CATAM 16.5 Permutation Groups

1 Groups and the Stripping Algorithm of Sims

1.1 Permutations

To define how an element g, of the symmetric group S_n, behaves, it suffices to consider its behaviour on the set $X = \{1, 2, \dots n\}$. As the language I will be using for this project is Python, the code will take $X = \{0, 1, \dots n-1\}$. However, there is no substantial difference between these paradigms

Now we can define how a permutation is stored in memory, in terms of a list. We consider the bijection $g : X \to X$. If we define an array of size n and fill the entries such that $\vec{g}[i] = g(i)$, in doing so we encapsulate all the needed information.

1.1.1 Question 1

Question 1 asks for a program to compute the inverse and product of two permutations, which under the current paradigm is very easy. ***Q1.py*** was written to achieve this and can be found in section 3. Firstly, let us consider inverses. Let $h = g^{-1}$

$$h(g(i)) = i \implies \vec{h}[\vec{g}[i]] = i$$

As g is a bijection, we have fully specified the list h. For products, it is just as simple. Let $h = f \circ g$

$$h(i) = f(g(i)) \implies \vec{h}[i] = \vec{f}[\vec{g}[i]]$$

The only issue that could arise is from the ordering of f and g, due to S_n being non-abelian. However if we keep with the standard convention of left and right multiplication, then there will be no issue.

As an example, we took $g = (1, 3, 4, 0, 2)$, $h = (2, 4, 1, 3, 0)$ and computed $g^{-1} = (3, 0, 4, 1, 2)$ and $g \circ h = (4, 2, 3, 0, 1)$ which agrees with the theoretical result.

Considering the complexity of our algorithms, it is clear to see that they are both $O(n)$, noting that we simply iterate through each i and assign each index a value at each step.

1.2 Stripping Algorithm

1.2.1 Question 2

Let the entries of the stripping algorithm, $\pi_1 \dots \pi_k$, generate G. Let us assume that, by the time we reach π_i, the current output $\rho_1 \dots \rho_l$ can generate $\pi_1 \dots \pi_{i-1}$. Now, supposing we combine some modifications as per the algorithm to fix any amount of entries, we may write $g = \rho^{-1}\pi_i$ with $\rho \in \langle \rho_1 \dots \rho_l \rangle$, where g may mutate throughout running of the program. Now, if g terminates as the identity, then $\rho = \pi_i$ it is clear that π_i is generated by the current output. If it does not terminate on the identity, the algorithm stipulates that g is added to the output, and so π_i is still generated by the output at that point. By induction, the output generates all π_i and hence G.

To find an upper bound of the number of permutation this algorithm output, we consider A as a box of data. If a permutation is on the i^{th} row and j^{th} column, it fixes all numbers $< i$ and sends i to j. We remark that no permutation can live on the diagonal or the lower half triangle ($j \leq i$), as it means j is both fixed and not fixed; a contradiction. So, we have an upper bound of $\frac{n(n-1)}{2}$. We also note that this bound may be realised by considering every possible transposition in S_n.

Let us consider the complexity of this algorithm. First note that elementary operations of a permutation of S_n, including reading, storing, comparison to the identity, taking inverses, and products are all $O(n)$.

In the worst case scenario we will have to change (or store) g on every row, i.e. fix 1, then fix 2 and so on. This will require on each row, at worst, to read the data in A, apply an inverse and a product, and finally compare to the identity. This gives a procedure on each row that is $O(n)$ because it is four concatenated $O(n)$ tasks. As this happens in every row, running this procedure on one element of the generating set is at worst an $O(n^2)$ task. Applying the worst case scenario to every generator, $\pi_1 \dots \pi_k$, we then see that our algorithm has complexity $O(k \cdot n^2)$.

