PL1 – ALGORITMIA Y COMPLEJIDAD

Juan Gil Aragonés (03247818B) y Lucas Marcos Vaquerizo (78120941S) – 27/03/2025

Índice

Tema 1

Hemos elegido los ejercicios 3, 5 y 9

Ejercicio 3:

(Ejercicio 2) Analiza la eficiencia del siguiente código:

```
1   def inserta_ordenado(lista :list, num :int) -> list:
2       """ Devuelve la lista con el número insertado en orden ascendente """
3       res = []                                                    # O(1)
4       if len(lista) == 0:                                         # 3*O(1) -> aprox(O(1))
5           res += [num]                                            # 3*O(1) -> aprox(O(1))
6       else:
7           pos = 0                                                 # 2*O(1) -> aprox(O(1))
8           while pos < len(lista) and lista[pos] < num:            # Aplicando la fórmula: (6*O(1) + [6*O(1) + (Cuerpo: 2*O(1))] * n) -> aprox(O(n))
9               pos += 1                                            # 2*O(1) -> aprox(O(1))
10          if pos == len(lista):                                   # 3*O(1) -> aprox(O(1))
11              res = lista + [num]                                 # 2*O(1) + O(n) -> aprox(O(n))
12          else:
13              res = lista[:pos] + [num] + lista[pos:]             # O(1) + 4*O(n) -> aprox(O(n))
14      return res                                                  # O(1) -> aprox(O(1))
```

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:

```
1   from ejercicio2 import *
2
3   pares = []                                                      # O(1)
4   impares = []                                                    # O(1)
5   for num in lista:                                               # Aplicando la fórmula: (O(1) + [O(1) + (Cuerpo: O(1) + O(n))] * n) -> aprox(O(n^2))
6       if num % 2 == 0:                                            # 3*O(1) -> aprox(O(1))
7           pares = inserta_ordenado(pares, num)                    # O(1) + O(n) -> aprox(O(n))
8       else:                                                       # 3*O(1) -> aprox(O(1))
9           impares = inserta_ordenado(impares, num)                # O(1) + O(n) -> aprox(O(n))
10  print(pares)                                                    # O(1)
11  print(impares)                                                  # O(1)
```

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 = $O(n^2)$.

Complejidad: $T(n) \in O(n^2)$

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:

```python
from math import sqrt

def esPrimo (num: int) -> bool:
    """Devuelve si el número introducido es primo. Complejidad: O(sqrt(n))"""

    i = 2                           # O(1)
    if num <= 0:                    # O(1)
        return False                    # O(1)
    while i <= sqrt(num):           # O(sqrt(n))
        if num % i == 0:                # O(1)
            return False                    # O(1)
        i+=1                            # O(1)
    return True                     # O(1)
```

Como se puede observar, todo son operaciones simples excepto un bucle while que se recorre $\sqrt{n}$ veces.

Complejidad: $T(n) \in O(\sqrt{n})$

Casos de prueba:

```
collected 5 items

Python/ejerciciosP1/Ej1_5.py::test_comprobar_primo_1000 PASSED                    [ 20%]
Python/ejerciciosP1/Ej1_5.py::test_comprobar_primo_3863 PASSED                    [ 40%]
Python/ejerciciosP1/Ej1_5.py::test_casos_limite_1 PASSED                          [ 60%]
Python/ejerciciosP1/Ej1_5.py::test_casos_limite_2 PASSED                          [ 80%]
Python/ejerciciosP1/Ej1_5.py::test_casos_limite_3 PASSED                          [100%]


------------------------------------------------------------------------------------ benchmark: 2 tests -------------------------
-----------------------------------------------------------------
Name (time in ms)                Min               Max              Mean            StdDev            Median               IQR
     Outliers        OPS (mops/s)           Rounds  Iterations
-------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------
test_comprobar_primo_1000     0.0002 (1.0)      0.0033 (1.0)      0.0002 (1.0)      0.0000 (1.0)      0.0002 (1.0)      0.0000 (1.0)
     817;4079  4,930,409,754.8027 (1.0)       50762         100
test_comprobar_primo_3863     0.0068 (38.42)    0.6945 (208.87)   0.0084 (41.17)    0.0037 (129.03)   0.0079 (39.11)    0.0002 (40.00)
   8342;21544    119,752,522.7934 (0.02)     144928           1
-------------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------------

Legend:
  Outliers: 1 Standard Deviation from Mean; 1.5 IQR (InterQuartile Range) from 1st Quartile and 3rd Quartile.
  OPS: Operations Per Second, computed as 1 / Mean
========================================= 5 passed in 5.52s =========================================
```

