



Universidad de Granada

[decsai.ugr.es](http://decsai.ugr.es)

# **Teoría de la Información y la Codificación**

**Grado en Ingeniería Informática**

**Seminario 2.- Plataforma láser para envío y recepción de información con códigos Huffman.**



**DECSAI**

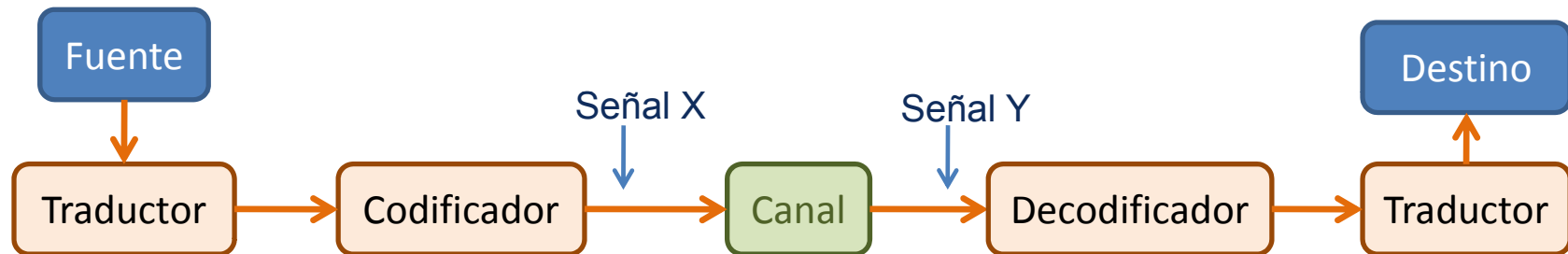
**Departamento de Ciencias de la  
Computación e Inteligencia Artificial**

- 1. Codificación en canales sin ruido**
- 2. Códigos instantáneos**
- 3. Códigos óptimos**

# SESIÓN 1

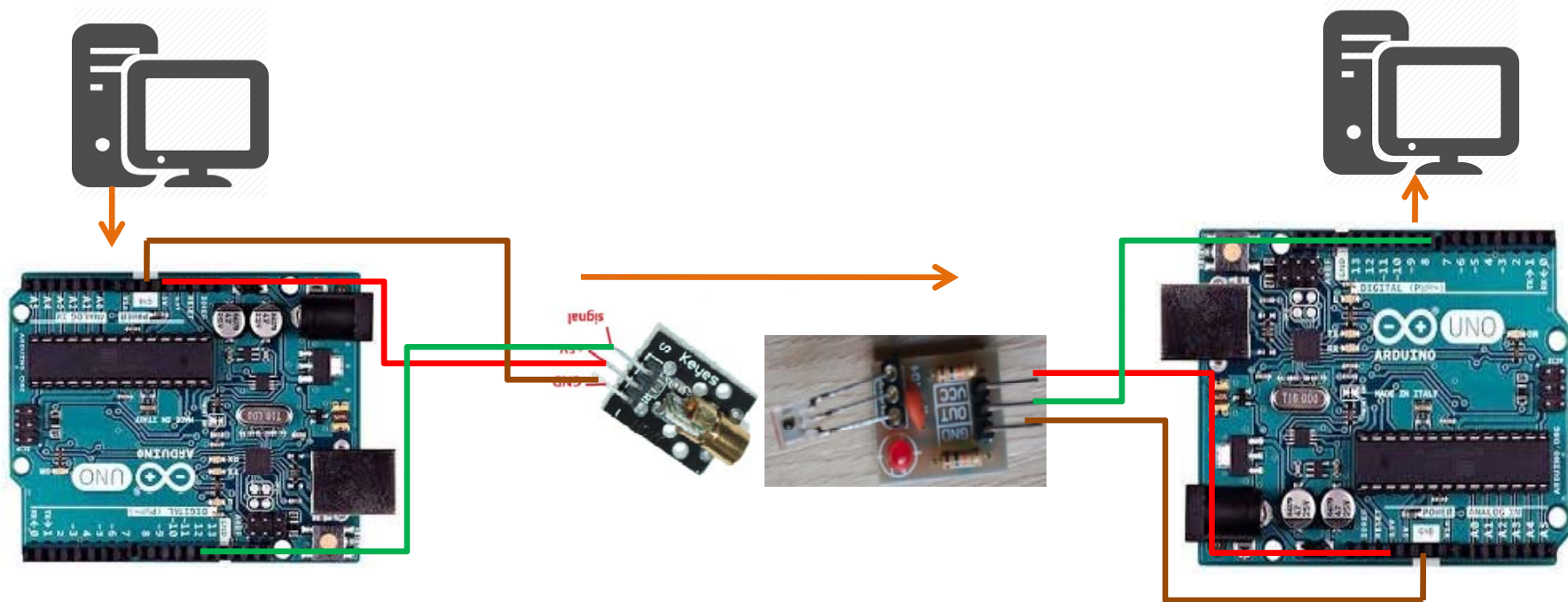
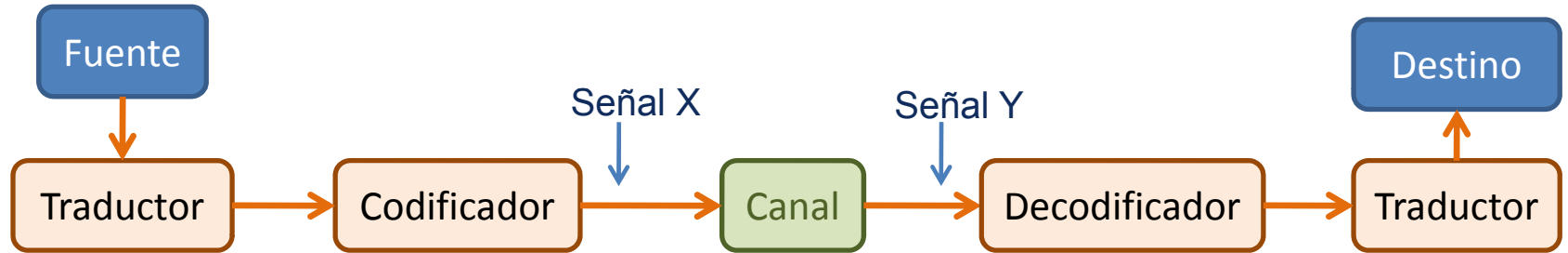
1. **Codificación en canales sin ruido**
2. **Códigos instantáneos**
3. **Códigos óptimos**

### – Base para la codificación en un canal sin ruido:



- En nuestro sistema de transmisión por láser, la **fuentes** será un PC conectado por USB a un Arduino equipado con el emisor láser (emisor).
- El **emisor** será un Arduino que recibe por USB una cadena en texto ASCII, la **traduce** a un código óptimo y la **codifica** en *laserbits*.
- El **receptor** será otro Arduino, equipado con un fotorreceptor que recibe *laserbits*, los **decodifica** al código óptimo y lo **traduce** a ASCII.
- El **destino** será otro PC, que recibe por USB los códigos ASCII decodificador por el Arduino receptor.

### – Base para la codificación en un canal sin ruido:



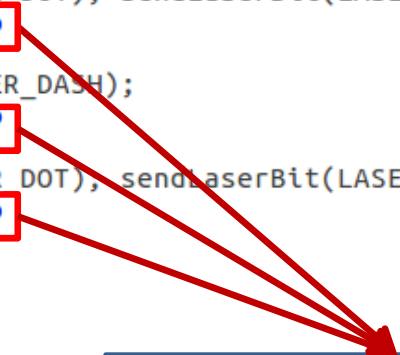
- En la práctica, implementaremos:
  - Códigos instantáneos.
  - Óptimos (mínimo número de bits/símbolo en promedio).
- Al ser códigos instantáneos, **¡ya no necesitamos una señalización de final de símbolo!**
- En la práctica 1 no ocurría esto y, cada vez que se enviaba un símbolo, era necesario indicar cuándo finalizaba.
- En la práctica 1 utilizábamos 3 tipos de señales:
  - **LASER\_DOT**, identificada como un “0”
  - **LASER\_DASH**, identificada como un “1”
  - **LASER\_NONE**, para señalar cuándo termina un símbolo.
- Utilizábamos **LASER\_NONE como señal de finalización de símbolo.**

Programa **emisorArdu** para la plataforma Arduino Uno (*flashback* del Seminario 1):

```

3 #include <ticcommardu.h>
4 #include <util/delay.h>
5
6
7 int main(void) {
8
9   initLaserEmitter(); // Iniciamos el láser
10
11   while (1) {
12
13     // Enviamos: ... -- ...
14     sendLaserBit(LASER_DOT); sendLaserBit(LASER_DOT); sendLaserBit(LASER_DOT);
15     sendLaserBit(LASER_NONE); // Fin de símbolo
16
17     sendLaserBit(LASER_DASH); sendLaserBit(LASER_DASH);
18     sendLaserBit(LASER_NONE); // Fin de símbolo
19
20     sendLaserBit(LASER_DOT); sendLaserBit(LASER_DOT); sendLaserBit(LASER_DOT);
21     sendLaserBit(LASER_NONE); // Fin de símbolo
22
23   }
24
25   return 0;
26 }
27

```



Esto ya no va a ser necesario



1. **Codificación en canales sin ruido**
2. **Códigos instantáneos**
3. **Códigos óptimos**

- En un **código instantáneo**, no existen dos símbolos en todo el código donde el código de uno de ellos sea el comienzo de otro
- Ejemplo: Alfabeto de 2 símbolos ‘A’ y ‘B’
  - Caracter ASCII ‘A’ codificado como “0010”.
  - Carácter ASCII ‘B’ codificado como “001”.

### – NO ES CÓDIGO INSTANTÁNEO

- Ejemplo: Alfabeto de 2 símbolos ‘A’ y ‘B’
  - Caracter ASCII ‘A’ codificado como “0010”.
  - Carácter ASCII ‘B’ codificado como “000”.

### – SÍ ES CÓDIGO INSTANTÁNEO

- Desde el punto de vista de las comunicaciones, los códigos instantáneos tienen una gran implicación práctica: No es necesario señalar la finalización del código. ¡Se ahorran bits y recursos!

- En nuestro caso, podemos ahorrar recursos de una forma drástica. Pensemos que, hasta ahora teníamos la siguiente codificación:
  - Enviar un “0”: Se enviaba un LASER\_DOT (2 laserbits, 1 en alta y 1 en baja).
  - Enviar un “1”: Se enviaba un “LASER\_DASH (3 laserbits, 2 en alta y 1 en baja).
- En nuestro código del seminario 1:
  - Si el láser no emitía se interpretaba como señal de finalización. Desaprovechábamos el estado “no enviando” sólo para señalar.
  - Si el láser emitía, se interpretaba como “0” o “1” dependiendo del número de ciclos que el láser estaba encendido.
- En esta práctica, seremos más eficientes y utilizaremos los estados del láser “emitiendo” y “apagado” para transmitir información.
  - **Si el láser está encendido un ciclo: Se interpreta como el bit 1.**
  - **Si el láser está apagado un ciclo: Se interpreta como el bit 0.**

- En la biblioteca `<timccommandu.h>`, crearemos dos constantes nuevas definidas como **LASER\_HIGH** y **LASER\_LOW**, para codificar los bits “1” y “0”, respectivamente:

```

2 #ifndef __TICCOMMARDU__
3
4 #define __TICCOMMARDU__
5
6 // Ciclo de reloj del procesador: 16MHz
7 #define F_CPU 16000000UL
8
9
10 // Tiempo de ciclo para comunicaciones por láser, en milisegundos
11 #define UMBRAL_U 6
12
13 // Tiempo de muestreo
14 #define SAMPLE_PERIOD (UMBRAL_U/3)
15
16
17 // Constante que representa una ráfaga corta (puntos)
18 #define LASER_DOT 0
19
20 // Constante que representa una ráfaga larga (raya)
21 #define LASER_DASH 1
22
23
24 // Constante que representa ninguna ráfaga
25 #define LASER_NONE 2
26
27
28 // Constante que representa el que laser está encendido
29 #define LASER_HIGH 1
30
31 // Constante que representa el que laser está apagado
32 #define LASER_LOW 0

```

- Modificaremos nuestra biblioteca **ticcommardu.h** y **ticcommardu.cpp** para incluir esta variación en el envío y la recepción de bits. Insertaremos una variante de las funciones **sendLaserBit** y **recvLaserBit**, que llamaremos **sendLaserBit2** y **recvLaserBit2**.

```

57 /**
58  * Función para enviar por láser un símbolo.
59  *
60  * Entradas:
61  *   what: Qué se envía. Puede ser uno de los valores siguientes:
62  *     LASER_HIGH para enviar un 1
63  *     LASER_LOW para enviar un 0
64  *
65  */
66 void sendLaserBit2(const unsigned char what);

94 /**
95  * Función para recibir un símbolo desde el fotorreceptor de láser.
96  *
97  * Entradas: Ninguna
98  *
99  * salidas
100  *   what: Qué se ha recibido. Puede ser uno de los valores siguientes:
101  *     LASER_HIGH para recibir un 1,
102  *     LASER_DASH para recibir un 0
103  */
104 void recvLaserBit2(unsigned char & what);

```

- La implementación en **ticcommardu.cpp (I)**:
- Sólo gastaremos 1 ciclo para enviar un 0 o un 1:

```

39 void sendLaserBit2(const unsigned char what) {
40
41     // Ponemos salida a alta si hay que enviar 1
42     if (what == LASER_HIGH)
43         PORTB |= _BV(DDB4);
44     else // 0 ponemos a baja si se envía 0
45         PORTB &= ~_BV(DDB4);
46
47     // Esperamos el tiempo de ciclo
48     _delay_ms(UMBRAL_U);
49 }

