permutation

April 28, 2023

[1]:
```
def is_sorted(lst):
    for i, el in enumerate(lst[1:]):
        if el < lst[i]: # i is the index of the previous element
            return False
    return True
```

[2]:
```
def RFC(p, n=-1):
    #input: p: a reduced word in array,
    #       n: number of blocks, default: length of reduced word - 1
    #output: a 2-d array, whose element is a block, starting at the right side
    if n == -1: n = 2 #len(p)-1
    result = []
    for comp in Compositions(len(p), length=n, min_part=0).list():
        begin = 0
        current = []
        for l in comp:
            current.append(p[begin: begin+l])
            begin = begin+l

        #test whether current is a legal RFC
        flag = True
        for i,c in enumerate(current):
            #whether each block is sorted: RF condition
            if is_sorted(c) == False:
                flag = False
                break
            #whether first element in each block >= its block index: RFC␣
↪condition
            if len(c) != 0 and c[0] < (n-i):
                flag = False
                break
        if flag == True: result.append(current)
    return result
```

[3]:
```
def big(e,A):
    #A is ascendingly sorted
    #return -1 if no such element is found
    for i,j in enumerate(A):
```

```
        if j > e:
            return i
    return -1
```

[4]:
```
import copy
def is_ei_zero(rfc):
    #if ei(rfc) = 0, return true, otherwise false
    for i in range(len(rfc)-1):

        if len(rfc[i]) > len(rfc[i+1]): return False #the lemma to quickly␣
↪filter

        left = copy.deepcopy(rfc[i])
        right = copy.deepcopy(rfc[i+1])

        #matching
        for j in reversed(range(len(right))): #start by the largest element in␣
↪right
            current = big(right[j],left) #smallest element in left that's␣
↪bigger than right[i]
            if current != -1:
                del right[j]
                del left[current]

        if len(left) != 0: return False

    return True
```

[5]:
```
def numberDescents(p):
    descend = 0
    for i in range(len(p)-1):
        if p[i] > p[i+1]: descend += 1
    return descend
```

[6]:
```
def highest_weight(p,n=-1):
    #given a permutation(an array), return its highest-weight RFC and␣
↪corresponding reduced word
    #n is the number of blocks, default as the length of reduced words - 1
    rws = Permutation(p).reduced_words()
    if n == -1: n = numberDescents(rws[0])+1 #len(rws[0])-1
    print("===============")
    count = 0
    for rw in rws:
        for rfc in RFC(rw,n): #number of blocks default as len(rw)-1
            if is_ei_zero(rfc):
                count += 1;
                print(rw)
```

```
                print(rfc)
#     if count > 2: print("this permutation ",p, "has more than 2 hightest␣
 ↪weight" )
    print("this permutation ",p, "has ",count," highest weight" )
```

[7]:
```
n = 5 #w in S_n
w = Permutation([2,1,5,4,3])
```

[8]:
```
highest_weight(w.inverse(),n-1)
```

```
===============
[3, 4, 1, 3]
[[], [], [3, 4], [1, 3]]
[4, 3, 1, 4]
[[], [4], [3], [1, 4]]
[4, 1, 3, 4]
[[], [], [4], [1, 3, 4]]
this permutation  [2, 1, 5, 4, 3] has  3  highest weight
```

[9]:
```
R = PolynomialRing(ZZ, ['x1','x2','x3','x4', 'x5'])
R.inject_variables()
```

```
Defining x1, x2, x3, x4, x5
```

[10]:
```
X = SchubertPolynomialRing(ZZ)
schubPoly = X(w).expand()
schubPoly
```

[10]:
```
x0^3*x1 + x0^2*x1^2 + x0^3*x2 + 2*x0^2*x1*x2 + x0*x1^2*x2 + x0^2*x2^2 +
x0*x1*x2^2 + x0^3*x3 + x0^2*x1*x3 + x0*x1^2*x3 + x0^2*x2*x3 + x0*x1*x2*x3 +
x0*x2^2*x3
```

[11]:
```
schubPoly = x1^3*x2 + x1^2*x2^2 + x1^3*x3 + 2*x1^2*x2*x3 + x1*x2^2*x3 +␣
 ↪x1^2*x3^2 + x1*x2*x3^2 + x1^3*x4 + x1^2*x2*x4 + x1*x2^2*x4 + x1^2*x3*x4 +␣
 ↪x1*x2*x3*x4 + x1*x3^2*x4
schubPoly
```

[11]:
```
x1^3*x2 + x1^2*x2^2 + x1^3*x3 + 2*x1^2*x2*x3 + x1*x2^2*x3 + x1^2*x3^2 +
x1*x2*x3^2 + x1^3*x4 + x1^2*x2*x4 + x1*x2^2*x4 + x1^2*x3*x4 + x1*x2*x3*x4 +
x1*x3^2*x4
```

[12]:
```
T = crystals.Tableaux(['A',3],shape=[2,1,1])
T.demazure_character([2,1,3])
```

[12]:
```
x1^2*x2*x3 + x1*x2^2*x3 + x1*x2*x3^2 + x1^2*x2*x4 + x1*x2^2*x4 + x1^2*x3*x4 +
x1*x2*x3*x4 + x1*x3^2*x4
```

[13]:
```
first = x1^2*x2*x3 + x1*x2^2*x3 + x1*x2*x3^2 + x1^2*x2*x4 + x1*x2^2*x4 +␣
 ↪x1^2*x3*x4 + x1*x2*x3*x4 + x1*x3^2*x4
```

[14]:
```
T = crystals.Tableaux(['A',2], shape = [2,2])
T.demazure_character([2])
```

[14]:
```
x1^2*x2^2 + x1^2*x2*x3 + x1^2*x3^2
```

[15]:
```
second = x1^2*x2^2 + x1^2*x2*x3 + x1^2*x3^2
```

[16]:
```
T = crystals.Tableaux(['A',3],shape=[3,1])
T.demazure_character([3,2])
```

[16]:
```
x1^3*x2 + x1^3*x3 + x1^3*x4
```

[17]:
```
third = x1^3*x2 + x1^3*x3 + x1^3*x4
```

[18]:
```
keySum = first + second + third
```

[19]:
```
#check the corollary
keySum == schubPoly
```

[19]:
```
True
```

[ ]:
```

```