1.2.2 Question 3

Question 3 asks for a program to find the new generators according to this procedure. ***Q3.py*** was written to achieve this. It can be found in Section 3. Listed below are a few examples of computation:

n	Initial Generators	New Generators
5	$(0,1,2,3,4) = e$	N/A
3	$(2,0,1)$ $(2,1,0)$	$(2,0,1)$ $(0,2,1)$
4	$(1,2,3,0)$	$(1,2,3,0)$
5	$(1,0,2,3,4)$ $(1,2,3,4,0)$ $(1,3,0,2,4)$ $(2,3,1,0,4)$	$(1,0,2,3,4)$ $(2,3,1,0,4)$ $(0,2,3,4,1)$ $(0,3,1,2,4)$

Table 1: Input and output generators.

We note that the first three examples are the trivial group, S_3 and $C_4 \leq S_4$ which are still generated by the new generators. The fourth example is more complicated but it is possible to show these two sets of generators generate each other, so must yield the same resultant group.

2 Orbit Stabilizer and Schreier's Theorem

2.1 Orbit Stabilizer Theorem

2.1.1 Question 4

Consider the map:

$$\Phi : G/G_\alpha \to Orb_G(\alpha) \qquad \Phi(gG_\alpha) = g\alpha$$

To prove this is a bijection, we must prove it is well defined, injective and surjective. To prove it is well defined and injective:

$$gG_\alpha = hG_\alpha \iff g^{-1}h \in G_\alpha \iff g^{-1}h\alpha = \alpha \iff h\alpha = g\alpha$$

Surjectivity follows from if $\beta \in Orb_G(\alpha) \implies \exists g \; g\alpha = \beta$ so the coset gG_α is the natural choice.

Orbit-Stabilizer Theorem: For G a finite group, $|G| = |G_\alpha||Orb_G(\alpha)|$.
Proof: Use the bijection we constructed and apply Lagrange's theorem.

$$|Orb_G(\alpha)| = |G/G_\alpha| = |G : G_\alpha| = |G|/|G_\alpha|$$

2.1.2 Question 5

Question 5 asks for a program to find the orbit and witness of some value α for a group generated by given permutations. ***Q5.py*** was written to achieve this and can be found in Section 3. Here are a few examples:

n	α	Generators	Orbit Witness Pairs
3	0	$(1,0,2)$	$0,(0,1,2)$ $1,(1,0,2)$
3	2	$(1,0,2)$	$2,(0,1,2)$
4	0	$(1,0,3,2)$ $(0,1,3,2)$	$0,(0,1,2,3)$ $1,(1,0,3,2)$
5	0	$(0,3,2,4,1)$ $(4,2,0,1,3)$	$0,(0,1,2,3,4)$ $1,(1,2,0,3,4)$ $2,(2,0,4,1,3)$ $3,(3,0,4,2,1)$ $4,(4,2,0,1,3)$

Table 2: Witness for members of the orbit of α.

The algorithm is fairly simple. First, we note that we only need to consider the action of generators:

- Initialise an array of size n which will be the output.
- Iterate through each generator and if for example $\pi_i\alpha = \beta$ then store π_i in the position β of the array.
- Repeat process replacing α with β. Storing $\pi_j\pi_i$ if π_j sends β to γ with γ not already covered.
- Continue the process until we can no longer expand the size of the orbit.
- Return the array.

Looking at the complexity of the algorithm, in the worst case scenario we must compute $\pi_i\alpha$ for every generator π_i, and every $\alpha \in \{0 \dots n-1\}$ which is an $O(k \cdot n)$ operation, and at worst this must be done n times, by finding only one new witness each time. This gives an $O(k \cdot n^2)$ algorithm.

2.2 Schreier's Theorem

2.2.1 Question 6

Let G_α be the stabilizer of α in G. Let Y generate G. Let T be a set of coset representatives with $\varphi : G \to T$ defined by $g\alpha = \varphi(g)\alpha$. Let $x = y_r \dots y_1 \in G_\alpha$ with $y_i \in Y$. Define t_1 as being in G_α and $t_{i+1} = \varphi(y_i t_i)$ for $i = 1 \dots r$.

