lee el nombre del archivo desde consola.
12.     printf("Nombre del archivo (maximo 100 caracteres):\n");
13.     scanf("%s", s); // Abrimos el archivo en modo lectura.
14.     f = fopen(s, "rb"); // En caso de no ser válido, terminamos el programa.
15.     if (f == NULL) {
16.         printf("No existe el archivo.");
17.         return -1;
18.     } // Abrimos el archivo codificado en modo de escritura concatenándole .ggf como extensión de ide
    ntificación
19.     strcat(s, ".ggf");
20.     f2 = fopen(s, "wb"); // Medimos el tamaño del archivo
21.     int prev = ftell(f);
22.     fseek(f, 0 L, SEEK_END);
23.     unsigned long sz = (long) ftell(f);
24.     fseek(f, prev, SEEK_SET);
25.     printf("El archivo mide %ld bytes\n", sz); // Inicialización del arreglo de árboles
26.     for (i = 0; i < 256; i++) {
27.         A[i].c = decBin(i); // Colocamos el número en binario como caracter de la estructura
28.         A[i].n = 0; // Número de apariciones en 0
29.         A[i].der = A[i].izq = NULL; // Apuntadores a hijos en NULL
30.     } // Se pasa el contenido del archivo a un arreglo de caracteres
31.     char * entrada = (char * ) malloc(sizeof(char) * sz);
32.     fread(entrada, sizeof(char), sz, f); // Se hace el conteo de cada byte posible
33.     for (i = 0; i < sz; i++) {
34.         A[binDec(entrada[i])].n++; // Convertimos el caracter recibido a entero y aumentamos la frecu
    encia de ese árbol
35.     } // Se ordena el arreglo de arboles en orden ascendente de acuerdo al numero de apariciones del
    byte
36.     ordena(A, 0, 255); // Se añade la tabla de codificación al inicio del archivo
37.     for (i = 0; i < 256; i++) {
38.         fprintf(f2, "%c%10ld", A[i].c, A[i].n);
39.     } // Se incertan los árboles a una lista
40.     for (i = 0; i < 256; i++) {
41.         insertaLista( & L, A + i);
42.     } // Se van juntando los árbole hasta que quede uno solo
43.     while (L - > sig != NULL) {
44.         dosMenoresArboles( & L);
45.     } // Se recorre el árbol creando la tabla de codificación
46.     recorreArbol(L - > a, cod, b, 0); // Por cada Byte del archivo fuente se va añadiendo su codifica
    ción al archivo destino
47.     c = 0; // Caracter que guardará la codificación a anotar en el archivo
48.     l = 7; // Índice para saber qué bit deberá ser prendido
```

```
49.     for (i = 0; i < sz; i++) {
50.         k = binDec(entrada[i]); // Convertimos el caracter a entero para saber el índice a buscar
51.         for (j = 0; cod[k][j] != -1; j++, l--
    ) { // Recorreremos la posición indicada del arreglo de codificaciones
52.             if (l == -1) { // En caso de que hayamos terminado los bits disponibles del caracter
53.                 l = 7; // Reiniciamos el índice
54.                 fprintf(f2, "%c", c); // Imprimimos el caracter en el archivo
55.                 c = 0; // Reiniciamos el caracter
56.             } // Si todavía hay espacio en el caracter, prenderemos o mantendremos apagado el bit en
    el caracter
57.             c = c | (cod[k][j] << l);
58.         }
59.     } // Se añade el último byte al archivo destino
60.     fprintf(f2, "%c", c); // Cerramos ambos archivos utilizados.
61.     fclose(f);
62.     fclose(f2); // Liberamos la memoria dinámica solicitada
63.     free(entrada);
64.     return 0;
65. }
```

Decodificación

```
1.  int main() {
2.      int k, l;
3.      char c[11], d, s[100];
4.      unsigned long i, j;
5.      arbol * a;
6.      arbol A[256];
7.      FILE * f = NULL;
8.      FILE * f2 = NULL;
9.      lista * L;
10.     L = NULL; // Se lee el nombre del archivo desde consola.
11.     printf("Nombre del archivo:\n");
12.     scanf("%s", s); // Abrimos el archivo en modo lectura.
13.     f = fopen(s, "rb"); // En caso de no ser válido, terminamos el programa.
