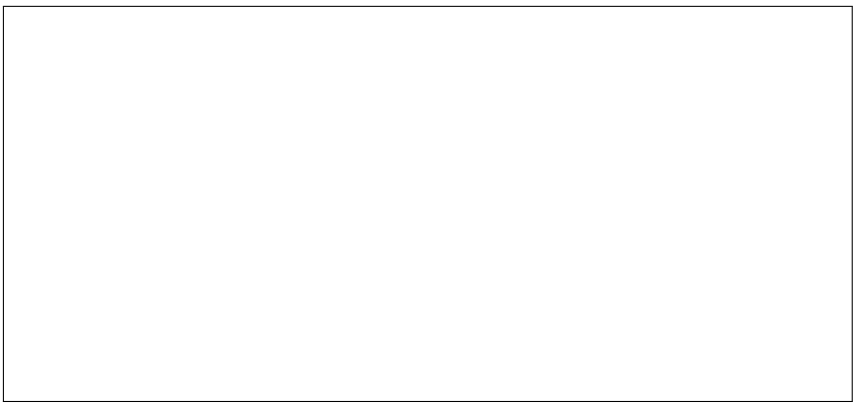
16.5



Permutation Groups

Contents

Introduction

This project is programmed in **Python 3.5**. Consult section A for program documentation, listings and information on the structure of the programming for the project, as appropriate. This report is written in LaTeX 2_ε.

1 Arithmetic

Q1. The functions `mp`(a, b) and `inv`(a) compute ab and a^{-1} (respectively), where $a, b \in S_n$. Here are some easily-checked results they give:

```
>>> inv([0,2,3,1,5,4])
[0, 3, 1, 2, 5, 4]
```

```
>>> mp([0,2,1,3],[0,1,3,2])
[0, 2, 3, 1]
```

2 Stripping

Q2. The stripping algorithm takes as input $X = (\pi_r)_{r=1}^k \subseteq S_n$ and returns $Y \subseteq S_n$. It is implemented as the function `strip`(X).

Theorem. $\langle Y \rangle = \langle X \rangle = G$.

Proof. ($Y \subseteq \langle X \rangle$): Let `X`, `A` refer to the current values of their respective variables, as declared in the code of `strip`, treating both as sets of their elements where appropriate. `X` is the current list of generators and `A` the stripping array with 0 representing empty components. Initially, `X` $= X$ and `A`$\backslash\{0\} = \emptyset$. Every step that modifies either of these either:

- Writes to `A` an element of `X`.
- Writes to `X` an element $a^{-1}x$, where $a \in$ `A`$\backslash\{0\}$ and $x \in$ `X`.
- Deletes an element of `X`.

Each of these preserves `X` $\cup$ (`A`$\backslash\{0\}$) $\subseteq \langle X \rangle$, so upon halting, $Y =$ `X` $\subseteq \langle X \rangle$.

($X \subseteq \langle Y \rangle$): By induction on $r \in [1, k]$. Let $r \in [1, k]$ and suppose truth for $s \in [1, r-1]$. Once π_r has been processed by the algorithm, it has the form $a_1^{-1} \ldots a_t^{-1} \pi_r$, where $\{a_i\}_{i=1}^t$ consists of permutations written to `A` at some point. Each of these must have been derived from π_s, $s < r$, so is in $Y \subseteq \langle Y \rangle$ by the induction hypothesis. It is either then output or deleted (in which case it's the identity); in either case, it is in $\langle Y \rangle$, so $\pi_r \in \langle Y \rangle$. □

