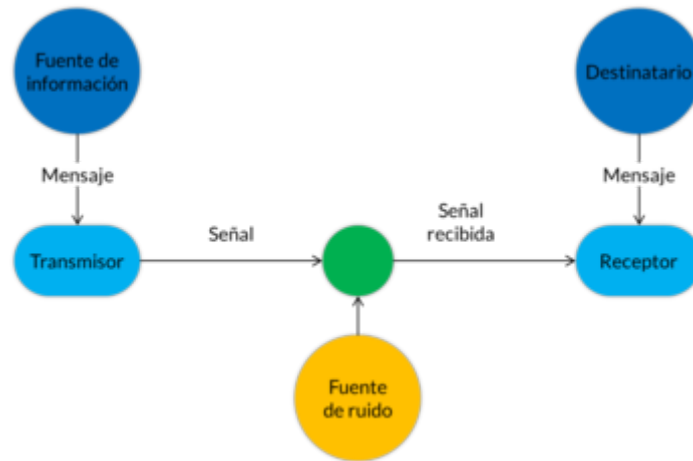


# TP Integrador N°1



**Alumnos:** Vega Imbalde Aureliano, Savarino Jeremías, Facundo Errobidart.

**Primera entrega:** 20/10/2022

**Segunda entrega:** 27/10/2022

**Teoría de la información**

**2° Cuatrimestre 2022**

<b>Resumen</b>	<b>1</b>
<b>Primer parte</b>	<b>1</b>
Introducción	1
Desarrollo	1
<b>Segunda parte</b>	<b>4</b>
Introducción	4
Desarrollo	4
<b>Conclusiones</b>	<b>10</b>
<a href="#">Enlace al repositorio</a>	

## Resumen

El trabajo consiste en dos partes, la primera que trata sobre las fuentes de información y sus tipos (nulas y no nulas) y sus respectivas propiedades. En base a un archivo dado por la cátedra se determinará qué tipo de fuente fue la que emitió la información en él y se determinarán ciertas propiedades en base a su tipo. La segunda parte, se adentra en la codificación de la información y en las propiedades de los códigos. Además se ven formas de re-codificar archivos con el fin de comprimirlos, en nuestro caso, optamos por el algoritmo de Huffman.

## Primera parte

### Introducción:

Se tiene un archivo provisto por la cátedra que contiene 10.000 caracteres emitidos por una fuente de información desconocida. En base al análisis de los símbolos emitidos por esta fuente logramos determinar el tipo de la misma. El alfabeto de la fuente es el conjunto  $C = \{ 'A', 'B', 'C' \}$

### Desarrollo:

Realizamos un programa en C para calcular las probabilidades de que un símbolo se dé si aparece otro previamente y los resultados para el archivo dado fueron:

```
Cantidad de veces que aparece cada caracter:
A: 1576
B: 6148
C: 2276
```

La cantidad de veces que ocurre cada transición es:

Ocurrencias:	A	B	C
A	140	1266	169
B	647	4821	680
C	787	61	1428

Esto se leería, por ejemplo, “La cantidad de veces que aparece B habiendo aparecido previamente A, es 647”.

A modo demostrativo, planteamos un **pseudocódigo** de la implementación:

```
leer(archivo, previo); //lee del archivo el primer caracter.
leer(archivo, actual); //lee del archivo el segundo caracter.
contador[actual] = 1; //pone un 1 en la posición del caracter actual en el vector.
MIENTRAS (NO eof(archivo))
    ocurrencias[actual][previo]++; // acumula en la matriz la cantidad de veces que aparece el caracter “actual”
    habiendo aparecido “previo” anteriormente.
    previo = actual;
    leer(archivo, actual);
    contador[actual]++; // acumula en la matriz la cantidad de veces que aparece el caracter “actual”.
FIN MIENTRAS
```

Luego, para obtener las probabilidades, se divide cada fila de la matriz de ocurrencias por la cantidad de veces que aparece el símbolo que hace referencia a cada columna.

