

Placa rotativa

Historia

La alcaldía de «Sucre» estrenó una restricción vehicular **rotativa** para los días hábiles: cada día de lunes a viernes, los autos de un grupo de placas no pueden circular por la ciudad. Los sábados y domingos todos los autos circulan libremente. Para que la medida no castigue siempre a los mismos, cada semana la restricción se corre un día.

Bob tiene un auto y quiere saber cuántos días se quedó sin poder usarlo desde que empezó la norma.

Tarea

Cada auto pertenece a un **grupo** según el último dígito de su placa:

Último dígito	1 o 2	3 o 4	5 o 6	7 u 8	9 o 0
Grupo	0	1	2	3	4

La norma entró en vigencia un lunes, que es el día **1**. Los días se agrupan en semanas de **7** días que empiezan en lunes: los días **1** a **7** son la semana **0**, los días **8** a **14** son la semana **1**, y así sucesivamente.

Cada día hábil no circula exactamente un grupo:

- En la **semana 0**, el lunes no circula el grupo **0**, el martes el grupo **1**, el miércoles el grupo **2**, el jueves el grupo **3** y el viernes el grupo **4**.
- Cada semana siguiente la restricción **se corre un día**: en la semana **1** el lunes no circula el grupo **1**, el martes el grupo **2**, ..., y el viernes el grupo **0**. En la semana **2** el lunes no circula el grupo **2**, y así sucesivamente, volviendo al grupo **0** después del grupo **4**.

Los sábados y domingos no hay restricción para ningún grupo.

El auto de Bob tiene la placa número P . Calcula en cuántos de los **primeros D días** de vigencia (los días **1** a D) su auto no pudo circular.

Implementación

Debes implementar la siguiente función.

En C++, en el archivo `placa.cpp`:

```
long long dias_restringidos(long long P, long long D);
```

En Python, en el archivo `placa.py`:

```
def dias_restringidos(P: int, D: int) -> int:
```

- **P**: el número de la placa del auto de Bob (solo importa su último dígito).
- **D**: la cantidad de días de vigencia que se cuentan, desde el día **1** hasta el día **D**.
- La función debe devolver la cantidad de días, entre los primeros **D** días de vigencia, en que el auto de Bob no pudo circular.
- El grader llama a esta función una sola vez por cada caso de prueba.

Tu código **no debe leer de la entrada estándar ni escribir en la salida estándar**: de eso se encarga el grader. Envía solo el archivo `placa.cpp` o `placa.py`.

Ejemplos

Ejemplo 1

```
dias_restringidos(1, 7)
```

La función debe devolver **1**. El dígito **1** es del grupo **0**, que no circula el lunes de la semana **0** (el día **1**). En los primeros **7** días ese es el único día restringido.

Ejemplo 2

```
dias_restringidos(1, 14)
```

La función debe devolver **2**. Además del lunes de la semana **0**, en la semana **1** (días **8** a **14**) el viernes no circula el grupo **0**, que es el grupo de Bob. Ese viernes es el día **12**. Son **2** días restringidos.

Ejemplo 3

```
dias_restringidos(7, 30)
```

La función debe devolver **4**. El dígito **7** es del grupo **3**. No circula el jueves de la semana **0**, el miércoles de la semana **1**, el martes de la semana **2** y el lunes de la

semana 3. En la semana 4 le tocaría el viernes, pero los 30 días solo llegan hasta el martes de esa semana (los días 29 y 30 son su lunes y su martes). Son 4.

Ejemplo 4

```
dias_restringidos(1234, 1000000000000000000)
```

La función debe devolver 142857142857142857. El dígito 4 es del grupo 1.

Restricciones

- $0 \leq P \leq 9999$
- $1 \leq D \leq 10^{18}$

Subtareas

Recuerda que el día 1 es lunes: el día D es lunes si $D \bmod 7 = 1$, viernes si $D \bmod 7 = 5$, sábado si $D \bmod 7 = 6$ y domingo si $D \bmod 7 = 0$.

Subtarea 1 (17 puntos)

- $D \leq 10^6$

Subtarea 2 (22 puntos)

- El día D es viernes, sábado o domingo.

Subtarea 3 (24 puntos)

- El día D es lunes.

Subtarea 4 (37 puntos)

- Sin restricciones adicionales.

Puntuación

Cada subtarea tiene varios casos de prueba. Obtienes los puntos de una subtarea solo si tu solución resuelve correctamente **todos** sus casos dentro del límite de tiempo y de memoria. Tu puntaje en el problema es la suma de los puntos de las subtareas que resuelves.