The point of the algorithm is that it finds a simpler generating set for G, by removing some redundant generators and redundant transitions $i \mapsto j$ within the remaining generators. As such, there's a simple upper bound on $|Y|$. Each element of Y must have been derived from a uniquely-indexed permutation $\pi_r \in X$, and must've been written to `A` in its strict upper triangle (because, at the point $m \in [1, n]$ at which a permutation π is written, it has been checked to fix $[1, m-1]$, and so $\pi(m) \geq m$, being a bijection). Moreover, there is never an overwrite. Hence,

$$|Y| \leq \min(|X|, \tfrac{1}{2}n(n-1))$$

Q3. Some sample output of `strip`. The permutations are, as ordered in the input, (1 2), (1 2 4), (1 2 4 3), (2 4 3). They are converted into (1 2), (2 4), (3 4).

```
>>> S = strip([[0,2,1,3,4],[0,2,4,3,1],[0,2,4,1,3],[0,1,4,2,3]])
>>> S[0]
[[0, 2, 1, 3, 4], [0, 1, 4, 3, 2], [0, 1, 2, 4, 3]]
>>> for x in S[1]: print(x)

[0, 0, 0, 0, 0]
[0, 0, [0, 2, 1, 3, 4], 0, 0]
[0, 0, 0, 0, [0, 1, 4, 3, 2]]
[0, 0, 0, 0, [0, 1, 2, 4, 3]]
[0, 0, 0, 0, 0]
```

3 Orbits

Q4. Denote the orbit of $\alpha \in G$ in a group G by $G(\alpha)$. The natural bijection is:[1]

$$\Theta : [G : G_\alpha] \leftrightarrow G(\alpha), \quad gG_\alpha \mapsto g(\alpha)$$

This gives rise to the orbit-stabiliser theorem:

Suppose G is finite. Then $|G| = |G_\alpha|\frac{|G|}{|G_\alpha|} = |G_\alpha||G : G_\alpha| = |G_\alpha||G(\alpha)|$.

Q5. The function `orb`(G, j) lists the orbit and matching witnesses of $j \in [1, n]$ in $G \leq S_n$. E.g. note that the 2-component (counting from 0) of each witness below lists the orbit of 2.

```
>>> for x in orb([[0,2,3,1,4],[0,2,1,3,4]],2): print(x)

[2, 3, 1]
[[0, 1, 2, 3, 4], [0, 2, 3, 1, 4], [0, 2, 1, 3, 4]]
```

The algorithm sequentially applies generators to t, thus obtaining elements in $G(t)$. It keeps newly-discovered elements, and then applies the generators to each of these in succession, each time adding newly-discovered elements (of $G(t)$) and eventually attempting to apply the generators to these. The growing listing of $G(t)$ is eventually exhausted (as $G(t) \subseteq [1, n]$ is finite), and is returned. It is a full listing of $G(t)$ for the following reason:

Let $x \in G(t)$. Then $\exists g \in G : g(t) = x$. G is finite, so it consists entirely of finite products from its generating set Y (and not their inverses). Thus, $\exists (g_i)_{i=1}^k \in Y^* : g = g_n...g_1$ $(n \in \mathbb{N}_0)$. If $n \geq 1$, by induction on n, we may suppose $g_{n-1}...g_1(t)$ is in the listing (since it's immediate for $n = 0$). Then applying the generator g_n to $g_{n-1}...g_1(t)$ yields x, which is thus listed.

Witnesses are computed by simply multiplying together the permutation whose application discovered the point in question in the orbit with the previously-computed witness of the element to which this permutation was applied (bearing in mind that t has witness ι).

4 Stabilisers

Q6. Denote $H = G_\alpha$ to emphasise that everything in this section holds for any subgroup H of a finite group G. For $a \in G$, denote $\overline{a} = \varphi(a)$. Let $A = \{\overline{yt}^{-1}yt : y \in Y,\ t \in T\}$.

Lemma. $\forall a, b \in G \quad \overline{a\overline{b}} = \overline{ab}$.

