

Funciones para crear o comprobar vectores de tipo entero

integer {base}: enteros
Documentación de R, 4.0.0

Contenido

1 Descripción	2
2 Forma de uso o sintaxis	2
3 Argumentos	2
4 Detalles	2
5 Valor devuelto	4
6 El tipo de almacenamiento entero	5
7 Notas	7
7.1 Sobre la verificación matemática de los números enteros	7
7.2 Sobre las denominaciones de los tipos, modos y clases numéricas	7
8 Referencias	8
9 También véase	8
10 Ejemplos	8
11 Código fuente	10
11.1 integer()	10
11.2 as.integer()	10
11.3 is.integer()	10

1 Descripción

La función `integer()` creará un vector de **tipo entero** ("integer"), que podrá almacenar un subconjunto amplio aunque limitado de los números enteros positivos y negativos.

La función `as.integer()` intentará convertir en un vector de tipo *entero* a cualquier tipo de objeto. Por su parte, la función `is.integer()` verificará si un objeto es de **tipo entero**.

2 Forma de uso o sintaxis

```
integer(length = 0)
as.integer(x, ...)
is.integer(x)
```

3 Argumentos

Tabla 1: Argumentos para las funciones de creación y verificación de vectores *enteros*

Argumento	Valor esperado	Propósito
<code>length=</code> <i>longitud</i>	Un valor entero mayor o igual a cero	Determina la longitud deseada del vector, es decir, el número de elementos que almacenará. El argumento de longitud aceptará números enteros no negativos. Los valores continuos o con decimales (tipo doble) serán convertidos a enteros y la aportación de más de un valor devolverá un mensaje de error.
<code>x=</code>	Un objeto	Un objeto para ser coaccionado o verificado como un vector de tipo <i>entero</i> .
<code>...</code>	Otros argumentos	Otros argumentos que serán pasados desde o hacia otras funciones.

4 Detalles

Un vector *entero* es una estructura fundamental de **R** destinada al almacenamiento exclusivo de datos de tipo *entero* en celdas contiguas, así como para realizar operaciones individuales o en paralelo con los elementos de esta celdas. Los vectores *enteros* subsisten como un legado de los lenguajes *C* y *Fortran*, los cuales forman parte del código interno de **R**. Estos lenguajes requieren todavía de este tipo de almacenamiento para algunas operaciones especiales.

El tipo de dato *entero* es una categorización de valores que permite representar y realizar operaciones con un subconjunto de los números enteros con signo del sistema de numeración decimal, es decir, de los enteros positivos y negativos. Este subconjunto es de rango amplio aunque limitado a 32 *bits*. El tipo *entero* permite almacenar de manera exacta los números sin fracciones o marca decimal, la cual puede tratarse del punto (`.`) o la coma (`,`) dependiendo del país.

Específicamente, **R** define al tipo *entero* como un **tipo de almacenamiento interno** de carácter exacto que podrá contener cualquier valor de un conjunto de números con signo sin marca decimal en un rango de 32 dígitos binarios (*bits*). Este rango equivale a la posibilidad de almacenar 2^{32} valores enteros distintos.

Debido a que, internamente, los valores posibles por dígito son dos (0 y 1), la base del exponente será el número dos (2). Por su parte, el valor del exponente es equivalente al número de dígitos disponibles para almacenamiento, en este caso, treinta y dos (32). La elevación de la base a este exponente permite obtener el número total de combinaciones disponibles para representar valores enteros en un rango de 32 *bits*.

No obstante, ya que **R** define al tipo *entero* como uno capaz de almacenar valores con signo, a nivel interno se destinará uno de los 32 *bits* a almacenar el valor del signo del número, con un cero (0) para valores positivos y un uno (1) para negativos. Por ello, el valor del exponente se reducirá en una unidad ($32 - 1$). Del mismo modo, dado que el intervalo de valores almacenables atraviesa el cero en la recta numérica, se deberá adicionar y sustraer una unidad en los límites del intervalo de almacenamiento. En consecuencia, el rango de enteros representables irá de $-2^{31} + 1$ a $2^{31} - 1$ (es decir, aproximadamente de $\mp 2 \times 10^9$).

