MOOC de Criptología Matemática. Números y su Representación

Leandro Marín

Módulo I. Sesión 2.
Dificultad Baja

La Base 10

- Nuestro sistema habitual para representar los números es la base 10.

La Base 10

- Nuestro sistema habitual para representar los números es la base 10.
- Eso significa que disponemos de 10 símbolos, $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$, y dependiendo de la posición de cada cifra, debemos multiplicar por la potencia de 10 que corresponda.

La Base 10

- Nuestro sistema habitual para representar los números es la base 10.
- Eso significa que disponemos de 10 símbolos, $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$, y dependiendo de la posición de cada cifra, debemos multiplicar por la potencia de 10 que corresponda.
- Así, el número $167 = 1 \cdot 10^2 + 6 \cdot 10^1 + 7 \cdot 10^0$

La Base 10

- Nuestro sistema habitual para representar los números es la base 10.
- Eso significa que disponemos de 10 símbolos, $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$, y dependiendo de la posición de cada cifra, debemos multiplicar por la potencia de 10 que corresponda.
- Así, el número $167 = 1 \cdot 10^2 + 6 \cdot 10^1 + 7 \cdot 10^0$
- En este ejemplo diremos que 1 es la cifra más significativa y 7 la menos significativa.

La Base 10

- Nuestro sistema habitual para representar los números es la base 10.
- Eso significa que disponemos de 10 símbolos, $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$, y dependiendo de la posición de cada cifra, debemos multiplicar por la potencia de 10 que corresponda.
- Así, el número $167 = 1 \cdot 10^2 + 6 \cdot 10^1 + 7 \cdot 10^0$
- En este ejemplo diremos que 1 es la cifra más significativa y 7 la menos significativa.
- Todo esto es es una convención, en realidad no existe ningún problema en hacerlo en otro orden o con otra base, y en informática de hecho se cambia habitualmente.

Las cifras en sage

- Las cifras en base 10 de un número se pueden obtener con el comando `digits(10)`.

```
n = 167
L = n.digits(10)
print L
```

Las cifras en sage

- Las cifras en base 10 de un número se pueden obtener con el comando `digits(10)`.

```
n = 167
L = n.digits(10)
print L
```

- Las cifras las obtenemos en forma de lista, en este caso `[7,6,1]`.

Las cifras en sage

- Las cifras en base 10 de un número se pueden obtener con el comando `digits(10)`.

```
n = 167
L = n.digits(10)
print L
```

- Las cifras las obtenemos en forma de lista, en este caso `[7,6,1]`.
- Como podemos ver, el orden es el inverso al habitual, el índice 0 es el que corresponde a la potencia 10^0, el índice 1 el que corresponde a la potencia 10^1 y el índice 2 el que corresponde a la potencia 10^2.

La Operación Inversa

- Para reconstruir un númeroa partir de sus cifras podemos utilizar el siguiente código:

```
n = ZZ([1,2,3,4,5],10)
print n
```

La Operación Inversa

- Para reconstruir un númeroa partir de sus cifras podemos utilizar el siguiente código:

```
n = ZZ([1,2,3,4,5],10)
print n
```

- El número obtenido es 54321.

La Operación Inversa

- Para reconstruir un númeroa partir de sus cifras podemos utilizar el siguiente código:

```
n = ZZ([1,2,3,4,5],10)
print n
```

- El número obtenido es 54321.
- Podemos usar variables para representar las listas.

```
n = 1925
cifras = n.digits(10)
m = ZZ(cifras,10)
m == n
```

La Operación Inversa

- Para reconstruir un númeroa partir de sus cifras podemos utilizar el siguiente código:

```
n = ZZ([1,2,3,4,5],10)
print n
```

- El número obtenido es 54321.
- Podemos usar variables para representar las listas.

```
n = 1925
cifras = n.digits(10)
m = ZZ(cifras,10)
m == n
```

- El resultado de esta condición lógica es `True`.

La Representación Binaria

- Aunque la representación en base 10 es la mas habitual entre los humanos, no sucede lo mismo en el mundo de la informática, donde la representación más habitual es la que se hace en base 2 o binaria.

La Representación Binaria

- Aunque la representación en base 10 es la mas habitual entre los humanos, no sucede lo mismo en el mundo de la informática, donde la representación más habitual es la que se hace en base 2 o binaria.
- En la representación binaria tenemos dos símbolos 0 y 1 y utilizamos multiplicamos por las potencias de 2 dependiendo de las posiciones de las cifras.

La Representación Binaria

- Aunque la representación en base 10 es la mas habitual entre los humanos, no sucede lo mismo en el mundo de la informática, donde la representación más habitual es la que se hace en base 2 o binaria.
- En la representación binaria tenemos dos símbolos 0 y 1 y utilizamos multiplicamos por las potencias de 2 dependiendo de las posiciones de las cifras.
- Así el número binario 1101 corresponde a $1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 8 + 4 + 1 = 13$.

La Representación Binaria

- Aunque la representación en base 10 es la mas habitual entre los humanos, no sucede lo mismo en el mundo de la informática, donde la representación más habitual es la que se hace en base 2 o binaria.
- En la representación binaria tenemos dos símbolos 0 y 1 y utilizamos multiplicamos por las potencias de 2 dependiendo de las posiciones de las cifras.
- Así el número binario 1101 corresponde a $1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 8 + 4 + 1 = 13$.
- Si le pedimos a `sage` las cifras en base 2 del número 13 obtenemos con el comando `print 13.digits(2)` obtenemos `[1, 0, 1, 1]`, es decir, las cifras binarias empezando por la menos significativa hasta las más significativa.

Fijando el Número de Cifras I

- En muchas ocasiones nos interesará que las listas de cifras de un número tengan una longitud fija.

Fijando el Número de Cifras I

- En muchas ocasiones nos interesará que las listas de cifras de un número tengan una longitud fija.
- Por ejemplo, supongamos un algoritmo en el cual los números sabemos que se pueden representar con 8 bits y tenemos que realizar algún tipo de operación con cada uno de esos 8 bits. Lo normal es que muchos valores tengan menos de 8 y debamos completar con ceros.

Fijando el Número de Cifras I

- En muchas ocasiones nos interesará que las listas de cifras de un número tengan una longitud fija.
- Por ejemplo, supongamos un algoritmo en el cual los números sabemos que se pueden representar con 8 bits y tenemos que realizar algún tipo de operación con cada uno de esos 8 bits. Lo normal es que muchos valores tengan menos de 8 y debamos completar con ceros.
- Eso lo podemos hacer con código, por ejemplo:

```
n = 19
L = n.digits(2)
while len(L) < 8:
  L.append(0)
```

Fijando el Número de Cifras I

- En muchas ocasiones nos interesará que las listas de cifras de un número tengan una longitud fija.
- Por ejemplo, supongamos un algoritmo en el cual los números sabemos que se pueden representar con 8 bits y tenemos que realizar algún tipo de operación con cada uno de esos 8 bits. Lo normal es que muchos valores tengan menos de 8 y debamos completar con ceros.
- Eso lo podemos hacer con código, por ejemplo:

```
n = 19
L = n.digits(2)
while len(L) < 8:
  L.append(0)
```

- En este código lo que se hace es que mientras que la longitud de la lista `L` sea menor que 8, añadimos ceros.

Fijando el Número de Cifras II

- Sin embargo es más sencillo aplicar un parámetro adicional optativo al comando `digits` que es `L = n.digits(2,None,8)`.

Fijando el Número de Cifras II

- Sin embargo es más sencillo aplicar un parámetro adicional optativo al comando `digits` que es
  `L = n.digits(2,None,8)`.
- En ambos casos, con el valor $\mathtt{n} = 19$ obtendríamos
  `L == [1, 1, 0, 0, 1, 0, 0, 0]`.

