Assessed Coursework

This set of exercises is **Assessed Coursework**, constituting **20% of the final mark** for this course. It is due on **Wednesday, 19 April 2023 at 5:00pm (BST)**.

The work should be yours only. You may not work in groups or even discuss these problems with anyone else. You have to upload your work on Moodle at the Gradescope link that will be provided. By submitting your solutions, you would be confirming that you have read and understood the School's rules on Plagiarism and Assessment Offences, which can be found here and here.

Your submission for this coursework should consist of two files: A Python file CompactList.py for Question 1 and SumFreeLists.pdf, for your solution to Question 2. The contents of your work must remain anonymous, so **do not include your name or your student number** in the files. Do not submit anything directly to your lecturer or class teacher. Instead, please put your candidate number in both files, for example, in a comment at the top of your Python code.

The deadline is sharp. Late work carries an automatic penalty of 5 deducted marks (out of 100) for every 24-hour period that the coursework is late. Submission of answers to this coursework is mandatory. Without any submission, your mark for this course is incomplete, and you may not be eligible for the award of a degree.

The two questions in the Assessed Coursework have equal weights.

Use of Python Facilities

In the Python implementations of the algorithms, you are not allowed to use any Python library functions beyond what might be explicitly stated in the questions. The aim of this assessment is **not** to measure your proficiency in using Python facilities to solve problems. The questions test your algorithm design and analysis skills, as well as your ability to implement algorithms in Python.

Question 1: Compact List Data Structure

In this task, you have to implement a data structure called a *compact list*. Compact lists are compact ways of representing large sets of integers, where the set contains long runs of consecutive integers. For example, the set of integers

$$\{-1000, -999, -998, \ldots, -10\} \cup \{100, 101, 102, \ldots, 200\} \cup \{1001, 1002, 1003, \ldots, 1999\}$$

is represented by the compact list $\langle -1000, -9, 100, 201, 1001, 2000\rangle$. The meaning of this list is that the numbers start at -1000, then the first number that is *not* in the list is -9, then the first number after -9 that is again in the set is 100, then the first number after 100 that is not in the set is 201, then the first number after 201 that is in the set is 1001, and the first number after 1001 that is not in the set is 2000. Since the list ends here, no number from 2000 and onwards is in the set.

As another example, the set $\{-5, -4, -3, 0, 1, 2, 7, 8, 9, 10\}$ is represented by the compact list $\langle -5, -2, 0, 3, 7, 11\rangle$.

Compact lists can also have an odd length. For example, the compact list $\langle 5, 70, 80\rangle$ represents the set $\{5, 6, 7, \ldots, 69, 80, \ldots\}$. Thus, we understand a compact list of odd length as representing a set which after a certain value (in this case 80) contains all integers from that value on. In practice, such as when we implement compact lists for integers, we let the set go up to a maximum value, which will fix as a variable `MAX`. We also set `MIN` as the minimum value of an integer included in a set. In other words, the sets that we represent as compact lists will be subsets of the universal set {`MIN`, `MIN`+1,. . . ,`MAX`}.

In general, a compact list of integers represents a set of integers as a list $\langle x_0, x_1, x_2, \dots \rangle$ of integers such that $x_0 < x_1 < x_2 < \dots$, with the meaning that all integers from x_0 to $x_1 - 1$, as well as all from x_2 to $x_3 - 1$, and so on, are included in the set. Thus, the included integers start at x_0, then the first integer larger than x_0 not to be included is x_1, then the next integer after x_1 to be included is x_2 and so on. The empty set is represented as an empty list (of length 0). If the compact list has an odd length, it contains all values from the last entry in the list up to the maximum value `MAX`. As a final example, the set of all integers is represented by the one-element compact list ⟨`MIN`⟩: that is, it consists of the single number `MIN`.

You have to implement a class called `CompactList` which implements various set operations, as described below. Before we describe the various functionalities, we quickly review the basic set operations and relations, to establish notation.

Given sets A and B,

- their *union* is the set $A \cup B$ defined as the set of all elements that are in A or in B or in both A and B,
- their *intersection* is the set $A \cap B$ defined as the set of all elements that are in A and in B,
- their *difference* is the set $A \setminus B$ defined as the set of all elements that are in A but not in B.

The set A is a *subset* of set B, written $A \subseteq B$, if each element of A is also an element of B. The sets A and B are considered to be *equal* if they have the same elements.

If all the sets that we consider are contained in some universal set U, then the *complement* of set A is the set $A^c = U \setminus A$. Below, whenever we mention union, intersection, difference or complement, we always refer to the sets represented by compact lists of integers. For example, the union of the set $A = \{10, \dots, 19\}$ represented by the compact list $\langle 10, 20 \rangle$, and the set $B = \{30, \dots, 39\}$ represented by the compact list $\langle 30, 40 \rangle$, is the set $A \cup B$ represented by the compact list $\langle 10, 20, 30, 40 \rangle$. As another example, the intersection of the set A with the set $C = \{15, \dots, 24\}$, represented by the compact list $\langle 15, 25 \rangle$, is the set $A \cap C = \{15, 16, 17, 18, 19\}$ represented by the compact list $\langle 15, 20 \rangle$.

