

# Práctica 1

## Programación y Entornos de Desarrollo

### 1º DAW

#### Parte de programación

Entendemos que un mensaje es cualquier sucesión de 0's y 1's (por ejemplo: {1,1,1,1,0,1,1,1,0,1,0,0,0,0,1,0,1,0,0,1,1,0,0,1,0,1}).

Al “enviar” un mensaje, este puede sufrir modificaciones en la transmisión. Los *códigos de Hamming* permiten detectar si un mensaje transmitido ha sufrido hasta un máximo de 2 modificaciones en la transmisión. La práctica consistirá en simular la transmisión de un mensaje, que puede sufrir modificaciones, y el receptor sabrá decir si hay modificaciones en el mensaje.

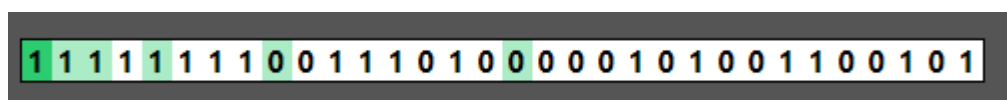
La práctica “tiene” 4 partes (si quieres, puedes hacer 4 clases que se dividan el trabajo, ni da más puntos ni es necesario).

1. Generar aleatoriamente un mensaje (sucesión de 0's y 1's) del tamaño indicado (puede estar dado por un input de usuario o como una constante)
2. Programar al “Sender” (quien envía el mensaje). El Sender recibe el mensaje, y tiene que construir un código de Hamming antes de “enviarlo”. Para esto hay que:
  - a. Calcular la cantidad mínima de bits de redundancia necesarios para crear el código de Hamming
  - b. Crear el código de Hamming, es decir, rellenar un nuevo array con el mensaje original y los nuevos bits de redundancia calculados
3. Programar el “Noise”: esta parte simula que el mensaje sufre modificaciones durante la transmisión. Para ello, el Noise recibe el código de Hamming que “envía” el Sender, y, de manera aleatoria, puede hacer 0, 1 o 2 modificaciones en el mensaje (cada una de estas opciones debe ser equiprobable). El mensaje modificado es el que “recibirá” la siguiente parte.
4. Programar el “Reciever”. El Reciever “recibe” el mensaje de Noise, y decide si ese mensaje:
  - a. Es igual al original (sin modificaciones)
  - b. Ha sufrido 1 modificación (y dice dónde [en qué posición] es la modificación)
  - c. Ha sufrido 2 modificaciones (aquí no pude decir dónde son)

#### Cómo crear un código de Hamming

Dado un mensaje, para crear un código de Hamming necesito añadir unos cuantos **bits de redundancia** que son los que luego permiten decidir si el mensaje a sufrido modificaciones. Sigamos un ejemplo visual con la web: <https://harryli0088.github.io/hamming-code/>.

El mensaje serían los bits en blanco:



Los bits en verde claro son los **bits de redundancia**, y el bit en verde oscuro es el **bit de redundancia global**. El bit de redundancia global siempre lo voy a poder calcular, así que realmente lo primero

que necesitamos saber es cuántos bits verde claro necesitamos. Obviamente, cuántos necesitemos dependerá del tamaño del mensaje a enviar, y la idea es no poner bits verde claro de más, solo los necesarios. Ojo, no tengas esta confusión al jugar con la web: el mensaje anterior y este:



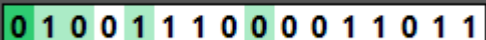
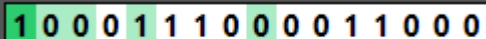
Tienen la misma longitud (hay el mismo número de bits blancos), lo que cambia es que se añade un nuevo bit verde claro que es solo de redundancia, no es de mensaje.

Para calcular el número *mínimo* de bits de redundancia que se necesitan (dada una longitud de mensaje), juega con la siguiente fórmula:  $2^n - (n + 1)$  [donde “n” es el número de bits de redundancia necesarios], y la longitud del mensaje. (Intuición de por qué esta fórmula: el mensaje tendrá una longitud L, y L + n + 1 [long. mensaje + bits de redundancia + bit de redundancia global], tiene que “caber” en una potencia de 2 [necesito tener suficientes dígitos binarios para poder cubrir todo]).

