DNNs F - Compositional generalisation in group structured data

1 Introduction & Hypothesis

My mini-project focuses on compositional generalisation in group structured data. In this report, I analyze how transformers and graph neural networks (GNNs) generalise subgroups. I hypothesise that transformers may struggle because they may have difficulty handling examples when they aren't explicitly trained on a diverse set of rules. In contrast, GNNs might leverage their inherent structure to better capture subgroup properties, a hypothesis I rigorously evaluate through a carefully structured, logically detailed experiment on only symmetric groups as they are compositionally rich.

2 Data Construction & Representation Choices

In this exercise we focus mainly on the symmetric group S_n and the modular arithmetic group Z_p^k and we will focus on how we represent these in this section. (See Appendix A.1 for full definitions)

2.1 Representation

Possible Cayley table representations include adjacency matrices, which encode operations directly as matrix entries (see Appendix A.2.1); positional encodings, which enhance the representation with positional information for better model interpretability (A.2.2); direct element pair representation (A.2.3); and graphical dependencies (A.2.4).

In terms of the choice of representation, this depends on the models that we use. For the transformer models, we use the direct element representation because it encodes each permutation element explicitly, preserving the inherent order and structure. This explicit encoding aligns well with the transformer's attention mechanism, enabling the model to learn relationships between elements more effectively.

Additionally, for GNNs we use the graphical dependencies. This approach leverages the natural graph structure of the permutation data, allowing the model to capture complex inter-element dependencies through message passing between nodes. The graphical representation is intrinsic to GNN architectures, enhancing their ability to model relational patterns in the data.

We encode each permutation as tokens rather than the actual elements. For example, consider $[1, 2, 3]$.

Two encoding schemes are available. In the first, the permutation is mapped to a unique integer $n \in \mathbb{Z}$. In the second, it is represented as $[\pi(1), \pi(2), \pi(3)]$, with $\pi : \mathbb{Z} \to \mathbb{Z}$ reordering the elements.

Both methods encapsulate the entire permutation as tokens, each offering its own advantages. We still shuffle the numbers in the combinations because the transformer can learn the operations using simple addition for both the symmetric group and modular arithmetic groups [trivially] otherwise. [1]

In this report, we employ a single integer encoding for the group $\mathbb{Z}_p^k$ while retaining a list structure for the symmetric groups S_n. This decision is motivated by the following: for groups with repeated elements, using a list encoding would simply replicate the dataset and make learning trivial; whereas preserving the list structure for S_n (and its subgroups) maintains the sequential information necessary to capture complex relational patterns, paired with the empirical evidence that these methods train better.

2.2 Sampling Strategies

We compare structured subgroup-based sampling with random sampling. Subgroup-based sampling ensures controlled structural diversity, perhaps enhancing compositional generalization, whereas random sampling produces a less structured distribution that may limit interpretability or provide variety.

[1] By representing the permutation as tokens and shuffling the numbers, the transformer should learn the underlying arithmetic operations rather than rely on fixed positions. This forces the model to capture the compositional structure, which generalizes to both symmetric and modular arithmetic groups.

3 Generalisation of groups in transformers

This section aims to summarise the results with transformer experiments and hypothesise how well they generalise with subgroups. Here we experiment over the modular arithmetic group first before analysing the symmetric groups. From section E we know that transformer generalise well with more than 50% of training data therefore will do our experiments with smaller training splits.

3.1 Methodology

We kept the model parameters (i.e., model size) constant for consistency while varying the hyperparameters through grid search and early stopping based on inference performance. A learning rate scheduler and validation set convergence were used to mitigate the influence of the initial hyperparameter settings (including optional warm-up LR). The primary hyperparameters searched were weight decay, dropout rate, batch size (and the number of epochs, learning rate). In general we found that dropout controlled the overfitting and we were looking for the highest value such that the training loss was still a decreasing function, this ended up around 0.5 on average, that batch size showed minimal changes as we were working in low data environments and then iterated over values for weight decay using known values from section E as a guide, ending up choosing in the order of 1e-2.

3.2 Experiment 1: Z_6^2, Z_9^2

Model	Accuracy (%)	Standard Deviation (%)
Random Sampling	80.44	24.21
Subgroup Sampling	67.96	35.67

Table 1: Accuracy and error margins for Z_6^2.[2]

In this experiment our method was to take all of the 1D and 2D subgroups. [3] We can immediately see that the model struggles to generalise but this is a very weak argument because we do not have statistical significance, seen with the high variability of the results and we are only testing one configuration and group. [4]
In an attempt to generalise our argument, we can look into a few other groups including Z_2^7 and Z_3^3 and in both of these we notice a consistent lack of generalisation behaviour. This would be described by the extreme where we have a lack of tokens observed leading to the models over-fitting on the training data and significantly struggle. See Appendix B.1 for full experimental details. Through these experiments, we start to believe that transformer struggle to generalise while extrapolating from subgroups to groups compared to random sampling. We can now see what happens when we try the same experiments on symmetric groups instead.

