Introducción

Lo más importante de un algoritmo es que sea correcto, es decir, que en todas las instancias que nos interesan nos de la respuesta correcta, y en el formato que se indica. Hablaremos un poco más de esto y veremos ejemplos de algoritmos que a simple vista parecen funcionar, pero fallan.

Correctitud de algoritmos

Es muy importante que los algoritmos que desarrollemos en verdad resuelvan el problema que estamos planteando. Esto es parecido a cuando en argumentos matemáticos queremos demostraciones que verdaderamente y de manera formal demuestren lo que se pide demostrar.

Así, cuando damos un algoritmo, este tiene que venir acompañado de una **descripción** clara de qué hace el algoritmo en cada momento. Y también debe venir acompañado de una **demostración de correctitud** que justifique por qué los pasos hechos en verdad dan la respuesta.

En la siguiente entrada hablaremos de cómo podemos dar demostraciones de que nuestros algoritmos son correctos. Por ahora nos enfocaremos en ver algoritmos que son incorrectos. Para que un algoritmo sea incorrecto basta con que falle en *una* de las instancias que debemos atender.

Fallos en el algoritmo: ¿Siempre primos?

Consideremos el siguiente problema algorítmico.

Problema. Determinar si cierta expresión es un número primo.

Entrada. Un entero positivo n.

Salida. Decir si la expresión n^2-n+41 es un número primo o no.

Si probamos manualmente los valores para $n=1,2,3,4$ obtenemos a los números $41,43,47,53$, que son todos ellos primos. Así, una propuesta de algoritmo podría ser la siguiente: responder siempre que sí.

En pseudocódigo:

```
definir expresion_prima(n):
    responder que sí
```

En código:

```
def expresion(n):
    print("El número {} sí es primo".format(n**2-n+41))
```

Hasta ahora parece que nuestro algoritmo marcha bien: da la respuesta correcta en los casos $1,2,3,4$, pues en todos ellos detecta correctamente que la expresión es un número primo. De hecho, el algoritmo sigue dando la respuesta correcta desde 1 hasta 10, como puedes verificar manualmente o con una tabla de primos.

```
for j in range(1,11):
    expresion(j)
```

```
El número 41 sí es primo
El número 43 sí es primo
El número 47 sí es primo
El número 53 sí es primo
El número 61 sí es primo
El número 71 sí es primo
El número 83 sí es primo
El número 97 sí es primo
El número 113 sí es primo
El número 131 sí es primo
```

¿Esto quiere decir que el algoritmo es correcto? Todavía no. De hecho, el algoritmo es incorrecto, pues falla para la instancia $n=41$. En efecto, por un lado el algoritmo dice lo siguiente.

```
expresion(41)
```

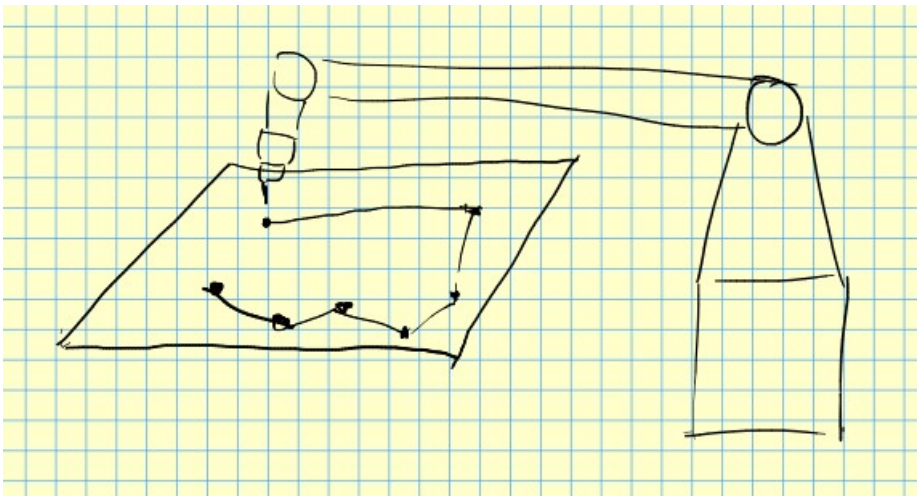
```
El número 1681 sí es primo
```

Sin embargo, manualmente podemos verificar muy fácilmente que esto es falso. La expresión correspondiente es $41^2-41+41=41^2$, que claramente no es un número primo pues tiene a 41 como divisor propio. $\square$

Fallos en el algoritmo: ¿El camino más corto?

En el ejemplo anterior falló una de las instancias, y por lo tanto el algoritmo es incorrecto. Veamos otro ejemplo de un algoritmo incorrecto.

Imagina que una empresa de circuitos eléctricos te contacta pues quieren que les ayudes a resolver el siguiente problema. Compraron un robot que debe elaborar un circuito eléctrico. Para ello, el robot debe conectar todos los puntos puestos sobre una placa, pero debe hacer esto de la manera más eficiente posible.



Un problema de este estilo todavía es algo informal. Para poder aplicarle herramientas matemáticas, es necesario construir un **modelo** que simplifique los detalles, pero que aún guarde los aspectos de lo que queremos resolver. Hablaremos más adelante de posibles formas de modelar, pero de momento podemos pensar que el problema queda en forma matemática como sigue:

Problema. Camino hamiltoniano euclideano mínimo.

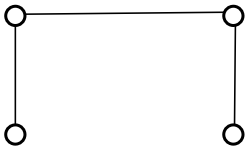
Entrada. Un conjunto de puntos $p_1,p_2,\ldots,p_n$ en el plano.

