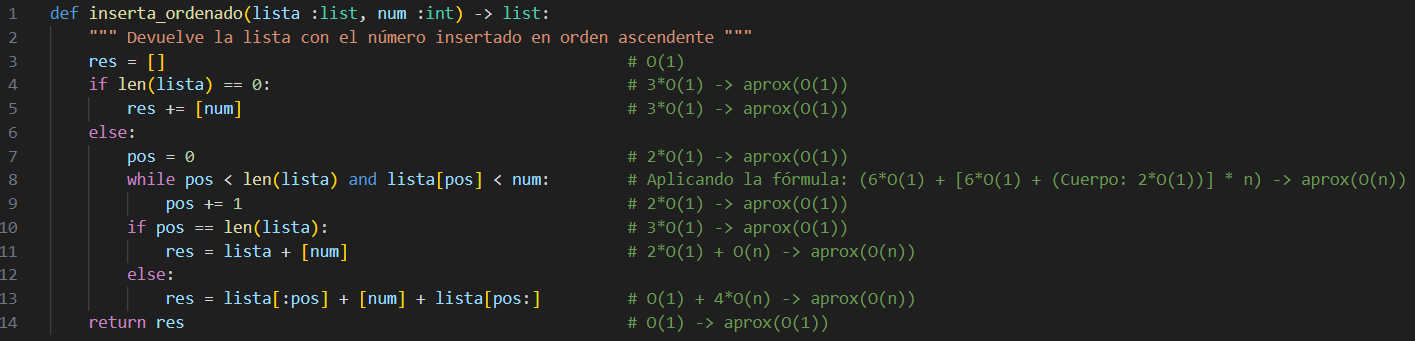
Pl1 – Algoritmia y complejidad

Juan Gil Aragonés (03247818B) y Lucas Marcos Vaquerizo (78120941S) – 21/03/2025

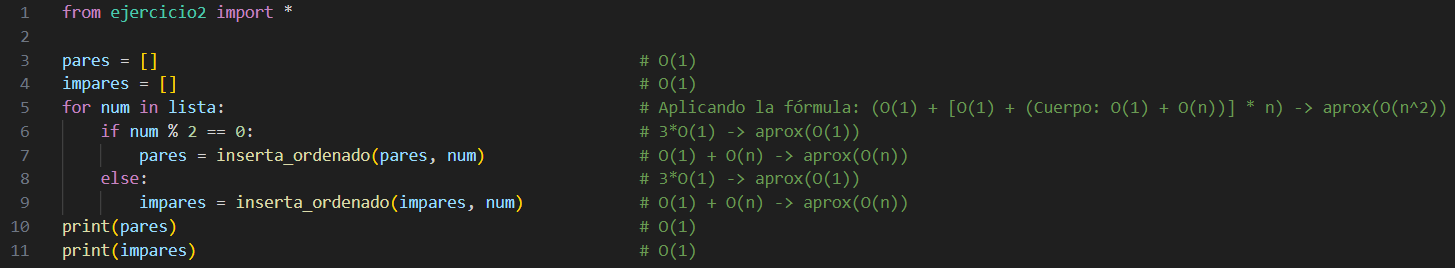
# Tema 1

Hemos elegido los ejercicios 3, 5 y 9

## Ejercicio 3:

(Ejercicio 2) Analiza la eficiencia del siguiente código:

Entrando en la segunda rama del condicional (línea 6), que es el peor caso, la peor opción es recorrer el while de la línea 8 (complejidad O(n)) y después sin importar en qué rama del condicional (línea 10) entres, ambas son complejidad ≈ O(n). Por lo tanto, la complejidad es  
O(n) + O(n) ≈ O(n)

(Ejercicio 3) Analiza la eficiencia del siguiente código que llama a la función del ejercicio 2:

El código contiene un bucle (línea 5). Este bucle contiene un condicional (línea 6) en el que ambas opciones implican una llamada a la función inserta\_ordenado del ejercicio 2, que es complejidad O(n). Así, da igual que opción del condicional recorras, la complejidad de este será  
≈ O(n). Como esta llamada se hace dentro de un bucle la complejidad del código será  
≈ O(n) \* n = O(n2).

