Combinatorial Computations Regarding Discrete Symmetries

Arman Shamsgovara
arman.shamsgovara@minervaskolan.se

under the direction of
Ken Åman
Minerva Gymnasium

Daniel Andrén
Lars Hellström
Department of Mathematics and Mathematical Statistics
Umeå University

Minerva Gymnasium
April 15, 2013
Abstract

In this paper the role of group theory within enumerative combinatorics was studied, with focus being on coloring problems with respect to symmetry. The objectives of this paper were to understand the role of group theory when solving coloring problems with symmetries involved, to develop computer programs that can solve these problems, and to find new areas of study within the subject. Algorithms that utilize Burnside's lemma and the famous Polya Enumeration Theorem (PET) were successfully developed and implemented in Python 3.2, to enable fast computation of the number of colorings of objects, with respect to symmetry. The programs were then run several times to compile a library of cycle indices of symmetry groups, and to solve example problems stated within the paper. A new area of study in the form of nested necklaces was introduced and began to be studied. It became evident that the PET is a very powerful theorem that completely solves coloring problems with respect to symmetry, and that Burnside's lemma is also very powerful, but less so than the former, since it solves one coloring case at a time, while the PET solves all cases at once.

Keywords: Cycle index, Group theory, Combinatorics, Colorings, Polya Enumeration, Burnside’s Lemma, Symmetry
## Contents

1 Introduction 2

2 Theory - A Group Theoretic Approach to Symmetries 4
   2.1 The Relation Between Symmetries and Groups 4
   2.2 Some Basics of Group Theory 10
   2.3 Groups and Permutations 15
   2.4 Symmetries Within Combinatorics 21
      2.4.1 Colorings of a Symmetric Object by Group Theory 21
      2.4.2 Generating Functions and Symmetry 27
   2.5 Symmetry in More than Two Dimensions 36

3 Designing the Counting Algorithms 38

4 Some Exotic Symmetry Groups 41
   4.1 Nested Necklaces 41
   4.2 Chemical Molecules 44

5 Results 47

6 Discussion 50
   6.1 Future Studies 51

7 Acknowledgements 52

8 Notations 53

A PYTHON-code using Burnside’s lemma 55

B PYTHON-code using Cycle Indices to Compute Colorings 59

C Group_funcs.py 66

D Polynomial.py 69

E Example Files Containing Groups of Symmetry 77

F Basic Elementary Combinatorics 78
1 Introduction

Throughout the years, mathematicians have utilized the idea of symmetry to simplify problems, describe phenomena and discover surprising new knowledge regarding many areas of mathematics. The most prominently known type of symmetry, and for many people the only one known, is the geometric one, with the frequently appearing symmetry line being the only thing focused upon. However, the world of symmetries is much vaster than that. Especially since the mathematical area of *Group Theory* appeared, symmetries have been extensively studied, and not just been seen as peculiar properties of shapes.

Today, most mathematical research regarding symmetries wants to break symmetries in problems. This is because symmetries act as constraints to problems, and can prove to be problematic to get past when obtaining solutions. Coloring problems, in which one wants to find the number of ways to color an object with a given amount of colors, can get especially hard to solve, since the object being colored could be a cube, a molecule, a regular 32-gon, or anything else that has some type of symmetry properties.

Solving coloring problems for asymmetrical objects is easy if one uses the binomial and multinomial theorems from elementary combinatorics, but when coloring a cube, one has to pay respect to the fact that the cube can be rotated. The problem of coloring symmetric objects has been solved by two notable theorems within group theory and the field of generating functions. The theorems in question, Burnside’s lemma and the Polya enumeration theorem (PET), provide efficient methods of solving coloring problems, but require some computation, which is why computer programs would be good to have when solving such problems.

The main objectives of this paper are to understand the role of group theory within combinatorics, to implement Python programs that can use group theory to count with the mentioned theorems, and to introduce and study some symmetry groups, both common
and new. A leading question throughout the paper regards the possibility to implement
group theoretic computer programs to solve coloring problems with Burnside’s lemma and
the PET. In this paper, we restrict ourselves to studying discrete symmetries on finite sets,
since they can actually be handled by a computer, and since they do not need as much
background reading as the continuous symmetries do.

The principle of multiplication, the binomial theorem and the multinomial theorem from
elementary combinatorics are presumed preliminaries of this paper, but if the reader feels
unsure on these topics, I have prepared an appendix regarding those theorems at the back
of the paper.
2 Theory - A Group Theoretic Approach to Symmetries

2.1 The Relation Between Symmetries and Groups

To define a symmetry formally is a task too hard for many a savant, mostly because of the so-thought simplicity and lack of general knowledge of the concept. To begin this extensive theory section, we define this concept that the whole project revolves around, the idea of a symmetry:

Definition 2.1. A symmetry of a set $X$ is defined as an operation $f : X \rightarrow X$ such that $\forall x \in X : x \in \{f(x)|x \in X\} \land f(x) \in X$, that is, $f$ is one-one and onto, a bijection from $X$ to itself. The operation $f$ must also be such that some certain property holds for all possible values of $f(x)$ when they hold for $x$.

![Figure 1: The symmetry lines of a square and a regular pentagon](image1.png)

The property mentioned within the definition could be any possible. As an example, a symmetry of a square would be a bijection from the set of its corners onto itself such that the corners in question always have the same two adjacent corners next to them. We cannot for example swap one pair of corners if they’re not diagonal opposites, since that would be an illegal action (imagine making that move with a paper square, it would not account for a reasonable symmetry). There are however eight operations that hold the property needed to be valid even for a paper square: We can perform up to 3 $90^\circ$ rotations, flip it diagonally or axially in 4 ways, and we could also leave the square as is (since that would trivially meet
the desired property, hence it is a trivial symmetry). Figure 1 (the left image) illustrates the four symmetry lines revealing the flip symmetries, and the rotations are easy to imagine.

![Figure 1: Illustration of four symmetry lines revealing flip symmetries](image)

Figure 2: Left: $\mathbb{Z}^2$, Right: $\mathbb{Z}^+ \times \mathbb{Z}$

**Example 2.1.** Consider the set $\mathbb{Z}^2$, the lattice consisting of all possible tuples $(x,y)$ of integer coordinates $x$ and $y$. Let $f(x,y) = (-x,y)$. It is rather straightforward to prove that $f$ is both one-to-one and onto $\mathbb{Z}^2$, thus $f$ is a bijection, and also a symmetry on $\mathbb{Z}^2$. Now, let the set considered be $\mathbb{Z}^+ \times \mathbb{Z}$. The operation $f$ is not a symmetry on the given set, since all the positive $x$-coordinates will be mapped into negative $x$-coordinates, that lie out of the set (compare with figure 2). Thus, $f$ is no bijection on $\mathbb{Z}^+ \times \mathbb{Z}$, and hence no symmetry of the set.

Now that we have established an idea of what symmetry is, we will continue by defining some of the symmetries’ properties, but first we have to define some necessary topics:

**Definition 2.2.** For two operations (or functions) $f$ and $g$ and some element $x$ in an arbitrary set, $(f \circ g)(x) = f(g(x))$. The $\circ$ is called the composition operator.

**Example 2.2.** Let $f(x) = x^2$, $g(x) = \sin(x)$. Then:

$$(f \circ g)(x) = f(g(x)) = \sin^2(x)$$

$$(g \circ f)(x) = g(f(x)) = \sin(x^2)$$
Definition 2.3. The **identity operation**, here denoted \( \iota \), is a symmetry operation such that for all symmetries \( s \):

\[
s \circ \iota = \iota \circ s = s,
\]

regardless of \( s \).

The identity operation is often called the **trivial symmetry**, since it is always a symmetry, for any object. It may seem a bit awkward to include this as a symmetry, but it is necessary for the following to work, and will not affect anything negatively. There is only one identity operation for any object, since leaving something as is can only be done in one way.

**Definition 2.4.** For an arbitrary symmetry operation \( s \), we denote it’s **inverse** by \( s^{-1} \), and the inverse is such that \( s \circ s^{-1} = s^{-1} \circ s = \iota \). We also say that all symmetries have an inverse operation.

It is easy to understand the latter part of the last definition, because if we for example flip a square about one of it’s main diagonals, we should be able to flip it back about the same diagonal. The same type of reasoning goes for all other symmetries. Another way to see this is to think that if all symmetries are bijections from a set to itself, that is all elements in the set gets a unique image, then there should be a bijection that goes the other way, from the images to their "pre-images" in the set.

**Theorem 2.1.** For three symmetry operations \( f, g \) and \( h \) of a common set, the operation \( \circ \) is associative, that is, \( f \circ (g \circ h) = (f \circ g) \circ h \).

**Proof** - Since \( f \) and \( h \) are symmetries, their inverses \( f^{-1} \) and \( h^{-1} \) must exist. Apply \( f^{-1} \) on the left of both sides of the equation, and \( h^{-1} \) on the right of both sides of the equation. The left side of the equation then becomes

\[
f^{-1} \circ f \circ (g \circ h) \circ h^{-1} = \iota \circ (g \circ h) \circ h^{-1} = g \circ h \circ h^{-1} = g,
\]
and the right-hand side becomes

\[ f^{-1} \circ (f \circ g) \circ h \circ h^{-1} = f^{-1} \circ (f \circ g) \circ \iota = f^{-1} \circ f \circ g = g. \]

Both sides are equal, thus the statement is proven. □

To summarize, a given symmetry \( s \) has to have an inverse, there has to be an identity symmetry, and all symmetries are associative under the composition operator. But those are the exact requirements that must hold for any group:

**Definition 2.5.** We define a **binary operator** to be an operator such that it maps two objects onto a third (but not necessarily different) one.

**Example 2.3.** The arithmetic operators \( +, -, \cdot \) and \( / \) are all binary, since they take in two numbers, and return a number as a result. For example, if we set \( f(x, y) = x + y \), then \( f \) is actually the binary operator \( + \) in disguise.

**Example 2.4.** The operation for function composition, \( \circ \) is also a binary operation, since it takes two mappings as input and returns their composition (which is a mapping) as output.

**Definition 2.6.** We define a **group** \(<S, *>\), to be a set \( S \) paired with the binary operation \(*\), such that the following three group axioms hold:

1. There exists an identity \( \iota \in S \), such that \( \forall x \in S : \iota \ast x = x \ast \iota = x \)
2. There exists an inverse for all elements in \( S \) such that \( \forall x \in S : x^{-1} \in S \quad \land \quad x \ast x^{-1} = x^{-1} \ast x = \iota \)
3. The operation \( * \) is associative, that is \( x_1 \ast (x_2 \ast x_3) = (x_1 \ast x_2) \ast x_3 \forall x_1, x_2, x_3 \in S \)

During the development of the modern algebra, groups were the earliest objects studied, and their theory is widely considered to be the portal to modern algebra. Group theory is a versatile tool that is utilized in many areas of mathematics, ranging from number theory
to combinatorics to geometry, and this paper will mainly focus on the group theory behind symmetries.

**Example 2.5.** Consider the set $\mathbb{Z}$, the set of integers, and the operation for addition, $\ + $. Does $\langle \mathbb{Z}, + \rangle$ form a valid group?

**Solution** - Well, we need to check for three things: associativity of the $+$ operator, an identity element, and inverses for all elements. The addition operator is associative, since $a + (b + c) = (a + b) + c$ for all integers $a, b, c$. Next, the integer 0 forms an identity element, since $a + 0 = 0 + a = a, \forall \in \mathbb{Z}$.

Finally, for any integer $a$, $-a$ forms the inverse, since $a + (-a) = (-a) + a = 0, \forall \in \mathbb{Z}$.

Thus all group axioms are fulfilled, and $\langle \mathbb{Z}, + \rangle$ is a group.

**Example 2.6.** Consider the set $\mathbb{R}$, the set of real numbers, and the operation for multiplication, $\cdot$. Does $\langle \mathbb{R}, \cdot \rangle$ form a valid group?

**Solution** - Again, we need to check for the three required axioms to hold. We know that multiplication is associative, and 1 obviously forms an identity element, but does there exist an inverse for all elements? For any real number $a$, $\frac{1}{a}$ has to be the inverse element, since $a \cdot \frac{1}{a} = \frac{1}{a} \cdot a = 1$, but if $a = 0$, then we have no inverse amongst the real numbers, so $\langle \mathbb{R}, \cdot \rangle$ is not a group. That being said, 0 is the only obstacle keeping the set of real numbers from forming a group under multiplication, so $\langle \mathbb{R} \setminus \{0\}, \cdot \rangle$ forms a group.

