University of Cambridge

Part II CATAM Project

Permutation Groups

Jonah M. Gibbon

April, 2022

Submitted in the fulfilment of the Mathematical Tripos, Part II.

Throughout this project, permutations $\pi : X \to X$ will be represented in cyclic form, but also as $2 \times n$ matrices (where the first row is the set X and the second row is the corresponding image). For example the permutation $(1,2)(3,4)$ when $n = 5$ could also be represented as

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 1 & 4 & 3 & 5 \end{pmatrix}$$

All code in this project will use the matrix representation, since permutations are represented uniquely this way.

Question 1

The program named Code 1, referenced on page 9, was used to multiply permutations together and calculate their inverses.[1] The following permutations were used as test data.

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 4 & 2 & 1 \end{pmatrix} \qquad B = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 1 & 3 & 4 \end{pmatrix}$$

The code outputted the correct answers:

$$AB = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 \end{pmatrix} \qquad A^{-1} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 1 & 2 \end{pmatrix}$$

The program finds a permutation π's inverse by calculating πx for all $x \in X$, and saving $\pi^{-1}(\pi x) = x$. This exploits the fact that the permutation is a bijection; otherwise this method would fail. Calculating the image of an element requires $O(1)$ operations, and saving the appropriate value also requires $O(1)$ operations. We conclude that this method is $O(n)$.

When multiplying two permutations π_1 and π_2 together the method is similar. The program calculates $\pi_1 x$, then $\pi_2(\pi_1 x)$, and saves this as the image of x. This requires a total of $O(1)$ operations per element, and hence is $O(n)$ as well.

Question 2

We aim to prove by induction on l that the set of permutations currently in the array, combined with the permutations $\pi_{l+1}, \ldots, \pi_k$ generate G, where l is the number of generators added/omitted to array. Clearly this is true when $l = 0$.

Suppose the case $l = m$ holds, where $m \in \mathbb{N}$. Also suppose that applying the algorithm to the permutation π_{m+1} causes it to be modified $0 \leq r \leq n$ times before either being discarded as the identity, or being added to the array. At this point, the permutation will be of the form $\pi' = g_r^{-1} g_{r-1}^{-1} \ldots g_1^{-1} g_0^{-1} \pi_{m+1}$, where each g_i is some permutation currently in the array (with the exception g_0 being the identity). Should π' be the identity, then π_{m+1} can be generated from the current permutations in the array and so can be discarded from the generating set. If π' is added to the array, then π_{m+1} can still be generated from the current modified set, namely $\pi_{m+1} = g_1 \ldots g_r \pi'$. Hence the inductive step holds. By setting $l = k$ the result follows immediately.

Notice that only the cells above the diagonal of the array ever gets filled. No permutation can get added on the diagonal, since then we would require the permutation to not fix some element in X, but also simultaneously fix that same element. If a permutation π was added below the diagonal, say at position (i, j) where $i > j$, then this permutation would not fix i, and in turn would not fix j (since $i \mapsto j$). But then the algorithm would never have got to row i in the first place, it would have stopped at (at least) row j, a contradiction. Therefore a bound for the total number of generators at the end of the algorithm is

$$\min\left(k\,, \frac{n(n-1)}{2}\right) \tag{1}$$

[1] The functions `Multiply` and `Inverse` will now be used implicitly throughout the rest of the project. They will not be referenced in any further code.

This proves that any subgroup of S_n can be generated by at most $n(n-1)/2$ elements.

This leads to the natural question - what is the minimum number of generators needed to produce any subgroup of S_n? We can find the better bound of only $n-1$ generators, by first decomposing G into a strong generating set (something that will be focus of Question 8). Essentially, we may find subgroups G_i which stabilise elements $\alpha_1, \ldots, \alpha_i$, and $G = G_0 \geq G_1 \geq \ldots \geq G_r = 1$, where each inclusion is strict. Starting with G_r we recursively add generators until we produce G, where each generator decreases the number of different orbits when the current subgroup acts on G. Since G_r gives n different orbits (it fixes every element in G), the number of generators for G can be bounded by $n-t$, where t is the number of different orbits in G, and since $t \geq 1$ the result follows.[2]

To bound the number of operations of the stripping algorithm, we first look at how to process one permutation π. On the ith row of the array, the algorithm will check if $\pi(i) = i$ ($O(1)$ operation), as well as check if the $\pi(i)$th cell is empty ($O(1)$ operation). Based on the outcome, the algorithm will either save the permutation ($O(n)$ operations), modify it ($O(n)$ operations from Question 1) or move directly onto row $i+1$. The worst of these options is if the permutation is modified, since it has the largest complexity, and the code cannot immediately move onto the next permutation. Assuming the longest running time, a permutation will be modified on all n rows before being discarded, taking $O(n^2)$ in total. Therefore an upper bound for the number of operations for this algorithm is $O(kn^2)$.