### Pseudocódigo:

Aclaración: en todos los pseudocódigos de este informe, el ciclo ‘desde hasta’ es **inclusive**.

```
DESDE i = 0 HASTA i = N-1 CON paso = 1 HACER
    DESDE j = 0 HASTA j = N-1 CON paso = 1 HACER
        probabilidades[i][j] = ocurrencias[i][j] / contador[j];
    FIN DESDE
FIN DESDE
```

Finalmente se obtiene:

Matriz de pasaje:			
	A	B	C
A	0.09	0.21	0.07
B	0.41	0.78	0.30
C	0.50	0.01	0.63

Esto indica que si la fuente de información emite un símbolo “A”, las probabilidades de que el símbolo siguiente sea “A”, “B” o “C” son 0.09, 0.41 y 0.50 respectivamente.

En caso de que la fuente de información fuera de memoria nula, deberíamos ver que los valores de la matriz generada tendrían que tender a  $\frac{1}{3}$ , ya que en este tipo de fuentes, los símbolos que emiten son estadísticamente independientes. Sin embargo, nos encontramos con casos donde si la fuente emite los símbolos A o B, la posibilidad de obtener un símbolo B a continuación es de 0.41 y 0.78 respectivamente. Por lo tanto, es correcto decir que el símbolo obtenido en el instante  $i$  va a condicionar al que se emitirá en el instante  $i+1$ , es decir, es una fuente de información con memoria **no nula**. Particularmente, una fuente markoviana.

Para determinar si la fuente es ergódica se verificó que en el grafo de la fuente no haya ningún sumidero, es decir, que no haya un vértice que no vaya a ningún otro vértice más allá de él mismo.

### Pseudocódigo:

$i = 0;$

HACER

aux = 0;

j = 0;

MIENTRAS (j < N && aux == 0) HACER

SI (i != j y probabilidades[j][i] > 0) ENTONCES

aux = 1;

j++;

FIN MIENTRAS

i++;

MIENTRAS (i < N y aux == 1);

RETURN aux;

En este caso, la fuente es **ergódica**.

Por lo tanto, se puede afirmar que todos los estados son alcanzables desde otro estado.

La distribución de probabilidades en cada t va variando con la evolución del proceso de emisión de símbolos, una vez que se estabiliza, se obtiene el vector estacionario.

Para calcular el vector estacionario se debe resolver el sistema de ecuaciones generado mediante la matriz de transición menos la identidad, igualando cada fila a 0 y además agregando una fila de todos 1s.

**Pseudocódigo para generar el sistema de ecuaciones:**

DESDE i = 0 HASTA i = N CON paso = 1 HACER

DESDE j = 0 HASTA j = N CON paso = 1 HACER

SI (i == 0) ENTONCES

MGauss[i][j] = 1; //la primera fila son todos 1s

SINO SI (j == N) ENTONCES

MGauss[i][j] = 0; //la última columna son todos 0s (excepto en la primera fila)

SINO

MGauss[i][j] = probabilidades[i-1][j];

SI (i == j+1) ENTONCES

MGauss[i][j]--; // se le resta la identidad a la matriz de probabilidades

FIN SI

FIN SI

FIN DESDE

FIN DESDE

Para resolver este sistema de ecuaciones aplicamos Gauss

**Pseudocódigo:**

DESDE i = 0 HASTA i = N CON paso = 1 HACER

DESDE j = 0 HASTA j = N CON paso = 1 HACER

aux1 = MGauss[j][i] / MGauss[i][i];

DESDE k = 0 HASTA k = N CON paso = 1 HACER

aux2 = MGauss[i][k] \* aux1;

MGauss[j][k] -= aux2;

```

FIN DESDE
FIN DESDE
FIN DESDE

```

```

DESDE i = N-1 HASTA i = 0 CON paso = -1 HACER
  DESDE j = N-1 HASTA j = i+1 CON paso = -1 HACER
    VecEstacionario[i] = VecEstacionario[i] - MGauss[i][j]*V[j];
  FIN DESDE
  VecEstacionario[i] = (VecEstacionario[i]+MGauss[i][N])/MGauss[i][i];
FIN DESDE

```

El **vector estacionario** nos dió los siguientes valores:

```

0.16
0.61
0.23

```

El cálculo de la entropía se hace de la siguiente manera:

$$H_1 = \sum_i p_i * \sum_j p_{j/i} \log \frac{1}{p_{j/i}}$$

siendo  $P_{i/j}$  las probabilidades condicionales de la matriz de transición y  $P_i$  las probabilidades estacionarias (valores del vector estacionario).

