Representación de la información

31 de marzo de 2017

Índice

Representación de datos numéricos

Veremos de qué manera puede ser tratada mediante computadoras la información correspondiente a números, textos, imágenes y otros datos. Necesitaremos conocer las formas de representación de datos, y comenzaremos por los datos numéricos.

Representación de datos numéricos

Hemos visto ejemplos de sistemas de numeración: en base 6, en base 10, o decimal, en base 2, o binario, en base 16, o hexadecimal, y en base 8, u octal; y sabemos convertir la representación de un número en cada una de estas bases, a los sistemas en las demás bases. Sin embargo, aún nos falta considerar la representación numérica de varios casos importantes:

- Hemos utilizado estos sistemas para representar únicamente números **enteros**. Nos falta ver de qué manera representar números racionales, es decir aquellos que tienen una parte fraccionaria (los "decimales").
- Además estos enteros han sido siempre **no negativos**, es decir, sabemos representar únicamente el 0 y los naturales. Nos falta considerar los negativos.
- Por otra parte, no nos hemos planteado el problema de la **cantidad de dígitos**. Idealmente, un sistema de numeración puede usar infinitos dígitos para representar números arbitrariamente grandes. Si bien esto es matemáticamente correcto, las computadoras son objetos físicos que tienen unas ciertas limitaciones, y con ellas no es posible representar números de infinita cantidad de dígitos.

En esta parte de la unidad mostraremos sistemas de representación utilizados en computación que permiten tratar estos problemas.

Clasificación de los números

Es conveniente repasar la clasificación de los diferentes conjuntos de números y conocer las diferencias importantes entre éstos. Los títulos en el cuadro (tomado de Wikipedia) son referencias a los artículos enciclopédicos sobre cada uno de esos conjuntos.

- Números complejos
- Complejos
- Reales
- Racionales
- Enteros
- Naturales
- Uno
- Naturales primos
- Naturales compuestos
- Cero
- Enteros negativos
- Fraccionarios
- Fracción propia
- Fracción impropia
- Irracionales
- Irracionales algebraicos
- Trascendentes
- Imaginarios

Preguntas

- El **cero**, ¿es un natural?
- ¿Existen números naturales negativos? ¿Y racionales negativos?
- ¿Es correcto decir que un racional tiene una parte decimal que es, o bien finita, o bien periódica?
- ¿Puede haber dos expresiones diferentes para el mismo número, en el mismo sistema de numeración decimal?
- El número $0.9999\ldots$ con 9 periódico, y el número 1, ¿son dos números diferentes o el mismo número? Si son diferentes, ¿qué número se encuentra entre ellos dos?
- El número 1 es a la vez natural y entero. ¿Por qué no puede haber un número que sea a la vez racional e irracional?
- ¿Por qué jamás podremos computar la sucesión completa de decimales de π?

Datos enteros

Veremos un sistema de representación de datos no negativos, llamado **sin signo**, y tres sistemas de representación de datos numéricos enteros, llamados **signo-magnitud**, **complemento a 2** y **notación en exceso**.

Datos fraccionarios

Para representar fraccionarios consideraremos los sistemas de **punto fijo** y **punto flotante**.

Rango de representación

Cada sistema de representación de datos numéricos tiene su propio **rango de representación** (que podemos abreviar **RR**), o intervalo de números representables. Ningún número fuera de este rango puede ser representado en dicho sistema. Conocer este intervalo es importante para saber con qué limitaciones puede enfrentarse un programa que utilice alguno de esos sistemas.

El rango de los números representados bajo un sistema está dado por sus **límites inferior y superior**, que definen qué zona de la recta numérica puede ser representada. Como ocurre con todo intervalo numérico cerrado, el rango de representación puede ser escrito como $[a, b]$, donde a y b son sus límites inferior y superior, respectivamente.

Por la forma en que están diseñados, algunos sistemas de representación sólo pueden representar números muy pequeños, o sólo positivos, o tanto negativos como positivos. En general, el RR **será más grande cuantos más dígitos binarios**, o bits, tenga el sistema. Sin embargo, el RR depende también de la forma como el sistema **utilice** esos dígitos binarios, ya que un sistema puede ser más o menos **eficiente** que otro en el uso de esos dígitos, aunque la cantidad de dígitos sea la misma en ambos sistemas.