Una vez calculados cuántos bits de redundancia necesito, puedo crear el código de Hamming de la siguiente manera:

1. Será un array de tamaño longitud del mensaje original + número de bits de redundancia + bit de redundancia global
2. En la posición 0 de este nuevo array, se coloca el bit de redundancia global
3. En las posiciones  $2^n$  (potencias de 2: 1, 2, 4, 8...) van los bits de redundancia (los verde claro)
4. En el resto de posiciones que quedan libres, va el mensaje original (en el mismo orden que el original)

Los bits de redundancia (el global también) se calculan para conseguir un número par de 1’s en el código de Hamming. El global será un 1 si, en el resto del código de Hamming tengo en total de 1’s impar (así todo el resto + el global son un número total de 1’s par):



Los bit de redundancia (los verde claro) también se calculan igual, son 1 o 0 para conseguir un número total de 1’s par, pero los bits de redundancia solo miran ciertas posiciones:

0 0 0000	1 1 0001	2 0 0010	3 0 0011
4 1 0100	5 1 0101	6 1 0110	7 0 0111
8 0 1000	9 0 1001	10 0 1010	11 1 1011
12 1 1100	13 0 1101	14 1 1110	15 1 1111

0 0 0000	1 1 0001	2 0 0010	3 0 0011
4 1 0100	5 1 0101	6 1 0110	7 0 0111
8 0 1000	9 0 1001	10 0 1010	11 1 1011
12 1 1100	13 0 1101	14 1 1110	15 1 1111

Las posiciones que mira cada bit de redundancia tienen que ver con su representación en binario (en las imágenes, el número de abajo [ $2 = 0010$ ,  $3 = 0011$ ,  $6 = 0110$ ...]). En concreto, cada bit de redundancia (que está en la posición  $2^n$ ) mira a todos los bits de posiciones que tienen un 1 en la misma posición que el bit de redundancia. Es decir, el bit en posición 2 mira a los bits de posiciones 3, 6, 7, 10, 11, 14 y 15 (en el ejemplo de la imagen de arriba) porque, en binario:

2 = 0010

3 = 0011

6 = 0110

7 = 0111

14 = 1110

15 = 1111

Todas estas posiciones son exactamente las que tienen, en su representación binaria, un 1 en la misma posición que tiene 2 un 1 en su representación binaria (de hecho, las posiciones que no mira son justamente las que *no* tienen un 1 en la misma posición que 2 en representación binaria). Una vez que determinas, para cada bit de redundancia, qué posiciones debe mirar, el valor del bit de redundancia se calcula para que el número total de 1's sea par.

Pista: usa las bitwise operations (<https://www.programiz.com/java-programming/bitwise-operators>) (en concreto '&') para calcular qué posiciones son las que tiene que mirar cada bit de redundancia.

Con esto ya tienes el código de Hamming. Ahora, queda saber por qué, si el mensaje (de hecho, si el código de Hamming) sufre una modificación, soy capaz de detectarla, y si sufre 2, soy capaz de decir que hay 2 modificaciones.

Pues en realidad, muy fácil: si el código de Hamming ha sufrido exactamente una modificación, entonces el código recibido por el Reciever tiene que tener mal alguno de los bits de redundancia (puede ser el global). Ejemplo:

0 0 0000	1 1 0001	2 0 0010	3 0 0011
4 1 0100	5 1 0101	6 1 0110	7 0 0111
8 0 1000	9 1 1001	10 0 1010	11 1 1011
12 1 1100	13 1 1101	14 0 1110	15 1 1111

El bit de posición 10 ha cambiado, y en consecuencia, los bits de redundancia de posición 2 y 8 están mal (deberían ser = 1 si el mensaje es el correcto). Además, sabemos exactamente que es el bit de

posición 10 el que está mal porque  $2 + 8 = 10$ . Hay otra forma de calcular, si se detecta un fallo, cuál es la posición del fallo, que te cuentan aquí:

[https://www.youtube.com/watch?v=b3NxrZOu\\_CE&t=0s](https://www.youtube.com/watch?v=b3NxrZOu_CE&t=0s)

El único bit que no pillaríamos con este truco sería el de posición 0, pero es muy fácil saber si el bit de posición 0 ha llegado modificado al Reciever.

