

---

**Decodificar Base64 (2)****T77863\_es**

---

(Este problema utiliza la función `char_to_base64` del problema "Decodificar Base64 (1)".)

Se trata de hacer un programa que, dada una secuencia de caracteres en la entrada (uno de los 64 que representan los dígitos de base 64), la **decodifique en sus bytes**. La decodificación funciona de la siguiente manera:

1. Primero, para cada grupo de 4 caracteres (o cuarteto) de la entrada  $c_1, c_2, c_3$  y  $c_4$ , los transformamos en sus dígitos correspondientes  $d_1, d_2, d_3, d_4$ , usando la función `char_to_base64`.
2. Después, con los 4 dígitos  $d_i$  reconstruimos el natural  $x$  aplicando la fórmula:

$$x = ((d_1 \cdot 64 + d_2) \cdot 64 + d_3) \cdot 64 + d_4$$

3. A continuación, reinterpretemos  $x$  en base 256, y extraemos las cifras, que ahora son 3:  $B_1, B_2$ , y  $B_3$ . El proceso es totalmente análogo a extraer las cifras de un número en base 10, pero en base 256. Lo que habremos hecho es calcular los dígitos  $B_i$  de la fórmula siguiente:

$$x = (B_1 \cdot 256 + B_2) \cdot 256 + B_3$$

(Hay que recordar que el proceso de extracción de las cifras trabajado en PRO1 produce las cifras al **revés**, comenzando por  $B_3$ )

4. Finalmente, mostramos  $B_1, B_2$  y  $B_3$  como números naturales por pantalla, en este orden.

La codificación en base 64 siempre tiene un número de caracteres múltiplo de 4, pero justo al final de la secuencia puede haber '=' o '==', que nos dice que el número de bytes del último cuarteto son 2 o 1, y no 3:

Si el último cuarteto tiene algún '=' al final:

- Si tiene '==': el número de bytes será 1; asignamos  $d_3 = 0$  y  $d_4 = 0$ , decodificamos según los pasos anteriores, pero solo mostramos  $B_1$  por pantalla.
- Si tiene '=': el número de bytes es 2; asignamos  $d_4 = 0$ , decodificamos según los pasos anteriores, pero solo mostramos  $B_1$  y  $B_2$  por pantalla.

**Entrada**

La entrada consiste en varios casos, donde cada caso es una secuencia de caracteres base 64 en la misma línea y con un punto al final.

**Salida**

La salida debe ser una línea para cada caso con los bytes como números naturales. Antes de cada byte, incluido el primero, debe haber un espacio.

## Observación

- Este problema tiene como centros de interés la **corrección** y la **legibilidad**. En particular, se valorará que el programa utilice funciones para evitar repetición y separar las diversas tareas.
- Si tenéis la función `char_to_base64` y el Juez os la ha aceptado, usadla directamente. Si no, copiad la siguiente definición, (y añadid `#include <algorithm>`):

```
int char_to_base64(char c) {  
    static char _syms[65] =  
        "ABCDEFGHIJKLMNOPQRSTUVWXYZ"  
        "abcdefghijklmnopqrstuvwxyz"  
        "0123456789+/";  
    return std::find(_syms, _syms + 64, c) - _syms;  
}
```

### Ejemplo de entrada 1

```
AAAA.  
AQEB.  
AgIC.  
////.  
AA==.  
AAA=  
AQ==.  
AQE=  
ZGRkZA==.  
CgAUAB4=  
////AAAA.
```

### Ejemplo de salida 1

```
0 0 0  
1 1 1  
2 2 2  
255 255 255  
0  
0 0  
1  
1 1  
100 100 100 100  
10 0 20 0 30  
255 255 255 0 0 0
```

### Ejemplo de entrada 2

```
LA==.  
Jg==.  
lg==.  
Sg==.  
3Q==.  
Mg==.  
Rw==.  
Ng==.  
BQ==.  
TA==.  
og==.
```

### Ejemplo de salida 2

```
44  
38  
150  
74  
221  
50  
71  
54  
5  
76  
162
```

### Ejemplo de entrada 3

```
9qE=  
xpM=  
b9M=  
DUk=  
CH0=  
rPA=  
kSk=  
wV8=  
bQg=  
ZvY=.
```

### Ejemplo de salida 3

```
246 161  
198 147  
111 211  
13 73  
8 125  
172 240  
145 41  
193 95  
109 8  
102 246
```

#### Ejemplo de entrada 4

u1Zi.  
q6nx.  
NJFe.  
tjy1.  
1Bq2.  
/eRh.  
q0A/.  
2K1L.  
ANVh.  
eg+n.

#### Ejemplo de salida 4

187 86 98  
171 169 241  
52 145 94  
182 60 181  
212 26 182  
253 228 97  
171 64 63  
216 173 75  
0 213 97  
122 15 167

#### Información del problema

Autor : Pau Fernández

Generación : 2025-10-30 16:19:54

© *Jutge.org*, 2006–2025.

<https://jutge.org>