```

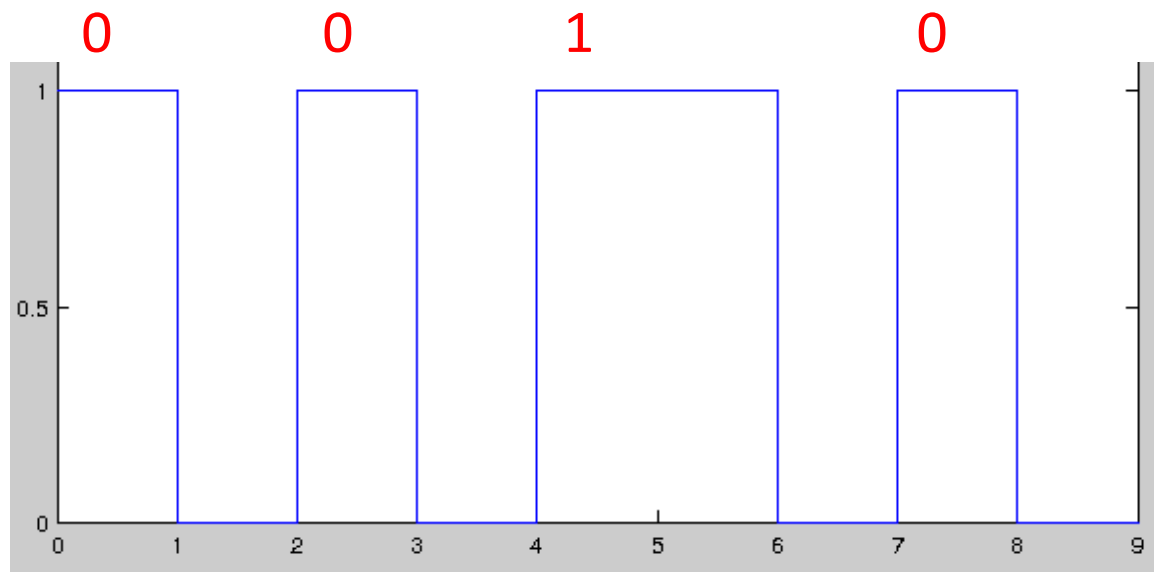
- La implementación en **ticcommardu.cpp (II)**:
  - Sólo gastamos un ciclo en leer:

```

114 void recvLaserBit2(unsigned char & what) {
115
116     unsigned char cBajo=0; // Contador de veces que se detecta voltaje bajo
117     unsigned char cAlto=0; // Contador de veces que se detecta voltaje alto
118     unsigned char tiempo=0; // Tiempo midiendo
119
120     // Pasamos un ciclo midiendo
121     while (tiempo < UMBRAL_U) {
122
123         // Asumimos que está en el PIN 8
124         dato= PINB & 0x01;
125
126         if (dato>0) // Miramos si está en alta o en baja
127             cAlto++;
128         else
129             cBajo++;
130
131         // Esperamos el periodo de muestreo
132         _delay_ms( SAMPLE_PERIOD );
133
134         // Y aumentamos el tiempo que hemos estado en la función
135         tiempo+= SAMPLE_PERIOD;
136     }
137
138     // Si hemos muestreado más altas que bajas, devolvemos LASER_HIGH
139     if (cAlto>cBajo)
140         what= LASER_HIGH;
141
142     else // En otro caso, es un LASER_LOW
143         what= LASER_LOW;
144 }

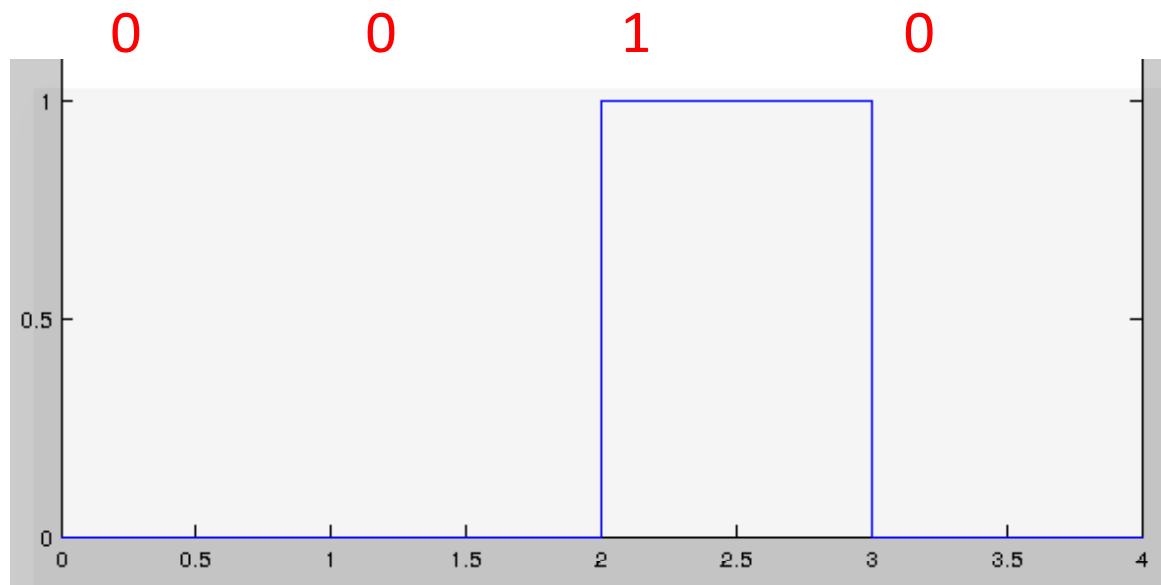
```

- El diagrama de comunicaciones para enviar los bits “0010”:
  - Antes, utilizando señalización de fin de símbolo:





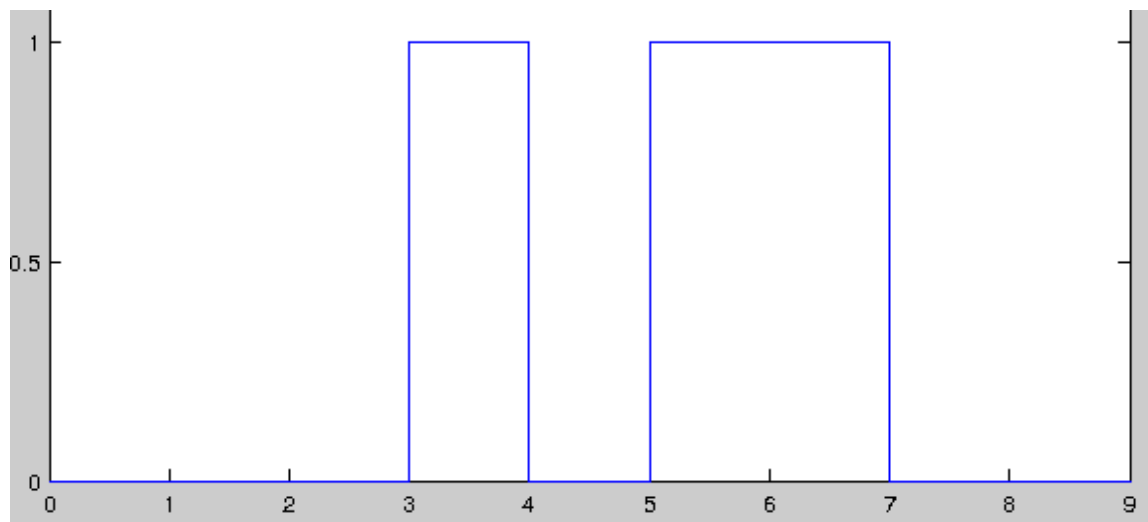
- El diagrama de comunicaciones para enviar los bits “0010”:
  - Ahora, sin usar señalización de fin de símbolo:



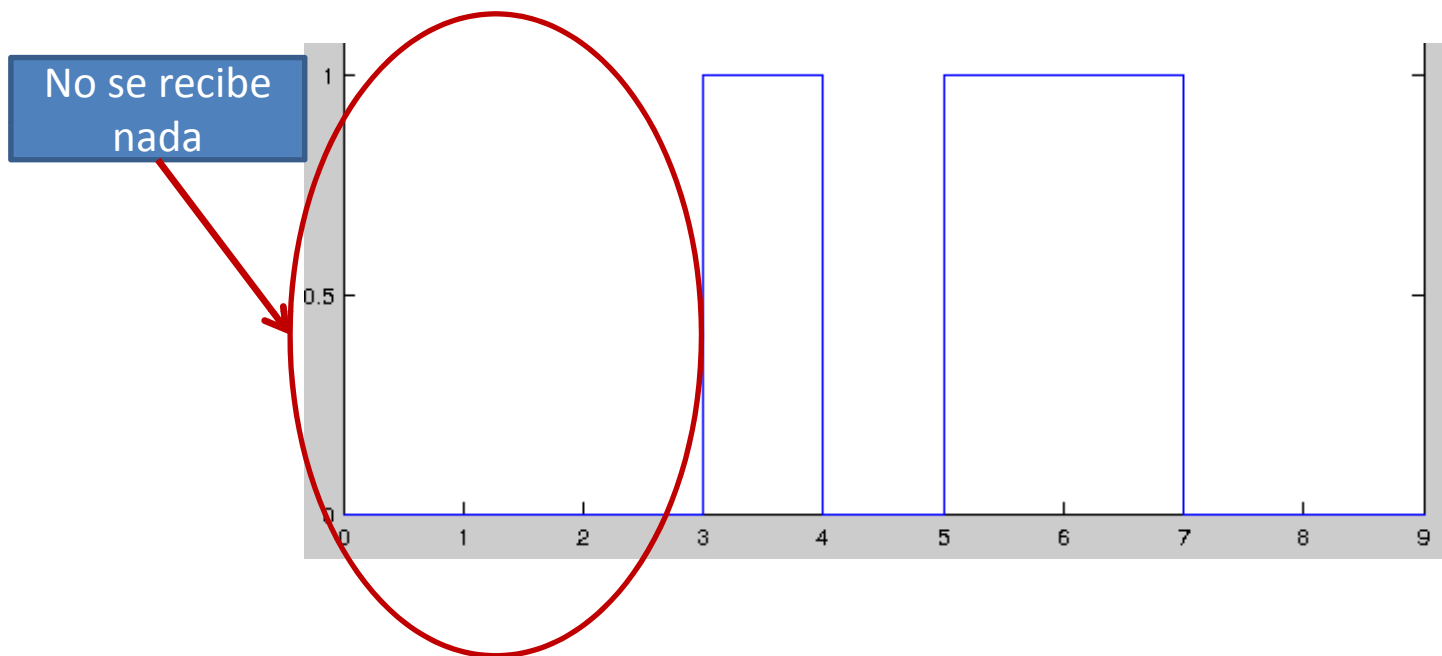
¡Se ahorran muchos ciclos!

- **Limitación:** El destino no puede saber cuándo el emisor está enviando el bit “0” o cuando ni siquiera está emitiendo.
- **Para solventar esta limitación**, podemos señalar el comienzo de un mensaje enviando el bit “1” al principio.
- Sin embargo, el receptor sigue sin saber cuándo se termina de recibir el mensaje.
- **Para solventar esta limitación**, incluiremos un nuevo código que utilizaremos al final del mensaje, para indicar su finalización.
- **Por tanto, el alfabeto del emisor estará formado por el alfabeto de la fuente + un símbolo adicional a utilizar como señalización de fin de mensaje.**

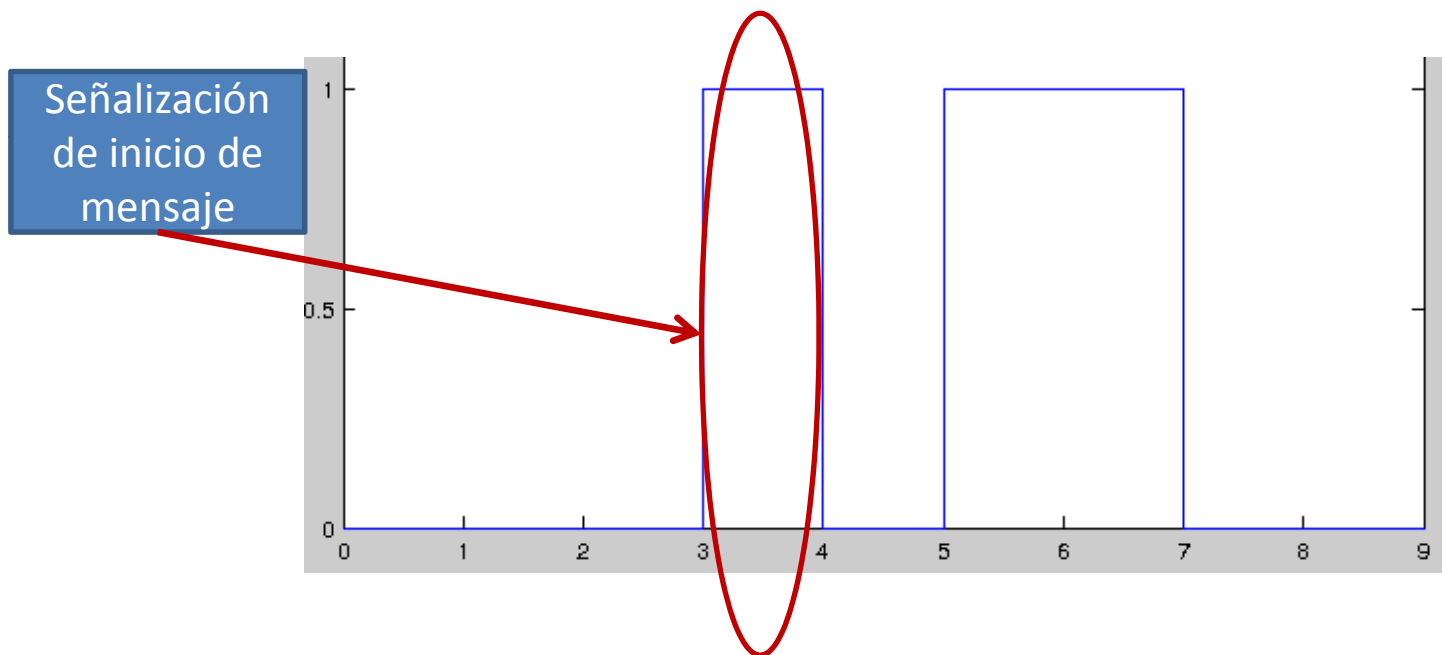
- Ejemplo: Supongamos que utilizamos un alfabeto {'A', 'B'}, codificado con un código **instantáneo** como:
  - A  $\rightarrow$  1
  - B  $\rightarrow$  01
  - Un nuevo código de finalización  $\rightarrow$  00
- Diagrama para enviar/recibir el mensaje “BA” desde la fuente al destino:



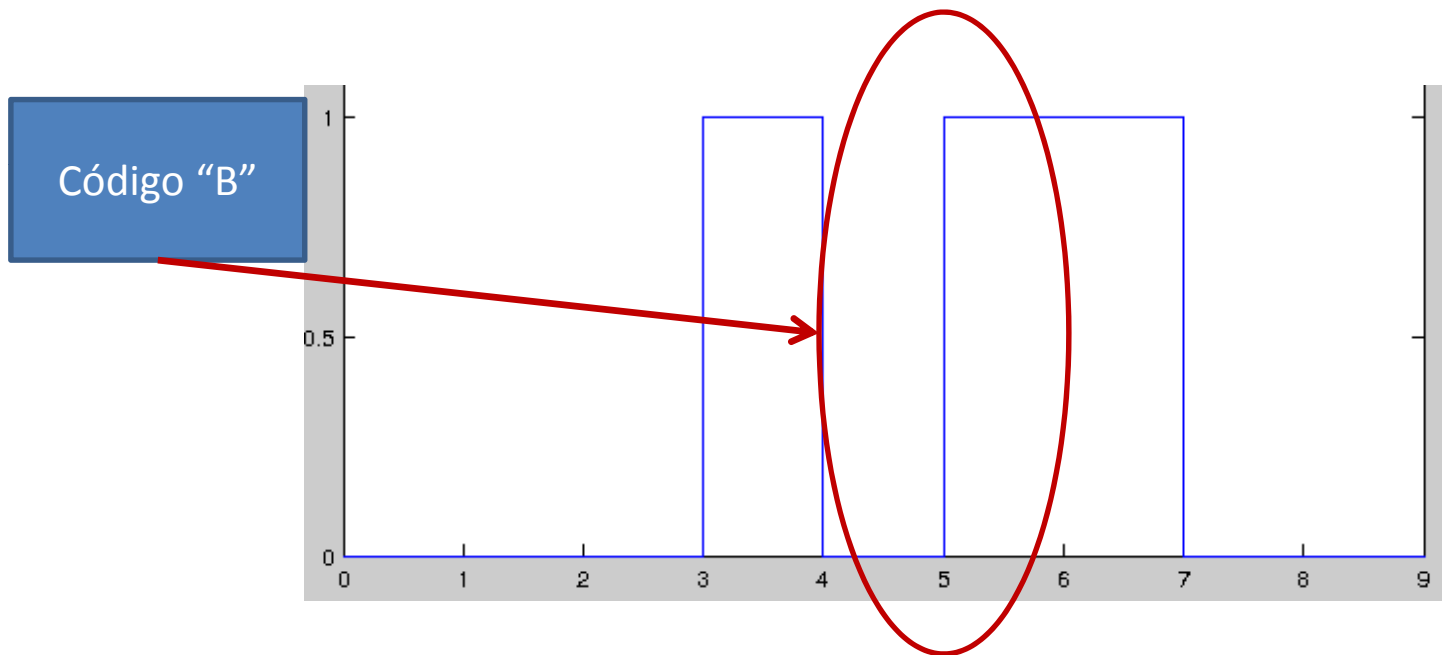
- Ejemplo: Supongamos que utilizamos un alfabeto {'A', 'B'}, codificado con un código **instantáneo** como:
  - $A \rightarrow 1$
  - $B \rightarrow 01$
  - Un nuevo código de finalización  $\rightarrow 00$
- Diagrama para enviar/recibir el mensaje "BA" desde la fuente al destino:



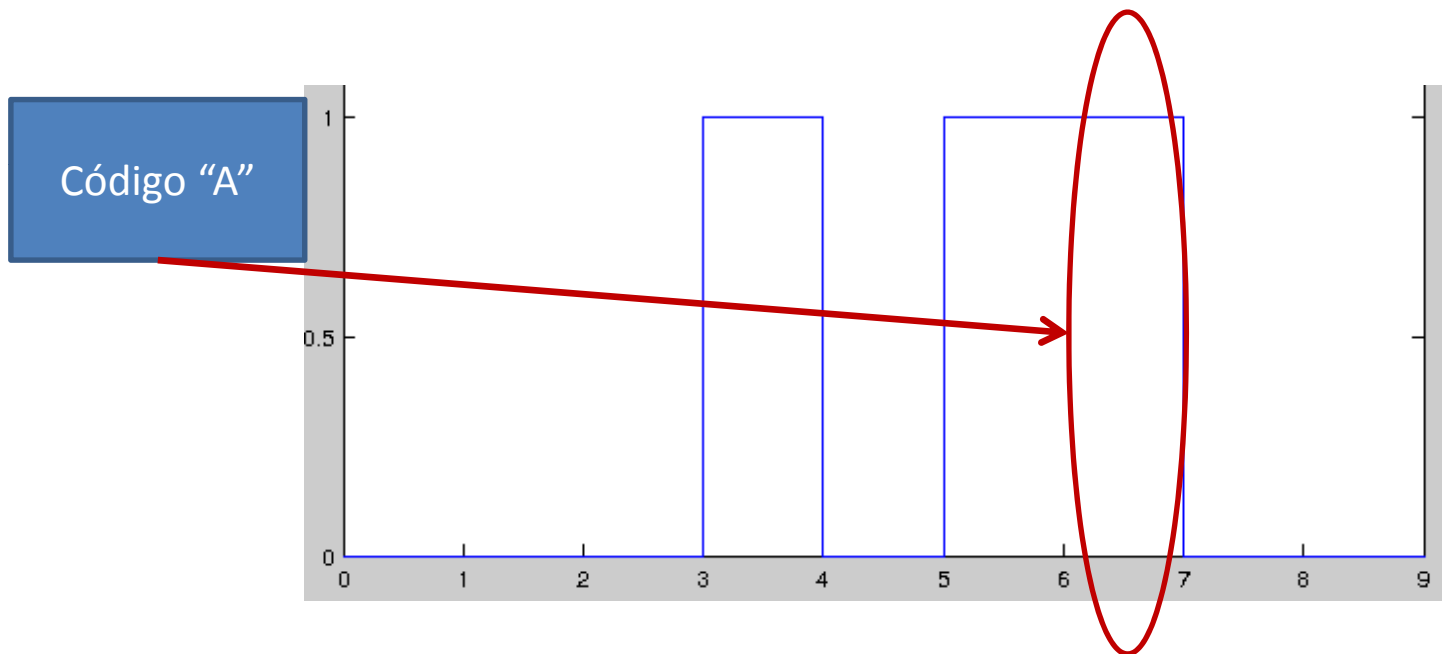
- Ejemplo: Supongamos que utilizamos un alfabeto {‘A’, ‘B’}, codificado con un código **instantáneo** como:
  - $A \rightarrow 1$
  - $B \rightarrow 01$
  - Un nuevo código de finalización  $\rightarrow 00$
- Diagrama para enviar/recibir el mensaje “BA” desde la fuente al destino:



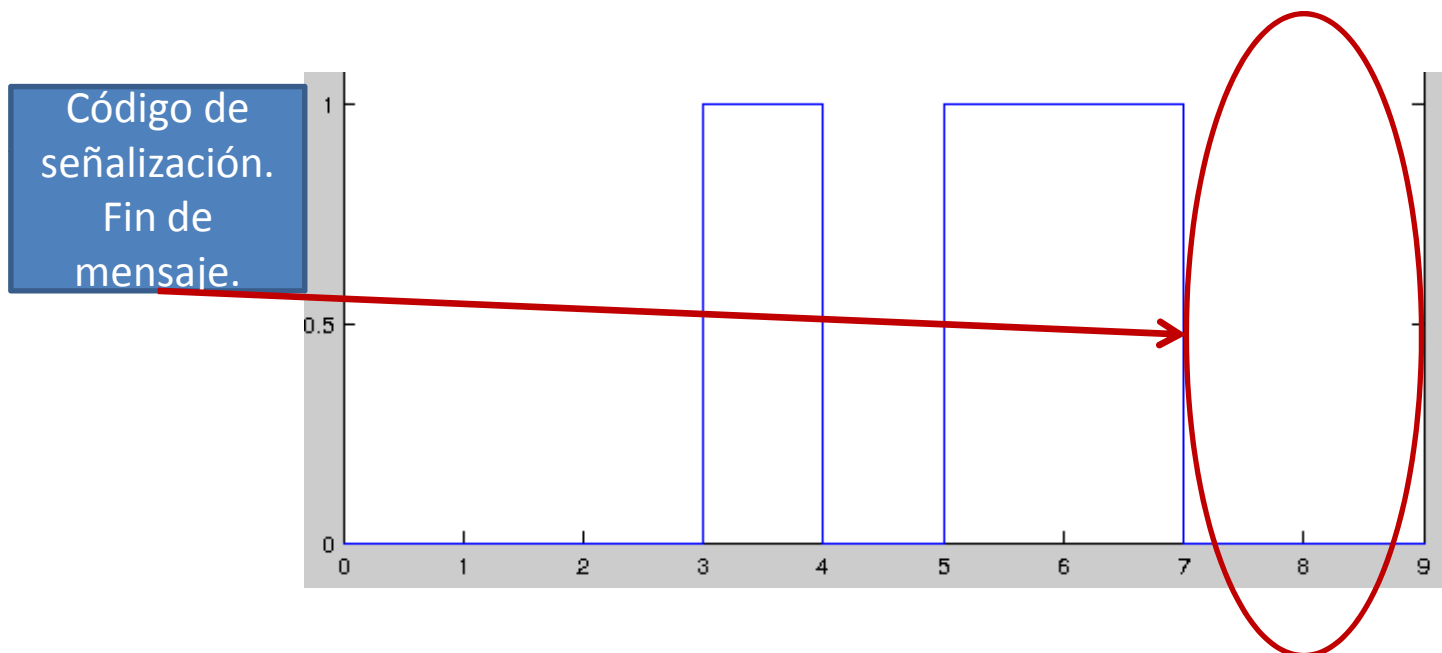
- Ejemplo: Supongamos que utilizamos un alfabeto {‘A’, ‘B’}, codificado con un código **instantáneo** como:
  - $A \rightarrow 1$
  - $B \rightarrow 01$
  - Un nuevo código de finalización  $\rightarrow 00$
- Diagrama para enviar/recibir el mensaje “BA” desde la fuente al destino:



- Ejemplo: Supongamos que utilizamos un alfabeto {'A', 'B'}, codificado con un código **instantáneo** como:
  - $A \rightarrow 1$
  - $B \rightarrow 01$
  - Un nuevo código de finalización  $\rightarrow 00$
- Diagrama para enviar/recibir el mensaje "BA" desde la fuente al destino:



- Ejemplo: Supongamos que utilizamos un alfabeto {'A', 'B'}, codificado con un código **instantáneo** como:
  - $A \rightarrow 1$
  - $B \rightarrow 01$
  - Un nuevo código de finalización  $\rightarrow 00$
- Diagrama para enviar/recibir el mensaje "BA" desde la fuente al destino:





- **Ejercicio de fin de sesión: Proyecto CodigoInstantaneo.**
  - Elaborar un código instantáneo que permita codificar los símbolos ASCII 'A', 'B' y 'C'.
  - Modificar el código del Arduino emisor y del Arduino receptor para enviar y recibir mensajes de este alfabeto (longitud máxima de 100 caracteres/mensaje). **No olvidar iniciar las comunicaciones con un ciclo en alta para indicar al receptor que comienzan las comunicaciones, y utilizar el símbolo de parada para finalizarlas.**
  - Elaborar programas de prueba para pedir por teclado una cadena de símbolos "A", "B" y/o "C", enviarlos por láser y recibirlos en otro PC (programas emisorPC y receptorPC).