Por lo tanto, decimos que el rango de representación depende a la vez de la **cantidad de dígitos** y de la **forma de funcionamiento** del sistema de representación.

Representación sin signo SS(k)

Consideremos primero qué ocurre cuando queremos representar números enteros **no negativos** (es decir, **positivos o cero**) sobre una cantidad fija de bits.

En el sistema **sin signo**, simplemente usamos el sistema binario de numeración, tal como lo conocemos, **pero limitándonos a una cantidad fija** de dígitos binarios o bits. Podemos entonces abreviar el nombre de este sistema como **SS(k)**, donde k es la cantidad fija de bits, o ancho, de cada número representado.

Rango de representación de SS(k)

¿Cuál será el rango de representación? El **cero** puede representarse, así que el límite inferior del rango de representación será 0. Pero ¿cuál será el límite superior? Es decir, si la cantidad de dígitos binarios en este sistema es k, ¿cuál es el número más grande que podremos representar?

Podemos estudiarlo de dos maneras.

1. Usando combinatoria

Contemos cuántos números diferentes podemos escribir con k dígitos binarios. Imaginemos un número binario cualquiera con k dígitos. El dígito de más a la derecha tiene únicamente dos posibilidades (0 o 1). Por cada una de éstas hay nuevamente dos posibilidades para el siguiente hacia la izquierda (lo que da las cuatro posibilidades 00, 01, 10, 11). Por cada una de éstas, hay dos posibilidades para el siguiente (dando las ocho posibilidades 000, 001, 010, 011, 100, 101, 110, 111), etc., y así hasta la posición k. No hay más posibilidades. Como hemos multiplicado 2 por sí mismo k veces, la cantidad de números que se pueden escribir es 2^k. Luego, el número más grande posible es $2^k - 1$. (**Pregunta**: ¿Por qué $2^k - 1$ y no 2^k?).

2. Usando álgebra

El número más grande que podemos representar en un sistema sin signo a k dígitos es, seguramente, aquel donde todos los k dígitos valen **1**. La Expresión General que hemos visto nos dice que si un número n está escrito en base 2, **con k dígitos**, entonces

$$n = x_{k-1} \times 2^{k-1} + ... + x_1 \times 2^1 + x_0 \times 2^0$$

y, si queremos escribir el más grande de todos, deberán ser todos los x_i iguales a 1. (**Pregunta**: ¿Por qué si el número n tiene k dígitos binarios, el índice del más significativo es $k - 1$ y no k?)

Esta suma vale entonces

$$\begin{aligned} x_{k-1} \times 2^{k-1} + ... + x_1 \times 2^1 + x_0 \times 2^0 = \\ = 1 \times 2^{k-1} + ... + 1 \times 2^1 + 1 \times 2^0 = \\ = 2^{k-1} + ... + 2^1 + 2^0 = \\ = 2^k - 1 \end{aligned}$$

Usando ambos argumentos hemos llegado a que el número más grande que podemos representar con k dígitos binarios es $2^k - 1$. Por lo tanto, **el rango de representación de un sistema sin signo a k dígitos, o SS(k), es** $[0, 2^k - 1]$. Todos los números representables en esta clase de sistemas son **positivos o cero**.

Ejemplos

- Para un sistema de representación sin signo a 8 bits: $[0, 2^8 - 1] = [0, 255]$
- Con 16 bits: $[0, 2^{16} - 1] = [0, 65{,}535]$
- Con 32 bits: $[0, 2^{32} - 1] = [0, 4{,}294{,}967{,}295]$

Representación con signo

En la vida diaria manejamos continuamente números negativos, y los distinguimos de los positivos simplemente agregando un signo "menos". Representar esos datos en la memoria de la computadora no es tan directo, porque, como hemos visto, la memoria **solamente puede alojar ceros y unos**. Es decir, ¡no podemos simplemente guardar un signo "menos"! Lo único que podemos hacer es almacenar secuencias de ceros y unos.

Esto no era un problema cuando los números eran no negativos. Para poder representar, ahora, tanto números **positivos como negativos**, necesitamos cambiar la forma de representación. Esto quiere decir que una secuencia particular de dígitos binarios, que en un sistema sin signo tiene un cierto significado, ahora tendrá un significado diferente. Algunas secuencias, que antes representaban números positivos, ahora representarán negativos.