Compare our results with symmetries to the group axioms. We have an identity symmetry, all symmetries have inverses, and symmetry composition is associative. It seems as if we can use groups and group theory to work with symmetries, that is, we can let a set of symmetries form a group under the composition operator, and treat them as any other group! Suddenly we can use mathematics to compute on symmetries. We are now ready to define some of the
most common symmetries and some groups that they form. These are widely known, but often denoted with differing notations:

We let the following table show our denotion of some types of symmetry:

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\iota$</td>
<td>The identity</td>
</tr>
<tr>
<td>$r_n$</td>
<td>Rotation by $n$ steps</td>
</tr>
<tr>
<td>$m_n$</td>
<td>Mirroring about an axis with index $n$</td>
</tr>
<tr>
<td>$d_n$</td>
<td>Mirroring about a diagonal with index $n$</td>
</tr>
</tbody>
</table>

**Example 2.7.** A triangle has six symmetries; three reflections, two rotations and the identity. See fig. 3 for two of these symmetries.

![Figure 3: A regular triangle, and two of its symmetries](image-url)
2.2 Some Basics of Group Theory

Definition 2.7. The order of a group \( G \), denoted \(|G|\), is equal to the number of elements within the set of the group.

Definition 2.8. The dihedral group \( D_n \) contains the symmetries of a regular polygon with \( n \) vertices.

Example 2.8. The group \( D_4 \) contains the symmetries of the regular quadrilateral, the square. As shown before, it contains four reflections, three rotations and the identity, making a total of 8 symmetries (\(|D_4| = 8\)).

Before defining anything else, we will establish a certain property amongst the dihedral groups:

Theorem 2.2. \(|D_n| = 2n\), where \(|D_n|\) denotes the order of \( D_n \), or the number of elements in \( D_n \).

Proof - Assume a regular polygon with \( n \) vertices. It is bound to have \( n - 1 \) rotational symmetries and the identity symmetry, making a total of \( n \) symmetries. We have two cases now:

1. \( n \) is even. In this case, all diameters of the polygon go from one vertex to an opposite vertex, thus making a total of \( 1.5n \) symmetries (we do not count the same axis of reflection twice). The rest of the symmetries are axial, with any axis being placed on the middle of a side perpendicular to the axis. These are \( n/2 \) such axes in total, so all-in-all we have \( 2n \) symmetries.

2. \( n \) is odd. In this case, all the symmetry axes go from a vertex to the middle of a side opposite to the vertex in question. There are \( n \) of them in total, one for each vertex, thus, there are \( 2n \) symmetries in total, even in this case. \( \square \)
We now continue by defining a type of group that has a lot of applications within e.g. theoretic chemistry: cyclic groups.

Figure 4: The cycle structure of $\mathbb{Z}_{10}$

**Definition 2.9.** We define $\mathbb{Z}_n$ (also denoted $C_n$) to be the **cyclic group on $n$ letters**. The only symmetries in this group are $n$ rotations (including the identity as a rotation of zero steps).

**Remark 1.** Note that there is a very useful interpretation of $\mathbb{Z}_n$ using modular arithmetic. Study the set of all integers modulo $n$ under addition modulo $n$, that is \( \{0, 1, 2, 3, \ldots, n-1\} \) (the set of all possible remainders when dividing an integer by $n$). It is straightforward to see that the group axioms hold, so it is a valid group. What is interesting here is that addition by 0 equals the identity, and that $n$ rotations also equals the identity, since $n \equiv 0 \pmod{n}$. Addition with any integer $k$ should result in moving a given integer $m$ by $m + k \pmod{n}$ steps in the set, starting over on 0 whenever we reach $n$, and vice versa should we end up with a subtraction. This group behaves exactly like $\mathbb{Z}_n$, making the two viewpoints equivalent.

**Example 2.9.** In fig. 4, the cyclic structure of $\mathbb{Z}_{10}$ is showed. Whenever we add one to a number in this group, we take one step in the direction of the arrows. One can define groups that have this cycle structure, and replace the numbers with any kind of defined val-
ues/states/whatever required, but mathematicians prefer to study the kind of cycles involving numbers. We will be using cycles later on in this paper, when we study colorings of graphs.

Now that we have encountered groups that have cyclic structures, we shall define these and their properties properly.

One peculiar thing to notice in cycles is that you could start with one element, repeatedly use the binary operator on the element with itself, and encounter all elements within the cycle. For example, in $\mathbb{Z}_5$, you could start with the element $1$, and then use the binary operator for addition modulo 5 to obtain, in chronological order, $1, 2, 3, 4, 0, 1, 2, 3, 4, \ldots$. Notice how we have reached all the elements of the group, by just using one element. We say that we just generated $\mathbb{Z}_5$ by using the element $1$. Another way to say it is that $1$ is a generator of $\mathbb{Z}_5$, since it by itself can generate the whole group. In general, all groups can be generated from a single, but also from more elements in them. We summarize in a definition:

**Definition 2.10.** For a given group $G$ with operator $\ast$, we define a generating set of $G$ to be a subset $\{a, b, c, \ldots, n\}$ of elements in $G$, such that all elements of $G$ can be written in the form $a^{x_1}b^{x_2}\ldots n^{x_k}$, for integers $x_i$, where $p^q = p \ast p \ast p \ast \ldots \ast p$ ($q$ times). We then say that $G$ is generated by $\{a, b, c, \ldots, n\}$.

**Definition 2.11.** We define a generator $a$ of a cyclic group $G$ with operator $\ast$ to be an element in $G$ such that all elements in the group can be written in the form $a^n = a \ast a \ast \ldots \ast a$ ($n$ times), for integers $n$. We say that the element $a$ generates $G$, and $G = \{a^n | n \in \mathbb{Z}\}$.

**Remark 2.** Note that in this context, $a^n$ could mean any repeated $n-1$ uses of a binary operator on $a$. For example, "$a^3$" could mean "$a + a + a$", "$a \cdot a \cdot a$", "$a \cup a \cup a$", and so on. "$a^n$" is just a convenient way of writing such identities.

The idea of a generator is simple; you begin with some arbitrary element, and apply the binary operation of the group on it with the generator repeatedly, until you have obtained
all elements of the group. Groups with a cyclic structure need only one generator (in the example above, a generator would be 1), but for most groups, we talk about a generating set consisting of multiple generators that, together with the group operation generates all of the group.

Example 2.10. Find generating sets for:

a) $D_4$, b) $C_5$

Solution:

a) $D_4$ is the group of symmetries of the square. A square can be rotated, mirrored axially and mirrored diagonally. How do we generate this? Well, we must be able to mirror a square somehow, and to rotate it, so $r_1$ and $m_1$ are possible generating elements. From these two operations, we can generate all rotations, the identity, and both axial mirrorings (the latter by rotating one step with $r_1$ followed by use of $m_1$ and three more uses of $r_1$). Can we generate the diagonal mirrorings too? The answer is yes, since $m_1r_1 = d_1$, and $r_1m_1 = d_2$, so a generating set would be $\{r_1, m_1\}$.

b) $C_5$ is the cyclic group on 5 letters, and consists only of rotations and the identity, thus, we only need one generator of the set, and we can easily choose $r_1$ to be the generator.

Definition 2.12. We define a cyclic group $G$ to be a group such that some element $a \in G$ generates $G$.

The reader may have already seen that $C_n$ is completely contained within $D_n$, just like e.g. $\mathbb{Z} \subset \mathbb{R}$. The group theoretic analogy of subsets are called subgroups, and are defined as follows.

Definition 2.13. We define a subgroup $H$ of a group $G$ to be a subset of $G$ such that $H$ fulfills the group axioms and is closed under the binary operation of $G$.

It is easily verified that from the group axioms, $H$ must contain inverses for all elements within it, and that it must also contain the identity element of $G$. 

13
In this sense, $C_n$ really is a subgroup of $D_n$, since the set of the former is a subset of the latter, and $C_n$ is a closed group.

Subgroups will appear when we prove a certain theorem regarding the connection between permutations and groups.

We will also define something known as cosets of a group, and that will come in handy when proving a famous lemma that this paper revolves around.

**Definition 2.14.** For a subgroup $H$ of a group $G$, we define a coset of $G$ containing $a$ as $aH = \{ah | h \in H\}$ or $Ha = \{ha | h \in H\}$. Observe the difference between the two. The two cosets are distinguished by naming the former to be the left coset and the latter to be the right coset of $G$.

**Theorem 2.3. (Lagrange’s theorem)**

For any group $G$ with subgroup $H$, if we denote the number of cosets belonging to $H$ by $k$,

$$|G| = k|H|$$

We omit the proof. See [4] for a complete proof.

Lagrange’s theorem is a beautiful result of group theory and finds many uses, as it tells us a great deal of information about what possible subgroups a given group could have. We will not see much of this theorem in the remaining parts of the paper, as we will only use it in a proof, but it is a very handy property of finite groups to bear in mind.
2.3 Groups and Permutations

Before looking at where symmetries arise in combinatorics, we must establish a fundamental property of all groups, namely that all groups can be translated into a group of permutations. Permutations are very useful to represent group elements, especially when we want to implement algorithms that can handle group actions.

**Definition 2.15.** We define a permutation to be a bijection of a set onto itself.

The basic way of denoting a permutation is by creating a "2 x n matrix"-ish representation, where n is the number of elements to be permuted. In the top row, you put the integers 1 to n in that order, and in the bottom row, you put any permutation of the top row. It will look like this:

\[
\begin{pmatrix}
1 & 2 & 3 & 4 \\
2 & 4 & 3 & 1 \\
\end{pmatrix}
\]

What this means is that the permutation is a function that takes in an integer, looks for the number in the top row in the table, and returns the number directly below it. If we call the above permutation P, then P(1) = 2, P(2) = 4, P(3) = 3, and P(4) = 1.

It is obvious that a permutation of a set is both one-one and onto, thereby giving it the status of a bijection. Since a permutation can be viewed as a mapping, then we may define composition of permutations, or as most sources call it, multiplication of permutations.

**Definition 2.16.** For two permutations P and Q, we define the product PQ to be the permutation achieved when permuting all elements of P according to Q.

**Example 2.11.** Say we have two permutations

\[
P = \begin{pmatrix}
1 & 2 & 3 & 4 \\
2 & 4 & 3 & 1 \\
\end{pmatrix}, Q = \begin{pmatrix}
1 & 2 & 3 & 4 \\
3 & 1 & 4 & 2 \\
\end{pmatrix}.
\]
Then the product $PQ$ is found by permuting the bottom row of $P$ according to $Q$. So, looking at the bottom row of $P$ and inserting each element into $Q$, we replace the 1 with 3, the 2 with 1, the 3 with 4, and the 4 with 2, giving us the product

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

Figure 5: A permutation, illustrated in three ways:

a) Viewing the permutation as a bijection from a set to itself

b) The same permutation, but with the arrows sorted out

c) The cycles of this particular permutation

There is one more type of group that we must be familiar with in order to proceed with the paper, and that one is the group of all permutations of $n$ letters (or numbers). All three
of the group axioms are held in such a group:

1. There exists an identity permutation, namely

\[
\begin{pmatrix}
1 & 2 & 3 & 4 & \ldots & n \\
1 & 2 & 3 & 4 & \ldots & n
\end{pmatrix}
\]

2. All permutations have inverses (namely, we swap the two rows in the matrix-like representation and sort the columns according to their top elements. Then the mapping acts backwards, making it an inverse).

3. Since permutations can be viewed as functions, and it is proven that function composition is associative (Theorem 2.1), then permutation multiplication is also associative.

Therefore the set of all \( n! \) \( n \)-letter permutations forms a group under permutation multiplication. We shall denote that set by \( S_n \), or the symmetric group on \( n \) letters (all symmetries may be viewed as permutations, hence giving it the term symmetric in the name).

Now, what we will do before branching off into the world of combinatorics, we shall prove that all groups can be viewed as a group of permutations, thereby making them subgroups of \( S_n \).

**Definition 2.17.** We define an isomorphism from a group \( G \) to another group \( G' \) as a bijection \( \phi : G \rightarrow G' \) such that

\[
\phi(xy) = \phi(x) \ast' \phi(y),
\]

for all \( x, y \) in \( G \), where \( \ast' \) is the binary operator of \( G' \).

The property of group isomorphism is more generally referred to as the homomorphism property, since a homomorphism is any function \( \phi \) that satisfies \( \phi(xy) = \phi(x)\phi(y) \).
Lemma 2.4. Let \( G, G' \) be two groups, and \( \phi : G \rightarrow G' \) be a one-to-one function such that the homomorphism property is fulfilled, that is, \( \phi(xy) = \phi(x)\phi(y) \) for all \( x, y \in G \). Then \( \phi(G) \) is a subgroup of \( G' \), and \( \phi \) is an isomorphism from \( G \) to this subgroup \( \phi(G) \).

