2.1 La conversión de enteros de su representación binaria a decimal es sencilla, porque estamos muy familiarizados con las representaciones decimales de 2. Ideé (o recalque) un método sistemático para convertir de la representación decimal de un entero a su representación binaria. Cuál manera encuentras más conveniente: ¿determinar los bits de izquierda a derecha o de derecha a izquierda? Ambos métodos son aceptables, pero una vez que tienes la idea, uno de ellos es más fácil de usar sistemáticamente que el otro. Prueba tu elección en algunos ejemplos y convierte los resultados binarios de vuelta a decimal para corroborarlos. ¿Tu método se extiende para convertir una representación decimal finita de un número racional no entero (como 0.1) a su representación binaria?

La representación binaria de un número decimal corresponde con su expresión como suma de potencias de dos. Como ejemplo veamos que

$$71 = (1000111)_2 = 1 \cdot 64 + 0 \cdot 32 + 0 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 1 \cdot 2 + 1 \cdot 1$$
$$71 = (1000111)_2 = 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$$

Diremos que un número *ocupa* el i-ésimo bit si en su representación binaria aparece el término $1 \cdot 2^i$. En el caso anterior decimos que 71 ocupa los bits 0, 1, 2 y 6.

A continuación, se exponen dos métodos distintos para convertir un número de su representación decimal a su representación binaria.

- **Método de izquierda a derecha (solo enteros)**

Este método se basa en realizar consecutivamente divisiones enteras hasta que el cociente (que cambia a lo largo del procedimiento junto con el dividendo) sea cero.

Podemos escribir el algoritmo en un lenguaje de programación que soporte una aritmética entera con las operaciones de división y módulo.

- Definir `n` como el número a representar en binario
- Definir `q=1`, `r=0`
- Mientras `q` sea distinto de cero
  - `q=n/2`
  - `r=n%2`
  - `n=n/2`
  - Imprimir `r`
- Recordemos que la impresión de lee de derecha a izquierda

Una de las ventajas de este método es ser conciso con las operaciones realizadas y la obtención de los bits de derecha a izquierda, es decir, en potencias cada vez más grandes de 2 comenzando por la más pequeña. Esto permite ahorrar tiempo de ejecución.

- **Método de derecha a izquierda**

Este método se basa en realizar restas consecutivas de potencias de dos al número, siempre que la potencia sea mayor o igual al mismo. Si es posible realizar la resta entonces el bit correspondiente a dicha potencia se ocupa en la representación binaria, sino entonces no se ocupa el bit y no se altera el número. El proceso acaba cuando el número se vuelve cero. Si el número tiene parte fraccionaria entonces el algoritmo se extiende con potencias negativas de 2 hasta el bit deseado.

Igualmente, el algoritmo se puede implementar en algún lenguaje de programación con aritmética entera y flotante. En este caso resulta importante describir los límites ya que debemos "comenzar" con una potencia de dos suficientemente grande (al menos más que el número) y "detenernos" al llegar a un bit decimal concreto.

- Definir `n` como el número a representar en binario (puede ser decimal)
- Definir `bits_ent=63`, `bits_dec=10` (alterables según se pida)
- Definir `pot=2^bits_ent`, `lim_inf=2^-bits_dec`
- Definir `parte_ent=floor(n)` y `parte_dec=n-parte_ent`
- Mientras `parte_ent` es distinta de cero
    - Si `pot<=parte_ent`
        - `parte_ent=parte_ent-pot`
        - Imprimir `1`
    - Sino
        - Imprimir `0`
    - `pot=pot/2`
- Imprimir `.` (punto binario)
- Mientras `parte_dec` es distinta de cero y `pot` no sea `lim_inf`
    - Si `pot>=parte_dec`
        - `parte_dec=parte_dec-pot`
        - Imprimir `1`
    - Sino
        - Imprimir `0`
    - `pot=pot/2`
- Recordemos que la impresión se lee correctamente de izquierda a derecha

Las ventajas de este método es la obtención de la representación binaria de un número decimal fraccionario, a diferencia del método anterior que solo resulta ser útil con enteros tal y como lo hemos descrito. Además, resulta ser mas sencillo de comprender y ejecutar sin la necesidad de una computadora.

3.2 Suponga que deseamos almacenar enteros usando solo una cadena de 16 bits (2 bytes). Esto se llama formato de entero corto. ¿Cuál es el rango de los enteros que pueden ser almacenados usando el complemento a 2? Expresa la respuesta usando potencias de 2 y también tradúcela a números en notación decimal.