Question 3

Code 3, referenced on page 10, implements the Stripping Algorithm of Sims.[3] The following input was used to test if the code was working properly:

$$\pi_1 = (1,2) \qquad \pi_2 = (1,2,3) \qquad \pi_3 = (2,3,5) \qquad \pi_4 = (1,5,3) \qquad \pi_5 = \mathrm{Id} \qquad n = 5$$

The group generated is equal to S_4 (up to relabelling), since we can produce consecutive transpositions $(2,3) = \pi_1\pi_2$ and $(3,5) = \pi_1\pi_2\pi_3$, and all generators fix the element 4. The output given was the set of permutations

$$\pi_1' = (1,2) \qquad \pi_2' = (2,3) \qquad \pi_3' = (3,5) \qquad \pi_4' = (1,5,3)$$

confirming the results of Question 2. Notice that the number of generators outputted are less than $k = 5$ and $n(n-1)/2 = 10$, confirming the result in equation (1). However it hasn't reduced the number of generators completely, since π_4' is redundant and can be generated from the first three (equals $\pi_3'\pi_2'\pi_1'\pi_2'$). This method serves as an efficient way to keep the number of generators of a permutation group relatively small, but will not output the smallest set of generators.

Question 4

Define the map $f : \mathrm{Orb}_G(\alpha) \to \{\text{left cosets of } G_\alpha\}$ by sending $g(\alpha) \mapsto gG_\alpha$. We first check that this map is well defined; if $g(\alpha) = h(\alpha)$ then $gG_\alpha = hG_\alpha$. We know $h^{-1}g(\alpha) = \alpha$ so $h^{-1}g \in G_\alpha$, or $gG_\alpha = hG_\alpha$. This map is injective, since if $gG_\alpha = hG_\alpha$ then $h^{-1}g \in G_\alpha$. So $h^{-1}g(\alpha) = \alpha$ or $h(\alpha) = g(\alpha)$. The map is also surjective, since all left cosets gG_α have the corresponding $g(\alpha)$ in $\mathrm{Orb}_G(\alpha)$ by the definition of an orbit. Therefore this map is a bijection.

The Orbit-Stabiliser theorem is as follows: Let G be a finite group acting on a set X. Then for all $\alpha \in X$, $|G| = |\mathrm{Orb}_G(\alpha)| \cdot |G_\alpha|$. The proof uses the fact that the set of left cosets of G_α partitions G,[4] and therefore $|\{\text{left cosets of } G_\alpha\}| \cdot |G_\alpha| = |G|$. Using the bijection in the first part, $|\mathrm{Orb}_G(\alpha)| = |\{\text{left cosets of } G_\alpha\}|$, and the result follows.

[2] A more rigorous proof is presented in Jerrum, "A compact representation for permutation groups"

[3] The function `SAOS` defined in Code 3 will be used implicitly throughout the rest of the project. It will no longer be referenced in any further code.

[4] A proof of this is covered in Part 1A Groups.

Question 5

Code 5, referenced on page 11, was used to calculate the orbit of an element $\alpha \in X$.[5] The following test data was used: $\pi_1 = (2,3,4)$, $\pi_2 = (1,5)$, $\alpha = 2$, $n = 5$, and the correct output was

Image of $\alpha = 2$	Witness Permutation
2	Id
3	$(2,3,4)$
4	$(2,4,3)$

Table 1: Test data for Question 5

To calculate the orbit of α, the code starts by creating a 'queue' which initially only contains α. It also creates a $n \times 2$ array which will save the elements of the orbit with their corresponding witnesses, and will also act as a hash table for efficient indexing. It proceeds to apply each generator to the first element in the queue, and saves any new element i to the i-th row of the array, as well as the generator that created it. It then adds i to the back of the queue. The code iterates through the elements of this queue, applying each generator to the current element, saving any new elements i and the generators that produced them, and appending these i to the back of the queue. The code will terminate when we reach the end of this queue.

This method does indeed produce the complete orbit. Suppose $\beta \in \mathrm{Orb}_G(\alpha)$, then $\beta = \pi_r \dots \pi_1(\alpha)$ where each π_i is a generator, and r is the smallest integer where this is possible. Let $P_i = \pi_i \dots \pi_1(\alpha)$, so that $P_r = \beta$. Then each P_i must be unique, otherwise we could find a smaller value for r. Therefore, by only testing new elements in the queue, we will generate the entire orbit.