Fijando el Número de Cifras II

- Sin embargo es más sencillo aplicar un parámetro adicional optativo al comando `digits` que es `L = n.digits(2,None,8)`.
- En ambos casos, con el valor $\mathtt{n} = 19$ obtendríamos `L == [1, 1, 0, 0, 1, 0, 0, 0]`.
- Si reconstruimos el número, vemos que

$$L[0]2^0 + L[1]2^1 + \cdots + L[7]2^7 = 2^0 + 2^1 + 2^4 = 1 + 2 + 16 = 19$$

Fijando el Número de Cifras II

- Sin embargo es más sencillo aplicar un parámetro adicional optativo al comando `digits` que es
  `L = n.digits(2,None,8)`.
- En ambos casos, con el valor `n` $= 19$ obtendríamos
  `L == [1, 1, 0, 0, 1, 0, 0, 0]`.
- Si reconstruimos el número, vemos que

$$L[0]2^0 + L[1]2^1 + \cdots + L[7]2^7 = 2^0 + 2^1 + 2^4 = 1 + 2 + 16 = 19$$

- Esto se puede hacer también para otras bases.

Recuperando el Número Original

- Tal y como hicimos en base 10, si tenemos una lista con las cifras de un número binario, podemos recuperar el número del siguiente modo:

```
L = [1,0,1,1,0,0,0,1]
n = ZZ(L,2)
print n
```

Recuperando el Número Original

- Tal y como hicimos en base 10, si tenemos una lista con las cifras de un número binario, podemos recuperar el número del siguiente modo:

```
L = [1,0,1,1,0,0,0,1]
n = ZZ(L,2)
print n
```

- El número asociado a esta lista de cifras binarias será: $n = \sum_{i=0}^{7} L[i]2^i = 2^0 + 2^2 + 2^3 + 2^7 = 141$

Recuperando el Número Original

- Tal y como hicimos en base 10, si tenemos una lista con las cifras de un número binario, podemos recuperar el número del siguiente modo:

```
L = [1,0,1,1,0,0,0,1]
n = ZZ(L,2)
print n
```

- El número asociado a esta lista de cifras binarias será: $n = \sum_{i=0}^{7} L[i]2^i = 2^0 + 2^2 + 2^3 + 2^7 = 141$
- Recordemos que el índice de la lista es el que nos indica la potencia de 2 por la que tenemos que multiplicar.

Recuperando el Número Original

- Tal y como hicimos en base 10, si tenemos una lista con las cifras de un número binario, podemos recuperar el número del siguiente modo:

```
L = [1,0,1,1,0,0,0,1]
n = ZZ(L,2)
print n
```

- El número asociado a esta lista de cifras binarias será: $n = \sum_{i=0}^{7} L[i]2^i = 2^0 + 2^2 + 2^3 + 2^7 = 141$
- Recordemos que el índice de la lista es el que nos indica la potencia de 2 por la que tenemos que multiplicar.
- No debemos confundir esto con la representación binaria del número que se hace con la notación habitual de mayor peso a la izquierda, lo podemos ver poniendo `bin(n)`.

Introducción

- En muchos casos, las cifras binarias de un número se utilizan para representar condiciones lógicas.

Introducción

- En muchos casos, las cifras binarias de un número se utilizan para representar condiciones lógicas.
- El símbolo 0 se utiliza para representar la condición de `Falso` y el símbolo 1 para representar la condición `Verdadero`.

Introducción

- En muchos casos, las cifras binarias de un número se utilizan para representar condiciones lógicas.
- El símbolo 0 se utiliza para representar la condición de `Falso` y el símbolo 1 para representar la condición `Verdadero`.
- Un número de k bits, en muchas ocasiones no representa realmente un número, sino k condiciones lógicas unidas.

Introducción

- En muchos casos, las cifras binarias de un número se utilizan para representar condiciones lógicas.
- El símbolo 0 se utiliza para representar la condición de `Falso` y el símbolo 1 para representar la condición `Verdadero`.
- Un número de k bits, en muchas ocasiones no representa realmente un número, sino k condiciones lógicas unidas.
- Vamos a ver en esta sección las operaciones lógicas más habituales y cómo se pueden agrupar.

La Operación `and` (y lógico)

- Dadas dos condiciones lógicas, `a` y `b`, la condición lógica `a and b` será cierta exactamente cuando sean ciertas a la vez `a` y `b`.

La Operación and (y lógico)

- Dadas dos condiciones lógicas, `a` y `b`, la condición lógica `a and b` será cierta exactamente cuando sean ciertas a la vez `a` y `b`.
- Por ejemplo:

```
for j in range(10):
    if 2*j+1<18 and 3*j-15>4:
        print j,2*j+1,3*j-15
```

nos dará los resultados

```
7 15 6
8 17 9
```

Es decir, que para los valores de $j = 7, 8$ tendremos que $2j + 1 < 18$ (concretamente los valores de $2j + 1$ son $15, 17$) y también $3j - 15 > 4$ (tomará los valores $6, 9$).

Vamos a visualizar todos los valores:

```
for j in range(10):
  if 2*j+1<18 and 3*j-15>4:
    print "Verdadero ",j,2*j+1,3*j-15
  else:
    print "Falso      ",j,2*j+1,3*j-15
```

nos dará los resultados

```
Falso      0  1  -15
Falso      1  3  -12
Falso      2  5  -9
Falso      3  7  -6
Falso      4  9  -3
Falso      5  11  0
Falso      6  13  3
Verdadero  7  15  6
Verdadero  8  17  9
Falso      9  19  12
```

Como podemos observar, en los valores de j entre 0 y 6 se incumple la segunda condición, mientras que en el valor $j = 9$ se incumple la primera.

Operando en Bits

Vamos a ver la tabla de operación de `and` tanto con valores booleanos como con la representación en bits:

`and`		

`&`		

Operando en Bits

Vamos a ver la tabla de operación de and tanto con valores booleanos como con la representación en bits:

and	False	

&	0	

Operando en Bits

Vamos a ver la tabla de operación de `and` tanto con valores booleanos como con la representación en bits:

`and`	`False`	`True`

`&`	0	1

Operando en Bits

Vamos a ver la tabla de operación de `and` tanto con valores booleanos como con la representación en bits:

and	False	True
False		

&	0	1
0		

Operando en Bits

Vamos a ver la tabla de operación de `and` tanto con valores booleanos como con la representación en bits:

`and`	`False`	`True`
`False`		
`True`		

&	0	1
0		
1		

Operando en Bits

Vamos a ver la tabla de operación de `and` tanto con valores booleanos como con la representación en bits:

and	False	True
False	False	
True		

&	0	1
0	0	
1		

Operando en Bits

Vamos a ver la tabla de operación de `and` tanto con valores booleanos como con la representación en bits:

and	False	True
False	False	False
True		

&	0	1
0	0	0
1		

Operando en Bits

Vamos a ver la tabla de operación de `and` tanto con valores booleanos como con la representación en bits:

and	False	True
False	False	False
True	False	

&	0	1
0	0	0
1	0	

Operando en Bits

Vamos a ver la tabla de operación de `and` tanto con valores booleanos como con la representación en bits:

and	False	True
False	False	False
True	False	True

&	0	1
0	0	0
1	0	1

La Operación `or` (o lógico)

- Dadas dos condiciones lógicas, `a` y `b`, la condición lógica `a or b` será cierta cuando sea ciertas alguna de las conciones `a` o `b` o las dos al mismo tiempo. Sólo será falso si ambas son falsas.