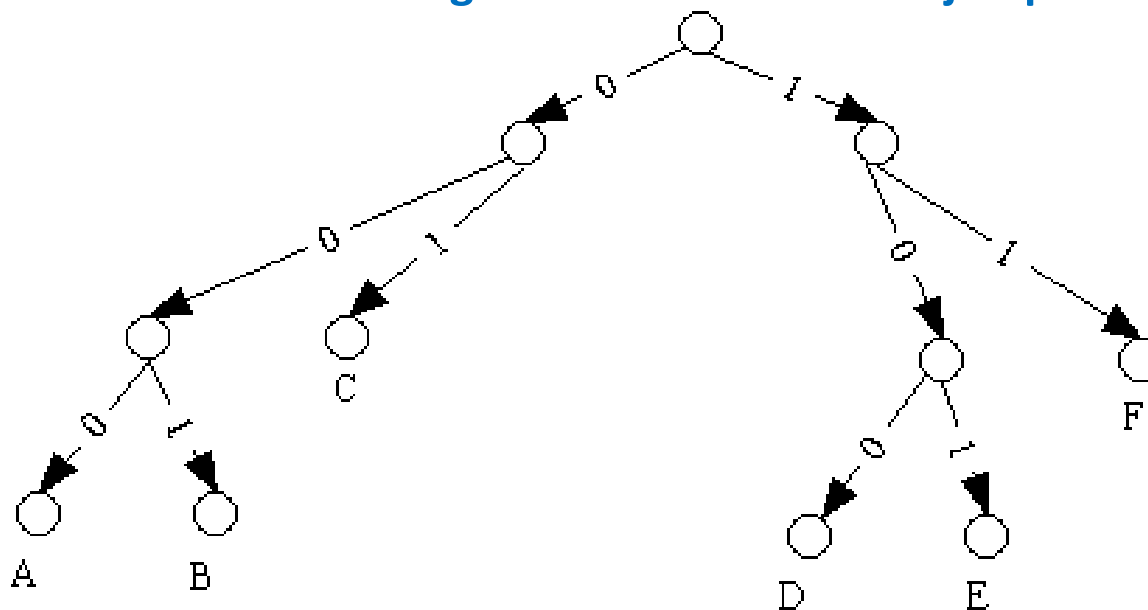
- Ejercicio de fin de sesión: Proyecto CodigoInstantaneo.
- **Reutilización del proyecto práctico final de la Práctica 1.**
- Los programas **emisorPC** y **receptorPC** pueden reutilizarse sin modificaciones. **A la hora de usarlos, tener en cuenta que sólo podemos escribir caracteres 'A', 'B' o 'C'.**
- El programa **emisorArdu**:
  - Modificarlo al programa **emisorArduInstantaneo**.
  - Debe tener un bucle infinito que:
    - Recoja por USB el mensaje desde **emisorPC**.
    - Traduzca cada símbolo en el código instantáneo.
    - Envíe los bits del símbolo.
    - Al finalizar, envíe los bits del código instantáneo correspondientes al símbolo de finalización.
    - Envíe por USB un mensaje “OK” al programa **emisorPC** si todo fue bien, o un error en caso contrario.

- Ejercicio de fin de sesión: Proyecto CodigoInstantaneo.
  - El programa **receptorArdu**:
    - Modificarlo al programa **receptorArduInstantaneo**.
    - Debe tener un bucle infinito que:
      - Mientras no se reciba un bit **LASER\_HIGH**, esté en estado “en espera”.
      - Cuando se reciba un bit **LASER\_HIGH**, pase al estado “recibiendo”.
      - Mientras esté en estado “recibiendo”, debe:
        - Leer bit a bit los *laserbits* recibidos por el fotorreceptor.
        - Tratar de decodificar, bit a bit, el código instantáneo.
        - Cuando se decodifique un símbolo, incluirlo en un buffer de salida.
        - Si el símbolo es el de finalización, pasar a estado “en espera”.
    - Enviar por USB el buffer al programa **receptorPC**.

# SESIÓN 2

1. **Codificación en canales sin ruido**
2. **Códigos instantáneos**
3. **Códigos óptimos**

- En la Teoría de la Información y la Codificación, un código para canales sin ruido se dice óptimo si la esperanza del número de bits utilizados para codificar un símbolo es mínima.
- **En este seminario, desarrollaremos códigos óptimos de Huffman para enviar y recibir información desde la plataforma láser construida con Arduino.**
- Representaremos el código mediante su árbol. Ejemplo:



- Elaboraremos la práctica en varias fases:
  - **Fase 1: Obtención del alfabeto y cálculo de probabilidades.**
  - **Fase 2: Cálculo del código óptimo y de su árbol.**
  - **Fase 3: Envío del árbol de código a los Arduinos emisor y receptor, para que puedan enviar y recibir la información con el mismo código.**
  - **Fase 4: Utilizar los programas emisorPC y receptorPC desarrollados en la práctica 1 para envío y recepción de mensajes.**
- 
- **Las Fases 1 y 2 se realizarán en PC.**
  - **La Fase 3 implica modificar los servidores de Arduino emisor y Arduino receptor para que puedan representar árboles de códigos.**
  - **La fase 4 ya está implementada desde la práctica 1.**

- Para la Fase 1, se proporciona un fragmento de *El Quijote* que contiene todas las letras del alfabeto castellano (salvo la ñ), junto con los símbolos de puntuación “:”, “,” y “;” (total: 30 símbolos).
- **Para mayor legibilidad, el fichero “quijote.txt” también tiene saltos de línea, que se deberán ignorar al leer el fichero para el cálculo de probabilidades.**
- **A todos los caracteres, se le deberá añadir uno más (código a utilizar para señalar el fin del mensaje).**
- El código de señalización de fin de mensaje sólo se utilizará una vez por trama enviada por láser, al final, para indicar que se termina la transmisión.



– ¿Cómo calcular las probabilidades?

1. **Crear un vector, y asignar a cada índice de componente un símbolo.** (Por ejemplo, la 'A' irá en la componente 0, la 'B' en la componente 2, etc..).
2. **Abrir el fichero y contabilizar en el vector cuántas veces aparece cada carácter.**
3. **Incluir en el vector, al final, una nueva componente para el símbolo de finalización. Indicar que aparece sólo 1 vez.**
4. **Dividir el número de veces que aparece cada caracter por el número total de caracteres del fichero +1 (sin contabilizar los símbolos de salto de línea, y +1 por el carácter de finalización).**

- Se facilita la biblioteca <utilsPC.h> con los prototipos de dos funciones para leer el fichero y para calcular sus probabilidades (ver fichero en carpeta **include** del material proporcionado por el profesor).

```
20 bool leerFichero(const char *filename, char *&contenido);
```

```
59 bool calcularProbabilidades(const char *cadena, char *&codigos, float *&probabilidades,  
60                             int &tamCodigos);
```

- Se facilita la biblioteca <utilsPC.h> con los prototipos de dos funciones para leer el fichero y para calcular sus probabilidades (ver fichero en carpeta **include** del material proporcionado por el profesor).

```
20 bool leerFichero(const char *filename, char *&contenido);
```

Nombre del fichero a leer

```
59 bool calcularProbabilidades(const char *cadena, char *&codigos, float *&probabilidades,  
60                             int &tamCodigos);
```

- Se facilita la biblioteca <utilsPC.h> con los prototipos de dos funciones para leer el fichero y para calcular sus probabilidades (ver fichero en carpeta **include** del material proporcionado por el profesor).

```
20 bool leerFichero(const char *filename, char *&contenido);
```

Salida con todo el contenido del fichero.

```
59 bool calcularProbabilidades(const char *cadena, char *&codigos, float *&probabilidades,  
60                             int &tamCodigos);
```

- Se facilita la biblioteca <utilsPC.h> con los prototipos de dos funciones para leer el fichero y para calcular sus probabilidades (ver fichero en carpeta **include** del material proporcionado por el profesor).

```
20 bool leerFichero(const char *filename, char *&contenido);
```

La función tiene como entrada el nombre de un fichero, lo abre, reserva memoria para todo su contenido, lo lee y devuelve en la variable “contenido”.

```
59 bool calcularProbabilidades(const char *cadena, char *&codigos, float *&probabilidades,  
60                             int &tamCodigos);
```

- Se facilita la biblioteca <utilsPC.h> con los prototipos de dos funciones para leer el fichero y para calcular sus probabilidades (ver fichero en carpeta **include** del material proporcionado por el profesor).

```
20 bool leerFichero(const char *filename, char *&contenido);
```

```
59 bool calcularProbabilidades(const char *cadena, char *&codigos, float *&probabilidades,  
60                             int *&tamCodigos);
```

Mensaje de entrada a utilizar para  
calcular las probabilidades de  
ocurrencia de cada carácter ASCII que  
contiene

- Se facilita la biblioteca <utilsPC.h> con los prototipos de dos funciones para leer el fichero y para calcular sus probabilidades (ver fichero en carpeta **include** del material proporcionado por el profesor).

```
20 bool leerFichero(const char *filename, char *&contenido);
```

```
59 bool calcularProbabilidades(const char *cadena, char *&codigos, float *&probabilidades,  
60 int &tamCodigos);
```

Vector de salida. Contiene una lista con todos los caracteres ASCII presentes en "cadena". Se reserva dentro de la función.

- Se facilita la biblioteca <utilsPC.h> con los prototipos de dos funciones para leer el fichero y para calcular sus probabilidades (ver fichero en carpeta **include** del material proporcionado por el profesor).

```
20 bool leerFichero(const char *filename, char *&contenido);
```

```
59 bool calcularProbabilidades(const char *cadena, char *&codigos, float *&probabilidades,  
60                             int &tamCodigos);
```

Vector de salida. Contiene una lista con las probabilidades de los ASCII devueltos en el vector "codigos". También se reserva dentro de la función.



- Se facilita la biblioteca <utilsPC.h> con los prototipos de dos funciones para leer el fichero y para calcular sus probabilidades (ver fichero en carpeta **include** del material proporcionado por el profesor).

```
20 bool leerFichero(const char *filename, char *&contenido);
```

```
59 bool calcularProbabilidades(const char *cadena, char *&codigos, float *&probabilidades,  
60 int &tamCodigos);
```

Parámetro de salida. Contiene el tamaño de los vectores reservados “codigos” y “probabilidades”.

- Se facilita la biblioteca <utilsPC.h> con los prototipos de dos funciones para leer el fichero y para calcular sus probabilidades (ver fichero en carpeta **include** del material proporcionado por el profesor).

```
20 bool leerFichero(const char *filename, char *&contenido);
```

```
59 bool calcularProbabilidades(const char *cadena, char *&codigos, float *&probabilidades,  
60                             int &tamCodigos);
```



La función tiene una cadena de caracteres “ASCII” como entrada. Ignora todos los que no sean letras del abecedario (salvo la ñ), y todos los símbolos que no sean “.”, “,” o “;”. Reserva memoria para “codigos” y “probabilidades”, y devuelve su tamaño en “tamCodigos”. ¡También reserva memoria para el carácter de finalización!

- Se facilita la biblioteca <utilsPC.h> con los prototipos de dos funciones para leer el fichero y para calcular sus probabilidades (ver fichero en carpeta **include** del material proporcionado por el profesor).

```
20 bool leerFichero(const char *filename, char *&contenido);
```

```
59 bool calcularProbabilidades(const char *cadena, char *&codigos, float *&probabilidades,  
60                             int &tamCodigos);
```



Además, en “codigos” devuelve los códigos válidos de la cadena (+1 extra para finalización de mensaje), y en probabilidades[i] la probabilidad de ocurrencia del código en codigos[i].

- Ejemplo: Suponiendo que la cadena de entrada contiene “aBbAa”:

```
59 bool calcularProbabilidades(const char *cadena, char *&codigos, float *&probabilidades,
60                             int &tamCodigos);
```

**Cadena** → “aBbAa”

Se proporciona la salida:

**codigos** → {'A', 'B', '(código de finalización cualquiera)'}  
**Probabilidades** → {3/6, 2/6, 1/6}

**tamCodigos** → 3

Con estas probabilidades, ya se puede aplicar el método para el cálculo del árbol de códigos óptimos.

- Ejercicio del Cuaderno de Prácticas:
  - El programa **PruebaProbabilidades**:
    - Consiste en crear las funciones anteriores para leer fichero y calcular probabilidades, y hacer un programa de prueba.
    - Se debe leer el fichero con el fragmento de *El Quijote* proporcionado en el material.
    - Se debe contabilizar todos los símbolos del alfabeto que aparecen en el texto. Obviar caracteres como ‘\n’ u otros que no sean del alfabeto.
    - Insertar un nuevo símbolo (a escoger por el alumno) como símbolo de finalización. **Este símbolo no debe pertenecer al alfabeto de la fuente**. Sólo se contabilizará una única vez: El fin del texto.
    - Calcular las probabilidades de cada símbolo.
    - Mostrar los símbolos por pantalla, ordenados por su probabilidad.

- Para la Fase 2, implementaremos un código Huffman.
- Tenemos que tener en cuenta que la representación debe ser eficiente y que **no puede utilizar memoria dinámica**, pues Arduino no tiene memoria ni capacidades para gestionar memoria dinámica.
- La representación del árbol del código será compartida por el PC y por Arduino, para facilitar que tanto el PC (a la hora de generar el código óptimo) como Arduino (a la hora de utilizarlo) puedan reutilizar el mismo código fuente en C.

