





A4 – Estructuras iterativas: bucles

1. Uno de los algoritmos más antiguos es el que calcula el máximo común divisor mediante restas sucesivas. Diseñalo.
2. El mínimo común múltiplo puede calcularse aumentando... hasta llegar a un múltiplo común. Diseña este algoritmo.
3. Se ha encontrado en Marte la siguiente operación de sumar, resuelta en una roca:

$$\begin{array}{r} \clubsuit \quad \diamondsuit \quad \spadesuit \\ \quad \quad \clubsuit \quad \heartsuit \\ \hline \diamondsuit \quad \spadesuit \quad \clubsuit \end{array}$$

Se desea descifrar el significado (o sea, el valor) de esos símbolos, supuesto que se ha usado el sistema de numeración decimal.

Una posibilidad es tantear todas las combinaciones de cifras posibles y ver cuáles de ellas funcionan:

			
0	0	0	0
0	0	0	1
...
0	0	0	9
0	0	1	0
...
0	9	9	9
1	0	0	0
...

4. Programa de adivinación (del libro de “Ejercicios [...] creativos [...]”)

27 Juego de adivinación

Consideremos el siguiente juego entre los jugadores A (adivino) y P (pensador): P piensa un número comprendido entre 1 y N (digamos 1000, por ejemplo), y A trata de adivinarlo, mediante tanteos sucesivos, hasta dar con él. Por cada tanteo de A, P da una respuesta orientativa de entre las siguientes:

Fallaste. El número pensado es menor que el tuyo.
Fallaste. Mi número es mayor.
Acertaste al fin.

Naturalmente, caben diferentes estrategias para el adivino:

- Una posibilidad, si el adivino no tiene prisa en acabar, consiste en tantear números sucesivamente: primero el 1, después el 2, etc. hasta acertar.
- Otra estrategia, sin duda la más astuta, consiste en tantear el número central de los posibles de modo que, al fallar, se limiten las posibilidades a la mitad (por eso se llama *bipartición* a este modo de tantear).
- También es posible jugar caprichosamente, tanteando un número al azar entre los posibles. Al igual que la anterior, esta estrategia también reduce el intervalo de posibilidades sensiblemente.

Se plantea desarrollar tres programas independientes: uno deberá desempeñar el papel del pensador; otro el del adivino (usando la estrategia de bipartición), y un tercero deberá efectuar la simulación del juego, asumiendo ambos papeles (y donde el adivino efectúa los tanteos a capricho).

Para simular la elección de números al azar, puede consultarse el ejercicio 3.13.

5. Diseña una función que calcule el cero de otra función según el método de bipartición.

Este ejercicio está tomado del libro "Ejercicios creativos y recreativos en C++" (v. <http://antares.sip.ucm.es/cpareja/libroCPP/>). En la página 227 puede encontrarse una breve explicación del mismo, quizá te resulte útil.

6. Ídem, mediante el algoritmo de Newton-Raphson.

7. Diseña un algoritmo que realice la descomposición clásica de un número en factores, a la manera clásica: se comienza dividiendo el número original entre el divisor más pequeño posible (2), se actualiza el dividendo y se continúa con ese divisor o con el siguiente, cuando haya de ser así:

```
60|2
30|2
15|3
5|5
1|
```

A5 – Bucles for-in

1. Diseña una función que imprime todos los divisores de un número. Bueno, en realidad, lo de que "imprime", no me gusta mucho, porque eso no es una función-función, sino una instrucción (un procedimiento). Anda, modifícala para que devuelva la lista de los divisores, y si es posible hazlo de forma que recorra los candidatos sin rebasar la raíz del dato.
2. Diseña una función que toma una lista de enteros como parámetro y devuelve el mínimo y el máximo valor de dicha lista.
3. Diseña una función que filtra las palabras de una lista, quedándose con las que están escritas en minúscula.
4. Reescribe el ejercicio de la suma marciana con for-in, en vez de while.

Se trata de una colección de bucles anidados. En cada uno de ellos, es posible aplicar un filtro, en vez de hacerlo al final.

5. Una fracción se puede simplificar calculando el máximo común divisor de sus términos... Diseña una función que da la fracción simplificada y otro que simplifica "in place" una fracción dada. Este asunto puede ilustrar el concepto de objetos mutables e inmutables.

6. La función siguiente busca divisores de un número para descartarlo como primo en caso de encontrarlos...

```
def es_primo(n):
    if n<2:
        return False
    for i in range(2, int(math.sqrt(n)+1)):
        if n%i == 0:
            return False
    return True
```

Vemos en primer lugar la instrucción return dentro de un bucle for, asunto que se desaconseja en casi todos los lenguajes de Programación. En Python, no pasa nada: es la costumbre Pythónica.

En segundo lugar, este algoritmo limita la comprobación a los posibles divisores que no rebasan la raíz del dato... ¿te parece correcto?