The complexity of the algorithm is bounded by $O(kn^2)$, where k is the number of generators of G. For each element in the queue, k generators are applied to it, each taking $O(1)$ operations. It checks if any new element has been produced, which is why the additional array was created - it allows the code to check if element i has already been produced with complexity $O(1)$ (not searching $O(n)$) since i would be saved in position i. However this does mean this code requires more memory to run. If this element has not been produced, it will save the element, and permutation that produced it, which is $O(n)$. Therefore for each element in the queue, the complexity is bounded by $O(kn)$, and since the queue has length at most n, the total complexity is bounded by $O(kn^2)$.

Question 6

We aim to prove by induction that $t_{i+1} = \varphi(y_i \dots y_1)$. If $g = hz$, where $g, h \in G$ and $z \in G_\alpha$, then $gG_\alpha = hzG_\alpha = hG_\alpha$ and so $\varphi(g) = \varphi(h)$. When $i = 1$, $t_2 = \varphi(y_1 t_1)$, but $t_1 \in G_\alpha$ and so $t_2 = \varphi(y_1)$ and the base case holds. Now assume that $t_{k+1} = \varphi(y_k \dots y_1)$ for some $k \in \mathbb{Z}$. We know that $y_k \dots y_1 = \varphi(y_k \dots y_1)g$ for some $g \in G_\alpha$ (left cosets partition G), so $t_{k+2} = \varphi(y_{k+1}\varphi(y_k \dots y_1)) = \varphi(y_{k+1} y_k \dots y_1 g^{-1})$. But $g^{-1} \in G_\alpha$, so this equals $\varphi(y_{k+1} \dots y_1)$. Therefore the inductive step holds. Immediately we conclude $t_{r+1} = \varphi(y_r \dots y_1) = \varphi(x) = t_1$.

Let $S = \langle \{\varphi(yt)^{-1} \cdot y \cdot t \mid y \in Y, t \in T\} \rangle$ and $t_1 = t_{r+1} = 1$; the identity. Then

$$\begin{aligned} x &= t_{r+1}^{-1} \cdot y_r \cdot t_r \cdot t_r^{-1} \cdot y_{r-1} \cdot t_{r-1} \cdot t_{r-1}^{-1} \cdot y_{r-2} \cdot t_{r-2} \dots t_2^{-1} \cdot y_1 \cdot t_1 \\ &= \varphi(y_r t_r)^{-1} y_r t_r \cdot \varphi(y_{r-1} t_{r-1})^{-1} y_{r-1} t_{r-1} \dots \varphi(y_1 t_1) y_1 t_1 \end{aligned}$$

It is visible that x can be generated by the elements in S, and since x was arbitrary, $G_\alpha \leq S$. To check the other inclusion, notice that $yt = \varphi(yt)g$, for some $g \in G_\alpha$, and so $yt(\alpha) = \varphi(yt)(\alpha)$. Therefore $\varphi(yt)^{-1} yt(\alpha) = \alpha$, and so all the generators fix α. Therefore $G_\alpha \geq S$, and so conclude that $G_\alpha = S$.

[5] The function `Orbit` defined in Code 5 will be used implicitly throughout the rest of the project, and will no longer be included in subsequent referencing.

Question 7

Code 7, referenced on page 12, was used to produce a generating set for the stabiliser of an element.[6] It works as follows: since the orbits form a bijection with the set of left cosets of G_α, the permutations produced by Code 5 form a complete set for T. Since each of these permutations send α to a different element, we use this to define the φ function. The code then iterates through creating S, and reduces it with the stripping algorithm.

The following test data was used: $Y = \{(1\,2), (1\,2\,3\,4\,5)\}$, $n = 5$ and $\alpha = 5$. In Question 9 there is a proof that $\langle Y \rangle = S_5$, and so we expect the stabiliser to equal to S_4, made up of all permutations that fix 5. The output was the set of generators

Stabiliser for $\alpha = 5$
$(1, 2)$
$(1, 3)$
$(1, 4)$
$(2, 3, 4)$

Table 2: Test data for Question 7

The first three of these generate S_4, confirming this code works.
Question 5 already has complexity $O(kn^2)$. Since $|Y| = k$ and $|T| \leq n$, then the set that generates G_α will have size at most kn. Each permutation has to be stored, taking a total of $O(kn^2)$ operations. Finally applying the stripping algorithm to the set of kn generators gives $O(kn^3)$ operations. Therefore computing the generators of a stabiliser is bounded by $O(kn^3)$.