**Pseudocódigo:**

```

DESDE i = 0 HASTA i = N-1 CON paso = 1 HACER
  aux = 0;
  DESDE j = 0 HASTA j = N-1 CON paso = 1 HACER
    SI (probabilidades[j][i] != 0) ENTONCES
      aux = aux - probabilidades[j][i] * log2(probabilidades[j][i]);
    FIN SI
  FIN DESDE
  H = H + VecEstacionario[i] * aux;
FIN DESDE
RETURN H;

```

La entropía (cantidad media de información por símbolo) dió **0.987 bits/símbolo**.

## Segunda parte

**Introducción:**

Para la segunda parte se reutiliza el mismo archivo, pero se toman secuencias de caracteres de este de longitud 3, 5 y 7 para representar tres códigos distintos. En base a ellos se calcularán distintas propiedades de ellos como cantidad de información, entropía, rendimiento, redundancia y que tan compacto es. Finalmente se comprime los archivos utilizando el algoritmo de Huffman.

## Desarrollo:

La idea principal a la hora de desarrollar los distintos algoritmos de esta segunda parte fue poder generalizar cada algoritmo para que, dependiendo de la variable “tamPalabra”, el programa pueda ejecutarse tanto para 3, 5 o 7 palabras.

Primero, para leer cada palabra y contar sus ocurrencias, se construyeron los siguientes algoritmos:

### Lectura de las palabras del archivo:

```
contPalabras=0;
tamLista=0;
inicializarLista(lista);
abrir ( Archivo);
MIENTRAS no termine el archivo, HACER:
    leer(pal);
    SI (pal.longitud==tamPalabra) ENTONCES:
        buscaYCuenta(lista,pal,tamLista);
        contPalabras=contPalabras+1;
    FIN SI
FIN MIENTRAS
```

### Aclaraciones :

- **lista** es un array del registro “nodoProb”, que está compuesto por : palabra (**pal**), ocurrencia de la palabra (**ocurrencia**), y probabilidad de aparición de la palabra (**prob**)
- **tamLista** es el tamaño de la lista. Su valor final será obtenido en la última iteración. Se irá modificando en la función buscaYCuenta.
- el tamaño de **pal** será 3 ,5 o 7, según haya elegido el usuario.

### Pseudocódigo para contar las ocurrencias(procedure buscaYCuenta):

```
encontrado=false;
i=0;
MIENTRAS (encontrado==false y i < tamLista) HACER:
    SI (pal==lista[i].pal), ENTONCES:
        encontrado=true;
    FIN SI
    i=i+1;
FIN MIENTRAS

SI (encontrado==false) ENTONCES:
    lista[i].pal=pal;
    lista[i].ocurrencia=1;
    tamLista=tamLista+1;
SINO:
    lista[i].ocurrencia=lista[i].ocurrencia+1;
FIN SI
```

- **Aclaraciones y comentarios :**

La idea de este algoritmo es ir recorriendo la lista de palabras, si encuentra la palabra, aumenta su ocurrencia en 1. Si la palabra no está en la lista, la agrega, aumenta su ocurrencia en 1 y aumenta el tamaño de la lista.

- **lista[i].pal** se refiere a la palabra código que se encuentra en la posición i de la lista.
- **lista[i].ocurrencia** se refiere a las ocurrencias de la palabra que se encuentra en la posición i del array.
- **encontrado** es una variable booleana que cambia a True cuando el algoritmo encuentra la palabra recién leída del archivo ( recibida por parámetro) en la lista.
- **tamLista** es el tamaño del array, el cual irá cambiando en base a la cantidad de palabras halladas. Indica la cantidad de palabras distintas encontradas.