### ¿Cómo construir el árbol del código Huffman?

- Repasemos el procedimiento para codificar el alfabeto {A, B, C, D} con las probabilidades  $p(A) = p(C) = 0,3$ ,  $p(B) = p(D) = 0,2$

### ¿Cómo construir el árbol del código Huffman? (I)

- Repasemos el procedimiento para codificar el alfabeto {A, B, C, D} con las probabilidades  $p(A) = p(C) = 0,3$ ,  $p(B) = p(D) = 0,2$
- En primer lugar, ordenar los símbolos por probabilidad decreciente:

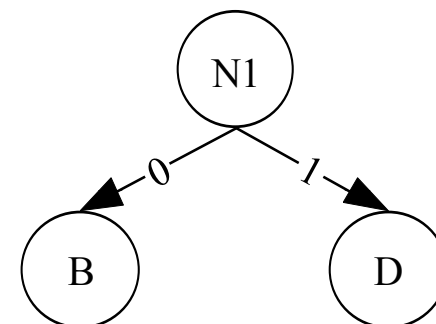
A	C	B	D
0,3	0,3	0,2	0,2

### ¿Cómo construir el árbol del código Huffman? (II)

- En segundo lugar, mientras quede más de un elemento en el vector, agrupar los dos símbolos de menor probabilidad en un nuevo árbol, asignando “0” al hijo izquierda y “1” al hijo derecha.
- Asignar a la raíz su probabilidad: La suma de las probabilidades de los elementos seleccionados en el vector.
- Asignar el contenido (código) del primer elemento seleccionado como hijo izquierda, y el contenido del segundo como hijo derecha.
- Eliminar los elementos seleccionados del vector, e insertar uno nuevo, asignado a la raíz del nuevo árbol creado.

A	C	B	D
0,3	0,3	0,2	0,2

A	C	N1
0,3	0,3	0,4





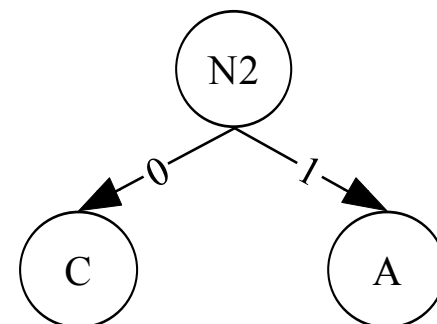
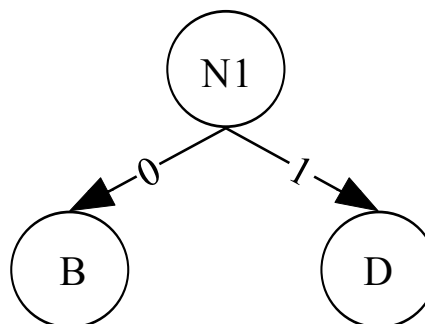
### ¿Cómo construir el árbol del código Huffman? (III)

- Como queda más de un elemento en el vector, reordenar por probabilidades:

N1	C	A
0,4	0,3	0,3

- Realizar el mismo procedimiento con los dos últimos elementos:
  - Creación de un nodo raíz, eliminación del vector, creación de un elemento en el vector con probabilidad igual a la suma de las probabilidades de los elementos involucrados:

N1	N2
0,4	0,6



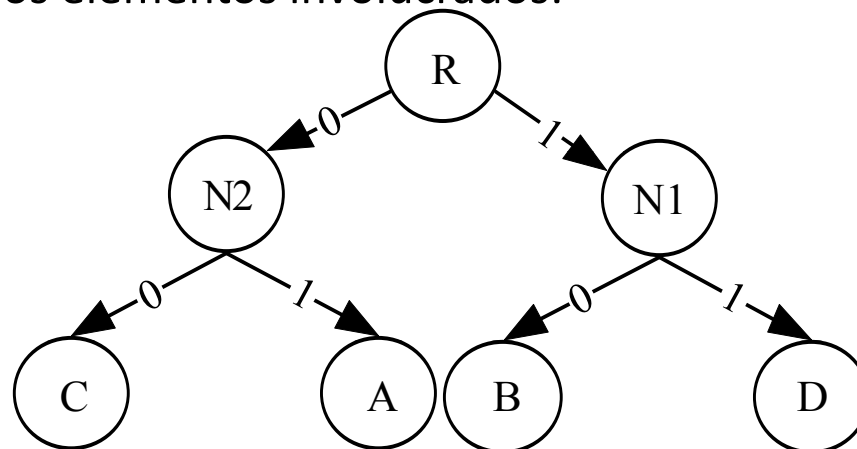
### ¿Cómo construir el árbol del código Huffman? (IV)

- Como queda más de un elemento en el vector, reordenar por probabilidades:

N2	N1
0,6	0,4

- Realizar el mismo procedimiento con los dos últimos elementos:
  - Creación de un nodo raíz, eliminación del vector, creación de un elemento en el vector con probabilidad igual a la suma de las probabilidades de los elementos involucrados:

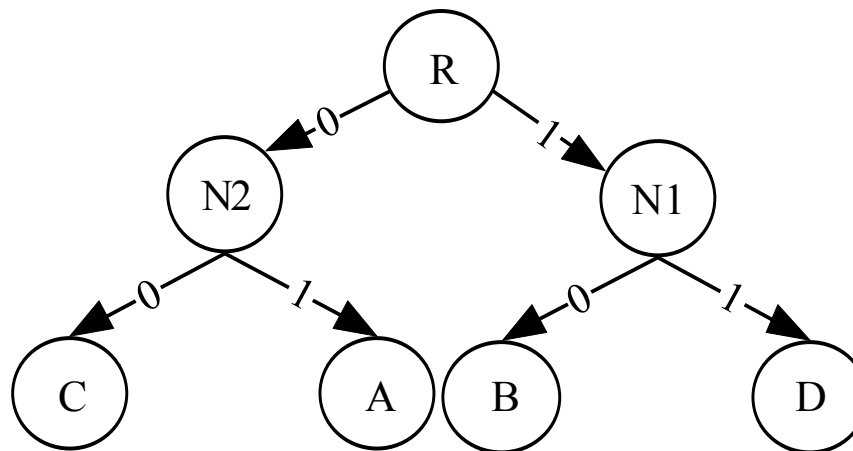
R
1



### ¿Cómo construir el árbol del código Huffman? (V)

- Como queda un único elemento en el vector, hemos terminado:

R
1



- Se ha creado un árbol de código Huffman que codifica al alfabeto de la siguiente forma:

- **C → 00**
- **A → 01**
- **B → 10**
- **D → 11**

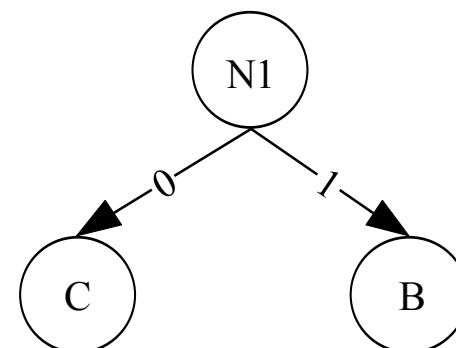
### Otro ejemplo más (I):

- Repasemos el procedimiento para codificar el alfabeto {A, B, C, D} con las probabilidades  $p(A)=0,4$ ,  $p(D)=0,3$ ,  $p(C)=0,2$ ,  $p(B)=0,1$
- En primer lugar, ordenar los símbolos por probabilidad decreciente:

A	D	C	B
0,4	0,3	0,2	0,1

- Seleccionamos los dos últimos valores del vector, correspondientes a los **nodos hoja "C" y "B"**.

A	D	N1
0,4	0,3	0,3



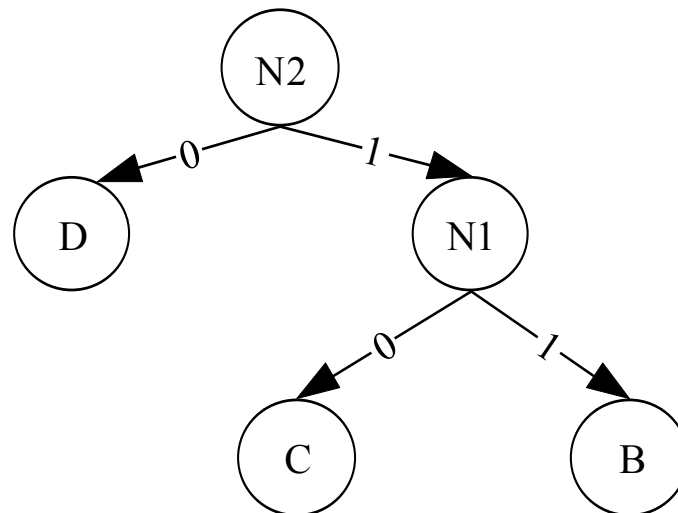
### Otro ejemplo más (II):

- Se reordenan por probabilidades (en este caso el vector queda igual):

A	D	N1
0,4	0,3	0,3

- Seleccionamos los dos últimos valores del vector, correspondientes al **nodo hoja "D"** y al **nodo intermedio N1**. Realizamos la misma operación:

A	N2
0,4	0,6

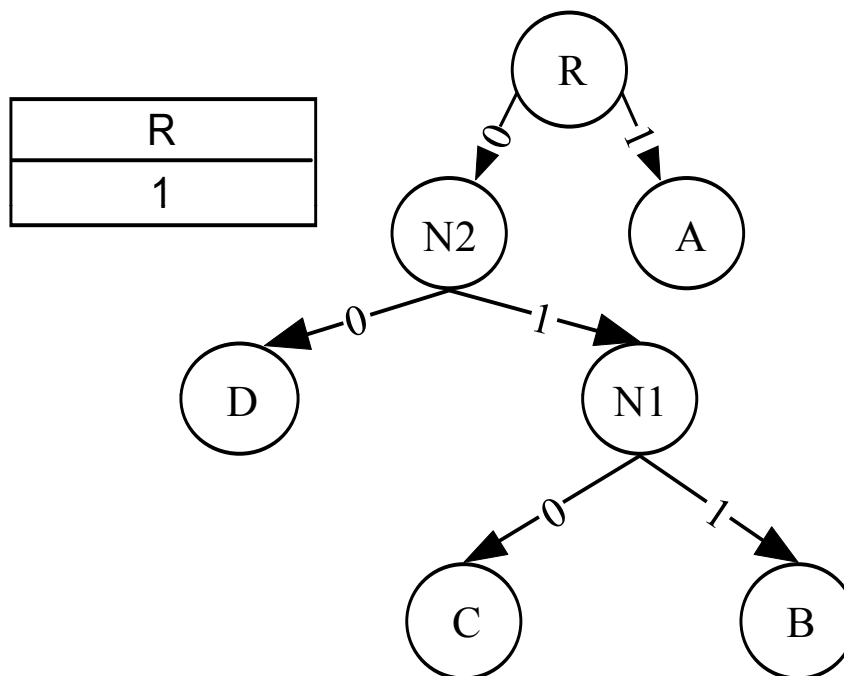


### Otro ejemplo más (III):

- Se reordenan por probabilidades:

N2	A
0,6	0,4

- Seleccionamos los dos últimos valores del vector, correspondientes al **nodo hoja "A"** y al **nodo intermedio N2**. Realizamos la misma operación:



## Sabiendo este procedimiento, tenemos que:

- Diseñar una estructura de datos que nos permita almacenar un árbol.
  - Que ocupe poco espacio (por las limitaciones de Arduino).
  - Que sea eficiente de recorrer (para codificar y decodificar rápidamente).
  - Que no utilice memoria dinámica (por las limitaciones de Arduino).
  - Que haga que la implementación del método anterior sea sencilla.
- 
- **Solución: Implementaremos el árbol con una matriz 2-D.**
- 
- ¿De qué tamaño?
  - ¿A qué se asocia cada dimensión?
  - ¿Qué valor contiene cada celda?

**Lo estudiamos a continuación.**

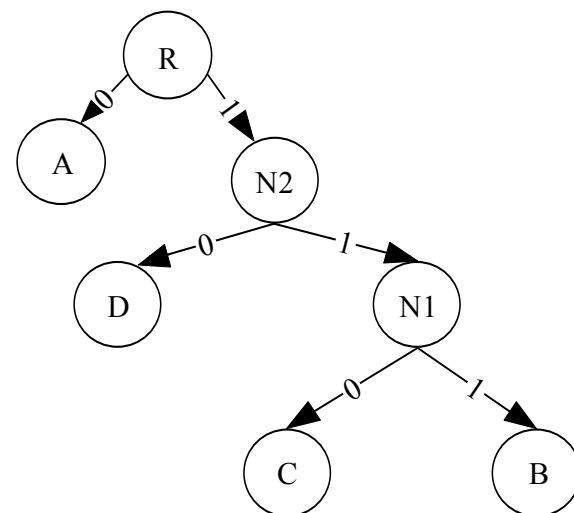
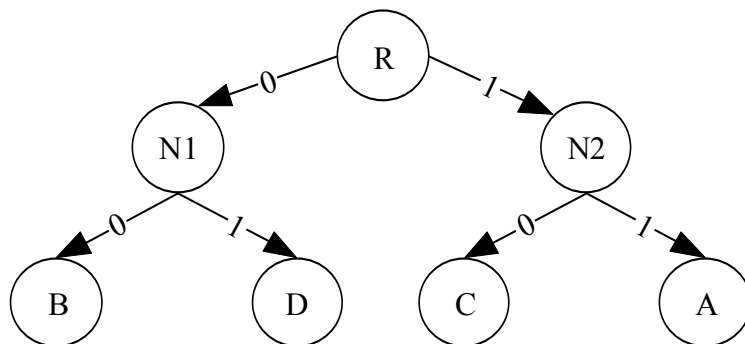
La representación de árbol será una matriz de  $n * 4$  componentes, donde “n” es el número máximo de nodos que podrá tener el árbol

Matriz Huffman[n][4] → **¿cuánto vale n?**

- En la primera dimensión, almacenaremos los nodos:
  - Huffman[0] es la información del nodo 0.
  - Huffman[1] es la información del nodo 1
  - Etc.
- Para cada nodo, necesitamos almacenar (por eficiencia al recorrer el árbol):
  - Su código asociado (si tiene).
  - La posición del hijo izquierda en el vector (si tiene).
  - La posición del hijo derecha en el vector (si tiene).
  - La posición del padre en el vector (si tiene).