3.3 Experiment 3: Symmetric Groups

Group	Method	Accuracy ± 1 std (%)
S_5	Random Sampling	100 ± 0.1
	Subgroup Sampling	70.4 ± 13.4

Figure 1: Accuracy across methods

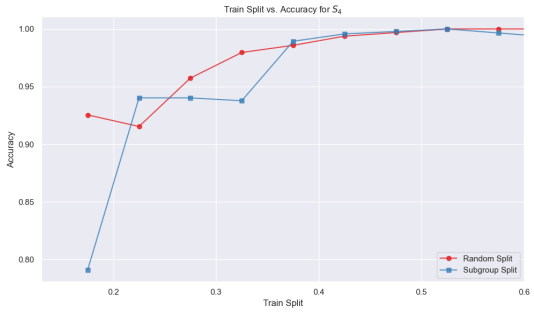

Figure 2: S_4 accuracy vs training split

At first we work with S_5 and we randomly chosen subgroups which are one and two-dimensional to meet a 30% data split. From this slice, we can notice that subgroups appear to hinder training performance within S_5 like above. To do further analysis of this we move into S_4 to try and get more detailed evaluation.
Initially, with S_4 we did experiments with random selections of the subgroups in order to try and notice any patterns in the efficiency of subgroups. In the figure you can see how the training split seems to affect the accuracy across using

[3] The dimensionality of a subgroup defined by the number of basis permutations, these are generally the smaller subgroups
[4] Note: For this we use tiny transformers and use at least 7 repeats

random splits as opposed to random subgroups. From this figure we seem to notice that complete random sampling is slightly more effective than random subgroup selection, which reinforces our initial results from S_5.
However, this graph is very noisy and we can hypothesise this is because some subgroups are much better than others. Therefore, now we can assign each individual subgroup a rank based on how often it appears in a training run that achieves a high accuracy. Now, using this total order on the subgroups, we can complete experiments while choosing each subgroup based on rank.
From the analysis we notice that training on just the first few ranked subgroup deems no significant improvement compared to random sampling. See Appendix B.2 for full experimental details as we analyse the hypothesis test that the models are different and conclude that there is no significance showing difference using an independent t-test. [5]

3.4 Conclusions

From this, we can hypothesise that Transformers struggle to generalise when trained on subgroups. We initially observe that sampling smaller, random subgroups yields worse performance than training on the entire available sample set; moreover, hand-picking "best" subgroups does not ultimately outperform random selection in a statistically significant way.
This may be because Transformers are data-hungry and require highly representative, diverse training data to learn robust representations. Randomly sampled subgroups may fail to capture the full variability of the dataset, leading to poorer generalisation.[6] Conversely, even when we attempt to select the most "informative" subgroups, we do not see a significant improvement in low-data environments. To draw firm conclusions, one would need to explore a broader range of model configurations, since our findings are specific to this "tiny" Transformer.[7] Finally, training on only 10% of a massive dataset may not fully capture the benefits of scale—larger Transformers, trained on more extensive data, often tend to generalise far better in real-world applications.

4 Generalisation of symmetric groups in GNNs

Having explored the efficacy of Transformers in capturing complex dependencies, a logical next step is to consider Graph Neural Networks (GNNs), which are naturally suited to structured data.[8] We use GNNs that have message-passing mechanism to learn representations that capture both local and global structure, regardless of node labels. This makes them particularly effective for learning functions under symmetry groups such as the symmetric group S_n or its subgroups, where permuting node labels should not alter underlying properties. [9] By designing GNN architectures that are invariant or equivariant to these permutations, one can directly leverage group-theoretic insights—reducing redundancy, improving generalization, and ensuring that the model's outputs remain consistent under re-labellings imposed by specific subgroups.

4.1 Methodology

For this particular task, I chose to use a Graphical Convolutional Network as my choice of GNN. This is a network that operates by letting each node update its features by aggregating information from its immediate neighbours. This approach is especially useful in graphs where edges can represent different relationships (e.g., in knowledge graphs or multi-relational data), and it supports more expressive modeling of how nodes influence one another. An example of GCNs' ability to capture group-based relationships is seen in their application to citation networks, where research papers form natural communities that enhance classification performance[10].
We train our RGCN-based model using mini-batches, computing gradients and updating weights each epoch via standard optimizers (Adam). We then evaluate performance on a validation set to monitor over-fitting and employ early stopping when the validation loss stops improving for a specified number of epochs [patience]. Hyperparameter tuning—covering factors such as learning rate, number of epochs, dropout rate, and hidden dimension size—is done via grid searches, using inference to control the change between runs, with the best configurations chosen based on validation accuracy and loss trends. This balanced approach helps the model converge efficiently and maintain good generalization.
See Appendix C.2 for full setup and algorithmic details. Out of the (hyper)parameters here, we chose the number of convolutional layers [3] and hidden channels [128] at first by training on a network and finding the minimal values where we were able to successfully represent the data [see Experiment 0]. Next, we chose the LR scheduler and convergence parameters empirically by observing some full runs for the patience values and then chose 'ReduceLROnPlateau' as a

[5] Omitted from main section as trivial results
[6] Vaswani et al. (2017) introduced the original Transformer, underscoring the importance of attention in processing diverse training.
[7] Kaplan et al. (2020) observe that some models exhibit abrupt gains, potentially due to emergent capabilities at specific sizes.
[8] Foundational works in deep learning often treat GNNs within a unified framework that emphasizes the preservation of structure.
[9] Research in graph representation similarly emphasizes the importance of leveraging these symmetries to enhance generalization.
[10] Kipf, T. N., Welling, M. (2017). *Semi-Supervised Classification with Graph Convolutional Networks* (ICLR).