Salida. Una orden de recorrer todos ellos de modo que la distancia total recorrida sea mínima.

Por ejemplo, en la siguiente instancia tenemos cuatro puntos por recorrer:

Una forma de recorrerlos es la siguiente:

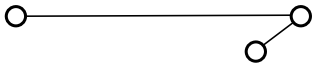
Pero una mejor forma de recorrerlos es la siguiente, pues en total se recorre menos distancia:



¿Podremos dar un algoritmo que resuelva de manera correcta todas las instancias? Una posible idea para resolver el problema puede ser la siguiente: comenzar en cualquier punto y de ahí ir siempre al punto más cercano que nos falte por visitar. Suena sensato: vamos optimizando en cada momento lo que se tiene que hacer, así que tal vez esto nos de la solución óptima.

Sin embargo, este algoritmo es incorrecto. Para ello, basta que encontremos una instancia en donde esta idea falle. Cuando buscamos contraejemplos para nuestras propuestas de algoritmo, es más fácil encontrarlas en instancias pequeñas. Con esto en mente, damos el siguiente ejemplo con tres puntos:

Si comenzamos con el punto de abajo, la idea de siempre ir al más cercano nos lleva a la derecha y luego a la izquierda como sigue:



Sin embargo, hay un mejor camino, como se muestra a continuación:

Así, nuestra propuesta de algoritmo falla en una instancia, y por lo tanto es un algoritmo incorrecto. \(\square\)

Fallos en el código

Aunque hayamos pensado muy bien nuestro algoritmo, es posible que cometamos algún error cuando lo implementemos, es decir, cuando lo pasemos a código en la computadora. Hay que ser particularmente cuidadosos con esto y hacer varios casos de prueba para ver que no estemos haciendo alguna cosa mal.

Ejemplo. Consideremos el siguiente problema:

Problema: Encuentra la posición donde está el valor máximo de una lista no vacía de enteros distintos.

Entrada: Una lista \(a_0,a_1,a_2,\ldots a_n\) de números.

Salida: El índice \(j\) tal que \(a_j\) es máximo.

Podemos resolver el problema como sigue. Tomamos la lista y la vamos leyendo de izquierda a derecha. Al inicio, declaramos \(a_0\) como el potencial máximo y entonces a \(0\) como el potencial índice que nos interesa regresar. De aquí, cada que llegamos a un nuevo elemento lo comparamos con el que hasta ahora sea el máximo. Si el nuevo elemento es mayor, entonces ahora lo declaramos como máximo y a su índice como el índice del máximo. Al llegar al final tendremos el índice que nos interesa.

Un intento de implementación de esto en código de Python es la siguiente:

```
def indice_max(L):
    maximo=L[0]
    ind_max=0
    for j in range(1,len(L)):
        if L[j]>maximo:
            maximo=L[j]
            ind_max=0
    print("El lugar donde está el máximo de L es el {}".format(maximo))
```

```
indice_max([1,-3,2,4,3,5])
indice_max([0,1,2,3,-1])
indice_max([-5,0,-1,3])
indice_max([0,6,-1,3,2,1,4])
```

```
El lugar donde está el máximo de L es el 5
El lugar donde está el máximo de L es el 3
El lugar donde está el máximo de L es el 3
El lugar donde está el máximo de L es el 6
```

En las primeras tres ejecuciones parece ser que `indice_max` está funcionando bien: está dando la posición en donde se encuentra el elemento más grande. En la primer lista el elemento más grande es el 5, que está en la posición 5 (recuerda que la primera es la posición 0).

Sin embargo, el algoritmo se equivoca en la última lista, ¡el máximo es 6 y no está en la posición 6! Nos gustaría que el algoritmo dijera que el máximo está en la posición 1. ¿Qué está fallando?

Lo que pasa es un error de código. Aunque diseñamos bien nuestro algoritmo, al final estamos pidiendo que el algoritmo imprima `maximo` en vez de `ind_max` que es lo que nos interesa. Tenemos que hacer este cambio y revisar con mucho cuidado si no hay algún otro error que nos pudiera dar respuestas incorrectas. $\square$

Tarea moral

Los siguientes problemas te ayudarán a practicar lo visto en esta entrada. Para resolverlos, necesitarás usar herramientas matemáticas, computacionales o ambas.

1. Una forma de que el algoritmo del que pusimos arriba no falle en la instancia 41 es pedir que diga que sí es primo cuando n no es múltiplo de 41 y que diga que no es primo cuando n sí es múltiplo de 41. ¿Esto da un algoritmo correcto?
2. Corrige el problema que tiene el código del último ejemplo. ¿Lo que comentamos es el único error que tenía? Realiza todas las correcciones necesarias. Luego, resuelve el problema análogo de encontrar el índice en donde está el mínimo. Da una descripción en palabras, pseudocódigo y código.
3. En el problema anterior, ¿cómo darías una justificación de que tu algoritmo en efecto siempre será correcto?
4. Resuelve a mano el problema del camino hamiltoniano euclideano mínimo para la instancia que consiste de los vértices de un pentágono regular de lado 1.
5. Encuentra contraejemplos para los siguientes algoritmos para resolver el problema del camino hamiltoniano euclideano mínimo:
   - Recorremos los puntos de izquierda a derecha.
   - Comenzamos con el de más a la izquierda, y siempre vamos hacia el más cercano
   - Conectamos los dos más cercanos, luego el más cercano a esos dos, luego el más cercano a esos tres y así sucesivamente.

Anterior
Problemas y algoritmos

Siguiente
Algoritmos correctos

Por Leonardo Ignacio Martínez Sandoval