Veremos los **sistemas de representación con signo** llamados **Signo-magnitud (SM)**, **Complemento a 2 (C2)** y **Notación en exceso**.

Es importante tener en cuenta que **solamente se puede operar entre datos representados con el mismo sistema de representación**, y que el **resultado** de toda operación **vuelve a estar representado en el mismo sistema**.

Preguntas

- ¿Cuáles son los límites del rango de representación de un sistema de representación numérica?
- Un número escrito en un sistema de representación **con signo**, ¿es siempre negativo?
- ¿Para qué querríamos escribir un número positivo en un sistema de representación con signo?

Sistema de Signo-magnitud SM(k)

El sistema de **Signo-magnitud** no es el más utilizado en la práctica, pero es el más sencillo de comprender. Se trata simplemente de utilizar un bit (el de más a la izquierda) para representar el **signo**. Si este bit tiene valor 0, el número representado es positivo; si es 1, es negativo. Los demás bits se utilizan para representar la **magnitud**, es decir, el **valor absoluto** del número en cuestión.

Ejemplos

- $7_{(10} = 00000111_{(2}$
- $-7_{(10} = 10000111_{(2}$

Como estamos reservando un bit para expresar el signo, ese bit ya no se puede usar para representar magnitud; y como el sistema tiene una cantidad de bits fija, el RR ya no podrá representar el número máximo que era posible con el sistema **sin signo**.

Rango de representación de SM(k)

- En todo número escrito en el sistema de signo-magnitud a k bits, ya sea positivo o negativo, hay un bit reservado para el signo, lo que implica que quedan $k-1$ bits para representar su valor absoluto.
- Siendo un valor absoluto, estos $k-1$ bits representan un número **no negativo**. Además este número está representado con el sistema **sin signo** sobre $k-1$ bits, es decir, SS(k-1).
- Este número no negativo en SS(k-1) tendrá un valor máximo representable que coincide con el **límite superior** del rango de representación **de SS(k-1)**.
- Sabemos que el rango de representación de SS(k) es $[0, 2^k - 1]$. Por lo tanto, el rango de SS(k-1), reemplazando, será $[0, 2^{k-1} - 1]$.
- Esto quiere decir que el número representable en SM(k) cuyo valor absoluto es máximo, es $2^{k-1} - 1$. Por lo tanto éste es el límite superior del rango de representación de SM(k).
- Pero en SM(k) también se puede representar su opuesto negativo, simplemente cambiando el bit más alto por 1. El opuesto del máximo positivo representable es a su vez el número más pequeño, negativo, representable: $-(2^{k-1} - 1)$.

Con lo cual hemos calculado tanto el límite inferior como el superior del rango de representación de SM(k), que, finalmente, es $[-(2^{k-1} - 1), 2^{k-1} - 1]$.

Limitaciones de Signo-Magnitud

Si bien **SM(k)** es simple, no es tan efectivo, por varias razones:

- Existen dos representaciones del 0 ("positiva" y "negativa"), lo cual desperdicia un representante.
- Esto acorta el rango de representación.
- La aritmética en SM no es fácil, ya que cada operación debe comenzar por averiguar si los operandos son positivos o negativos, operar con los valores absolutos y ajustar el resultado de acuerdo al signo reconocido anteriormente.
- El problema aritmético se agrava con la existencia de las dos representaciones del cero: cada vez que un programa quisiera comparar un valor resultado de un cómputo con 0, debería hacer **dos** comparaciones.

Sistema de Complemento a 2

Por estos motivos, el sistema de SM dejó de usarse y se diseñó un sistema que eliminó estos problemas, el sistema de **complemento a 2**.

Para comprender el sistema de complemento a 2 es necesario primero conocer la **operación** de complementar a 2.

Operación de Complemento a 2

La **operación** de complementar a 2 consiste aritméticamente en obtener el **opuesto** de un número (el que tiene el mismo valor absoluto pero signo opuesto).

Para obtener el complemento a 2 de un número escrito en base 2, **se invierte cada uno de los bits (reemplazando 0 por 1 y viceversa) y al resultado se le suma 1**.

Otra forma

Otro modo de calcular el complemento a 2 de un número en base 2 es **copiar los bits, desde la derecha, hasta el primer 1 inclusive; e invertir todos los demás a la izquierda**.