Question 8

By producing a stabiliser chain $G = G_0 \geq G_1 \geq G_2 \geq \ldots \geq G_r = \{e\}$, where G_i is the stabiliser for elements $\alpha_1, \ldots, \alpha_i$, each of which have been picked such that $|\mathrm{Orb}_{G_{i-1}}(\alpha_i)| > 1$, we can conclude from repeatedly applying the Orbit Stabiliser theorem that

$$|G| = \prod_{i=1}^{r} |\mathrm{Orb}_{G_{i-1}}(\alpha_i)| \tag{2}$$

Notice $|G| \leq |S_n| = n!$ is finite, so the orbit stabilizer theorem applies.

This is an efficient way of calculating the size of a permutation group. Suppose there were k initial generators, so stripping them would take $O(kn^2)$ operations and give a generating set of size less than n^2 (from equation (1)). Iterating through X to find an element α_i with a non-trivial orbit would therefore take at most $O(n^2 \cdot n^2 \cdot n)$ operations. Calculating the corresponding stabilizer (from Question 7) would also take $O(n^2 \cdot n^3)$ operations, with this stabilizer having at most n^2 generators as well. All of this would iterate at most $\log_2(|G|)$ times, and so we conclude that the algorithm has complexity at most.

$$O\left(\log_2(|G|)\, n^5\right) \tag{3}$$

Since $|G| \leq n! \leq n^n$, $\log_2(|G|) \leq n^2$. Hence this algorithm runs in polynomial time.

If we didn't use the stripping algorithm at every stage, the number of generators would grow out of control. Suppose k generators were initially inputted in Code 8, and G_i and α_i are defined as in the start of this question. Then after the first iteration we would have saved $k \times |T| = k \times |\mathrm{Orb}_{G_0}(\alpha_1)|$ generators, after the second iteration we would have saved $k \times |\mathrm{Orb}_{G_0}(\alpha_1)| \times |\mathrm{Orb}_{G_1}(\alpha_2)|$, and after the code had completed we would have saved $k|G|$ generators. A lower bound for the complexity would therefore be $O(kn|G|)$, since each generator would have to be saved. Should $|G|$ be large ($20! \approx 10^{18}$) a large amount of memory would be required. More importantly, since $|G|$ could equal $n!$ then this program would run

[6] The function `Stabiliser` defined in Code 7 will be used implicitly throughout the rest of the project, and will not be included in subsequent referencing.

in exponential time, and so is not efficient.

Code 8, referenced on page 13, was used to calculate the order of the group produced by a set of given generators.[7] The following set was used as test data:

$$y_1 = (1,2,3) \qquad y_2 = (1,2,4) \qquad y_3 = (1,2,5) \qquad y_4 = (1,2,6) \qquad n = 6$$

The output was the table

i	$\lvert G_i\rvert$	Gens before strip	Gens after strip	α_{i+1}	$\lvert \mathrm{Orb}_{G_i}(\alpha_{i+1})\rvert$
0	360	4	4	1	6
1	60	24	9	2	5
2	12	45	5	3	4
3	3	20	2	4	3
4	1	6	0	$*$	$*$

Table 3: Test data for Code 8

Therefore $|G| = 360$. This means that $G = A_6$, since the generators only contain even permutations and $6!/2$ different permutations have been produced.[8]

Another application of this code is to calculate the number of Rubik's Cube combinations. Each configuration can be represented as a permutation of 48 elements, more exactly, these are the 48 'outer' parts of each face. The figure below represents an example of such a numbering.

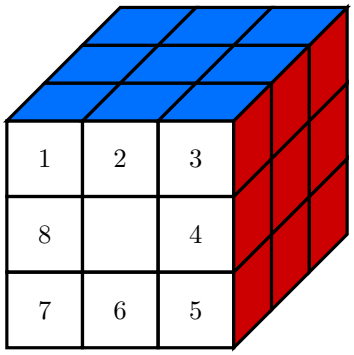

Figure 1: Rubik's Cube with the white face labelled

The generators of this group are the 90 degree rotations of each face, referenced on page 8 as tables. The answer is the surprising large number

$$|G| = 43252003274489856000$$

This is testament to the method behind this algorithm, that it would be near impossible to list all combinations, but the algorithm works in $O\left(\log_2\left(|G|\right) n^5\right) \approx O(10^{10})$ operations, a program most commercial computers could implement.

A major application of this algorithm which does not appear immediately obvious is that we can efficiently test an element for membership in finite groups. Given a set of generators $\pi_1, \ldots, \pi_k$ for a known group G, we calculate the size of the group generated by the new set $\pi_1, \ldots, \pi_k, x$ for some element x. Should this be equal to $|G|$, then we conclude that x must lie in this group. Otherwise, we can conclude that it doesn't.