Proof - To show that \( \phi(G) \) is a subgroup of \( G' \), we need to show closure of the image, that the identity of \( G \) is in \( G' \), and that all elements of \( \phi(G) \) have inverses within the same group. Let the identity of \( G' \) be \( \iota' \). Then, by assumption:

\[
\iota' \phi(\iota) = \phi(\iota) = \phi(\iota \iota) = \phi(\iota)\phi(\iota).
\]

Canceling out \( \phi(\iota) \) on both sides, we see that \( \iota' = \phi(\iota) \), so the identity is transferred from \( G \) to \( G' \) by the mapping. Moving on, for elements \( x \) in \( G \),

\[
\iota' = \phi(\iota) = \phi(xx^{-1}) = \phi(x)\phi(x^{-1}) = x' \phi(x^{-1}),
\]

so inverses must also be mapped to \( G' \), since \( \phi(x^{-1}) = (x')^{-1} = (\phi(x))^{-1} \). Finally, we shall prove the closure of \( \phi(G) \). Note that the homomorphism property states that: \( \phi(xy) = \phi(x)\phi(y) \), so we can conclude that any two elements \( \phi(x), \phi(y) \in G' \) form a product within \( G' \), since \( \phi(xy) \) obviously lies within \( G' \). Thus the proof is ended. \( \Box \)

Theorem 2.5. Cayley’s Theorem

Every group is isomorphic to a subgroup of \( S_n \), that is, every group can be expressed as a group of permutations.

Proof - From [3]

In the lemma, we have shown that any group \( G \) is isomorphic to a subgroup of some symmetric group, say \( S_G \) (the subgroup in question) if there exists a mapping \( \phi : G \rightarrow S_G \) such
that

\[ \phi(xy) = \phi(x) \ast \phi(y). \]

We need to find such a mapping. Let \( \lambda_x(g) = xg, \forall g \in G \) (lambda stands for \textit{left multiplication}). Then \( \lambda_x(a) = \lambda_x(b) \iff xa = xb \iff a = b \) by cancellation, so it is a one-to-one function that maps \( G \) on itself. If \( \lambda_x \) does also map the whole of \( G \) on itself, then \( \phi(x) = \lambda_x \) is a good candidate for a function, since then \( \lambda_x \) is a bijection from \( G \) to itself, and can be viewed as a permutation of \( G \).

Indeed, since \( \lambda_x(x^{-1}c) = x(x^{-1}c) = c \) for all \( c, x \in G \), it is a bijection on \( G \), and can be viewed as a permutation. Let us examine the function \( \phi(x) = \lambda_x, x \in G \). If \( \phi(xy) = \phi(x)\phi(y) \), then we have an isomorphism and are done. Let \( g \) be the argument for lambda. The lefthand side equals \( \phi(xy)(g) = \lambda_{xy}g = (xy)g \), and the righthand side equals \( \phi(x)\phi(y) = \lambda_x(\lambda_y(g)) = x(yg) \), since lambda is a permutation and should be viewed upon as a function. The one thing left is to see if \( (xy)g = x(yg), x, y, g, \in G \), but associativity holds within \( G \), and the proof is complete. \( \square \)

This is a nifty property of groups, since we can rest upon Cayley’s theorem to say that we can "translate" group elements from whatever form they are into permutations. Thus, if we would find out that some fact hold for a group, we can say that the same fact holds for the permutation group, and vice versa. Also we can say that if two groups are isomorphic to one group of permutations, then everything pertaining to one of the groups holds for the other one too. They are essentially the same group, but we have labeled their respective group elements differently. This is especially handy when we want to represent groups within computer programs.

Now that we can translate symmetries into permutations, there are a couple more things that we need to know about permutations. Recall that cyclic structures often appear in
groups. However, they can also be found within permutations.

**Definition 2.18.** We define a **cycle of a permutation** as a subset of elements in the range of the permutation such that they form a closed set under the permutation, and such that all elements in the subset are generated by each other. If the subset contains the element \(a, b, c, \ldots, x\), then we write the cycle as \((a, b, c, \ldots, x)\).

**Definition 2.19.** We define the **product of two cycles** as the permutation achieved when multiplying the permutations corresponding to the two cycles.

**Example 2.12.** Regard part (c) of figure 5. There are four cycles of the permutation

\[
\begin{pmatrix}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
5 & 1 & 3 & 6 & 4 & 2 & 8 & 10 & 9 & 7
\end{pmatrix},
\]

namely \((1,5,4,6,2)\), \((7,8,10)\), \((3)\) and \((9)\). The corresponding permutations of a cycle is formed by letting each number in the cycle be mapped to the number on its right (looping back when reaching the end of the notation), and letting all other numbers be mapped to themselves. For example, the permutation corresponding to \((1,5,4,6,2)\) is

\[
\begin{pmatrix}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
5 & 1 & 3 & 6 & 4 & 2 & 7 & 8 & 9 & 10
\end{pmatrix}.
\]

The reader may verify that the product \((1,5,4,6,2)(7,8,10)(3)(9)\) is equal to the above permutation. Often cycles of length 1 are omitted when writing out cycle products, so \((1,5,4,6,2)(7,8,10)\) is also equal to this particular permutation.

Later on we will deal with a theorem that craves the cyclic representations of the symmetries in a group to give us the number of colorings.
2.4 Symmetries Within Combinatorics

2.4.1 Colorings of a Symmetric Object by Group Theory

In this section we shall mostly discuss the different colorings of a given graph. There is actually one theorem within group theory regarding symmetries which gives rise to a standard method utilized to find the different colorings of a graph. Before discussing that theorem, we need to have a look at a theorem called the Orbit-Stabilizer theorem:

**Definition 2.20.** We define the orbit of an element $i$ in a set with respect to a group $G$ as

$$O(i) = \{g(i)|g \in G\}.$$  

![Diagram of a symmetric object and its symmetries](image)

Figure 6: The orbit of the identity in $D_4$. Observe that $\iota \in O(\iota)$.  
$r$: rotation, $m$: axial mirrorings, $d$: diagonal flips, $\iota$: the identity
What the orbit of $i$ does is keeping track of where the element $i$ ends up after being acted upon by any group element in a given group.

**Definition 2.21.** We define the **stabilizer** of an element $i$ in a set with respect to a group $G$ as

$$\mathcal{G}(i) = \{g \in G | g(i) = i\}.$$

What the stabilizer does is keeping track of which group elements in a given group that maps a certain element $i$ back on itself.

**Lemma 2.6. (Orbit-Stabilizer theorem)**

If we denote the set of orbits to a given group $G$ with respect to an element $i$ by $\mathcal{O}(i)$, the stabilizer for the same element by $\mathcal{G}(i)$, and denote the set that $G$ is acting on by $N$, then we have:

$$|\mathcal{O}(i)| \cdot |\mathcal{G}(i)| = |G|, \forall i \in N$$

**Proof** - Let $g, h \in G$. Assume that $h(i) = g(i)$ for a given element $i \in G$. Then we have:

$$g(i) = h(i) \Leftrightarrow (h^{-1}g)(i) = i \Rightarrow h^{-1}g \in \mathcal{G}(i) \Leftrightarrow g \cdot \mathcal{G}(i) = h \cdot \mathcal{G}(i)$$

Note that in the last equality, both sides of the equation represent cosets of the stabilizer of $G$. Therefore, $g$ and $h$ are in the same coset of $\mathcal{G}(i)$.

Now, note that the number of elements in an orbit are in a one-to-one correspondence to the cosets of $\mathcal{G}(i)$. Lagrange’s theorem states that for any subgroup $H$ of $G$,

$$|H| \cdot (\#\text{cosets of } H) = |G|$$

Now, look at $\mathcal{G}(i)$. It is clearly a subgroup of $G$, since it contains elements of $G$ s.t. all the group axioms, together with the closure of the composition operator, holds. Let $H$ in the
lemma be replaced by $\mathcal{G}(i)$. We now have to find the number of cosets to the stabilizer. But the cosets are all the members of $\mathcal{O}(i)$, so we must have that

$$|\mathcal{G}(i)| \cdot |\mathcal{O}(i)| = |G|,$$

which is the sought expression. ◯

We are now ready to prove Burnside’s lemma, the most important one in this particular section of the paper.

**Theorem 2.7. (Burnside’s lemma)**

Let $G$ be a group of permutations on a finite set $N$. Also, denote by $K$ the number of orbits that may be constructed by using the elements of $G$ (we call these the $G$-orbits), let $g$ denote any element of $G$, and let $b$ denote the amount of fixed elements of $g$ ($|\{i \in N | g(i) = i\}|$). Then,

$$K = \frac{1}{|G|} \cdot \sum_{g \in G} b(g)$$

**Proof** - The trick here is to count the amount of pairs $(g, i) \in G \times N$ such that $g(i) = i$, in two different ways.

Step 1:

Let all elements $g$ of $G$ act on all elements of $N$. Compute the sum

$$\sum_{g \in G} b(g) = |\{(g, i) \in G \times N | g(i) = i\}| = \sum_{i \in N} |\mathcal{G}(i)|,$$

where $\mathcal{G}(i)$ is the stabilizer of $G$ with respect to $i$. By the Orbit-Stabilizer theorem,

$$\sum_{i \in N} |\mathcal{G}(i)| = \sum_{i \in N} \frac{|G|}{|\mathcal{O}(i)|}.$$
Since $|G|$ is independent of $i$,

$$\sum_{i \in N} \frac{|G|}{|O(i)|} = |G| \cdot \sum_{i \in N} \frac{1}{|O(i)|}$$

Step 2:

Let $B = \{O(i)| i \in N\}$ denote the set of orbits of $G$. $B$ is a set of sets of the elements of $N$. Then:

$$|G| \cdot \sum_{i \in N} \frac{1}{|O(i)|} = |G| \cdot \sum_{C \in B} \sum_{i \in C} \frac{1}{|O(i)|},$$

where $C$ is an orbit in $B$. Furthermore, $O(i) = C$, since they are essentially the same orbit. Then,

$$|G| \cdot \sum_{C \in B} \sum_{i \in C} \frac{1}{|O(i)|} = |G| \cdot \sum_{C \in B} \sum_{i \in C} \frac{1}{|C|} = |G| \cdot \sum_{C \in B} \frac{1}{|C|} \cdot |C| = |G| \cdot |B|$$

Have in mind that $B$ is the set of $G$-orbits on $N$, so $|B| = K$, since $K$ counts the number of $G$-orbits. We summarize:

$$\sum_{g \in G} b(g) = |G| \cdot K$$

Divide by $|G|$ to complete the proof. [5] □

Now, why would we be interested in finding the number of orbits in a given group? Suppose that we, for example, would like to make a necklace of a set number of white and black beads, and that we want to know how many distinguishable necklaces can be made, with respect to rotations and flips. Now, one way of finding the answer would be to try rotating and flipping all the different possible necklaces, and see which one are different. That would be extremely timeconsuming and cumbersome. Another way would be to find the number of orbits, since that takes care of all identical necklaces. Since all necklaces identical with respect to a given symmetry form an orbit of their own, the number of orbits essentially gives us the number of equivalence classes of necklaces with respect to the symmetry group.
of the necklaces. We can easily choose the n’th dihedral group \( D_n \) to be the symmetry group in the case of necklaces, since we could view a basic necklace as a regular polygon.

In appendix A I have added some python-code that utilizes Burnside’s lemma to compute in how many ways one can color an object with respect to its symmetry group, given the number of subelements of each color. It works quite well as long as the number of subelements is about ten or less, but I should warn that the runtime grows at a factorial rate with the number of colors, since more colors multiply the number of combinations, and to generate all of them can them take quite some time (e.g. about 1/2 hour for 10 colors and 10 elements on a MacBook).

**Example 2.13.** Let us use Burnside’s lemma to compute the number of ways to color the corners of a square with two colors, black and white. As shown in figure 7, there are sixteen ways to color the corners of the square with two colors. We want to find the number of orbits of the fourth dihedral group \( D_4 \). The formula obtained becomes

\[
K = \frac{1}{|D_4|} \cdot \sum_{g \in D_4} b(g),
\]

where \( b(g) \) is the number of colorings that stay fixed when applying \( g \). We take each symmetry in \( D_4 \) and check with figure 7 which colorings that stay the same after each symmetry operation:

<table>
<thead>
<tr>
<th>Symmetry (g)</th>
<th>Fixed states</th>
<th>b(g)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \iota )</td>
<td>All states</td>
<td>16</td>
</tr>
<tr>
<td>( r_1 )</td>
<td>a,b</td>
<td>2</td>
</tr>
<tr>
<td>( r_2 )</td>
<td>a,b,k,l</td>
<td>4</td>
</tr>
<tr>
<td>( r_3 )</td>
<td>a,b</td>
<td>2</td>
</tr>
<tr>
<td>( m_1 )</td>
<td>a,b,g,j</td>
<td>4</td>
</tr>
<tr>
<td>( m_2 )</td>
<td>a,b,h,i</td>
<td>4</td>
</tr>
<tr>
<td>( d_1 )</td>
<td>a,b,d,f,k,l,n,p</td>
<td>8</td>
</tr>
<tr>
<td>( d_2 )</td>
<td>a,b,c,e,k,l,m,o</td>
<td>8</td>
</tr>
</tbody>
</table>

25
Summing up, $\sum_{g \in D_4} b(g) = 16 + 2 + 4 + 2 + 4 + 4 + 8 + 8 = 48$. Dividing by $|D_4| = 8$, we get that there are $K = 48/8 = 6$ orbits in total. This is equal to the number of colorings, so the number of ways to bicolor the corners of a square with respect to symmetry is 6.

