Soluciones OIFem 2020
Entrenamiento 2

Tres pasos para adelante, dos pasos para atrás

Teoría

- Recursión

Solución

El propio problema nos describe ya una solución recursiva, traduciendo directamente el enunciado.

Código

C++ Solución Recursiva

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

int collatz(long long k) {
  // Case base.
  if (k <= 1) return 1;
  // Recursión sobre un problema de menor tamaño, caso "k" par.
  if (k % 2 == 0) return collatz(k / 2) + 1;
  // Recursión sobre un problema de menor tamaño, caso "k" impar.
  return collatz(3 * k + 1) + 1;
}

int main() {
  int n;
  cin >> n;
  for (int i = 0; i < n; ++i) {
    long long k;
    cin >> k;
    cout << collatz(k) << '\n';
  }
  return 0;
}
```

C++ Solución Iterativa

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

int collatz(long long k) {
  int res = 1;
  while (k > 1) {
    if (k % 2 == 0) {
      k /= 2;
    } else {
      k = 3 * k + 1;
    }
    ++res;
  }
  return res;
}

int main() {
  int n;
  cin >> n;
  for (int i = 0; i < n; ++i) {
    long long k;
    cin >> k;
    cout << collatz(k) << '\n';
  }
  return 0;
}
```

César y sus cartas secretas

Teoría

- Uso de funciones
- Equivalencia entre `char` e `int`
- Uso del operador módulo, `%`

Solución

Por simplicidad, el enunciado ya nos indica que el espacio no se codificará, con lo que sólo tenemos que preocuparnos por las 26 letras del alfabeto inglés (o el castellano, si evitamos la ñ y diacríticas —tildes, diéresis, etc.—). El segundo concepto básico es ajustar cada carácter al rango 0..25. Para esto, démonos cuenta de que en ASCII los caracteres tienen valores numéricos sucesivos, con lo que `c - 'a'` nos dará tal número entre 0 y 25 para cualquier carácter minúsculo de entrada `c`. Con estos ingredientes, ya podemos sumar el entero que representa la "clave" de cifrado/descrifrado `n`, asegurándonos de aplicarle módulo 26. A este resultado le añadimos el valor de `'a'` para deshacer la operacion `c - 'a'` anterior.

Código

C++ solución larga

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

void encode(string & str, int n) {
  for (char c : str) {
    if (c == ' ') {
      cout << c;
    } else {
      int i = ((c - 'a') + n) % 26 + 'a';
      cout << char(i);
    }
  }
  cout << '\n';
}

void decode(string & str, int n) {
  for (char c : str) {
    if (c == ' ') {
      cout << c;
    } else {
      int i = ((c - 'a') - n) % 26;
      if (i < 0) i += 26;  // ¡atencion a modulos negativos!
      cout << char(i + 'a');
    }
  }
  cout << '\n';
}
```

```cpp
int main() {
  int n;
  cin >> n;
  cin.get();

  string str;
  getline(cin, str);

  encode(str, n);

  cin >> n;
  cin.get();

  getline(cin, str);

  decode(str, n);

  return 0;
};
```

C++ Solución corta

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

void encriptar(int n, string & texto) {
    n = (n+26)%26; // cuidado con los módulos negativos
    for (int c = 0; c < (int) texto.length(); c++) {
        if (texto[c] != ' ')
            texto[c] = (texto[c] - 'a' + n)%26 + 'a';
    }
}

int main() {
    int n;
    cin >> n;
    string texto;
    cin.ignore();
    getline(cin, texto);
    encriptar(n, texto);
    cout << texto << "\n";
    cin >> n;
    cin.ignore();
    getline(cin, texto);
    encriptar(-n, texto); // encriptar -n es lo mismo que decriptar n
    cout << texto;
    return 0;
}
```

Embaldosando el château

Teoría

- Recusión
- Algoritmo de Euclides para el máximo común divisor.

Solución

El problema se reduce al cálculo del máximo común divisor (comunmente denominado GCD, de sus siglas en inglés —*greatest common divisor*—). Sin embargo, implementarlo de manera *naïve* examinando números uno a uno y utilizando el módulo es demasiado lento (y el juez fallará para estos casos con "timeout").

Lo más sencillo es implementar el conocido como "algoritmo de Euclides".

Código

C++

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

// Usamos el método de Euclides (link arriba)
long long int gcd(long long int a, long long int b){
    if (!b) // comprobar si !b es lo mismo que comprobar si b == 0
        return a;
    return gcd(b, a%b);
}

int main() {
    int n;
    long long int a, b;
    cin >> n;

    for (int i = 0; i < n; i++){
        cin >> a >> b;
        cout << gcd(a,b) <<'\n';
    }

