Exercise 1: Sorting Lists of Pairs (50 marks)

(Note that in this exercise, questions may be attempted without having completed all previous questions)

Consider the abstract Python class below:

In [ ]:
```
class Comparison:
    def __init__(self):
        pass

    #returns True if the two objects are comparable,
    #False otherwise
    def areComparable(self, other):
        raise Exception("NotImplementedException")

    #returns True if the two objects are equal,
    #False otherwise
    def __eq__(self, other):
        raise Exception("NotImplementedException")

    #returns True if self > other,
    #False otherwise
    def __gt__(self, other):
        raise Exception("NotImplementedException")

    #returns True if self < other,
    #False otherwise
    def __lt__(self, other):
        raise Exception("NotImplementedException")

    def __ne__(self, other):
        return not self.__eq__(other)

    def __ge__(self, other):
        return self.__eq__(other) or self.__gt__(other)

    def __le__(self, other):
        return self.__eq__(other) or self.__lt__(other)

    def compare(self, other):
        if self.areComparable(other) is False:
            return None
        elif self == other:
            return 0
        elif self < other:
            return -1
        elif self > other:
            return 1
        else:
            assert False, "Inconsistent operation definitions"
```

The Comparison class provides a way to model items that are not always comparable. For instance, the pair of integers $(5, 10)$ is greater than $(4, 8)$, but it is not comparable to $(6, 5)$, because $5 < 6$ and $10 > 5$.

In this exercise, we will look into different ways to sort list of pairs. We will suppose that the pairs in a list are all different.

Question 1.1 (10 marks)

The rules of comparison between two Pairs (a, b) and (c, d) are:

- $(a, b) == (c, d)$ if and only if $a == c$ and $b == d$,
- $(a, b) > (c, d)$ if and only if ($a > c$ and $b \geq d$) or ($a \geq c$ and $b > d$),
- $(a, b) < (c, d)$ if and only if ($a < c$ and $b \leq d$) or ($a \leq c$ and $b < d$).

We say that (a, b) and (c, d) are comparable if

- $(a, b) == (c, d)$, or
- $(a, b) > (c, d)$, or
- $(a, b) < (c, d)$.

We ask that you implement the rules above in the class called Pair below, by completing the functions that have a comment "#TODO" in the body. Note that the class Pair inherits from Comparison.

In [ ]:
```
class Pair(Comparison):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return "({},{})".format(self.x, self.y)

    def areComparable(self, other):
        #TODO
        pass

    def __eq__(self, other):
        #TODO
        pass

    def __gt__(self, other):
        #TODO
        pass

    def __lt__(self, other):
        #TODO
        pass
```

We provide a test class below. You don't need to edit it, but the class Pair you write need pass these tests!

In [ ]:
```
import unittest

class TestPair(unittest.TestCase):
    def setUp(self):
        self.v00 = Pair(0,0)
        self.v01 = Pair(0,1)
        self.v10 = Pair(1, 0)
        self.v11 = Pair(1, 1)
        self.v21 = Pair(2, 1)
        self.v31 = Pair(3, 1)
        self.v23 = Pair(2, 3)
        self.v23other = Pair(2, 3)

    def test_areComparable(self):
        self.assertTrue(self.v00.areComparable(self.v01))
        self.assertTrue(self.v01.areComparable(self.v00))

        self.assertTrue(self.v11.areComparable(self.v00))
        self.assertTrue(self.v00.areComparable(self.v11))

        self.assertTrue(self.v21.areComparable(self.v23))
```

```
        self.assertTrue(self.v23.areComparable(self.v21))

        self.assertTrue(self.v23.areComparable(self.v23))

        self.assertTrue(self.v23.areComparable(self.v23other))
        self.assertTrue(self.v23other.areComparable(self.v23))

        self.assertFalse(self.v01.areComparable(self.v10))
        self.assertFalse(self.v10.areComparable(self.v01))

        self.assertFalse(self.v31.areComparable(self.v23))
        self.assertFalse(self.v23.areComparable(self.v31))

    def test_eq(self):
        self.assertTrue(self.v00 == self.v00)
        self.assertTrue(self.v21 == self.v21)
        self.assertTrue(self.v23 == self.v23)
        self.assertTrue(self.v23 == self.v23other)

        self.assertFalse(self.v00 == self.v11)
        self.assertFalse(self.v21 == self.v11)
        self.assertFalse(self.v21 == self.v23)

    def test_ne(self):
        self.assertFalse(self.v00 != self.v00)
        self.assertFalse(self.v21 != self.v21)
        self.assertFalse(self.v23 != self.v23)
        self.assertFalse(self.v23 != self.v23other)

        self.assertTrue(self.v00 != self.v11)
        self.assertTrue(self.v21 != self.v11)
        self.assertTrue(self.v21 != self.v23)

    def test_gt(self):
        self.assertTrue(self.v01 > self.v00)
        self.assertTrue(self.v10 > self.v00)
        self.assertTrue(self.v31 > self.v21)

        self.assertFalse(self.v00 > self.v01)
        self.assertFalse(self.v00 > self.v01)
        self.assertFalse(self.v21 > self.v31)

        self.assertFalse(self.v10 > self.v01)
        self.assertFalse(self.v01 > self.v10)
        self.assertFalse(self.v31 > self.v23)
        self.assertFalse(self.v23 > self.v31)

    def test_lt(self):
        self.assertFalse(self.v01 < self.v00)
        self.assertFalse(self.v10 < self.v00)
```

```
        self.assertFalse(self.v31 < self.v21)

        self.assertTrue(self.v00 < self.v01)
        self.assertTrue(self.v00 < self.v01)
        self.assertTrue(self.v21 < self.v31)

        self.assertFalse(self.v10 < self.v01)
        self.assertFalse(self.v01 < self.v10)
        self.assertFalse(self.v31 < self.v23)
        self.assertFalse(self.v23 < self.v31)
```

In [ ]:
```
test = TestPair()
suite = unittest.TestLoader().loadTestsFromModule(test)
unittest.TextTestRunner().run(suite)
```

In the following questions, we suppose that we have a set of Pairs, and that not every two pairs in that set are comparable.

Question 1.2 (10 marks)

Given a list l of Pairs in no particular order, use a sorting algorithm similar to *selection sort* to sort l such that, at the end of the algorithm, for every two pairs $l[i] = (a, b)$ and $l[j] = (c, d)$ at index i and j in l, respectively, with $i < j$, we have:

- either $(a, b) \leq (c, d)$,
- or (a, b) and (c, d) are not comparable.

In [ ]:
```
def pairSort(l):
    #TODO
    pass
```

Again, we provide a test class below. You don't need to edit it, but the method pairsort you wrote needs to pass these tests!

In [ ]:
```
import unittest
import random

class TestPairSort(unittest.TestCase):
    def setUp(self):
        self.PairClass = Pair
        self.sortAlgo = pairSort
```

```
    def test1(self):
        #in this test we suppose x=0 for all entries
        #this means that the algorithm should do a regular bubble sort
        #we also suppose everything is already ordered
        l = [self.PairClass(0, 1), self.PairClass(0, 3), self.PairClas
s(0, 4), self.PairClass(0, 6),
            self.PairClass(0, 8), self.PairClass(0, 15), ]
        self.sortAlgo(l)
        #for item in l: print(item)
        self.checkorder(l)

    def test2(self):
        #in this test we suppose x=0 for all entries
        #this means that the sort algorithms should do a "regular" sor
t
        l = [self.PairClass(0, 8), self.PairClass(0, 4), self.PairClas
s(0, 3), self.PairClass(0, 9),
            self.PairClass(0, 10), self.PairClass(0, 5), ]
        self.sortAlgo(l)
        #for item in sortedl: print(item)
        self.checkorder(l)


    def test3(self):
        #in this test we suppose x and y are not fixed
        #we also suppose that everything is in a good order
        l = [self.PairClass(5, 8), self.PairClass(5, 10), self.PairCla
ss(6, 10), self.PairClass(7, 12),
            self.PairClass(5, 12), self.PairClass(9, 12), ]
        sortedl = self.sortAlgo(l)
        #for item in sortedl: print(item)
        self.checkorder(l)

    def test4(self):
        #in this test we suppose x and y are not fixed
        #we suppose there is no two pairs that are comparable
        l = [self.PairClass(8, 8), self.PairClass(5, 10), self.PairCla
ss(10, 8), self.PairClass(12, 5)]
        self.sortAlgo(l)
        #for item in l: print(item)
        self.checkorder(l)

    def test5(self):
        #in this test we suppose x and y are not fixed
        #the input is in arbitrary order
        #we only test one case
        l = [self.PairClass(5, 8), self.PairClass(10, 10), self.PairCl
ass(12, 5), self.PairClass(9, 5),
             self.PairClass(5, 10), self.PairClass(7, 2), self.PairCla
```

```
ss(6, 10) ]
        self.sortAlgo(l)
        #for item in l: print(item)
        self.checkorder(l)

    def test6(self):
        #in this test we suppose x and y are not fixed
        #the input is in arbitrary order
        #we generate many cases
        for size in range(0, 100):
            #create a list
            l = []
            xs = random.sample(range(0, 100), size)
            xy = random.sample(range(0, 100), size)
            for i in range(0, size):
                l.append(self.PairClass(xs[i], xy[i]))
            self.sortAlgo(l)
            #for item in l: print(item)
            self.checkorder(l)

    def checkorder(self, l):
        for i in range(0, len(l)-1):
            for j in range(i, len(l)-1):
                if l[i].areComparable(l[j]):
                    self.assertTrue(l[i] <= l[j], "{} and {} are not w
ell ordered".format(l[i], l[j]) )
```

In [ ]:
```
test = TestPairSort()
suite = unittest.TestLoader().loadTestsFromModule(test)
unittest.TextTestRunner().run(suite)
```

Question 1.3 (5 marks)

Suppose we have a list of Pairs of integers. We want to implement a comparison mechanism between two pairs of integers using a function **pairKey**, which takes a pair as input and outputs a single number. The function pairKey should be such that for two pairs (a, b) and (c, d),

- $(a, b) > (c, d)$ implies $pairKey((a, b)) > pairKey((c, d))$,
- $(a, b) < (c, d)$ implies $pairKey((a, b)) < pairKey((c, d))$,
- $(a, b) == (c, d)$ implies $pairKey((a, b)) == pairKey((c, d))$.

In [ ]:
```
def pairKey(self):
    #TODO
    return #TODO
```

If you have defined the function pairKey correctly, we should now be able to use the built-in sort function of Python to sort our Pairs:

In [ ]:
```
def pairSortWithKey(l):
    l.sort(key = pairKey)
```

(For more information on what the above does, please refer to https://docs.python.org/3/howto/sorting.html (https://docs.python.org/3/howto/sorting.html))

In [ ]:
```
class TestKeyPairSort(TestPairSort):
    def setUp(self):
        self.PairClass = Pair
        self.sortAlgo = pairSortWithKey
```

In [ ]:
```
test = TestKeyPairSort()
suite = unittest.TestLoader().loadTestsFromModule(test)
unittest.TextTestRunner().run(suite)
```

Question 1.4 (5 marks)

Prove that the pairKey function you have defined above provides a guarantee that, if used as key for Python's sort, then the list will be sorted as stated in Queston 1.2.

(write your proof here)

Question 1.5 (5 marks)

(We have *not* covered the concept of stability in this unit. If you are not yet familiar with this concept, learning about it is part of the question.)

Use the fact that Python's sort is stable to provide a simple solution to sort a list of pairs as stated in Question 1.2. Use unit testing as in the previous questions to test your solution. Both your solution and the unit testing will be assessed. (no need to rewrite *new* test cases; use the old ones).

In [ ]:
```
def pairSortUsingStability(l):
    #TODO
    pass
```

In [ ]:
```
class TestPairSortUsingStability(TestPairSort):
    def setUp(self):
        self.PairClass = #TODO
        self.sortAlgo = #TODO
```

In [ ]:
```
test = TestPairSortUsingStability()
suite = unittest.TestLoader().loadTestsFromModule(test)
unittest.TextTestRunner().run(suite)
```

Question 1.6 (10 marks)

Define a function pairSortFast that takes a list l of n Pairs as input and sort this list in $O(n)$ time in the worst case (not the amortised worst case). Suppose that any pair (a, b) in l is such that a and $b \in \{0, \ldots, U\}$, where U is a small integer. Hint: *a clever hashing function* may help.

In [ ]:
```
#TODO
```

In [ ]:
```
class TestPairSortFast(TestPairSort):
    def setUp(self):
        self.PairClass = #TODO
        self.sortAlgo = #TODO
```

In [ ]:
```
test = TestPairSortFast()
suite = unittest.TestLoader().loadTestsFromModule(test)
unittest.TextTestRunner().run(suite)
```

Question 1.7 (5 marks)

Thoroughly benchmark all the sorting algorithms you have written, and plot their running time as a function of the size of the input list. You may use methods and code seen in the lectures or tutes and pracs during the semester.

Exercise 2 (25 marks)

We are again interested in lists of pairs of integers. However, this time, we want to store these pairs into a rooted tree T. More precisely, each node of T will store a pair of l. This tree T will have the property that for each inner node k that stores a pair (a, b), the pair (c, d) stored at a node in the subtree of k satisfies $(a, b) \leq (c, d)$. Furthermore, the pairs stored at sibling nodes must be non-comparable.

We will suppose that the root of the tree is a dummy pair $(-1, -1)$.

You need to provide a unit test for each question in this exercise where code is required. Each question is evaluated on the correctness of your code and the thoroughness of your unit tests.

Question 2.1 (10 marks)

Define a class that stores the tree T for any given list of pairs. The __init__() function must take a list as input and build the tree accordingly. An insert() function must allow the insertion of a pair to an initialised tree. In $O()$, what is the running time complexity of the functions you defined?

Question 2.2 (5 marks)

In the class you have defined in Question 2.1, can there be multiple trees that store the same list of pairs? (including after insertions?). Prove your answer.

(write your answer here).

Question 2.3 (5 marks)

We now relax the constraint that T must be "exactly" a tree. A child node which stores a pair (c, d) must now have as a parent all the nodes that have a pair (a, b) such that $(a, b) \leq (c, d)$.

Design a class (it may be based on the previous one) to allow and build this representation.

Question 2.4 (5 marks)

Using the class written in Question 2.3, design and code an algorithm that traverses all the nodes in that data structure and outputs the stored pairs in sorted order (as defined in Question 1.2).

Exercise 3 (25 marks)

You are given a tree T with an integer value (negative or positive) at each node. We want to select a subtree of T (with the same root) that maximises the sum of the values at its nodes. Note that the answer is trivially T if all nodes have a non-negative value.

Question 3.1 (5 marks)

Define a data structure to store T.

Question 3.2 (15 marks)

Design and code a dynamic programming algorithm that solves this problem.

Question 3.3 (5 marks)

Write and run unit tests and performance tests.