En los dos primeros tests comprobamos la duración media y coste para el ordenador del algoritmo para los números 1000 y 3863.

En los casos límite comprobamos lo que ocurre cuando el algoritmo recibe números inesperados como 0 y -1 o muy cerca de los casos base como 2.

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:

$$T(n) = \begin{cases} O(1) & si \quad 0 \leq n \leq 1 \\ T(n-1) + O(1) & si \quad 1 < n \end{cases}$$

Siguiendo el teorema maestro:

$$T_1(n) = \left.\begin{cases} f(n) & \text{si } 0 \leq n < c \\ a \cdot T_1(n-c) + b \cdot n^k & \text{si } c \leq n \end{cases}\right] \Rightarrow T_1(n) \in \begin{cases} \Theta(n^k) & \text{si } a < 1 \\ \Theta(n^{k+1}) & \text{si } a = 1 \\ \Theta(a^{n/c}) & \text{si } a > 1 \end{cases}$$

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

Complejidad: $T(n) \in O(n)$

Casos de prueba:

```
collected 5 items

Python/ejerciciosP1/Ej1_9.py::test_comprobar_sumatorio_20 PASSED                    [ 20%]
Python/ejerciciosP1/Ej1_9.py::test_comprobar_sumatorio_10 PASSED                    [ 40%]
Python/ejerciciosP1/Ej1_9.py::test_casos_limite_1 PASSED                            [ 60%]
Python/ejerciciosP1/Ej1_9.py::test_casos_limite_2 PASSED                            [ 80%]
Python/ejerciciosP1/Ej1_9.py::test_casos_limite_3 PASSED                            [100%]


------------------------------------------------------------------------------------ benchmark: 2 t
ests -------------------------------------------------------------------------------------
Name (time in ms)                  Min               Max              Mean            StdDev            M
edian              IQR             Outliers       OPS (mops/s)           Rounds  Iterations
-------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------
test_comprobar_sumatorio_10     0.0006 (1.0)      0.1334 (3.47)     0.0007 (1.0)      0.0005 (1.94)     0
.0007 (1.0)      0.0000 (1.0)       2412;16755  1,404,122,953.9033 (1.0)      181818          10
test_comprobar_sumatorio_20     0.0011 (1.86)     0.0385 (1.0)      0.0013 (1.81)     0.0003 (1.0)      0
.0013 (1.90)     0.0000 (1.25)      1514;8203    776,033,344.8818 (0.55)      87720           10
-------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------


Legend:
  Outliers: 1 Standard Deviation from Mean; 1.5 IQR (InterQuartile Range) from 1st Quartile and 3rd Quart
ile.
  OPS: Operations Per Second, computed as 1 / Mean
========================================= 5 passed in 4.75s =========================================
```

En este ejercicio hemos comprobado la eficiencia con casos relativamente grandes. No se podía aumentar más los valores debido a que si no excedía la capacidad de recursión máxima predefinida de Python y el algoritmo debía implementarse de forma recursiva.

Los casos límite incluyen valores negativos, 0 y un valor normal para comprobar que el algoritmo responde correctamente antes estos valores no habituales.

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.

