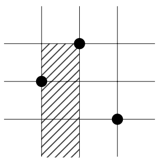
E-402-STFO PROBLEMS FOR MODULE B

CREATED BY HENNING ULFARSSON

This is the second module concernded with permutations. You get a perfect score for this module by getting 70 points or more.

1. Mesh patterns (25) points

Mesh patterns are a common generalization of many other types of patterns, such as the consecutive and classical patterns we have discussed in class. They were introduced by Petter Brändén and Anders Claesson in the paper *Mesh patterns and the expansin of permutation statistics as sums of permutation patterns*, available at `http://www.combinatorics.org/ojs/index.php/eljc/article/download/v18i2p5/pdf`. For example the mesh pattern



occurs in the permutation 136452 as the subsequences 362, 342 and 452. Note that the subsequence 352 is not a valid occurrence since the 4 occupies one of the shaded regions in the pattern (between the 2 and the 3 and below the 3).

It is convenient to represent mesh patterns as a tuple containing the underlying classical pattern and a sorted list of the shaded squares (where a square is represented by its lower left corner). Our example would be represented as

```
(Permutation([2,3,1]), [(1,0), (1,1), (1,2)])
```

Problem 1 (15 points)**.** Write a function `mBp1` that takes as input a permutation (as a `Permutation` object) and a mesh pattern and returns `True` if the permutation contains the pattern, but `False` otherwise. For example:

```
Input: mBp1(Permutation([1,3,6,4,5,2]), (Permutation([2,3,1]), [(1,0),(1,1),(1,2)]))
Output: True
```

Problem 2 (10 points)**.** Recall that we can implement the stack-sort operation as follows

```
def S(perm):

    if len(perm) <= 1:
    return perm

    lperm = list(perm)

    m  = max(lperm)
    mi = lperm.index(m)

    return S(lperm[:mi])+S(lperm[mi+1:])+[m]
```

Date: Updated November 19, 2013.

Also recall that we claimed that permutations `perm` such that `S(S(perm))` are the identity permutation are not recognized by classical patterns. They are however recognized by one classical pattern and one mesh pattern (with exaclyt one shaded square). Given that extra information, write a function `mBp2()` that outputs these two patterns in a tuple, such that the first entry in the tuple is the classical pattern and the second one is the mesh pattern. (Note that your function should take no input)

2. The RSK-correspondence (50 points)

The RSK-correspondence was discovered by Robinson, Schensted, Knuth and Schützenberger (and so should really be called the RSKS-correspondence). It is a bijection between permutations and pairs of Young tableaux. See p. 7 of `http://www.ms.unimelb.edu.au/publications/hd8.pdf` for the image of RSK of all permutations of length 3. For an interactive applet that constructs the tableaux go to `http://www.math.uconn.edu/~troby/Goggin/BumpingAlg.html`.

Problem 3 (15 points)**.** Write a function `mBp3(perm)` that given a permutation `perm` outputs the pairs of Young tableaux in the form of a tuple. Each tableaux should be a list of lists.

```
Input: mBp3(Permutation([3,5,2,1,4,6]))
Output: ([[1,4,6], [2,5], [3]], [[1,2,6], [3,5], [4]])
```

Note that your function should be able to handle permutations of length 1000.

Problem 4 (10 points)**.** Permutations whose Young tableaux have at most 3 cells in the first row avoid a mystery classical pattern. Write a function `mBp4()` that outputs this pattern.

Problem 5 (10 points)**.** Permutations whose Young tableaux have at most 3 cells in the first column avoid a mystery classical pattern. Write a function `mBp5()` that outputs this pattern.

Problem 6 (15 points)**.** Permutations whose Young tableaux are hook-shaped (i.e., every row, except the first one, has at most one cell) avoid two mystery classical patterns and two mystery mesh patterns. Write a function `mBp6()` that outputs these patterns in a set, e.g., if you put the patterns in a list called `L` then you should return `Set(L)`.