simple adaptive learning scheme that tries to avoid early stopping and encourage grokking. Finally, we did a grid search on the remaining tunable hyperparameters. [11]

4.2 Experiment 0

In our initial experiment, our goal was to identify model parameters that enable the learning of symmetric groups. To achieve this, we conducted a preliminary search—essentially an inference-based or binary search—over the number of channels and convolutional layers, aiming to determine the minimal architecture capable of learning S_4 from randomly split data. Binary search is an efficient algorithm that repeatedly halves the search space at each step, allowing us to quickly converge on an optimal configuration.

As a result of this search, we trained on 90% of S_4 and in particular with 128 hidden channels and 3 convolutional layers we achieved an accuracy of over 75% consistently on the test set. [12]

4.3 Experiment 1: S_4

The initial experiments on S_4 seem to imply that subgroups tend to generalise better than random splits in low-resource environments.
This figure provides strong evidence that, across different training splits, subgroup-based sampling consistently achieves statistically significant performance compared to training on random splits of the Cayley table. See Appendix B.3.1 for full experimental setup and the statistical analysis demonstrating the significance of this difference.
The initial conclusion of the results from this suggests that we can say that the GNN will leverage the underlying algorithmic structure of the subgroups and use this to effectively learn the elements of the Cayley table. This is extremely useful as we are able to achieve performance of above 40% accuracy for a 6% training split, which is significantly better than the Transformer's performance [13]. An additional note is that the error bars appear tighter for high training splits like at 30%. This shows us how sensitive deep learning models can be to training structure. The next steps for this experiment would be to analyse the broader applicability of these findings and how it works across domains. An attempt at S_5 was tried but was unsuccessful mainly owing to limitations of GPU hours. See Appendix B.4.1 for details.

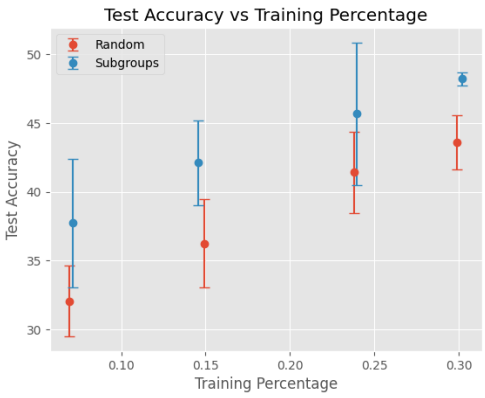

Figure 3: GNN results with S_4

5 Summary & Further Analysis

In summary, our experiments suggest that when training on subgroups, the Graph Convolutional Network (GCN) outperforms transformers on symmetric groups. However, our analysis remains preliminary, as we have yet to evaluate the performance of the graph-based network on other group types, such as those found in modular arithmetic. The discrepancy in performance suggests Transformers rely heavily on statistical generalization—capturing frequent patterns explicitly—whereas GNNs inherently exploit compositional generalization, allowing them to generalize learned rules about subgroup interactions to new contexts. This architectural difference highlights the advantage of explicitly compositional inductive biases in structured data scenarios
Additionally, both architectures exhibit unique strengths that merit further investigation. For instance, the GCN's superior performance on symmetric groups highlights its ability to capture structural properties effectively, whereas the Transformer's design might be better suited for tasks involving more nuanced or sequential relationships. In total, our findings underscore the importance of tailoring model architectures to the specific characteristics of the subgroup being modeled.
As a next step, we should extend our analysis to include a broader range of group types, refine our evaluation metrics, and conduct a more comprehensive comparison between Transformer-based models and GNNs to better understand their respective advantages and limitations in subgroup learning.

[11] Choosing a dropout of 0.5, patience of around 1000 iterations, weight decay of 0.03 and an initial lr of 0.01.
[12] This is sufficient compared with 50% for 2 conv layers, similarly 55% for 64 channels. With larger models taking too long to run
[13] Achieved around 30% at optimal

Appendix Content

A Appendix A : Data Construction

A.1 Definitions

We define the symmetric group S_n as the group consisting of all permutations of n distinct elements, with group operation given by permutation composition. Formally,

$$S_n = \{\sigma \mid \sigma \text{ is a bijection from } \{1, \ldots, n\} \text{ to itself}\}.$$

We define the modular arithmetic addition group $\mathbb{Z}_p^k$ as the direct product of k copies of the cyclic group of integers modulo p, i.e.,

$$\mathbb{Z}_p^k = \underbrace{\mathbb{Z}_p \times \mathbb{Z}_p \times \cdots \times \mathbb{Z}_p}_{k \text{ times}},$$

where each component has group operation defined as addition modulo p.

A.1.1 Subgroup Definition and Calculation

We define subgroups H as subsets of a group G that satisfy the group axioms under the operation inherited from G. Formally, a subset $H \subseteq G$ is a subgroup if for all $a, b \in H$, we have $ab^{-1} \in H$.