Si solo disponemos de una cadena de 16 bits para representar enteros a través del formato del complemento a dos, podemos ver fácilmente que el número más grande que podemos almacenar es aquel que ocupa los primeros 15 bits, es decir

$$(0111\ 1111\ 1111\ 1111)_2 = 2^{15} - 1 = 32{,}767$$

Es incorrecto pensar que el más grande es aquel que ocupa los 16 bits pues este número es el complemento a dos del número 1, es decir, se trata del -1.

El número más pequeño que podemos almacenar resulta ser aquel que solo ocupa el bit 16 con los 15 bits restantes en cero, es decir

$$(1000\ 0000\ 0000\ 0000)_2 = -2^{15} = -32{,}768$$

Es incorrecto pensar que el más pequeño es el complemento a dos del número más grande (que resulta ser $(1000\ 0000\ 0000\ 0001)_2 = -32{,}767$) pues aún podemos restarle 1 para obtener el resultado anterior, que, por cierto, no es el complemento a dos de ningún número positivo según estas condiciones.

Como conclusión, en el formato de entero corto podemos representar todo número entero x que verifique $-2^{15} \leq x \leq 2^{15} - 1$ o bien $-32768 \leq x \leq 32767$.

3.3 Usando un formato de 8 bits por simplicidad, de la representación del complemento a dos para los siguientes enteros: 1, 10, 100, -1, -10, -100. Verifique que la adición de un número negativo con su contraparte positiva es cero, como se requiere, cuando el bit overflow es descartado.

- Tenemos que $1 = (0000\ 0001)_2$, entonces su complemento a dos resulta ser $-1 = (1111\ 1111)_2$. Su suma resulta ser $(1\ 0000\ 0000)_2$ que es cero por el bit overflow.

$$\begin{array}{r} 0000\ 0001 \\ +\ 1111\ 1111 \\ \hline 1\ 0000\ 0000 \end{array}$$

- Para $10 = (0000\ 1010)_2$, tenemos que su complemento a dos es $-10 = (1111\ 0110)_2$, luego su suma es $(1\ 0000\ 0000)_2$ que es cero por el bit overflow.

$$\begin{array}{r} 0000\ 1010 \\ +\ 1111\ 0110 \\ \hline 1\ 0000\ 0000 \end{array}$$

- Para $100 = (0011\ 0100)_2$ tenemos que su complemento a dos es $-100 = (1100\ 1100)_2$, obteniendo una suma de $(1\ 0000\ 0000)_2$ que es cero por el bit overflow.

$$\begin{array}{r} 0011\ 0100 \\ +\ 1100\ 1100 \\ \hline 1\ 0000\ 0000 \end{array}$$

- Para los números -1, -10 y -100 tenemos que sus respectivos complementos a dos son los mencionados en cada inciso anterior y evidentemente podemos ver que la suma con sus contrapartes positivas resulta en $(1\ 0000\ 0000)_2$, que resulta ser cero por el bit overflow.

3.4 Demuestra que si un entero x entre -2^{31} y $2^{31}-1$ es representado usando el complemento a dos en una cadena de 32 bits, el bit mas a la izquierda es 1 si x es negativo y es 0 si x es positivo o cero.

- Sea x entero tal que $-(2^{31}-1) \leq x < 0$. Entonces existe $-x$ tal que $0 < -x \leq 2^{31}-1$ que tiene una representación binaria que tiene a lo más 31 bits. Esto implica que necesariamente el complemento a dos de $-x$ resulta tener una representación binaria de a lo más 31 bits, en particular con el bit mas a la izquierda siendo el más significativo. Pero el complemento a dos de $-x$ resulta ser x. Por lo tanto, el bit más significativo de x resulta ser el ubicado más a la izquierda.
  El caso en que $x = -2^{31}$ es trivial pues $x = (1000\ 0000\ 0000\ 0000)_2$.

- Sea x entero tal que $0 \leq x < 2^{31}-1$. Es claro que la representación binaria de x tiene a lo más 31 bits, ya que si tuviera más de 31 bits implicaría que $x \geq 2^{31}$, lo cual contradice la hipótesis. Por lo tanto, dado que tenemos una cadena de 32 bits, el bit más a la izquierda resulta ser cero.

3.6 Muestre los detalles para las sumas enteras $50 + (-100)$, $100 + (-50)$ y $50 + 50$, usando un formato de 8 bits.

- Para $50 = (0001\ 1010)_2$ y $-100 = (1100\ 1100)_2$ tenemos que

$$\begin{array}{r} 0000\ 1010 \\ +\ 1111\ 0110 \\ \hline 1110\ 0110 \end{array}$$

Lo que resulta consistente pues obtenemos el complemento a dos de 50, es decir $-50 = (1110\ 0110)_2$. Concluimos que la resta de un número más grande que otro es consistente según el formato del complemento a dos.