Propiedad fundamental

El resultado de esta operación, C2(a), es el opuesto del número original a, y por lo tanto tiene la propiedad de que a y C2(a) suman 0:

$$C2(a) + a = 0$$

Comprobación

Podemos comprobar si la complementación fue bien hecha aplicando la **propiedad fundamental** del complemento. Si, al sumar nuestro resultado con el número original, no obtenemos 0, corresponde revisar la operación.

Ejemplos

- Busquemos el complemento a 2 de 111010. Invirtiendo todos los bits, obtenemos 000101. Sumando 1, queda 000110.
- Busquemos el complemento a 2 de 0011. Invirtiendo todos los bits, obtenemos 1100. Sumando 1, queda 1101.
- Comprobemos que el resultado obtenido en el último caso, 1101, es efectivamente el opuesto de 0011: $0011 + 1101 = 0$.

Representación en Complemento a 2

Ahora que contamos con la **operación de complementar a 2**, podemos ver cómo se construye el **sistema de representación en Complemento a 2**.

Para representar un número a en complemento a 2 a k bits, comenzamos por considerar su signo:

- Si a es positivo o cero, lo representamos como en SM(k), es decir, lo escribimos en base 2 a k bits.
- Si a es negativo, tomamos su valor absoluto y lo complementamos a 2.

Ejemplos

- Representemos el número 17 en complemento a 2 con 8 bits. Como es positivo, lo escribimos en base 2, obteniendo 00010001, que es 17 en notación complemento a 2 con 8 bits.
- Representemos el número -17 en complemento a 2 con 8 bits. Como es negativo, escribimos su valor absoluto en base 2, que es 00010001, y lo complementamos a 2. El resultado final es 11101111 que es -17 en notación complemento a 2 con 8 bits.

Conversión de C2 a base 10

Para convertir un número n, escrito en el sistema de complemento a 2, a decimal, lo primero es determinar el signo. Si el bit más alto es 1, n es negativo. En otro caso, n es positivo. Utilizaremos esta información enseguida.

- Si n es positivo, se interpreta el número como en el sistema sin signo, es decir, se utiliza la Expresión General para hacer la conversión de base como normalmente.
- Si n es negativo, se lo complementa a 2, obteniendo el opuesto de n. Este número, que ahora es positivo, se convierte a base 10 como en el caso anterior; y finalmente se le agrega el signo "-" para reflejar el hecho de que es negativo.

Ejemplos

- Convertir a decimal $n = 00010001$. Es positivo, luego, aplicamos la Expresión General dando $17_{(10}$.
- Convertir a decimal $n = 11101111$. Es negativo; luego, lo complementamos a 2 obteniendo 00010001. Aplicamos la Expresión General obteniendo $17_{(10}$. Como n era negativo, agregamos el signo menos y obtenemos el resultado final $-17_{(10}$.

RR de C2 con k bits

La forma de utilizar los bits en el sistema de complemento a 2 permite recuperar un representante que estaba desperdiciado en SM.

El rango de representación del sistema complemento a 2 sobre k bits es $[-(2^{k-1}), 2^{k-1} - 1]$. El límite superior del RR de C2 es el mismo que el de SM, pero el **límite inferior** es menor; luego el RR de C2 es mayor que el de SM.

El sistema de complemento a 2 tiene otras ventajas sobre SM:

- El cero tiene una única representación, lo que facilita las comparaciones.

- Las cuentas se hacen bit a bit, en lugar de requerir comprobaciones de signo.
- El mecanismo de cálculo es eficiente y fácil de implementar en hardware.
- Solamente se requiere diseñar un algoritmo para **sumar**, no uno para sumar y otro para restar.

Comparando rangos de representación

Diferentes sistemas, entonces, tienen diferentes rangos de representación. Si construimos un cuadro donde podamos comparar los rangos de representación **sin signo, signo-magnitud y complemento a 2** para una misma cantidad de bits, veremos que todas las combinaciones de bits están utilizadas, sólo que de diferente forma.

El cuadro comparativo para cuatro bits mostrará que las combinaciones 0000...1111 representan los primeros 16 números no negativos para el sistema sin signo, mientras que esas mismas combinaciones tienen otro significado en los sistemas con signo. En éstos últimos, una misma combinación con el bit más significativo en 1 siempre es negativa, pero el orden en que aparecen esas combinaciones es diferente entre SM y C2.