De este modo, las funciones básicas para crear vectores *enteros* son `integer()`, `as.integer()` y `c()`. Por su parte, la función `is.integer()` se utilizará para comprobar si un objeto es de **tipo entero** (`"integer"`). En particular, `as.integer()`, `is.integer()` y `c()` son funciones **primitivas**, por lo que su código fuente está implementado de manera interna y no será visible directamente por la usuaria. Para más información sobre la forma de utilización de las funciones mencionadas, puedes consultar la sección *Valor devuelto* en esta misma página de ayuda.

Como objetos, los vectores *enteros* son un **tipo de vector** atómico o fundamental de **R**, por lo que no podrán convertirse en objetos más simples ni contener elementos que no sean del mismo tipo. Los **vectores atómicos** están definidos a nivel interno por su **tipo de almacenamiento**. El tipo de almacenamiento determina el tipo de vector y las operaciones que le son aplicables.

El **tipo interno de almacenamiento** de un vector *entero* tendrá asignada como atributo la etiqueta inglesa *entero* (`"integer"`) y su **modo de almacenamiento** tendrá asociada como atributo la etiqueta inglesa *numérico* (`"numeric"`). Además, los vectores *enteros* son una **clase de objeto** del lenguaje, por lo que también podrán extender sus propiedades a nuevos objetos.

Los vectores de tipo *entero* (`"integer"`) podrán almacenar valores enteros en forma más compacta que otros **tipos de almacenamiento numérico**, como el almacenamiento **dobles** (`"double"`), que puede guardar igualmente enteros de manera exacta en un rango mucho más amplio. Sin embargo, las capacidades actuales de los ordenadores vuelven imperceptibles, en la práctica, estas ganancias en eficiencia derivadas del uso del tipo de datos *entero*.

Para cada tipo de **vector** atómico, salvo para los vectores **crudos** (`"raw"`), existe un tipo propio de **valor no disponible** NA. Así, al tipo *entero* le corresponderá el valor no disponible `NA_integer_`. Sin embargo, los valores no disponibles de los vectores *enteros* serán mostrados en pantalla solamente con los caracteres NA.

Si deseas asegurarte de que los vectores *enteros* reciban sólo valores no disponibles de tipo *entero*, puedes utilizar el valor `NA_integer_` (en vez de la forma más simple NA) en las operaciones de **asignación**. Para más información puedes consultar más adelante la sección *Ejemplos*, así como la página de ayuda de los **valores no disponibles**.

5 Valor devuelto

Cualquier cifra tecleada sin comillas (" ") en la consola de **R** será devuelta como un número por el lenguaje. Es decir, **R** podrá identificar *literales* numéricos en el código fuente y los devolverá como valores numéricos constantes. Sin embargo, para devolver directamente un valor de tipo *entero* tendremos que añadir, sin mediar espacio, el sufijo L (proveniente del tipo LONG INTEGER de C) a todos los literales enteros tecleados.

Por ejemplo, para crear el valor entero 1 deberemos usar el sufijo L para calificarlo como un dato *entero*. Entonces, el literal numérico 1L tecleado en la consola será devuelto como la constante entera 1 y tendrá asociada internamente la etiqueta "integer". En cambio, si sólo tecleamos el literal numérico 1, éste será devuelto como un valor de tipo *doble*.

R reconocerá literales numéricos en el sistema de numeración decimal y hexadecimal. Los valores en base decimal se escribirán literalmente, mientras que los valores en base hexadecimal deberán estar acompañados del prefijo 0x o 0X para indicar que se trata de números en esa base. Particularmente, si queremos caracterizar a un literal numérico de cualquier base como un *entero* tendremos que añadir siempre el sufijo L. Por ejemplo, la cifra 0x10L devolverá el valor entero 16 a partir de la representación hexadecimal ingresada. Por otro lado, la cifra en notación científica 1e3L devolverá el valor entero 1000 y será equivalente a haber ingresado el literal numérico 1000L.

El acceso a las celdas o los elementos de un vector *entero* se realizará por medio de las operaciones de **indización** ([]) de sus elementos tal como, por ejemplo: un_vector[i]; en donde i será el número índice del elemento que se desea **extraer** o **almacenar**.

La función integer() creará un vector *entero* con el número de elementos especificado en el argumento de longitud, length=. Al momento de su creación, cada elemento del vector será igual a cero (0L). Luego, se podrán **asignar** valores enteros positivos o negativos, así como **valores no disponibles** (NA), al vector recientemente creado.

La función as.integer() intentará coaccionar al objeto referido en el argumento x= al tipo *entero* y lo devolverá como un vector de este tipo. Si la coacción no ha sido exitosa, el resultado será un valor no disponible (NA).