To identify and calculate subgroups, we:

- Fix a generating set and compute the generated elements,
- Verify closure under group operation and inverses,
- Ensure identity element inclusion.

A.2 Summary of Representation Types

We summarize four distinct representation methods for symmetric groups (S_n), using the symmetric group S_3 as a running example. The permutations in S_3 explicitly map positions to elements, for instance, the permutation $(1, 3, 2)$ means element 1 maps to 1, element 2 maps to 3, and element 3 maps to 2.

The permutations of S_3 in lexicographic order are:

$$\{(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)\}$$

The Cayley table for the group operation (composition of permutations) on S_3 is:

	(1,2,3)	**(1,3,2)**	**(2,1,3)**	**(2,3,1)**	**(3,1,2)**	**(3,2,1)**
(1,2,3)	(1,2,3)	(1,3,2)	(2,1,3)	(2,3,1)	(3,1,2)	(3,2,1)
(1,3,2)	(1,3,2)	(1,2,3)	(3,1,2)	(3,2,1)	(2,1,3)	(2,3,1)
(2,1,3)	(2,1,3)	(3,1,2)	(1,2,3)	(3,2,1)	(1,3,2)	(2,3,1)
(2,3,1)	(2,3,1)	(3,2,1)	(1,3,2)	(1,2,3)	(2,3,1)	(2,1,3)
(3,1,2)	(3,1,2)	(2,1,3)	(2,3,1)	(3,2,1)	(1,2,3)	(1,3,2)
(3,2,1)	(3,2,1)	(2,3,1)	(2,1,3)	(1,3,2)	(1,2,3)	(3,2,1)

Figure 4: Cayley Table for Permutation Composition

A.2.1 Adjacency Matrix Representation

The adjacency matrix explicitly represents the composition of permutations as a table where each cell corresponds to the operation result.
The Cayley table shown above itself serves as an adjacency matrix for S_3, taken without the headers.

A.2.2 Positional Encoding Representation

Positional encoding explicitly encodes position-element mappings within a permutation.
Example with permutation $(3, 1, 2)$:

$$[(1, 3), (2, 1), (3, 2)]$$

We would then input two of these lists into out models and look for the output in this form.

A.2.3 Direct Element Pair Representation

Direct concatenation encodes two permutations explicitly as input pairs.
Example:

- Input permutations: $(1, 2, 3)$ and $(2, 3, 1)$
- Concatenated input: $[1, 2, 3, 2, 3, 1]$
- Output permutation (composition): $[3, 1, 2]$

A.2.4 Graph-based Representation

Graph representation translates permutations into graph structures:
Example with permutations $p = (1, 2, 3)$ and $q = (2, 3, 1)$:

- Nodes: elements $\{1, 2, 3\}$, encoded with one-hot vectors.
- Edges for permutation p: $(1 \to 1, 2 \to 2, 3 \to 3)$.
- Edges for permutation q: $(1 \to 2, 2 \to 3, 3 \to 1)$.
- Edge types differentiate between input permutations.

This representation is particularly beneficial for graph neural network models due to its explicit relational structure.
Backlink: Representation design choices

B Results Appendix

B.1 Extra experiments with Z

After doing a through grid search through learning rate, weight decay, batch size, dropout, the following results were found. Note we kept the Transformer extremely small and did not change at all for these experiments as in each case we have tested that the transformer is capable of learning the operation [by testing on 90% training set] and making sure this learns effectively.

```
model = Transformer(embedding_dim=64, number_heads=4, num_layers=2, hidden_dim=64)
```

B.1.1 $\mathbf{Z}_7^2$

Considering all the subgroups of this as our training set, we end up with a 14% training split. For random sampling at this percentage, we are able to achieve at least 15% accuracy on the test set expectedly due to the symmetry of the operator since this is an Abelian group. This can be supported by manual analysis of the sets. However, for the subgroups, we are able to learn effectively on the training set, achieving $\sim 100\%$ even after using simple methods like increasing the dropout rates and weight decay as explained in the original Grokking paper, we are unable to achieve above 1% accuracy on the test set. A hypothesis as to why this occurs is that the model is struggling to extrapolate outside of this limited set it has been given. We also tried ...

B.1.2 $\mathbf{Z}_3^3$

In this case we have a much richer model in terms of the training statistics. In this case for all the subgroups we end up using around 50% of the data points. As we know from section E, we would expect a random sample to get to 100% easily. [This has been verified]. However, after learning on only subgroups we only get an average performance of 80% across epochs, receiving [92, 65, 80] the first 3 times I ran this. This further supports evidence that subgroups hinder training.

B.2 Ranked Experiments with S_4

B.2.1 How to form ranks

Top 3 ranked subgroups	Frequency
$(1,2,4,3)$, $(1,3,2,4)$, $(1,4,3,2)$, $(1,4,2,3)$, $(1,3,4,2)$, $(1,2,3,4)$	11
$(4,1,3,2)$, $(2,4,3,1)$, $(1,2,3,4)$	11
$(1,4,3,2)$, $(3,2,1,4)$, $(3,4,1,2)$, $(1,2,3,4)$	9

Table 2: Occurrences of each subgroup.