An example of this is the Rubik's subgroup generated by 5 face rotations instead of 6. By adding the 6th rotation, the size of the group does not increase, proving that there exists *some* combination of the 5 face rotations that produces the 6th.

[7] The function `Order` defined in Code 8 will no longer be included in subsequent referencing.

[8] It is true in general that A_n is generated by the permutations $(1,2,3), \ldots, (1,2,n)$. A proof is laid out in Theorem 3.3 in Conrad, "Generating Sets".

By outputting the order of a group, a lot can be determined about its structure. This method could give a way of determining if certain permutation groups are not simple by applying the Sylow theorems. These theorems depend on the prime factorisation of the size of a group (currently a difficult task that cannot be done in polynomial time), however this algorithm already gives the answer in a partially factorised form, allowing for quicker processing.

Question 9

The two permutations $(1,2)$ and $(1,2,\ldots,n)$ generate S_n. Let $\sigma = (1,2,\ldots,n)$. Then

$$\sigma^k(1,2)\sigma^{-k} = (\sigma^k(1), \sigma^k(2)) = (k+1, k+2) \tag{4}$$

But consecutive transpositions generate S_n and hence the set $\{(g,h) \in S_n \times S_n : \langle g,h\rangle = S_n\}$ is non-empty and $P_n > 0$.[9]

If $(g,h) \in A_n \times A_n$ then no odd permutation could be produced by these generators. Since $|A_n| = n!/2$, the probability that this occurs is $1/4$, and therefore $P_n \leq 3/4$ ($k = 3/4$). This is only true for $n \geq 2$, since $A_1 = S_1 = \{1\}$.

When n is very small we can calculate each of the combinations of permutations, and by brute force find P_n. However there are $n!(n!-1)/2$ combinations to check, which diverges quickly, and so this method is not feasible for large n. Code 9, referenced on page 14, was used to calculate the exact values of P_n for $1 \leq n \leq 6$.

n	P_n
1	1
2	3/4
3	1/2
4	3/8
5	19/40
6	53/120

Table 4: Exact values for P_n

We can see that the values aren't larger than $3/4$, apart from the case when $n = 1$, as discussed.

To generate a random permutation we could pick a random number from a list from 1 to n, save this as the image of 1. Then pick a random element from this list, and if it hasn't already been chosen, set this as the image of 2, and so on. However this is inefficient, since it unnecessarily checks to see if numbers have already been chosen. Instead we can generate permutations using the Fisher-Yates shuffling algorithm, which does not require these checks. It starts with an arbitrary permutation, say the identity, and chooses a random number γ_1 from 1 to n. It then proceeds to switch the n-th element with the γ_1-th element. It then generates a random number γ_2 from 1 to $n-1$, and switches element $n-1$ with γ_2, and so forth. A simple induction proves that this method is unbiased, and we can see that this does not require to check that any element has already been taken.

Suppose we wish to estimate P_n for $1 \leq n \leq k$ for some $k \in \mathbb{Z}$, accurate to 2 decimal places at a 0.95 confidence level. We would therefore have to estimate a single P_n at a $\sqrt[k]{0.95}$ confidence level. Suppose m independent samples $X_i \sim \text{Bernoulli}(P_n)$ are taken. If m is sufficiently large, by the central limit theorem we can assume

$$\frac{1}{m}\sum_{i=1}^{m} X_i \sim P_n + \frac{\sqrt{P_n\,(1-P_n)}}{\sqrt{m}}Y \tag{5}$$

where $Y \sim \mathcal{N}(0,1)$. Therefore a $\sqrt[k]{0.95}$ confidence interval is $[-\Phi((1+\sqrt[k]{0.95})/2), \Phi((1+\sqrt[k]{0.95})/2)]$. Maximising $\sqrt{P_n(1-P_n)}$ over $P_n \in [0,1]$ gives $1/2$, and so it suffices to choose a sample of size

$$m = \left(\Phi\left(\frac{1+\sqrt[k]{0.95}}{2}\right) \times 10^2\right)^2 \tag{6}$$

[9] A straightforward proof that consecutive transpositions generate S_n is laid out in Conrad, "Generating Sets"

When $k = 14$, $m \approx 8.4 \times 10^4$, which is too large. However choosing $846 < m$ ensures all estimates should be accurate to at least one decimal place, to give an indication of accuracy.

Code 10, referenced on page 15, was used to generate random permutations and produce estimates for P_n when $k = 14$, as well as plot them against n in Figure 2, when $m = 3000$.