Cuando se intente coaccionar un valor al tipo *entero*, o se ingrese un literal entero en la consola, y éste sea mayor o menor a los límites del intervalo de almacenamiento, se devolverá un **valor no disponible** (NA). Este comportamiento es diferente al de S, que devolvía un valor del mismo signo igual al límite del intervalo de almacenamiento.

Cuando as.integer() reciba valores con números reales, las fracciones decimales de estos valores serán truncados hacia el cero siempre y cuando se encuentren dentro del intervalo de almacenamiento entero. Esto significa que as.integer(x) funcionará igual que trunc(x) en estos casos. Del mismo modo, las partes imaginarias de los valores de **números complejos** se descartarán con una advertencia.

as.integer() podrá convertir en valores enteros a aquellas cadenas de caracteres (es decir, literales entre comillas) que contengan representaciones de números del sistema decimal o hexadecimal entre espacios en blanco iniciales o finales. No obstante, cada cadena de caracteres deberá contener una sola cifra, sin espacios intercalados, de lo contrario esos elementos serán coaccionados en valores no disponibles (NA). Algunas plataformas de sistema operativo podrían aceptar la coacción de cadenas de caracteres que representen números en otros sistemas de numeración diferentes al decimal o al hexadecimal, como el binario o el octal.

`as.integer()` eliminará los atributos, incluidos los nombres, de los objetos coaccionados, tal como lo hace la función `as.vector()`. Para asegurarte de que un objeto `x` permanezca con el tipo entero sin perder sus atributos, podrás asignar a un vector la etiqueta del tipo entero ("integer") con la función `storage.mode()`, por ejemplo, como en: `storage.mode(x) <- "integer"`. Esta forma de coacción hacia el tipo entero tiene la ventaja de modificar el **tipo de almacenamiento** sin eliminar los atributos del objeto,

La función `c()` devolverá un vector entero si se utiliza para combinar valores enteros que estén separados por comas, siempre que al final de cada valor numérico se añada, sin mediar espacio, la letra ele mayúscula (L), por ejemplo: `c(1L, 2L, 3L)`. El resultado de la combinación de elementos enteros creará un vector de tipo entero si ninguno de los valores combinados tiene valores decimales. Para mayor información sobre la combinación de elementos para crear vectores de un determinado tipo puedes ver la página de ayuda de la función `c()`.

La función `is.integer()` devolverá el **valor lógico** verdadero (TRUE) o falso (FALSE) dependiendo de si el objeto referido en el argumento `x` es de tipo entero, es decir, de si el vector tiene asociada internamente la etiqueta "integer". Toma en cuenta que esta función sólo verificará el **tipo de almacenamiento específico** del vector, no si los objetos contenidos en él son, matemáticamente, números enteros; para más información puedes ver las secciones *Nota* y *Ejemplos* en esta página de ayuda. En el caso de los **factores**, que asocian números enteros a valores categóricos, `is.integer()` devolverá el valor lógico falso (FALSE) al momento de verificar el tipo del objeto.

6 El tipo de almacenamiento entero

En computación, un tipo de *almacenamiento* o de *dato* es la categorización de un conjunto de valores y un conjunto de operaciones definidas para esos valores. Así, un tipo de almacenamiento caracteriza a un conjunto de objetos con la misma representación. De esta manera, el tipo de dato entero, llamado también *integral*, permite representar a un subconjunto de los números enteros (\mathbb{Z}) y realizar operaciones matemáticas con ellos.

En términos coloquiales, los enteros son números que se representan sin la marca del separador de decimales, ya sea que la marca decimal utilizada sea el punto (.) o la coma (,), y pueden ser números tanto positivos como negativos incluyendo al cero (0). En términos matemáticos, los números enteros se refieren al conjunto \mathbb{Z} que comprende a los subconjuntos de los números naturales \mathbb{N} , al número cero (0) y a los números naturales con signo negativo $-\mathbb{N}$, tal que: $\mathbb{Z} \ni \{\infty_{\mathbb{N}}^-, \dots, -3, -2, -1, 0, +1, +2, +3, \dots, \infty_{\mathbb{N}}^+\}$.