These were generated by training the model on random selections of subgroups and incrementing the score of each subgroup based on its occurrence in the runs that converge on a solution above 90% accuracy. This was run for over 100 iterations and we get the table above as the head of that table and we use the first 2 subgroups from this for our further experiments.
We will be using the fact 6% is equivalent to training the model on S_4 with only the top rank subgroup as the training set and 11% is equivalent to the top 2 subgroups.

B.2.2 Raw results

From the results in table 3, where we can see the experiments repeated twice out of the ten total. From these results, we can see a slight increase in performance of the sub-grouped model and can hypothesise this will be slightly better however, we can trivially see that this will not be statistically significant.

B.2.3 T test setup and results

For completion, we perform independent t-tests on these results to see if we can notice any significance in these results. The t-test yields a t-statistic of 0.338 and p-value of 0.7408. Since the p-value is much larger than any significance level we can say that there is no statistical difference between the accuracy scores of random sampling and highest rank subgroup sampling. This is assuming the accuracy score is noisy enough to fit a normal distribution and the variances

Method	Training Split (%)	Test Accuracy (%)
Subgroups	6	22.407
	6	22.870
	11	25.588
	11	25.488
Random	6	23.985
	6	25.461
	11	23.930
	11	24.708

Table 3: Initial Results across methods

of both sampling methods are roughly equivalent, the second of which may not hold as we scale but is assumed for this exercise.
The performance variation under subgroup-based sampling may reflect theoretical insights from group theory: certain subgroups generate richer, more generalizable rules, enabling better compositional generalization when selected appropriately. Our experiments suggest that subgroup richness (e.g., dimensionality and generator choice) directly impacts the compositional quality of learned representations.

B.3 GNN Results

B.3.1 Experiment with S_4

These experiments show that random subgroup choices tend to perform better than random splits of the table when considering GCNs and keeping the percentage of training data consistent. Here we will outline how these results were acquired.

B.4 Raw results

Method	Training Split (%)	Test Accuracy (%)
Subgroups	7	37.74 ± 4.65
	15	42.11 ± 3.09
	24	45.66 ± 5.17
	30	48.20 ± 0.48
Random	7	32.07 ± 2.56
	15	36.26 ± 3.18
	24	41.43 ± 2.95
	30	43.59 ± 1.95

Table 4: Results accuracy $\pm$ standard deviation for Different Training Splits

These results represent the results of training the GCN in low-resource environments. This was generated by choosing arbitrary subgroups and repeating at least 5 times to get the test accuracy with a standard deviation. More repeats were made for experiments with high standard deviation like the training split of 24%.
Then using these percentage splits we chose random splits. In this case we re-tuned some of the hyper-parameters like patience due to early convergence occurring occasionally as well as learning rate.

Additionally, we attempted this with many different hyper-parameters settings, keeping results from which the validation set converged as well as the training set as we noticed for many experiments the validation appeared to converge before the training had. Additionally, in some experiments we used the loss function paired with the validation accuracy in order to achieve a better notion of convergence and this achieved better performance especially for lower percentage splits.

We performed an independent t-test combining results across all training splits, obtaining a t-statistic of 1.91 with a p-value of 0.064. At a 10% significance level, we therefore reject the null hypothesis of equivalence, providing statistical evidence that subgroup-based sampling methods yield significantly different performance compared to random sampling. The independent t-test performed (Welch's variant) assumes the independence and approximate normality of accuracy measurements but does not require equal variances between groups, which aligns with the observed data characteristics in our experiments.

Note: We combined all simulated accuracy samples into two larger groups—one for subgroup sampling and one for random sampling—and carried out an independent Welch's t-test ($scipy.stats.ttest\_indwithequal\_var = False$) to evaluate if the observed difference was statistically significant.

B.4.1 Experiment with S_5

With promising results for S_4 let us see how this generalises to S_5. We are not able to do as extensive an evaluation as S_4 due to limitations of GPU hours, but we can take a training split as a data point and evaluate this. [14]

Since S_5 has a more subgroup-rich structure, we should be able to take lower training percentages than S_4 and still expect good performance. In this case, I have chosen 14% training sets. I initially ran this and got 32% for random sampling and 25% for subgroup sampling, however, upon further analysis for these results, training had not converged even though we had finished the max_epochs of 7000.

[14] We are not able to do as extensive hyper-parameter tuning or as many repeats

C Graph Neural Network

Here I am outlining all my design decisions and the reasoning behind them here along with pseduocode derived from my notebook.

C.1 Input format

C.1.1 Pseudocode

Input dataset is the symmetricdataset(n) defined

Algorithm 1 create_graph_dataset(tensor_dataset, n)

Input: tensor_dataset, integer n
Output: graph_data_list
graph_data_list $\leftarrow$ empty list
for each (x, y) in tensor_dataset **do**
    $p, q \leftarrow x$; p_edge_src, p_edge_tgt $\leftarrow$ empty lists
    for $i \leftarrow 0$ to $n-1$ **do**
        Append i to p_edge_src
        Append $(p[i]-1)$ to p_edge_tgt
    end for
    q_edge_src, q_edge_tgt $\leftarrow$ empty list
    for $i \leftarrow 0$ to $n-1$ **do**
        Append i to q_edge_src
        Append $(q[i]-1)$ to q_edge_tgt
    end for
    edge_index $\leftarrow$ Tensor([p_edge_src $\|$ q_edge_src, p_edge_tgt $\|$ q_edge_tgt])
    edge_type $\leftarrow$ Tensor(0,0,. . . ,1, 1)
    x_features $\leftarrow$ Identity matrix of size $n \times n$
    y_target $\leftarrow y-1$
    data_obj $\leftarrow$ torch.Data(x = x_features, $edge\_index$ = edge_index, $edge\_type$ = edge_type, y = y_target)
    Append data_obj to graph_data_list
end for
return graph_data_list