**Example 2.14.** Let us repeat the exercise of bicoloring the square, but a bit smarter this time around. Label the corners going clockwise by $c_1, c_2, c_3$ and $c_4$, and let us write $c_i = c_j$ if the colors of the two corners $c_i$ and $c_j$ are the same. Let us go through each symmetry again. Firstly, the identity preserves all properties every time, so that symmetry brings 16 to the
total sum.

Moving on, what conditions on the colors of the corners should hold for \( r_1 \) to fix the square? Well, since \( c_1 \) is mapped on to \( c_2 \), then we must have that \( c_1 = c_2 \), and similarly, \( c_2 = c_3, c_3 = c_4, c_4 = c_1 \), so all corners must share the same color to be a fixed state by \( r_1 \).

We have two colors, so there are two possibilities. Note that \( r_3 \) works the same way as \( r_1 \), but in the opposite direction, so \( b(r_3) = 2 \).

For \( r_2 \) to fix states, \( c_1 = c_3 = c_1 \) and \( c_2 = c_4 = c_2 \), so we have partitioned the corners into two sets each of size 2. We have \( 2^2 = 4 \) ways to distribute colors to these sets. Since \( m_1 \) and \( m_2 \) also partitions the corners into two sets, they each bring 4 to the sum. Now \( d_1 \) and \( d_2 \) remain. For \( d_1 \) to fix a colored state, \( c_1 = c_3 \), so we have 3 parts to consider now, \( \{c_1, c_3\}, \{c_2\} \) and \( \{c_4\} \). We can distribute two colors in \( 2^3 = 8 \) ways, so \( b(d_1) = 8 \). \( d_2 \) works exactly like \( d_1 \) but in the opposite direction, so \( b(d_2) = b(d_1) \).

Summarizing, \( \sum_{g \in D_4} b(g) = 16 + 2 \cdot 2 + 4 \cdot 3 + 8 \cdot 2 = 48 \), and \( K = 48/8 = 6 \).

### 2.4.2 Generating Functions and Symmetry

There is however another way to color a graph using symmetries. To do this, we need to use a multivariate polynomial called the cycle index of a given symmetry group. [2]

**Definition 2.22.** We define a generating function of a sequence \( \{a_n\}_{n=0}^k \) to be a formal power series of degree \( k \) such that the coefficient in front of the \( n \)th term is equal to \( a_n \).

The definition will become much clearer after a couple of examples. We will not delve very deep into the world of generating functions, but we will make use of some rules applying to them. Now, to the examples:

**Example 2.15.** The function

\[
f(x) = \frac{1}{1 - x} = \sum_{k=0}^{\infty} x^k
\]

27
generates a power series which has the coefficients 1,1,1,1,1... Therefore it is a generating
function of the series \( \{1\}_0^\infty \).

**Example 2.16.** Let

\[
f(x) = (1 + x)^n = \sum_{k=0}^{n} \binom{n}{k} x^k \text{ (by the binomial theorem)}
\]

Then \( f(x) \) is the generating function for the \( n^\text{th} \) row in Pascal’s triangle. [7]

**Example 2.17.** In [7], this is one of the first demonstrations of the power of generating
functions. The Fibonacci numbers are defined as

\[
F_{n+1} = F_n + F_{n-1} , \ F_0 = 0, F_1 = 1.
\]

Let us form the generating function

\[
F(x) = \sum_{n=0}^{\infty} F_n x^n.
\]

Multiplying both sides of the first relation with \( x^n \) and summing from \( n = 1 \) to \( \infty \), we get:

\[
LHS = \sum_{n=1}^{\infty} F_{n+1} x^n = \frac{(\sum_{n=0}^{\infty} F_n x^n) - x}{x} = \frac{F(x) - x}{x}
\]

\[
RHS = \sum_{n=1}^{\infty} F_n x^n + F_{n-1} x^n = F(x) + xF(x)
\]

Thus

\[
\frac{F(x) - x}{x} = F(x) + xF(x).
\]
Solving for $F(x)$, we get

$$F(x) - x = xF(x) + x^2 F(x) \Rightarrow F(x) - xF(x) - x^2 F(x) = x \Rightarrow F(x) = \frac{x}{1 - x - x^2}.$$ 

This means that the $n$'th Fibonacci number is the coefficient before the $n$'th term in the expansion of $\frac{x}{1 - x - x^2}$. By factoring the denominator and using partial fractions, one can find the explicit expression for the $n$'th Fibonacci number, but that is beyond the scope of this text.

Now that we have an idea of what a generating function is, we return to the coloring problem. There is a type of multivariate generating function that stores a lot of data about the symmetries of a given object – the cycle index.

**Definition 2.23.** We define the type of a permutation $g$ be a partition $[1^{\alpha_1}, 2^{\alpha_2}, 3^{\alpha_3}, \ldots, n^{\alpha_n}]$ such that $g$ can be written as $\sum_{i=1}^{n} \alpha_i$ cycles of length $i$.

**Example 2.18.** The permutation from figure 5 in section 2.3 could be written $(1, 5, 4, 6, 2)(7, 8, 10)(3)(9)$. There are two cycles of length 1, one cycle of length 3, and one of length 5. Thus its type is $[1^2, 3^1, 5^1]$.

**Remark 3.** Note that we do not exponentiate the numbers in the type of a permutation, the numbers are in this case symbols only, expressing something about the permutation in question. We do not write the type of the permutation in example 2.18 as $[1, 3, 5]$, that would not make any sense in this context.

**Definition 2.24.** We define the cycle index of a group $G$ to be a multivariate polynomial of the form

$$\zeta_G(x_1, x_2, x_3, \ldots, x_n) = \frac{1}{|G|} \sum_{\sum_j \alpha_j = n} c(\alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_n) \prod_{i=1}^{n} x_i^{\alpha_i},$$
where \( c(\alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_n) \) is the number of permutations that have type \([1^{\alpha_1}, 2^{\alpha_2}, 3^{\alpha_3}, \ldots, n^{\alpha_n}]\)

It is often a heavy task to find the cycle index of a group, but it is a useful tool utilized within a theorem that we shall introduce later on.

**Example 2.19.** Find the cycle index of \(D_4\).

*Solution:* Label the corners of a square with 1, 2, 3, 4 clockwise. Then:

<table>
<thead>
<tr>
<th>Symmetry operation</th>
<th>Permutation</th>
<th>Cycles (length : amount)</th>
<th>Corresponding Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \iota )</td>
<td>1 2 3 4</td>
<td>1 : 4</td>
<td>( x_1^4 )</td>
</tr>
<tr>
<td>( r_1 )</td>
<td>2 3 4 1</td>
<td>4 : 1</td>
<td>( x_4 )</td>
</tr>
<tr>
<td>( r_2 )</td>
<td>3 4 1 2</td>
<td>2 : 2</td>
<td>( x_2^2 )</td>
</tr>
<tr>
<td>( r_3 )</td>
<td>4 1 2 3</td>
<td>4 : 1</td>
<td>( x_4 )</td>
</tr>
<tr>
<td>( m_1 )</td>
<td>2 1 4 3</td>
<td>2 : 2</td>
<td>( x_2^2 )</td>
</tr>
<tr>
<td>( m_2 )</td>
<td>4 3 2 1</td>
<td>4 : 1</td>
<td>( x_2 )</td>
</tr>
<tr>
<td>( d_1 )</td>
<td>3 2 1 4</td>
<td>1 : 2, 2 : 1</td>
<td>( x_1^2 x_2 )</td>
</tr>
<tr>
<td>( d_2 )</td>
<td>1 4 3 2</td>
<td>1 : 2, 2 : 1</td>
<td>( x_1^2 x_2 )</td>
</tr>
</tbody>
</table>

Also, \(|D_4| = 8\). Summing up the rightmost terms in the table, we find the cycle index to be:

\[
\zeta_{D_4}(x_1, x_2, x_3, x_4) = \frac{1}{8} \left( x_1^4 + 3x_2^2 + 2x_1^2x_2 + 2x_4 \right)
\]

Figure 8 captures the procedure of finding the cycle index of a given group, study it if you do not fully understand the above example. Now, we shall introduce the theorem that utilizes the cycle index to find the number of inequivalent colorings, the Polya Enumeration Theorem.
Figure 8: The procedure of finding the cycle index of a given group, here $D_4$.

1. Label all elements being acted upon, and translate each symmetry to permutations.
2. Find the cycles of each permutation achieved.
3. Convert each cycle product into a term of the form $x_1^{j_1}x_2^{j_2} \ldots$, where the indices are cycle lengths and the exponents are the number of cycles of the corresponding length.
4. Sum all the terms up, and divide by the order of the group. You now have the cycle index of the group.
Theorem 2.8. The Polya Enumeration Theorem (PET)

Let $D$ be a set of variables $a, b, c, \ldots, x$ each denoting a distinct color.

Let $U_D(a,b,c,\ldots x)$ be a generating function for the number of ways to color an object with up to $|D|$ colors.

For a given group $G$ with cycle index $\zeta_G(x_1, x_2, x_3, \ldots, x_n),

\[ U_D(a,b,c,\ldots x) = \zeta_G(\sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_n), \]

where

$$\sigma_k = a^k + b^k + c^k + \ldots + x^k.$$  

Proof - See [2] for a complete proof, as it is quite long. □

Example 2.20. Let us examine the number of ways to color a square with 2 colors (say $b$ and $w$ for black and white). Recall that we found the cycle index to be

$$\zeta_{D_4}(x_1, x_2, x_3, x_4) = \frac{1}{8} \left( x_1^4 + 3x_2^2 + 2x_1^2x_2 + 2x_4 \right).$$

Now, let us replace all variables $x_i$ with the polynomial $(b^i + w^i)$. Then,

$$U_{\{b,w\}}(b, w) = \frac{1}{8} \left( (b + w)^4 + 3(b^2 + w^2)^2 + 2(b + w)^2(b^2 + w^2) + 2(b^4 + w^4) \right).$$

Expanding, we find that

$$U_{\{b,w\}}(b, w) = b^4 + 2b^2w^2 + w^4 + b^3w + bw^3.$$  

Now, each term represents a way to color the square, and we see that either all corners are
black, all are white, two are black and two are white, three are black and one white, or one black and three white. Also, there are two possible colorings in the 2 black-2 white scenario, but only one in all other cases.

Figure 9: The ways to 2-color the corners of a square

This is a great mechanical way to find the number of colorings, and it is also a very nice theorem combining group theory and generating functions. If one is more specific in asking for, say, the number of colorings involving 2 black and two white vertices, we do not need to expand the giant polynomial expression in the theorem, one could use some common tricks pertaining to generating functions.

Example 2.21. Let us study a more complicated group than the square, say the cube for example. We will see later on that the cycle index for the cube’s symmetry group $C$ is

$$
\zeta_C(x_1, x_2, x_3, x_4, x_5, x_6) = \frac{1}{24} \left( 8x_3^2 + 3x_1^2x_2^2 + x_1^6 + 6x_1^2x_4 + 6x_2^3 \right).
$$

Let us deduce the number of ways to color a cube with 3 colors, and such that each color gets two faces of the cube. Inserting the necessary polynomials (we denote red with r, blue with b
and yellow with $y$), we get:

$$U_{\{b,r,y\}}(b,r,y) = \frac{1}{24} \left( 8(b^3 + r^3 + y^3)^2 + 3(b + r + y)^2(b^2 + r^2 + y^2)^2 
\right. $$

$\left. + (b + r + y)^6 + 6(b + r + y)^2(b^4 + r^4 + y^4) + 6(b^2 + r^2 + y^2)^3 \right)$. 

Let us scrutinize the expression. We want to end up with terms of the form $b^2r^2y^2$, and the first term cannot give us such terms (one variable will have an exponent of at least 3, which is too much). Therefore we can completely skip the first term, and focus on the others! Also, the term second from the end will provide exponents of at least 4, so we only have the expression

$$\frac{1}{24} \left( 3(b + r + y)^2(b^2 + r^2 + y^2)^2 + (b + r + y)^6 + 6(b^2 + r^2 + y^2)^3 \right).$$

to consider. We will consider one term at a time.

The first one expands to

$$3(b^2 + r^2 + y^2 + 2br + 2by + 2ry)(b^4 + r^4 + y^4 + 2b^2r^2 + 2b^2y^2 + 2r^2y^2).$$

Note that none of the last three terms in the first parenthesis will combine with any in the second parenthesis to form $b^2r^2y^2$, and vice versa for the first three terms in the second parenthesis. We can just cross them over, and end up with

$$3(b^2 + r^2 + y^2)(2b^2r^2 + 2b^2y^2 + 2r^2y^2).$$

For each term in the first parenthesis, there is exactly one term in the other that will form $b^2r^2y^2$ with it, so we end up with three terms of the form $2b^2r^2y^2$, and multiplication with the coefficient 3 we get 18 from this term. Remember 18 for later.
The next term we shall consider is \((b + r + y)^6\). The exponent is scary, but we can use the **multinomial theorem** to see that the answer for this term is

\[
\binom{6}{2, 2, 2} = \frac{6!}{2!2!2!} = \frac{720}{8} = 90,
\]

bringing us to a total of \(90 + 18 = 108\).