Por otro lado, los números positivos quedan representados por combinaciones idénticas en los tres sistemas, hasta donde lo permite el rango de representación de cada uno.

Si descartamos el bit de signo y consideramos sólo las magnitudes, los números negativos en SM aparecen con sus magnitudes crecientes alejándose del 0, mientras que en C2 esas magnitudes comienzan en cero al representar el negativo más pequeño posible y crecen a medida que se acercan al cero.

Aritmética en C2

Una gran ventaja que aporta el sistema en Complemento a 2 es que los diseñadores de hardware no necesitan implementar algoritmos de resta además de los de la suma. Cuando se necesita efectuar una resta, **se complementa el sustraendo** y luego se lo **suma** al minuendo. Las computadoras no restan: siempre suman.

Por ejemplo, la operación $9-8$ se realiza como $9+(-8)$, donde (-8) es el complemento a 2 de 8.

Preguntas

- Un número en complemento a 2, ¿tiene siempre su bit más a la izquierda en 1?
- El complemento a 2 de un número, es decir, **C2(x)**, ¿es siempre un número negativo?
- ¿Quién es **C2(0)**?

- ¿Cuánto vale **C2(C2(x))**? Es decir, ¿qué pasa si complemento a 2 el complemento a 2 de x?
- ¿Cuánto vale **x + C2(x)**? Es decir, ¿qué pasa si sumo a x su propio complemento a 2?
- ¿Cómo puedo verificar si calculé correctamente un complemento a 2?

Overflow o desbordamiento en C2

En todo sistema de ancho fijo, la suma de **dos números positivos, o de dos números negativos** puede dar un resultado que sea imposible de representar debido a las limitaciones del rango de representación. Este problema se conoce como desbordamiento, u *overflow*. Cuando ocurre una situación de overflow, el resultado de la operación **no es válido** y debe ser descartado.

Si conocemos los valores en decimal de dos números que queremos sumar, usando nuestro conocimiento del rango de representación del sistema podemos saber si el resultado quedará dentro de ese rango, y así sabemos, de antemano, si ese resultado será válido. Pero las computadoras no tienen forma de conocer a priori esta condición, ya que todo lo que tienen es la representación en C2 de ambos números. Por eso necesitan alguna forma de detectar las situaciones de overflow, y el modo más fácil para ellas es comprobar los dos últimos bits de la fila de bits de acarreo o *carry*.

El último bit de la fila de carry, el que se posiciona en la última de las k columnas de la representación, se llama *carry-in*. El siguiente bit de carry, el que ya no puede acarrearse sobre ningún dígito válido porque se han rebasado los k dígitos de la representación, se llama el *carry-out*.

- Si, luego de efectuar una suma en C2, los valores de los bits de *carry-in* y *carry-out* son **iguales**, entonces la computadora detecta que el resultado no ha desbordado y que **la suma es válida**. La operación de suma se ha efectuado exitosamente.
- Si, luego de efectuar una suma en C2, los valores de los bits de *carry-in* y *carry-out* son **diferentes**, entonces la computadora detecta que el resultado ha desbordado y que **la suma no es válida**. La operación de suma no se ha llevado a cabo exitosamente, y el resultado debe ser descartado.

Suma sin overflow

Siguiendo atentamente la secuencia de bits de carry podemos detectar, igual que lo hace la computadora, si se producirá un desbordamiento. En el caso de la operación $23 + (-9)$, el resultado (que es 14) cae dentro del rango de representación, y esto se refleja en los bits de *carry-in* y de *carry-out*, cuyos valores son iguales.

Suma con overflow

En el caso de la operación $123 + 9$ en C2 a 8 bits, el resultado (que es 132) cae fuera del rango de representación. Esto se refleja en los bits de *carry-in* y de *carry-out*, que son diferentes. El resultado no es válido y debe ser descartado.

Preguntas - ¿Qué condición sobre los bits de carry permite asegurar que **no habrá** overflow? - ¿Para qué sistemas de representación numérica usamos la condición de detección de overflow? - ¿Puede existir overflow al sumar dos números de diferente signo? - ¿Qué condición sobre los bits **de signo** de los operandos permite asegurar que **no habrá** overflow? - ¿Puede haber casos de overflow al sumar dos números negativos? - ¿Puede haber casos de overflow al restar dos números?