Con el siguiente algoritmo calculamos tanto la cantidad de información como la entropía:

### Pseudocodigo:

```

cantidadDeInformacion = 0;
entropia = 0;
DESDE i = 0 HASTA i = tamLista, CON paso = 1 HACER:
    lista[i].prob = lista[i].ocurrencia / cantPalabras;
    cantidadDeInformacion = cantidadDeInformacion + log2(1/lista[i].prob);
    entropia = entropia + lista[i].prob * log3(1 / lista[i].prob);
FIN DESDE

```

### Aclaraciones y comentarios:

La idea de este algoritmo es recorrer toda la lista de palabras, y calcular la entropía y la cantidad de información en base a las distintas palabras y sus frecuencias.

Para los cálculos nos basamos en las siguientes ecuaciones para la entropía, longitud media y cantidad de información respectivamente:

$$L = \sum_{i=1}^q p_i l_i$$

$$H = \sum \log_r(1/P_i) * P_i$$

$$I = \sum \log_2(1/P_i)$$

- **lista[i].prob**, se refiere a la probabilidad de aparición de la palabra que se encuentra en la posición i del array.
- **cantPalabras** son la cantidad de palabras diferentes( del respectivo tamaño) que fueron encontradas
- **tamLista** es el tamaño del array descrito anteriormente.

Entonces los resultados obtenidos son los siguientes :

```
-----  
Leyendo archivo con palabras de 3 caracteres  
-----
```

```
Cantidad de informacion: 178.89  
Longitud media: 2.99  
Entropia: 2.08
```

```
-----  
Leyendo archivo con palabras de 5 caracteres  
-----
```

```
Cantidad de informacion: 1151.58  
Longitud media: 4.99  
Entropia: 3.28
```

```
-----  
Leyendo archivo con palabras de 7 caracteres  
-----
```

```
Cantidad de informacion: 3232.89  
Longitud media: 6.99  
Entropia: 4.34
```

Con este sencillo algoritmo, se calcula la **inecuación de Kraft/McMillan**, siguiendo la definición dada por la cátedra que es:

$$\sum_{i=1}^q r^{-l_i} \leq 1 \quad \text{INECUACIÓN de KRAFT}$$

**Pseudocódigo:**

```
kraft=0;  
DESDE i=0 HASTA i = tamLista , CON paso = 1 HACER:  
    kraft=kraft + 3**(-longitudMedia)
```

También con el siguiente algoritmo calculamos las longitudes medias de los códigos, siguiendo la siguiente definición de la cátedra:

$$L = \sum_{i=1}^q p_i l_i$$

**Pseudocódigo:**

```
longitudMedia=0;  
DESDE i=0 HASTA i = tamLista , CON paso = 1 HACER:  
    longitudMedia=longitudMedia+ lista[i].prob* len(lista[i].pal);  
FIN DESDE
```

**Aclaración:**

- **len(lista[i].pal)** es la longitud de la palabra código que se encuentra en la posición i del array
- **lista[i].prob**, se refiere a la probabilidad de aparición de la palabra que se encuentra en la posición i del array.



Los resultados obtenidos son los siguientes:

```
Resultado Kraft/McMillan: 1.00
El codigo de longitud media 3.000000 cumple la inecuacion de Kraft/MacMillan (es compacto)

Resultado Kraft/McMillan: 0.54
El codigo de longitud media 5.000000 cumple la inecuacion de Kraft/MacMillan (es compacto)

Resultado Kraft/McMillan: 0.16
El codigo de longitud media 7.000000 cumple la inecuacion de Kraft/MacMillan (es compacto)
```

Como se puede observar en los resultados, para las tres longitudes se cumple que el lado izquierdo de la ecuación es menor o igual a uno por lo cual es condición necesaria y suficiente para decir que son códigos instantáneos para los diferentes casos, dado que todas las palabras son de la misma longitud, no es posible que una palabra sea prefijo de otra, dado que si ocurriese, es la misma palabra.

Además, por McMillan sabemos que si la inecuación se cumple también se cumple la recíproca del postulado de Kraft, es decir que los códigos son unívocos.