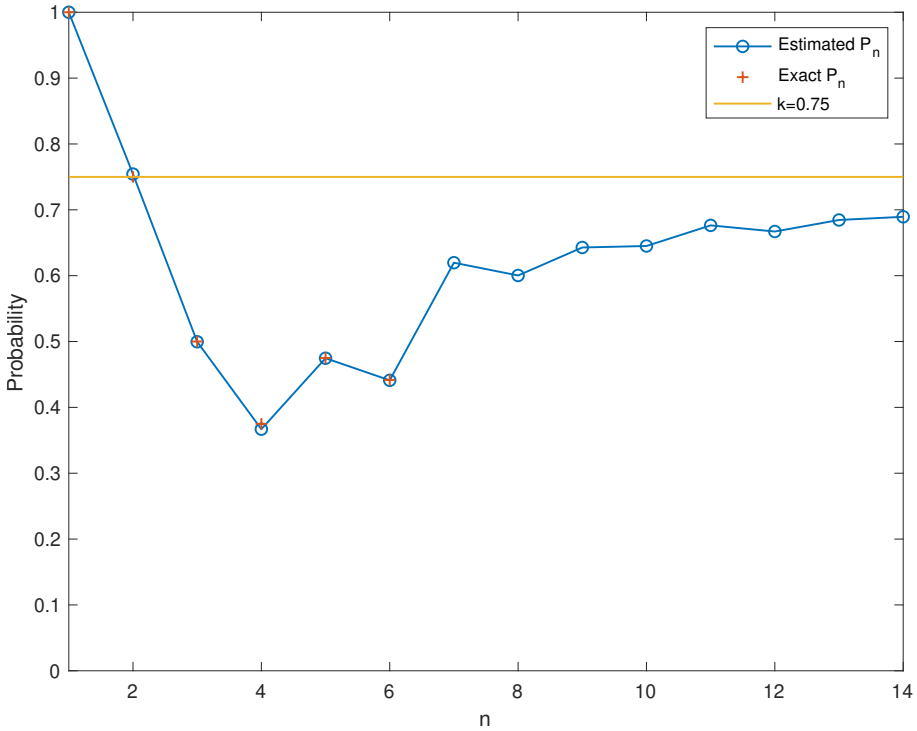

Figure 2: P_n for different n

Firstly we see that the first few estimates are incredibly close to the exact values of P_n in Table 4, which confirms the algorithm is indeed producing random permutations. We can also see that $0 < P_n \leq 3/4$, confirming the argument at the start of the question.

An interesting feature is that it appears that P_n tends to some constant. This is in fact true, with $P_n \to 3/4$ as $n \to \infty$. This is a corollary on Dixen's theorem, which states the proportion of pairs of permutations from S_n that generate either the group S_n or A_n is greater than

$$1 - \frac{2}{(\log \log n)^2} \tag{7}$$

for sufficiently large n.[10] We can see that this lower bound tends to 1, and since the probability that at least one of the generators being odd is 3/4, the result follows.

References

Conrad, Keith. "Generating Sets". In: (). URL: https://kconrad.math.uconn.edu/blurbs/grouptheory/genset.pdf.

Dixon, John D. "The probability of generating the symmetric group". In: *Mathematische Zeitschrift* 110.3 (1969), pp. 199–205. DOI: 10.1007/BF01110210. URL: https://doi.org/10.1007/BF01110210.

Jerrum, Mark. "A compact representation for permutation groups". In: *Journal of Algorithms* 7.1 (1986), pp. 60–78. ISSN: 0196-6774. DOI: https://doi.org/10.1016/0196-6774(86)90038-6. URL: https://www.sciencedirect.com/science/article/pii/0196677486900386.

[10] The proof that Dixen presented can be found in Dixon, "The probability of generating the symmetric group"

Rubik's Cube Permutations

These are the generators inputted into Code 8 when calculating the size of the Rubik's Cube group. A 2×48 size table won't fit onto this document, so only the elements that get changed by the generator have been noted, and all other faces on the Rubik's Cube that have been omitted can be considered fixed.

1	2	3	4	5	6	7	8	9	15	16	27	28	29	37	38	39	41	42	43
7	8	1	2	3	4	5	6	39	37	38	41	42	43	27	28	29	15	16	9

Table 5: White Face Rotation

3	4	5	9	10	11	12	13	14	15	16	17	23	24	35	36	37	43	44	45
43	44	45	15	16	9	10	11	12	13	14	37	35	36	3	4	5	23	24	27

Table 6: Red Face Rotation

1	2	3	9	10	11	17	18	19	25	26	27	33	34	35	36	37	38	39	40
9	10	11	17	18	19	25	26	27	1	2	3	39	40	33	34	35	36	37	38