Extensión de signo en C2

Para poder efectuar una suma de dos números, ambos operandos deben estar representados en el mismo sistema de representación.

- Una suma de dos operandos donde uno esté, por ejemplo, en SM y el otro en C2, no tiene sentido aritmético.
- Además, la cantidad de bits de representación debe ser la misma.

En una suma en C2, si uno de los operandos estuviera expresado en un sistema con menos bits que el otro, será necesario convertirlo al sistema del otro (**extenderlo**) y operar con ambos en ese sistema de mayor ancho.

Si el operando en el sistema de menor ancho es positivo, la extensión se realiza simplemente **completando con ceros a la izquierda** hasta obtener la cantidad de dígitos del otro sistema. Si el operando del menor ancho es negativo, la extensión de signo se hace **agregando unos**.

Ejemplos

- $A + B = 00101011_{(2} + 00101_{(2}$
  - A está en C_2^8 y B en C_2^5
  - Se completa B (positivo) como $00000101_{(2}$
- $A + B = 1010_{(2} + 0110100_{(2}$
  - A está en C_2^4 y B en C_2^7
  - Se completa A (negativo) como $1111010_{(2}$

Notación en exceso o *bias*

En un sistema de notación en exceso, se elige un intervalo $[a, b]$ de enteros a representar, y todos los valores dentro del intervalo se representan con una secuencia de bits de la misma longitud.

La cantidad de bits deberá ser la necesaria para representar todos los enteros del intervalo, inclusive los límites, y por lo tanto estará en función de la longitud del intervalo. Un intervalo $[a, b]$ de enteros, con sus límites incluidos, comprende

exactamente $n = b - a + 1$ valores. Esta longitud del intervalo debe ser cubierta con una cantidad k de bits suficiente, lo cual obliga a que $2^k \geq n$. Supongamos que n sea una potencia de 2 para facilitar las ideas, de forma que $2^k = n$.

Las 2^k secuencias de k bits, ordenadas como de costumbre según su valor aritmético, se aplican a los enteros en $[a, b]$, uno por uno. Es decir, si usamos 3 bits, las secuencias serán 000, 001, 010, ... hasta 111; y los valores representados serán respectivamente:

- 000 = a
- 001 = $a + 1$
- 010 = $a + 2$
- ...
- 111 = b

Notemos que tanto a como b pueden ser **negativos**. Así podemos representar intervalos de enteros arbitrarios con secuencias de k bits, lo que nos vuelve a dar un sistema de representación con signo.

Con este método no es necesario que el bit de orden más alto represente el signo. Tampoco que el intervalo contenga la misma cantidad de números negativos que positivos o cero, aunque para la mayoría de las aplicaciones es lo más razonable.

El sistema en exceso se utiliza como componente de otro sistema de representación más complejo, la representación en punto flotante.

Conversión entre exceso y decimal

Una vez establecido un sistema en exceso:

- Para calcular la secuencia binaria que corresponde a un valor decimal d, a d **le restamos** a y luego convertimos el resultado (que es **no negativo**) a binario sin signo.
- Para calcular el valor decimal d representado por una secuencia binaria, convertimos la secuencia a decimal como en un sistema sin signo, y al resultado (**no negativo**) le **sumamos** el valor de a.

Ejemplo

Representemos en sistema en exceso el intervalo $[-3, 4]$ (que contiene $4-(-3)+1 = 8$ enteros). Como necesitamos 8 secuencias binarias, usaremos 3 bits que producirán las secuencias 000, 001, ..., 111.

- Para calcular la secuencia que corresponde al número 2, hacemos $2-(-3) = 5$ y el resultado será la secuencia **101**.

- Para calcular el valor decimal que está representando la secuencia **011**, convertimos 011 a decimal, que es 3, y le sumamos -3; el resultado es 0.

Preguntas sobre Notación en Exceso

- Dado un valor decimal a representar, ¿cómo calculamos el binario?
- Dado un binario, ¿cómo calculamos el valor decimal representado?
- El sistema en exceso ¿destina un bit para representar el signo?
- ¿Se puede representar un intervalo que no contenga el cero?
- ¿Cómo se comparan dos números en exceso para saber cuál es el mayor?

Continuará