¿Cuántos nodos como máximo podemos tener? Fijémonos en los ejemplos anteriores en los que usábamos 4 códigos:



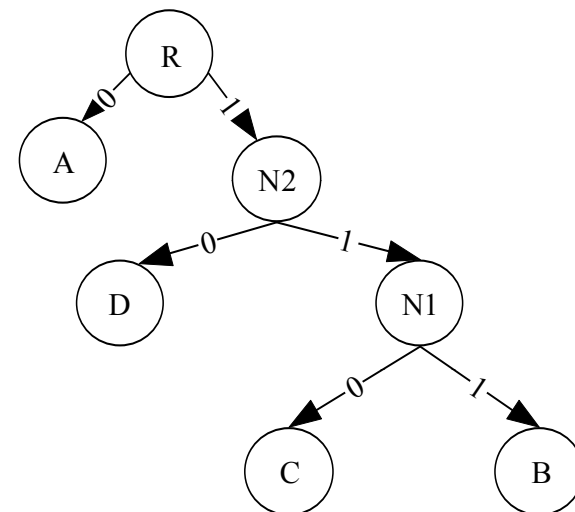
- Siempre, independientemente de la forma del árbol, necesitábamos 7 nodos (4 para nodos hoja + 3 intermedios)
- ¿Y si hubiésemos necesitado codificar 8 símbolos? Hubiera sido: 8 nodos hoja + 4 nodos en el nivel superior, + 2 nodos en el nivel superior, + 1 nodo raíz.
- **¿Y para almacenar 32 símbolos?  $32=2^5$ ; número de nodos  $n=2^{(5+1)}-1=$**

**Necesitamos almacenar y direccionar, como mucho, 63 nodos. Con un tipo char basta y sobra para direccionar los nodos dentro de un vector de nodos. También es suficiente para direccionar nodos en nuestra matriz.**

- Asociaremos cada nodo con una posición de la matriz en su primera dimensión.
- La componente **Huffman[i][0]** tendrá el código ASCII asociado al nodo i.
- La componente **Huffman[i][1]** tendrá el índice de la matriz donde está el hijo izquierda del nodo i, o valor -1 si no tiene.
- La componente **Huffman[i][2]** tendrá el índice de la matriz donde está el hijo derecha del nodo i, o valor -1 si no tiene.
- La componente **Huffman[i][3]** tendrá el índice de la matriz donde está el padre del nodo i, o valor -1 si es la raíz.

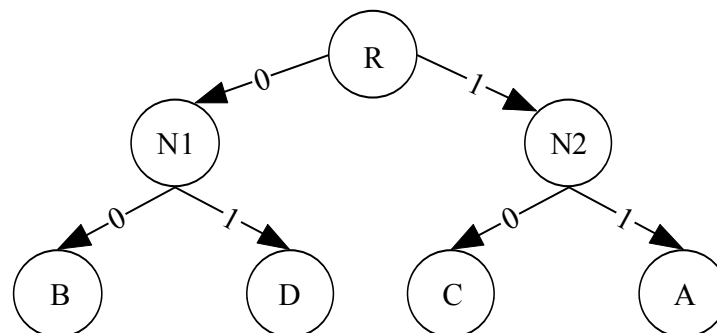
- Para simplificar la elaboración del algoritmo para obtener el código Huffman, guardaremos todos los nodos hoja en las primeras componentes de la matriz, e iremos añadiendo nodos y referenciando los existentes según vaya operando el algoritmo.
- En particular, la siguiente matriz es un ejemplo de cómo se codifica el siguiente árbol:

0	1	2	3	4	5	6
A	B	C	D	--	--	--
-1	-1	-1	-1	2	3	0
-1	-1	-1	-1	1	4	5
6	4	4	5	5	6	-1



- Otro ejemplo: Codificación del siguiente árbol:

0	1	2	3	4	5	6
A	B	C	D	--	--	--
-1	-1	-1	-1	1	2	4
-1	-1	-1	-1	3	0	5
5	4	5	4	6	6	-1



- En el código fuente para implementar el código Huffman, en cada paso se irá generando un nuevo nodo. Tendremos únicamente que indicar cuáles son sus hijos (que se asumen ya creados) y, para esos hijos, indicar cuál es su padre.
- Necesitaremos un vector auxiliar **Orden** para reordenar probabilidades.

## ¿Cómo construir el árbol del código Huffman? (I)

- Codificaremos el alfabeto {A, B, C, D} con las probabilidades  $p(A) = p(C) = 0,3$ ,  $p(B) = p(D) = 0,2$
- Vector **codigos**  $\rightarrow \{'A', 'B', 'C', 'D'\}$
- Vector **Prob**  $\rightarrow \{0.3, 0.2, 0.3, 0.2\}$
- En primer lugar, ordenar los símbolos por probabilidad decreciente:
  - Vector **Orden**  $\rightarrow \{0, 2, 1, 3\}$
  - Vector **CopiaProb**  $\rightarrow \{0.3, 0.3, 0.2, 0.2\}$
  - El código de mayor probabilidad es `codigos[ Orden[0] ]`
  - El siguiente código de mayor probabilidad es `codigos[ Orden[1] ]`
  - Etc.

### ¿Cómo construir el árbol del código Huffman? (II)

- Inicializamos nuestra tabla Huffman con los nodos hoja. Sin hijos y con padres desconocidos.
- Vector **codigos**  $\rightarrow \{'A', 'B', 'C', 'D'\}$
- Vector **Prob**  $\rightarrow \{0.3, 0.3, 0.2, 0.2\}$
- Vector **Orden**  $\rightarrow \{0, 2, 1, 3\}$ , con **tamOrden**= 4
- Variable **treeSize**=4, porque actualmente el árbol tiene 4 nodos (además “4” es el primer elemento vacío donde se puede crear un nodo).

0	1	2	3	4	5	6
---	---	---	---	---	---	---

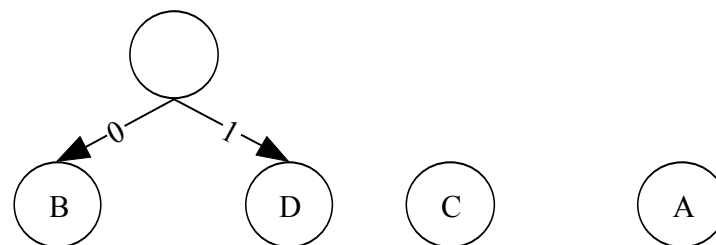
<b>A</b>	<b>B</b>	<b>C</b>	<b>D</b>	--	--	--
-1	-1	-1	-1	??	??	??
-1	-1	-1	-1	??	??	??
??	??	??	??	??	??	??



### ¿Cómo construir el árbol del código Huffman? (III)

- Seleccionamos los nodos de menor probabilidad (Orden[3] y Orden[2]).
- Los agrupamos en un nuevo nodo, quedando así:

0	1	2	3	4	5	6
A	B	C	D	--	--	--
-1	-1	-1	-1	1	??	??
-1	-1	-1	-1	3	??	??
??	4	??	4	??	??	??



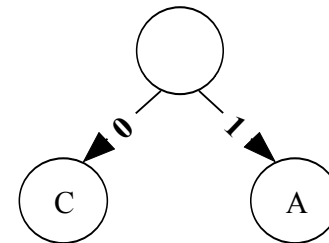
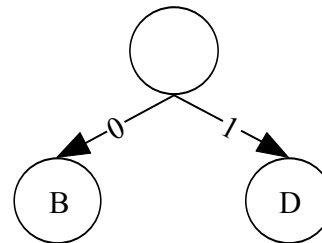
- Reorganizamos probabilidades y el orden:

- Vector **CopiaProb**  $\rightarrow \{0.4, 0.3, 0.3\}$
- Vector **Orden**  $\rightarrow \{4, 2, 0\}$ , con **tamOrden= 3**
- Variable **treeSize=5** (ya hay un nodo más).

### ¿Cómo construir el árbol del código Huffman? (IV)

- Seleccionamos los nodos de menor probabilidad (Orden[2] y Orden[1]).
- Los agrupamos en un nuevo nodo, quedando así:

0	1	2	3	4	5	6
A	B	C	D	--	--	--
-1	-1	-1	-1	1	<b>2</b>	??
-1	-1	-1	-1	3	<b>0</b>	??
<b>5</b>	4	<b>5</b>	4	??	??	??



- Reorganizamos probabilidades y el orden:

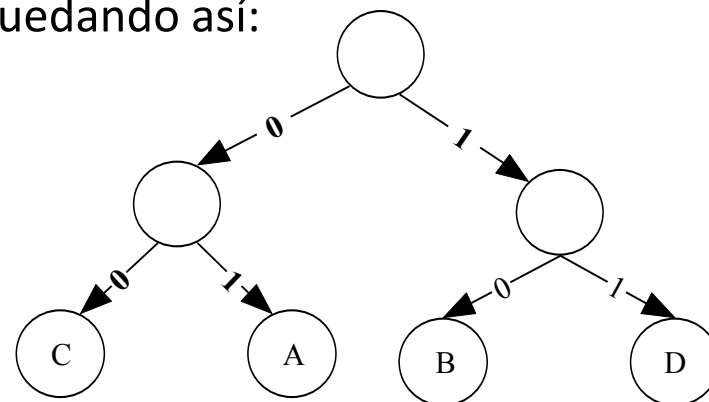
- Vector **CopiaProb** → {0.6, 0.4}
- Vector **Orden** → {5, 4}, con **tamOrden= 2**
- Variable **treeSize=6** (ya hay un nodo más).



### ¿Cómo construir el árbol del código Huffman? (V)

- Seleccionamos los nodos de menor probabilidad (Orden[1] y Orden[0]).
- Los agrupamos en un nuevo nodo, quedando así:

0	1	2	3	4	5	6
A	B	C	D	--	--	--
-1	-1	-1	-1	1	2	5
-1	-1	-1	-1	3	0	4
5	4	5	4	6	6	-1



- Reorganizamos probabilidades y el orden:
- Vector **CopiaProb** → {1}
- Vector **Orden** → {0}, con **tamOrden= 1**
- Variable **treeSize=7** (ya hay un nodo más).
- Sólo queda un elemento en Orden → Se ha terminado. Se pone -1 al padre de la raíz.

### ¿Cómo codificar un símbolo con el árbol ya construido?

- Para codificar un símbolo, buscamos el nodo hoja que contiene el símbolo desde el comienzo de la tabla.
- Una vez encontrado, visitamos sus padres hasta llegar a la raíz.
- El código es la secuencia de 0's y 1's generada **desde la raíz al nodo**.

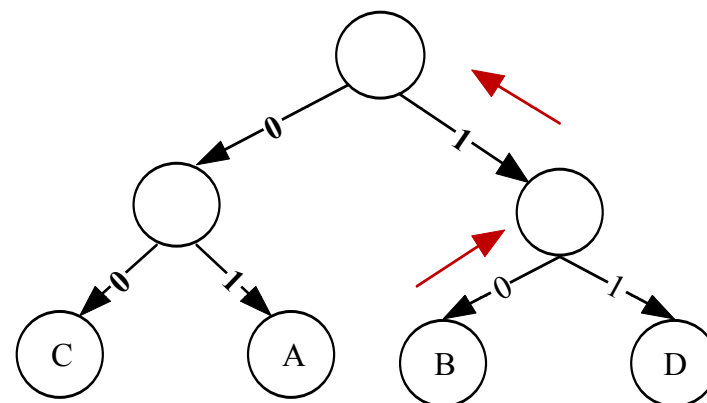
### – Ejemplo: Codificar el símbolo 'B':

0	1	2	3	4	5	6
A	B	C	D	--	--	--
-1	-1	-1	-1	1	2	5
-1	-1	-1	-1	3	0	4
5	4	5	4	6	6	-1

↑  
Paso 1

↑  
Paso 2

↑  
Paso 3



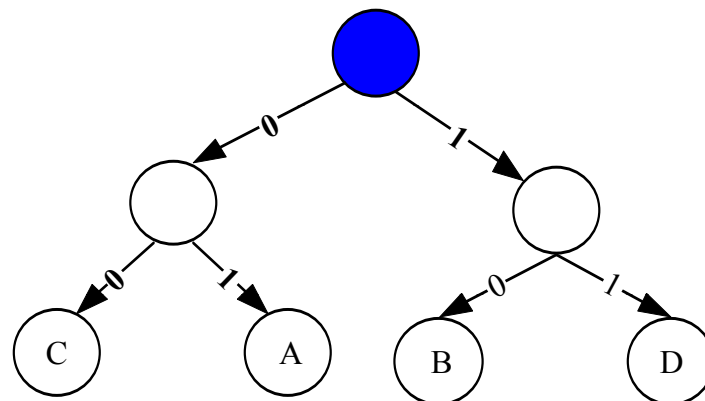
**El código es 10, de longitud 2**

### ¿Cómo decodificar un símbolo con el árbol ya construido? (I)

- Para decodificar un símbolo, partimos desde la raíz.
- Vamos recorriendo el árbol hasta llegar a un nodo hoja, según el código recibido.
- **Ejemplo: Decodificar el mensaje 0111: Partimos desde la raíz.**