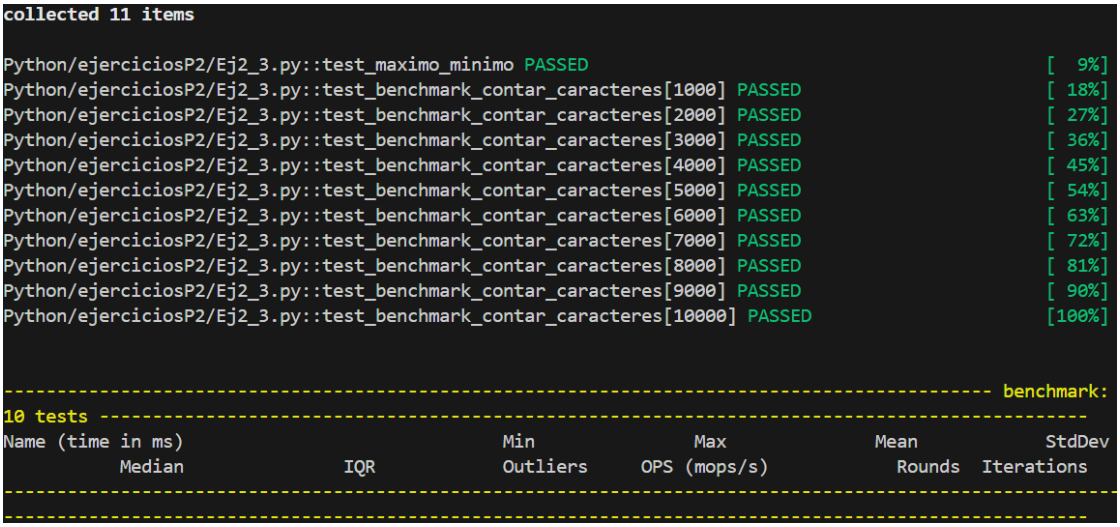
Análisis de complejidad:

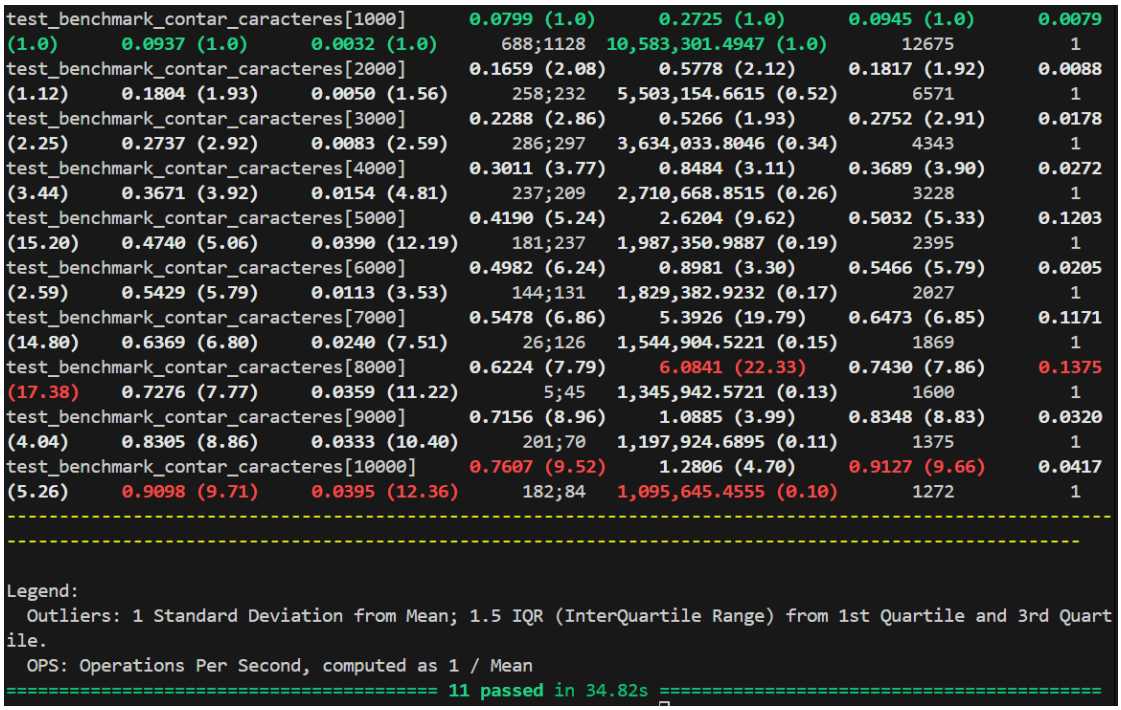
Este algoritmo no emplea recursividad. La inicialización de este no emplea bucles, sólo operaciones simples, por lo que es complejidad O(1). Después recorremos el vector entero de dos en dos, por lo que la complejidad es n/2 ≈ O(n) y dentro del bucle únicamente hay operaciones simples de comparación.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