**Complejidad:**

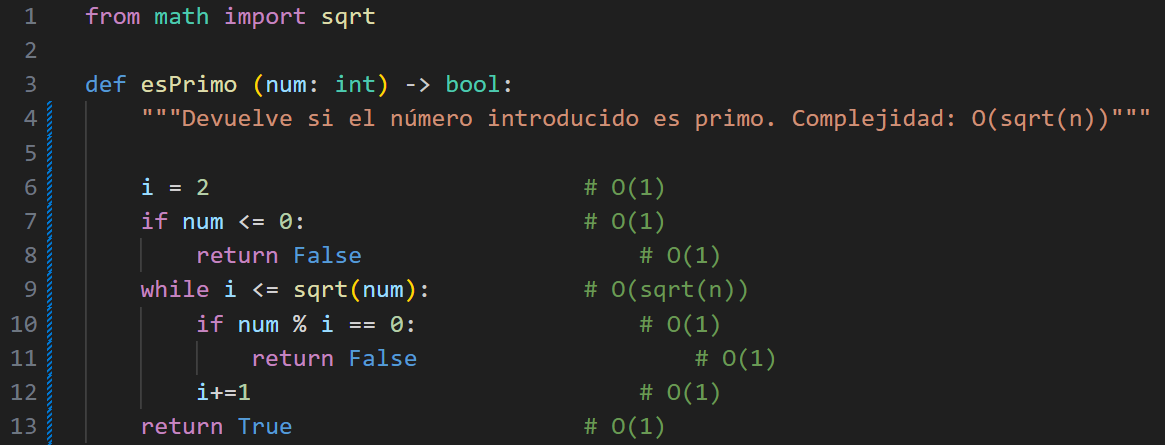
## Ejercicio 5:

Realiza una función que determine si un número recibido como parámetro es  
primo. Realiza un análisis de eficiencia y complejidad:

#### Descripción del algoritmo:

Se trata de comprobar de uno en uno si cada número menor que el introducido es divisor de este (comprobando si el módulo de ambos es 0). La idea es que no necesitamos comprobar todos los números menores que el introducido, sino simplemente los que sean menores o iguales que su raíz, puesto que el mayor divisor de un número que no sea este mismo es como mucho su raíz, o si es mayor que su raíz, entonces este divisor deberá ser multiplicado por uno menor que la raíz del introducido para dar este como resultado, y por tanto existirá un divisor menor que su raíz.

#### Análisis de complejidad:



Como se puede observar, todo son operaciones simples excepto un bucle while que se recorre veces.

**Complejidad:**

## Ejercicio 9:

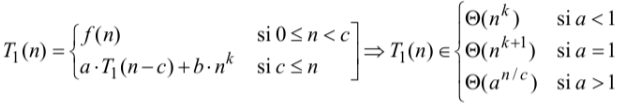
Realiza una función recursiva que calcule el siguiente sumatorio:  
S = 1 + 2 + 3 + 4 + ⋯ + n − 1 + n  
Realiza un análisis de eficiencia y de complejidad.

#### Descripción del algoritmo:

Se trata de un algoritmo que simplemente va sumando el número introducido (n) al sumatorio que estamos buscando pero de n-1 y recursivamente va calculando todos los sumatorios hasta llegar al caso base n=1, en el que devuelve 1. Se asegura que n sea un entero mayor que 0.

#### Análisis de complejidad:

El algoritmo en si sólo emplea operaciones simples, sin embargo, se llama recursivamente a sí mismo y suma el resultado (operación simple). Esto lo hace n veces. Por tanto, nos queda la siguiente ecuación de recurrencia:

Siguiendo el teorema maestro:

En el que a = 1, c = 1, k = 0:

**Complejidad:**

# Tema 2

Hemos elegido los ejercicios 3 y 4, además del 6

## Ejercicio 3:

Se dispone de un vector 𝑉 formado por 𝑛 datos, del que se quiere encontrar el  
elemento mínimo del vector y el elemento máximo del vector. El tipo de los datos  
no es relevante, pero la comparación entre dos datos para ver cuál es menor es  
muy costosa, implementar un método voraz que realice un máximo de 3𝑛/2 comparaciones.

#### Descripción del algoritmo:

La idea del algoritmo es ir comparando los elementos de V de 2 en 2. Después, comparamos el mayor de estos con el máximo que tengamos hasta ahora y el menor de estos con el mínimo que tengamos hasta ahora. Así por cada 2 elementos se hacen 3 comparaciones. Este es un algoritmo voraz puesto que solucionamos el problema parcialmente repetidas veces hasta que hayamos recorrido todos los elementos de V y hayamos encontrado una solución óptima global.