For the purposes of our set operations, the universal set is the set {`MIN`,`MIN`+1,. . . ,`MAX`} of integers, and so we take complements with respect to this set. For example, the complement of set A is the set {`MIN`, . . . , 9} ∪ {20, . . . }, represented by the compact list ⟨`MIN`, 10, 20⟩.

The following list (a)–(m) describes the functionalities that you have to implement. You can use the template file `CompactList.py` on Moodle.

(a) Constructor for a compact list from a Python list: `__init__(alist)`
Creates a compact list to represent the set of integers given by the Python list `alist`.
As an example, `CompactList([3,4,5,8])` initialises the compact list $\langle 3, 6, 8, 9 \rangle$.

(b) Instance method: `cardinality()`
Returns the number of elements in the set represented by this compact list.
The worst-case time should be $O(n)$ where n is the length of the compact list.

(c) Instance method: `insert(value)`
Inserts `value` into the set represented by this compact list if `value` is not already contained in the set.
The worst-case time should be $O(n)$ where n is the length of the compact list.

(d) Instance method: `delete(value)`
Removes `value` from the set represented by this compact list if `value` is in the set, otherwise do nothing.
The worst-case time should be $O(n)$ where n is the length of the compact list.

(e) Instance method: `contains(value)`
Returns `True` if `value` is an element of the set represented by this compact list, otherwise returns `False`.

The worst-case time should be $O(\log n)$, where n is the length of this compact list.

(*f*) Instance method: `subsetOf(cl)`
Returns `True` if the set represented by this compact list is a subset of the set represented by `cl`, otherwise returns `False`.
The worst-case time should be $O(m + n)$ where m is the length of this compact list and n is the length of `cl`.

(*g*) Instance method: `equals(cl)`
Returns `True` if the set represented by this compact list has the same elements as the set represented by `cl`, otherwise `False`.
The worst-case time should be $O(m + n)$ where m is the length of this compact list and n is the length of `cl`.

(*h*) Instance method: `isEmpty()`
Returns `True` if the compact list represents the empty set, `False` if not.
The worst-case running time should be $O(1)$.

(*i*) Instance method: `complement()`
Returns the complement of the set represented by this compact list, also represented as a compact list, where the complement is with respect to the universal set of integers determined by `MIN` and `MAX`.
The worst-case time should be $O(n)$ where n is the length of the compact list.

(*j*) Instance method: `union(cl)`
Returns the union of the set represented by this compact list with the set represented by the compact list `cl`.
The worst-case time should be $O(m + n)$ where m is the length of this compact list and n is the length of `cl`.

(*k*) Instance method: `intersection(cl)`
Returns the intersection of the set represented by this compact list with the set represented by the compact list `cl`.
The worst-case time should be $O(m + n)$ where m is the length of this compact list and n is the length of `cl`.

(*l*) Instance method: `difference(cl)`
Returns the set difference `this\cl` of the set represented by this compact list with the set represented by `cl` removed.
The worst-case time should be $O(m + n)$ where m is the length of this compact list and n is the length of `cl`.

(*m*) Instance method: `__str__()`
Returns a string representation of this compact list ⟨`x_0, x_1, x_2, ...`⟩ in the form `[x_0,x_1 - 1] U [x_2,x_3 - 1] U ...`
If this compact list is empty, returns `empty`.

Hints:

1. Finding the union or the intersection of two compact lists is very similar to the `Merge()` method that we saw when we looked at Merge Sort.
2. Once you have implemented `union(cl)` and `complement()` and you have made sure that they work, then it is very simple to implement `intersection(cl)`, `difference(cl)`, `subsetOf(cl)`, `equals(cl)`, as well as `insert(value)` and `delete(value)`, by writing these set operations and relations in terms of `union(cl)` and `complement()`. For example, $A \cap B = (A^c \cup B^c)^c$ and $A \setminus B = A \cap B^c$.

Remember to explain all your code with *comments*, when needed, and to *indent* properly. Do not use any Python modules or libraries, except the standard functionalities of Python lists.

[50 points]

Question 2: Sum-Free Lists

Given three sets A, B, and C, we would like to determine whether there exists a triple (a, b, c) such that $a \in A$, $b \in B$, $c \in C$, and $a + b + c = 0$.

The following Python function receives three lists representing these sets as input and claims to output such a triple as a list `[a,b,c]`, if it exists, or `None` otherwise.

```python
def SumFreeLists(list1,list2,list3):
    list2.sort()
    list3.sort()
    for i in range(0,len(list1)-1):
        j=0
        k=len(list3)-1
        while j<len(list2) and k>=0:
            if list1[i]+list2[j]+list3[k]<0:
                j += 1
            elif list1[i]+list2[j]+list3[k]>0:
                k -= 1
            else:
                return [list1[i], list2[j], list3[k]]
    return None
```

Prove that the algorithm is correct using loop invariants. Note that you will need two separate loop-invariant arguments: one for the inner loop and one for the outer loop.

[50 points]