Lemma: $t_{r+1} = t_1$.
Proof: First note that for $g, h \in G$ that $\varphi(gh)\alpha = gh\alpha = g\varphi(h)\alpha$. Let $t_{r+1}\alpha = y_r \dots y_i t_i \alpha$, then $t_{r+1}\alpha = y_r \dots y_i \varphi(y_{i-1}t_{i-1})\alpha = y_r \dots y_i y_{i-1} t_{i-1}\alpha = y_r \dots y_{i-1} t_{i-1}\alpha$. By induction it follows that $t_{r+1}\alpha = y_r \dots y_1 t_1 = x t_1 \alpha$. Finally $t_{r+1}\alpha = x\alpha = \alpha$. Then as $t_{r+1} \in G_\alpha$ we must have $t_{r+1} = t_1$.

Proposition: G_α is generated by $S = \{\ \varphi(yt)^{-1}yt \mid y \in Y, t \in T\}$
Proof: Let $H = \langle S \rangle$ we see $\varphi(yt)^{-1}yt\alpha = \varphi(yt)^{-1}\varphi(yt)\alpha = \alpha$ so $H \le G_\alpha$. Suppose $xt_1 = y_r \dots y_{i+1} y_i t_i h$ for $h \in H$. (This is trivially true for $i = 1$ with $h = e$) Then $xt_1 = y_r \dots y_{i+1}\varphi(y_i t_i)(\varphi(y_i t_i)^{-1} y_i t_i)h$ $= y_r \dots y_{i+1} t_{i+1} h'$ with $h' \in H$. It follows by induction that $\exists\ h \in H$ such that $xt_1 = t_{r+1}h$ or indeed $t_1^{-1} x t_1 = h$. To finish we note that conjugation by t_1 yields a bijection G_α to itself, thus $H = G_\alpha$.

2.2.2 Question 7

Question 7 asks for a program to find generators of a stabilizer. ***Q7.py*** was written to achieve this and can be found in Section 3. Here are a few example computations.

n	α	Generators	Stabilizer Generators
3	0	$(1,0,2)$	N/A
3	2	$(1,0,2)$	$(1,0,2)$
		$(1,0,3,2)$	
4	0	$(0,1,3,2)$	$(0,1,3,2)$
			$(0,2,3,1,4)$
			$(0,3,2,4,1)$
			$(0,4,3,2,1)$
		$(0,3,2,4,1)$	$(0,1,3,4,2)$
5	0	$(4,2,0,1,3)$	$(0,1,4,2,3)$
			$(0,2,3,4,1)$
			$(0,3,1,2,4)$
		$(1,0,2,3,4)$	$(0,4,3,2,1)$
		$(1,2,3,4,0)$	$(0,1,3,2,4)$
		$(1,3,0,2,4)$	$(0,1,4,3,2)$
5	0	$(2,3,1,0,4)$	$(0,1,2,4,3)$

Table 3: Stabilizer group generators.

To compute the time complexity of this algorithm we note that the first step of computing the set T is an $O(k \cdot n^2)$ procedure. Then we must create the generators for the stabilizer; from Schreier's theorem we must compute $|Y| \cdot |T|$ permutations, with two products, an inverse, and an application of φ which is reading an array. These are all $O(n)$ operations so the computation of any new generator is an $O(n)$ task. As we take $k = |Y|$ and we know that $|T| \le n$ we can see that computing all the generators required by the theorem is an $O(k \cdot n^2)$ algorithm. Applying the reduction algorithm on the set of generators (size at most $k \cdot n$) gives a time complexity of $O((k \cdot n) \cdot n^2) = O(k \cdot n^3)$ by the discussion in Section 1.2.1. Finally, concatenating all three parts of the algorithm together give a resultant complexity of $O(k \cdot n^3)$.

2.2.3 Question 8

Question 8 asks for a program to find the order of a group G. We use the code created in question 7 to find the generators of stabilizers. Then we fix each element of X in turn, finding each successive stabilizer and orbit as we go. Then we finally use the Orbit Stabilizer theorem to find the final result.