Finally, we shall consider the term \(6(b^2 + r^2 + y^2)^3\). Can we simplify this? Yes we can, since determining the coefficient for \(b^2r^2y^2\) in \(6(b^2 + r^2 + y^2)^3\) is equivalent to determining the coefficient for the term \(bry\) in the expansion of \(6(b + r + y)^3\). We now use the multinomial theorem again to find the answer to be

\[
\binom{3}{1, 1, 1} = \frac{3!}{1!1!1!} = 6 = 6,
\]

thus making this term contribute with the coefficient \(6 \cdot 6 = 36\).

Our total sum of the coefficients is now \(108 + 36 = 144\). Dividing by the order of the group (we have to do that too!), we get our final answer to be

\[
\#\text{colorings} = \frac{144}{24} = 6.
\]

Thus there are 6 different ways to color a cube with three colors s.t. all colors get two faces.

**Remark 4.** Note that the Polya Enumeration Theorem solves the coloring problem of any finite group for all possible partitionings of a given number of colors in one go, by creating a polynomial in which all partitionings are represented by a specific term each. Compare that with Burnside’s lemma, that has to be used once for all partitionings of a given number of colors. The PET is much more efficient if one wants to consider all possible scenarios, but Burnside’s lemma is easier to use for more specific scenarios.
2.5 Symmetry in More than Two Dimensions

One object that is classic when we speak of colorings is the cube, since we would often want to know in how many ways a cube can be colored. However the cube bears a property that makes it hard to find out the answer by a brute-force approach: It has threedimensional symmetry! The fact that we have so many ways of rotating and flipping and turning a cube (24 in fact) makes it a horribly timeconsuming, not to say nauseating process to search out each and every coloring.

Fear not however, since there are a couple of ways to ease the burden. Firstly, we might want to label the faces of the cube (we are not interested in coloring the corners here), so to begin with we can construct the dual graph of the cube, which will turn out to be an octahedron.

![Figure 10: A cube with its dual octahedron](image)

Now, we just label the vertices of the graph (the corners of the octahedron), in some manner. What we want to do to proceed with the problem is to find all the group elements in the symmetry group of the square, and then apply Cayley’s theorem: we shall express the symmetries of the cube as permutations, thus reducing the problem from a whopping three dimensions to a single!

To find the group elements, we reason like this:
Choose one face of the cube, and let it be fixed to face you. This can be done in 6 ways. Now, we have fixed one face, and we can rotate the cube about an axis going straight into the side facing you and coming out of the opposite side. There are four rotations in total. By the multiplication principle, we can fix a face and then rotate it in \(6 \cdot 4 = 24\) ways. We have found 24 symmetries of the cube so far. Are there any more? No, since we have covered every single situation in which any face of the cube faces us. Thus, there are 24 symmetries in total.

Now, we do not have to introduce certain symbols for rotations of the cube, we can just use permutations to denote all 24 symmetries. That is indeed still a little cumbersome, thinking about the number of numbers we have to deal with, but the problem can now be solved by finding the cycle index of the group, or by Burnside’s lemma if you wish.

Let us use the Python code in the appendix to find cycle indices of some of the Platonic solids (Observe that we find the cycle indices with respect to how the faces of each solid interchange). The cycle indices for the dodecahedron and the icosahedron have been looked up in [2], as their symmetries were too hard to analyze for the author.

<table>
<thead>
<tr>
<th>Solid</th>
<th>Cycle index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tetrahedron</td>
<td>(\frac{1}{12}(3x_2^3 + x_1^4 + 8x_1x_3))</td>
</tr>
<tr>
<td>Cube</td>
<td>(\frac{1}{24}(8x_2^2 + 3x_1^2x_2^2 + x_1^6 + 6x_1^2x_4 + 6x_2^3))</td>
</tr>
<tr>
<td>Octahedron</td>
<td>(\frac{1}{24}(6x_2^4 + 8x_1^2x_3^2 + x_1^8 + 9x_2^4))</td>
</tr>
<tr>
<td>Cube, with inversion of faces</td>
<td>(\frac{1}{48}(6x_2x_4 + 8x_3^2 + 9x_1^2x_2^2 + 8x_6 + 3x_1^4x_2 + 6x_1^2x_4 + 7x_2^3 + x_1^6))</td>
</tr>
<tr>
<td>Dodecahedron</td>
<td>(\frac{1}{60}(x_1^{12} + 24x_1^2x_5^2 + 15x_1^6 + 20x_3^4))</td>
</tr>
<tr>
<td>Icosahedron</td>
<td>(\frac{1}{60}(x_1^{20} + 20x_1^2x_3^6 + 15x_2^{10} + 24x_3^4))</td>
</tr>
</tbody>
</table>
3 Designing the Counting Algorithms

We have now seen two extraordinary theorems that solve all problems related to colorings with respect to symmetries. However they both are computationally heavy for the common man, especially when the symmetry groups are large. The author has implemented the necessary tools to use both theorems with a computer, and everything is found in the appendix. In this section we will lay down every piece of the mathematical jigsaw puzzle that makes the code work. We go through each problem that needed to be solved in order to make the program work, and describe the solutions.

Problem 1 - Beginning the procedure

Let us consider an arbitrary symmetry group, and that we want to find the number of colorings such that some partitioning of the colors holds. The group is an abstract object, and the symmetry elements of one group may be drastically different from those in another, as when we compare e.g. rotations of squares with rotations of octahedrons. We need a general approach to represent arbitrary groups as data structures. How do we easiest represent the group, when performing the computations in the computer?

Solution

This is where Cayley’s theorem steps in. It tells us that we can translate each element of the group to permutations, we just have to establish an isomorphism mapping the elements of the group to a subgroup of a symmetric group on \( n \) letters. This isomorphism can be achieved if we label the objects being mapped around by the symmetries. Thus we can represent all groups as permutation groups. We can use arrays or lists (or even tuples in Python) when representing a permutation in a computer, since the indexing of the arrays can be used as the top row of any permutation (That is, we store in the \( n \)’th cell the number to which \( n \) is
mapped). We can store all permutations in a group within a matrix, or a list of lists. This can be done for any finite group of discrete symmetries, so the problem is solved.

**Problem 2**

A group of symmetries can be very big, and we do not want to jot down each and every permutation by hand (for example, the octahedron has a total of 24 symmetries). It is very cumbersome to translate a whole group into permutations, and is not feasible even for smaller groups such as $D_4$ (in that case, we need 32 numbers to describe all bottom rows of the permutations). How do we reduce the amount of work being done while still telling the computer what the group looks like?

**Solution**

For a given group, there exists a *generating set* of elements that together can generate the whole group with some frequent use of the composition operator. We need only include some generators of the set in the files, letting the computer to generate the rest of the elements. This reduces the manual work quite a lot, since e.g. $D_4$ only needs 3 generators in total to be generated.

**Problem 3**

To generate a group from its generating set, we need to use the composition operator. How do we perform composition of symmetry operations when we have represented each symmetry with a permutation?

**Solution**

As described in section 2.3, we need only map each element of a permutation $h$ with the permutation $g$ in order to find the product $hg$. Since we will represent all the group elements
as permutations, we use *permutation multiplication* as the permutation analogue to symmetry composition.

**Problem 4**

To use the PET on symmetry groups, the cycle index is an unavoidable necessity. How do we find the cycle index of a given group when it is represented in the computer?

**Solution**

We take a group element, and focus on the set that it permutes. We begin with the first element in that set, and map it around until we have completed it’s cycle. We note the length of it, and continue with the next element. If it has already been reached by a previous element’s cycle, then we ignore it, otherwise we take it and map it around, just like before. We iterate this for all the symmetry operations in the group. [1]

**Problems 5, 6**

We do also need some permutation generator, generating all possible distributions of colors for Burnside’s lemma, and we need the computer to know how to compute with polynomials for the PET.

**Solution**

It is not that hard to implement recursive algorithms that generate all permutations, and we really won’t use Burnside’s lemma on groups with more than 10-12 elements, so Python’s maximum depth of 100 recursions is more than enough. As for the polynomials, the author has solved it by object-oriented programming. Three classes have been implemented, one for variables, one for crossproduct terms, and one for the polynomial in its entity. Have a look at appendix D for the code and comments.
4 Some Exotic Symmetry Groups

In this section we shall examine some objects that have nonstandard symmetry groups. The author is not sure of whether these objects have been studied before, and he has not found any data pertaining to it either, so anything beyond here may be denoted in a nonstandard way, or be completely new. The code in the appendix will be used when computing cycle indices here, since the groups are large.

4.1 Nested Necklaces

We have seen how regular polygons can represent regular necklaces, but what if we generalized the idea?

Definition 4.1. We define a regularly nested necklace of degrees \( m:n \) \((m,n \in \mathbb{Z}^+)\) to be a necklace with \( m \) other necklaces treced on it, each holding \( n \) beads.

Definition 4.2. We inductively define a regularly nested necklace of degrees \( a:b:c:\ldots:m:n \) \((a,b,c,\ldots,m,n \in \mathbb{Z}^+)\) to be a nested necklace of degrees \( a:b:c:\ldots:m \), holding \( n \) beads/necklace in the innermost necklaces.

Figure 11: A nested necklace of degrees 3:3
In this paper, we will only consider necklaces of degrees n:n. Regard the following diagram representing the necklace of degrees 3:3.

Figure 12: The nested necklace of degrees 3:3, and its corresponding diagram

From the right image, we quickly see that $D_3$ works on the big, outer necklace, and that $C_3$ works on each inner necklace (we cannot magically flip a necklace that has a thread going through it, topologists will be all over you if you think otherwise!). We state a definition:

**Definition 4.3.** We denote the symmetry group acting on a nested necklace of degrees $a:b:c:d:...:n$ by $D_{a:b:c:d:...:n}$, and we call it the $a:b:c:...:n$'th Dihedral group.

**Theorem 4.1.** $|D_{m:n}| = 2mn^m$, when $m > 2$.

**Proof** - The outer necklace can be rotated and flipped according to the symmetries of $D_m$, and each of the $m$ inner necklaces are acted upon by $C_n$. The $m$ inner necklaces can be oriented in $|D_m| = 2m$ ways, and each of the $m$ inner necklaces have $|C_n| = n$ orientations. From the principle of multiplication we get that the total number of symmetries are

$$|D_{m:n}| = |D_m| \cdot |C_n|^m = 2m \cdot n^m = 2mn^m.$$

Note that $m > 2$ is a requirement for the outermost band to have a dihedral symmetry, since
flips and rotations of a point \((D_1)\) or a line \((D_2)\) would equal each other. \(\square\)

**Example 4.1.** The nested necklace of degrees \(3 : 3\) has \(2 \cdot 3 \cdot 3^3 = 162\) symmetries.

We generalize this theorem to nested necklaces of degrees \(a : b : c : d : \ldots : n\).

**Theorem 4.2.** If \(x_1 > 2\),

\[
|D_{x_1:x_2:x_3: \ldots :x_n}| = 2 \prod_{i=1}^{n} x_i^{\prod_{j=1}^{n-i} x_j}
\]

**Proof** - We proceed by induction on the number of degrees. We have already checked the cases with necklaces of degrees \(x_1\) and \(x_1 : x_2\). Assume now that we have some arbitrary nested necklace of degrees \(x_1 : x_2 : x_3 : \ldots : x_{n-1}\), and that the order of \(D_{x_1:x_2:x_3: \ldots :x_{n-1}}\) equals \(2 \prod_{i=1}^{n-1} x_i^{\prod_{j=1}^{n-i} x_j}\). Replace each of the innermost \(\prod_{i=1}^{n-1} x_i^{x_i}\) beads with necklaces of \(x_n\) beads each. Each of the innermost necklaces can be oriented in \(|C_{x_n}| = x_n\) ways, so by the principle of multiplication, those necklaces can achieve a total of \(x_n^{\prod_{j=1}^{n-1} x_j}\) states. Using the principle of multiplication again, multiplying with the amount of symmetries of the necklace of degrees \(x_1 : x_2 : x_3 : \ldots : x_{n-1}\), the sought expression is achieved. \(\square\)

Let us explore the cycle indices for some nested necklaces of degrees \(n : n\). Since these symmetry groups are a tad too big and complicated to be analyzed manually, we will use the PYTHON code in the appendix.
Example 4.2. Let us use the Python code in the appendix to compute the cycle indices for the $n$th Dihedral group for some small $n$:

<table>
<thead>
<tr>
<th>$n$</th>
<th>Cycle index of $D_{n,n}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>$\frac{1}{162}(54x_1x_2x_6 + x_1^9 + 26x_3^3 + 12x_3^2x_3^2 + 6x_3^6x_3 + 27x_1x_3^4 + 36x_9)$</td>
</tr>
<tr>
<td>4</td>
<td>$\frac{1}{2048}(40x_2^6x_4 + 64x_1^4x_4x_8 + 160x_2^4x_4^2 + 32x_1^8x_4^2 + 4x_1^{12}x_4^2 + x_1^{16} + 256x_4^2x_8 + 32x_1^4x_2^2x_8 + 57x_2^8 + 208x_2^4x_8 + 20x_1^4x_2^6 + 64x_2^5x_4^3 + 64x_1^4x_4^3 + 64x_1^4x_2^2x_4^2 + 24x_1^8x_2^2x_4 + 14x_1^8x_4^2 + 8x_1^{12}x_4 + 56x_1^4x_2^4x_4 + 224x_4^4 + 16x_1^8x_8 + 256x_{16} + 64x_2^2x_4x_8 + 320x_8^2)$</td>
</tr>
<tr>
<td>5</td>
<td>$\frac{1}{31250}(625x_1^2x_2^{12} + 640x_1^{10}x_3^5 + 10000x_1x_2^2x_1^{10} + 1280x_5^5x_5^4 + 10000x_25 + 20x_1^{20}x_5 + 160x_1^{15}x_5^2 + x_1^{25} + 5000x_1x_2^7x_10 + 3524x_5^5)$</td>
</tr>
</tbody>
</table>

Note that theorem 4.1 holds in the table.