    return 0;
}
```

Números perfectos

Teoría

- Manipulación numérica
- Bucles
- Módulo

Solución

El proceso por fuerza bruta consiste simplemente en iterar desde 2..n-1 y, para cada candidato a divisor, comprobar si lo es mediante el módulo, añadiendolo a una variable acumulador que al final se comprueba si es igual al propio número de entrada n. Esto tiene un coste $O(n)$.

Sin embargo, podemos ahorrarnos $\sqrt{n}$ iteraciones si nos damos cuenta de que todo divisor menor que la raíz cuadrada de un número tiene un "simétrico" mayor que la raíz cuadrada, que se obtiene mediante n/i. Véanse los comentarios en el código de la solución para más detalles y un ejemplo.

Código

C++

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

bool perfect(long long int n) {
  // El 1 siempre es divisor de todo entero.
  long long sum = 1;
  // Busquemos los divisores de n. Nótese nunca hace falta buscar más allá de la
  // raíz cuadrada de n.
  for (long long i = 2; i * i <= n; i++) {
    if (n % i == 0) {  // Hemos encontrado un divisor.
      // Nótese que, ya que i sólo va desde 2 hasta int(sqrt(n)), tenemos que
      // tener en cuenta divisores > int(sqrt(n)). Observemos que todo divisor
      // menor que la raíz cuadrada tiene un "simétrico" mayor que la raíz
      // cuadrada, que se obtiene mediante n/i.
      //
      // Por ejemplo, para 28, un número perfecto con divisores 2, 4, 7 y 14 y
      // una int(sqrt(28)) == 5, 28 / 2 = 14, 28 / 4 = 7.
      if (i * i < n) sum += (i + n / i);
      else sum += i;
    }
  }
  // Si la suma de los divisores de n es igual al propio n, y n no es es 1 (ya
  // que excluimos el propio número de la lista de divisores a considerar), el
  // número es perfecto.
  if (sum == n && n != 1) return true;
  return false;
}
```

```cpp
int main() {
  long long n;
  cin >> n;
  int count = 0;
  for (long long i = 0; i < n; ++i) {
    if (perfect(i)) ++count;
  }
  cout << count << endl;

  return 0;
}
```

Tres dedos en cada mano

Teoría

- Bases numéricas
- Conversión de tipos (typecasting)

Solución

Para convertir un número de base 10 a base 6, iremos dividiendo el número original entre 6 y guardando los restos en una string. Esa string de restos conforma la solución final, solo que "dada la vuelta". Por lo tanto, invertimos la string.

Código

C++

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

string convert(int base10) {
        if (!base10) {
                // si base10 == 0, devolvemos "0"
                return "0";
        }
        string respuesta = "";
        while(base10) {
                // vamos iterativamente guardando los restos en la respuesta hasta
                ↪  que base10 sea igual a 0
                respuesta += to_string(base10 % 6);
                base10 /= 6;
        }
        reverse(respuesta.begin(), respuesta.end()); // invertimos la respuesta
        ↪  para que esté en el orden correcto
        return respuesta;
}

int main() {
        int N, num;
        cin >> N;
        while(N--) {
                cin >> num;
                cout << convert(num) << "\n";
        }
        return 0;
}
```

Poniendo la mesa

Teoría

- Recursión múltiple

Solución

Resolveremos este problema usando recursión múltiple. Si es tu turno, se invocará a la función `miTurno()`, que calculará la resta correspondiente en copas (1 si son impares o 2 si son pares) e invocará a la función `turnoHermano()`, que restará una copa al total e invocará a `miTurno()`, siguiendo esta doble invocación hasta tener un número no positivo de copas. Aquí llegamos al caso base de la recusión: 0 viajes.

Código

C++

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

int miTurno(int copas); // para poder hacer la recursión múltiple

int turnoHermano(int copas) {
        if (copas <= 0) {
                // si no hay un número positivo de copas, no hacen falta viajes
                return 0;
        }
        return 1 + miTurno(copas-1);
}

int miTurno(int copas) {
        if (copas <= 0) {
                // si no hay un número positivo de copas, no hacen falta viajes
                return 0;
        }
        if (copas % 2 == 0) {
                // si es mi turno y hay un número par, llevo dos copas
                return 1 + turnoHermano(copas-2);
        }
        // si no, llevo una copa
        return 1 + turnoHermano(copas-1);
}

int main() {
        int copas;
        while(true) {
                cin >> copas;
                if (!copas) break;
                cout << turnoHermano(copas) << "\n";
        }
        return 0;
}
```

Cluedo

Solución

La clave para resolver este problema con el menor número de llamadas a `Theory()` es el llevar en todo momento la cuenta de los asesinos, armas y lugares ya descartados.

Esto se puede hacer de una manera muy sencilla, con tres ints. Si el int asesino es igual a 3, esto significa que el número del asesino será igual o mayor a 3. Cuando empezamos la partida, solo podemos asegurar que el asesino, arma y lugar serán mayores o iguales a 1, que es el mínimo.

Por lo tanto, empezamos lanzando esa teoría y guardando la pista que recibimos. Si esta en cualquier momento es 0, devolvemos nuestros valores actuales de los tres ints, ya que tienen que ser los correctos. Sin embargo, si la pista nos dice que uno de los tres está mal, incrementaremos el valor de ese int, ya que podemos asegurar que no es ni ese valor ni ninguno de los inferiores, al haberlos probado anteriormente. De esta forma, conseguiremos la solución en el mínimo número de intentos posible.

Código

C++

```cpp
#include "grader.h"
#include "cluedo.h"

void Solve() {
        int asesino = 1; // declaramos e inicializamos los tres ints
        int arma = 1;
        int lugar = 1;
        int pista;
        while(true) {
                pista = Theory(asesino, lugar, arma); // lanzamos nuestra teoría
                if (pista == 0) {
                    // ya sabemos la respuesta
                    return;
                }
                else if (pista == 1) {
                    // el asesino estaba mal -> probamos con valores mayores
                    asesino++;
                }
                else if (pista == 2) {
                    // el lugar estaba mal -> probamos con valores mayores
                    lugar++;
                }
                else {
                    // el arma estaba mal -> probamos con valores mayores
                    arma++;
                }
        }
}
```