We let $G = G_0$ and consider G_i to be the stabilizer of G_{i-1} fixing i. Then it follows that

$$G \geq G_1 \geq \cdots \geq G_n \cong \{e\}$$

The orbit stabilizer theorem then gives by induction:

$$|G_i| = \prod_{j=i+1}^{n} |Orb_{G_{j-1}}(j)|$$
$$|G| = \prod_{j=1}^{n} |Orb_{G_{j-1}}(j)|$$

Q7.py was written to achieve this and can be found in section 3. A few example computations are shown for groups which we already know:

n	Known Group	Generators	Order
6	$\{e\}$	$(0,1,2,3,4,5)$	1
3	C_3	$(1,2,0)$ $(2,0,1)$	3
3	S_3	$(1,2,0)$ $(0,2,1)$	6
4	$V = K_4$	$(1,0,3,2)$ $(2,3,0,1)$	4
5	S_5	$(1,2,3,4,0)$ $(1,0,2,3,4)$	120
5	A_5	$(1,2,3,4,0)$ $(1,0,3,2,4)$	60
11	S_{11}	$(1,2,3,4,5,6,7,8,9,10,0)$ $(1,0,2,3,4,5,6,7,8,9,10)$	$39916800 = 11!$

Table 4: Computed orders of groups.

It was also tested for S_{20} with analogous generators to S_{11} with a quick computation time.

We also used ***Q8.py*** to show the orders of each subgroup and the count of generators before applying the stripping algorithm. The program was run on 5 groups with generators listed below.

Group	Generating Set
$V = K_4$	$(1,0,3,2)$, $(2,3,0,1)$
S_3	$(1,2,0)$, $(0,2,1)$
A_4	$(1,2,3,4,0)$, $(1,0,3,2,4)$
S_6	$(1,2,3,4,5,0)$, $(1,0,2,3,4,5)$
$C_2 \times V \leq S_6$	$(1,0,2,3,4,5)$, $(0,1,3,2,5,4)$, $(0,1,4,5,2,3)$

Table 5: Initial generating sets.

Initial Group	Iteration	Orbit Size	Subgroup Size	Generators	
				Before	After
$V = K_4$	0	N/A	120	2	2
	1	4	1	8	0
S_3	0	N/A	6	2	2
	1	3	2	6	1
	2	2	1	2	0
A_4	0	N/A	60	2	2
	1	5	12	10	4
	2	4	3	16	2
	3	3	1	6	0
S_6	0	N/A	720	2	2
	1	6	120	12	6
	2	5	24	30	5
	3	4	6	20	3
	4	3	2	9	1
	5	2	1	2	0
$C_2 \times V \leq S_6$	0	N/A	8	3	3
	1	2	4	6	2
	2	1	4	2	2
	3	4	1	8	0

Table 6: The size of subgroups and generator sets.

In the event that we fail to use the stripping algorithm after each iteration, we note that the number of generators is equal to the number of generators in the previous step multiplied by the orbit of the subgroup. This means that if we started with two elements generating S_{20}, then at the end we are having to run the computations and logic for $2 \cdot 20!$ generators. A completely unfeasible task. Whereas with stripping, it is fairly quick to run.

2.2.4 Question 9

Let P_n be the probability that two elements of S_n generates S_n. We know, from IA, that the following elements expressed in cycle notation $(12), (123 \dots n)$ generate S_n so $P_n > 0$.

Consider A_n. The probability that both elements chosen lie in A_n is $\frac{1}{4}$ and hence we must have $P_n \leq \frac{3}{4}$ due to A_n being a subgroup.

We may tabulate results for small n by simply exhausting every option. A program has been written to run this computation and can be found in section 3.

n	2	3	4
P_n	0.75	0.5	0.375

Table 7: P_n for n small.

To generate a random permutation efficiently is very simple. First let us begin with an array of numbers $\vec{A} = [1, \dots n]$. We generate a random number, k, between 1 and the size of the array and define $\vec{g}[1] = k$.

Now we remove k from $\vec{A}$ and iterate the process with $\vec{g}[2] \dots \vec{g}[n]$ until A has no more elements. We will have exhausted all the numbers 1 to n without repeats and no significant inefficiencies.