La Operación or (o lógico)

- Dadas dos condiciones lógicas, a y b, la condición lógica a or b será cierta cuando sea ciertas alguna de las conciones a o b o las dos al mismo tiempo. Sólo será falso si ambas son falsas.
- Por ejemplo:

```
for j in range(7):
  if 2*j+1<6 or 3*j-15>0:
    print j,2*j+1,3*j-15
```

nos dará los resultados

```
0 1 -15
1 3 -12
2 5 -9
6 13 3
```

Es decir, que para los valores de $j = 0, 1, 2, 6$ tendremos que o bien $2j + 1 < 6$ (concretamente los casos $j = 0, 1, 2$) o $3j - 15 > 0$ (lo que añade el caso $j = 6$).

Vamos a visualizar todos los valores:

```
for j in range(7):
  if 2*j+1<6 or 3*j-15>0:
    print "Verdadero ",j,2*j+1,3*j-15
  else:
    print "Falso     ",j,2*j+1,3*j-15
```

nos dará los resultados

```
Verdadero  0 1 -15
Verdadero  1 3 -12
Verdadero  2 5 -9
Falso      3 7 -6
Falso      4 9 -3
Falso      5 11 0
Verdadero  6 13 3
```

Como podemos observar, en los valores de j entre 0 y 2 se cumple la primera condición, mientras que en el valor $j = 6$ se cumple la segunda. En este caso no se cumplen las dos en ningún momento, pero si se hubiera dado ese caso, or hubiese sido cierto.

Operando en Bits

Vamos a ver la tabla de operación de `or` tanto con valores booleanos como con la representación en bits:

`or`		

|		

Operando en Bits

Vamos a ver la tabla de operación de `or` tanto con valores booleanos como con la representación en bits:

or	False	

|	0	

Operando en Bits

Vamos a ver la tabla de operación de `or` tanto con valores booleanos como con la representación en bits:

`or`	`False`	`True`

`|`	0	1

Operando en Bits

Vamos a ver la tabla de operación de `or` tanto con valores booleanos como con la representación en bits:

`or`	`False`	`True`
`False`		

`|`	0	1
0		

Operando en Bits

Vamos a ver la tabla de operación de `or` tanto con valores booleanos como con la representación en bits:

`or`	`False`	`True`
`False`		
`True`		

`|`	0	1
0		
1		

Operando en Bits

Vamos a ver la tabla de operación de `or` tanto con valores booleanos como con la representación en bits:

`or`	`False`	`True`
`False`	`False`	
`True`		

`|`	0	1
0	0	
1		

Operando en Bits

Vamos a ver la tabla de operación de `or` tanto con valores booleanos como con la representación en bits:

`or`	`False`	`True`
`False`	`False`	`True`
`True`		

`|`	0	1
0	0	1
1		

Operando en Bits

Vamos a ver la tabla de operación de `or` tanto con valores booleanos como con la representación en bits:

`or`	`False`	`True`
`False`	`False`	`True`
`True`	`True`	

`|`	0	1
0	0	1
1	1	

Operando en Bits

Vamos a ver la tabla de operación de `or` tanto con valores booleanos como con la representación en bits:

or	False	True
False	False	True
True	True	True

|	0	1
0	0	1
1	1	1

La Operación `not` (no lógico)

- Dadas una condición lógica a, la condición lógica `not` a será cierta cuando sea a sea falsa y falsa cuendo a sea verdadera.

La Operación not (no lógico)

- Dadas una condición lógica a, la condición lógica not a será cierta cuando sea a sea falsa y falsa cuendo a sea verdadera.
- Utilizando not con and y or podemos hacer sentencias más complejas, por ejemplo,

```
for j in range(7):
  if 2*j+1<6 and not 3*j-15>0:
    print j,2*j+1,3*j-15
```

nos dará los resultados

```
0 1 -15
1 3 -12
2 5 -9
```

que indican que para estos tres valores $j = 0, 1, 2$ la condición $2j + 1 < 6$ es cierta y la condición $3j - 15 > 0$ es falsa.

Operando bits en bloques

- Los operadores lógicos bit a bit se utilizan muy frecuentemente en criptografía para definir operaciones de cifrado y descifrado.

Operando bits en bloques

- Los operadores lógicos bit a bit se utilizan muy frecuentemente en criptografía para definir operaciones de cifrado y descifrado.
- Si a y b son números de k bits podemos definir operaciones bit a bit entre a y b dando como resultado un número de k bits c que tendrá en la cifra binaria i-ésima la operación correspondiente entre la cifra i-ésima del número a y la cifra i-ésima del número b.

Operando bits en bloques

- Los operadores lógicos bit a bit se utilizan muy frecuentemente en criptografía para definir operaciones de cifrado y descifrado.
- Si a y b son números de k bits podemos definir operaciones bit a bit entre a y b dando como resultado un número de k bits c que tendrá en la cifra binaria i-ésima la operación correspondiente entre la cifra i-ésima del número a y la cifra i-ésima del número b.
- Veamos un ejemplo de cada una de las operaciones.

La operación &

- Sea `a` el número decimal 10, que escrito en binario es 1010 y `b` el número decimal 13 que escrito en binario es 1101, entonces `c = a & b` se calcularía del siguiente modo:

La operación &

- Sea a el número decimal 10, que escrito en binario es 1010 y b el número decimal 13 que escrito en binario es 1101, entonces c = a & b se calcularía del siguiente modo:

a

La operación &

- Sea `a` el número decimal 10, que escrito en binario es 1010 y `b` el número decimal 13 que escrito en binario es 1101, entonces `c = a & b` se calcularía del siguiente modo:

a	1

La operación &

- Sea a el número decimal 10, que escrito en binario es 1010 y b el número decimal 13 que escrito en binario es 1101, entonces c = a & b se calcularía del siguiente modo:

a	1	0

La operación &

- Sea a el número decimal 10, que escrito en binario es 1010 y b el número decimal 13 que escrito en binario es 1101, entonces c = a & b se calcularía del siguiente modo:

a	1	0	1

La operación &

- Sea `a` el número decimal 10, que escrito en binario es 1010 y `b` el número decimal 13 que escrito en binario es 1101, entonces `c = a & b` se calcularía del siguiente modo:

a	1	0	1	0

La operación &

- Sea a el número decimal 10, que escrito en binario es 1010 y b el número decimal 13 que escrito en binario es 1101, entonces c = a & b se calcularía del siguiente modo:

a	1	0	1	0
b				

La operación &

- Sea `a` el número decimal 10, que escrito en binario es 1010 y `b` el número decimal 13 que escrito en binario es 1101, entonces `c = a & b` se calcularía del siguiente modo:

a	1	0	1	0
b	1			

La operación &

- Sea `a` el número decimal 10, que escrito en binario es 1010 y `b` el número decimal 13 que escrito en binario es 1101, entonces `c = a & b` se calcularía del siguiente modo:

a	1	0	1	0
b	1	1		

La operación &

- Sea `a` el número decimal 10, que escrito en binario es 1010 y `b` el número decimal 13 que escrito en binario es 1101, entonces `c = a & b` se calcularía del siguiente modo:

a	1	0	1	0
b	1	1	0	

La operación &

- Sea a el número decimal 10, que escrito en binario es 1010 y b el número decimal 13 que escrito en binario es 1101, entonces c = a & b se calcularía del siguiente modo:

a	1	0	1	0
b	1	1	0	1

La operación &

- Sea `a` el número decimal 10, que escrito en binario es 1010 y `b` el número decimal 13 que escrito en binario es 1101, entonces `c = a & b` se calcularía del siguiente modo:

`a`	1	0	1	0
`b`	1	1	0	1
`c = a & b`				

La operación &

- Sea `a` el número decimal 10, que escrito en binario es 1010 y `b` el número decimal 13 que escrito en binario es 1101, entonces `c = a & b` se calcularía del siguiente modo:

`a`	1	0	1	0
`b`	1	1	0	1
`c = a & b`	1			

La operación &

- Sea a el número decimal 10, que escrito en binario es 1010 y b el número decimal 13 que escrito en binario es 1101, entonces c = a & b se calcularía del siguiente modo:

a	1	0	1	0
b	1	1	0	1
c = a & b	1	0		

La operación &

- Sea `a` el número decimal 10, que escrito en binario es 1010 y `b` el número decimal 13 que escrito en binario es 1101, entonces `c = a & b` se calcularía del siguiente modo:

`a`	1	0	1	0
`b`	1	1	0	1
`c = a & b`	1	0	0	

La operación &

- Sea a el número decimal 10, que escrito en binario es 1010 y b el número decimal 13 que escrito en binario es 1101, entonces c = a & b se calcularía del siguiente modo:

a	1	0	1	0
b	1	1	0	1
c = a & b	1	0	0	0

La operación &

- Sea `a` el número decimal 10, que escrito en binario es 1010 y `b` el número decimal 13 que escrito en binario es 1101, entonces `c = a & b` se calcularía del siguiente modo:

a	1	0	1	0
b	1	1	0	1
c = a & b	1	0	0	0

- Cada columna individual se calcula haciendo la operación `and` que hemos visto anteriormente, el valor de c en decimal sería 8.

La operación &

- Sea a el número decimal 10, que escrito en binario es 1010 y b el número decimal 13 que escrito en binario es 1101, entonces c = a & b se calcularía del siguiente modo:

a	1	0	1	0
b	1	1	0	1
c = a & b	1	0	0	0

- Cada columna individual se calcula haciendo la operación `and` que hemos visto anteriormente, el valor de c en decimal sería 8.
- Si escribimos `print 10&13` nos dará el valor 8.

La operación |

- Sea `a` el número decimal 11, que escrito en binario es 1011 y `b` el número decimal 13 que escrito en binario es 1101, entonces `c = a | b` se calcularía del siguiente modo:

La operación |

- Sea `a` el número decimal 11, que escrito en binario es 1011 y `b` el número decimal 13 que escrito en binario es 1101, entonces `c = a | b` se calcularía del siguiente modo:

`a`

La operación |

- Sea `a` el número decimal 11, que escrito en binario es 1011 y `b` el número decimal 13 que escrito en binario es 1101, entonces `c = a | b` se calcularía del siguiente modo:

a	1

La operación |

- Sea `a` el número decimal 11, que escrito en binario es 1011 y `b` el número decimal 13 que escrito en binario es 1101, entonces `c = a | b` se calcularía del siguiente modo:

a	1	0

La operación |

- Sea a el número decimal 11, que escrito en binario es 1011 y b el número decimal 13 que escrito en binario es 1101, entonces c = a | b se calcularía del siguiente modo:

a	1	0	1

La operación |

- Sea `a` el número decimal 11, que escrito en binario es 1011 y `b` el número decimal 13 que escrito en binario es 1101, entonces `c = a | b` se calcularía del siguiente modo:

a	1	0	1	1

La operación |

- Sea a el número decimal 11, que escrito en binario es 1011 y b el número decimal 13 que escrito en binario es 1101, entonces c = a | b se calcularía del siguiente modo:

a	1	0	1	1
b				

La operación |

- Sea a el número decimal 11, que escrito en binario es 1011 y b el número decimal 13 que escrito en binario es 1101, entonces c = a | b se calcularía del siguiente modo:

a	1	0	1	1
b	1			

La operación |

- Sea `a` el número decimal 11, que escrito en binario es 1011 y `b` el número decimal 13 que escrito en binario es 1101, entonces `c = a | b` se calcularía del siguiente modo:

a	1	0	1	1
b	1	1		

La operación |

- Sea `a` el número decimal 11, que escrito en binario es 1011 y `b` el número decimal 13 que escrito en binario es 1101, entonces `c = a | b` se calcularía del siguiente modo:

a	1	0	1	1
b	1	1	0	

La operación |

- Sea `a` el número decimal 11, que escrito en binario es 1011 y `b` el número decimal 13 que escrito en binario es 1101, entonces `c = a | b` se calcularía del siguiente modo:

a	1	0	1	1
b	1	1	0	1

La operación |

- Sea `a` el número decimal 11, que escrito en binario es 1011 y `b` el número decimal 13 que escrito en binario es 1101, entonces `c = a | b` se calcularía del siguiente modo:

`a`	1	0	1	1
`b`	1	1	0	1
`c = a | b`				

La operación |

- Sea `a` el número decimal 11, que escrito en binario es 1011 y `b` el número decimal 13 que escrito en binario es 1101, entonces `c = a | b` se calcularía del siguiente modo:

`a`	1	0	1	1
`b`	1	1	0	1
`c = a | b`	1			

La operación |

- Sea a el número decimal 11, que escrito en binario es 1011 y b el número decimal 13 que escrito en binario es 1101, entonces c = a | b se calcularía del siguiente modo:

a	1	0	1	1
b	1	1	0	1
c = a | b	1	1		

La operación |

- Sea `a` el número decimal 11, que escrito en binario es 1011 y `b` el número decimal 13 que escrito en binario es 1101, entonces `c = a | b` se calcularía del siguiente modo:

`a`	1	0	1	1
`b`	1	1	0	1
`c = a | b`	1	1	1	

La operación |

- Sea `a` el número decimal 11, que escrito en binario es 1011 y `b` el número decimal 13 que escrito en binario es 1101, entonces `c = a | b` se calcularía del siguiente modo:

`a`	1	0	1	1
`b`	1	1	0	1
`c = a	b`	1	1	1	1

La operación |

- Sea a el número decimal 11, que escrito en binario es 1011 y b el número decimal 13 que escrito en binario es 1101, entonces c = a | b se calcularía del siguiente modo:

a	1	0	1	1
b	1	1	0	1
c = a | b	1	1	1	1

- Cada columna individual se calcula haciendo la operación or que hemos visto anteriormente, el valor de c en decimal sería 15.

La operación `|`

- Sea `a` el número decimal 11, que escrito en binario es 1011 y `b` el número decimal 13 que escrito en binario es 1101, entonces `c = a | b` se calcularía del siguiente modo:

a	1	0	1	1
b	1	1	0	1
c = a | b	1	1	1	1

- Cada columna individual se calcula haciendo la operación `or` que hemos visto anteriormente, el valor de c en decimal sería 15.
- Si escribimos `print 11|13` nos dará el valor 15.

La operación ^^ (o exclusivo)

- En criptografía es muy común una tercera operación llamada *o exclusivo* o *xor* que bit a bit es cierta si uno de los dos operandos es cierto y falsa si, o bien los dos son falsos o los dos son verdaderos. Es similar a `or`, pero en el caso en que ambos operandos sean verdaderos, `xor` es falso.