14.     if (f == NULL) {
15.         printf("No existe el archivo.");
16.         return -1;
17.     } // Recorremos el caracter NULL para eliminar la extensión incluída en la codificación
18.     s[strlen(s) - 4] = '\0'; // Abrimos el archivo donde decodificaremos
19.     f2 = fopen(s, "wb"); // Medimos el tamaño del archivo
20.     int prev = ftell(f);
21.     fseek(f, 0 L, SEEK_END);
22.     unsigned long sz = (long) ftell(f);
23.     fseek(f, prev, SEEK_SET);
24.     printf("El archivo a leer mide %ld bytes\n", sz); // Se pasa el contenido del archivo a un arregl
    o de caracteres
25.     char * entrada = (char * ) malloc(sizeof(char) * sz);
26.     fread(entrada, sizeof(char), sz, f); // Se cargan los datos al arreglo de árboles
27.     for (i = 0; i < 256; i++) {
28.         A[i].c = entrada[i * 11]; // Guardamos el caracter acorde a la posición
29.         strncpy(c, entrada + i * 11 + 1, 10); // Los siguientes 10 caracteres nos dirán su frecuencia

30.         A[i].n = atol(c); // Convertimos el caracter a entero
31.         A[i].der = A[i].izq = NULL; // Iniciamos los apuntadores a los hijos en NULL
```

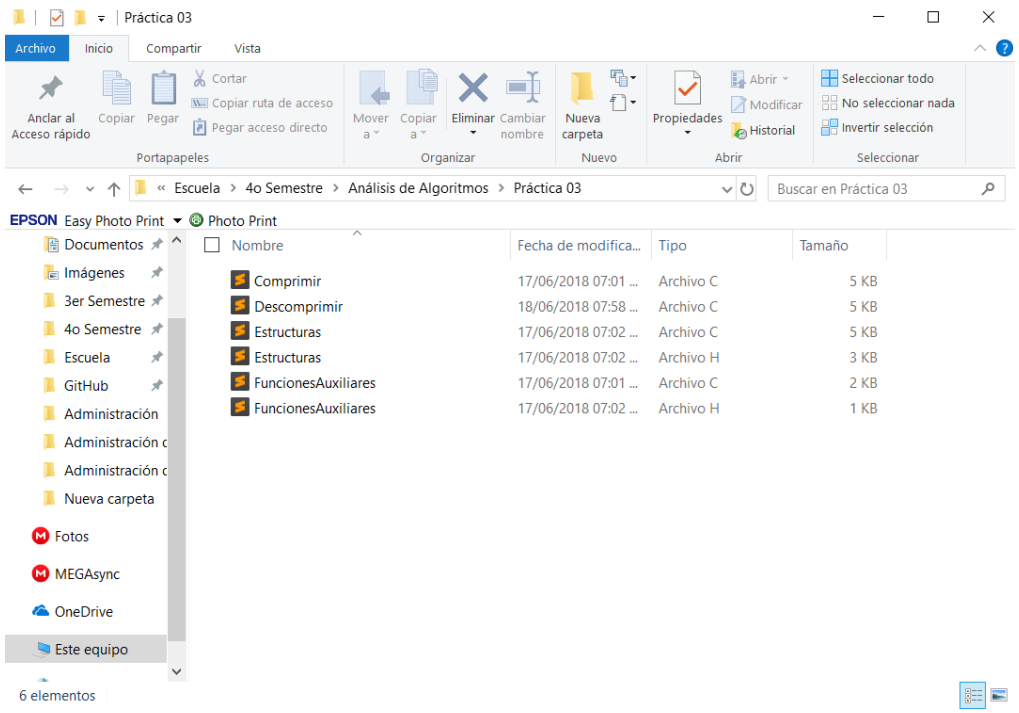
```
32.     } // Se ponen los árboles en una lista
33.     for (i = 0; i < 256; i++) {
34.         insertaLista( & L, A + i);
35.     } // Se van juntando los árbole hasta que quede uno solo
36.     while (L - > sig != NULL) {
37.         dosMenoresArboles( & L);
38.     }
39.     /*Se va recorriendo bit a bit el archivo fuente mientras    se va restaurando el original*/
40.     a = L - > a; // Copiamos el primer árbol de la lista
41.     arbol * aux = a; // Colocamos el apuntador a ese árbol // Iniciamos en el caracter a partir del c
    ual terminamos de tomar los valores de codificación // Concluiremos cuando ya hayamos llegado al valo
    r total de frecuencias
42.     for (i = 2816, j = 0; j < a - > n; i++) {
43.         d = entrada[i];
44.         for (k = 7; k > -1; k--) { // Analizaremos los 8 bits de cada caracter
45.             l = (d >> k) & 1; // Por medio de una máscara extraemos el bit a analizar
46.             if (!l) { // Si el bit es 0, bajaremos a la izquierda
47.                 aux = aux - > izq;
48.             } else { // Si es 1, bajaremos a la derecha
49.                 aux = aux - > der;
50.             }
51.             if (!(aux - > izq)) { // En caso de llegar a una hoja
52.                 j++; // Aumentamos n indicando que ya leímos una letra
53.                 fprintf(f2, "%c", aux - > c); // Imprimimos el caracter encontrado
54.                 aux = a; // Volvemos al nodo raíz
55.             }
56.         }
57.     } // Cerramos ambos archivos utilizados.
