Assessed Coursework

This set of exercises is **Assessed Coursework**, constituting **20% of the final mark** for this course. It is due on **Wednesday, 6 April 2022 at 4pm (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 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 a single ZIP file, named YourCandidateNumber.zip. (You can find your candidate number on LSE For You.) This ZIP file should contain two files: minheaplist.py and SCSL.pdf. 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.

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: Heaplist Data Structure

For this question, you will implement a data structure called min-heaplist, which is a variant of a min-heap. In a min-heaplist, a collection of min-heaps are stored in a circular doubly linked list, which we call *rootlist*. The head pointer (called `min`) of this linked list points to the min-heap that contains the minimum value stored in the whole data structure. The rootlist is a doubly linked list such that each item in the list has pointers to the next and the previous item in the list. The rootlist is also circular; that is the next pointer of the last item points to the head of list and the previous pointer of the first item points to the last item. The idea behind the design of a min-heaplist is to make `insert()` and `decreaseKey()` operations in $O(1)$ time, while the performance of extractMin degrades compared to a normal min-heap. In situations where `decreaseKey()` is called much more often than `extractMax()`, this approach would lead to improved running times.

We will remove the restriction that individual min-heaps need to be nearly complete binary trees, filling up layer by layer, and left to right in each layer. Any binary tree structure can be a min-heap, as long as the min-heap property is satisfied: for each internal node, the value stored in this node is less than or equal to the value stored in any child. Therefore, we will not represent min-heaps as lists, but as dynamic linked data structures. We will have a Node class to represent a node of a min-heap.

```
class Node:
    def __init__(self,val,le,ri,pa):
        self.value = val
        self.left=le
        self.right=ri
        self.parent=pa
```

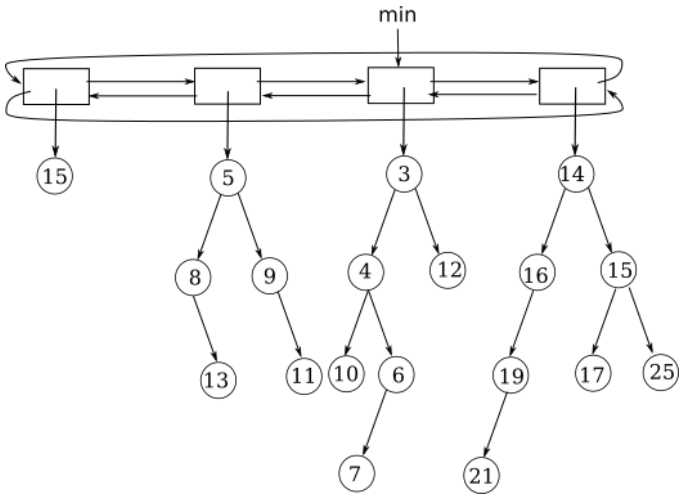
Each item in the rootlist of a min-heaplist will point to the root of a min-heap in the collection. The following class will represent the items in the rootlist.

```
class Item:
    def __init__(self,aroot,p,n):
        self.heap=aroot
        self.previous=p
        self.next=n
```

Finally, we can represent a min-heaplist using a pointer to the head of its rootlist.

```
class MinHeaplist:
    def __init__(self):
        self.min = None
```

Below is an example how a min-heapList could look like. (Parent links are not shown to reduce clutter in the picture.)



The operations supported by a min-heaplist are listed below.

- The `insert(x)` function insert a new value to the min-heaplist in $O(1)$ time, by adding a single-element min-heap to the rootlist and ensuring that `min` attribute is maintained properly.
- The `linkheaps(h1,h2)` function combines two min-heaps to make a single min-heap out of them. The pointers `h1` and `h2` point to the roots of the min-heaps, respectively. A pointer to the root of the new min-heap is returned.
- The `extractMin()` function removes and returns an element with the minimum value. This can done simply by disconnecting any children of the node to be removed from their parent and adding them as an individual min-heap to the rootlist. After the minimum-value element is extracted, `extractMin()` function goes through the entire rootlist and combines

all the min-heaps in the collection into a single min-heap, by calling `linkheaps()` function for two min-heaps at a time. This "clean-up" operation tries to prevent the build-up of a large number of min-heaps in the collection.

- The `decreaseKey(n,k)` function decreases the value stored at the node `n` to a given value `k`, while maintaining the properties of a min-heaplist properly. This can be done in $O(1)$ time, by moving node `n` — as a min-heap of a single element with the new key `k` — as well as any min-heap rooted at a child of `n` into the rootlist.
- The `union(H)` function combines the current min-heaplist (the one pointed to by `self`) with another min-heaplist `H` given as an argument. The resulting min-heaplist is then stored in the current min-heaplist. This can be done simply by combining rootlists of the two min-heaplists.

Question 2: The Shortest Common Supersequence Problem

For a given sequence S, a *supersequence* is a sequence T that has S as a subsequence. For example, $\langle 8, 2, 9, 3, 6, 9, 2, 7, 3\rangle$ is a supersequence of $\langle 2, 9, 6, 7\rangle$. In the *shortest common supersequence problem*, we are given two sequences X and Y and we are asked to find a shortest sequence Z that is a supersequence of both X and Y. In this question, we are interested in computing only the length of a shortest common supersequence.
The following Python function claims to find the length of a shortest common supersequence of two input sequences, given as Python lists. 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. (No loop invariant is required for the initialisation loops.)

```
1   def SCSLength(X, Y):
2     m = len(X)
3     n = len(Y)
4     T = [[0 for x in range(n + 1)] for y in range(m + 1)]
5
6     for i in range(m + 1):
7         T[i][0] = i
8     for j in range(n + 1):
9         T[0][j] = j
10
11    for i in range(1, m + 1):
12        for j in range(1, n + 1):
13            if X[i - 1] == Y[j - 1]:
14                T[i][j] = T[i - 1][j - 1] + 1
15            else:
16                T[i][j] = 1 + min(T[i - 1][j], T[i][j - 1])
17
18    return T[m][n]
```