De esta manera, el conjunto de los números enteros \mathbb{Z} hereda las propiedades de los números naturales \mathbb{N} con la adición de los subconjuntos que incluyen al cero (0) y a los naturales negativos $-\mathbb{N}$. Las propiedades más importantes de los números naturales son: 1) entre un número natural a y su sucesor $a + 1$ no existe ningún número natural, 2) entre dos números naturales no sucesivos siempre habrá un número finito de números naturales, 3) todo número natural después del número uno (1) es precedido por otro número natural y 4) los números naturales son infinitos.

El número entero más grande que podrá representarse en una computadora depende del tamaño de *agrupación del dato*. El tamaño de agrupación se refiere al número de dígitos binarios o *bits* usados para almacenar un número y es variable entre plataformas de sistema (32 o 64 bits, usualmente), lenguajes de programación y tipos de almacenamiento. Por este motivo, el subconjunto de enteros disponibles para ser representados también podrá variar en algunas ocasiones.

Entonces, si el tamaño de agrupación para números enteros es de un número de dígitos o *bits* n , los enteros en base decimal tendrán una representación interna de n dígitos binarios, los cuales podrán adoptar sólo uno de dos valores: cero (0) o uno (1). De esta forma, un número entero tendrá una representación binaria a nivel interno con n dígitos d y $n - 1$ posiciones contadas de derecha a izquierda a partir de cero: $d_{n-1} \dots d_3 d_2 d_1 d_0$.

En caso de que los enteros puedan adoptar valores positivos y negativos, la posición $n - 1$ correspondiente al dígito de extrema izquierda d_{n-1} se destinará a almacenar el valor del signo y será conocida como el *bit* o banderín de signo. Así, el número cero (0) se usará para identificar a los números mayores o iguales a cero (es decir, los positivos), y el número uno (1) para identificar a los números menores a cero (es decir, los negativos), y corresponderán al valor del signo positivo (+) y negativo (−) de los números en sistema decimal, respectivamente.

Ya que el banderín de signo ocupará un dígito completo en la representación de valores en sistema binario, el número de posiciones disponibles efectivamente para almacenar a un entero será de $n - 1$ dígitos binarios. En consecuencia, si se trata de enteros con signo, el rango de representación binaria de estos irá de $-2^{n-1} + 1$ a $2^{n-1} - 1$. La adición y sustracción de una unidad en los límites del intervalo es consecuencia del conteo a partir de cero. Si, en cambio, el entero no tiene un dígito o banderín de signo, el número de enteros representables será de 2^n dígitos e irá, usualmente, de 0 a $2^n - 1$.

Por ejemplo, cuando el tamaño de agrupación del dato sea de 32 bits, se podrán representar hasta 2^{32} valores sin signo. Es decir, el tipo entero tendrá 4 294 967 296 posiciones de almacenamiento. No obstante, si el tamaño de agrupación cuenta con un dígito destinado a representar al signo positivo o negativo, entonces el rango de enteros representables se dividirá en mitades e irá de $-2^{31} + 1$ a $+2^{31} - 1$, o sea, de −2 147 483 647 a 2 147 483 647.

Las diferencias entre el formato de almacenamiento *entero* y la representación en pantalla de las cifras segurá las reglas básicas de la conversión aritmética entre los sistemas de numeración decimal y binario. Así, la conversión aritmética de valores en base decimal a base binaria, y viceversa, seguirá un algoritmo estándar en el caso de los enteros positivos y uno especial, conocido en computación como el *complemento a dos*, en caso de los enteros negativos. A continuación se sintetiza el algoritmo para los cuatro casos posibles:

- **Conversión de enteros positivos en base decimal a base binaria.** La conversión de un número entero mayor o igual a cero en base decimal x_{10} a su equivalente en base binaria x_2 se obtendrá al concatenar cada residuo $r_1 \dots r_n$ obtenido de la sucesiva división del número decimal x_{10} y sus cocientes $q_1 \dots q_{n-1}$ entre la base b (2) hasta que el último cociente q_n sea igual a cero. Una vez que se haya obtenido el número binario positivo x_2 se deberá extender la base binaria a su izquierda. Es decir, si el tamaño de agrupación del dato es de m número de *bits* y el número de dígitos de x_2 es n , entonces el número de ceros (0) a concatenar a la izquierda estará dado por $m - n$ siempre que $m \geq 0$.
- **Conversión de enteros negativos en base decimal a base binaria.** La conversión de un número entero negativo en base decimal x_{10} a la base binaria x_2 seguirá el mismo procedimiento que el de los enteros positivos pero, adicionalmente, se negará el número obtenido a partir de la concatenación de residuos $y_2 = \neg(r_n \dots r_1)$ y luego se sumará una unidad binaria a dicho número: $x_2 = y_2 + 1_2$. Así, una vez que se haya obtenido el número negativo x_2 deberemos extender la base binaria a su izquierda. Es decir, si el tamaño de agrupación del dato es de m número de *bits* y el número de dígitos de x_2 es n , entonces el número de unos (1) a concatenar a la izquierda estará dado por $m - n$ siempre que $m \geq n$.