We may now use this to calculate approximations for P_n by generating a few pairs at each n and running the algorithm to compute the order of the group generated. Then we can compare the order against $n!$ to see if is indeed S_n. Some results are tabulated and graphed below.

n	5	6	7	8	9	10	11	12
P_n	0.5	0.42	0.6	0.65	0.61	0.64	0.71	0.65

Table 8: Approximations of P_n

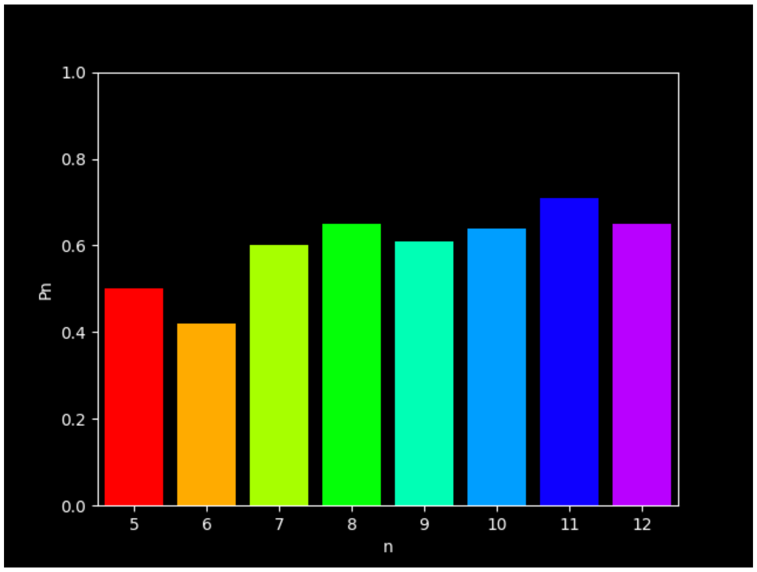

Figure 1: P_n approximations against n

We can see, perhaps contrary to intuition, that P_n does not go to 0 but instead appears to increase. We note from an earlier discussion that 0.75 is the probability that the group generated will not be a subgroup of A_n. This then lends itself to the idea that, in the limit as $n \to \infty$, the probability that we generate S_n or A_n approaches 0.75 or 0.25 respectively.

3 Code

3.1 Q1.py

```python
def Inverse(f):
    g = [0 for i in range(len(f))]
    for i in range(len(f)):
        g[f[i]] = i
    return g

def Product(f, g):
    h = [0 for i in range(len(f))]
    for i in range(len(f)):
        h[i] = f[g[i]]
    return h


if __name__ == "__main__":
    g = [1, 3, 4, 0, 2]
    h = [2, 4, 1, 3, 0]

    print(g, h)
    print(Inverse(g))
    print(Product(g, h))
```

3.2 Q3.py

```
from Q1 import Inverse, Product

def ReduceGenerators (n, gens):
    A = [[None for i in range(n)] for j in range(n)]
    for i in range(len(gens)):
        g = gens[i]
        for j in range(n):
            if (g[j] == j): continue
            f = A[j][g[j]]
            if (f != None):
                g = Product(Inverse(f), g)
                continue
            A[j][g[j]] = g
            break
    #print(A)
    return SimplifyArray(A)

def SimplifyArray(A):
    arr = []
    for i in range(len(A)):
        for j in range(len(A[i])):
            if (A[i][j] != None):
                arr.append(A[i][j])
    return arr

if __name__ == "__main__":
    print(ReduceGenerators(5, [[0, 1, 2, 3, 4]]))
    print(ReduceGenerators(3, [[2,0,1], [2,1,0]]))
    print(ReduceGenerators(4, [[1,2,3,0]]))
    print(ReduceGenerators(5, [[1,0,2,3,4], [1,2,3,4,0], [1,3,0,2,4], [2,3,1,0,4]]))
```

