Algoritmos y bases de datos

Ejercicio 1:

Para el primer ejercicio, desde la clasePuertasAct1, creo la clase PuertaLogica, usando el constructor __init__, y añadiendo dos métodos que me devolverán el nombre y la salida de la puerta. Luego creamos la clase PuertaBinaria, con su constructor __init__, creando los objetos pinA y PinB y asignándoles un valor de None, vacíos, para luego darles un valor. Luego creamos un metodo desde donde el usuario podrá darle un valor al pin A, solo si pinA no tiene valor todavia, lo mismo para el pinB. Luego creamos un metedo que nos servira para asignar el proximo pin. Empezamos creando las puertas logicas, primero la puerta And, donde, desde los metodos de obtenerPinA y obtenerPinB, asignaremos un valor entre 0 y 1 a las variables a y b. Al ser una puerta and, nos devolverá True si a y b son iguales a 1, si no devolverá false. Para la puerta OR, igual pero se nos devolverá True si a o b valen 1. Para crear la puertaNOT, debemos crear una clase llamada puertaUnaria, que será similar a la puertaBinaria, pero con un solo metodo para recibir el pin. La puerta NOT simplemente cambiará por false o por True según reciba el valor del pin, siendo este el contrario. Usando la clase conector, conectaremos dos puertas, lo cual nos servirá para utilizar el circuito de la demostración. Por último, las puertas NAND, NOR y XOR. Las puertas NAND y NOR son iguales a las AND y OR pero al revés, al tener una puerta NOT delante. Y la puerta XOR devolverá True si a es igual a 1 y b es igual a 0, o bien a es igual a 0 y b igual a 1. Para el programa principal, tenemos varios casos. Primero, importamos toda la clase con from ClasePuertasAct1 import *, y definimos una función que contendrá el programa. Aquí, probamos varios casos para comprender el funcionamiento de las nuevas clases añadidas.

Ejercicio 2:

Para este ejercicio, creamos una clase específica para realizar la comprobación de la ley de Morgan. Declaramos en el constructor __init__ todas los atributos a usar, y luego el método ComprobarEcuacion, que nos servirá para verificar que se cumple dicha igualdad. El programa principal es donde asignaremos el valor de cada elemento de la ecuación con True y False. Luego llamaremos al atributo para comprobar la ecuación y nos devolverá que en efecto ambas ecuaciones son equivalentes.

Ejercicio 3:

Creo dos funciones en la clases fracciones, una (get_NUM) que devuelve el numerador y otra get_DEN que devuelve el denominador de la propia fracción. Además añado el resto de operadores comparativos gracias a los métodos mágicos de python.

Ejercicio 4:

Creo una función que implemento en el constructor en la que hay un bucle en el que si el denominador es un número negativo, el denominador pasa a ser el valor absoluto del denominador y el numerador pasa a ser el negado del mismo.

Ejercicio 5:

Para este ejercicio, en la clase Fechas, en el constructor metemos un if con un raise exception para tratar la excepción de que el usuario en cualquier momento meta un valor erróneo de año o día, y creamos los atributos dia año y mes. Luego con el constructor __str__ devolvemos la fecha en str. A partir de ese momento todo son métodos, uno que verifica si el año es bisiesto, si la fecha introducida es mayor, menos, igual, no igual, mayor o igual y menor o igual que la fecha actual, y calcula el día siguiente. En el programa principal importamos datetime para pasar la fecha actual, y le damos al usuario la opción de dar la fecha separada por "," o por "/", y desde ahí comparara fechas.

Ejercicio 6:

Este ejercicio tiene una estructura similar al anterior, pero esta vez teniendo en cuenta las horas. En el constructor __init__ introducimos la excepción necesaria. En cuanto a métodos, tenemos todos los métodos comparativos básicos implementados con los métodos mágicos de python, y varios métodos que hacen cálculos con horas. En el programa principal definimos ciertas horas para hacer los cálculos y las comparaciones necesarias

Ejercicio 7:

Creamos la clase polinomio en la que vamos a introducir una lista con los coeficientes de los polinomios en orden ascendente, es decir, el menor grado en la primera posición de la lista y el mayor grado en la última. Después creamos funciones para la suma, resta, multiplicación y división de polinomios.