C.1.2 Explanation

For this network, a graph is required as input, which consists of node features and connectivity information. For clarity, we assume 1-based indexing and illustrate the process with an example.
Suppose we are composing two permutations, $p = [3, 1, 4, 2]$ and $q = [2, 4, 1, 3]$. The operation of interest is the composition of p with q in S_4.
For each permutation, directed edges are created such that the source node corresponds to the position in the permutation and the target node corresponds to the value at that position. For instance, in permutation p:

p[1] = 3 implies an edge from node 1 to node 3, p[2] = 1

Complete edges for p is: (1, 3), (2, 1), (3, 4), (4, 2). For q, they are: (1, 2), (2, 4), (3, 1), (4, 3).

After constructing the edges for each permutation they are merged into a single structure : Edge Index Tensor. A 2-row tensor that specifies the connectivity of the graph. The first row contains the source nodes for each edge and rhe second row contains the corresponding target nodes.
We then annotate each edge with an edge type to indicate its origin from the respective permutation graph:
Next, node features are created. Each node is assigned a feature vector that describes it. One-hot encoding is used, meaning node 1 is represented as [1, 0, 0, 0], node 2 as [0, 1, 0, 0], and so on. This encoding aids the network in identifying the nodes, while subsequent layers can learn more complex representations based on connectivity.
Finally, the target graph is prepared, which would be $[1, 2, 3, 4]$ from p and q

C.2 Choice of type of GNN

There are many different types of GNNs that we could have chosen for this task. In particular for this we chose to use a Graph Convolution Network. This is a type of Graph Neural Netowrk that works by letting each node in a graph update its own feature representation by gathering information from its immediate neighbours.

The main operations are:

1. Aggregation: Collect feature information from neighbouring nodes in the graph
2. Transformation: Aggregated features are combined using a set of learned weights. Along with a non-linear activation function

This is then repeated over several layers allowing each node to add information from a wider part of the graph.

This type of GNN has a lower complexity compared to other models and is effective for learning local structures which is why it was chosen for this task.

C.2.1 Pseudocode

Algorithm 2 RGCNModel: Relational Graph Convolutional Network

Input: n, hidden_channels, dropout, in_channels = n, out_channels = n, num_relations = 2
Output: Predicted node embeddings
Define **class** RGCNModel(nn.Module):
    Initialize:
        conv1 ← RGCNConv(in_channels, hidden_channels, num_relations)
        bn1 ← BatchNorm1d(hidden_channels)
        conv2 ← RGCNConv(hidden_channels, hidden_channels, num_relations)
        bn2 ← BatchNorm1d(hidden_channels)
        conv3 ← RGCNConv(hidden_channels, hidden_channels, num_relations)
        fc_dropout ← Dropout(p=dropout)
        fc ← Linear(hidden_channels, out_channels)
        self.dropout ← dropout
    Define forward function:
        Extract (x, $edge\_index$, $edge\_type$) from input data
        x ← bn1(conv1(x, edge_index, edge_type))
        x ← ReLU(x)
        x ← Dropout(x)
        x ← bn2(conv2(x, edge_index, edge_type))
        x ← conv3(x, edge_index, edge_type)
        x ← ReLU(x)
        x ← Dropout(x)
        x ← fc_dropout(x)
        x ← fc(x)
        return x

Main Hyperparameters & Parameters

1. Number of relations is trivially 2 since we have 2 input graphs
2. Number of convolutional layers: 3 (`RGCNConv`)
3. Hidden channels: $hidden\_channels$
4. Learning rate, max_epochs, weight decay, dropout
5. Learning rate scheduler and patience:
    - Scheduler: `ReduceLROnPlateau`
    - Patience: 100 epochs for the scheduler, 700 for early stopping

Algorithm 3 Train and Evaluate RGCN Model

(Hyper)Parameters n, b_size ← 32, num_relations ← 2, in_chan, out_chan ← *n*, hid_chan ← 128, dropout
num_epochs ← 5000, patience ← 700, wd, lr
train_dataset, val_dataset, test_dataset ← create_graph_dataset(dataset, n) ▷ Create datasets
Initialize train_loader, val_loder, test_loader
model ← RGCNModel(n, in_chan, hid_chan, out_chan, num_relations, dropout) ▷ Initialize the model
criterion ← CrossEntropyLoss()
optimizer ← AdamW(model.parameters(), lr, wd) ▷ Define loss and optimizer
scheduler ← ReduceLROnPlateau(optimizer, mode='min', factor, patience=100)

Initialize: best_val_loss, best_val_acc, patience_counter, best_model_state
for *epoch* ← 0 to num_epochs **do** ▷ Training loop
    model.train()
    train_loss ← 0.0
    for each data in train_loader **do**
        optimizer.zero_grad()
        out ← model(data)
        y ← data.y.view(-1)
        loss ← criterion(out, y)
        loss.backward()
        optimizer.step()
        train_loss ← train_loss + loss.item()
    end for
    avg_train_loss ← train_loss / len(train_loader)
    val_loss, val_acc, _ ← evaluate(model, val_loader, criterion) ▷ Validation step
    scheduler.step(val_loss)

    if val_loss < best_val_loss **then** ▷ Early stopping and checkpointing
        best_val_loss ← val_loss ; Reset patience_counter
    else
        patience_counter ++
    end if
    if val_acc > best_val_acc **then** best_val_acc ← val_acc; Save best_model_state
    end if
    if patience_counter ≥ patience **then** break
    end if
end for

model.load_state_dict(best_model_state) ▷ Load the best model state

test_loss, test_acc, test_preds, test_labels ← evaluate(model, test_loader, criterion) ▷ Evaluation

D Transformer Implementation

D.1 Transformer Code

Algorithm 4 Encoder Only Transformer Model

Input: *input_len*, *output_dim*, *dropout*, *vocab_size*, *embed_dim*, *num_heads*, *hidden_dim*, *num_layers*
Output: Transformed token representations
Define **class** Transformer(nn.Module):
    Initialize:
        embed ← Embedding(vocab_size, embed_dim)
        encoder_layer ← TransformerEncoderLayer(**kwargs)
        encoder ← TransformerEncoder(encoder_layer, num_layers)
        fc ← Linear(embed_dim, vocab_size)
        positional_encoding ← Learnable Parameter of shape (input_len, embed_dim)
    Define forward function:
        Input: x of shape (batch_size, $2n$)
        $x \leftarrow$ embed(x)
        $x \leftarrow x + positional\_encoding$ (optional)
        $x \leftarrow$ permute(x, (1, 0, 2)) ▷ Switch dimensions around for Encoder
        $x \leftarrow$ encoder(x)
        $x \leftarrow$ permute(x, (1, 0, 2))
        $n \leftarrow x.shape[1]//2$
        $x \leftarrow$ view(x, (batch_size, n, 2, embed_dim))
        $x \leftarrow$ mean(x, dim=2)
        $x \leftarrow$ fc(x)
        return x

D.2 Invocation / Training Loop

Algorithm 5 Training a Transformer Model on Dataset

$input\_len \leftarrow 2k/2n$, $output\_dim \leftarrow k/n$, $vocab\_size \leftarrow p$
dataset, sorted_subgroups, map_to_int, map_to_tokens ← repeated_cyclic_modulo_dataset_subgroups_flat(p, k)
total_samples ← len(dataset)
train_dataset, val_dataset, test_dataset ← split(dataset, [train_size, val_size, test_size])
Initialize train_loader, test_loader, val_loader
model ← Transformer(**kwargs)
criterion ← CrossEntropyLoss()
optimizer ← AdamW(model.parameters(), lr, wd) ▷ Initialize Model, Loss, and Optimizer

for *epoch* ← 0 to num_epochs **do** ▷ Training Loop
    model.train()
    train_loss ← 0
    for batch_inputs, batch_targets in train_loader **do**
        optimizer.zero_grad()
        logits ← model(batch_inputs)
        logits ← reshape(logits, -1, vocab_size)
        batch_targets ← reshape(batch_targets, -1)
        loss ← criterion(logits, batch_targets)
        loss.backward()
        optimizer.step()
        train_loss ← train_loss + loss.item()
        clip_gradients(model)
    end for
    avg_train_loss ← train_loss / len(train_loader)
    ▷ Validation Step
    avg_val_loss, val_acc ← evaluate(model, val_loader, criterion)
end for

E Dataset pseudocode

E.1 Generating Z_p^n

E.1.1 Generating group

Algorithm 6 Repeated Cyclic Modulo Dataset for $\mathbb{Z}_p^k$

function repeated_cyclic_modulo_dataset_subgroups_flat(p, k) ▷ Generate dataset for $\mathbb{Z}_p^k$ with cyclic addition

    alphabet ← torch.arange(p) ▷ Create the alphabet for $\mathbb{Z}_p$
    elements ← cartesian product of k copies of alphabet ▷ Generate all group elements in $\mathbb{Z}_p^k$
    group_elements ← cartesian product of k copies of alphabet
    if $k = 1$ **then**
        group_elements ← group_elements.unsqueeze(1)
    end if
    n ← number of elements in group_elements ▷ Generate input pairs
    indices ← torch.arange(n)
    prod_indices ← cartesian product of indices
    X_1 ← group_elements[prod_indices[:, 0]]
    X_2 ← group_elements[prod_indices[:, 1]]
    X ← concatenate(X_1, X_2 along dim=1)
    $Y \leftarrow (X_1 + X_2) \mod p$
    tokens ← random permutation of p ▷ Create a mapping from integers to tokens
    map_to_tokens ← i: tokens[i] for i in range(p)
    map_to_int ← tokens[i]: i for i in range(p)