- Para $100 = (0011\ 0100)_2$ y del inciso anterior $-50 = (1110\ 0110)_2$ tenemos que

$$\begin{array}{r} 0011\ 0100 \\ +\ 1110\ 0110 \\ \hline 0001\ 1010 \end{array}$$

Lo que resulta consistente pues $50 = (0001\ 1010)_2$. La resta de un número más pequeño que otro es consistente.

- Con $50 = (0001\ 1010)_2$ veamos que

$$\begin{array}{r} 0001\ 1010 \\ +\ 0001\ 1010 \\ \hline 0011\ 0100 \end{array}$$

Obteniendo $100 = (0011\ 0100)_2$. La suma de un número consigo mismo se mantiene en el complemento a dos.

3.7 *(D. Goldberg)* Además de la división entre cero, ¿hay alguna otra operación de división que pueda resultar en un overflow entero?

Sí.

Consideremos el número $n = -2^{31}$, es decir, el entero más pequeño que podemos tener con nuestra aritmética de 32 bits.

Si efectuamos la división entera $n/-1 = -2^{31}/-1$ obtendríamos el resultado 2^{31}, sin embargo, esto no lo podemos representar como un entero pues el entero más grande es $2^{31} - 1$.

Por lo tanto, **la división $\frac{-2^{31}}{-1}$ ocasiona overflow**.

3.8 ¿Cuál es el número de punto flotante más grande en este sistema, asumiendo que el campo significando puede almacenar solamente los bits $b_1, \cdots, b_{23}$ y el exponente está limitado por $-128 \leq E < 127$?. No olvides que el bit escondido b_0 es 1.

El número de punto flotante mas grande para este sistema (que de hecho es el de *simple precisión*) es aquel que tiene **signo positivo**, su **mantisa con todos los bits encendidos** y su **exponente máximo**, esquemáticamente este número es:

0	ebits(127)	11111111111111111111111

Es decir, tenemos 23 bits en toda la mantisa, sin embargo, recordemos que el **bit escondido**, aunque no está representado, está a la izquierda del punto binario. Dicho esto, podemos representar a este número como:

$$1.11111111111111111111111 \times 2^{127}$$

El exponente 127 nos indica mover el punto binario un total de **127 posiciones a la derecha**, resultando en un número muy grande con muchos ceros a la derecha (pues solo tenemos 23 bits 1 que podemos recorrer), visto esquemáticamente este número es:

$1 \cdots 1$	$0 \cdots 0$
24 bits 1	104 bits 0

Finalmente concluimos que, en este número de 128 bits, dado que solo tiene algunos bits 1, **tiene como valor la suma de las potencias que aportan dichos bits**. Recordemos que los bits se enumeran a partir de cero y la potencia representada por el bit b es 2^b, por lo tanto su valor es

$$\sum_{b=104}^{127} 2^b = 2^{127} + \cdots + 2^{104} \approx 3.4028235 \times 10^{38}$$

3.10 ¿Cuál es el entero positivo más pequeño que no es exactamente representado como un número de punto flotante en este sistema?

Consideremos que un número binario en formato $(1.b_1 \cdots b_{23}b_{24} \cdots) \times 2^E$. Al momento de guardarlo como un número de punto flotante, como solo tenemos 23 bits en la mantisa (ya que no se guarda el bit escondido) se toman los primeros 23 y se redondea el bit 24.

Dicho esto, consideremos el número

$$(1.00000000000000000000001) \times 2^{24} = 2^{24} + 2^0 = 16{,}777{,}217$$

Este número en particular tiene **23 bits ceros** en su representación y **un par de bits 1**, en las posiciones 0 y 24 respectivamente. Veamos que al convertirlo a un número de punto flotante tenemos, por el efecto del redondeo:

$$\boldsymbol{fl}\left((1.00000000000000000000001) \times 2^{24}\right) = (1.00000000000000000000001) \times 2^{24}$$

Pero esto implica que el número ha sido redondeado, de hecho, **ahora tenemos un número distinto al original** ya que éste tiene **22 bits ceros** en su representación y **un par de bits 1**, en las posiciones 0 y 24 respectivamente, dando un valor decimal de:

$$(1.00000000000000000000001) \times 2^{24} = 2^{24} + 2^1 = 16{,}777{,}218$$

Por lo tanto, el número entero positivo más pequeño que **no es exactamente representado** como un número de punto flotante en este sistema es $2^{24} + 1 = 16{,}777{,}217$.

Cabe recalcar que el número anterior $2^{24} = 16{,}777{,}216$ sí está bien representado como número de punto flotante, así como todos los anteriores por solo tener una representación decimal de a lo más 23 bits en la mantisa.