Proof. $\overline{b} \in bH$, so $bH = \overline{b}H$, so $b^{-1}\overline{b} = (ab)^{-1}a\overline{b} \in H$, so $abH = a\overline{b}H$, so $\overline{ab} = \overline{a\overline{b}}$. □

Lemma. $\forall i \in [0, r] \quad t_{i+1} = \overline{y_i...y_1}$. *In particular,* $t_{r+1} = \overline{y_r...y_1} = \overline{x} = \overline{e} = t_1$.

Proof. By induction on i (and immediate for $i = 0$ since $\overline{e} = t_1$). Let $i \in [1, r]$ and suppose truth for $i - 1$. Then $t_{i+1} = \overline{y_i t_i} = \overline{y_i\overline{y_{i-1}...y_1}} = \overline{y_i y_{i-1}...y_1}$. □

Lemma. $\overline{e}^{-1}x\overline{e} \in \langle A \rangle$.

Proof. Suppose $r \neq 0$ (else immediate). Then

$$\begin{aligned} x = y_r...y_1 &= (t_{r+1}t_{r+1}^{-1})y_r(t_r t_r^{-1})y_{r-1}(t_{r-1}...t_3^{-1})y_2(t_2 t_2^{-1})y_1(t_1 t_1^{-1}) \\ &= \overline{e}\underbrace{(\overline{y_r t_r}^{-1}y_r t_r)(\overline{y_{r-1}t_{r-1}}^{-1}y_{r-1}t_{r-1})...(\overline{y_2 t_2}^{-1}y_2 t_2)(\overline{y_1 t_1}^{-1}y_1 t_1)}_{\in\langle A\rangle}\overline{e}^{-1} \end{aligned}$$

□

Proof of Schreier's Theorem. $A \subseteq H$ is immediate since $\forall y \in Y\ \forall t \in T \quad (yt)H = \overline{yt}H$. The latest lemma applied to arbitrary $x \in H$ (for which $\exists (y_i)_{i=1}^k \in Y^* : y_k...y_1 = x$) then shows that $\overline{e}^{-1}H\overline{e} \subseteq \langle A \rangle \subseteq H$. $|\overline{e}^{-1}H\overline{e}| = |H|$, so $\langle A \rangle = H$. □

[1] Formally, this means that $\forall A \in [G : G_\alpha]\ \exists! \beta \in G(\alpha) : \forall g \in A \quad g(\alpha) = \beta$.

Q7. The function `stab`(G, j) lists the stabiliser of $j \in [1, n]$ in $G \leq S_n$. It does not use the stripping algorithm to simplify its result;[2] such behaviour may be achieved by composing it with `strip`. On the orbit example, this gives:

```
>>> stab([[0,2,3,1,4],[0,2,1,3,4]],2)
[[0, 1, 2, 3, 4], [0, 3, 2, 1, 4], [0, 3, 2, 1, 4], [0, 1, 2,
   3, 4], [0, 3, 2, 1, 4], [0, 1, 2, 3, 4]]
>>> strip(stab([[0,2,3,1,4],[0,2,1,3,4]],2))[0]
[[0, 3, 2, 1, 4]]
```

As predicted by the orbit-stabiliser theorem, as the orbit has size 3 and the stabiliser is generated by one 2-cycle, the full group $\langle (1\ 2), (1\ 2\ 3) \rangle$ has order 6.

5 Orders

Q8. The function `order`(G) returns the order of $G \leq S_n$, and derives a sequence of intermediate subgroups in doing so, whose number of generators it prints in order, both before and after stripping, marked `B:` and `A:` respectively. For each subgroup, it also prints the size of the orbit found inside it (marked `O:`); thus, the order of each subgroup can be computed by multiplying together orbit sizes from the lowest one up to and including the subgroup in question. In particular, the product of all these numbers is the order of the group itself.