Luego de calcular la **entropía y la longitud media**, obtuvimos el **rendimiento y la redundancia** a partir de las siguientes ecuaciones:

- $\text{rendimiento} = \text{entropía} / \text{longitudMedia}$ ;
- $\text{redundancia} = 1 - \text{rendimiento}$ ;

Los resultados obtenidos fueron:

Para palabras código de **tres** caracteres:

```
-----
Redundancia: 0.30
Rendimiento: 0.70
-----
```

Para palabras código de **cinco** caracteres:

```
-----
Redundancia: 0.34
Rendimiento: 0.66
-----
```

Para palabras código de **siete** caracteres:

```
-----
Redundancia: 0.38
Rendimiento: 0.62
-----
```

Se realizó la codificación de los archivos mediante algoritmo de Huffman, por lo cual se armó un árbol binario en el cual en los niveles más altos se encuentran las palabras más frecuentes, dejando a las menos frecuentes en los niveles inferiores, lo que hace que requieran más bits para su representación.

Para ello, se definió un struct `nodoHuffman`, el cual contiene la palabra (puede ser vacía si se tratase de un nodo interno), la sumatoria de ocurrencias y punteros a ambos hijos del árbol binario.

Entonces la estructura quedaría tal que:

```
struct nodoHuffman {
```

```

    string pal
    int ocurrencias
    punteros *izq, *der
}

```

La idea básica de nuestro algoritmo es construir inicialmente un arreglo de nodos, donde todos sean raíces. Una vez construido este arreglo, iteramos sobre él buscando dos elementos con menor cantidad de ocurrencias y los agregamos al árbol, de la siguiente manera. El nodo creado a partir de la suma de dos ya existentes también se inserta en el arreglo y se vuelve a considerar para ser sumado, de tal forma que iteramos hasta que en el arreglo existe un solo elemento, el árbol de Huffman completo:

```

nodoHuffman izq, der, tope;

```

```

MIENTRAS (NO hayUnSoloElemento(arregloSubArboles)) HACER

```

```

    izq = obtenerMinimo(arregloSubArboles)

```

```

    der = obtenerMinimo(arregloSubArboles)

```

```

    tope = creaNodo("", izq.ocurrencias + der.ocurrencias)

```

```

    tope.izq = izq

```

```

    tope.der = der

```

```

    insertarNuevoNodo(arregloSubArboles, tope)

```

```

FIN MIENTRAS

```

Un par de consideraciones a tener en cuenta es que la función **obtenerMinimo** una vez que obtiene el elemento del arreglo de subárboles, lo remueve del arreglo, por lo cual es seguro llamar de nuevo a la misma función para obtener nuevamente el segundo mínimo.

La función **creaNodo** simplemente realiza la alocaación de memoria e inicializa los valores del tope del nuevo subárbol. Finalmente, **insertarNuevoNodo** mete en el arreglo de subárboles el nuevo nodo creado.

Al final del proceso, queda construido un **árbol binario de Huffman**, el cual recorreremos para construir un diccionario: un array de elementos que contiene la palabra original y la traducción binaria, que luego utilizaremos para generar los archivos codificados. Para ello tenemos una función que va construyendo los códigos a medida que se recorre el árbol.

```

FUNCION escribirCodigosHuffman(nodoHuffman raiz, int arr[], int tope, nodoDiccionario diccionario[], int tamDiccionario)

```

```

SI (raiz.izq)

```

```

    arr[tope] = 0

```

```

    escribirCodigosHuffman(raiz.izq, arr, tope+1, diccionario, tamDiccionario)

```

```

FIN SI

```

```

SI (esHoja(raiz))

```

```

    insertarEnDiccionario(raiz, diccionario[tamDiccionario], arr, tope)

```

```

    tamDiccionario = tamDiccionario + 1

```

```

FIN SI

```

```

SI (raiz.der)