Esto nos dice algo muy interesante, **a partir de este número entero los siguientes van a ser redondeados a algún múltiplo de 2, 4, etc.** Esto se debe al tamaño limitado de la mantisa y al redondeo que propiamente hace para almacenar en un punto flotante. Es por esta razón que, aunque parezca que la aritmética de punto flotante es exacta cuando se trata de enteros, en realidad hay errores numéricos a partir de enteros lo suficientemente grandes, teniendo una mayor separación entre ellos conforme crece su valor.

3.11 Suponga que cambiamos el formato de la mantisa de tal manera que sus cotas ahora son $\frac{1}{2} \leq S < 1$. Entonces cambiamos la expansión binaria a la forma

$$S = (0.\, b_1 b_2 b_3 b_4 \cdots)_2, \qquad \text{con} \quad b_1 = 1$$

y cambiamos nuestro sistema de punto flotante a uno tal que la mantisa almacena únicamente los bits $b_2 \cdots b_{24}$, con el exponente limitado por $-128 \leq E \leq 127$ como antes. ¿Cuál es el mayor número de punto flotante en este sistema? ¿Cuál es el menor número de punto flotante en este sistema (recordando que el número debe estar normalizado con el bit escondido $b_1 = 1$)? ¿Cuál es el menor número entero positivo que no es representado exactamente como un número de punto flotante en este sistema?

Dado que ahora el bit escondido corresponde al bit $1/2$, esto implica que el formato esquemático de nuestros números flotantes ahora es

signo	ebits(E)	$b_2 b_3 \cdots b_{23} b_{24}$

A modo de ilustración tenemos, algunos números en punto flotante en este sistema son:

$$(1.00000000000000000000000) \times 2^0 = 0.5$$
$$(1.00000000000000000000001) \times 2^0 \approx 0.500000059605$$
$$(1.11111111111111111111111) \times 2^0 \approx 0.999999940395$$
$$(1.00000000000000000000000) \times 2^1 = 1$$
$$(1.00000000000000000000001) \times 2^1 \approx 1.00000011921$$
$$(1.00000000000000000000010) \times 2^1 \approx 1.00000013842$$

- El número más grande que se puede representar en este sistema es el que tiene solo bits 1 en toda su mantisa, tiene signo positivo y está elevado al exponente más grande posible, es decir el número

0	ebits(127)	11111111111111111111111

Cuyo valor, considerando el bit escondido con valor de $1/2$, resulta ser:

$$(0.\,\mathbf{1}11111111111111111111111)_2 \times 2^{127} = \sum_{b=103}^{126} 2^b \approx 1.7014117 \times 10^{38}$$

- El número más pequeño que se puede representar es el que tiene solo bits 1 en su mantisa, tiene signo negativo y está elevado al exponente más grande, es decir el número con representación y valor dado por:

1	ebits(127)	11111111111111111111111

$$-(0.\,\mathbf{1}11111111111111111111111)_2 \times 2^{127} = -\sum_{b=103}^{126} 2^b \approx -1.7014117 \times 10^{38}$$

- El número entero positivo más pequeño que no se puede representar exactamente como un número de punto flotante en este sistema resulta ser

$$(1.0000\ 0000\ 0000\ 0000\ 0000\ 0001)_2 \times 2^{24} = 2^{24} + 1 = 16{,}777{,}217$$

Ya que su representación binaria también es $(0.1\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001)_2 \times 2^{25}$, la cual tiene un total de 25 bits a la derecha del punto binario. Recordemos que solo tenemos una mantisa de 23 bits (el bit escondido no se almacena, pero sabemos que está allí), por lo que el último se redondeará:

$$\begin{aligned} \boldsymbol{fl}(16{,}777{,}217) &= \boldsymbol{fl}((0.1\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001)_2 \times 2^{25}) \\ &= (1.0000\ 0000\ 0000\ 0000\ 0000\ 001) \times 2^{25} \end{aligned}$$

O bien, esquemáticamente como

1	ebits(25)	0000 0000 0000 0000 0000 001

Si convertimos este resultado a decimal veremos que en realidad se almacenó el número 16,777,218 ya que tiene el valor

$$\begin{aligned} &(0.\mathbf{1}\ 0000\ 0000\ 0000\ 0000\ 0000\ 001)_2 \times 2^{25} \\ &= 1\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010 = 2^{24} + 2 = 16{,}777{,}218 \end{aligned}$$

Por lo tanto, el número entero positivo más pequeño que **no se puede almacenar con exactitud en este sistema de punto flotante es 16, 777, 217.**