4.2 Chemical Molecules

It is possible to use the PET and Burnside’s lemma to find the number of molecules in some cases. The following example is from the final exam of the Swedish chemistry olympiad 2008, [6]:

Example 4.3. How many polychlorinated biphenyles with chemical formula $C_{12}H_8Cl_2$ do there exist?

![Figure 13: The carbon skeleton of PCB molecules](image)

It is possible to solve this problem by performing a somewhat structured brute force, but we will proceed by the solutions that stand in this paper. We need to find the symmetries of the carbon skeleton. We can flip each of the benzene rings 180° about the horizontal axis, and we
can flip the whole molecule about the vertical axis. The generating set can be chosen to be one of the minor flips, and the major flip. We would have a total of \(2 \cdot 2 \cdot 2 = 8\) symmetries, since we can choose whether to flip the whole molecule, and whether to flip each of the benzene rings.

Now, how do we use the PET, or Burnside’s lemma to solve this problem? Well, from highschool chemistry we know that each carbon atom in a benzene ring can hold one other complex. Looking at figure 12, the two carbon atoms connecting the rings are already occupied, so we have ten remaining carbon atoms that can each hold either a hydrogen atom or a chlorine atom. We can see this problem as coloring these 10 carbon atoms by hydrogen or by chlorine. Using Burnside’s lemma, we see that there are 12 ways of coloring this molecule such that there are two chlorine-colored carbon atoms.

**Example 4.4.** How many polychlorinated biphenyles consisting of only the carbon skeleton, hydrogen and chlorine do there exist?

Using the Python code, the cycle index is found to be

\[
\frac{1}{8}(x_1^2x_2^4 + x_1^10 + 2x_2x_4^2 + 2x_5^5 + 2x_1^6x_2^2),
\]

and the generating function for two colors is

\[
x_1^10 + 3x_1^9x_2 + 12x_1^8x_2^2 + 24x_1^7x_3^3 + 42x_1^6x_4^4 + 46x_1^5x_5^5 + 42x_1^4x_6^6 + 24x_1^3x_7^7 + 12x_1^2x_8^8 + 3x_1x_9^9 + x_2^{10}.
\]

Summing up the coefficients, we see that there are 210 possible such colorings. However, two of these terms correspond to molecules with zero or only one chlorine atom (hence they are not polychlorinated biphenyles), and crossing these terms over, there remain 206 possibilities.

Let us expand our view a little.
Example 4.5. Look at the following carbon skeleton:

![Carbon Skeleton]

Figure 14: An interesting carbon skeleton

Let us call such a skeleton, colored with chlorine atoms, a polychlorated quadrophenyle. How many of these molecules, PCQ, do there exist?

This molecule will take the shape of a tetrahedron of benzene rings, and will have the symmetry of a tetrahedron, but we can also flip each benzene ring by $180^\circ$.

Using the Python code, we find that the cycle index is

$$
\frac{1}{192}(12x_2^{10} + 32x_1x_2x_3x_6^2 + 24x_2^6x_4^2 + 32x_1^5x_3^5 + x_1^{20} + 12x_2^2x_4^4 + 32x_1x_2^2x_3^5 + x_1^4x_8^3 + 32x_1^5x_3x_6^2 + 6x_1^{12}x_2^4 + 4x_1^{16}x_2^2 + 4x_1^8x_2^6).
$$

Summing up all the coefficients in the corresponding generating function (which is too large to be readable here), and subtracting the possibilities with one or zero chlorine atoms, there are 13696 such polychlorinated molecules.
5 Results

This paper has resulted in a number of results, both theoretical and practical. The main result of this project is the Python code that can solve coloring problems regarding symmetrical objects. Functions that perform group theoretic computations have been implemented to make computing with the PET and Burnside’s lemma possible. The functions in question are described in section 4, but as a summary, group generation, cycle finding in permutations, a polynomial class for the Polya Enumeration Theorem and a permutation generator for Burnside’s lemma have been implemented, debugged and optimized.

This program has been used to find the cycle indices of several groups within the theory section. The groups in question range in complexity from $C_2$ to $D_{5:5}$ and the symmetry group of the cube. Below we give a list of some of the symmetry groups that we have studied, along with their respective cycle indices. Note that all cycle indices in the table except those of the dodecahedron and icosahedron have been found by using the Python code in the appendix.

<table>
<thead>
<tr>
<th>Group</th>
<th>Cycle index</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&lt;{\iota}, \circ&gt;$</td>
<td>$\frac{1}{\iota}(x_1^{\vert H\vert})$, where $H$ is the set that $\iota$ acts upon</td>
</tr>
<tr>
<td>$C_3$</td>
<td>$\frac{1}{3}(2x_3 + x_1^3)$</td>
</tr>
<tr>
<td>$C_4$</td>
<td>$\frac{1}{4}(x_2^2 + x_1^4 + 2x_4)$</td>
</tr>
<tr>
<td>$C_5$</td>
<td>$\frac{1}{5}(x_1^5 + 4x_5)$</td>
</tr>
<tr>
<td>$D_3$</td>
<td>$\frac{1}{6}(2x_3 + x_1^3 + 3x_1x_2)$</td>
</tr>
<tr>
<td>$D_4$</td>
<td>$\frac{1}{8}(x_4^4 + 3x_2^2 + 2x_1^2x_2 + 2x_4)$</td>
</tr>
<tr>
<td>$D_5$</td>
<td>$\frac{1}{10}(x_1^5 + 4x_5 + 5x_1x_2^2)$</td>
</tr>
<tr>
<td>$S_3$ (= $D_3$)</td>
<td>$\frac{1}{6}(2x_3 + x_1^3 + 3x_1x_2)$</td>
</tr>
<tr>
<td>$S_4$</td>
<td>$\frac{1}{24}(6x_1^2x_2 + 3x_2^2 + x_1^4 + 8x_1x_3 + 6x_4)$</td>
</tr>
<tr>
<td>$S_5$</td>
<td>$\frac{1}{120}(24x_5 + 20x_1^2x_3 + 10x_1^3x_2 + 15x_1x_2^2 + 30x_1x_4 + x_1^5 + 20x_2x_3)$</td>
</tr>
<tr>
<td>Group</td>
<td>Cycle index</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>$D_{3:3}$</td>
<td>$\frac{1}{162}(54x_1x_2x_6 + x_1^9 + 26x_1^3x_3^2 + 12x_1^3x_3^2 + 6x_1^6x_3 + 27x_1x_2^4 + 36x_9)$</td>
</tr>
<tr>
<td>$D_{4:4}$</td>
<td>$\frac{1}{2048}(40x_2^6x_4 + 64x_1^4x_4x_8 + 160x_2^4x_4^2 + 32x_1^8x_2^2 + 4x_1^{12}x_2^2 + x_1^{16} + 256x_1^2x_8 + 32x_1^4x_3^2x_8 + 57x_8^8 + 208x_2^4x_8 + 20x_1^4x_2^6 + 64x_2^4x_8^3 + 64x_1^4x_2^2x_4^2 + 24x_1^8x_2^2x_4 + 14x_8^4x_2^4 + 8x_1^{12}x_4 + 56x_1^4x_4x_4 + 224x_1^4 + 16x_1^8x_8 + 256x_1^6x_8 + 64x_2^2x_4x_8 + 320x_2^4)$</td>
</tr>
<tr>
<td>$D_{5:5}$</td>
<td>$\frac{1}{31250}(625x_1x_2^{12} + 640x_1^{10}x_3^3 + 10000x_1^6x_2^2x_1^2 + 1280x_1^5x_3^4 + 10000x_2^{25} + 20x_1^{20}x_5 + 160x_1^{15}x_5^2 + x_1^{25} + 5000x_1x_2^7 x_10 + 3524x_5^5)$</td>
</tr>
<tr>
<td>Tetrahedron</td>
<td>$\frac{1}{12}(3x_2^2 + x_4^4 + 8x_1x_3)$</td>
</tr>
<tr>
<td>Cube</td>
<td>$\frac{1}{24}(8x_3^2 + 3x_1^2x_2^2 + x_1^4 + 6x_1^2x_4 + 6x_2^2)$</td>
</tr>
<tr>
<td>Octahedron</td>
<td>$\frac{1}{24}(6x_1^2 + 8x_2^2x_2^2 + x_3^8 + 9x_4^2)$</td>
</tr>
<tr>
<td>Cube, with</td>
<td>$\frac{1}{48}(6x_2x_4 + 8x_3^2 + 9x_1x_2^2 + 8x_6 + 3x_1^4x_2 + 6x_1^2x_4 + 7x_3^2 + x_1^6)$</td>
</tr>
<tr>
<td>inversion of faces</td>
<td></td>
</tr>
<tr>
<td>Dodecahedron</td>
<td>$\frac{1}{60}(x_1^{12} + 24x_1^2x_3^2 + 15x_1^6 + 20x_3^4)$</td>
</tr>
<tr>
<td>Icosahedron</td>
<td>$\frac{1}{60}(x_1^{20} + 20x_1^2x_6^2 + 15x_1^{10} + 24x_2^4)$</td>
</tr>
<tr>
<td>PCB molecule</td>
<td>$\frac{1}{8}(x_1^2x_2^4 + x_1^{10} + 2x_2x_4^2 + 2x_3 + 2x_1^6x_2^2)$</td>
</tr>
<tr>
<td>PCQ molecule</td>
<td>$\frac{1}{192}(12x_1^{10} + 32x_1x_2^2x_3x_6^2 + 24x_2x_4^2 + 32x_1^5x_5^3 + x_1^{20} + 12x_2^2x_4^4 + 32x_1^2x_2^5x_3^5 + x_1^{12}x_2^4 + 6x_1^{12}x_2^2 + 4x_1^{16}x_2^2 + 4x_8^8x_6^2)$</td>
</tr>
</tbody>
</table>

This library of cycle indices can be looked upon as a future reference when searching for certain cycle indices.
A couple of definitions and theorems regarding nested necklaces have been stated and proven, namely:

**Definition 5.1.** *We define a regularly nested necklace of degrees* $m:n$ ($m,n \in \mathbb{Z}^+$) *to be a necklace with* $m$ *other necklaces threaded on it, each holding* $n$ *beads.*

**Definition 5.2.** *We inductively define a regularly nested necklace of degrees* $a:b:c:\ldots:m:n$ $(a,b,c,\ldots,m,n \in \mathbb{Z}^+)$ *to be a nested necklace of degrees* $a:b:c:\ldots:m$, *holding* $n$ *beads/necklace in the innermost necklaces.*

**Definition 5.3.** *We denote the symmetry group acting on a nested necklace of degrees* $a:b:c:d:\ldots:n$ *by* $D_{a:b:c:d:\ldots:n}$, *and we call it the* $a:b:c:\ldots:n$'th *Dihedral group.*

**Theorem 5.1.** $|D_{m:n}| = 2mn^m$, when $m > 2$.

**Theorem 5.2.** If $x_1 > 2$,

$$|D_{x_1:x_2:x_3:\ldots:x_n}| = 2 \prod_{i=1}^{n} x_i \prod_{j=1}^{i-1} x_j$$

![Figure 15: A nested necklace of degrees 3:3](image-url)
6 Discussion

The primary feeling is that the problem of coloring with respect to symmetry has got general solutions, and that Burnside’s lemma and the PET greatly reduces the time and energy taken to solve these problems. With the aid of computers, it is especially easy and timesaving to solve these problems, and the PYTHON code provided with this paper can be of importance in future and more detailed studies on certain symmetry groups.