Table 7: Blue Face Rotation

1	7	8	19	20	21	25	26	27	28	29	30	31	32	33	39	40	41	47	48
33	39	40	47	48	41	31	32	25	26	27	28	29	30	21	19	20	1	7	8

Table 8: Orange Face Rotation

11	12	13	17	18	19	20	21	22	23	24	25	31	32	33	34	35	45	46	47
45	46	47	23	24	17	18	19	20	21	22	35	33	34	11	12	13	31	32	25

Table 9: Yellow Face Rotation

5	6	7	13	14	15	21	22	23	29	30	31	41	42	43	44	45	46	47	48
29	30	31	5	6	7	13	14	15	21	22	23	47	48	41	42	43	44	45	46

Table 10: Green Face Rotation

Code

Code 1

```
A=[1 2 3 4;3 4 2 1]
B=[1 2 3 4;2 1 3 4]

Multiply(A,B)
Inverse(A)

function answer = Multiply(A,B)
    Counter=1;
    answer=zeros(2,size(A,2));
    while Counter<=size(A,2)
        answer(1,Counter)=Counter;
        answer(2,Counter)=A(2,B(2,Counter));
        Counter=Counter+1;
    end
end

function answer = Inverse(A)
    answer=zeros(2,size(A,2));
    answer(1,:)=A(1,:);
    Counter=1;
    while Counter<=size(A,2)
        answer(2,A(2,Counter))=Counter;
        Counter=Counter+1;
    end
end
```

Code 3

```
[Array,NewGenerators]=SAOS(5,{[1 2 3 4 5;2 1 3 4 5],[1 2 3 4 5;2 3 1 4 5],...
    [1 2 3 4 5;1 3 5 4 2],[1 2 3 4 5;5 2 1 4 3],[1 2 3 4 5;1 2 3 4 5]})


function [Array,NewGenerators] = SAOS(n,OldGenerators)
Identity=[1:n;1:n];
NewGenerators=cell(1,0);

Array=cell(n,n);
Counter=1;
while Counter<=size(OldGenerators,2)
    RowCounter=1;
    %Checks if current permutation is the identity
    while RowCounter<=n
        if OldGenerators{Counter}==Identity
            break
        end
        %Checks if the permutation fixes Counter
        if OldGenerators{Counter}(2,RowCounter)~=RowCounter
            %If cell is empty, fills cell in array
            if isempty(Array{RowCounter,OldGenerators{Counter}...
                (2,RowCounter)})==1
                Array{RowCounter,OldGenerators{Counter}(2,RowCounter)}=...
                        OldGenerators{Counter};
                NewGenerators={NewGenerators{1,:} OldGenerators{Counter}};
                    break
            %If cell isn't empty, modifys permutation to fix Counter
            else
                OldGenerators{Counter}=Multiply(Inverse(Array{...
                    RowCounter,OldGenerators{Counter}(2,RowCounter)}),...
                    OldGenerators{Counter});
            end
        end
        RowCounter=RowCounter+1;
    end
    Counter=Counter+1;
end
end
```

Code 5

```
n=5;
Generators={[1 2 3 4 5;1 3 4 2 5],[1 2 3 4 5;5 2 3 4 1]};
Alpha=2;

Orb=Orbit(n,Generators,Alpha);
disp(Orb)

function [answer,siz] = Orbit(n,Generators,Alpha)
answer=cell(n,2);
answer{Alpha,1}=Alpha;
answer{Alpha,2}=[1:n;1:n];
Queue=[Alpha];

Counter=1;
UpperLimit=1;
while Counter<=UpperLimit
    GenCounter=1;
    while GenCounter<=size(Generators,2)
        Image=Generators{1,GenCounter}(2,Queue(Counter));
        if isempty(answer{Image,1})==1
            UpperLimit=UpperLimit+1;
            Queue=[Queue,Image];
            answer{Image,1}=Image;
            answer{Image,2}=Multiply(Generators{1,GenCounter},answer{Queue(Counter),2});
        end
        GenCounter=GenCounter+1;
    end
    Counter=Counter+1;
end
siz=size(Queue,2);
end
```

Code 7

```
n=5;
Generators={[1 2 3 4 5;2 1 3 4 5],[1 2 3 4 5; 2 3 4 5 1]};
Alpha=5;
Stab=Stabiliser(n,Generators,Alpha);
[~,Stab]=SAOS(n,Stab);
disp(Stab)

function S = Stabiliser(n,Generators,Alpha)
OrbitArray=Orbit(n,Generators,Alpha);
S=cell(1,0);

Counter=1;
while Counter<=n
    SubCounter=1;
    while SubCounter<=size(Generators,2)
        if size(OrbitArray{Counter,1},1)==0
            break
        end
        %Calculates yt
        Perm=Multiply(Generators{1,SubCounter},OrbitArray{Counter,2});
        %Equals Varphi(yt)
        Varphi=OrbitArray{Perm(2,Alpha),2};
        S={S{1,:} Multiply(Inverse(Varphi),Perm)};
        SubCounter=SubCounter+1;
    end
    Counter=Counter+1;
end
end
```