0	1	2	3	4	5	6
A	B	C	D	--	--	--
-1	-1	-1	-1	1	2	5
-1	-1	-1	-1	3	0	4
5	4	5	4	6	6	-1

Paso 1

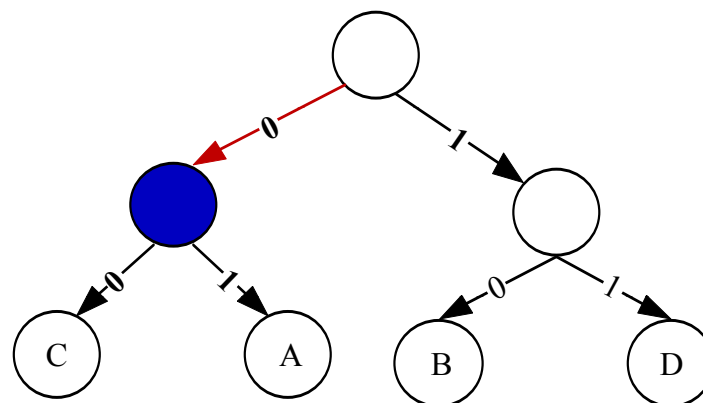


### ¿Cómo decodificar un símbolo con el árbol ya construido? (II)

- Para decodificar un símbolo, partimos desde la raíz.
- Vamos recorriendo el árbol hasta llegar a un nodo hoja, según el código recibido.
- **Ejemplo: Decodificar el mensaje 0111: Decodificamos el 0.**
- **Como es un 0, nos movemos al hijo izquierda**

0	1	2	3	4	5	6
A	B	C	D	--	--	--
-1	-1	-1	-1	1	2	<b>5</b>
-1	-1	-1	-1	3	0	4
5	4	5	4	6	6	-1

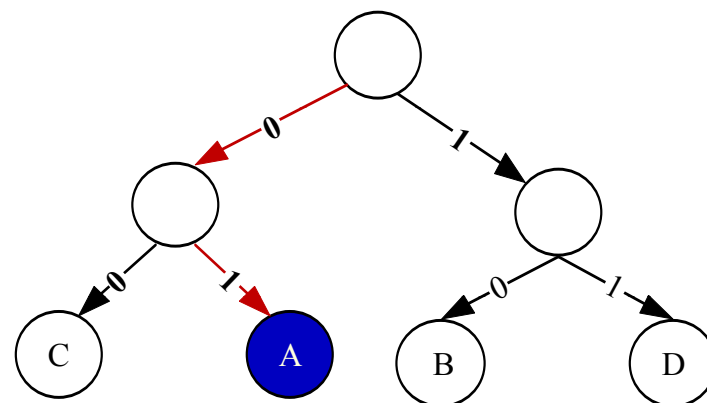
↑  
**Paso 2**



### ¿Cómo decodificar un símbolo con el árbol ya construido? (III)

- Para decodificar un símbolo, partimos desde la raíz.
- Vamos recorriendo el árbol hasta llegar a un nodo hoja, según el código recibido.
- **Ejemplo: Decodificar el mensaje 0111: Decodificamos el 1.**
- **Como es un 1, nos movemos al hijo derecha.**

0	1	2	3	4	5	6
A	B	C	D	--	--	--
-1	-1	-1	-1	1	2	<b>5</b>
-1	-1	-1	-1	3	<b>0</b>	4
5	4	5	4	6	6	-1

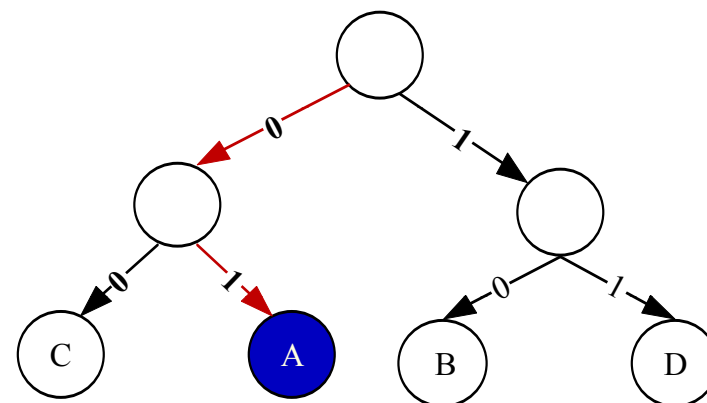


↑  
**Paso 3**

### ¿Cómo decodificar un símbolo con el árbol ya construido? (IV)

- Para decodificar un símbolo, partimos desde la raíz.
- Vamos recorriendo el árbol hasta llegar a un nodo hoja, según el código recibido.
- **Ejemplo: Decodificar el mensaje 0111: Llegamos a un nodo hoja.**
- **Como es nodo hoja, decodificamos 'A'.**
- **Reinicializamos el árbol.**

0	1	2	3	4	5	6
A	B	C	D	--	--	--
-1	-1	-1	-1	1	2	<b>5</b>
-1	-1	-1	-1	3	<b>0</b>	4
5	4	5	4	6	6	-1



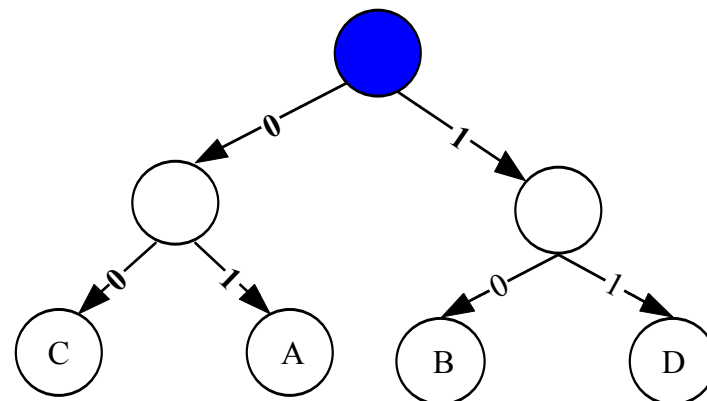
Paso 3

### ¿Cómo decodificar un símbolo con el árbol ya construido? (V)

- Para decodificar un símbolo, partimos desde la raíz.
- Vamos recorriendo el árbol hasta llegar a un nodo hoja, según el código recibido.
- **Ejemplo: Decodificar el mensaje 0111: Tras la reinicialización, volvemos a estar en la raíz.**

0	1	2	3	4	5	6
A	B	C	D	--	--	--
-1	-1	-1	-1	1	2	5
-1	-1	-1	-1	3	0	4
5	4	5	4	6	6	-1

Paso 1

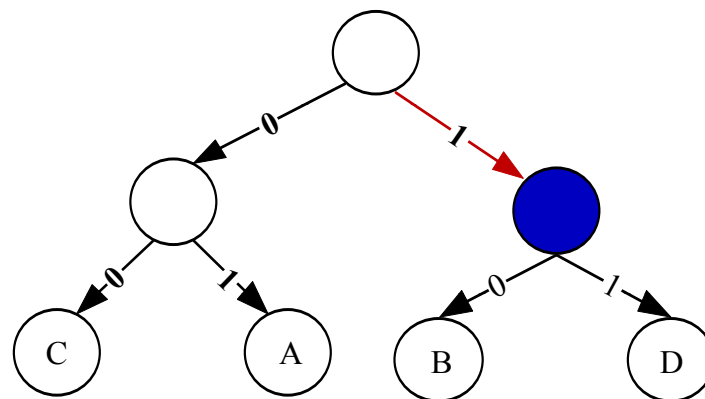


## ¿Cómo decodificar un símbolo con el árbol ya construido? (VI)

- Para decodificar un símbolo, partimos desde la raíz.
- Vamos recorriendo el árbol hasta llegar a un nodo hoja, según el código recibido.
- **Ejemplo: Decodificar el mensaje 0111: Decodificamos el 1.**
- **Como es un 1, nos movemos al hijo derecha.**

0	1	2	3	4	5	6
A	B	C	D	--	--	--
-1	-1	-1	-1	1	2	5
-1	-1	-1	-1	3	0	4
5	4	5	4	6	6	-1

## Paso 2



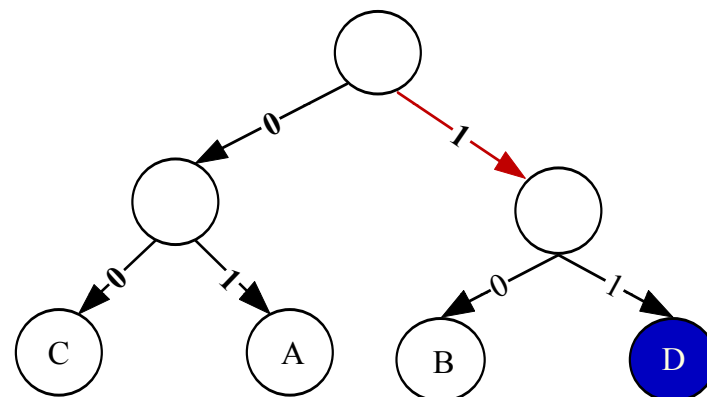


### ¿Cómo decodificar un símbolo con el árbol ya construido? (VII)

- Para decodificar un símbolo, partimos desde la raíz.
- Vamos recorriendo el árbol hasta llegar a un nodo hoja, según el código recibido.
- **Ejemplo: Decodificar el mensaje 0111: Decodificamos el 1.**
- **Como es un 1, nos movemos al hijo derecha.**

0	1	2	3	4	5	6
A	B	C	D	--	--	--
-1	-1	-1	-1	1	2	5
-1	-1	-1	-1	<b>3</b>	0	<b>4</b>
5	4	5	4	6	6	-1

↑  
**Paso 3**

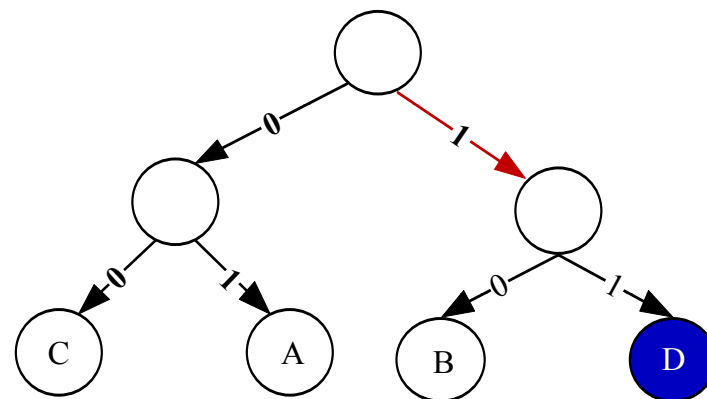


### ¿Cómo decodificar un símbolo con el árbol ya construido? (VIII)

- Para decodificar un símbolo, partimos desde la raíz.
- Vamos recorriendo el árbol hasta llegar a un nodo hoja, según el código recibido.
- **Ejemplo: Decodificar el mensaje 0111: Llegamos a un nodo hoja.**
- **Como es nodo hoja, decodificamos 'D'.**
- **Reinicializamos el árbol.**

0	1	2	3	4	5	6
A	B	C	D	--	--	--
-1	-1	-1	-1	1	2	5
-1	-1	-1	-1	<b>3</b>	0	<b>4</b>
5	4	5	4	6	6	-1

↑  
**Paso 3**

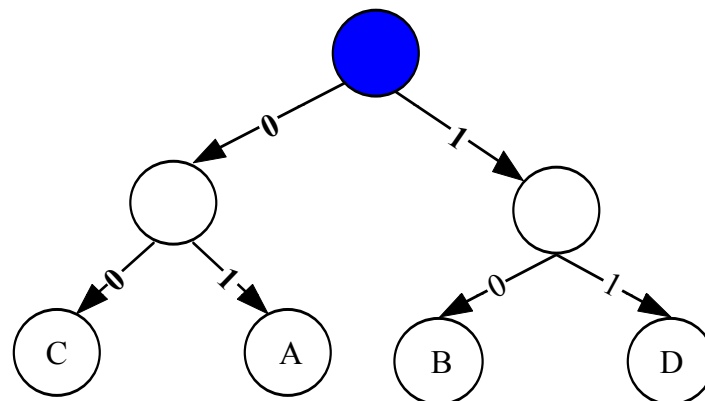


### ¿Cómo decodificar un símbolo con el árbol ya construido? (IX)

- Para decodificar un símbolo, partimos desde la raíz.
- Vamos recorriendo el árbol hasta llegar a un nodo hoja, según el código recibido.
- **Ejemplo: Decodificar el mensaje 0111: Fin del mensaje.**
- **Se decodifica como 'AD'.**

0	1	2	3	4	5	6
A	B	C	D	--	--	--
-1	-1	-1	-1	1	2	5
-1	-1	-1	-1	3	0	4
5	4	5	4	6	6	-1

Paso 1



## Ejercicio (Cuaderno de prácticas): HuffmanTree

- Se desea codificar con un código Huffman el siguiente mensaje en castellano:

*Ata la jaca a la estaca*

- Asumiendo la no sensibilidad a mayúsculas y minúsculas, **¿Cuántos símbolos diferentes se deben codificar? ¿Cuál es la probabilidad de ocurrencia de cada símbolo?**
- **Partiendo del análisis anterior, elabore a mano la construcción de la tabla que representa el árbol de codificación Huffman para codificar de forma óptima el mensaje.**
- **No considere codificar todos los símbolos del idioma castellano, sólo los que aparecen en el mensaje.**