It is evident that the solutions to coloring problems involving symmetry are much harder to grasp theoretically than otherwise, paying heed to the abstract level of group theory and of generating functions. It would be a good idea to find direct formulas for as many cycle indices as possible, since that would mean a lot of saved computing time when finding the cycle index by computer. This is however a hard task, and it may need more time than that of this paper, even for somewhat regular symmetry groups (for $C_n$ the task has been solved however, see [2]).

As for which theorem to prefer between Burnside’s lemma and the PET, the algorithm of the former must be executed once for each partitioning of the colors that you want to color an object with, but that of the latter solves the cases of all partitionings of a given number of colors at the same time. The PET is just much more efficient than Burnside’s lemma. I would personally not recommend Burnside’s lemma for situations when the number of elements to be colored is large, since we then would need to generate all colorings, which would take factorial time. The PET only needs to perform a lot of polynomial multiplication at worst, and would then solve all possible cases in one go, making it more attractive in this scenario.

When using the PET, it seems better to me to save the cycle index of a group for later use, instead of saving all possible generating functions decided by different color partitionings, since the latter generating functions would be too big and incomprehensible, should we save
them in a table. The cycle index can be used to find all of those polynomials, and is a lot easier to interpret as is.

All in all, this project has resulted in a couple of great computer programs that one can toy with anytime, nested necklaces have been introduced and begun to be studied, and a solid cycle index library has been compiled for future references.

6.1 Future Studies

There are many ways to go from here, especially with the computer programs readily constructed. One can focus on the cycle indices of molecules to solve coloring problems arising in chemistry, or one can continue to study the nested necklaces. Other than that, focusing on bicoloring symmetric objects may also be a way to proceed, since that would be a logical first step toward a greater understanding of colorings with respect to symmetries. Overall, one could focus on studying certain types of symmetry groups in detail, and try to find direct formulas for their respective cycle indices.

Other than that, one could try to extend the library of cycle indices by using the programs constructed. Also, optimization and other enhancements to the programs could be a way to proceed too, since that would enhance future studies. The optimal scenario would be if the whole program could be translated to a faster language, like C/C++ or maybe Java, since that would make future research go much more efficiently than with Python, although that could be a minor project on its own.

There seem to be things to do in this area of mathematics, even if today’s researchers focus on breaking symmetries, rather than keeping them.
7 Acknowledgements

There are many I would like to thank for helping me perform this project. First and foremost, I would like to thank Lars Hellström and Daniel Andrén at the department for mathematics and mathematical statistics for helping me throughout the project. In particular, I would like to thank Lars Hellström for helping me understand the relation between symmetries and groups, and for explaining the theory behind Burnside’s lemma for me, and I would like to thank Daniel Andrén for helping me construct and debug the PYTHON code in the appendix, and for explaining the PET.

I would also like to thank Ken Åman for his advice on the report, and for being supportive in general about my project.

Moreover, I would like to thank the people behind L\LaTeX\ and PYTHON for creating such powerful software, the former for efficient construction of scientific PDF’s, and the latter for enabling efficient computations whilst keeping the code simple.
8 Notations

Here we list the symbols used in the paper, along with their meanings.

<table>
<thead>
<tr>
<th>Notations</th>
<th>Meaning</th>
<th>Other notations</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathbb{Z}$</td>
<td>The set of all integers</td>
<td></td>
</tr>
<tr>
<td>$\mathbb{R}$</td>
<td>The set of all real numbers</td>
<td>$\mathbb{X} \times \mathbb{X}$</td>
</tr>
<tr>
<td>$\mathbb{X}^2$</td>
<td>The cartesian product of a set $\mathbb{X}$ with itself</td>
<td>$\mathbb{X}\times\mathbb{X}$</td>
</tr>
<tr>
<td>$\mathbb{A} \setminus \mathbb{B}$</td>
<td>The set containing all elements of $\mathbb{A}$ that do not occur in $\mathbb{B}$</td>
<td>$\mathbb{A} - \mathbb{B}$</td>
</tr>
<tr>
<td>$\mathbb{A} \subseteq \mathbb{B}$</td>
<td>$\mathbb{A}$ is a subset of $\mathbb{B}$</td>
<td></td>
</tr>
<tr>
<td>$\forall$</td>
<td>For all</td>
<td></td>
</tr>
<tr>
<td>$\langle S, \ast \rangle$</td>
<td>The group consisting of the set $S$ and the binary operator $\ast$</td>
<td></td>
</tr>
<tr>
<td>$\mathbb{Z}_n$</td>
<td>The group of all integers modulo $n$ under addition</td>
<td></td>
</tr>
<tr>
<td>$\langle a \rangle$</td>
<td>The cyclic subgroup generated by the element $a$ within an arbitrary group</td>
<td></td>
</tr>
<tr>
<td>$D_n$</td>
<td>The $n$'th dihedral group</td>
<td></td>
</tr>
<tr>
<td>$D_{x_1,x_2,x_3,...,x_n}$</td>
<td>The $x_1 : x_2 : x_3 : \ldots : x_n$'th dihedral group</td>
<td></td>
</tr>
<tr>
<td>$\circ$</td>
<td>Function composition</td>
<td></td>
</tr>
<tr>
<td>$i$</td>
<td>The identity operator</td>
<td></td>
</tr>
<tr>
<td>$r_n$</td>
<td>The operation equaling rotation by $n$ steps</td>
<td></td>
</tr>
<tr>
<td>$m_n$</td>
<td>The operation equaling reflection about the $n$'th axis</td>
<td></td>
</tr>
<tr>
<td>$d_n$</td>
<td>The operation equaling reflection about the $n$'th diagonal</td>
<td></td>
</tr>
<tr>
<td>$O(i)$</td>
<td>The orbit of $i$</td>
<td></td>
</tr>
<tr>
<td>$G(i)$</td>
<td>The stabilizer of $i$</td>
<td></td>
</tr>
<tr>
<td>$</td>
<td>G</td>
<td>$</td>
</tr>
<tr>
<td>$a^{-1}$</td>
<td>The inverse element of $a$</td>
<td></td>
</tr>
<tr>
<td>$\land$</td>
<td>Logical AND-operator</td>
<td></td>
</tr>
<tr>
<td>$\in$</td>
<td>in</td>
<td></td>
</tr>
<tr>
<td>$S_n$</td>
<td>The symmetric group on $n$ letters</td>
<td></td>
</tr>
<tr>
<td>$n!$</td>
<td>$n \cdot (n - 1) \cdot (n - 2) \cdots 3 \cdot 2 \cdot 1$</td>
<td></td>
</tr>
<tr>
<td>$\binom{n}{k}$</td>
<td>The number of $k$-combinations from $n$ objects</td>
<td></td>
</tr>
<tr>
<td>$\binom{n}{x_1,x_2,...,x_n}$</td>
<td>The number of $x_1, x_2, \ldots, x_n$-combinations from $n$ objects</td>
<td></td>
</tr>
<tr>
<td>$\sum_{i=0}^n a_i$</td>
<td>$a_0 + a_1 + a_2 + \ldots + a_n$</td>
<td></td>
</tr>
<tr>
<td>$\prod_{i=0}^n a_i$</td>
<td>$a_0 \cdot a_1 \cdot a_2 \cdot \ldots \cdot a_n$</td>
<td></td>
</tr>
<tr>
<td>${a_i}_{i=0}^n$</td>
<td>The sequence $a_0, a_1, a_2, \ldots, a_n$</td>
<td></td>
</tr>
<tr>
<td>$\zeta_G(x_1, x_2, \ldots, x_n)$</td>
<td>The cycle index of $G$</td>
<td></td>
</tr>
</tbody>
</table>

53
References


A   PYTHON-code using Burnside’s lemma

#!/usr/local/bin/python

"""This program utilizes Burnside’s lemma to find out the number of
inequivalent colorings of a given graph w.r.t. symmetry. The program will ask for
a file containing the symmetry group of the graph, and then for a partitioning
of colors. It will then perform the necessary counting to find the number of
inequivalent colorings.

Written by Arman Shamsgovara as part of a mathematical project regarding
symmetries in 2012/2013""

from Group_funcs import*
from copy import*

def permult(a,b):
    c=[i for i in a]
    for i in range(len(c)):
        c[i]=a[b[i]]
    return(c)

"""This function counts the sum of elements
in a list amount.""

def sum(amount):
    sumt=0
for i in amount:
    sumt+=i
return(sumt)

"""This function generates all possible colorings of a regular polygon via recursion. Input consists of a matrix in which to save the colorings, the polygon being coloured, the number of vertices in the polygon, and a list containing the number of corners in the current permutation that may be colored in a certain color.""

def permute(matrix, necklace, length, colors):
    # If we're done, we save the necklace in matrix, and go back.
    if len(necklace)==length:
        matrix.append(deepcopy(necklace))
        return(matrix)
    col=deepcopy(colors)
    # Check for each color if we can color another vertex
    for i in range(len(col)):
        if col[i]: # We can afford another vertex in this color
            # We color a vertex...
            necklace.append(i)
            col[i]-=1

            # ...we continue to the next vertex...
            matrix=permute(matrix, necklace, length, col)
...we go back, we take the color away, and we signal
# that we may use the color once more when generating
# other permutations.
col[i]+=1
del(necklace[-1])
return(matrix)

"""main starts here""
filename=input("I vilken fil finns symmetrigruppen sparad: ")
G=Generate_Group(filename) # G will hold all the group elements
amount=[] # The number of each color
k=int(input("Ange antal färger: ")) # The amount of colors

# We input the number of each color here
for i in range(k-1):
    amount.append(int(input("Ange antal kulor av färg {}: ",format(i+1)))))
amount.append(len(G[0])-sum(amount))

corners=sum(amount) # The number of corners
matrix=permute([], [], corners, amount) # All possible permutations

count=0 # The number of fixed positions
for g in G: # We count the number of fixed states w.r.t the elements of G here
    for state in matrix:
        if tuple(permult(state, g))==tuple(state):
            count+=1

print(count//len(G)) # The final result is printed
B  PYTHON-code using Cycle Indices to Compute Colorings

The following PYTHON-code extends the usability of the code in appendix A, since we can make it work with any group that we want. There is a catch though, and that is that we have to write down the group elements in their regular representation (as permutations) before we use the program, which can be very cumbersome if we want to study groups of order $> 10$. Therefore I have created a function that generates a group from it’s generators, so for e.g. the symmetry group of the edges of a cube three generators of the group elements can actually replace the total group of order 24, which is very helpful and timesaving.

A couple of example files will be shown in appendix C.

```python
#!/usr/local/bin/python

"""This program utilizes Polya enumeration to find out the number of inequivalent colorings of a given graph w.r.t. symmetry. The program will ask for a file containing the symmetry group of the graph, and then compute the group’s cycle index, a multivariate polynomial revealing the number of and the length of the cycles within the group. After that, the program will construct the generating function that holds the necessary coefficients that tell the number of ways the graph can be colored with $n$ colors. Last but not least, the program searches for the coefficient corresponding to the number of ways the graph can be colored with a desired number of each color you color it with.

Written by Arman Shamsgovara as a part of a mathematical project regarding symmetries in 2012/2013"""
```
from copy import *
from collections import *
from Group_funcs import *
from Polynomial import *

""" This function takes in a group element (permutation) g, and finds all cycles within it. """

def cycleVector(g):
    # Boolean array revealing which elements in the set of the group we haven’t visited yet
    k=[1 for i in g]
    toBeReturned=[]

    for i in range(len(g)):
        # We haven’t used this one yet...
        if k[i]:
            # The order (length) of the cycle of i is stored here
            currentlength=0
            # Dummy-index
            ind=i

            # Loop until we’ve gone 1 lap around the cycle
            while k[ind]:
                k[ind]=0
                ind=g[ind]
currentlength+=1

found=0

# Check if we already have found a cycle of order currentlength
for j in range(len(toBeReturned)):
    if toBeReturned[j][0]==currentlength:
        toBeReturned[j]=(currentlength, toBeReturned[j][1]+1)
        found=1
        break

# Create a new tuple if we haven't found that particular length yet
if not found:
    toBeReturned.append((currentlength, 1))

toBeReturned.sort()
return(tuple(toBeReturned),len(toBeReturned))

"""The generated dictionary is converted to the coefficient-free cycle-index of
the group computed with.
Input: a dictionary f, and an empty Polynom Poly
Output: the final cycle-index, stored in the Poly variable"""

def ConvertHashToPolynomial(f,Poly):
    while len(f):
        tup=f.popitem()
        VList=[Variabel(i[0], i[1]) for i in tup[0][0]]
newPT=PolynomTerm(tup[1], deepcopy(VList))
Poly=Poly+Polynom([newPT])
return(Poly)

"""We search for the term whose coefficient matches the colarr array""

def coefficientSearch(SuperPoly, colarr):
    for term in SuperPoly.terms:
        if len(term.variabler)==colors:
            found=1
            for variable in term.variabler:
                if not variable.exponent==colarr[variable.indice-1]:
                    found=0
                    break
            if found:
                # We're done
                print(term.koefficient)
                return