58.     fclose(f);
59.     fclose(f2); // Liberamos la memoria dinámica solicitada
60.     free(entrada);
61.     return 0;
62. }
```

Anexo

Compilación

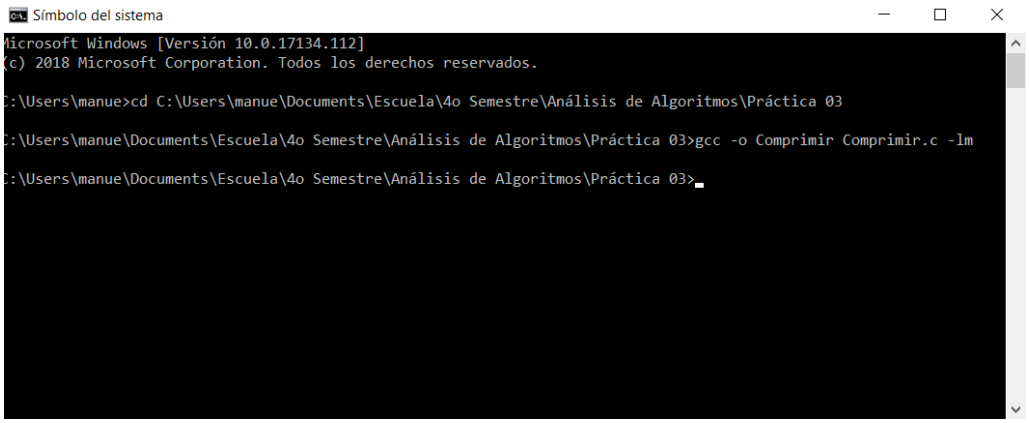
En Windows, ejecutar las siguientes instrucciones:

1. Colocar los archivos en una misma carpeta:

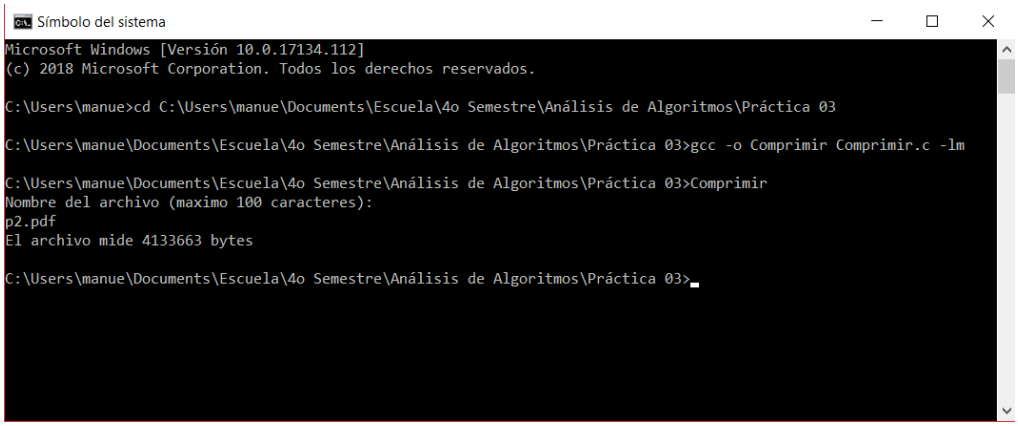


2. Acceder por medio de un terminal hasta la ruta donde se encuentren los archivos:

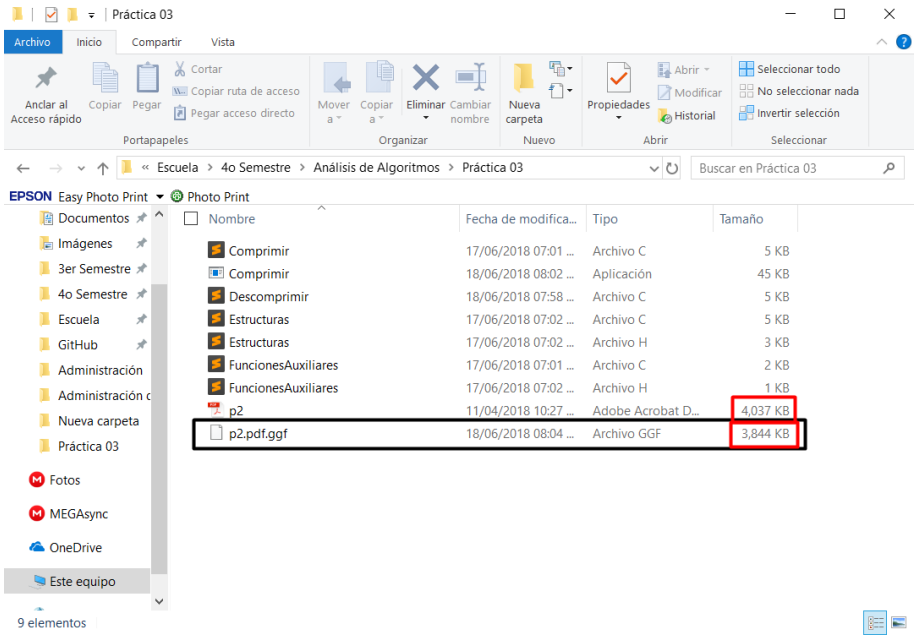
3. Escribir el siguiente comando para compilar el programa:



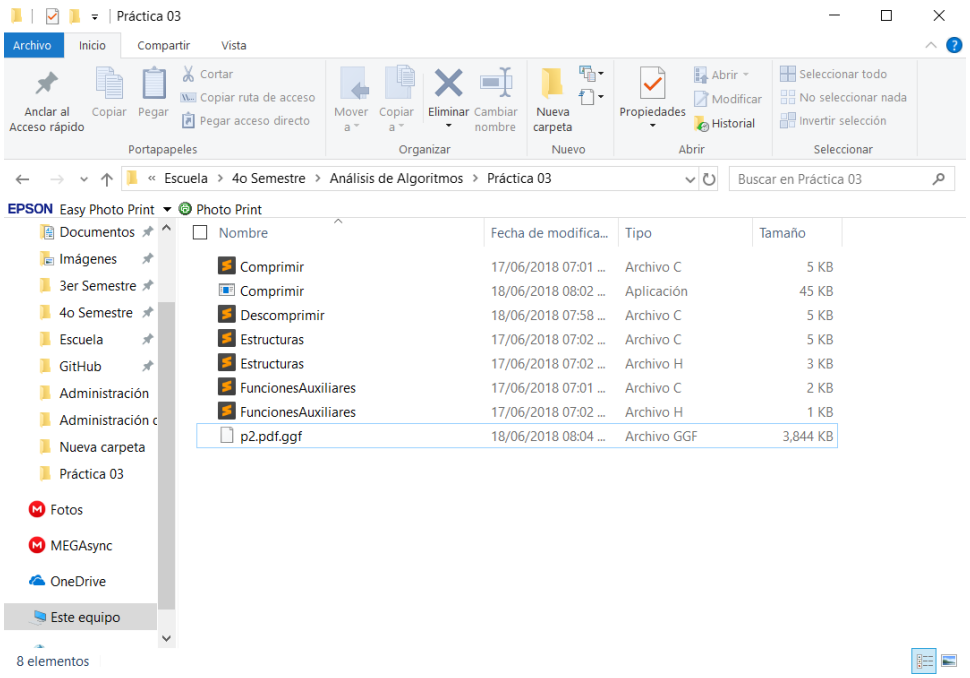
4. Ejecutamos el programa escribiendo adicionalmente el nombre del archivo que deseamos comprimir:



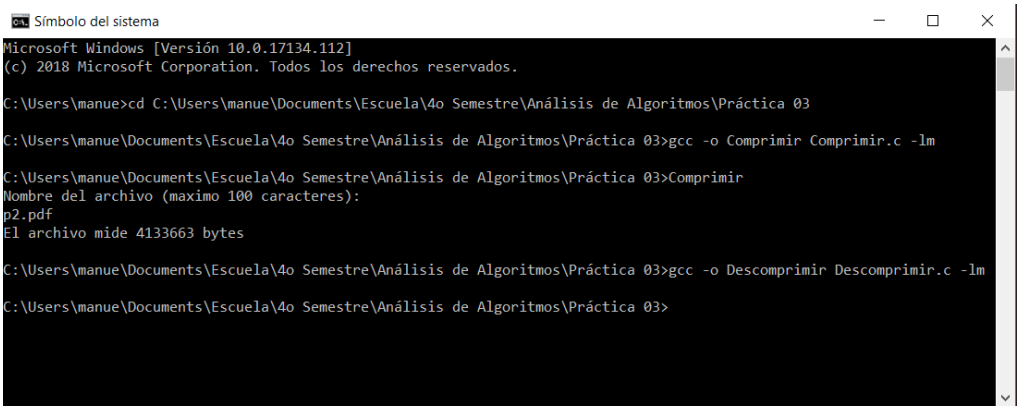
5. Verificar los resultados del código que resulta en un archivo con extensión .ggf:



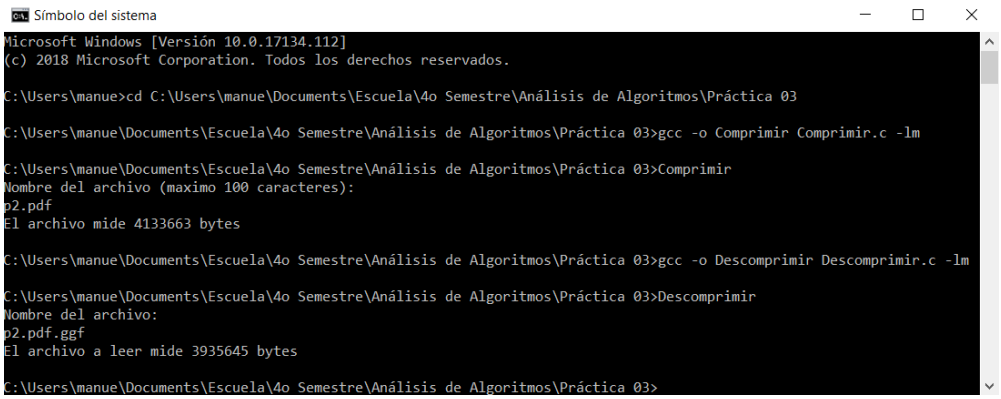
6. Procederemos a eliminar el archivo original y decodificar el archivo nuevo:



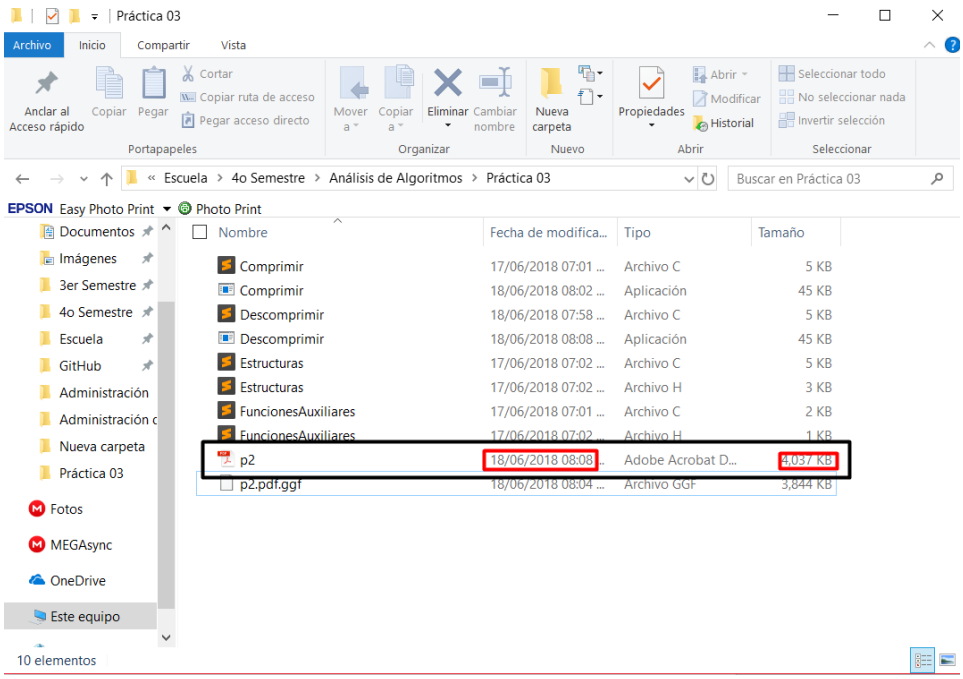
7. Escribiremos el siguiente comando en la terminal para compilar el decodificador:



8. Ejecutaremos el programa incluyendo el nombre del archivo a decodificar:



9. Comprobaremos el resultado con el archivo nuevo:



Estructuras

```
1.  /*  Estructura que hará la función de un árbol binario*/
2.  typedef struct arbol {
3.      unsigned long n;
4.      char c;
5.      struct arbol * izq;
6.      struct arbol * der;
7.  }
8.  arbol;
9.  /*  Estructura que simulará una lista de árboles binarios*/
10. typedef struct lista {
11.     struct arbol * a;
12.     struct lista * sig;
13. }
14. lista;
```

```
1.  //CABECERA
2.  #
3.  include "Estructuras.h" //DEFINICIÓN DE FUNCIONES