### Ejercicio (Cuaderno de prácticas): HuffmanTree. Parte II. Implementación

- Se deberá crear una nueva biblioteca (**Huffman.h** y **Huffman.cpp**) que permita crear una representación en memoria estática del árbol, así como métodos para codificar y decodificar un código.
- Por ejemplo, para representar el árbol Huffman, se puede utilizar una estructura de datos como la siguiente:

```

5 /**
6  * Estructura que representa un árbol en Arduino, como un vector de nodos.
7  * Una variable x de tipo ArduTree tendrá representación de vector de 2
8  * dimensiones. x[i] será el nodo i del árbol. La raíz del árbol siempre
9  * deberá ser x[0]. x[i][0] contendrá el código ASCII asociado
10 * al código Huffman (sólo para nodos hojas). x[i][1] tendrá
11 * el índice del hijo a la izquierda (si tiene) o -1 en caso contrario.
12 * x[i][2] tendrá el índice del hijo a la derecha (si tiene) o -1 en caso
13 * contrario. x[i][3] tendrá el índice del padre del nodo.
14 *
15 * La estructura ArduTree tiene capacidad para almacenar hasta un máximo de
16 * 32 nodos hoja (+16 en el nivel superior, +8 en el superior, etc.),
17 * suficiente para las prácticas de la asignatura
18 */
19 struct ArduTree {
20     unsigned char root; // Indica el índice de la raíz del árbol
21     char x[32+16+8+4+2+1][4]; // Arbol representado como vector de nodos
22 };
23

```

### Ejercicio (Cuaderno de prácticas): HuffmanTree. Parte II. Implementación

- El árbol de Huffman se genera desde PC, pero también se utilizará en Arduino. Construiremos una nueva biblioteca **<Huffman.h>**, estándar y que pueda ser reutilizada tanto por el compilador **gcc/g++** como por **avr-gcc**.
- Se deberá incluir una función en una nueva biblioteca **<Huffman.h>** y su correspondiente fichero **.cpp** para, dado un conjunto de símbolos de entrada con sus probabilidades de ocurrencia, se genere el árbol de Huffman asociado.

```

24
25 /**
26  * Función para hallar un árbol que representa un código de Huffman
27  * @param codigos Lista de códigos a usar
28  * @param probabilidades Probabilidades de los códigos en el vector "codigos"
29  * @param tamCodigos Tamaño del vector anterior
30  * @param salida Arbol de salida, indicando en salida.root la raíz del árbol.
31  * Las "tamCodigos" primeras componentes de "salida" son los nodos hoja.
32  * @return true si la operación se realizó con éxito, false en otro caso
33  */
34 bool Huffman(char *codigos, float *probabilidades, int tamCodigos, ArduTree &salida);

```

### Ejercicio (Cuaderno de prácticas): HuffmanTree. Parte II. Implementación

- Necesitaremos, también, otra función **HuffmanEncoder** que, dado un carácter ASCII de entrada válido, devuelva el código Huffman asociado.
- Esta función tendrá que conocer el árbol Huffman para poder generar el código del mismo.
- Devuelve dos valores:
  - Un unsigned short (2 bytes) **code**, con los bits del código Huffman.
  - Un unsigned short **nBits**, con el número de bits útiles en **code**.

```

37 /**
38  * Codifica un caracter ASCII con un código Huffman, según el árbol del código
39  * @param tree Arbol del código Huffman a usar para la codificación
40  * @param ascii Código ASCII a codificar
41  * @param code Código de salida (en bits)
42  * @param nBits Número de bits útiles de code, desde el menos al más significativo
43  * @return true si la operación se realizó con éxito, false en otro caso
44  */
45 bool HuffmanEncoder(const ArduTree &tree, unsigned char ascii, unsigned short &code, unsigned short &nBits);
46

```

### Ejercicio (Cuaderno de prácticas): HuffmanTree. Parte II. Implementación

- Necesitaremos una función para inicializar el decodificador (es decir, indicar como nodo activo la raíz para poder decodificar).
- Llamaremos a esta función **InitHuffmanDecoder**.
- **Tiene como entrada:**
  - Un árbol que representa el código Huffman.
- **Tiene como salida:**
  - Un entero **decodificaPOS** que apunta a la raíz del árbol tras la ejecución de la función.

```

47
48 /**
49  * Inicializa el decodificador Huffman para decodificar un código
50  * @param tree Arbol a usar para la decodificación
51  * @param decodificaPOS Parámetro de salida Establece el nodo inicial para poder decodificar (la raíz).
52  * @post El decodificador Huffman queda inicializado para comenzar a decodificar
53  * un código Huffman. decodificaPOS apunta a la raíz del árbol.
54  */
55 void InitHuffmanDecoder(const ArduTree &tree, int &decodificaPOS)

```



- Una función más nos servirá para viajar por el árbol y decodificar símbolos. Llamaremos a esta función **HuffmanDecodeBit**.
- **Tiene como entrada:**
  - Un árbol que representa el código Huffman.
  - Un valor **bit** true/false indicando si se debe decodificar 1 o 0.
- **Tiene como salida:**
  - Un carácter **code**, si se decodifica el código llegando a un nodo hoja
  - Un entero **decodificaPOS** con el siguiente estado a visitar (si no es nodo hoja).
  - Devuelve true si se ha llegado a un nodo hoja. false en otro caso.

```

57 /**
58  * Inserta un nuevo bit en el decodificador para decodificar un código Huffman
59  * @param tree Arbol a usar para la decodificación
60  * @param bit Bit a decodificar (true valor 1; false valor 0)
61  * @param code Parámetro de salida. Código decodificado, si el decodificador
62  * ha dado la salida como true.
63  * @param decodificaPOS Parámetro de entrada y de salida. Contiene un entero
64  * apuntando al nodo actual del árbol sobre el que decodificar. Si la función
65  * devuelve false (no se ha decodificado aún), se actualiza al siguiente nodo del
66  * árbol a visitar en la decodificación.
67
68  * @return true si el bit decodificado de entrada, junto con el estado actual
69  * del decodificador, ha dado como resultado en la decodificación de un símbolo
70  * ASCII.
71  */
72 bool HuffmanDecodeBit(const ArduTree &tree, bool bit, unsigned char &code, int &decodificaPOS);

```

## **Ejercicio (Cuaderno de prácticas): HuffmanTree. Parte II. Implementación**

- El objetivo del proyecto **HuffmanTree** consiste en implementar la creación del árbol de codificación Huffman, un codificador y un decodificador del código, y realizar pruebas de su funcionamiento.
- Crear un programa para PC que considere como alfabeto los caracteres (en mayúsculas) de la frase *“Ata la jaca a la estaca”*, **junto con otro símbolo adicional de señalización de la finalización del mensaje.**
- Utilice la función **Huffman** para crear el árbol, y muéstrelo por pantalla. Compruebe que el árbol se corresponde con el que ha calculado a mano.
- Utilice la función **HuffmanEncoder** para codificar los símbolos **‘A’, ‘C’, ‘E’**. Muestre el código generado por pantalla para cada símbolo y compruebe que se corresponde con la codificación realizada a mano.
- Decodifique los códigos generados combinando el uso de las funciones **InitHuffmanDecoder** y **HuffmanDecodeBit**. Compruebe que la decodificación es la adecuada.

### Ejercicio (Cuaderno de prácticas): Proyecto ShareTree

- El objetivo del proyecto **ShareTree** consiste en implementar funciones para compartir un árbol de codificación Huffman entre PC y Arduino, desarrollando funciones para enviar y recibir un árbol de codificación Huffman por USB, y un programa de prueba para comprobar que las comunicaciones se efectúan correctamente.
- Deberemos modificar la biblioteca **<ticcomm.pc.h>** para incluir dos funciones que envíen y reciban, respectivamente, un árbol de Huffman por USB.
- La función **sendHuffmanUSB**:

```

51
52 /**
53  * Función que envía un árbol Huffman a Arduino por USB
54  * @param pd Descriptor del puerto USB de comunicaciones
55  * @param tree Árbol a enviar
56  */
57 void sendHuffmanUSB(int &pd, const ArduTree &tree);
58

```

## Ejercicio (Cuaderno de prácticas): Proyecto ShareTree

- La función **receiveHuffmanUSB**:

```
59
60 /**
61  * Función que recibe un árbol Huffman desde Arduino por USB
62  * @param pd Descriptor del puerto USB de comunicaciones
63  * @param tree Arbol que se recibe
64  */
65 void receiveHuffmanUSB(int &pd, ArduTree &tree);
66
```

- Ambas funciones deberán declararse en el fichero **ticcommpc.h** e implementarse en **ticcommpc.cpp**.
- Deberán utilizar las funciones básicas para envío y recepción de bytes por USB de la biblioteca de comunicaciones USB que se utiliza en las prácticas.

### Ejercicio (Cuaderno de prácticas): Proyecto ShareTree

- La parte de Arduino será similar: Deberemos implementar dos funciones para enviar y recibir un árbol de codificación Huffman por USB.
- Deberemos modificar la biblioteca **<ticcommardu.h>** para incluir dos funciones que envíen y reciban, respectivamente, un árbol de Huffman por USB.
- La función **sendHuffmanUSB**:

```

134
135 /**
136  * Función que envía un árbol Huffman a PC por USB
137  * @param pd Descriptor del puerto USB de comunicaciones
138  * @param tree Arbol a enviar
139  */
140 void sendHuffmanUSB(const ArduTree &tree);
141

```

### Ejercicio (Cuaderno de prácticas): Proyecto ShareTree

- La función **receiveHuffmanUSB** (de la biblioteca **<ticcommardu.h>**):

```

142
143 /**
144  * Función que recibe un árbol Huffman desde PC por USB
145  * @param pd Descriptor del puerto USB de comunicaciones
146  * @param tree Arbol que se recibe
147  */
148 void receiveHuffmanUSB(ArduTree &tree);
149

```

- Ambas funciones deberán declararse en el fichero **ticcommardu.h** e implementarse en **ticcommardu.cpp**.
- Deberán utilizar las funciones básicas para envío y recepción de bytes por USB de la biblioteca **UART** que se utiliza en las prácticas.

## Ejercicio (Cuaderno de prácticas): Proyecto ShareTree

- Tras implementar estas funciones, se deberá crear un proyecto software con 2 programas:
  - **ShareTreePC.** Debe generar el árbol de código Huffman relativo al mensaje “Ata la jaca a la estaca”, mostrarlo por pantalla, enviarlo por USB a Arduino y recibirlo por USB desde Arduino. Se mostrará este código recibido por pantalla y se comparará con el enviado, para comprobar que no hay errores en las comunicaciones entre los dos puntos.
  - **ShareTreeArdu.** Debe tener un bucle infinito que:
    - Reciba por USB un árbol de codificación Huffman desde el PC, y lo guarde en una variable.
    - Envíe de vuelta por USB el árbol de código Huffman almacenado en la variable.

### Ejercicio (Cuaderno de prácticas): Proyecto HuffmanProject

- El proyecto **HuffmanProject** recopila los avances realizados en todos los proyectos prácticos de la sesión, y los amplía para enviar y recibir por láser los códigos Huffman generados en un mensaje.
- El proyecto consta de 4 aplicaciones:
  - **emisorHuffmanPC**: Se encargará de leer el fichero con el fragmento de *El Quijote*, crear el árbol de Huffman, enviarlo al Arduino emisor y quedar a la espera de que el usuario introduzca mensajes por teclado. Estos mensajes se enviarán a Arduino emisor (**¡sólo si son válidos!**), el cual los codificará y enviará por láser.
  - **receptorHuffmanPC**: Se encargará de leer el fichero con el fragmento de *El Quijote*, crear el árbol de Huffman, enviarlo al Arduino receptor y quedar a la espera de que el Arduino receptor le envíe mensajes. Enviará estos mensajes para ser mostrados por consola, hasta que el usuario pulse **CRTL-C**. **En este caso, la aplicación finalizará liberando todos los recursos de memoria y de puertos abiertos reservados, de haberlos.**



## Ejercicio (Cuaderno de prácticas): Proyecto HuffmanProject

- El proyecto consta de 4 aplicaciones:
  - **emisorHuffmanArdu**: Se encargará de leer por USB el árbol de Huffman que le envía el emisor PC. Acto seguido, quedará a la espera de que el emisor PC le envíe tramas de mensajes (no superiores a 100 Bytes). Tras recibir por USB el mensaje a enviar:
    - Enviará por láser un BIT **LASER\_HIGH**, para indicar el comienzo de las comunicaciones.
    - Para cada símbolo del mensaje:
      - Lo codificará con el código de Huffman dado en el árbol.
      - Enviará los bits del código por láser.
      - Pasará a procesar el siguiente símbolo del mensaje.
    - Al finalizar, enviará el símbolo de señalización de fin de mensaje y volverá a quedar a la espera de una nueva trama de datos por USB.

## Ejercicio (Cuaderno de prácticas): Proyecto HuffmanProject

- El proyecto consta de 4 aplicaciones:
  - **receptorHuffmanArdu**: Se encargará de leer por USB el árbol de Huffman que le envía el emisor PC. Acto seguido, entrará en modo **“En espera”**. Cuando detecte una señal **LASER\_HIGH** por el fotorreceptor, leerá el bit de comienzo de transmisión de datos y pasará al estado **“Recibiendo”**.
  - Mientras esté en estado **“Recibiendo”**:
    - Leerá un bit recibido por láser.
    - Intentará decodificar el bit.
    - Si se consigue decodificar un símbolo, se añadirá a un buffer y se reinicializará el decodificador.
    - Si el símbolo era la señalización de finalización, se enviará el buffer por USB al receptor PC y se pasará al estado **“En espera”**.



Universidad de Granada

[decsai.ugr.es](http://decsai.ugr.es)

# **Teoría de la Información y la Codificación**

**Grado en Ingeniería Informática**

**Seminario 2.- Plataforma láser para envío y recepción de información con códigos Huffman.**



**DECSAI**

**Departamento de Ciencias de la  
Computación e Inteligencia Artificial**