3. The Catalan Structures (50 points)

In this section we consider combinatorial structures enumerated by the Catalan numbers: we call those Catalan structures. Below is the skeleton of an abstract class for Catalan structures. You will be asked to replace `pass` with suitable code.

```
class Catalan(SageObject):
    """
    The base class for all Catalan structures
    """

    def __init__(self, obj=None):
        self.obj = self.neutral_element if obj is None else obj

    def _repr_(self):
        return "%s(%s)" % (self.__class__.__name__, repr(self.obj))

    def __eq__(self, other):
        return self.obj == other.obj
```

```
    def cons(self, other=None):
        raise NotImplementedError

    def decons(self):
        raise NotImplementedError

    def is_neutral(self):
        return self.obj == self.neutral_element

    def map_to(self, cls):
        """
        The image of self under the canonical bijection
        induced by the class of self and cls
        """
        pass

    @classmethod
    def structures(cls, n):
        """
        Generates all structures of size n
        """
        pass
```

Problem 7 (5 points)**.** Implement the `decons` method in the (concrete) Catalan class of 132-avoiding permutations[1] as below. (Think about how you can take a 132-avoiding permutation and decompose it into two 132-avoiding permutations.)

```
class Av132(Catalan):
    """
    The class of 132-avoiding permutations
    """
    neutral_element = []

    def cons(self, other=None):
        """
        Constructs a 132-avoiding permutation from
        the 132-avoiding permutations self and other
        """
        pass

    def decons(self):
        """
        Deconstructs the 132-avoiding permutation self
        into two 132-avoiding permutations:
        'decons' is the inverse of 'cons'
        """
        pass
```

To facilitate testing write a function `mBp7` that takes a permutation that avoids 132 and returns its decomposition into two smaller permutations that avoid 132

```
Input: mBp7(Av132([6,5,8,7,9,2,1,4,3]))
Output: (Av132([2,1,4,3]), Av132([2,1,4,3]))
```

[1] That is, the permutations avoiding the classical pattern 132

Problem 8 (5 points)**.** Implement the `cons` method for the 132-avoiding permutations. (Think about how you can take two 132-avoiding permutations and build a new 132-avoiding permutation, by somehow "gluing them" together.) To facilitate testing write a function `mBp8(avperm1, avperm2)` that takes two permutations `avperm1` and `avperm2` that avoid 132 and returns their "gluing".

```
Input: mBp8(Av132([2,1,4,3,5]), Av132([1,2,3,4]))
Output: Av132([6,5,8,7,9,10,1,2,3,4])
```

Problem 9 (7 points)**.** Implement the `cons` method for the class of Dyck paths, which are paths using steps $(1,1)$ and $(1,-1)$ that start at $(0,0)$, ends at $(2n,0)$, and never go below the x-axis. (You can learn about Dyck paths here - `http://mathworld.wolfram.com/DyckPath.html`.) In your code it is convenient to use `1` to denote the upstep $(1,1)$ and `0` the downstep $(1,-1)$, so the Dyck path `[1,1,0,0,1,0]` is the Dyck path that goes two steps up, then two steps down (and touches the x-axis), then one step up and finally, one step down.

```
class Dyck(Catalan):
    """
    The class of Dyck paths
    """
    neutral_element = []

    def cons(self, other=None):
        """
        Constructs a Dyck path from
        the Dyck paths self and other
        """
        pass

    def decons(self):
        """
        Deconstructs the Dyck path self
        into two Dyck paths:
        'decons' is the inverse of 'cons'
        """
        pass
```

To facilitate testing write a function `mBp9` that takes two Dyck paths and returns their "gluing".

```
Input: mBp9(Dyck([1,1,0,0]), Dyck([1,0]))
Output: Dyck([1,1,1,0,0,0,1,0])
```

Problem 10 (8 points)**.** Implement the `decons` method for the class of Dyck paths. To facilitate testing write a function `mBp10` that takes a Dyck path and returns its decomposition.

```
Input: mBp10(Dyck([1,1,1,0,0,0,1,0]))
Output: (Dyck([1,1,0,0]), Dyck([1,0]))
```

Problem 11 (10 points)**.** Fill in the missing code for the `structures` method in the abstract Catalan class. Use it for generating 132-avoiding permutations and Dyck paths. To facilitate testing write the function `mBp11(n)` that outputs the 132-avoiding permutations in a set.

```
Input: mBp11(3)
Output: Set([Av132([1,2,3]), Av132([2,1,3]), Av132([2,3,1]),\
Av132([3,1,2]), Av132([3,2,1])])
```

Problem 12 (15 points)**.** Fill in the missing code for the `map_to` method in the abstract Catalan class. Use it to map a 132-avoiding permutation to a Dyck path. To facilitate testing, write the function `mBp12` that inputs a 132-avoiding permutation and outputs a Dyck-path.

```
Input: mBp12(Av132([4,3,2,5,1]))
Output: Dyck([1,1,0,1,0,1,0,0,1,0])
```

School of Computer Science, Reykjavik University, Menntavegi 1, 101 Reykjavík, Iceland

E-mail address: `henningu@ru.is`