4.      /*Descripción: Función que unirá dos árboles y mantendrá como valor la suma de ambas frecuenciasR
    ecibe: arbol *a (apuntador a un árbol binario), arbol *b (apuntador a un árbol binario)Devuelve: arbo
    l* (apuntador al nuevo árbol donde convergen los dos árboles)Observaciones:*/
5.  arbol * nuevoArbol(arbol * a, arbol * b) {
6.          arbol * nvo;
7.          nvo = (arbol * ) malloc(sizeof(arbol));
8.          nvo - > n = a - > n + b - > n;
9.          nvo - > der = a;
10.         nvo - > izq = b;
11.         return nvo;
12.     }
13.     /*Descripción: Función que recorrerá el árbol de codificación de manera recursiva guardando las c
    odificaciones  en bits para cada caracter encontradoRecibe: arbol *a (apuntador al nodo donde nos enc
    ontremos en el árbol), char c[256][256] (arreglo de caracteres    que guardará las codificaciones de
    bits en el caracter respectivo, char *b (arreglo de caracteres que   contendrá la codificación en bit
    s temporalmente, int n (nivel del árbol donde nos encontremos)Devuelve: Observaciones:*/
14. void recorreArbol(arbol * a, char c[256][256], char * b, int n) {
15.         if (!(a - > izq)) { // Si el hijo izquierdo es un valor NULL
16.             int i;
17.             int k = binDec(a - > c); // Tomamos el caracter del nodo hoja
18.             for (i = 0; i < n; i++) {
19.                 c[k][i] = b[i]; // Copiaremos el recorrido binario para el caracter respectivo
20.             }
21.             c[k][i] = -1; // Colocamos -1 para indicar el fin de la codificación
22.             return;
23.         }
24.         b[n] = 0; // Colocamos 0 para el recorrido por la izquierda
