Soluciones entrenos OIE

Gritos en Halloween

Enlace al problema: https://www.aceptaelreto.com/problem/statement.php?id=574

En Halloween todos los que se disfrazan intentan asustar a cualquier incauto con el que se cruzan. El problema es que, después de cruzarse con unos cuantos dando gritos, la gente se acostumbra y ya nadie pasa miedo.

La consecuencia directa es una espiral de gritos cada vez más largos. El primero da un grito, el segundo un grito más largo, el tercero otro aún más largo, y así sucesivamente.

La asociación Unidos Contra el Miedo ha identificado el patrón que siguen en su pueblo los "fantasmas" de la Fundación por la Diversión de la Infancia. El primer disfrazado simplemente grita "BUH". El segundo disfrazado suelta el mismo grito que el anterior, después un nuevo BUH pero con una U más, y después vuelve a repetir el grito del anterior. El tercer disfrazado vuelve a hacer lo mismo. El resultado es que los primeros gritos son los siguientes (los espacios se ponen por claridad, pero no hay separación real):

Orden	*Grito*
1	BUH
2	BUH BUUH BUH
3	BUHBUUHBUH BUUUH BUHBUUHBUH
4	BUHBUUHBUHBUUUHBUHBUUHBUH BUUUUH BUHBUUHBUHBUUUHBUHBUUHBUH

¿Qué grito dará el disfrazado número n? Como el grito puede ser bastante largo, nos conformamos con preguntar por una letra concreta.

Input Format

La entrada estará formada por distintos casos de prueba, cada uno en una línea.

Cada caso de prueba consiste en dos números: el primero n que indica el número que ocupa la persona disfrazada por la que preguntamos, y el segundo, k, el número de letra concreta del grito que queremos conocer (entre 1 y la longitud del grito).

Constraints

Se garantiza que el grito por el que se pregunta tendrá menos de 2^{63} letras.

Output format

Por cada caso de prueba se escribirá, en una línea independiente, la letra concreta por la que se pregunta. Recuerda que los gritos reales tienen todas sus letras seguidas.

Solución

En primer lugar, comenzaremos hablando de las soluciones incorrectas. Podemos sentirnos tentados a generar primero la cadena correspondiente al grito por el que se nos pregunta, y después mirar en la posición k. Sin embargo, esta es una solución ineficiente tanto en tiempo como en espacio. Por un lado, concatenar dos strings en C++ es una operación lenta. Por otro lado, cada caracter ocupa 1 Byte, con lo que si nuestra cadena tiene 2^{63} caracteres, ocupará $2^{63}B \approx 8 \cdot 10^9 Gb$, muy por encima de los $4Mb$ permitidos.

Examinemos ahora el proceso de construcción de la cadena, que está dividida en 3 partes. Por ejemplo, para la de orden 3:

$$\underbrace{\text{BUHBUUHBUH}}_{\text{parte inicial}}\ \underbrace{\text{BUUUH}}_{\text{parte intermedia}}\ \underbrace{\text{BUHBUUHBUH}}_{\text{parte final}}$$

Su construcción viene dada por las ecuaciones:

$parte\ inicial(n) = parte\ final(n) = parte\ inicial(n-1) + parte\ intermedia(n-1) + parte\ final(n-1)$
$parte\ intermedia(n) = BU...UH$ (con n letras U)

siendo + la concatenación de cadenas. Además, para $n = 1$ tendríamos:

$parte\ inicial(1) = parte\ final(1) =$ “” (cadena vacía)

Para resolver el problema vamos a hacer uso de estas ecuaciones. Para ello, dado un orden n y una posición k, tenemos:

- k pertenece a la **parte intermedia**.
  En este caso, simplemente tenemos que ver si es la primera letra (en cuyo caso devolvemos B), la última (devolviendo H) o cualquier otra (devolviendo U).
- k pertenece a la **parte inicial**.
  Tenemos que mirar a qué letra se corresponde esa misma posición en la cadena de orden $n-1$.
- k pertenece a la **parte final**.
  Tenemos que mirar a qué letra se corresponde en la cadena de orden $n-1$. Sin embargo, tenemos que cambiar la posición que miramos, para que caiga en la letra adecuada dentro de la cadena de orden inferior. Tendremos que restar para ello el tamaño de la parte inicial e intermedia del grito de orden n.

orden $n-1$ BUHBUUHBUH

orden n BUHBUUHBUH BUUUH BUHBUUHBUH

Por lo tanto, solo nos queda conocer el tamaño de cada parte de la cadena para cierto orden. Para ello, simplemente tenemos que adaptar las ecuaciones anteriormente mencionadas:

$tam\ parte\ intermedia(n) = n + 2$
$tam\ parte\ inicial(n) = tam\ parte\ final(n) = 2\ tam\ parte\ inicial(n-1) + n + 2$

Y como caso base:

$parte\ inicial(1) = parte\ final(1) = 0$

Simplemente tenemos que precalcular estos valores para todos los órdenes posibles antes de la ejecución.

Complejidad temporal: por un lado, precalculamos los tamaños de los gritos, con complejidad en $O(n)$. Por otro lado, en cada caso tendremos que hacer una recursión con profundidad a lo sumo n, con lo que de nuevo la complejidad está en $O(n)$.

Complejidad espacial: para el precálculo tendremos que almacenar n valores, por lo que está en $O(n)$. Para cada llamada no necesitamos almacenar nada, así que está en $O(1)$.

Nótese que, como en cada valor aproximadamente se duplica el tamaño de la cadena, $n \leq 63$ (puesto que nos dicen que nunca se preguntará por una cadena de más de 2^{63} letras).

Código

C++

```cpp
#include <iostream>
#include <vector>
using namespace std;

using ll = long long int;
using vll = vector<ll>;

//Lo inicializamos con una posición llena extra para que el índice del array
//  coincida con el tamaño de la palabra de dicho orden
vll tam_parte_inicial = {0, 0};

char recursion(int n, ll k) {
        //Tenemos 3 casos:
        //  1. Nos encontramos en la parte inicial de la palabra:
        if (k <= tam_parte_inicial[n]) {
                return recursion(n - 1, k);
        }
        //  2. Nos encontramos en la parte central
        else if (k <= tam_parte_inicial[n] + n + 2) {
                int indice = k - tam_parte_inicial[n];
                if (indice == 1) return 'B';
                else if (indice == n + 2) return 'H';
                else return 'U';
        }
        //  3. Nos encontramos en la parte final
        else {
                return recursion(n - 1, k - (tam_parte_inicial[n] + n + 2));
        }
}

bool resolver() {
        int n; ll k;
        cin >> n >> k;
        if (!cin) return false;
        cout << recursion(n, k) << '\n';
        return true;
}
```

```
int main() {
        //En primer lugar precalculamos los tamaños de cada una de las
        // partes de la palabra dependiendo del orden de la misma
        for (int i = 2; i <= 63; ++i) {
                tam_parte_inicial.push_back(2 * tam_parte_inicial[i - 1] + i + 1);
        }

        while (resolver()) {}
        return 0;
}
```