This is the function's output for the example groups.

```
B: 3      A: 3      O: 7
B: 21     A: 8      O: 6
B: 48     A: 3      O: 4
B: 12     A: 0
Order: 168
```

Group 1

```
B: 3      A: 3      O: 11
B: 33     A: 20     O: 10
B: 200    A: 15     O: 9
B: 135    A: 7      O: 8
B: 56     A: 0
Order: 7920
```

Group 2

```
B: 3      A: 3      O: 3
B: 9      A: 4      O: 7
B: 28     A: 15     O: 6
B: 90     A: 11     O: 5
B: 55     A: 7      O: 4
B: 28     A: 4      O: 3
B: 12     A: 2      O: 2
B: 4      A: 1      O: 2
B: 2      A: 0
Order: 30240
```

Group 3

```
B: 2      A: 2      O: 15
B: 30     A: 14     O: 12
B: 168    A: 20     O: 9
B: 180    A: 12     O: 6
B: 72     A: 7      O: 2
B: 14     A: 6      O: 2
B: 12     A: 5      O: 2
B: 10     A: 4      O: 2
B: 8      A: 3      O: 3
B: 9      A: 1      O: 2
B: 2      A: 0
Order: 933120
```

Group 4

```
B: 2      A: 2      O: 16
B: 32     A: 14     O: 14
B: 196    A: 36     O: 12
B: 432    A: 25     O: 10
B: 250    A: 16     O: 8
B: 128    A: 9      O: 6
B: 54     A: 4      O: 4
B: 16     A: 1      O: 2
B: 2      A: 0
Order: 10321920
```

Group 5

[2] This is to prevent redundant stripping in the `order` algorithm.

The unlisted function `order_`(G, k) runs a variant of `order`(G) that either uses the stripping algorithm as usual ($k = 1$), computes generators of each stabiliser with the modification of checking if each has already been computed ($k = 2$), or has no redundancy mechanism in place at all. The $k = 2$ option is intended as a simple optimisation.

The three variants have a standard pattern of behaviour across the example groups. Group 4 is a typical case, demonstrated below.

```
>>> order_(O[4],0)
B: 2
B: 30
B: 360
B: 3240
B: 19440
B: 38880
B: 77760
B: 155520
B: 311040
B: 933120
Memory limit
   exceeded.
```

```
>>> order_(O[4],1)
B: 2       A: 2
B: 30      A: 14
B: 168     A: 20
B: 180     A: 12
B: 72      A: 7
B: 14      A: 6
B: 12      A: 5
B: 10      A: 4
B: 8       A: 3
B: 9       A: 1
B: 2       A: 0
933120
```

```
>>> order_(O[4],2)
B: 2
B: 16
B: 119
B: 388
B: 96
B: 48
B: 24
B: 12
B: 6
B: 2
B: 1
933120
```

Evidently, without any optimisation, the algorithm runs extremely slowly and accumulates a huge amount of extraneous data. For the example groups, both the stripping algorithm and simple redundancy check perform similarly well, though the stripping algorithm is slightly faster, empirically. Indeed, the generating set after stripping is usually smaller than the generating set after the simple check.

Nonetheless, the stripping algorithm doesn't typically find a generating set of smallest order. For example, in among the example groups, testing each subset of their generating groups as to whether it generates a subgroup of full order[3] reveals that $(1, 2, 3, 4)(5, 6, 7, 8)$ is redundant in the second group, yet the stripping algorithm doesn't remove it.

[3]Some generators are the unique generators permuting certain elements and as such must appear in every generating subset, lest the generated subgroup be unable to permute said element.

6 Complexities

I've left these until last because I have no confidence I can do them, and didn't want them to interfere with the rest of my nice project. Excuse the messiness and informality; I'm in a rush.

`mp,inv.` These routines generate lists of length n, and for each component, execute finitely-many integer reads and writes ($O(1)$). Hence, they have complexity $nO(1) = O(n)$.