Complejidad: $T(n) \in O(n)$

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.

Casos de prueba:





En este ejercicio hemos hecho primero un test simple con valores predefinidos y luego hemos creado listas de gran tamaño con un generador y comprobado la eficiencia con benchmark para rangos de valores de 1000 a 10000.

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.

Casos de prueba:

```
collected 1 item

Python/ejerciciosP2/Ej2_4.py::test_calcular_peso_minimo PASSED                                [100%]


------------------------------------------------- benchmark: 1 tests --------------------------------
-----------------
Name (time in ms)                Min     Max    Mean  StdDev  Median     IQR  Outliers     OPS (mops/s)
Rounds  Iterations
-----------------------------------------------------------------------------------------------------
-----------------
test_calcular_peso_minimo     0.0564  0.5302  0.0631  0.0078  0.0619  0.0029   544;696  15,859,172.6460
 17890           1
-----------------------------------------------------------------------------------------------------
-----------------

Legend:
  Outliers: 1 Standard Deviation from Mean; 1.5 IQR (InterQuartile Range) from 1st Quartile and 3rd Quart
ile.
  OPS: Operations Per Second, computed as 1 / Mean
========================================== 1 passed in 4.36s ===========================================
PS D:\Universidad\Cursos\2º 2º cuatri\Programacion_2o_cuatri>
```

En este ejercicio hemos usado un test para comprobar la velocidad y rendimiento del algoritmo en el ordenador con benchmark. No hay casos límite que tenga sentido comprobar.

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 iterando 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:

```
1  def minimo_pistas (reservas: list[tuple[int,int]]) -> int:
2      """Devuelve el minimo numero de pistas necesarias para que todas las reservas puedan llevarse a cabo sin solaparse"""
3
4      if len(reservas) == 0:
5          print("Error, la lista de reservas esta vacia")
6          return None
7      else:
8          # Creamos una lista con tuplas que registran eventos de inicio o fin de reserva
9          horas = []
10         for reserva in reservas:                # O(n)
11             horas.append((reserva[0], 1))
12             horas.append((reserva[1], 0))
13
14         # Ordenamos la lista.
15         horas.sort()                            # O(n log n)
16
17         # Inicializamos las variables pistasMaximasUsadas y pistasLiberadas.
18         pistasMaximasUsadas = 0
19         pistasLiberadas = 0
20
21         # Recorremos las horas de una en una y vemos si empieza o acaba alguna reserva.
22         for hora in horas:                      # O(n)
23             if (hora[1] == 1):
24                 if (pistasLiberadas > 0):
25                     pistasLiberadas -= 1
26                 else:
27                     pistasMaximasUsadas += 1
28             elif (hora[1] == 0):
29                 pistasLiberadas += 1
30
31         return pistasMaximasUsadas
```

El algoritmo no emplea recursividad. El resto de operaciones son elementales exceptuando un bucle (línea 9), que es O(n), ordenar la lista de eventos (horas), que es O(n log n), y el bucle final que recorre la lista de eventos de uno en uno, contabilizando las pistas máximas usadas en cada iteración, que es O(n).