25.         recorreArbol(a - > izq, c, b, n + 1); // Llamamos recursivamente a la función bajando un nive
    l por la izquierda
26.         b[n] = 1; // Sustituimos el valor por 1 para hacer el recorrido por la derecha desde este pun
    to
27.         recorreArbol(a - > der, c, b, n + 1); // Llamamos recursivamente a la función bajando un nive
    l por la derecha
28.         return;
29.     }
30.     /*Descripción: Función que insertará un árbol al final de la listaRecibe: lista **l (apuntador al
     inicio de la lista), arbol *a (árbol que desea añadirse)Devuelve: Observaciones:*/
31. void insertaLista(lista * * l, arbol * a) {
32.         lista * * aux = l;
```

```
33.         while ( * aux != NULL) {
34.             aux = & (( * aux) - > sig);
35.         } * aux = (lista * ) malloc(sizeof(lista));
36.         ( * aux) - > a = a;
37.         ( * aux) - > sig = NULL;
38.         return;
39.     }
40.     /*Descripción: Función que unirá los dos primeros árboles de la lista para posteriormente inserta
    rlo    en la lista acorde a su frecuenciaRecibe: lista **l (apuntador al inicio de la lista)Devuelve:
     Observaciones:*/
41. void dosMenoresArboles(lista * * l) {
42.         lista * aux, * nvo; // Dos apuntadores a elementos de la lista
43.         arbol * a;
44.         nvo = aux = * l; // Colocamos ambos en el inicio de la lista
45.         * l = ( * l) - > sig - > sig; // Avanzamos la lista dos posiciones
46.         nvo - > a = nuevoArbol(nvo - > a, nvo - > sig - > a); // Unimos los primeros elementos de la
    lista
47.         free(aux - > sig); // Liberamos el espacio requerido por malloc
48.         if (!( * l) || nvo - > a - > n <= ( * l) - > a - > n) { // Si la lista apunta a un NULL o la
    frecuencia a donde apunta es mayor a la actual
49.             nvo - > sig = * l; // El nuevo elemento de la lista apuntará ya sea al NULL o al elemento
     con mayor frecuencia
50.             * l = nvo; // La lista apuntará al nuevo elemento convergido
51.         } else { // En caso de ser una posición válida con frecuencia menor a la del nuevo elemento
52.             aux = * l; // El auxiliar guardará el primer elemento actual de la lista // Mientras la p
    osición sea válida y la frecuencia del siguiente elemento sea menor al elemento que desea añadirse
53.             while (aux - > sig && nvo - > a - > n > aux - > sig - > a - > n) {
54.                 aux = aux - > sig;
55.             } // En cuanto encontramos su posición, insertamos el elemento
56.             nvo - > sig = aux - > sig;
57.             aux - > sig = nvo;
58.         }
59.     }
60.     /*Descripción: Función que ordenará el arreglo de árboles en orden ascendente de acuerdo al númer
    o de apariciones del byteRecibe: arbol *A (arreglo de árboles a ordenar),int izq (límite inferior),in
    t der (límite superior)Devuelve: Observaciones: La ordenación se realiza por medio de Merge Sort*/
61. void ordena(arbol * A, int izq, int der) {
62.     if (izq == der) {
63.         return;
64.     }
65.     int k = 0;
66.     arbol z[1 + der - izq];
67.     int m = (izq + der) / 2;
68.     int m1 = m + 1;
69.     int m2 = izq;
70.     ordena(A, izq, m);
71.     ordena(A, m1, der);
72.     while (izq <= m && m1 <= der) {
73.         if (A[izq].n <= A[m1].n) {
74.             z[k] = A[izq];
75.             izq++;
76.         } else {
77.             z[k] = A[m1];
78.             m1++;
79.         }
80.         k++;
81.     }
82.     while (izq <= m) {
83.         z[k] = A[izq];
84.         k++;
85.         izq++;
```

```
    }
    while (m1 <= der) {
        z[k] = A[m1];
        k++;
        m1++;
    }
    for (k = 0; k <= (der - m2); k++) {
        A[k + m2] = z[k];
    }
}
```

Funciones Auxiliares

```
//CABECERA
#
include "FuncionesAuxiliares.h"
    /*Descripción: Función que convertirá un número entero a su equivalente binarioRecibe: int n (número a convertir)Devuelve: char (binario del número)Observaciones:*/
char decBin(int n) {
        char c = 0;
        int i;
        for (i = 128; i > 0; i /= 2) {
            if (n >= i) {
                c = c | i;
                n -= i;
            }
        }
        return c;
    }
    /*Descripción: Función que convertirá un número binario a su equivalente decimalRecibe: char c (variable que contiene el binario del número)Devuelve: unsigned int (binario en su representación de entero)Observaciones:*/
unsigned int binDec(char c) {
        int i;
        unsigned n = 0;
        for (i = 0; i < 8; i++) {
            if ((c >> i) & 1) {
                n += pow(2, i);
            }
        }
        return n;
    }
    /*Descripción: Función que imprimirá un valor binario en el documentoRecibe: char n (representación binaria a imprimir)Devuelve: Observaciones:*/
void imprimeBinario(char n) {
    int i;
    for (i = 7; i >= 0; i--) {
        printf("%d", (n >> i) & 1);
    }
}
```