- **Conversión de enteros positivos en base binaria a base decimal.** La conversión de un entero binario x_2 mayor o igual a cero a su equivalente en base decimal x_{10} se obtendrá al sumar el valor de cada dígito multiplicado por la base binaria (2) elevada a la potencia correspondiente a la posición del dígito, así: $x_{10} = d_{n-1} \cdot 2^{n-1} + \dots + d^3 \cdot 2^3 + d_2 \cdot 2^2 + d_1 \cdot 2^1 + d_0 \cdot 2^0$.
- **Conversión de enteros negativos en base binaria a base decimal.** La conversión de un entero binario x_2 menor a cero a su equivalente en base decimal x_{10} se podrá realizar con los siguientes pasos. Primero, se deberá realizar la negación (\neg) de los valores del número binario x_2 . Luego, a este número se le deberá sumar una unidad en base binaria: $y_2 = \neg x_2 + 1_2$. Después de realizar la negación y adición de una unidad, al número binario resultante y_2 se le podrá convertir a la base decimal con el procedimiento indicado para la conversión de binarios mayores o iguales a cero: $y_{10} = d_{n-1} \cdot 2^{n-1} + \dots + d^3 \cdot 2^3 + d_2 \cdot 2^2 + d_1 \cdot 2^1 + d_0 \cdot 2^0$. Finalmente, al número en base decimal y_{10} obtenido, se le deberá multiplicar por menos uno para invertirle el signo: $-y_{10}$. El resultado de esta multiplicación será el número negativo buscado en base decimal x_{10} . A continuación, se especifica el resumen de este procedimiento: $x_{10} = (\neg x_2 + 1_2)_{10} \times -1_{10} = (y_2)_{10} \times -1 = y_{10} \times -1 = -y_{10}$.

A diferencia de la representación de los números reales, cuyo almacenamiento nunca es totalmente exacto, el almacenamiento exacto de los números enteros en base binaria será posible siempre dentro del intervalo de almacenamiento.

7 Notas

7.1 Sobre la verificación matemática de los números enteros

La expresión `is.integer(x)` no verificará si un objeto `x` contiene valores con números enteros, sino cuál es el **tipo de almacenamiento** del vector. Para verificar que los elementos de un vector sean números enteros, en el sentido matemático (\mathbb{Z}), utilice la función `round()` de la forma indicada en la función `es.numeroentero(x)` construida en la sección de *Ejemplos* de esta misma página de ayuda.

7.2 Sobre las denominaciones de los tipos, modos y clases numéricas

"integer" es la etiqueta que identifica el **tipo interno de almacenamiento** que permite a los vectores *enteros* trabajar con números de este tipo. Por otro lado, la etiqueta "double" identifica al tipo interno de almacenamiento *doble* que permite a los vectores trabajar con valores continuos o reales. En cambio, la etiqueta "numeric" es el nombre de un **modo de almacenamiento** de datos, que agrupa tanto al tipo "integer" como al tipo "double" bajo una sola categoría. En este sentido, cuando hablamos del *tipo* del vector, nos referimos solamente al tipo interno de almacenamiento, nunca a su modo de almacenamiento.

Los vectores, además de poseer tipos y modos de almacenamiento, son también objetos del lenguaje. Los objetos consisten en entidades abstractas que representan a entidades concretas del mundo real. Como abstracciones, poseen sólo algunos atributos del objeto representado. Así, la *clase* de un objeto en R consistirá en el subconjunto de atributos que determina al tipo de objeto representado.

Uno de los atributos más importantes de cualquier objeto son los métodos y funciones específicas que pueden aplicársele. A partir de S4 (el antecedente de **R**) se definió formalmente a la clase "integer" como una con métodos específicos para objetos con el tipo de almacenamiento "integer". Esto supone que los vectores de tipo "integer", si bien cuentan con un modo de almacenamiento definido como "numeric", pertenecen a una clase con un conjunto de métodos propio. Para obtener más información, puedes consultar la sección *Ejemplos* en esta misma página de ayuda.