3.3 Q5.py

```
from Q1 import Product
from Q3 import ReduceGenerators, SimplifyArray

def GetOrbitWitness (n, el, perms):
    perms = ReduceGenerators(n, perms)


    oldOrbit = [None for i in range(n)]
    orbit = [None for i in range(n)]
    orbit[el] = [i for i in range(n)]
    while (orbit != oldOrbit):
        # I KNEW THIS WOULD CAUSE some memory issue
        # oldOrbit = orbit
        for i in range(n): oldOrbit[i] = orbit[i]

        for k in range(len(perms)):
            g = perms[k]
            for j in range(n):
                if (not oldOrbit[j]): continue
                if (orbit[g[j]] == None):
                    orbit[g[j]] = Product(g, oldOrbit[j])


    return orbit


def PrintQ5Data(n, a, gens):
    print(n, a, gens, end=" ")
    data = (GetOrbitWitness(n, a, gens))
    for i in range(len(data)):
        if (data[i] != None):
            print(str(i) + ", " + str(data[i]), end=" ")

    print("")


if __name__ == "__main__":

    PrintQ5Data(3, 0, [[1,0,2]])
    PrintQ5Data(3, 2, [[1,0,2]])
    PrintQ5Data(4, 0, [[1, 0, 3, 2], [0, 1, 3, 2]])
    PrintQ5Data(5, 0, [[0, 3, 2, 4, 1], [4, 2, 0, 1, 3]])
```

3.4 Q7.py

```
from Q5 import GetOrbitWitness
from Q3 import SimplifyArray, ReduceGenerators
from Q1 import Product, Inverse

def GetStabiliserGenerators(n, el, perms, T):

    def Phi(g):
        return T[g[el]]

    A = [[None for i in range(len(perms))] for j in range(n)]

    for i in range(n):
        t = T[i]
        if (not t): continue
        for j in range(len(perms)):
            y = perms[j]
            yt = Product(y, t)
            A[i][j] = Product(Inverse(Phi(yt)), yt)

    #print(len(SimplifyArray(A)), end=" ")
    return ReduceGenerators(n, SimplifyArray(A))

def PrintQ7Data(n, a, gens):
    T = GetOrbitWitness(n, a, gens)
    data = GetStabiliserGenerators(n, a, gens, T)
    print(n, a, gens, data)

if __name__ == "__main__":
    PrintQ7Data(3, 0, [[1,0,2]])
    PrintQ7Data(3, 2, [[1,0,2]])
    PrintQ7Data(4, 0, [[1, 0, 3, 2], [0, 1, 3, 2]])
    PrintQ7Data(5, 0, [[0, 3, 2, 4, 1], [4, 2, 0, 1, 3]])
    PrintQ7Data(5, 0, [[1,0,2,3,4], [1,2,3,4,0], [1,3,0,2,4], [2,3,1,0,4]])
```

3.5 Q8.py

```
from Q7 import GetStabiliserGenerators
from Q5 import GetOrbitWitness
from Q3 import SimplifyArray, ReduceGenerators

def ComputeGroupOrder(n, perms):
    order = 1
    perms = ReduceGenerators(n, perms)

    for i in range(n):
        if (len(perms) == 0): break
        T = GetOrbitWitness(n, i, perms)
        orbitSize = len(SimplifyArray([T]))
        order *= orbitSize
        #print (order, end= " ")
        perms = GetStabiliserGenerators(n, i, perms, T)
    #print()

    # remove hashtags before this to get subgroup data

    return order

def PrintQ8Data(n, gens):
    order = ComputeGroupOrder(n, gens)
    print(n, gens, order)

if __name__ == "__main__":
    # Main Data

    PrintQ8Data(6, [[0,1,2,3,4,5]])
    PrintQ8Data(3, [[1,2,0], [2,0,1]])
    PrintQ8Data(3, [[1,2,0], [0,2,1]])
    PrintQ8Data(4, [[1,0,3,2], [2,3,0,1]])
    PrintQ8Data(5, [[1,2,3,4,0], [1,0,2,3,4]])
    PrintQ8Data(5, [[1,2,3,4,0], [1,0,3,2,4]])
    PrintQ8Data(11, [[1,2,3,4,5,6,7,8,9,10,0], [1,0,2,3,4,5,6,7,8,9,10]])

    # Do Subgroups
    print()

    # If you want to run this part and get the data you need to take the print statements
    # out of comments in ComputeGroupOrder Function

    #ComputeGroupOrder(4, [[1,0,3,2], [2,3,0,1]])
    #ComputeGroupOrder(3, [[1,2,0], [0,2,1]])
    #ComputeGroupOrder(5, [[1,2,3,4,0], [1,0,3,2,4]])
    #ComputeGroupOrder(6, [[1,2,3,4,5,0], [1,0,2,3,4,5]])
    #ComputeGroupOrder(6, [[1,0,2,3,4,5], [0,1,3,2,5,4], [0,1,4,5,2,3]])

    # I am speeeeeeeed for S_20
    print

    PrintQ8Data(20, [[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,0], [1,0,2,3,4,5,6,7,8,9,10,11,12
```