Como la complejidad sería O(n) + O(n log n) + O(n):

Complejidad: $T(n) \in O(n \log n)$

Casos de prueba:

```
collected 3 items

Python/ejerciciosP2/Ej2_6.py::test_minimo_pista_vacia PASSED                    [ 33%]
Python/ejerciciosP2/Ej2_6.py::test_minimo_pista PASSED                          [ 66%]
Python/ejerciciosP2/Ej2_6.py::test_benchmark_contar_caracteres PASSED           [100%]

========================================= 3 passed in 1.37s =========================================
PS D:\Universidad\Cursos\2º 2º cuatri\Programacion_2o_cuatri>
```

En este caso hemos usado la herramienta timer2 en vez de benchmark, haciendo tests para los casos en que se pase una lista vacía y una lista con un elemento no tupla. También medimos el tiempo que tarda en ejecutarse el algoritmo con valores grandes (del 1000 al 10000).

Tema 3

Hemos elegido los ejercicios 3 y 7

Ejercicio 3:

Se tiene acceso a una función (x) de la que se sabe que en el intervalo real [$p1$, $p2$] tiene un único mínimo local en el punto $x0$, que es estrictamente decreciente entre [$p1$, $x0$] y que es estrictamente creciente entre [$x0$, $p2$]. Hay que observar que $x0$ puede coincidir con $p1$ o con $p2$.

Se tiene que buscar, de la manera más eficiente posible, todos los puntos x (si es que existen) del intervalo [$p1$, $p2$] tales que la función f tome un cierto valor k, es decir, se busca el conjunto de valores {$x \in [p1, p2]$ tal que $(x) = k$}. Para simplificar el proceso, en vez del valor exacto de cada x puede indicarse un intervalo de valores [α, β], con $\beta - \alpha < \varepsilon$, donde se encuentre x. Los datos del algoritmo serán el intervalo [$p1$, $p2$], el valor k que se está buscando, y el valor ε para la aproximación.

Descripción del algoritmo:

La idea del algoritmo es encontrar un intervalo (de tamaño máximo ε) que contenga al punto mínimo mediante búsqueda ternaria y devolver el punto medio de este intervalo, para después mediante búsqueda binaria en los trozos a ambos lados del mínimo encontrar intervalos (de tamaño máximo ε) con los dos puntos que estamos buscando. Al devolver el punto medio de estos intervalos estamos encontrando los puntos buscados con un error menor a $\varepsilon/2$ (de la búsqueda del mínimo) + $\varepsilon/2$ (de la búsqueda de cada punto) = ε.

Análisis de complejidad:

```python
def encontrar_minimo(f, p1: int, p2: int, w):
    #Usamos la busqueda ternaria para encontrar el valor minimo de la función (con una aprximacion de w).
    while p2 - p1 > w:
        corte1 = p1 + (p2 - p1)/3
        corte2 = p2 - (p2 - p1)/3

        if f(corte1) < f(corte2):
            p2 = corte2
        else:
            p1 = corte1
    return (p1 + p2)/2
```

La búsqueda del mínimo es búsqueda ternaria, que tiene complejidad O(log n). Aunque no tiene una implementación recursiva, el algoritmo divide el problema original en 3 y trabaja de nuevo en la parte que contiene el intervalo que estamos buscando. Así, la ecuación correspondiente sería:

$$T(n) = \begin{cases} O(1) & si \quad 0 \le n \le 3 \\ T(n/3) + O(1) & si \quad 3 < n \end{cases}$$

Siguiendo el teorema maestro:

$$T_2(n) = \begin{cases} g(n) & \text{si } 0 \le n < c \\ a \cdot T_2(n/c) + b \cdot n^k & \text{si } c \le n \end{cases} \Rightarrow T_2(n) \in \begin{cases} \Theta(n^k) & \text{si } a < c^k \\ \Theta(n^k \cdot \log n) & \text{si } a = c^k \\ \Theta(n^{\log_c a}) & \text{si } a > c^k \end{cases}$$

En el que a = 1, c = 3, k = 0 y $a=c^k$:

Complejidad: $T(n) \in O(\log n)$

```python
def busqueda_binaria_en_monotonia(f, p1, p2, k, w, decrece):
    #Utilizamos busqueda binaria para encontrar los valores de k, solo se retornará uno pues nos aseguramos de que la función sea
    #monótona en el intervalo que le introducimos.
    if decrece:
        if (f(p1) < k or f(p2) > k):
            return None
    else:
        if (f(p1) > k or f(p2) < k):
            return None

    while p2 - p1 > w:
        medio = (p2 + p1)/2

        if (f(medio) > k):
            if decrece:
                p1 = medio
            else:
                p2 = medio
        else:
            if decrece:
                p2 = medio
            else:
                p1 = medio
    return (p1 + p2)/2
```

La búsqueda de los puntos una vez conocido el mínimo a ambos lados de este es búsqueda binaria, que sabemos que es O(log n):

Complejidad: $T(n) \in O(\log n)$

```python
def encontrar_valores(f, p1, p2, k, w):
    #usamos las funciones anteriores para encontrar el mínimo en la función y así poder buscar los valores en cada lado de la función.

    min = encontrar_minimo(f, p1, p2, w)                        #O(log n)
    f_min = f(min)

    x1 = busqueda_binaria_en_monotonia(f, p1, min, k, w, True)     #O(log n)
    x2 = busqueda_binaria_en_monotonia(f, min, p2, k, w, False)    #O(log n)

    return (round(x1, 3), round(x2, 3))
```

Así, el algoritmo final tendrá una complejidad de 3*log n ≈ O(log n)

Complejidad: $T(n) \in O(\log n)$

Casos de prueba:

```
collected 1 item

Python/ejerciciosP3/Ej3_3.py::test_encontrar_valores PASSED                    [100%]

========================================== 1 passed in 0.07s ==========================================
```

En este ejercicio hemos hecho simplemente un test para comprobar que el algoritmo funciona correctamente. No hay casos límite relevantes y sencillos de comprobar.

Ejercicio 7:

Se da como entrada una matriz de adyacencia de un grafo dirigido que representa el callejero de una ciudad. Diseñar un algoritmo Divide y Vencerás que trasponga esta matriz

Descripción del algoritmo:

El algoritmo es del tipo Divide y Vencerás y la idea es dividir la matriz en cuatro submatrices usando como referencia el punto medio. Después llamas recursivamente a la función, que transpondrá estas submatrices. Los casos base son especiales puesto que hay que considerar cuando nos encontremos con matrices de una fila o una columna. Estos casos se tratan aparte.

Finalmente, se unen las matrices ya transpuestas y se devuelve el resultado.

Análisis de complejidad:

Se trata de un algoritmo recursivo que divide el problema en 4 casos base más pequeños. La complejidad extra es n^2 porque se trata de un bucle, complejidad n, que concatena listas, lo que tiene a su vez complejidad n. Además, dividir las matrices también tiene complejidad n^2 porque divide listas, con complejidad n, en un bucle, complejidad n.

Así, tenemos la ecuación de recurrencia:

$$T(n) = \begin{cases} O(n) & si \quad 0 \leq n \leq 4 \\ 4T(n/4) + O(n^2) & si \quad 4 < n \end{cases}$$

Siguiendo el teorema maestro:

$$T_2(n) = \begin{cases} g(n) & \text{si } 0 \leq n < c \\ a \cdot T_2(n/c) + b \cdot n^k & \text{si } c \leq n \end{cases} \Rightarrow T_2(n) \in \begin{cases} \Theta(n^k) & \text{si } a < c^k \\ \Theta(n^k \cdot \log n) & \text{si } a = c^k \\ \Theta(n^{\log_c a}) & \text{si } a > c^k \end{cases}$$

En el que a = 4, c = 4, k = 2 y $a<c^k$:

Complejidad: $T(n) \in O(n^2)$

Casos de prueba:

```
collected 2 items

Python/ejerciciosP3/Ej3_7.py::test_traspuesta PASSED                    [ 50%]
Python/ejerciciosP3/Ej3_7.py::test_traspuesta_vacia PASSED              [100%]

========================================= 2 passed in 0.13s =========================================
```

En este ejercicio hemos hecho un test que comprueba si la matriz es traspuesta correctamente para varias matrices de ejemplo dadas y un test límite que comprueba si el algoritmo se rompe al introducir una matriz vacía.