Puedes enviar varias soluciones: para cada subtarea se toma el mejor resultado entre todos tus envíos.

Grader de ejemplo

En la sección de archivos del problema encontrarás el grader de ejemplo (`grader.cpp` y `grader.py`), un archivo base para tu solución y los ejemplos. El grader que usa el juez es distinto, pero lee y escribe en el mismo formato.

Compilar y ejecutar en C++ (en la carpeta `cpp/`):

```
g++ -std=c++20 -O2 -o placa grader.cpp placa.cpp
./placa < ../examples/0-01.in
```

Ejecutar en Python (en la carpeta `py/`):

```
python3 grader.py < ../examples/0-01.in
```

El grader de ejemplo lee la entrada en este formato:

- línea 1: dos enteros `P D`.

Luego llama a `dias_restringidos` y escribe en la salida, en una línea, el número que devuelve.

Entrada y salida del grader para el ejemplo 1:

Entrada	Salida
1 7	1

Re2

Historia

Redo, un fanático de los sistemas numéricos, un día, mientras jugaba con los números, encontró una mezcla peculiar entre las bases **2** y **10**.

Tomó la representación en base **2** de un número entero no negativo n , la interpretó como si fuera un número en base **10** y le restó el número original n . Haciendo esto para $n = 0, 1, 2, 3, 4, \dots$ encontró la siguiente serie:

0, 0, 8, 8, 96, ...

Por ejemplo, para $n = 4$ la representación en base **2** es **100**; leída en base **10** es el número cien, y $100 - 4 = 96$.

Tarea

El primer término de la serie corresponde a $n = 0$, el segundo a $n = 1$ y, en general, el k -ésimo término corresponde a $n = k - 1$.

Redo ahora quiere hallar una fórmula que permita obtener cualquier término de la serie. Para comenzar, quiere determinar la diferencia entre el k -ésimo término y el $(k - 1)$ -ésimo término.

Ayuda a Redo a encontrar el valor exacto de dicha diferencia. Ten en cuenta que la respuesta puede ser muy grande y **no caber en un entero de 64 bits**: por eso la función la devuelve como texto.

Implementación

Debes implementar la siguiente función.

En C++, en el archivo `redos.cpp`:

```
std::string diferencia(long long k);
```

En Python, en el archivo `redos.py`:

```
def diferencia(k: int) -> str:
```

- k : la posición del término de la serie (k -ésimo término, que corresponde a $n = k - 1$).

- La función debe devolver la diferencia exacta entre el k -ésimo término y el $(k - 1)$ -ésimo término, escrita en base 10 como texto (por ejemplo "88" o "0"), sin espacios ni ceros a la izquierda.
- El grader llama a esta función una sola vez por cada caso de prueba.

Tu código **no debe leer de la entrada estándar ni escribir en la salida estándar**: de eso se encarga el grader. Envía solo el archivo `redos.cpp` o `redos.py`.

Ejemplos

Ejemplo 1

```
diferencia(5)
```

La función debe devolver "88". Debemos calcular la diferencia entre el 5.º término y el 4.º término. El 5.º término corresponde a $n = 4$ ($100 - 4 = 96$) y el 4.º término corresponde a $n = 3$ ($11 - 3 = 8$). Entonces la respuesta es $96 - 8 = 88$.

Ejemplo 2

```
diferencia(4)
```

La función debe devolver "0". El 4.º término corresponde a $n = 3$ ($11 - 3 = 8$) y el 3.º término corresponde a $n = 2$ ($10 - 2 = 8$). Entonces la respuesta es $8 - 8 = 0$.

Restricciones

- $2 \leq k \leq 10^{18}$

Subtareas

Subtarea 1 (7 puntos)

- k es un número par.
- $2 \leq k \leq 10^3$

Subtarea 2 (9 puntos)

- Al dividir k entre 4, el residuo es 3.
- $2 \leq k \leq 10^3$

Subtarea 3 (14 puntos)

- $2 \leq k \leq 10^3$

Subtarea 4 (6 puntos)

- $k - 1$ es una potencia de 10; es decir, $k = 10^m + 1$ para algún entero $m \geq 1$ (por ejemplo, $k = 11$, $k = 101$ o $k = 1001$).

Subtarea 5 (26 puntos)

- $k - 1$ es una potencia de 2; es decir, $k = 2^t + 1$ para algún entero $t \geq 1$ (por ejemplo, $k = 3$, $k = 5$ o $k = 9$).