Ya que **R** utiliza indistintamente al modo de almacenamiento "numeric" para agrupar en una categoría a los vectores almacenados con los tipos de internos de almacenamiento "integer" y "double", pero aplica los métodos de la clase "integer" para el tipo "integer" y los métodos de la clase "numeric" para el tipo "double", las usuarias podrían encontrarse en la situación de no estar seguras de qué funciones son válidas para un objeto numérico. Para evitar confusiones sobre el tipo de objeto de que se trata, puedes consultar siempre el tipo interno de almacenamiento con la función `storage.mode()` o, directamente, la clase del objeto con la función `class()`.

8 Referencias

Becker, Richard A., John M. Chambers, y Allan R. Wilks. *The New S Language: A Programming Environment for Data Analysis and Graphics*. Wadsworth & Brookscole. Boca Raton, FL: CRC Press, 1988.

9 También véase

Las funciones `numeric()` y `storage.mode()` para más información sobre otros tipos de almacenamiento numérico.

10 Ejemplos

```
## as.integer() truncará al coaccionar:
x <- pi * c(-1:1, 10) # -3.141593 0.000000 3.141593 31.415927

as.integer(x) # -3 0 3 31

## is.integer() comprobará el tipo de
## almacenamiento, no el valor numérico
is.integer(1) # ¡es falso!
is.integer(1L) # ¡es verdadero!
is.integer(NA) # ¡es falso!
is.integer(NA_integer_) # ¡es verdadero!

## Función para comprobar que los elementos de un vector entero
## también son matemáticamente números enteros (Z)
es.numeroentero <- function(x, tol = .Machine$double.eps^0.5) {
```



```

    abs(x - round(x)) < tol
}

es.numeroentero(1) # es verdadero
es.numeroentero(1.0) # es verdadero
es.numeroentero(1.00001) # es falso
(x <- seq(1, 5, by = 0.5)) # 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
es.numeroentero(x) #--> TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE

## Conversión de enteros de 32 bits, positivos o negativos,
## del sistema decimal al sistema binario
decAbin <- function(x) {
  vapply(as.integer(x), function(x) {
    b <- substr(as.character(rev(intToBits(x))), 2L, 2L)
    paste0(c(b[1L], " ", b[2:32]), collapse = "")
  }, "") |>
    noquote()
}

(x<-decAbin(+3L)) #0 00000000000000000000000000000011 # Banderín + a la izquierda
(y<-decAbin(-3L)) #1 11111111111111111111111111111101 # Banderín - a la izquierda

## Conversión de enteros de 32 bits, positivos o negativos,
## del sistema binario al sistema decimal

binAdec <- function(d) {
  # Recibe una cadena de caracteres y elimina
  # los espacios dentro de ella
  d |> as.character() |>
   gsub(pattern = " ", replacement = "", x = _) |>
    strsplit(split = "") |> unlist() |>
    as.integer() -> d
  len <- length(d)
  # Exponentes para la base
  exp <- seq.int(from = len - 1, to = 0, by = -1)
  # Decidir qué hacer para el caso positivo (0) y negativo (1)
  if (d[1] == 0) {
    sum(d*(2^exp))
  } else {
    b <- as.integer(!d); u <- integer(length = len)
    z <- integer(length = len); u[len] <- 1L
    m <- cbind(b, u, z); acarreo <- 0L
    for (i in len:2) {
      s <- m[i, 1] + m[i, 2] + acarreo
      acarreo <- 0L
      if (s == 0 | s == 1) {
        m[i, 3] <- s
      } else {
        m[i, 3] <- 0L
        acarreo <- 1L
      }
    }
  }
}

```

```

    }
  }
  d <- m[, 3]
  sum(d*(2^exp))*(-1)
}

binAdec(x) # +3
binAdec(y) # -3

## Para verificar las funciones de la clase "integer" y "numeric"
methods(class = "integer") # as.data.frame      coerce      Ops

methods(class = "numeric") # all.equal      as.data.frame  as.Date  as.POSIXct
                           # as.POSIXlt    as.raster      coerce    Ops

```

11 Código fuente

11.1 integer()

```

function (length = 0L)
  .Internal(vector("integer", length))

```

11.2 as.integer()

```

function (x)
  .Primitive("as.integer")

```

11.3 is.integer()

```

function (x)
  .Primitive("is.integer")

```

[Paquete base versión 4.2.2 [Índice](#)]