La operación ^^ (o exclusivo)

- En criptografía es muy común una tercera operación llamada *o exclusivo* o *xor* que bit a bit es cierta si uno de los dos operandos es cierto y falsa si, o bien los dos son falsos o los dos son verdaderos. Es similar a `or`, pero en el caso en que ambos operandos sean verdaderos, `xor` es falso.
- Sea `a` el número decimal 11, que escrito en binario es 1011 y `b` el número decimal 13 que escrito en binario es 1101, entonces el *o exclusivo* de ambos, `c = a ^^ b`, se calcularía del siguiente modo:

La operación ^^ (o exclusivo)

- En criptografía es muy común una tercera operación llamada *o exclusivo* o *xor* que bit a bit es cierta si uno de los dos operandos es cierto y falsa si, o bien los dos son falsos o los dos son verdaderos. Es similar a `or`, pero en el caso en que ambos operandos sean verdaderos, `xor` es falso.
- Sea `a` el número decimal 11, que escrito en binario es 1011 y `b` el número decimal 13 que escrito en binario es 1101, entonces el *o exclusivo* de ambos, `c = a ^^ b`, se calcularía del siguiente modo:

`a`

La operación ^^ (o exclusivo)

- En criptografía es muy común una tercera operación llamada *o exclusivo* o *xor* que bit a bit es cierta si uno de los dos operandos es cierto y falsa si, o bien los dos son falsos o los dos son verdaderos. Es similar a `or`, pero en el caso en que ambos operandos sean verdaderos, `xor` es falso.
- Sea `a` el número decimal 11, que escrito en binario es 1011 y `b` el número decimal 13 que escrito en binario es 1101, entonces el *o exclusivo* de ambos, `c = a ^^ b`, se calcularía del siguiente modo:

a	1

La operación ^^ (o exclusivo)

- En criptografía es muy común una tercera operación llamada *o exclusivo* o *xor* que bit a bit es cierta si uno de los dos operandos es cierto y falsa si, o bien los dos son falsos o los dos son verdaderos. Es similar a `or`, pero en el caso en que ambos operandos sean verdaderos, `xor` es falso.
- Sea `a` el número decimal 11, que escrito en binario es 1011 y `b` el número decimal 13 que escrito en binario es 1101, entonces el *o exclusivo* de ambos, `c = a ^^ b`, se calcularía del siguiente modo:

a			1	0

La operación ^^ (o exclusivo)

- En criptografía es muy común una tercera operación llamada *o exclusivo* o *xor* que bit a bit es cierta si uno de los dos operandos es cierto y falsa si, o bien los dos son falsos o los dos son verdaderos. Es similar a `or`, pero en el caso en que ambos operandos sean verdaderos, `xor` es falso.
- Sea `a` el número decimal 11, que escrito en binario es 1011 y `b` el número decimal 13 que escrito en binario es 1101, entonces el *o exclusivo* de ambos, `c = a ^^ b`, se calcularía del siguiente modo:

a	1	0	1

La operación ^^ (o exclusivo)

- En criptografía es muy común una tercera operación llamada *o exclusivo* o *xor* que bit a bit es cierta si uno de los dos operandos es cierto y falsa si, o bien los dos son falsos o los dos son verdaderos. Es similar a `or`, pero en el caso en que ambos operandos sean verdaderos, `xor` es falso.
- Sea `a` el número decimal 11, que escrito en binario es 1011 y `b` el número decimal 13 que escrito en binario es 1101, entonces el *o exclusivo* de ambos, `c = a ^^ b`, se calcularía del siguiente modo:

a	1	0	1	1

La operación ^^ (o exclusivo)

- En criptografía es muy común una tercera operación llamada *o exclusivo* o *xor* que bit a bit es cierta si uno de los dos operandos es cierto y falsa si, o bien los dos son falsos o los dos son verdaderos. Es similar a `or`, pero en el caso en que ambos operandos sean verdaderos, `xor` es falso.
- Sea `a` el número decimal 11, que escrito en binario es 1011 y `b` el número decimal 13 que escrito en binario es 1101, entonces el *o exclusivo* de ambos, `c = a ^^ b`, se calcularía del siguiente modo:

a	1	0	1	1
b				

La operación ^^ (o exclusivo)

- En criptografía es muy común una tercera operación llamada *o exclusivo* o *xor* que bit a bit es cierta si uno de los dos operandos es cierto y falsa si, o bien los dos son falsos o los dos son verdaderos. Es similar a `or`, pero en el caso en que ambos operandos sean verdaderos, `xor` es falso.
- Sea `a` el número decimal 11, que escrito en binario es 1011 y `b` el número decimal 13 que escrito en binario es 1101, entonces el *o exclusivo* de ambos, `c = a ^^ b`, se calcularía del siguiente modo:

a	1	0	1	1
b	1			

La operación ^^ (o exclusivo)

- En criptografía es muy común una tercera operación llamada *o exclusivo* o *xor* que bit a bit es cierta si uno de los dos operandos es cierto y falsa si, o bien los dos son falsos o los dos son verdaderos. Es similar a `or`, pero en el caso en que ambos operandos sean verdaderos, `xor` es falso.
- Sea `a` el número decimal 11, que escrito en binario es 1011 y `b` el número decimal 13 que escrito en binario es 1101, entonces el *o exclusivo* de ambos, `c = a ^^ b`, se calcularía del siguiente modo:

a	1	0	1	1
b	1	1		

La operación ^^ (o exclusivo)

- En criptografía es muy común una tercera operación llamada *o exclusivo* o *xor* que bit a bit es cierta si uno de los dos operandos es cierto y falsa si, o bien los dos son falsos o los dos son verdaderos. Es similar a `or`, pero en el caso en que ambos operandos sean verdaderos, `xor` es falso.
- Sea `a` el número decimal 11, que escrito en binario es 1011 y `b` el número decimal 13 que escrito en binario es 1101, entonces el *o exclusivo* de ambos, `c = a ^^ b`, se calcularía del siguiente modo:

a	1	0	1	1
b	1	1	0	

La operación ^^ (o exclusivo)

- En criptografía es muy común una tercera operación llamada *o exclusivo* o *xor* que bit a bit es cierta si uno de los dos operandos es cierto y falsa si, o bien los dos son falsos o los dos son verdaderos. Es similar a `or`, pero en el caso en que ambos operandos sean verdaderos, `xor` es falso.
- Sea `a` el número decimal 11, que escrito en binario es 1011 y `b` el número decimal 13 que escrito en binario es 1101, entonces el *o exclusivo* de ambos, `c = a ^^ b`, se calcularía del siguiente modo:

a	1	0	1	1
b	1	1	0	1

La operación ^^ (o exclusivo)

- En criptografía es muy común una tercera operación llamada *o exclusivo* o *xor* que bit a bit es cierta si uno de los dos operandos es cierto y falsa si, o bien los dos son falsos o los dos son verdaderos. Es similar a `or`, pero en el caso en que ambos operandos sean verdaderos, `xor` es falso.
- Sea `a` el número decimal 11, que escrito en binario es 1011 y `b` el número decimal 13 que escrito en binario es 1101, entonces el *o exclusivo* de ambos, `c = a ^^ b`, se calcularía del siguiente modo:

`a`	1	0	1	1
`b`	1	1	0	1
`c = a ^^ b`				

La operación ^^ (o exclusivo)

- En criptografía es muy común una tercera operación llamada *o exclusivo* o *xor* que bit a bit es cierta si uno de los dos operandos es cierto y falsa si, o bien los dos son falsos o los dos son verdaderos. Es similar a `or`, pero en el caso en que ambos operandos sean verdaderos, `xor` es falso.
- Sea `a` el número decimal 11, que escrito en binario es 1011 y `b` el número decimal 13 que escrito en binario es 1101, entonces el *o exclusivo* de ambos, `c = a ^^ b`, se calcularía del siguiente modo:

`a`	1	0	1	1
`b`	1	1	0	1
`c = a ^^ b`	0			

La operación ^^ (o exclusivo)

- En criptografía es muy común una tercera operación llamada *o exclusivo* o *xor* que bit a bit es cierta si uno de los dos operandos es cierto y falsa si, o bien los dos son falsos o los dos son verdaderos. Es similar a `or`, pero en el caso en que ambos operandos sean verdaderos, `xor` es falso.
- Sea `a` el número decimal 11, que escrito en binario es 1011 y `b` el número decimal 13 que escrito en binario es 1101, entonces el *o exclusivo* de ambos, `c = a ^^ b`, se calcularía del siguiente modo:

`a`	1	0	1	1
`b`	1	1	0	1
`c = a ^^ b`	0	1		

La operación ^^ (o exclusivo)