3.6 Q9.py

```
import random
import matplotlib as mpl
import matplotlib.pyplot as plt
import numpy as np
from Q8 import ComputeGroupOrder

def SetupAxes (x, y, showGrid=True):

    x = np.array(x)
    y = np.array(y)
    xMax = np.max(x) + 0.5
    xMin = 4.5
    yMax = 1
    yMin = 0

    fig, ax = plt.subplots()
    plt.xlabel('n', color='white')
    plt.ylabel('Pn', color='white')
    fig.set_facecolor("black")
    ax.set_facecolor("black")
    ax.spines['bottom'].set_color('white')
    ax.spines['top'].set_color('white')
    ax.spines['left'].set_color('white')
    ax.spines['right'].set_color('white')
    ax.tick_params(colors='white')
    ax.set_xlim(xMin, xMax)
    ax.set_ylim(yMin, yMax)
    if (showGrid):
        ax.grid(color='white', linewidth=0.4, alpha=0.3, zorder=0)
    return fig, ax


def GenerateRandomPermutation(n):

    values = [i for i in range(n)]
    perm = [-1 for i in range(n)]

    for i in range(n):
        t = random.randint(0,n-i-1)
        perm[i] = values[t]
        values = values[:t] + values[t+1:]

    return perm

def PairCheck(n):
    g = GenerateRandomPermutation(n)
    h = GenerateRandomPermutation(n)
    ord = ComputeGroupOrder(n, [g,h])
    return (ord == np.math.factorial(n))

def GetSn2Probability(n, perms):
    l = len(perms)
    tot = 0
    for i in range(l):
        for j in range(l):
            ord = ComputeGroupOrder(n, [perms[i],perms[j]])
            tot += (ord == np.math.factorial(n))
    prob = tot / (l * l)
    return prob

def SmallProbabilities():
    perms2 = [[0,1], [1,0]]
    perms3 = [[0,1,2], [0,2,1], [1,0,2], [1,2,0], [2,0,1], [2,1,0]]
    perms4 = [[0,1,2,3], [0,1,3,2], [0,2,1,3], [0,2,3,1], [0,3,1,2], [0,3,2,1],
              [1,0,2,3], [1,0,3,2], [1,2,0,3], [1,2,3,0], [1,3,0,2], [1,3,2,0],
              [2,0,1,3], [2,0,3,1], [2,1,0,3], [2,1,3,0], [2,3,0,1], [2,3,1,0],
              [3,0,1,2], [3,0,2,1], [3,1,0,2], [3,1,2,0], [3,2,0,1], [3,2,1,0]]

    print(GetSn2Probability(2, perms2))
    print(GetSn2Probability(3, perms3))
    print(GetSn2Probability(4, perms4))

def EstimatePn(n):
    k = 100
    total = 0
    for i in range(k):
        total += PairCheck(n)
    prob = total/k
    return(prob)

def GraphPn():
    m = 13
    ns = [i for i in range(5, m)]
    P = [0 for i in range(m)]

    for n in range(5, m):
        print(n)
        P[n] = EstimatePn(n)

    colours = [mpl.cm.hsv((i -5) * 0.8 / (m-6)) for i in range(5, m)]

    P = P[5:]

    fig, ax = SetupAxes(ns, P, showGrid=False)
    plt.bar(ns, P, color=colours)
    plt.show()


if __name__ == "__main__":
    SmallProbabilities()

    GraphPn()
```