    X ← tensor of mapped token values from original X ▷ Convert dataset to tokenized format
    Y ← tensor of mapped token values from original Y
    dataset ← TensorDataset(X, Y)
    return dataset, sorted_subgroups, map_to_tokens, map_to_int
end function

E.1.2 Generating the subgroup

Algorithm 7 Subgroup Generation

```
function closure_of_generators(generators, m, k)        ▷ Compute the subgroup generated by given generators Z_m^k
    identity ← tuple of zeros
    subgroup ← set(generators) ∪ {identity}
    changed ← True
    while changed do
        changed ← False
        new_elements ← empty set
        for a in subgroup do
            for b in subgroup do
                s ← tuple(a[i] + b[i]  mod m for i in 0, ..., k − 1)
                if s ∉ subgroup then
                    new_elements ← new_elements ∪{s}
                end if
            end for
        end for
        if new_elements ≠ ∅ then
            subgroup ← subgroup ∪ new_elements
            changed ← True
        end if
    end while
    return sorted(subgroup)
end function
function generate_all_subgroups_nonprime(m, k)          ▷ Generate all subgroups of Z_m^k for non-prime m
    all_elements ← all k-tuples with entries in {0, ..., m − 1}
    identity ← tuple of k zeros
    group_order ← m^k
    seen ← empty set
    trivial ← frozenset({identity})                     ▷ Yield the trivial subgroup
    seen ← seen ∪ {trivial}
    yield trivial
    nonzero ← all_elements \ {identity}
    for d ← 1 to k do
        for each generators in combinations(nonzero, d) do
            subgroup ← closure_of_generators(generators, m, k)
            if |subgroup| = group_order then
                continue                                ▷ Skip if subgroup is the full group
            end if
            if subgroup ∉ seen then
                seen ← seen ∪ {subgroup}
                yield subgroup                          ▷ Yield so that we are generating on the fly
            end if
        end for
    end for
end function
```

E.2 Generating S_n

E.2.1 Generating the group

Algorithm 8 Generate Dataset for Symmetric Group S_n

function symmetric_dataset(n) ▷ Generate dataset for S_n, representing the Cayley table
    elements ← {1, 2, ..., n} ▷ Generate all permutations of $\{1, 2, ..., n\}$
    permutations ← all permutations of elements
    $X\_alphabet$ ← [], $Y\_alphabet$ ← []
    for perm_1 in permutations **do**
        for perm_2 in permutations **do**
            perm_out ← apply perm_2 to perm_1 to elements
            Append $[\text{perm}_1, \text{perm}_2]$ to $X\_alphabet$
            Append [perm_out] to $Y\_alphabet$
        end for
    end for
    X ← torch.tensor($X\_alphabet$)
    Y ← torch.tensor($Y\_alphabet$)

    perm ← random permutation of $\{1, ..., n\}$
    map_to_tokens ← {i + 1: perm[i] for i in range(n)}
    map_to_int ← {perm[i]: i + 1 for i in range(n)}
    X ← apply map_to_tokens to X ▷ Map dataset to tokenized format
    Y ← apply map_to_tokens to Y
    dataset ← TensorDataset(X, Y)
    return dataset, map_to_tokens, map_to_int
end function

E.2.2 Generating subgroups

Algorithm 9 Subgroup Closure and Enumeration

```
function closure_from_generators(generators, comp_table, cache)                ▷ Compute subgroup closure
    key ← frozenset(generators)
    if key in cache then return cache[key]
    end if
    closure ← set(generators), changed ← True
    while changed do
        new_elements ← {comp_table[a][b] for a, b in closure}
        new_closure ← closure ∪ new_elements
        if new_closure = closure then
            changed ← False
        else
            closure ← new_closure
        end if
    end while
    cache[key] ← frozenset(closure)
    return cache[key]
end function
function enumerate_subgroups(generators, start, group_elements, comp_table, cache, seen, result)    ▷ Recursive subgroup enumeration
    H ← closure_from_generators(generators ∪ {0}, comp_table, cache)
    if H ∈ seen then return
    end if
    seen, result ← seen ∪ {H}, result ∪ {H}
    for i ← start to len(group_elements) do
        g ← group_elements[i]
        if g ∉ H then
            enumerate_subgroups(generators ∪ {g}, i + 1, group_elements, comp_table, cache, seen, result)
        end if
    end for
end function
function all_subgroups_ordered(perms, comp_table)
    enumerate_subgroups((), 0, range(1, len(perms)), comp_table, cache, seen, result)
    return result
end function
```

E.3 Split Function

Algorithm 10 Dataset Splitting Based on Subgroups or Random Sampling

```
perms, mapping ← symmetric_group(n)
comp_table ← precompute_composition_table(perms, mapping)
subgroups ← all_subgroups_ordered(perms, comp_table)
formatted_subgroups ← [
      convert_subgroup_indices_to_permutations(sg, perms)
      for sg in sorted(subgroups) ]
subgroups, potentials ← split_symmetric_subgroup_(formatted_subgroups)
train_dataset, test ← split_cayley_dataset_by_subgroups(dataset, subgroups, [0,1])
if len(test) = 0 or len(train_dataset) = 0 then
    continue
else
    flag ← False
    break
end if
val_dataset, test_dataset ← random_split(test, [len(test)//2, len(test) - len(test)//2])
```