```
# draw AVL Tree
#!/usr/bin/python3
import sys
import random
import time
from PyQt5.QtCore import *
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *
import re

class TreeNode(object):
    def __init__(self, val, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
        self.height = 1

def find_min(node):
    """Find min value node"""
    while node and node.left:
        node = node.left
    return node


def find_max(node):
    """Find max value node"""
    while node and node.right:
        node = node.right
    return node

def find(value, node):
    """Find node with val equal to value"""
    while node:
        if value < node.val:
            node = node.left
        elif value > node.val:
            node = node.right
        else:
            return node

def insert(value, node):
    """Insert value into node by following BST properties"""
    if node is None:
        return TreeNode(value)

    if value < node.val:
        node.left = insert(value, node.left)

    elif value > node.val:
        node.right = insert(value, node.right)
    else:
        # duplicate, ignore it
        return node
    return node
"""insert the value and balance"""
def insert_balance( root, key):
    # Step 1 - Perform normal BST
```

```
    if not root:
        return TreeNode(key)
    elif key < root.val:
        root.left = insert_balance(root.left, key)
    else:
        root.right = insert_balance(root.right, key)

    # Step 2 - Update the height of the
    # ancestor node
    root.height = 1 + max(getHeight(root.left),
                          getHeight(root.right))

    # Step 3 - Get the balance factor
    balance = getBalance(root)

    # Step 4 - If the node is unbalanced,
    # then try out the 4 cases
    # Case 1 - Left Left
    if balance > 1 and key < root.left.val:
        return rightRotate(root)

    # Case 2 - Right Right
    if balance < -1 and key > root.right.val:
        return leftRotate(root)

    # Case 3 - Left Right
    if balance > 1 and key > root.left.val:
        root.left = leftRotate(root.left)
        return rightRotate(root)

    # Case 4 - Right Left
    if balance < -1 and key < root.right.val:
        root.right = rightRotate(root.right)
        return leftRotate(root)

    return root
def leftRotate(z):
    y = z.right
    T2 = y.left

    # Perform rotation
    y.left = z
    z.right = T2

    # Update heights
    z.height = 1 + max(getHeight(z.left),
                       getHeight(z.right))
    y.height = 1 + max(getHeight(y.left),
                       getHeight(y.right))

    # Return the new root
    return y

def rightRotate(z):
    y = z.left
    T3 = y.right
```

```
    # Perform rotation
    y.right = z
    z.left = T3

    # Update heights
    z.height = 1 + max(getHeight(z.left),
                       getHeight(z.right))
    y.height = 1 + max(getHeight(y.left),
                       getHeight(y.right))

    # Return the new root
    return y

def getHeight(root):
    if not root:
        return 0

    return root.height

def getBalance(root):
    if not root:
        return 0

    return getHeight(root.left) - getHeight(root.right)
def delete(value, node):
    """Deletes node from the tree
    Return a pointer to the resulting tree
    """
    if node is None:
        return None

    if value < node.val:
        node.left = delete(value, node.left)
    elif value > node.val:
        node.right = delete(value, node.right)
    elif node.left and node.right:
        tmp_cell = find_min(node.right)
        node.val = tmp_cell.val
        node.right = delete(node.val, node.right)
    else:
        if node.left is None:
            node = node.right
        elif node.right is None:
            node = node.left
    return node

class AsciiNode(object):
    left = None
    right = None

    # length of the edge from this node to its children
    edge_length = 0
    height = 0
    lablen = 0

    # -1 = left, 0 = root, 1 = right
    parent_dir = 0
```

```
    # max supported unit32 in dec, 10 digits max
    label = ''


MAX_HEIGHT = 1000
lprofile = [0] * MAX_HEIGHT
rprofile = [0] * MAX_HEIGHT
INFINITY = (1 << 20)

# adjust gap between left and right nodes
gap = 3

def build_ascii_tree_recursive(t):
    """
    :type t: TreeNode
    """
    if t is None:
        return None

    node = AsciiNode()
    node.left = build_ascii_tree_recursive(t.left)
    node.right = build_ascii_tree_recursive(t.right)

    if node.left:
        node.left.parent_dir = -1

    if node.right:
        node.right.parent_dir = 1

    node.label = '{}'.format(t.val)
    node.lablen = len(node.label)
    return node

# Copy the tree into the ascii node structure
def build_ascii_tree(t):
    if t is None:
        return None
    node = build_ascii_tree_recursive(t)
    node.parent_dir = 0
    return node

# The following function fills in the lprofile array for the given tree.
# It assumes that the center of the label of the root of this tree
# is located at a position (x,y).  It assumes that the edge_length
# fields have been computed for this tree.
def compute_lprofile(node, x, y):
    if node is None:
        return

    isleft = (node.parent_dir == -1)
    lprofile[y] = min(lprofile[y], x - ((node.lablen - isleft) // 2))
    if node.left:
        i = 1
        while (i <= node.edge_length and y + i < MAX_HEIGHT):
            lprofile[y + i] = min(lprofile[y + i], x - i)
            i += 1
```

```
    compute_lprofile(node.left, x - node.edge_length - 1, y + node.edge_length + 1)
    compute_lprofile(node.right, x + node.edge_length + 1, y + node.edge_length +
1)

def compute_rprofile(node, x, y):
    if node is None:
        return

    notleft = (node.parent_dir != -1)
    rprofile[y] = max(rprofile[y], x + ((node.lablen - notleft) // 2))
    if node.right is not None:
        i = 1
        while i <= node.edge_length and y + i < MAX_HEIGHT:
            rprofile[y + i] = max(rprofile[y + i], x + i)
            i += 1

    compute_rprofile(node.left, x - node.edge_length - 1, y + node.edge_length + 1)
    compute_rprofile(node.right, x + node.edge_length + 1, y + node.edge_length +
1)

# This function fills in the edge_length and
# height fields of the specified tree
def compute_edge_lengths(node):
    if node is None:
        return
    compute_edge_lengths(node.left)
    compute_edge_lengths(node.right)

    # first fill in the edge_length of node
    if (node.right is None and node.left is None):
        node.edge_length = 0
    else:
        if node.left:
            i = 0
            while (i < node.left.height and i < MAX_HEIGHT):
                rprofile[i] = -INFINITY
                i += 1
            compute_rprofile(node.left, 0, 0)
            hmin = node.left.height
        else:
            hmin = 0

        if node.right is not None:
            i = 0
            while (i < node.right.height and i < MAX_HEIGHT):
                lprofile[i] = INFINITY
                i += 1
            compute_lprofile(node.right, 0, 0)
            hmin = min(node.right.height, hmin)
        else:
            hmin = 0

        delta = 4
        i = 0
        while (i < hmin):
            delta = max(delta, gap + 1 + rprofile[i] - lprofile[i])
```

```
            i += 1

        # If the node has two children of height 1, then we allow the
        # two leaves to be within 1, instead of 2
        if (((node.left is not None and node.left.height == 1) or (
                        node.right is not None and node.right.height == 1)) and
delta > 4):
            delta -= 1
        node.edge_length = ((delta + 1) // 2) - 1

    # now fill in the height of node
    h = 1
    if node.left:
        h = max(node.left.height + node.edge_length + 1, h)
    if node.right:
        h = max(node.right.height + node.edge_length + 1, h)
    node.height = h

# used for printing next node in the same level,
# this is the x coordinate of the next char printed
print_next = 0

node_x = []
node_y = []
final_draw_number = []
temp_x = 0
temp_y = 0
# This function prints the given level of the given tree, assuming
# that the node has the given x coordinate.
def print_level(node, x, level):
    global temp_x
    global print_next
    global final_draw_number

    if node is None:
        return
    isleft = (node.parent_dir == -1)
    if level == 0:
        spaces = (x - print_next - ((node.lablen - isleft) // 2))
        sys.stdout.write(' ' * spaces)
        temp_x += spaces + 1

        node_x.append(temp_x)
        node_y.append(temp_y)
        print_next += spaces
        sys.stdout.write(node.label)
        final_draw_number.append(int(node.label))
        print_next += node.lablen

    elif node.edge_length >= level:
        if node.left:
            spaces = (x - print_next - level)
            sys.stdout.write(' ' * spaces)
            temp_x += spaces + 1
            print_next += spaces
            sys.stdout.write('/')
            print_next += 1
```

```
        if node.right:
            spaces = (x - print_next + level)
            sys.stdout.write(' ' * spaces)
            temp_x += spaces + 1
            print_next += spaces
            sys.stdout.write('\\')
            print_next += 1

    else:
        same_level = True
        print_level(node.left,
                    x - node.edge_length - 1,
                    level - node.edge_length - 1)
        same_level = True
        print_level(node.right,
                    x + node.edge_length + 1,
                    level - node.edge_length - 1)

# prints ascii tree for given Tree structure
def drawtree(t):
    global temp_x
    global temp_y
    if t is None:
        return
    proot = build_ascii_tree(t)
    compute_edge_lengths(proot)
    i = 0
    while (i < proot.height and i < MAX_HEIGHT):
        lprofile[i] = INFINITY
        i += 1

    compute_lprofile(proot, 0, 0)
    xmin = 0
    i = 0

    while (i < proot.height and i < MAX_HEIGHT):
        xmin = min(xmin, lprofile[i])
        i += 1

    i = 0
    global print_next
    while (i < proot.height):
        print_next = 0
        temp_y = i
        temp_x = 0
        print_level(proot, -xmin, i)
        print('')
        i += 1

    if proot.height >= MAX_HEIGHT:
        print(("This tree is taller than %d, and may be drawn
incorrectly.".format(MAX_HEIGHT)))

def deserialize(string):
    if string == '{}':
        return None
```

```
    nodes = [None if val == '#' else TreeNode(int(val))
             for val in string.strip('[]{}').split(',')]
    kids = nodes[::-1]
    root = kids.pop()
    for node in nodes:
        if node:
            if kids:
                node.left = kids.pop()
            if kids:
                node.right = kids.pop()
    return root

def draw_random_bst(n, balanced=False):
    """ Draw random binary search tree of n nodes
    """
    from random import randint
    nums = set()
    max_num = 10 * n
    if 0 < n < MAX_HEIGHT:
        while len(nums) != n:
            nums.add(randint(1, max_num))

    draw_bst(list(nums), balanced=balanced)


def draw_level_order(string):
    """ The serialization of a binary tree follows a level order traversal,
    where '#' signifies a path terminator where no node exists below.
    e.g. '{3,9,20,#,#,15,7}'
          3
         / \
        9  20
           / \
          15 7
    """
    drawtree(deserialize(string))

#global var to keep track of index in deserialize_preorder
currIndex = 0

#deserialize to preorder
def deserialize_preorder(nodes, key, min=float("-infinity"),
max=float("infinity")):
    global currIndex
    if currIndex >=  len(nodes): return None

    root = None

    if min < key < max:
        root = TreeNode(key)
        currIndex += 1

        if currIndex < len(nodes):
            root.left = deserialize_preorder(nodes, nodes[currIndex], min, key)

        if currIndex < len(nodes):
            root.right = deserialize_preorder(nodes, nodes[currIndex], key, max)
```

```
    return root

# Build bst from (sorted) nodes. Used to auto-balance the tree.
def to_bst(nodes, start, end):
    if start > end: return None

    mid = (start + end) / 2
    root = TreeNode(nodes[int(mid)])

    root.left = to_bst(nodes, start, mid-1)
    root.right = to_bst(nodes, mid+1,end)

    return root

def draw_bst(nodes, preorder=False, postorder=False,balanced=False):
    if not nodes: return

    #Convert list to ints if first element is int, otherwise use list of str
    if re.match("\d", str(nodes[0])):
         nodes = [int(x) for x in nodes]

    if balanced:
        root = TreeNode(nodes[0])
        for num in nodes[1:]:
            root = insert_balance( root,num)

    elif not preorder and not postorder:
        root = TreeNode(nodes[0])
        for num in nodes[1:]:
            root = insert(num, root)

    elif preorder:
        if type(nodes[0]) is int:
            root = deserialize_preorder(nodes, nodes[0])
        else:
            #Use "" as lower bound for string.
            #Append 'z' to max string to act as upper bound.
            root = deserialize_preorder(nodes, nodes[0], "", max(nodes) + "z")

        if currIndex != len(nodes):
            print("Not valid preorder sequence.")

    elif postorder:
        root = None #To do.

    drawtree(root)

class Main(QMainWindow):
    def __init__(self, parent = None):
        QMainWindow.__init__(self, parent)
        self.InitUi()

    def InitUi(self):
        self.textBox = self.CreateTextbox()
        self.button_OK = self.CreateButton_OK()
        self.button_clear = self.CreateButton_clear()
```

```
        self.button_insert = self.CreateButton_insert()
        self.button_delete = self.CreateButton_delete()
        self.button_balance = self.CreateButton_balance()

        self.show()
    def CreateTextbox(self):
        lb = QLabel(self, text = "TextBox: ")
        lb.move(180,60)
        textBox = QLineEdit(self)
        textBox.resize(500,30)
        textBox.move(250,60)
        return textBox

    def CreateButton_OK(self):
        button1 = QPushButton(self, text = "OK")
        button1.resize(50,35)
        button1.move(800, 60)
        button1.clicked.connect(lambda: self.setAmount())
        button1.clicked.connect(lambda: print("OK"))
        return button1

    def CreateButton_clear(self):
        button2 = QPushButton(self, text = "clear")
        button2.resize(50,35)
        button2.move(800, 95)
        # button2.clicked.connect(lambda: self.setAmount())
        button2.clicked.connect(lambda: print("clear"))
        return button2

    def CreateButton_insert(self):
        button2 = QPushButton(self, text = "insert")
        button2.resize(50,35)
        button2.move(800, 130)
        button2.clicked.connect(lambda: self.insert_node())
        button2.clicked.connect(lambda: print("insert"))
        return button2

    def CreateButton_delete(self):
        button2 = QPushButton(self, text = "delete")
        button2.resize(50,35)
        button2.move(800, 165)
        button2.clicked.connect(lambda: self.delete_node())
        button2.clicked.connect(lambda: print("delete"))
        return button2

    def CreateButton_balance(self):
        button2 = QPushButton(self, text = "balance")
        button2.resize(50,35)
        button2.move(800, 195)
        button2.clicked.connect(lambda: self.balance_node())
        button2.clicked.connect(lambda: print("balance"))
        return button2

    def balance_node(self):
        draw_bst(self.numbers, balanced=True)
        for i in range(len(node_x)):
            self.label(final_draw_number[i], node_x[i], node_y[i])
```

```
def insert_node(self):
    text = self.textBox.text()
    self.check = True

    #處理字串 從 "5,2,7,1,7,3" 變成 self.numbers = [5,2,7,1,7,3]
    i = 0
    tmp = ""
    while i < len(text):
        if text[i] == ",":
            self.numbers.append(int(tmp))
            tmp = ""
        if text[i] != ",":
            tmp = tmp + text[i]
        i += 1

    print(self.numbers)
    draw_bst(self.numbers)

    for i in range(len(node_x)):
        self.label(final_draw_number[i], node_x[i], node_y[i])

def delete_node(self):
    text = self.textBox.text()
    self.check = True

    #處理字串 從 "5,2,7,1,7,3" 變成 self.numbers = [5,2,7,1,7,3]
    i = 0
    tmp = ""
    while i < len(text):
        if text[i] == ",":
            self.numbers.remove(int(tmp))
            tmp = ""
        if text[i] != ",":
            tmp = tmp + text[i]
        i += 1

    print(self.numbers)
    draw_bst(self.numbers)

    for i in range(len(node_x)):
        self.label(final_draw_number[i], node_x[i], node_y[i])
def setAmount(self):
    text = self.textBox.text()
    if text == "":
        text = "5,2,7,1,7,3,10,8,4,6"
    self.check = True

    #處理字串 從 "5,2,7,1,7,3" 變成 self.numbers = [5,2,7,1,7,3]
    i = 0
    tmp = ""
    c = []
    while i < len(text):
        if text[i] == ",":
            c.append(int(tmp))
            tmp = ""
        if text[i] != ",":
```

```
                tmp = tmp + text[i]
            i += 1
        c.append(int(tmp))
        self.numbers = c
        print(self.numbers)
        draw_bst(self.numbers)

        for i in range(len(node_x)):
            self.label(final_draw_number[i], node_x[i], node_y[i])

    def label(self, t, x, y):
        lb = QLabel(self, text = "<font color=\"red\">" + str(t) + "</font>")
        lb.setAlignment(Qt.AlignCenter)
        lb.resize(30,30)
        lb.move((x+10)*10,(y+10)*30)
        lb.setStyleSheet("border: 3px solid blue; border-radius:10px;
QFrame{background-color:rgb(0,0,255)}")
        lb.show()
        # lb.hide()
        self.button_clear.clicked.connect(lambda: lb.hide()) #clear the label
        self.button_clear.clicked.connect(lambda: node_x.clear()) #clear the label
        self.button_clear.clicked.connect(lambda: node_y.clear()) #clear the label
        self.button_clear.clicked.connect(lambda: final_draw_number.clear())

        self.button_insert.clicked.connect(lambda: lb.hide()) #clear the label
        self.button_insert.clicked.connect(lambda: node_x.clear()) #clear the label
        self.button_insert.clicked.connect(lambda: node_y.clear()) #clear the label
        self.button_insert.clicked.connect(lambda: final_draw_number.clear())

        self.button_delete.clicked.connect(lambda: lb.hide()) #clear the label
        self.button_delete.clicked.connect(lambda: node_x.clear()) #clear the label
        self.button_delete.clicked.connect(lambda: node_y.clear()) #clear the label
        self.button_delete.clicked.connect(lambda: final_draw_number.clear())

        self.button_balance.clicked.connect(lambda: lb.hide()) #clear the label
        self.button_balance.clicked.connect(lambda: node_x.clear())
        self.button_balance.clicked.connect(lambda: node_y.clear())
        self.button_balance.clicked.connect(lambda: final_draw_number.clear())

def main():
    app = QApplication(sys.argv)
    main_window = Main()
    main_window.resize(10000,10000)
    main_window.show()
    app.exec_()

if __name__ == "__main__":
    main()
```

```
# Python code to insert a node in AVL tree

# Generic tree node class
class TreeNode(object):
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
        self.height = 1

# AVL tree class which supports the
# Insert operation
class AVL_Tree(object):

    # Recursive function to insert key in
    # subtree rooted with node and returns
    # new root of subtree.
    def insert(self, root, key):

        # Step 1 - Perform normal BST
        if not root:
            return TreeNode(key)
        elif key < root.val:
            root.left = self.insert(root.left, key)
        else:
            root.right = self.insert(root.right, key)

        # Step 2 - Update the height of the
        # ancestor node
        root.height = 1 + max(self.getHeight(root.left),
                              self.getHeight(root.right))

        # Step 3 - Get the balance factor
        balance = self.getBalance(root)

        # Step 4 - If the node is unbalanced,
        # then try out the 4 cases
        # Case 1 - Left Left
        if balance > 1 and key < root.left.val:
            return self.rightRotate(root)

        # Case 2 - Right Right
        if balance < -1 and key > root.right.val:
            return self.leftRotate(root)

        # Case 3 - Left Right
        if balance > 1 and key > root.left.val:
            root.left = self.leftRotate(root.left)
            return self.rightRotate(root)

        # Case 4 - Right Left
        if balance < -1 and key < root.right.val:
            root.right = self.rightRotate(root.right)
            return self.leftRotate(root)

        return root
```

```
    def leftRotate(self, z):

        y = z.right
        T2 = y.left

        # Perform rotation
        y.left = z
        z.right = T2

        # Update heights
        z.height = 1 + max(self.getHeight(z.left),
                           self.getHeight(z.right))
        y.height = 1 + max(self.getHeight(y.left),
                           self.getHeight(y.right))

        # Return the new root
        return y

    def rightRotate(self, z):

        y = z.left
        T3 = y.right

        # Perform rotation
        y.right = z
        z.left = T3

        # Update heights
        z.height = 1 + max(self.getHeight(z.left),
                           self.getHeight(z.right))
        y.height = 1 + max(self.getHeight(y.left),
                           self.getHeight(y.right))

        # Return the new root
        return y

    def getHeight(self, root):
        if not root:
            return 0

        return root.height

    def getBalance(self, root):
        if not root:
            return 0

        return self.getHeight(root.left) - self.getHeight(root.right)

    def preOrder(self, root):

        if not root:
            return

        print("{0} ".format(root.val), end="")
        self.preOrder(root.left)
        self.preOrder(root.right)
```

```
# Driver program to test above function
myTree = AVL_Tree()
root = None

root = myTree.insert(root, 10)
root = myTree.insert(root, 20)
root = myTree.insert(root, 30)
root = myTree.insert(root, 40)
root = myTree.insert(root, 50)
root = myTree.insert(root, 25)

"""The constructed AVL Tree would be
            30
           /  \
         20   40
        /  \     \
       10  25    50"""

# Preorder Traversal
print("Preorder traversal of the",
      "constructed AVL tree is")
myTree.preOrder(root)
print()

# This code is contributed by Ajitesh Pathak

# Python program to print complete Koch Curve.
from turtle import *

# function to create koch snowflake or koch curve
def snowflake(lengthSide, levels):
    if levels == 0:
        forward(lengthSide)
        return
    lengthSide /= 3.0
    snowflake(lengthSide, levels-1)
    left(60)
    snowflake(lengthSide, levels-1)
    right(120)
    snowflake(lengthSide, levels-1)
    left(60)
    snowflake(lengthSide, levels-1)

# main function
if __name__ == "__main__":
    # defining the speed of the turtle
    speed(0)
    length = 500.0

    # Pull the pen up – no drawing when moving.
    # Move the turtle backward by distance, opposite
    # to the direction the turtle is headed.
    # Do not change the turtle’s heading.
    penup()

    backward(length/2.0)
```

```
    # Pull the pen down – drawing when moving.
    pendown()
    for i in range(3):
        snowflake(length, 3) # 更改這裡 length 是圖的長度, 另一個變數是要多少褶皺
        right(120)

     # To control the closing windows of the turtle
    mainloop()
```

```
#Sierpiński_rectangle1
import turtle
def s(n, l):

    if n == 0: # stop conditions

        # draw filled rectangle

        turtle.color('black')
        turtle.begin_fill()
        for _ in range (4):
            turtle.forward(l)
            turtle.left(90)
        turtle.end_fill()

    else: # recursion

        # around center point create 8 smalles rectangles.
        # create two rectangles on every side
        # so you have to repeat it four times

        for _ in range(4):
            # first rectangle
            s(n-1, l/3)
            turtle.forward(l/3)

            # second rectangle
            s(n-1, l/3)
            turtle.forward(l/3)

            # go to next corner
            turtle.forward(l/3)
            turtle.left(90)

        # update screen
        turtle.update()

# --- main ---

# stop updating screen (to make it faster)
turtle.tracer(0)

# start
```

```
s(5, 400)

# event loop
turtle.done()
#Sierpiński_triangle1
import turtle

def drawTriangle(points,color,myTurtle):
    myTurtle.fillcolor(color)
    myTurtle.up()
    myTurtle.goto(points[0][0],points[0][1])
    myTurtle.down()
    myTurtle.begin_fill()
    myTurtle.goto(points[1][0],points[1][1])
    myTurtle.goto(points[2][0],points[2][1])
    myTurtle.goto(points[0][0],points[0][1])
    myTurtle.end_fill()

def getMid(p1,p2):
    return ( (p1[0]+p2[0]) / 2, (p1[1] + p2[1]) / 2)

def sierpinski(points,degree,myTurtle):
    colormap = ['blue','red','green','white','yellow',
                'violet','orange']
    drawTriangle(points,colormap[degree],myTurtle)
    if degree > 0:
        sierpinski([points[0],
                        getMid(points[0], points[1]),
                        getMid(points[0], points[2])],
                   degree-1, myTurtle)
        sierpinski([points[1],
                        getMid(points[0], points[1]),
                        getMid(points[1], points[2])],
                   degree-1, myTurtle)
        sierpinski([points[2],
                        getMid(points[2], points[1]),
                        getMid(points[0], points[2])],
                   degree-1, myTurtle)

def main():
   myTurtle = turtle.Turtle()
   myWin = turtle.Screen()
   # myPoints = [[-100,-50],[0,100],[100,-50]]
   myPoints = [[-200,-100],[0,200],[200,-100]]
   sierpinski(myPoints,5,myTurtle)
   myWin.exitonclick()

main()
```

```
#Insertion_Sort
import sys
import random
import time
from PyQt5.QtCore import *
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *

class Main(QMainWindow):
    def __init__(self, parent = None):
        QMainWindow.__init__(self, parent)
        self.InitUi()

    def InitUi(self):
        self.textBox = self.CreateTextbox()
        self.button = self.CreateButton()
        self.show()

    def CreateTextbox(self):
        lb = QLabel(self, text = "TextBox: ")
        lb.move(180,60)
        textBox = QLineEdit(self)
        textBox.resize(500,30)
        textBox.move(250,60)
        return textBox

    def CreateButton(self):
        button = QPushButton(self, text = "OK")
        button.resize(30,30)
        button.move(800, 60)
        button.clicked.connect(lambda: self.setAmount())
        button.clicked.connect(lambda: print("OK"))
        return button

    def setAmount(self):
        #init data
        text = self.textBox.text()

    def showAnswer(self, Str):

        lb = QLabel(self, text = string)
        lb.move(50,100 + self.count * 20)
        lb.resize(200,100)

        lb.show()
        self.button.clicked.connect(lambda: lb.clear()) #clear the label
        self.count += 1

def main():
    app = QApplication(sys.argv)
    main_window = Main()
    main_window.resize(1000,1000)
    main_window.show()
    app.exec_()

if __name__ == "__main__": main()
```

```
#Selection_Sort
import sys
import random
import time
from PyQt5.QtCore import *
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *

class Main(QMainWindow):
    def __init__(self, parent = None):
        QMainWindow.__init__(self, parent)
        self.InitUi()

    def InitUi(self):
        self.textBox = self.CreateTextbox()
        self.button = self.CreateButton()
        self.show()
        self.cards_amount = 0
        self.cards = []
        self.cards_number = []
        self.answer_string = ""
        self.count = 0

    def CreateTextbox(self):
        lb = QLabel(self, text = "TextBox: ")
        lb.move(180,60)
        textBox = QLineEdit(self)
        textBox.resize(500,30)
        textBox.move(250,60)
        return textBox

    def CreateButton(self):
        button = QPushButton(self, text = "OK")
        button.resize(30,30)
        button.move(800, 60)
        button.clicked.connect(lambda: self.setAmount())
        button.clicked.connect(lambda: print("OK"))
        return button

    def setAmount(self):
        #init data
        text = self.textBox.text()
        self.cards_amount = 0
        self.cards = []
        self.cards_number = []
        self.answer_string = ""
        self.count = 0
        if text == "":
            text = "dQ,h8,h6,c4,c5,h2,d7"

        #convert string to number of list
        i = 0
        tmp = ""
        while i < len(text):
            if text[i] == ",":
                self.cards.append(tmp)
                tmp = ""
```

```
            if text[i] != ",":
                tmp = tmp + text[i]
            i += 1
        self.cards.append(tmp)

        self.cards_amount = len(self.cards)
        switcher_flower = {
            'c': 0,
            'd': 1,
            'h': 2,
            's': 3
        }
        switcher_number = {
            '2': 1,
            '3': 2,
            '4': 3,
            '5': 4,
            '6': 5,
            '7': 6,
            '8': 7,
            '9': 8,
            'J': 10,
            'Q': 11,
            'K': 12,
            'A': 13
        }
        for i in range(self.cards_amount):
            tmp = switcher_flower.get(self.cards[i][0], 0)*13 +
switcher_number.get(self.cards[i][1], 10)
            self.cards_number.append(tmp)

        self.start_sorting()

    def start_sorting(self):
        #self.insertion_sort()
        self.selection_sort()

    def selection_sort(self):
        numbers = self.cards_number
        numbers_str = self.cards
        self.showAnswer(numbers_str)
        for i in range(len(numbers)):
            max = i
            for j in range(i + 1, len(numbers)):
                if numbers[j] > numbers[max]:
                    max = j
            if max != i:
                numbers[i] , numbers[max] = numbers[max] , numbers[i]
                numbers_str[i] , numbers_str[max] = numbers_str[max] ,
numbers_str[i]
                self.showAnswer(numbers_str)

    def showAnswer(self, Str):
        string = str(self.count + 1) + ': '
        string =  string + Str[0]
        for i in range(1,len(Str)):
```

```
            string =  string  + ',' + Str[i]
        lb = QLabel(self, text = string)
        lb.move(50,100 + self.count * 20)
        lb.resize(200,100)
        lb.show()
        self.button.clicked.connect(lambda: lb.clear()) #clear the label
        self.count += 1

def main():
    app = QApplication(sys.argv)
    main_window = Main()
    main_window.resize(1000,1000)
    main_window.show()
    app.exec_()

if __name__ == "__main__":
    main()
```

```
#Bubble_Sort3
import sys
import random
import time
from PyQt5.QtCore import *
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *

class AnotherWindow(QWidget):
    """
    This "window" is a QWidget. If it has no parent, it
    will appear as a free-floating window as we want.
    """
    def __init__(self,n,o):
        super().__init__()
        self.setStyleSheet('QFrame{background-color:rgb(0,0,255)}')
        self.number = n # 第幾個視窗
        self.output = o
        self.count = len(self.output)
        self.setWindowTitle(str(n))
        self.create_label()

    def create_label(self):
        x = 100
        y = 50
        for i in range(0, self.count):
            self.number_label(i + self.number * 10, 10, y)
            for j in range(len(self.output[0])):
                self.label(self.output[i][j], x, y)
                x += 100
            y += 110
            x = 100

    def number_label(self, t, x, y):
        nlb = QLabel(self, text = "<font color=\"yellow\">" + str(t+1) + "</font>")
```

```
        nlb.setStyleSheet(("font: 18pt;"))
        nlb.move(x,y)
    def label(self, t, x, y):
        temp = t * 10
        lb = QLabel(self, text = "<font color=\"red\">" + str(t) + "</font>")
        lb.setAlignment(Qt.AlignCenter)
        lb.resize(temp, temp)
        lb.move(x-temp/2,y-temp/2)
        lb.setStyleSheet("border: 3px solid blue; border-radius:"+str(temp/2)+"px;
QFrame{background-color:rgb(0,0,255)}")

class Main(QMainWindow):
    def __init__(self, parent = None):
        QMainWindow.__init__(self, parent)
        self.InitUi()

    def InitUi(self):
        self.textBox = self.CreateTextbox()
        self.button = self.CreateButton()
        self.w = [[None] for i in range(10)]  # No external window yet.
        self.show()
        self.numbers = [] # 使用者輸入的數字
        self.output_answer = [[0 for i in range(10)] for j in range(100)] # 紀錄要輸
出的數字
        self.count = 0 # 紀錄要交換幾次

    def CreateTextbox(self):
        lb = QLabel(self, text = "TextBox: ")
        lb.move(180,60)
        textBox = QLineEdit(self)
        textBox.resize(500,30)
        textBox.move(250,60)
        return textBox

    def CreateButton(self):
        button = QPushButton(self, text = "OK")
        button.resize(30,30)
        button.move(800, 60)
        button.clicked.connect(lambda: self.setAmount())
        button.clicked.connect(lambda: print("OK"))
        return button

    def setAmount(self):
        text = self.textBox.text()
        if text == "":
            text = "5,2,7,1,7,3,10,8,4,6"
        self.check = True

        #處理字串 從 "5,2,7,1,7,3" 變成 self.numbers = [5,2,7,1,7,3]
        i = 0
        tmp = ""
        c = []
        while i < len(text):
            if text[i] == ",":
                c.append(int(tmp))
                tmp = ""
            if text[i] != ",":
```

```
                tmp = tmp + text[i]
            i += 1
        c.append(int(tmp))
        self.numbers = c
        self.bubble_sort()

    def bubble_sort(self):
        for i in range(len(self.numbers)):
            self.output_answer[self.count][i] = self.numbers[i]
        swapped = True
        while swapped:
            swapped = False
            for i in range(len(self.numbers) - 1):
                if self.numbers[i] < self.numbers[i + 1]:
                    self.numbers[i], self.numbers[i + 1] = self.numbers[i + 1],
self.numbers[i]
                    self.count += 1
                    for i in range(len(self.numbers)):
                        self.output_answer[self.count][i] = self.numbers[i]
                    swapped = True
        self.count += 1

        for i in range(len(self.numbers)):
            self.output_answer[self.count][i] = self.numbers[i]

        #輸出結果
        i = 0
        for i in range(self.count // 10):
            self.w[i] = AnotherWindow(i,self.output_answer[i * 10 : i * 10 + 10])
            self.w[i].show()
        i += 1
        self.w[i] = AnotherWindow(i,self.output_answer[i * 10 : i * 10 + self.count
% 10 + 1])
        self.w[i].show()

        self.numbers = []
        self.output_answer = [[0 for i in range(10)] for j in range(100)]
        self.count = 0

def main():
    app = QApplication(sys.argv)
    main_window = Main()
    main_window.resize(10000,10000)
    main_window.show()
    app.exec_()

if __name__ == "__main__":
    main()
```

```
#Counting_Sort2

import sys
import random
import time
from PyQt5.QtCore import *
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *


class AnotherWindow(QMainWindow):
    """
    This "window" is a QWidget. If it has no parent, it
    will appear as a free-floating window as we want.
    """
    def __init__(self, on, os):
        super().__init__()
        self.resize(2000,2000)
        self.widget = QWidget()
        self.createScroll() # Scroll Area which contains the widgets, set as the
centralWidget
        self.show()

        self.output_numbers = on
        self.output_string = os
        self.vbox = QVBoxLayout()
        self.createVbox()

    def createScroll(self):
        self.scroll = QScrollArea()
        self.scroll.setVerticalScrollBarPolicy(Qt.ScrollBarAlwaysOn)
        self.scroll.setHorizontalScrollBarPolicy(Qt.ScrollBarAlwaysOff)
        self.scroll.setWidgetResizable(True)
        self.scroll.setWidget(self.widget)
        self.setCentralWidget(self.scroll)

    def createVbox(self):
        color = ["#5bc0de","#5cb85c","#0275d8","#f0ad4e","#d9534f","red"]
        count = 1
        for i in range(len(self.output_numbers)):
            object = QLabel(self.output_string[i])
            if self.output_numbers[i] >= (count+1) * 100:
                count += 1
            object.setStyleSheet("color : "+color[count-1])
            object.move(100,100+i)
            self.vbox.addWidget(object)

        self.widget.setLayout(self.vbox)
        self.move(180,100)


class Main(QMainWindow):
    def __init__(self, parent = None):
        QMainWindow.__init__(self, parent)
        self.InitUi()

    def InitUi(self):
```

```
        self.button = self.CreateButton()
        self.w = None  # No external window yet.
        self.numbers = []

    def CreateButton(self):
        button = QPushButton(self, text = "OK")
        button.resize(30,30)
        button.clicked.connect(lambda: self.setAmount())
        button.clicked.connect(lambda: print("OK"))
        return button

    def counting_sort(self):
        TOTAL_NUMBER = len(self.numbers)
        MAX = 599
        self.output_string = ["" for i in range(len(self.numbers))] # 整個字串的結果
        self.output_numbers = [0 for i in range(len(self.numbers))] # http status
code 的結果

        count = [ 0 for i in range(MAX)]

        for i in range(TOTAL_NUMBER):
            count[self.numbers[i]] += 1

        for i in range(MAX):
            count[i] += count[i-1]

        for i in range(TOTAL_NUMBER):
            self.output_string[count[self.numbers[i]] - 1] = self.lines[i]
            self.output_numbers[count[self.numbers[i]] - 1] = self.numbers[i]
            count[self.numbers[i]] -= 1


    def setAmount(self):
        fp = open("access.log", "r")
        self.lines = fp.readlines()
        count = 0
        tmp = []
        for l in self.lines:
            for i in range(len(l)):
                if count == 2:
                    i += 1
                    for j in range(3):
                        tmp.append(int(l[i+j]))
                    self.numbers.append(100 * tmp[0] + 10 * tmp[1] + tmp[2])
                    count = 0
                    tmp.clear()
                    break
                if l[i] == '"':
                    count += 1
        fp.close()

        self.counting_sort()
        self.w = AnotherWindow(self.output_numbers,self.output_string)
        self.output_file()

    def output_file(self):
        f = open("demofile2.txt", "w")
```

```
        for i in self.output_string:
            f.write(i+'\n')

def main():
    app = QApplication(sys.argv)
    main_window = Main()
    main_window.show()
    app.exec_()

if __name__ == "__main__":
    main()



#Shell_Sort2
import sys
import random
import time
import csv
from PyQt5.QtCore import *
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *

class AnotherWindow(QMainWindow):
    """
    This "window" is a QWidget. If it has no parent, it
    will appear as a free-floating window as we want.
    """
    def __init__(self, data):
        super().__init__()
        self.resize(2000,2000)
        self.widget = QWidget()
        self.createScroll() # Scroll Area which contains the widgets, set as the
centralWidget
        self.show()
        self.data_string = data
        self.vbox = QVBoxLayout()
        self.createVbox()

    def createScroll(self):
        self.scroll = QScrollArea()
        self.scroll.setVerticalScrollBarPolicy(Qt.ScrollBarAlwaysOn)
        self.scroll.setHorizontalScrollBarPolicy(Qt.ScrollBarAlwaysOff)
        self.scroll.setWidgetResizable(True)
        self.scroll.setWidget(self.widget)
        self.setCentralWidget(self.scroll)

    def createVbox(self):
        color = ["#d9534f","#f0ad4e","#5cb85c","#0275d8","red"]
        count = 1
        for i in range(len(self.data_string)):
            temp = ""
            object = QLabel(self)
            for j in range(len(self.data_string[i])):
                if j == 0:
                    temp += self.data_string[i][j]
                elif float(self.data_string[i][10]) < 0 and j == 10:
```

```
                    temp += " , <font color=\"#d9534f\">" + self.data_string[i][10]
+ "</font>"
                elif float(self.data_string[i][10]) == 0 and j == 10:
                    temp += " , <font color=\"#f0ad4e\">" + self.data_string[i][10]
+ "</font>"
                elif float(self.data_string[i][10]) < 10000000000 and j == 10:
                    temp += " , <font color=\"#5cb85c\">" + self.data_string[i][10]
+ "</font>"
                elif float(self.data_string[i][10]) > 10000000000 and j == 10:
                    temp += " , <font color=\"#0275d8\">" + self.data_string[i][10]
+ "</font>"
                else:
                    temp += " , " + self.data_string[i][j]
            object.setText(temp)
            object.move(100,100+i)
            self.vbox.addWidget(object)

        self.widget.setLayout(self.vbox)
        self.move(180,100)


class Main(QMainWindow):
    def __init__(self, parent = None):
        QMainWindow.__init__(self, parent)
        self.InitUi()

    def InitUi(self):
        self.button = self.CreateButton()
        self.w = None  # No external window yet.
        self.data_string = []

    def CreateButton(self):
        button = QPushButton(self, text = "OK")
        button.resize(30,30)
        button.clicked.connect(lambda: self.setAmount())
        button.clicked.connect(lambda: print("OK"))
        return button

    def shell_sort(self):
        n = len(self.data_string)
        # Rearrange elements at each n/2, n/4, n/8, ... intervals
        interval = n // 2
        while interval > 0:
            for i in range(interval, n):
                temp = self.data_string[i]
                j = i
                while j >= interval and float(self.data_string[j - interval][10]) >
float(temp[10]):
                    self.data_string[j] = self.data_string[j - interval]
                    j -= interval
                self.data_string[j] = temp
            interval //= 2

    def setAmount(self):
        self.data_string = []
        #把 csv 檔的資料存成 list(data_string,data_number)
        with open('500_constituents_financial.csv') as csv_file:
```

```
            csv_reader = csv.reader(csv_file, delimiter=',')
            check = False #第一列不存取
            for row in csv_reader:
                if check:
                    self.data_string.append(row)
                check = True
        self.shell_sort()
        self.w = AnotherWindow(self.data_string)
        self.output_file()

    def output_file(self):
        f = open("demofile2.txt", "w")
        for i in self.data_string:
            for j in i:
                f.write(j+'\n')

def main():
    app = QApplication(sys.argv)
    main_window = Main()
    main_window.show()
    app.exec_()

if __name__ == "__main__":
    main()


#quick sort
data = [89, 34, 23, 78, 67, 100, 66, 29, 79, 55, 78, 88, 92, 96, 96, 23]

def quicksort(data, left, right): # 輸入資料，和從兩邊開始的位置
    if left >= right :            # 如果左邊大於右邊，就跳出function
        return

    i = left                      # 左邊的代理人
    j = right                     # 右邊的代理人
    key = data[left]                 # 基準點

    while i != j:
        while data[j] > key and i < j:   # 從右邊開始找，找比基準點小的值
            j -= 1
        while data[i] <= key and i < j:  # 從左邊開始找，找比基準點大的值
            i += 1
        if i < j:                        # 當左右代理人沒有相遇時，互換值
            data[i], data[j] = data[j], data[i]

    # 將基準點歸換至代理人相遇點
    data[left] = data[i]
    data[i] = key

    quicksort(data, left, i-1)   # 繼續處理較小部分的子循環
    quicksort(data, i+1, right)  # 繼續處理較大部分的子循環

quicksort(data, 0, len(data)-1)
print(data)
```