- En criptografía es muy común una tercera operación llamada *o exclusivo* o *xor* que bit a bit es cierta si uno de los dos operandos es cierto y falsa si, o bien los dos son falsos o los dos son verdaderos. Es similar a `or`, pero en el caso en que ambos operandos sean verdaderos, `xor` es falso.
- Sea `a` el número decimal 11, que escrito en binario es 1011 y `b` el número decimal 13 que escrito en binario es 1101, entonces el *o exclusivo* de ambos, `c = a ^^ b`, se calcularía del siguiente modo:

`a`	1	0	1	1
`b`	1	1	0	1
`c = a ^^ b`	0	1	1	

La operación ^^ (o exclusivo)

- En criptografía es muy común una tercera operación llamada *o exclusivo* o *xor* que bit a bit es cierta si uno de los dos operandos es cierto y falsa si, o bien los dos son falsos o los dos son verdaderos. Es similar a `or`, pero en el caso en que ambos operandos sean verdaderos, `xor` es falso.
- Sea `a` el número decimal 11, que escrito en binario es 1011 y `b` el número decimal 13 que escrito en binario es 1101, entonces el *o exclusivo* de ambos, `c = a ^^ b`, se calcularía del siguiente modo:

`a`	1	0	1	1
`b`	1	1	0	1
`c = a ^^ b`	0	1	1	0

La operación ^^ (o exclusivo)

- En criptografía es muy común una tercera operación llamada *o exclusivo* o *xor* que bit a bit es cierta si uno de los dos operandos es cierto y falsa si, o bien los dos son falsos o los dos son verdaderos. Es similar a `or`, pero en el caso en que ambos operandos sean verdaderos, `xor` es falso.
- Sea `a` el número decimal 11, que escrito en binario es 1011 y `b` el número decimal 13 que escrito en binario es 1101, entonces el *o exclusivo* de ambos, `c = a ^^ b`, se calcularía del siguiente modo:

a	1	0	1	1
b	1	1	0	1
c = a ^^ b	0	1	1	0

- Cada columna individual se calcula haciendo la operación *o exclusivo* a los bits correspondientes. El valor de c en decimal sería 6.

La operación ^^ (o exclusivo)

- En criptografía es muy común una tercera operación llamada *o exclusivo* o *xor* que bit a bit es cierta si uno de los dos operandos es cierto y falsa si, o bien los dos son falsos o los dos son verdaderos. Es similar a `or`, pero en el caso en que ambos operandos sean verdaderos, `xor` es falso.
- Sea `a` el número decimal 11, que escrito en binario es 1011 y `b` el número decimal 13 que escrito en binario es 1101, entonces el *o exclusivo* de ambos, `c = a ^^ b`, se calcularía del siguiente modo:

`a`	1	0	1	1
`b`	1	1	0	1
`c = a ^^ b`	0	1	1	0

- Cada columna individual se calcula haciendo la operación *o exclusivo* a los bits correspondientes. El valor de c en decimal sería 6.
- Si escribimos `print 11^^13` nos dará el valor 6.

La operación ^^ (o exclusivo)

- En criptografía es muy común una tercera operación llamada *o exclusivo* o *xor* que bit a bit es cierta si uno de los dos operandos es cierto y falsa si, o bien los dos son falsos o los dos son verdaderos. Es similar a `or`, pero en el caso en que ambos operandos sean verdaderos, `xor` es falso.
- Sea `a` el número decimal 11, que escrito en binario es 1011 y `b` el número decimal 13 que escrito en binario es 1101, entonces el *o exclusivo* de ambos, `c = a ^^ b`, se calcularía del siguiente modo:

`a`	1	0	1	1
`b`	1	1	0	1
`c = a ^^ b`	0	1	1	0

- Cada columna individual se calcula haciendo la operación *o exclusivo* a los bits correspondientes. El valor de c en decimal sería 6.
- Si escribimos `print 11^^13` nos dará el valor 6.
- En criptografía se suele representar mediante el símbolo $c = a \oplus b$.

La Operación `not` sobre Números

- Otra operación interesante es la negación de bits aplicado a grupos de bits.

La Operación `not` sobre Números

- Otra operación interesante es la negación de bits aplicado a grupos de bits.
- Es decir, cambiar en un número binario todos los ceros por unos y todos los unos por ceros.

La Operación `not` sobre Números

- Otra operación interesante es la negación de bits aplicado a grupos de bits.
- Es decir, cambiar en un número binario todos los ceros por unos y todos los unos por ceros.
- El problema es que desde el punto de vista numérico podemos poner ceros a la izquierda y el número no cambia, es decir, los números binarios 001101 y 1101 son el mismo, por tanto su negación podría ser tanto 110010 y 0010.

La Operación `not` sobre Números

- Otra operación interesante es la negación de bits aplicado a grupos de bits.
- Es decir, cambiar en un número binario todos los ceros por unos y todos los unos por ceros.
- El problema es que desde el punto de vista numérico podemos poner ceros a la izquierda y el número no cambia, es decir, los números binarios `001101` y `1101` son el mismo, por tanto su negación podría ser tanto `110010` y `0010`.
- Para garantizar que no existan dudas al respecto, podemos utilizar la operación *o exclusivo* con la constante `111...111`.

La Operación `not` sobre Números

- Otra operación interesante es la negación de bits aplicado a grupos de bits.
- Es decir, cambiar en un número binario todos los ceros por unos y todos los unos por ceros.
- El problema es que desde el punto de vista numérico podemos poner ceros a la izquierda y el número no cambia, es decir, los números binarios `001101` y `1101` son el mismo, por tanto su negación podría ser tanto `110010` y `0010`.
- Para garantizar que no existan dudas al respecto, podemos utilizar la operación *o exclusivo* con la constante `111...111`.
- Así tenemos `0b111111 ^^ 0b001101 == 0b110010` y también `0b1111 ^^ 0b1101 == 0b0010`

La Notación Hexadecimal

- Aparte de la base 10 y la base 2, una base muy importante en criptografía es la base 16.

La Notación Hexadecimal

- Aparte de la base 10 y la base 2, una base muy importante en criptografía es la base 16.
- Para poder escribir los números en base 16 necesitamos 16 símbolos diferentes que representen las cifras.

La Notación Hexadecimal

- Aparte de la base 10 y la base 2, una base muy importante en criptografía es la base 16.
- Para poder escribir los números en base 16 necesitamos 16 símbolos diferentes que representen las cifras.
- Lo que se suele hacer es representar las cifras del 0 al 9 con los números habituales y a partir del 9 utilizar los siguientes símbolos: $\mathtt{a} = 10$, $\mathtt{b} = 11$, $\mathtt{c} = 12$, $\mathtt{d} = 13$, $\mathtt{e} = 14$ y $\mathtt{f} = 15$

La Notación Hexadecimal

- Aparte de la base 10 y la base 2, una base muy importante en criptografía es la base 16.
- Para poder escribir los números en base 16 necesitamos 16 símbolos diferentes que representen las cifras.
- Lo que se suele hacer es representar las cifras del 0 al 9 con los números habituales y a partir del 9 utilizar los siguientes símbolos: $\mathtt{a} = 10$, $\mathtt{b} = 11$, $\mathtt{c} = 12$, $\mathtt{d} = 13$, $\mathtt{e} = 14$ y $\mathtt{f} = 15$
- Así el número hexadecimal

$$\mathtt{a3b0} = \mathtt{a} \cdot 16^3 + 3 \cdot 16^2 + \mathtt{b} \cdot 16^1 + 0 \cdot 16^0 =$$

$$10 \cdot 16^3 + 3 \cdot 16^2 + 11 \cdot 16^1 + 0 \cdot 16^0 = 41904$$

Representación Hexadecimal y Binaria I

- Una de las principales ventajas de la representación hexadecimal es que la conversión entre binario y hexadecimal es muy sencilla.

Representación Hexadecimal y Binaria I

- Una de las principales ventajas de la representación hexadecimal es que la conversión entre binario y hexadecimal es muy sencilla.
- La razón es que $16 = 2^4$ es una potencia de 2, de esta forma cada cuatro cifras binarias se pueden agrupar para formar una cifra hexadecimal.

Representación Hexadecimal y Binaria I

- Una de las principales ventajas de la representación hexadecimal es que la conversión entre binario y hexadecimal es muy sencilla.
- La razón es que $16 = 2^4$ es una potencia de 2, de esta forma cada cuatro cifras binarias se pueden agrupar para formar una cifra hexadecimal.
- Por ejemplo, el número binario

$$10011100 =$$

$$\underbrace{1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4}_{(1\cdot 2^3+0\cdot 2^2+0\cdot 2^1+1\cdot 2^0)\cdot 16} + 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 =$$

$$9 \cdot 16 + 12 = 9 \cdot 16 + c = 9c$$

Representación Hexadecimal y Binaria II

- Podemos hacer este cálculo directamente mirando las cifras binarias en grupos de 4 cifras cada grupo de cuatro cifras debe convertirse en una cifra entre 0 y 15.

$$\underbrace{1001}_{9}\underbrace{1100}_{12=c} = 9c$$

Representación Hexadecimal y Binaria II

- Podemos hacer este cálculo directamente mirando las cifras binarias en grupos de 4 cifras cada grupo de cuatro cifras debe convertirse en una cifra entre 0 y 15.

$$\underbrace{1001}_{9}\underbrace{1100}_{12=c} = 9c$$

- En el caso de que el número de cifras binarias no sea múltiplo de 4 debemos completar con ceros a la izquierda.

$$110100110 = \underbrace{0001}_{1}\underbrace{1010}_{10=a}\underbrace{0110}_{6} = 1a6$$

Representación Hexadecimal y Binaria II

- Podemos hacer este cálculo directamente mirando las cifras binarias en grupos de 4 cifras cada grupo de cuatro cifras debe convertirse en una cifra entre 0 y 15.

$$\underbrace{1001}_{9}\underbrace{1100}_{12=\mathrm{c}} = 9c$$

- En el caso de que el número de cifras binarias no sea múltiplo de 4 debemos completar con ceros a la izquierda.

$$110100110 = \underbrace{0001}_{1}\underbrace{1010}_{10=\mathrm{a}}\underbrace{0110}_{6} = 1a6$$

- Podemos comprobar el resultado en sage utilizando

```
0b110100110 == 0x1a6
```

Representación Hexadecimal y Binaria II

- Podemos hacer este cálculo directamente mirando las cifras binarias en grupos de 4 cifras cada grupo de cuatro cifras debe convertirse en una cifra entre 0 y 15.

$$\underbrace{1001}_{9}\,\underbrace{1100}_{12=c} = 9c$$

- En el caso de que el número de cifras binarias no sea múltiplo de 4 debemos completar con ceros a la izquierda.

$$110100110 = \underbrace{0001}_{1}\,\underbrace{1010}_{10=a}\,\underbrace{0110}_{6} = 1a6$$

- Podemos comprobar el resultado en sage utilizando

```
0b110100110 == 0x1a6
```

- Los prefijos `0b` y `0x` nos permiten diferenciar números binarios y hexadecimales en sage

Representación Hexadecimal y Binaria II

- Podemos hacer este cálculo directamente mirando las cifras binarias en grupos de 4 cifras cada grupo de cuatro cifras debe convertirse en una cifra entre 0 y 15.

$$\underbrace{1001}_{9}\,\underbrace{1100}_{12=c} = 9c$$

- En el caso de que el número de cifras binarias no sea múltiplo de 4 debemos completar con ceros a la izquierda.

$$110100110 = \underbrace{0001}_{1}\,\underbrace{1010}_{10=a}\,\underbrace{0110}_{6} = 1a6$$

- Podemos comprobar el resultado en sage utilizando

```
0b110100110 == 0x1a6
```

- Los prefijos `0b` y `0x` nos permiten diferenciar números binarios y hexadecimales en sage
- Aunque podamos utilizar sage o cualquier programa informático, es conveniente conocer de memoria las 16 cifras hexadecimales escritas en binario, puesto que esta conversión se realizará contínuamente. La representación hexadecimal se usa como un tipo de representación binaria compacta.

Tabla de Conversión

0000 = 0	0001 = 1	0010 = 2	0011 = 3
0100 = 4	0101 = 5	0110 = 6	0111 = 7
1000 = 8	1001 = 9	1010 = a	1011 = b
1100 = c	1101 = d	1110 = e	1111 = f

Particiones de Listas y Rotaciones de Bits I

- Una operación muy habitual en criptografía es la rotación de bits. Vamos a ver cómo se puede hacer esta operaciún utilizando particiones de listas.

Particiones de Listas y Rotaciones de Bits I

- Una operación muy habitual en criptografía es la rotación de bits. Vamos a ver cómo se puede hacer esta operaciún utilizando particiones de listas.
- Supongamos que tenemos la lista `L = range(10) = [0,1,2,3,4,5,6,7,8,9]`. Una rotación a la izquierda de esta lista consisitiría en mover el elemento que está en la posición n a la $n-1$, es decir, el 9 iría a la posición del 8, el 8 a la del 7, y así sucesivamente hasta llegar al 1 que iría a la posición del 0. Finalmente el 0 iría a la posición que ha dejado libre el 9.

Particiones de Listas y Rotaciones de Bits I

- Una operación muy habitual en criptografía es la rotación de bits. Vamos a ver cómo se puede hacer esta operaciún utilizando particiones de listas.
- Supongamos que tenemos la lista `L = range(10) = [0,1,2,3,4,5,6,7,8,9]`. Una rotación a la izquierda de esta lista consisitiría en mover el elemento que está en la posición n a la $n-1$, es decir, el 9 iría a la posición del 8, el 8 a la del 7, y así sucesivamente hasta llegar al 1 que iría a la posición del 0. Finalmente el 0 iría a la posición que ha dejado libre el 9.
- Es decir, `[0,1,2,3,4,5,6,7,8,9]` se transforma en `[1,2,3,4,5,6,7,8,9,0]`.

Particiones de Listas y Rotaciones de Bits I

- Una operación muy habitual en criptografía es la rotación de bits. Vamos a ver cómo se puede hacer esta operaciún utilizando particiones de listas.
- Supongamos que tenemos la lista `L = range(10) = [0,1,2,3,4,5,6,7,8,9]`. Una rotación a la izquierda de esta lista consisitiría en mover el elemento que está en la posición n a la $n-1$, es decir, el 9 iría a la posición del 8, el 8 a la del 7, y así sucesivamente hasta llegar al 1 que iría a la posición del 0. Finalmente el 0 iría a la posición que ha dejado libre el 9.
- Es decir, `[0,1,2,3,4,5,6,7,8,9]` se transforma en `[1,2,3,4,5,6,7,8,9,0]`.
- Esta operación se puede hacer con sublistas. Si `L` es una lista, `L[k:]` es la sublista de todos los elementos a partir de la posición `k` y `L[:k]` es la sublista con las posiciones estrictamente menores que `k`. Por lo tanto:

$$\mathtt{L} = [\underbrace{0}_{\mathtt{L[:1]}}, \underbrace{1,2,3,4,5,6,7,8,9}_{\mathtt{L[1:]}}] \mapsto [\underbrace{1,2,3,4,5,6,7,8,9}_{\mathtt{L[1:]}}, \underbrace{0}_{\mathtt{L[:1]}}] = \mathtt{L}[1:] + \mathtt{L}[:1]$$