Subtarea 6 (38 puntos)

- Sin restricciones adicionales.

Puntuación

Cada subtarea tiene varios casos de prueba. Obtienes los puntos de una subtarea solo si tu solución resuelve correctamente **todos** sus casos dentro del límite de tiempo y de memoria. Tu puntaje en el problema es la suma de los puntos de las subtareas que resuelves.

Puedes enviar varias soluciones: para cada subtarea se toma el mejor resultado entre todos tus envíos.

Grader de ejemplo

En la sección de archivos del problema encontrarás el grader de ejemplo (`grader.cpp` y `grader.py`), un archivo base para tu solución y los ejemplos. El grader que usa el juez es distinto, pero lee y escribe en el mismo formato.

Compilar y ejecutar en C++ (en la carpeta `cpp/`):

```
g++ -std=c++20 -O2 -o redos grader.cpp redos.cpp
./redos < ../examples/0-01.in
```

Ejecutar en Python (en la carpeta `py/`):

```
python3 grader.py < ../examples/0-01.in
```

El grader de ejemplo lee la entrada en este formato:

- línea 1: el entero k .

Luego llama a `diferencia(k)` y escribe en la salida, en una sola línea, el texto que devuelve la función.

Entrada y salida del grader para el ejemplo 1:

Entrada	Salida
5	88

La leyenda de Zelma

Historia

La batalla final ha llegado, y el gran héroe Enlace está en una carrera contra el tiempo. El temible Rey del Mal, Ganandor, está a punto de despertar de su sueño letárgico. Para sellarlo nuevamente, Enlace necesita armar la mítica reliquia conocida como el "Triángulo del Poder".

Enlace ha encontrado el pedestal sagrado donde debe poner los fragmentos de poder. Enlace ya tiene reunidos los 3 fragmentos, cada uno con un número mágico grabado en sus tres puntas.

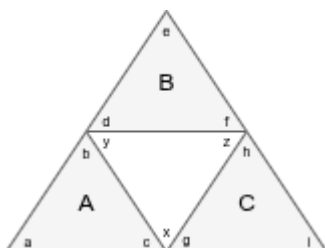
Al llegar al pedestal, nota que en el centro del pedestal se encuentra un fragmento que yace anclado en el medio. No puede sacarlo de ahí para intercambiarlo con los demás, pero sí puede rotarlo sobre su propio eje.

Los 3 fragmentos restantes están sueltos. Enlace puede colocarlos en cualquiera de las 3 ranuras exteriores (intercambiarlos de posición) y también puede rotarlos.

Para que el Triángulo del Poder se active, las uniones mágicas deben resonar perfectamente. En las tres uniones internas del gran triángulo convergen exactamente las puntas de 3 fragmentos distintos. Para que el ensamblaje sea válido, los 3 valores que chocan en cada una de esas tres uniones deben ser exactamente iguales.

Si Enlace logra armarlo correctamente, la reliquia generará un nivel de poder igual a la suma de las 3 puntas exteriores. Como Enlace quiere asegurar la derrota de Ganandor, necesita que le ayudes a saber si es posible armar el Triángulo del Poder y, en caso de que se pueda, calcular el poder máximo que puede liberar.

Tarea



En la imagen, **A** es la ranura de abajo a la izquierda, **B** la ranura de arriba y **C** la ranura de abajo a la derecha; el fragmento central está "de cabeza" (con una punta hacia abajo).

- Los valores de cada fragmento se dan siempre **en sentido horario**.
- Para un fragmento suelto colocado en una ranura, el orden es: **punta inferior izquierda, punta superior, punta inferior derecha**. Por ejemplo, en la ranura A son a, b, c ; en la ranura B, d, e, f ; en la ranura C, g, h, i .
- Para el fragmento central el orden es: **punta inferior, punta superior izquierda, punta superior derecha**, es decir x, y, z .
- Rotar un fragmento desplaza sus valores de forma circular conservando el sentido horario: un fragmento dado como $p q r$ puede quedar como $p q r$, como $r p q$ o como $q r p$. Esto vale también para el fragmento central: $x y z$ puede quedar como $z x y$ o como $y z x$. Los fragmentos **no** se pueden voltear (dar la vuelta como en un espejo).

Las tres uniones internas son:

- Unión izquierda: punta superior de A , punta inferior izquierda de B y punta superior izquierda del centro ($b = d = y$ en la imagen).
- Unión derecha: punta inferior derecha de B , punta superior de C y punta superior derecha del centro ($f = h = z$).
- Unión inferior: punta inferior derecha de A , punta inferior izquierda de C y punta inferior del centro ($c = g = x$).

Las tres puntas exteriores son la punta inferior izquierda de A , la punta superior de B y la punta inferior derecha de C (en la imagen, $a + e + i$).

Un **ensamblaje** consiste en colocar cada fragmento suelto en una ranura distinta (A, B o C), elegir la rotación de cada fragmento suelto y elegir la rotación del fragmento central. Es **válido** si en cada una de las tres uniones internas los 3 valores son iguales, y su **poder** es la suma de las tres puntas exteriores.

Dados los 3 fragmentos sueltos y el fragmento central, determina el poder máximo entre todos los ensamblajes válidos, o indica que no existe ninguno.

Implementación

Debes implementar la siguiente función.

En C++, en el archivo `zelma.cpp` (que incluye `zelma.h`):

```
int poder_maximo(std::vector<std::vector<int>> sueltos, std::vector<int>
centro);
```

En Python, en el archivo `zelma.py`:

```
def poder_maximo(sueltos: list[list[int]], centro: list[int]) -> int:
```

- **suelos**: una lista de 3 fragmentos suelos; cada uno es una lista de 3 enteros con sus valores en sentido horario tal como quedarían en una ranura sin rotarlo (punta inferior izquierda, punta superior, punta inferior derecha). Es decir, **suelos[0]** es *a b c*, **suelos[1]** es *d e f* y **suelos[2]** es *g h i*.
- **centro**: una lista de 3 enteros con los valores del fragmento central en el orden punta inferior, punta superior izquierda, punta superior derecha, es decir *x y z*.
- La función debe devolver el poder máximo de un ensamblaje válido. Si no existe ningún ensamblaje válido, debe devolver la constante **IMPOSIBLE**, que vale -1000000 . En C++ está definida en **zelma.h**; en Python, al inicio del archivo base **zelma.py** (no borres esa línea, o devuelve directamente -1000000). Ningún ensamblaje válido puede tener ese poder, porque el poder siempre está entre -3000 y 3000 .
- El grader llama a esta función una sola vez por cada caso de prueba.

Tu código **no debe leer de la entrada estándar ni escribir en la salida estándar**: de eso se encarga el grader. Envía solo el archivo **zelma.cpp** o **zelma.py**.

Ejemplos

Ejemplo 1

```
poder_maximo([[8, 3, 10], [5, 15, 3], [20, 5, 8]], [5, 3, 8])
```

La función debe devolver **45**. Una forma de lograr el poder máximo es dejar el fragmento central sin rotar ($x = 5, y = 3, z = 8$) y colocar:

- En la ranura *A*, el segundo fragmento rotado como **15 3 5**.
- En la ranura *B*, el primer fragmento rotado como **3 10 8**.
- En la ranura *C*, el tercer fragmento rotado como **5 8 20**.

Las uniones quedan: izquierda $3 = 3 = 3$, derecha $8 = 8 = 8$ e inferior $5 = 5 = 5$. El poder es $15 + 10 + 20 = 45$.

Ejemplo 2

```
poder_maximo([[1, 2, 3], [4, 5, 6], [7, 8, 9]], [10, 11, 12])
```

La función debe devolver **IMPOSIBLE** (es decir, -1000000). Todos los valores son distintos, así que nunca pueden coincidir tres puntas en una misma unión.

Restricciones

- $-1000 \leq$ valor de cada punta ≤ 1000
- **sue**ltos tiene exactamente 3 fragmentos, y cada fragmento (incluido **cent**ro) tiene exactamente 3 valores.

Subtareas

Subtarea 1 (11 puntos)

- Las tres puntas del fragmento central tienen el mismo valor ($x = y = z$).

Subtarea 2 (16 puntos)

- Un hada mágica te ha ayudado rotando los fragmentos: ninguno (incluido el central) necesita ser rotado, solo falta decidir en qué ranura va cada fragmento suelto. Se garantiza que existe un ensamblaje válido de esta forma y que logra el poder máximo.
- $0 \leq$ valor de cada punta ≤ 1000 .

Subtarea 3 (19 puntos)

- Si es posible armar el Triángulo del Poder, se puede hacer sin cambiar de ranura ningún fragmento suelto: el primero en la ranura **A**, el segundo en la ranura **B** y el tercero en la ranura **C**. Solo puede hacer falta rotar fragmentos, incluido el central. Puede que no exista ningún ensamblaje válido.

Subtarea 4 (21 puntos)

- Se garantiza que existe al menos un ensamblaje válido.

Subtarea 5 (33 puntos)

- Sin restricciones adicionales.

Puntuación

Cada subtarea tiene varios casos de prueba. Obtienes los puntos de una subtarea solo si tu solución resuelve correctamente **todos** sus casos dentro del límite de tiempo y de memoria. Tu puntaje en el problema es la suma de los puntos de las subtareas que resuelves.

Puedes enviar varias soluciones: para cada subtarea se toma el mejor resultado entre todos tus envíos.

Grader de ejemplo

En la sección de archivos del problema encontrarás el grader de ejemplo (`grader.cpp` y `grader.py`), un archivo base para tu solución (`zelma.cpp` y `zelma.py`), el archivo `zelma.h` y los ejemplos. El grader que usa el juez es distinto, pero lee y escribe en el mismo formato.

Compilar y ejecutar en C++ (en la carpeta `cpp/`):

```
g++ -std=c++20 -O2 -o zelma grader.cpp zelma.cpp
./zelma < ../examples/0-01.in
```

Ejecutar en Python (en la carpeta `py/`):

```
python3 grader.py < ../examples/0-01.in
```

El grader de ejemplo lee la entrada en este formato:

- líneas 1 a 3: los 3 fragmentos sueltos, uno por línea, con sus 3 valores separados por un espacio (`sueルト[0]` , `sueルト[1]` y `sueルト[2]`);
- línea 4: los 3 valores del fragmento central (`centro`).

Luego llama a `poder_maximo` y escribe en la salida una línea con el número que devuelve, o la palabra `IMPOSIBLE` si devuelve la constante `IMPOSIBLE` .

Entrada y salida del grader para el ejemplo 1:

Entrada	Salida
8 3 10 5 15 3 20 5 8 5 3 8	45

Somarithmo faltante

Historia

En algún lugar del vasto universo de la imaginación existen entes denominados **Somarithmos**, que viven formados en una única sucesión. Hay dos tipos de Somarithmos: los números enteros positivos, denominados **Zahlen+**, y los ceros, denominados **Oudens**.

La sucesión de Somarithmos es infinita hacia la izquierda y **termina siempre en 1**. Escrita de izquierda a derecha, su parte final es:

```
... 0 0 0 0 0 5 0 0 0 0 4 0 0 0 3 0 0 2 0 1
```

La sucesión es tan larga que nadie puede verla completa: con un telescopio solo se observa un tramo de Somarithmos consecutivos, que puede estar en cualquier parte de la sucesión. Un día, mientras observabas uno de esos tramos, uno de los Somarithmos se perdió y en su lugar quedó un hueco. ¿Puedes descubrir quién falta?

Tarea

La regla que forma la sucesión es la siguiente: **cada Zahlen+ k está precedido inmediatamente por exactamente k Oudens**, y a la izquierda de esos k Oudens está el Zahlen+ $k + 1$. Dicho de otra forma, si se lee la sucesión de derecha a izquierda, se obtiene **1, 0, 2, 0, 0, 3, 0, 0, 0, 4, ...**: cada número k va seguido de k ceros y después aparece $k + 1$.

Se observa un tramo de K elementos consecutivos de la sucesión. El tramo puede empezar y terminar en cualquier posición; en particular, puede incluir o no el **1** final. Por ejemplo, estos son tramos de **7** elementos:

```
0 3 0 0 2 0 1
0 0 0 0 5 0 0
```

En el primero se ven tres Zahlen+ (**3**, **2** y **1**) y cuatro Oudens. Observa que solo se ve uno de los tres Oudens que preceden al **3**, porque el tramo empieza cortado. En el segundo se ven el **5**, cuatro de los cinco Oudens que lo preceden y dos de los cuatro Oudens que lo siguen.

Exactamente uno de los K elementos del tramo se perdió. Se garantiza que en el tramo se ve **al menos un Zahlen+**. Tu tarea es determinar el ente faltante, que puede ser un Ouden o un Zahlen+.

Implementación

Debes implementar la siguiente función.

En C++, en el archivo `somarithmo.cpp`:

```
int faltante(std::vector<int> a);
```

En Python, en el archivo `somarithmo.py`:

```
def faltante(a: list[int]) -> int:
```

- `a`: los K elementos del tramo observado, en orden de izquierda a derecha. El elemento perdido aparece como -1 ; todos los demás valen 0 (un Ouden) o un entero positivo (un Zahlen+).
- La función debe devolver 0 si el elemento perdido es un Ouden, o el valor k si el elemento perdido es el Zahlen+ k .
- El grader llama a esta función una sola vez por cada caso de prueba.

Tu código **no debe leer de la entrada estándar ni escribir en la salida estándar**: de eso se encarga el grader. Envía solo el archivo `somarithmo.cpp` o `somarithmo.py`.

Ejemplos

Ejemplo 1

```
faltante([3, 0, -1, 2, 0, 1])
```

La función debe devolver 0 . El tramo es `3 0 0 2 0 1`, el final de la sucesión: falta uno de los dos Oudens que preceden al `2`.

Ejemplo 2

```
faltante([-1, 0, 0, 0, 3])
```

La función debe devolver 4 . El `3` está precedido por exactamente tres Oudens, y justo antes de ellos está el `4`. Observa que el elemento perdido puede ser mayor que todos los que se ven.

Ejemplo 3

```
faltante([0, 5, 0, 0, 0, 0, -1, 0, 0])
```

La función debe devolver **4**. El tramo es **0 5 0 0 0 0 4 0 0**: el **4** está precedido por cuatro Oudens, y antes de ellos está el **5**.

Ejemplo 4

```
faltante([0, 0, 0, 234, 0, 0, -1, 0])
```

La función debe devolver **0**. Al **234** le siguen **233** Oudens antes de llegar al **233**, así que el elemento perdido es uno de esos Oudens.

Ejemplo 5

```
faltante([2, 0, -1])
```

La función debe devolver **1**. El tramo es **2 0 1**, el final de la sucesión, y el que se perdió es el **1** final.

Restricciones

- $2 \leq K \leq 200\,000$
- Los elementos de **a** distintos de -1 coinciden exactamente con un tramo de K elementos consecutivos de la sucesión de Somarithmos.
- Exactamente un elemento de **a** vale -1 .
- Al menos un elemento de **a** es un Zahlen+ (es mayor que **0**).
- Todos los valores del tramo, incluido el del elemento perdido, son menores o iguales que 10^9 .

Subtareas

Subtarea 1 (12 puntos)

- Todos los valores del tramo, incluido el del elemento perdido, son menores o iguales que **500**.

Subtarea 2 (3 puntos)

- El elemento perdido está inmediatamente a la izquierda o inmediatamente a la derecha de un Zahlen+ visible.

Subtarea 3 (18 puntos)

- El elemento perdido es un Zahlen+ (la respuesta es mayor que 0).

Subtarea 4 (28 puntos)

- Hay al menos un Zahlen+ visible a la izquierda del elemento perdido.

Subtarea 5 (39 puntos)

- Sin restricciones adicionales.

Puntuación

Cada subtarea tiene varios casos de prueba. Obtienes los puntos de una subtarea solo si tu solución resuelve correctamente **todos** sus casos dentro del límite de tiempo y de memoria. Tu puntaje en el problema es la suma de los puntos de las subtareas que resuelves.

Puedes enviar varias soluciones: para cada subtarea se toma el mejor resultado entre todos tus envíos.

Grader de ejemplo

En la sección de archivos del problema encontrarás el grader de ejemplo (`grader.cpp` y `grader.py`), un archivo base para tu solución y los ejemplos. El grader que usa el juez es distinto, pero lee y escribe en el mismo formato.

Compilar y ejecutar en C++ (en la carpeta `cpp/`):

```
g++ -std=c++20 -O2 -o somarithmo grader.cpp somarithmo.cpp
./somarithmo < ../examples/0-01.in
```

Ejecutar en Python (en la carpeta `py/`):

```
python3 grader.py < ../examples/0-01.in
```

El grader de ejemplo lee la entrada en este formato:

- línea 1: el entero K ;
- línea 2: los K elementos del tramo, separados por un espacio, donde el elemento perdido se escribe con el carácter `*` (el grader lo convierte en -1 antes de llamar a tu función).

Luego llama a `faltante` y escribe en la salida `Ouden` si la función devuelve `0`, o `Zahlen+{k}` si devuelve un valor k positivo (la palabra `Zahlen+` seguida del valor entre llaves, sin espacios; por ejemplo, `Zahlen+{4}`).

Entrada y salida del grader para el ejemplo 1:

Entrada	Salida
6 3 0 * 2 0 1	0uden