Por último, podemos saber que hay 2 fallos en el mensaje recibido (pero no dónde) si ocurre lo siguiente: se detecta que alguno de los bits de redundancia está mal, pero el bit de redundancia global está bien. Esto es porque, si hay exactamente un fallo, se ha cambiado en algún sitio un 0 por un 1 o al revés, por tanto el número total de 1's (que es lo que cuenta el bit de redundancia global) tiene que estar mal con respecto a lo que dice el bit de paridad global. Pero si hay otro fallo, el número de 1's vuelve a estar de acuerdo con lo que dice el bit de paridad global. Aquí hay un doble fallo:

0 0 0000	1 1 0001	2 0 0010	3 0 0011
4 1 0100	5 1 0101	6 1 0110	7 0 0111
8 0 1000	9 1 1001	10 0 1010	11 1 1011
12 1 1100	13 0 1101	14 0 1110	15 1 1111

Los bits de posiciones 1, 2 y 4 están mal, pero el bit de posición 0 está bien.

## Entregable

El entregable será un fichero llamado "nombre\_alumno\_practica\_1.java" que ejecuta el programa. No es necesario, pero agradecerá comentarios en el código que separen secciones:

```

// creación del mensaje
// código

// acciones del Sender
// código

// acciones del Noise
// código

// acciones del Reciever
// código

// método 1 que calcula tal
// código

// método 2 que hace cual
// código

```

Os recuerdo lo que pone en la programación:

Podrán realizarse diferentes prácticas diferenciadas del conjunto de ejercicios prácticos diarios a lo largo del trimestre. En caso de realizarse estas prácticas diferenciadas de los ejercicios de clase (es decir, prácticas desarrolladas por los alumnos de manera autónoma fuera del horario lectivo semanal) por evaluación, éstas tendrán un peso del **30%** del total de la nota. En caso contrario, este porcentaje se acumulará al de las pruebas teóricas.

La entrega de estas prácticas (caso de proponerse) será **obligatoria**, debiendo ser **presentadas y aprobadas** (nota igual o superior a cinco) en las fechas, formato y plazos que considere el titular de la materia. En caso contrario la evaluación estará suspensa.

En caso de detectarse copia en alguna práctica o prueba, éstas serán evaluadas con 0, todas ellas (práctica copiada si se ha facilitado el plagio (porque impide la correcta evaluación del aprendizaje) y copias (porque falsean los resultados académicos)). Así mismo, se tomarán las medidas oportunas, acorde con la normativa del Decreto 32/2019, de 9 de abril, del Consejo de Gobierno, por el que se establece el marco regulador de la convivencia en los centros docentes de la Comunidad de Madrid(BOCM de 15 de Abril de 2019).

El código desarrollado por los alumnos deberá ser legible y cumplir con unos mínimos de calidad para poder ser evaluado. El no cumplir con los siguientes mínimos de calidad supondrá una rebaja significativa de la calificación de la prueba/ejercicio/práctica:

- Convención de nombres para clases, constantes, atributos, variables, métodos y otros artefactos relacionados con la programación.
- Clases sin paquetizar
- Mala organización de las clases donde atributos y métodos estáticos y no estáticos estén entremezclados.

- Código duplicado
- Métodos excesivamente largos
- Clases excesivamente complejas o con una lógica de negocio muy difusa
- Exceso de acomplamiento
- Falta de cohesión en la definición de las clases
- Código que no se utiliza
- Exceso de comentarios que explican más el código que la lógica que desarrollan

Dicha calificación se verá afectada por la cantidad de faltas de ortografía, limpieza y orden en los documentos tal y como tiene establecido el Centro.

## Parte de Entornos de desarrollo

Aplicar los conocimientos adquiridos de control de versiones con GIT y GitHub y de refactorización a la vez que se desarrolla el proyecto (mirar los apuntes que os paso y también lo que encontréis por internet).

En particular, se pide que:

- El código quede bien refactorizado, con nombres significativos, métodos, recicle de código...
- Se use GIT y GitHub siguiendo las buenas prácticas (no desarrollar todo en main, commits que muestren como se va desarrollando el proyecto, usar ramas, tags, merges o rebases según tenga sentido, nombres significativos...)

**Nota:** dado que esta parte de entornos la he subido más tarde, seguro que muchos habéis hecho un commit muy grade con toda una parte del código ya desarrollada y no se ve cómo ha ido evolucionando el proyecto (ejemplo: te creas una rama “sender” para desarrollar el sender, y la historia de esa rama se vería algo así como commit1: consigo detectar las potencias de 2; commit2: consigo detectar qué valor tiene que tener una posición de redundancia; commit3: consigo formar el array del código de hamming ...). No pasa nada si queda algo confuso en los primeros commits, pero por lo menos en alguna parte se tiene que ver que podéis seguir buenas prácticas (es decir, si el código ya lo tienes funcional pero queda refactorizar, puedes mostrar que sabes seguir buenas prácticas al refactorizar).