"""We construct the polynomial that gives us all possible ways of colouring
an object with colors different colors (the polynomial is a generating function).
Input:
- The cycle-index of a group (An instance of Polynom)
- The integer number of colours
- The integer order of the group in question""

def PolyaPolyFinder(CycleIndex, colors, order):
SuperPoly=Polynom()

# We create SuperPoly below
for term in CycleIndex.terms:

    # This will represent a polynomial created by the PolynomTerm term after the
    # swap of variables for polynomials
    res=1

    # Create a separate Polynom p for each variable in the (possibly)
    # cross-product PolynomTerm term, and multiply by res
    for vari in term.variabler:
        p=Polynom([PolynomTerm(1,[Variabel(col+1 , vari.indice)])
            for col in range(colors)])
        p=p**vari.exponent
        res=p*res

    # Add res to SuperPoly, and repeat the process with
    # the next term in the cycle-index (if any)
    SuperPoly=SuperPoly+res*term.koefficient

# We divide each coefficient in SuperPoly by the order of our group
for i in range(len(SuperPoly.terms)):
    SuperPoly.terms[i].koefficient//=order

return(SuperPoly)
# Input
filename=input("I vilken fil finns gruppen sparad? ")

# Use this on files with generating sets
T=Generate_Group(filename)

# Use this on files holding complete groups only
# G=Groupfinder('cube.txt')

# Classify all group elements g in our group according to their
cycles and the length of the cycles, and store the amount of times each
type of permutation has arised within a dictionary f

f={}
for g in T:
    tup=cycleVector(g)
    f.setdefault(tup,0)
    f[tup]+=1

# We find the cycle-index
CycleIndex=ConvertHashToPolynomial(f,Polynom())
print("Cykel-index: 1/{0:.0f}\n\{{1:s}\} = \{2:s\}\).format(len(T),\{{\}\},\{\}\).format(CycleIndex))

# Step two in the process
colors=int(input("Hur många färger? "))
This is the polynomial where we substitute all the variables of the cycle-index for polynomials representing the colors we want to handle

SuperPoly=PolyaPolyFinder(CycleIndex, colors, len(T))

Uncomment the following if you want to see the complete polynomial
(It will be very big in most cases, hence also hard to read)
print(SuperPoly)

Step three
We ask for the number of each color

colarr=[int(input("Hur många av färg {}? ".format(i+1))) for i in range(colors-1)]

sum=0
for i in colarr:
    sum+=i
colarr.append(len(T[0])-sum)

coefficientSearch(SuperPoly, colarr) # We search for the appropriate term
from copy import*  

"""Multiplication of permutations"""

def per_mul(a,b):
    return([b[i] for i in a])

"""This function generates all elements of a group by backtracking. 
Input: A group G, a list of generators gen and a current permutation 
Output: The generated group """

def Generate(G, gen):
    opened=set([tuple(i) for i in G])
    closed=set()

    while len(opened):
        h=opened.pop()
        closed.add(h)
        for g in gen:
            curr=tuple(per_mul(list(h),list(g)))
            if not curr in closed:
                opened.add(curr)

    G=[list(i) for i in closed]
    return(G)
In this function, the generating set of a group is read from a file with its name stored in filename. This function should be used for files with only the generators of a group within them.

The file should have the following content:

- Row 1: A number N, telling how many group generators are represented within the file
- Row 2 - N+1: a permutation of the numbers 1 to M, for some integer M (M constant for all lines 2-(N+1) in the file).

```python
def Generate_Group(filename):
    f = open(filename, "r")
    generators = []
    numofelements = int(f.readline())

    for i in range(numofelements):
        generators.append([int(p) - 1 for p in list(f.readline().split())])

    f.close()

    # Store all generators within a group matrix
    G = [copy(i) for i in generators]

    # Generate all other elements of the group
    G = Generate(G, generators)
    return(G)
```
In this function, a group $G$ is read from a file with its name stored in filename. This function should be used for files with complete groups within them. The file should have the following content:

- Row 1: A number $N$, telling how many group elements are represented within the file
- Row 2 - $N$+1: a permutation of the numbers 1 to $M$, for some integer $M$

(M constant for all lines 2-$(N+1)$ in the file). ""

```python
def Groupfinder(filename):
    f=open(filename,"r")
    G=[]
    numofelements=int(f.readline())
    for i in range(numofelements):
        G.append([int(p)-1 for p in f.readline().split()])
    f.close()
    return(G)
```
from copy import *

"""Variable class.
Attributes:
- An integer index
- An integer exponent
This class is needed for PolynomTerm and Polynom to work.""

class Variable:
    def __init__(self, ind, ex):
        self.indice=ind
        self.exponent=ex

"""For printing purposes""
    def __str__(self):
        if self.exponent==0:
            return("")
        elif self.exponent==1:
            return("((x_{})")
        else:
            return("((x_{})^{})")

"""A class representing a (possibly cross-product) term.
Attributes:
- A coefficient number
- A list of all the different variables multiplied to create this term

Operations:
- Addition
- Subtraction (somewhat superfluous)
- Multiplication (number with PolynomTerm or PolynomTerm with PolynomTerm)

Methods:
- Sorting of the variables according to their indices
- Comparison with another instance of the PolynomTerm to see if they differ only by their coefficients

This class is needed for Polynom to work.
This class inherits attributes from the Variabel class"""

class PolynomTerm:
    def __init__(self, coefficient=0, variabellista=[]):
        self.koefficient=coefficient
        self.variabler=copy(variabellista)

    def __add__(self, other):
        newT=PolynomTerm(self.koefficient, deepcopy(self.variabler))
        newT.koefficient+=other.koefficient
        return(newT)

    def __sub__(self, other):
        newT=PolynomTerm(self.koefficient, deepcopy(self.variabler))
        newT.koefficient-=other.koefficient
        return(newT)
return(newT)

def __mul__(self, other):
    newT=PolynomTerm(self.koefficient, deepcopy(self.variabler))

    if isinstance(other, int) or isinstance(other, float):
        newT.koefficient*=other
    return(newT)

    if isinstance(other, Polynom):
        newP=Polynom()
        for i in range(len(other.terms)):
            newP=newP+newT*other.terms[i]
        return(newP)

    newT.koefficient*=other.koefficient
    for i in range(len(other.variabler)):
        found=0
        for j in range(len(self.variabler)):
            if other.variabler[i].indice==self.variabler[j].indice:
                newT.variabler[j].exponent+=other.variabler[i].exponent
                found=1
                break
        if not found:
            newT.variabler.append(Variabel(other.variabler[i].indice,
                                            other.variabler[i].exponent))
return(newT)

"""For printing purposes"""

def __str__(self):
    if self.koefficient==0:
        return(" 0")
    if self.koefficient==1:
        res= " 
    else:
        res= " {}".format(self.koefficient)
    for i in self.variabler:
        res+=str(i)
    res+=" 
    return(res)

def equalInVariables(self, other):
    if not len(other.variabler)==len(self.variabler):
        return(0)
    for i in other.variabler:
        found=0
        for j in self.variabler:
            if jindice==iindice and j.exponent==i.exponent:
                found=1
                break
        if not found:
            return(0)
def sortForIndices(self):
    for i in range(len(self.variabler)):
        done=1
        for j in range(len(self.variabler)-1):
            if self.variabler[j].indice>self.variabler[j+1].indice:
                done=0
                (self.variabler[j], self.variabler[j+1])=
                (self.variabler[j+1], self.variabler[j])
        if done:
            break

"""A class representing a Power series (Polynomial).
Attributes:
- A list of the terms (instances of PolynomTerm) in the series
Operations:
- Addition
- Subtraction (somewhat superfluous)
- Multiplication (Polynom with Polynom, Polynom with number or Polynom with PolynomTerm)
- Exponentiation
This class inherits attributes from the Variabel and PolynomTerm classes"""
class Polynom:
    def __init__(self, termer=[]):
        self.terms=deepcopy(termer)

    def __str__(self):
        if len(self.terms)==0:
            return('0')

        res="" 
        for i in range(len(self.terms)-1):
            res+=str(self.terms[i])
            res="+"
        res+=str(self.terms[-1])
        return(res)

    def __add__(self, other):
        if isinstance(other, PolynomTerm):
            p=Polynom([other])
            return(self+p)

        newP=Polynom(deepcopy(self.terms))
        for i in range(len(other.terms)):
            #Check if the term already exists
            found=0
            for j in range(len(newP.terms)):
                if newP.terms[j].equalInVariables(other.terms[i]):
                    newP.terms[j]=newP.terms[j]+other.terms[i]
found=1
break
if not found:
    newP.terms.append(other.terms[i])

return(newP)

def __sub__(self, other):
    newP=Polynom(deepcopy(self.terms))
    for i in range(len(other.terms)):
        #Check if the term already exists
        found=0
        for j in range(len(newP.terms)):
            if newP.terms[j].equalInVariables(other.terms[i]):
                newP.terms[j]=newP.terms[j]-other.terms[i]
                found=1
                break
        if not found:
            newP.terms.append(other.terms[i])
            newP.terms[-1].koefficient*=-1
    return(newP)

def __mul__(self, other):
    if isinstance(other, int) or isinstance(other, float):
        newP=Polynom(deepcopy(self.terms))
        for i in range(len(newP.terms)):
            newP.terms[i]=newP.terms[i]*other
return(newP)

if isinstance(other, PolynomTerm):
    newP=Polynom()
    for i in range(len(self.terms)):
        newP=newP+self.terms[i]*other
    return(newP)

newP=Polynom()
for i in range(len(self.terms)):
    newP=newP+(self.terms[i])*other
    for i in range(len(newP.terms)):
        newP.terms[i].sortForIndices()
return(newP)

def __pow__(self, expo):
    if expo==1:
        newP=Polynom(deepcopy(self.terms))
        return(newP)
    else:
        p=self**(expo//2)
        p=p*p
        if expo%2==0:
            return(p)
        else:
            return(self*p)
E Example Files Containing Groups of Symmetry

To use the code in Appendix B, you need to write down the group you want to study in its regular representation, that is, you must translate all group elements to a row containing the integers 1,2,3,...,n, in some permuted order. You begin the file with an integer N telling how many permutations are saved in the file. Then, on the N following rows, you write down the bottom rows of the permutations corresponding to the group elements, one for each row.

If you have a large group, and do not want to write down all the elements, you may just write down the elements in the generating set of the group (and set N to be the amount of generators). Then you can use the <Generate_Group> function and let the computer generate everything for you. I strongly recommend the second way if you know the generators.

Example E.1. This is the content of the file for the full group of symmetries of $D_4$:

```
8
1 2 3 4
2 3 4 1
3 4 1 2
4 1 2 3
2 1 4 3
4 3 2 1
3 2 1 4
1 4 3 2
```

Example E.2. This is the content of the file for the generating set of $D_4$:

```
3
2 3 4 1
3 2 1 4
2 1 4 3
```
Basic Elementary Combinatorics

In this appendix, we briefly present some of the presumed preliminaries of this paper.

Definition F.1. The principle of multiplication
If $k$ choices are available at one instance, and there are $m$ choices at a second instance regardless of the choice at the previous instance, there are $km$ combined choices in total.

Definition F.2. We define the factorial of a positive integer $n$ as the product

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1$$

Definition F.3. We define the number of ways to choose $k$ objects from $n$ objects, regardless of order, as $\binom{n}{k}$ ("n choose k"). We define the value of this expression to be

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Theorem F.1. The binomial theorem

$$(x + y)^n = \sum_{i=0}^{n} \binom{n}{i} x^i y^{n-i}$$

Proof - The left hand side consists of several identical polynomials multiplied together. Each resulting term will be of the form $x^i y^{n-i}$, since the exponents must sum up to $n$. To achieve a specific term, we must choose $i$ polynomials to contribute with the $x$-factors, and then the $y$-terms fall out automatically. This can be done in $\binom{n}{i}$ ways. Thus the theorem is proven. $\square$
The binomial theorem can be generalized as follows:

**Definition F.4.** We define the number of ways to group $n$ objects into sets of sizes $a, b, c, \ldots, k$ ($a + b + c + \ldots + k = n$), regardless of order, as

$$\binom{n}{a, b, c, \ldots, k} = \frac{n!}{a!b!c!\ldots k!}$$

**Theorem F.2. The multinomial theorem**

The coefficient of the term $a^m b^n \ldots k^w$ ($m+n+\ldots+w = z$) in the expansion of $(a+b+\ldots+k)^z$ is

$$\binom{z}{m, n, \ldots, w}$$

**Proof** - The proof is similar to that of the binomial theorem, but here we have to choose $m$ $a$-terms, $n$ $b$-terms, and so on, from $z$ polynomials, thus the coefficient of the $a^m b^n \ldots k^w$-term must be

$$\binom{z}{m, n, \ldots, w}$$