```

```

    arr[tope] = 1

```

*escribirCodigosHuffman(raiz.der, arr, tope+1, diccionario, tamDiccionario)*

FIN SI

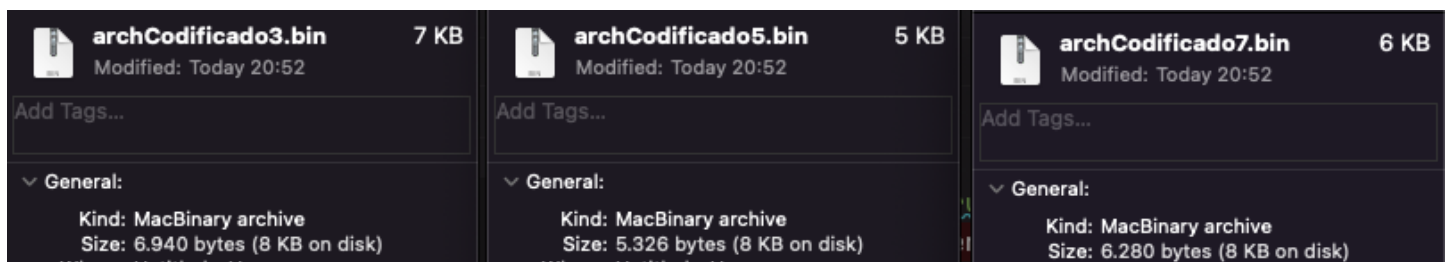
FIN FUNCION

La variable **arr** es un arreglo que contiene el código que se va generando a medida que se recorre el árbol, la variable **tope** se utiliza debido a que no se debe usar todo el arreglo, sino una porción de este que es válida, dado que más allá de este habrá valores de instancias anteriores de la recursividad.

Finalmente, una vez generado el diccionario, lo ordenamos por frecuencia de palabra de la más a la menos frecuente para acelerar el proceso de compresión y luego leemos el archivo original nuevamente y lo comprimimos usando el diccionario.

El archivo codificado contiene el diccionario generado como header. Los primeros 4 bytes contienen el tamaño del diccionario (de aquí en adelante  $n$ ) y luego los siguientes  $n * 10$  bytes contienen el diccionario propiamente dicho, con los primeros 8 bytes conteniendo la palabra original del archivo y los últimos 2 la codificación. Luego se encuentra el archivo codificado propiamente dicho. El programa contiene una opción para verificar que el la compresión se haya realizado correctamente y reconstruye en texto original en la consola.

Los tres archivos quedan de un tamaño menor al original, como se puede apreciar en la siguiente imagen:



## Conclusiones

El archivo presentado por la cátedra posee una fuente de información no nula. Además, dado que todos los estados son alcanzables desde otro estado, se puede afirmar que es ergódica. Al ser ergódica, tiene un vector estacionario, el cual es  $[0.16, 0.61, 0.23]$ . Y con este vector se puede calcular su entropía, dando 0.987 bits/símbolo.

Por otro lado, entropía para las palabras código de 3 caracteres es 2.08, lo que significa que para obtener un valor cercano al 100 % de rendimiento, la longitud media del código debería ser cercana a 2.08.

Como se puede observar en los resultados, el rendimiento para palabras de 3 caracteres es 0.7, mientras que para palabras de 7 caracteres es 0.62.

El hecho de que a medida que tomamos más caracteres para representar la palabra código, el rendimiento sea más bajo, significa que cuanto más caracteres tenga la palabra código, la diferencia entre la entropía y la longitud media de las palabras será mayor.

Por ende para obtener una mayor cantidad de información, sería conveniente tomar palabras código de menor longitud media.

En la compresión de archivos, una particularidad que notamos a la hora de generar los códigos fue que inicialmente pensamos que el archivo tomado de a palabras de 3 caracteres era el que iba a ver la mayor compresión al ser el menos entrópico. Finalmente resultó ser el menos comprimido, y esto se debe a que los archivos binarios codificados tienen un tamaño de palabra de 2 bytes (16 bits) para poder escribir sin truncar las codificaciones de palabras de los archivos de palabras de 5 y 7 caracteres. Por ello, incluso si la codificación de Huffman para cada palabra utiliza menos bits, cada vez que ésta se escribe en el archivo ocupa una longitud fija de 2 bytes. Por lo tanto, cuantas más palabras tenga el archivo, más grande será, sin

importar que la codificación para las palabras sea de menos bits. Una forma de contrarrestar esto sería escribiendo un tamaño de palabra dinámico basado en el código más largo, pero sin duda haría recuperar esta información un proceso más engorroso.