Ejercicio 8:

Creamos la clase complejos en la que vamos a introducir dos números separados por coma, el primero es la parte real del número complejo y el segundo número es la parte imaginaria del complejo. Una vez creado el complejo implementamos la suma, resta, multiplicación, división de dos complejos. Por último el módulo del complejo que se calcula haciendo la raíz cuadrada del (la parte real al cuadrado + la parte imaginaria al cuadrado).

Ejercicio 9:

Creamos una dado que recibe el nombre de un jugador y el número de dados, y definimos una tirada, una función que añade un valor a la lista de última tirada, que son los valores del dado.

Creamos una clase juego que recibe el número de turnos y el número de dados por turnos, es decir, los dados que se tiran en cada turno. Los valores del dado se meten en una lista por cada turno, además al final del juego se suman todas las puntuaciones de todos los turnos para calcular la puntuación de cada jugador.

EJERCICIO 11:

Creamos dos bucles para la obtención del mínimo de una lista, La primera es de orden n^2 por lo que tiene que haber un bucle for dentro de otro bucle for. Recorriendo la lista n veces por cada elemento.
En cambio en la segunda función que es lineal es de orden n por lo que solo tiene que recorrer la lista en un bucle for.

Ejercicio 12:

En el 12 primero creamos una función que buscara un vector, además de establecer una constante MAXN. En el programa principal declaramos las variables enteras 'n' y 'x', y el vector entero v, con el número de espacios de MAXN+1. Luego creamos 3 casos, donde gracias a la librería de ctime podemos declarar cuanto tarda en buscar el vector en el mejor caso, el peor y el intermedio. Tal y como pide el experimento, la función del para buscar el vector se compone de un algoritmo que realiza la indexación, siendo esta de orden $O(1)$.

Ejercicio 13:

Ejercicio 14:

Para este ejercicio importamos timeit y random, y hacemos una función que sea prueba1 donde implementaremos el operador del a una lista, y otra función llamada prueba2 donde implementaremos el operador del en un diccionario. En el programa principal usamos la librería timeit para hacer uso de las funciones y comparar el tiempo entre ambos experimentos, siendo más rápido el uso del diccionario.

Ejercicio 15:

Creamos un vector pidiendo el tamaño del mismo, lo recorremos con dos bucles for, uno dentro del otro aplicando el algoritmo llamado el método burbuja por el cual comparamos los valores de cada posición con el de la siguiente posición y si es mayor que el siguiente lo cambiamos de posicion, asi ordenamos la lista de menor a mayor para poder acceder fácilmente al número que buscamos.
Al final solicitamos la posición del k-ésimo número del array y lo imprimimos por pantalla.

Ejercicio 16:

No se podrá mejorar el rendimiento porque la ordenación requiere de n*log(n) de tiempo de complejidad por lo que no es posible reducir este tiempo.

Ejercicio 17:

Creamos una lista pidiendo el número de elementos al usuario,creamos una variable suma , recorremos y sumamos cada valor de la lista a suma. Imprimimos la variable suma.
Este algoritmo es lineal por lo que es ***O(n)*** ya que solo recorremos una vez la lista.

Ejercicio 18:

Creamos una lista pidiendo el número de elementos al usuario,creamos una variable contador recorremos anidando dos bucles for, dentro del segundo bucle for metemos un if que si los valores de la lista de i y la lista de j el contador sume 1, al final dentro del 1 bucle.

for introducimos un if que si contador es mayor que dos i el elemento no está en la lista de los repetidos que lo añada.
La complejidad de esta función es de orden O(n**2) porque recorre la lista por cada elemento de la lista.
Es mejor un diccionario.