`strip.` This first copies a list ($O(n)$), reads n ($O(1)$), generates an $n \times n$ matrix of zeros ($O(n^2)$) and writes l ($O(1)$). This sums to $O(n^2)$. It then processes each generator once (k loops), each time processing at most n points in $[1, n]$. For each of these points, it at worst reads some permutations ($O(1)$) and multiplies/inverts them ($O(n)$). The second loop thus contributes at most $nO(n) = O(n^2)$. Then, while still in the first loop, a list deletion may be performed ($O(n)$); the first loop thus contributes at most $k(O(n^2)+O(n)) = O(k)O(n^2)$. Thus, the total complexity is $O(n^2) + O(k)O(n^2) = O(k)O(n^2)$.

`orb.` The initial overhead is $O(n)$. Then, counting through loops as before, we have an upper bound n (as $|O| \leq n$) $\times(k \times (1 + n + 1 + n)) + 1$ (since `in` is $O(n)$) – i.e. a total of $O(n) + O(n)(O(k)O(n) + O(1)) = O(k)O(n^2)$.

`stab.` The initial overhead is dominated by one instance of `orb`, namely $O(k)O(n^2)$. Then, counting through loops (noting that the orbit has size at most n), we have an upper bound of $k \times n(O(n) + nO(1) + (2O(n) + 1) = O(k)O(n^2)$.

`order.` This algorithm is recursive. At each recursive step, its dominant terms comprise a `strip`, and $n\times$ `orb`. The number of recursive steps is $O(\log(|G|)) \leq O(\log(n!)) = O(n \log(n))$. Thus, we have a bound of $O(n \log(n)) \times (O(k)O(n^2)+nO(k)O(n^2)) = O(n \log(n))O(k)O(n^3) \leq O(k)O(n^5)$. The k here actually varies (it's not just the input k) but as a result of successive stripping before input, we may bound it below $\frac{1}{2}n(n-1)) = O(n^2)$. Thus, the final complexity is $O(n^7)$.

A Programs

All functions are contained in the module `A.py`. `InitA.py` is a script that, when run in a Python shell, loads all functions into the global memory.

```
def mp(a,b):
    return [a[b[i]] for i in range(len(a))]

def inv(a):
    y = [0 for i in range(len(a))]
    for i in range(1,len(a)):
        y[a[i]] = i
    return y

def strip(X):
    X = list(X)
    if len(X)==0:
        return [[],[]]
    n = len(X[0]) - 1
    A = [[0 for i in range(n+1)] for i in range(n+1)]
    l = 0
    while l in range(len(X)):
        for m in range(1,n+1):
            if X[l][m] == m:
                pass
            elif A[m][X[l][m]] != 0:
                X[l] = mp(inv(A[m][X[l][m]]),X[l])
                # X[l] now fixes [1,m]
            else:
                A[m][X[l][m]] = X[l]
                l = l + 1
                break           # X[l] is placed
        else:
            del X[l]            # X[l] is deleted
    return [X,A]

def orb(G,j):
    if len(G) == 0:
        return [[j],['id']]
    Y = [j]
    Z = [list(range(len(G[0])))]
    i = 0
    while i in range(len(Y)):
        for g in G:
            y = g[Y[i]]
            if not(y in Y):
                Y.append(y)
                Z.append(mp(g,Z[i]))
        i += 1
    return [Y,Z]
```

```
def stab(G,j):
    if len(G) == 0:
        return G
    Y = []
    T = orb(G,j)[1]
    for y in G:
        for t in T:
            u = mp(y,t)
            for s in T:
                if s[j] == u[j]:
                    v = s
                    break
            Y.append(mp(inv(v),u))
    return Y

def order(G):
    p = len(G)
    G = strip(G)[0]
    print('B:','{:<5}'.format(str(p)),'A:','{:<5}'.format(len(G
       )),end="␣")
    if len(G) == 0:
        print('\nOrder:',end="␣")
        return 1
    n = len(G[0]) - 1
    for j in range(1, n+1):
        m = len(orb(G,j)[0])
        if m != 1:
            print('O:',m)
            return m * order(stab(G,j))
```