Particiones de Listas y Rotaciones de Bits II

- Este mismo método nos permite rotar a la izquierda más de una posición. Supongamos que queremos trasladar los tres últimos elementos del inicio de la lista al final (rotar tres posiciones). Entonces

$$\mathtt{L} = [\underbrace{0,1,2}_{\mathtt{L[:3]}},\underbrace{3,4,5,6,7,8,9}_{\mathtt{L[3:]}}] \mapsto [\underbrace{3,4,5,6,7,8,9}_{\mathtt{L[3:]}},\underbrace{0,1,2}_{\mathtt{L[:3]}}] = \mathtt{L}[3:] + \mathtt{L}[:3]$$

Particiones de Listas y Rotaciones de Bits II

- Este mismo método nos permite rotar a la izquierda más de una posición. Supongamos que queremos trasladar los tres últimos elementos del inicio de la lista al final (rotar tres posiciones). Entonces

$$\mathtt{L} = [\underbrace{0,1,2}_{\mathtt{L[:3]}}, \underbrace{3,4,5,6,7,8,9}_{\mathtt{L[3:]}}] \mapsto [\underbrace{3,4,5,6,7,8,9}_{\mathtt{L[3:]}}, \underbrace{0,1,2}_{\mathtt{L[:3]}}] = \mathtt{L}[3:] + \mathtt{L}[:3]$$

- En general, la rotación a la izquierda de `k` posiciones se puede hacer como `L[k:]+L[:k]`.

Particiones de Listas y Rotaciones de Bits II

- Este mismo método nos permite rotar a la izquierda más de una posición. Supongamos que queremos trasladar los tres últimos elementos del inicio de la lista al final (rotar tres posiciones). Entonces

$$\mathtt{L} = [\underbrace{0,1,2}_{\mathtt{L}[:3]},\underbrace{3,4,5,6,7,8,9}_{\mathtt{L}[3:]}] \mapsto [\underbrace{3,4,5,6,7,8,9}_{\mathtt{L}[3:]},\underbrace{0,1,2}_{\mathtt{L}[:3]}] = \mathtt{L}[3:] + \mathtt{L}[:3]$$

- En general, la rotación a la izquierda de `k` posiciones se puede hacer como `L[k:]+L[:k]`.
- Si en lugar de rotar una posición a la izquierda lo queremos hacer a la derecha, es decir, `L = range(10) = [0,1,2,3,4,5,6,7,8,9]` queremos convertirlo en `[9,0,1,2,3,4,5,6,7,8]` podemos hacerlo también dándonos cuenta de que rotar a la derecha una posición es lo mismo que rotar a la izquierda nueve posiciones. En general, rotar a la derecha `k` posiciones es lo mismo que rotar a la izquierda `n-k` posiciones siendo `n` la longitud de la lista, por lo que podemos usar `L[n-k:]+L[:n-k]`.

Particiones de Listas y Rotaciones de Bits III

- Esto lo aplicaremos especialmente a rotaciones de bits. Supongamos que tenemos `B = [1,0,1,1,0,1,0,0]` y queremos rotarlo una posición a la izquierda, entonces tendríamos `B[1:]+B[:1] = [0,1,1,0,1,0,0,1]`.

Particiones de Listas y Rotaciones de Bits III

- Esto lo aplicaremos especialmente a rotaciones de bits. Supongamos que tenemos `B = [1,0,1,1,0,1,0,0]` y queremos rotarlo una posición a la izquierda, entonces tendríamos `B[1:]+B[:1] = [0,1,1,0,1,0,0,1]`.
- Muchos sistemas criptográficos se referirán a los bits de un número ordenados con el bit menos significativo a la derecha, mientras que las listas correspondientes a los bits de un número en sage nos vienen con el bit menos significativo en la posición 0, es decir, a la izquierda. Veámoslo con este ejemplo:

```
B = [1,0,1,1,0,1,0,0]
print ZZ(B,2),bin(ZZ(B,2))
```

Particiones de Listas y Rotaciones de Bits III

- Esto lo aplicaremos especialmente a rotaciones de bits. Supongamos que tenemos `B = [1,0,1,1,0,1,0,0]` y queremos rotarlo una posición a la izquierda, entonces tendríamos `B[1:]+B[:1] = [0,1,1,0,1,0,0,1]`.
- Muchos sistemas criptográficos se referirán a los bits de un número ordenados con el bit menos significativo a la derecha, mientras que las listas correspondientes a los bits de un número en sage nos vienen con el bit menos significativo en la posición 0, es decir, a la izquierda. Veámoslo con este ejemplo:

```
B = [1,0,1,1,0,1,0,0]
print ZZ(B,2),bin(ZZ(B,2))
```

- El número que nos escribirá es 45 que en binario es `0b101101` o escrito con 8 cifras 00101101.

Particiones de Listas y Rotaciones de Bits III

- Esto lo aplicaremos especialmente a rotaciones de bits. Supongamos que tenemos `B = [1,0,1,1,0,1,0,0]` y queremos rotarlo una posición a la izquierda, entonces tendríamos `B[1:]+B[:1] = [0,1,1,0,1,0,0,1]`.
- Muchos sistemas criptográficos se referirán a los bits de un número ordenados con el bit menos significativo a la derecha, mientras que las listas correspondientes a los bits de un número en sage nos vienen con el bit menos significativo en la posición 0, es decir, a la izquierda. Veámoslo con este ejemplo:

```
B = [1,0,1,1,0,1,0,0]
print ZZ(B,2),bin(ZZ(B,2))
```

- El número que nos escribirá es 45 que en binario es `0b101101` o escrito con 8 cifras 00101101.
- Si el sistema criptográfico representa los números con el bit menos significativo a la derecha y nos piden, por ejemplo, rotar a la derecha k posiciones, podemos utilizar la lista de cifras binarias, pero debemos tener en cuenta que sage nos ha cambiado el orden por lo que, o bien tendremos que cambiar el orden usando `reverse` o bien tendremos que hacer la rotación hacia el otro lado.

Particiones de Listas y Rotaciones de Bits III

- Esto lo aplicaremos especialmente a rotaciones de bits. Supongamos que tenemos `B = [1,0,1,1,0,1,0,0]` y queremos rotarlo una posición a la izquierda, entonces tendríamos `B[1:]+B[:1] = [0,1,1,0,1,0,0,1]`.
- Muchos sistemas criptográficos se referirán a los bits de un número ordenados con el bit menos significativo a la derecha, mientras que las listas correspondientes a los bits de un número en sage nos vienen con el bit menos significativo en la posición 0, es decir, a la izquierda. Veámoslo con este ejemplo:

```
B = [1,0,1,1,0,1,0,0]
print ZZ(B,2),bin(ZZ(B,2))
```

- El número que nos escribirá es 45 que en binario es `0b101101` o escrito con 8 cifras 00101101.
- Si el sistema criptográfico representa los números con el bit menos significativo a la derecha y nos piden, por ejemplo, rotar a la derecha k posiciones, podemos utilizar la lista de cifras binarias, pero debemos tener en cuenta que sage nos ha cambiado el orden por lo que, o bien tendremos que cambiar el orden usando `reverse` o bien tendremos que hacer la rotación hacia el otro lado.
- Puede parecer un poco confuso y si tienes experiencia en programación seguro que te has encontrado alguna vez con lo de *big endian* y *little endian*, un verdadero caos en la transmisión de datos y fuente de errores muy comunes en programación.

Cuidado

Es fundamental tener muy claro cómo representa el programa los números en relación con sus bits y cómo lo hace nuestro sistema criptográfico. Todos los estándares nos lo precisarán claramente y debemos prestar mucha atención a ello.