Code 8

```
n=6;
Generators={[1 2 3 4 5 6;2 3 1 4 5 6],[1 2 3 4 5 6;2 4 3 1 5 6],...
    [1 2 3 4 5 6;2 5 3 4 1 6],[1 2 3 4 5 6;2 6 3 4 5 1]};

tic
Order(n,Generators)
toc

function Data=Order(n,Generators)
Data=zeros(1,5);
Data(1,2)=size(Generators,2);
GenArray=Generators;
[~,GenArray]=SAOS(n,GenArray);
Data(1,3)=size(GenArray,2);

while size(GenArray,2)~=0
    Counter=1;
    while Counter<=n
        [~,Size]=Orbit(n,GenArray,Counter);
        if Size>1
            break
        end
        Counter=Counter+1;
    end
    if Counter==n+1
        break
    end
    Data(end,4)=Counter;
    Data(end,5)=Size;
    Data=[Data;0,0,0,0,0];
    GenArray=Stabiliser(n,GenArray,Counter);
    Data(end,2)=size(GenArray,2);
    [~,GenArray]=SAOS(n,GenArray);
    Data(end,3)=size(GenArray,2);
end
Data(end,1)=1;
Counter=size(Data,1)-1;
while Counter>=1
    Data(Counter,1)=Data(Counter+1,1)*Data(Counter,5);
    Counter=Counter-1;
end
end
```

Code 9

```
n=5;
Identity=[1:n];
Perms=perms(Identity);
Tally=0;

Counter=1;
tic
while Counter<=factorial(n)-2
    A=[1:n ;Perms(Counter,:)];
    SubCounter=Counter+1;
    while SubCounter<=factorial(n)-1
        B=[1:n; Perms(SubCounter,:)];
        lineLength = fprintf('Row = %f, Column = %f',Counter/factorial(n),...
            SubCounter/factorial(n));
        O=Order(n,{A,B});
        if O(1,1)==factorial(n)
            Tally=Tally+1;
        end
        fprintf(repmat('\b',1,lineLength))
        SubCounter=SubCounter+1;
    end
    Counter=Counter+1;
end
toc
format rat
disp(2*Tally/factorial(n)^2)
```

Code 10

```
tic
StartN=1;
FinishN=10;
SampleSize=1000;

ExactPN=[1 1;2 3/4;3 1/2;4 3/8;5 19/40;6 53/120];
Tally=zeros(FinishN-StartN+1,2);
n=StartN;
while n<=FinishN
Counter=1;
SubTally=0;
while Counter<=SampleSize
    lineLength = fprintf('n = %d, %.1f%% complete',n,Counter*100/SampleSize);
    Perms=random(2,n);
    Data=Order(n,Perms);
    if Data(1,1)==factorial(n)
        SubTally=SubTally+1;
    end
    Counter=Counter+1;
    fprintf(repmat('\b',1,lineLength))
end
Tally(n-StartN+1,1)=n;
Tally(n-StartN+1,2)=SubTally/SampleSize;
n=n+1;
end
disp(Tally)

figure
plot(linspace(StartN,FinishN,FinishN-StartN+1),Tally(:,2),'-o','LineWidth',1)
hold on
scatter(ExactPN(:,1),ExactPN(:,2),'+','LineWidth',1)
plot(linspace(StartN,FinishN,FinishN-StartN+1),linspace(0.75,0.75,FinishN-StartN+1),'LineWidth',1)
legend('Estimated P_{n}','Exact P_{n}','k=0.75')
xlabel('n')
ylabel('Probability')
xlim([StartN,FinishN])
ylim([0,1])
print('Image_1','-depsc')
toc

function answer = random(quantity,n)
answer=cell(1,0);
Counter=1;
while Counter<=quantity
    Perm=[1:n;1:n];
    SubCounter=1;
    while SubCounter<=n-1
        Carry=Perm(2,SubCounter);
        RandIndex=randi(n-SubCounter+1)+SubCounter-1;
        Perm(2,SubCounter)=Perm(2,RandIndex);
        Perm(2,RandIndex)=Carry;
        SubCounter=SubCounter+1;
    end
    answer={answer{:,:} Perm};
```

```
    Counter=Counter+1;
end
end
```