#### Análisis de complejidad:

Este algoritmo no emplea recursividad. La inicialización de este no emplea bucles, sólo operaciones simples, por lo que es complejidad O(1). Después recorremos el vector entero de dos en dos, por lo que la complejidad es n/2 ≈ O(n) y dentro del bucle únicamente hay operaciones simples de comparación. En el caso de que V sea de longitud impar comparamos también el último elemento. Esto es de complejidad O(1). Entonces:

**Complejidad:**

Por otro lado nos interesa en este caso conocer el número de comparaciones. Por cada pareja de elementos se realizan 3 comparaciones, una entre los dos elementos, otra para ver si el mayor es mayor que el máximo y otra para ver si el menor es menor que el mínimo. Así, nos quedarían 3\*(n/2) comparaciones. Además de estas hay que considerar el caso en que el vector sea impar, lo que añade 2 comparaciones más (con el máximo y mínimo obtenidos hasta ahora). Esta última parte para conjuntos grandes de elementos es irrelevante.   
Por lo tanto, tenemos ≈ 3n/2 comparaciones.

## Ejercicio 4:

En un país, hay varias ciudades que necesitan estar conectadas mediante una red de fibra óptica para mejorar las comunicaciones. Cada tramo de fibra óptica entre dos ciudades tiene un coste específico de instalación en euros. El objetivo consiste en diseñar un algoritmo que conecte todas las ciudades, minimizando el coste total de instalación. Para ello, partimos de una lista de todas las ciudades y, una lista de posibles tramos de fibra óptica entre las ciudades con sus respectivos costes de instalación. Restricciones:

• Cada ciudad debe estar conectada a la red de fibra óptica.  
• La red debe ser continua, es decir, no debe haber ciudades aisladas.

#### Descripción del algoritmo:

El problema realmente se puede reducir a encontrar el árbol de peso mínimo del grafo formado por las ciudades como vértices y los costes de fibra óptica entre las ciudades como aristas.

Para resolverlo hemos empleado el algoritmo de Kruskal, que va formando el árbol eligiendo la arista de menor peso que no haya sido seleccionada anteriormente y que no forme ciclo con las elegidas anteriormente hasta que se hayan comprobado todas las aristas.

## Ejercicio 6:

Dado un conjunto de reservas de pistas de pádel, cada una con una hora de inicio  
y otra de finalización, diseñe un algoritmo que calcule el número mínimo de pistas  
necesarias para que todas las reservas puedan llevarse a cabo sin conflictos de  
horario.  
Para ello se dispone de una lista de intervalos de tiempo, donde cada intervalo está  
representado por una hora de inicio y una de finalización de la reserva.

#### Descripción del algoritmo:

Creamos una lista con tuplas (eventos de finalización o de comienzo de reserva) en las que el primer elemento es una de las dos horas de una reserva. Si es la hora de inicio de la reserva, lo representamos con un 1 en el segundo elemento de la tupla, y si es la hora de finalización, con un 0.

Ordenamos la lista. La función sort ordena la tupla según su primer elemento de menor a mayor y, si este elemento es igual, ordena según el segundo elemento, también de menor a mayor. Es decir, ordenará por horas y, dentro de cada hora, pondrá primero los eventos de finalización de una reserva y después los de inicio.

pistasMaximasUsadas representará el máximo número de pistas usadas hasta cierto momento y pistasLiberadas representará (pistasMaximasUsadas - nº de pistas en uso en cierto momento),  
o lo que es lo mismo, las pistas libres que hace falta ocupar para llegar a pistasMaximasUsadas.

Recorremos las horas de una en una y vemos si empieza o acaba alguna reserva:

Si empieza una reserva y pistasLiberadas es 0, significa que estamos en el número máximo de pistas empleadas hasta el momento, por lo que al necesitar una pista más aumentará en 1 el número máximo de pistas usadas hasta ahora.

Si acaba una pista, aumenta en 1 el número de pistasLiberadas.

Dentro de la misma hora, al ir primero los eventos de finalización de reserva y después los de inicio (al haberse ordenado con sort), se liberarán primero las pistas para poder ser ocupadas por las reservas que empiezan.

Se trata de un algoritmo voraz ya que va avanzando por las horas en las que ocurre algo y va calculando el número de pistas máximas hasta ese momento. Esto lo repite hasta que no hay eventos restantes por comprobar.

#### Análisis de complejidad:

# Tema 3

Hemos elegido los ejercicios 3 y 6

## Ejercicio 3:

## Ejercicio 6: