

# **CHANGE RINGING: WHERE MATH AND MUSIC MEET**

Eine Maturitätsarbeit an der  
KANTONSSCHULE LIMMATTAL

vorgelegt von  
**CAROLINE DULAY**

Klasse M6i  
im Fach Mathematik

betreut von  
**Andreas Pfenninger**

## Zusammenfassung

In dieser Arbeit wurde die englische Tradition des Change Ringings (Wechselkläutens) historisch, mathematisch und anhand eines Python-Programms untersucht. Im Change Ringing werden Glocken von einer Gruppe Glockenläuter\*innen nach einem bestimmten Schema geläutet, sodass alle möglichen Reihenfolgen der Glocken genau ein Mal vorkommen. Dabei können die einzelnen Glocken von einer Reihenfolge zur nächsten nicht mehr als eine Position in der Reihe nach vorne oder nach hinten verschoben werden. Idealerweise wechseln alle Glocken regelmässig ihre Position in der Reihe, damit es für die Glockenläuter\*innen spannend bleibt. Dieser Brauch wurde im 17. Jahrhundert entwickelt und wird bis heute praktiziert. Change Ringing ist ein Schnittpunkt der Musik und der Mathematik und beide Aspekte werden hier betrachtet. Schliesslich wird beschrieben, wie ein Programm, welches alle 24 erlaubten Sequenzen von Reihenfolgen für vier Glocken finden kann, programmiert wurde.

## Abstract

In this project, the English tradition of change ringing is explored historically, mathematically, and with Python programming. Change ringing is the ringing of church bells by a group of ringers in a prescribed pattern, so that a set of bells is rung in all possible sequences, without moving the position of any bell in the sequence more than one position forwards or backwards. Ideally, all bells change positions regularly in order to keep the sequence interesting for the bell ringers. This tradition originated in England in the 17<sup>th</sup> century and is still practiced today. Change ringing sits at the intersection of music and mathematics, and both of these aspects will be explored here. In addition, the process of creating a Python program that was able to find the 24 valid change ringing sequences for a set of four bells is described.

# Contents

<b>Abstract</b>	<b>i</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 History</b>	<b>2</b>
2.1 A Short History of Bells . . . . .	2
2.1.1 The Earliest Bells in Asia . . . . .	2
2.1.2 Beginnings of Bells in Europe . . . . .	2
2.1.3 Bells in Early Christianity . . . . .	2
2.1.4 Bells on the British Isles . . . . .	3
2.2 Bells in the Middle Ages . . . . .	3
2.2.1 Casting and Hanging Bells in the Middle Ages . . . . .	3
2.2.2 Importance of Bells in the Middle Ages . . . . .	3
2.2.3 St. Dunstan of Canterbury . . . . .	3
2.3 Bells in the Modern Day and Other Uses for Bells . . . . .	4
2.3.1 Baptism of a Bell . . . . .	4
2.3.2 Carillons . . . . .	4
2.3.3 Decorations and Inscriptions on Bells . . . . .	5
2.3.4 Bell Demise . . . . .	6
2.4 The History and Practice of Change Ringing . . . . .	7
2.4.1 The Beginnings of Change Ringing . . . . .	7
2.4.2 Basic Ringing . . . . .	8
2.4.3 Methods . . . . .	9
2.4.4 Change Ringing Societies . . . . .	9
2.4.5 Is Change Ringing Music? . . . . .	10
<b>3 Mathematics</b>	<b>11</b>
3.1 Basic Definitions of Change Ringing . . . . .	11
3.2 Excursion into Group Theory . . . . .	12
3.3 Graphical Illustration . . . . .	15
3.3.1 Examples for 4 Bells . . . . .	16
3.4 Finding Hamiltonian Cycles . . . . .	19
<b>4 Programming</b>	<b>20</b>
4.1 Beginning to Program . . . . .	20
4.2 Programming Changes “by Hand” . . . . .	21
4.2.1 Change on 3 Bells . . . . .	21
4.2.2 Change on 4 Bells (Reverse Canterbury) . . . . .	21
4.3 Random sequences . . . . .	23
4.3.1 Function to Swap Items: <code>swap(pos)</code> . . . . .	23
4.3.2 Choose Random Swap: <code>randomswap(possible_swaps)</code> . . . . .	23
4.3.3 List to Word: <code>convert(s)</code> . . . . .	24
4.3.4 Main Loop . . . . .	24
4.3.5 Checking Whether a Sequence is an Extent . . . . .	24
4.3.6 Output . . . . .	25
4.4 Program to Find Valid Extents on 4 Bells . . . . .	26
4.4.1 Function to Test Rows: <code>test_rows</code> . . . . .	26
4.4.2 Function to Build Extents: <code>extend_rows</code> . . . . .	26
4.4.3 Main Loop to Build Extents . . . . .	28
4.4.4 Output . . . . .	28
4.4.5 Audio . . . . .	30
<b>5 Discussion and Conclusions</b>	<b>31</b>
<b>6 References</b>	<b>32</b>
<b>List of Tables</b>	<b>34</b>

<b>List of Figures</b>	<b>34</b>
<b>Acknowledgments</b>	<b>35</b>
<b>Einhaltung rechtlicher Vorgaben</b>	<b>35</b>
<b>A Appendix: Full Process of Programming</b>	<b>36</b>
A.1 Starting With Changes on 3 Bells . . . . .	36
A.1.1 Printing a List . . . . .	36
A.1.2 Swapping the Positions of Two Items in a List . . . . .	36
A.1.3 Extent on 3 Bells . . . . .	36
A.1.4 Extent on 3 Bells With Loop . . . . .	36
A.2 Programmed Extents on 4 Bells . . . . .	37
A.2.1 Plain Bob . . . . .	37
A.2.2 Reverse Canterbury . . . . .	37
A.2.3 Double Court . . . . .	38
A.3 Generating a Random Set of Changes . . . . .	38
A.3.1 Create Change on 3 Bells . . . . .	38
A.3.2 Creating a Random Set of Changes on 3 Bells . . . . .	39
A.3.3 Random Changes on 4 Bells of a Random Length . . . . .	40
A.4 Extents on 4 Bells . . . . .	41
A.4.1 Program Finds All Possible Extents on 4 Bells . . . . .	41
A.4.2 Program Finds Extents on 4 Bells With Sounds . . . . .	43
A.4.3 Output Printing All Extents . . . . .	45

# 1 Introduction

For this project I wanted to research something that combined my interests in music and mathematics. My uncle, who double majored in mathematics and music and studied in England for part of his studies, suggested the topic of change ringing and I thought it sounded interesting. Change ringing is the English art of ringing church bells in defined sequences. It originated in England around the 17<sup>th</sup> century, when the technology for hanging bells developed so that a single bell could be rung exactly once with one pull on a rope. Bell ringers became interested in ringing bells in different patterns. One sequence of bells is called a **change** and for  $n$  bells there are  $n!$  different permutations, or ways to arrange the bells in a sequence. For the ringers the next step was to arrange the changes in such a way that they could ring all the possible permutations of their  $n$  bells consecutively. Starting with the bells in descending order of pitch, a special change called **rounds**, they swap two consecutive bells in the sequence to create a new change. They do this until they have gone through all the ways to order the bells and come back to rounds with the bells in order of descending pitch. A cycle of all the permutations is called an **extent** and change ringers have found various **methods** to order the permutations for  $n > 3$ . [1]

In this project I investigated the questions:

- What is the history of change ringing?
- How can change ringing be mathematically described?
- How can the changes be generated by a computer program?

Due to the interdisciplinary nature of this paper, I used many different sources. The main source for the mathematical section is a bachelor thesis *Campanology - Ringing the changes*, written by Fabia Weber, a student at the ETH Zürich, supervised by PD Dr. Lorenz Halbeisen [1]. It is a deep dive into the group and graph theory related to change ringing. For the historical section, I used a paper on the history of bells in Europe [2] for the early history; for the more general history and the description of change ringing itself I used multiple books on ringing: *The History and Art of Change-Ringing* [3] and *Church Bells and Bell-Ringing* [4]. In addition to these books, the films *Still Ringing After All These Years: A Short History of Bells* [5] and *The Craft of Bellringing* [6] were very helpful. In the programming part, I consulted various online fora and internet pages. In particular I used the website GeeksforGeeks to help me construct my Python code.

This paper is written in English because change ringing is an almost exclusively English art, so sources on change ringing and the technical terms used in change ringing are all in English. In addition, my mathematical education has been in English for the last two years, because I am in an immersion class. Therefore it was easier for me to look for mathematical information in English as well.

## 2 History

### 2.1 A Short History of Bells

#### 2.1.1 The Earliest Bells in Asia

Bells have existed for such a long time that it is hard to imagine a world without them. They are classified as idiophones: Instruments which sound through vibration of the instrument itself, without the help of strings, membranes or columns of air [7]. Bells are likely descendants of early rattles, which were first of natural origin, such as shriveled fruit with seeds, but were later made artificially by filling dried, hollow pouches, such as dried animal bladders, with seeds or small pebbles. Eventually rattles, and later bells, were made of ceramic. [2]

Both the oldest ceramic rattles and the oldest ceramic bells that have been studied by archaeologists were found in China and date to the early third millennium BC. After ceramic bells come bells made out of bronze, which might have been a result of bronze pots being used as musical instruments in religious rituals. Larger bells, which were rung by striking the outside wall, and small bells, which were rung with a bell tongue inside the bell, came to be used for a variety of settings, from animal herding to the high courts. [2][8]

The first bronze bells were hammered from a flat sheet of copper. Later, the Chinese learned to cast bells out of liquid metal and to tune them to ring particular musical tones. The bell makers continued to perfect their craft and soon a bell was as much a work of art as it was a musical instrument. The knowledge of bell making also started to spread to cultures westwards of China. [8][3][2]

#### 2.1.2 Beginnings of Bells in Europe



Figure 1: Bell from Herculaneum, 1st century AD cf. [2], figure 10

By the second half of the first millennium BC, the people in the Greco-Roman cultural area had bells called *tintinnabulum* in Latin that were used for purposes such as rituals and household signaling. In the household, they were used for calling the servants, waking the slaves and as doorbells. There were also larger bells called *aes* that were used for ritual purposes and were believed to have mystical powers, such as being able to break spells and scare away evil forces. The oldest hand bell from this period was found while excavating Herculaneum, one of the cities buried under the ashes of the Vesuvius in 79 AD (see figure 1). [2]

#### 2.1.3 Bells in Early Christianity

When Christianity became the official religion of the Roman Empire in the 4<sup>th</sup> century AD, rituals and traditions were just being formed and there were many local variants of Christianity. Some took inspiration from local ceremonies and etiquette or from pagan rituals. Hand bells were used by western Christians during funeral processions, just as the ancient Greeks had done. Another use for bells was calling the people together for prayer: These were not the church bells we are used to today, but rather one person running through the streets ringing a small bell. [2]

As liturgical forms became standardized in the 6–7<sup>th</sup> century AD, the local variants started to disappear. The first use of bell-like instruments in the liturgy was probably in the 6<sup>th</sup> century. There is a description of a bell hanging from the ceiling of Tours Cathedral near the altar of St. Martin, which is thought to have been used in a similar way to an altar bell today<sup>1</sup>. Bells were also used to mark the division of the monastic day into canonical hours, each of which began with a prayer. [2]

<sup>1</sup>the altar bell is used “to draw attention to the precise moment when transubstantiation – the conversion of the bread and wine into the body and blood of Christ – takes place” [9]

### 2.1.4 Bells on the British Isles

St. Patrick is associated with bringing hand bells to Ireland. For him, bells were an important part of being a missionary. They were primarily used to call believers together and he would give his disciples a bell when he sent them as missionaries to a new area. These bells, called Celtic hand bells, consisted of two pieces of metal for the body with a handle on top (as seen in figure 2). As Christianity spread, these bells then made their way to England with the missionaries. They were thought to have special powers, like being able to terrify demons and get rid of storms. Those of holy men were even carried into battle as a sign of God's protection. [2][5]



Figure 2: The Cloc ind Édachta, the bell of St. Patrick, said to be the oldest preserved bell from Ireland [10]

## 2.2 Bells in the Middle Ages

### 2.2.1 Casting and Hanging Bells in the Middle Ages

As the missionaries settled down and built abbeys, there came a need for larger and more resonant bells [5]. Medieval bell founders used methods the Romans had used to cast bronze statues to cast their bells. First, a wax model was created and a mold was created around it. The molds were then fired to melt the wax out, and bronze was poured into the mold in its place. When the bronze bell had cooled, it could be removed from the mold and tuned. These cast bells were not very large at first: The largest bell from the early period had a diameter around 30 cm. These bells may have, for example, been hung on the wall of a church and rung by means of a rope. [11]

The exact uses for bells in this period remain unknown, partly due to the myriad of words in medieval texts that might indicate the use of a bell. The most common word for bell is *campanum*, but the words *clocca*<sup>2</sup> and *glocca*, especially in Irish texts, and *tintinnabulum*, adopted from the Romans, are also used [2]. Another word, *signum*, which translates directly translates to “sign” might also indicate the use of a bell. [11]

### 2.2.2 Importance of Bells in the Middle Ages

Bells grew in size and importance and by the 12<sup>th</sup> century, bell towers were commonplace [11]. In large churches, there were even multiple bells. These bells were used in the Church for many reasons. There was usually a *sermon bell*, a signal to the congregation that a sermon was about to start. The *Sanctus bell* was rung during the Sanctus<sup>3</sup> and at other important times during the liturgy [14]. The *Passing bell* was sounded to call people to pray for the departing soul of a neighbor and was followed by the *death knell*, the bell announcing their death. [4]

Bells also had secular functions, for example the *market bell* regulated the hours of business, the *curfew bell* indicated the start of curfew and the *fire bell* warned of a fire in town [4]. Another interesting bell is the *pancake bell*, which was rung on Shrove Tuesday, the day before Ash Wednesday, and which some interpret as a signal to stop work and prepare pancakes [15]. Additionally a landowner could increase his social rank by building a church with a bell tower on his land [3].

### 2.2.3 St. Dunstan of Canterbury

St. Dunstan, the patron saint of bell ringers, was a prominent religious figure in the 10<sup>th</sup> century. Born in Baltonsborough, south of Bristol, he was a gifted craftsman and scholar and became an advisor to the court of King Athelstan, who is regarded as the first King of England. Later he decided to become a monk at the Abbey of St. Mary in Glastonbury. While there, he used his (relatively large) inheritance to improve the church and gain influence in the kingdom: Eventually he became Archbishop of Canterbury,

<sup>2</sup>This is a stem of the word clock, derived from bell, because bells told the time before the invention of clocks [12]

<sup>3</sup>“an ancient Christian hymn of adoration sung or said immediately before the prayer of consecration in traditional liturgies” [13]



effectively the leader of the Church in England. His connection to bells comes with his interests in music and knowledge of metalworking, which he used to experiment with the design and methods of casting bells. He was canonized after his death and his other patronages include blacksmiths, goldsmiths, locksmiths, musicians, and silversmiths. [16]

## 2.3 Bells in the Modern Day and Other Uses for Bells

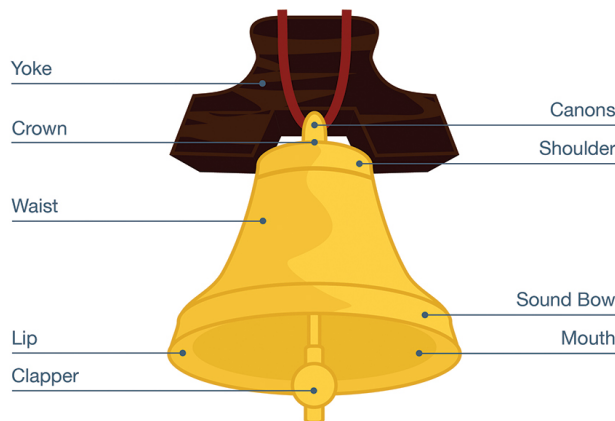


Figure 3: The parts of a church bell [17]

Church bells today bear a strong resemblance to the bells of the Middle Ages (see figure 3). The bells rung before services today are descendants of the sermon bell. In some churches, there are also still bells, usually hand bells, which are still rung during the transubstantiation<sup>4</sup>. Even though we now have clocks to tell the time, many bell towers still ring quarter and full hours. In Switzerland, the ringing of bells is regulated locally. In the city of Zürich, for example, the bells ring at 7 o'clock in the morning, at 11 o'clock for lunch and at dusk in the evening [19]. The earliest that bells still ring in the morning is in Maschwanden, Canton Zürich, where they ring at 4:45 a.m. for 3 minutes every morning but Sunday [20].

### 2.3.1 Baptism of a Bell

Another tradition that survives to this day in the Roman Catholic Church is the tradition of baptizing a new bell. This tradition was originally connected to the bell's supposed magical power: Bells were said to be able to ward off evil spirits and storms, which they clearly could not have done without the Church's blessing. For the baptismal ceremony, the bell is hung on a low frame so the bishop can walk all the way around it. The bishop blesses the bell with holy water, salt and holy oil and psalms are chanted. In some places the bells even have godmothers and godfathers. After the ceremony there is a community celebration. [21]

### 2.3.2 Carillons

In some places, sets of bells are hung together to form a carillon. A carillon is a keyboard instrument built into a bell tower that plays bells instead of plucking a string like a harpsichord or pushing air through a pipe like an organ. In a carillon, there are at least two octaves of tuned bells tuned chromatically and their clappers are attached to ropes, which are connected to the keyboard. The keyboard is set up in a similar way to an organ but instead of the keys as one would find on a piano or an organ, the keyboard is made up of pegs that are struck with the bottom of a closed fist. As with an organ there can also be a pedal keyboard, which controls the lowest octave of the top keyboard. [22]

In Switzerland, the canton of Wallis has kept another type of carillon alive. The *Walliser Carillon* has only on average 3 - 6 bells, which can swing back and forth and also have a rope attached to their clappers. The ropes are threaded into the ringing chamber, so the bells can be rung by one ringer with foot pedals, their hands and even their elbows. One bell was often swung along as a bass accompaniment, rung by a board mounted on the stock (yoke) of the bell on which ringers can stand to make the bell swing. [23]

<sup>4</sup>The part of the mass where "the Eucharistic elements at their consecration become the body and blood of Christ while keeping only the appearances of bread and wine" [18]

### 2.3.3 Decorations and Inscriptions on Bells

Every large bell is a hand crafted work of art, so it is common to give it an individual identity. Commonly a bell will be marked with its name, the emblem or initials of its founder, and a prayer or verse from the Bible. Bells were also commonly decorated with patterns on the rim, shoulder and crown. [21]

Figure 4 shows a bell from the Evangelical Reformed Church in Maschwanden, Switzerland, whose sound I used in the programming part of this project. On the rim the name of the founder, Keller, and the place of founding, Unterstrass, can be seen. On the waist is a verse from Psalm 22. Below this quote there is a wreath of flowers and above it is an angel to protect the bell. On the shoulder of the bell there are elaborate leaf designs and the canons have detailed faces of old men on them.



Figure 4: The largest in the belltower of Maschwanden, Switzerland

Inscription:

ICH WILL DEINEN NAMEN  
MEINEN BRÜDERN VERKÜNDEN  
INMITTEN DER GEMEINDE  
WILL ICH DICH LOBEN

PS. 22. 23.

("I will declare your name to my people; in the assembly I will praise you." [24])

### 2.3.4 Bell Demise

Even bells do not last forever. The oldest bells still in use are hundreds of years old, but not all bells are so lucky.

After King Henry VIII decided to break ties with the Catholic Church in Rome, the new Church of England saw many new reforms. Thomas Cromwell, who carried out these reforms, declared the monasteries unnecessary; the official excuse was that the monks and nuns were corrupt and immoral and were not helping the poor as they were supposed to. In reality, the King's finances were not doing very well and with the dissolution of the monasteries he was able to appropriate their property for himself. In this process many monastic bells were melted down or given or sold to churches in the area. [5][4][25]

More dramatically, many bells were lost during the wars in the 20<sup>th</sup> century. The fate of bells in the First World War varied by country. Because metals were essential for making guns, in March 1917 a decree was issued by the government in Germany that bells were to be collected and melted down. For many, this act of turning the bells into artillery was very disturbing, because it gave the bells quite a different meaning. In the First World War around 44% of German bells were melted down. In Great Britain, regulations were introduced limiting bell-ringing to an act of warning. Some say that this silence was deafening, but the end of the war brought back the bells in all their glory. On the day of the defeat of the Germans, November 11<sup>th</sup> 1918, the bells across Great Britain tolled to mark the end of the war. Some bells in Russia were taken from their towers to be stored at Nikolsky Monastery near Moscow. Even though this silenced them, they were kept safe for the duration of the war. [26][27]

In the Second World War, the National Socialist German Workers' (Nazi) Party confiscated the bells not only in all its occupied territories, but also in Germany. The seized bells were graded with a system ranging from A to D. The bells ranked A had been cast most recently and were melted down first, and the bells categorized B and C soon followed. But the bells in category D, cast before 1740, were spared and treated like art. In addition, each bell tower was left with one bell to ring in an emergency. In total around 175'000 bells were seized and of those it is estimated that 150'000 ( $\frac{6}{7}$  of the bells seized) were melted down. Some communities buried their bells before the Nazis took them so that they could not be melted down and some of these are still being found to this day. [29][27]



Figure 5: Glockenfriedhof in Hamburg's harbor [28]

In Italy, a prewar agreement had been made with the Vatican that sought to protect half of the bells in church towers, but the other half was therefore not safe from being taken. But as in Germany, each bell tower was left with one. The confiscated bells were broken up in Italy and sent to Hamburg, because the Italian smelting plants did not have enough capacity. The two largest refineries were in Hamburg along with many collection points for bells, known as bell cemeteries or *Glockenfriedhöfe*. [29][27]

With carillons being very important musical instruments in both Belgium and the Netherlands, there were a lot of bells to take. A carillon consists of at least 23 bells and in the First and Second World Wars, many of them were destroyed. It is said that two thirds of the bells in Belgium were taken by the Nazis. As if confiscating the bells were not enough, the Nazis commissioned small bells from foundries in the Netherlands bearing the inscription: "the bells too are fighting for a new Europe." These bells were then given to the leading Nazis who had played key roles in the seizures. [27]

Somehow the bells of Norway, Denmark and Luxembourg remained mostly untouched. In France, the bells were deemed cultural heritage and the Vichy government offered the country's bronze statues instead. Due to bombing raids and other attacks, even some French bells did not survive, but many made it through the war. In Britain, the decision was made that church bells should be rung only in the case

of a German invasion. After an allied victory at El Alamein, Churchill ordered all the church bells of England to ring out. [27][5]

## 2.4 The History and Practice of Change Ringing

### 2.4.1 The Beginnings of Change Ringing

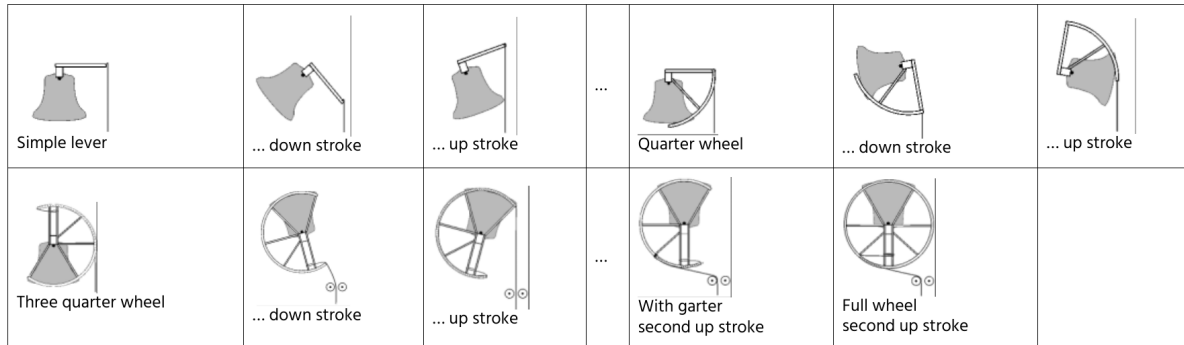


Figure 6: The evolution of methods of hanging bells and up and down strokes on the different mounts [30]

Around the time of the English Reformation, the technology for hanging bells was also changing. When the monastic bells were relocated, they were hung on newly invented quarter and half wheels instead of on simple levers (see figure 6). This made them easier to ring, but bell ringing still required a great deal of strength. In addition, a bell could be started and kept in motion by its ringer, but bells could not be rung in a specific sequence. With the introduction of full wheels (see figures 6 and 7), what is now known as change ringing became possible. A bell on a full wheel can be rung deliberately and, with the help of the stay on the wheel (see C in figure 7), can pause in an upside-down position to wait for the next stroke. [3] The rope that controls the bell ringing is guided by a wheel, which serves as a track into which the rope winds and unwinds as the bell swings. On the side opposite the wheel is the stay, which rests against a slider when the bell is in the upside-down position. The stay is intentionally the weakest link in the system: If the rope is over-pulled, the stay will break, but can be replaced easily.

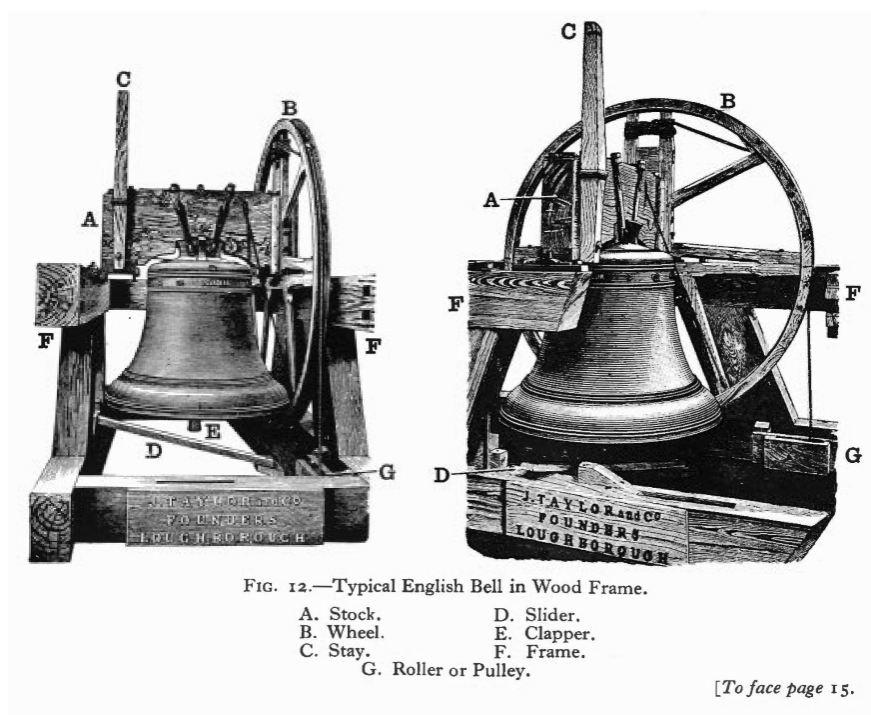


Figure 7: Bell hung on a full wheel in a wooden frame [3]

In change ringing, a set of bells is hung on full wheels on a stock mounted in a frame of wood or metal. Each individual bell rope gets threaded through a hole in the belfry, where the bells are hung, to a ringing chamber below. The bells are always hung so that they swing in different directions because if they swung together, the force they created while swinging would be enough to rock and even break the bell tower. When they face in different directions, they still exert forces on the tower, but not all on one axis, so there is less danger to the tower, bells, and ringers. [3][6]

### 2.4.2 Basic Ringing

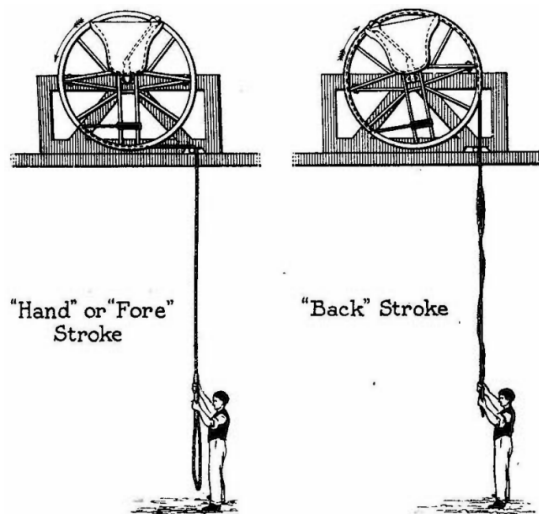


Figure 8: The two strokes, that can be used to ring an upset bell [3]

In order to ring changes, bells first have to be set in an upside-down position. This is accomplished by **ringing up** the bell, letting its momentum carry it higher and higher with every stroke. There are two strokes, as seen in figure 8, that can be done with a full wheel: The hand stroke, which is the stroke in which the rope is uncoiled from the wheel and hangs in a loop at the end; and the back stroke, in which the rope re-coils onto the wheel and the ringer is only holding the end of the rope in his hands at the end. Each stroke results in one ring of the bell, and the ringer can influence the timing of the rings based on the timing of the rope pulls. This control, however, has its limits: During change ringing, the bell does not come to rest against the stay, but stays in motion as the rope on the wheel coils and uncoils. [3][4][6]

In change ringing, each bell is rung by a single ringer. The bells are all numbered with the highest pitched bell, the **treble**, being number 1. The lowest bell is named the **tenor** and is bell  $n$  in a group of  $n$  bells. The first thing a new ringer learns after learning how to control their bell is how to ring **rounds**: The ringers coordinate their movements so that the bells are rung in order from the highest- to the lowest-pitched bell (see definition 3.1.1). [1][3]

After learning to ring rounds, a ringer learns to ring **call changes**. Due to the momentum of the bells, a bell can only change positions with the bell rung directly before or directly after itself. When the bell ringers ring call changes, one of the ringers acts as a kind of conductor, telling the other ringers when a change is to be made in the ringing order. Usually, each change is repeated multiple times before the next one is called. [31]



### 2.4.3 Methods

Finally, ringers learn **methods** that connect sequences of changes in a systematic way. For example, one of the easiest ways of moving a bell through a set of changes is **plain hunting**.



Figure 9: Blue bell plain hunting

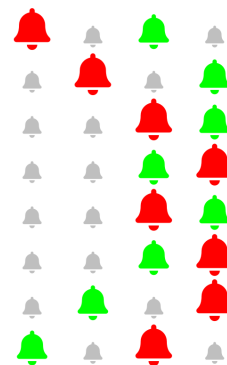


Figure 10: Green and red bell dodging

A bell is **hunting up**, when it moves in successive changes from the first to the last place and **hunting down**, when it moves back to the start. In figure 9, the blue bell hunts up first and then hunts down. It stays in the same place for one change in the middle, because here two of the other bells would have to switch to create a new permutation in the next change. Two bells **dodge** with each other when they meet while hunting in opposite directions and change places three times before resuming their hunting (see figure 10) [31]. [3]

Methods are used to ring **extents**, or sequences in which all possible permutations of  $n$  bells are played. If the number of bells is larger than 7, it is rather inconvenient (as seen in table 1) to ring full extents: For example to ring the 3'628'801 changes in an extent on 10 bells would take 84 days. Instead, **peals** of at least 5000 changes are rung. The most common number of bells is eight, with the first seven ringing changes and the tenor always striking last. The ringers learn these methods by heart and play them without any sheet music or other aids. When ringing they then have to know their place in the next change and remember who should be ringing before them. They have to see the other bell ringers in the room, who could be next to or across from them. This skill is called **ropesight** and is developed over time. Someone with good ropesight can monitor all the ropes at once while ringing their own part. [4][6][21]

Interestingly, it took less than a century for the first notions of change ringing to develop into an established art form. According to the first published set of rules for change ringing, *Tintinnalogia*<sup>5</sup>, which was published in 1668, change ringing had been developed in fifty or sixty years before its release [32]. In the book, a number of different methods to ring bells are described. Just nine years later the man who printed *Tintinnalogia*, Fabian Stedman, wrote his own book on ringing. *Campanalogia*<sup>6</sup> is Stedman's improvement of *Tintinnalogia*, with more peals and easier explanations of the topic [33]. The ground rules and methods laid out in these books are still followed and rung today [5].

### 2.4.4 Change Ringing Societies

Change ringers often organized themselves in societies and guilds. These societies keep track of the peals rung by their members and organize events. One of the oldest and most prestigious societies is The Ancient Society of College Youths. It has existed since 1637. Fabian Stedman, the author of *Tintinnalogia*, was a member of this society. [21][34]

Today, change ringing societies exist mainly in England, but groups of ringers also practice in other parts of the world. Many older people take up ringing after retirement, but there are also groups of younger change ringers at many universities. For many, change ringing is also a source of friendships and

<sup>5</sup>Full title: *Tintinnalogia, or, the Art of Ringing (Wherein is laid down plain and easie Rules for Ringing all sorts of Plain Changes. Together with Directions for Pricking and Ringing all Cross Peals; with a full Discovery of the Mystery and Grounds of each Peal. As Also Instructions for Hanging of Bells, with all things belonging thereunto.)*

<sup>6</sup>Full title: *Campanalogia: or The art of ringing improved (With plain and easie Rules to guide the Practitioner in the Ringing all kinds of Changes, to which is added, great variety of new peals.)*

community. The Central Council of Church Bell Ringers, an umbrella organization for change ringing societies, support and encourage ringing and has many resources available for new and experienced ringers alike. [5][6][35]

#### 2.4.5 Is Change Ringing Music?

Is change ringing music? This is an interesting philosophical question with some depth. I will try to answer it briefly here.

In R. Murray Schafer's book *The Soundscape* [36], two fundamental concepts of music are defined: The *Apollonian* and the *Dionysian* concepts. The Apollonian concept focuses on the sonic properties of the world, looking at music from an external point of view, as if it were a reminder for humans of the harmony of the universe sent by the gods themselves. In the Apollonian concept, music is basically organized sound. From this point of view, music is exact, mathematical and serene, and it is associated with the concept of a utopia or a harmony of the spheres. This concept is the basis of Pythagorean tuning and Schönberg's twelve-tone music. In the Dionysian concept, on the other hand, music is portrayed as a subjective emotion. Here music is an irrational concept stemming from humans themselves. The Dionysian concept is the basis for the musical expression one can find in the Romantic period, with the many expressive devices used from this period onward such as dynamics, tempo fluctuations and tonal coloring. In the Dionysian concept, music can be anything if it is put into the context of a musical performance meant to convey the emotions of the performers to a listening audience. [36]

Most things we think of as music meet both definitions: Both a Taylor Swift concert and a Bach Passion require both technical competence and emotional investment from the performers. Change ringing, in contrast, fits only one of these two concepts. It fits perfectly into the Apollonian definition of music: It is organized and exceedingly mathematical, as I will show in the next section. It does not, however, meet the Dionysian concept: Change ringing does not convey emotions and a listener would not go to the ringing of a peal to listen to it as music. The goal is more to work together to ring a logical structure rather than to perform for an audience. Change ringing is therefore both music and not music.

### 3 Mathematics

#### 3.1 Basic Definitions of Change Ringing

**Definition 3.1.1.** In change ringing we have  $n$  bells, where  $n \geq 2$ . The bells are numbered 1 to  $n$  and ordered by pitch in descending order. The bell with number 1, being the highest pitched bell, is called the **treble**. The lowest pitched bell,  $n$ , is known as the **tenor**. A **change** is the ringing of the  $n$  bells in a particular order or arrangement. In other words, a change corresponds to the ringing of a permutation of  $[n]$ , the ordered set containing all integers from 1 to  $n$  in their natural order. The special change 1 2 3 ...  $n$ , in which the bells are rung in order of descending pitch, is called **rounds**. The first bell of a change is defined as the **lead**. *cf. [1], Definition 1.1.1*

**Definition 3.1.2.** An **extent** consists of  $n! + 1$  successive changes, satisfying:

- (i) the first and last change are both rounds;
- (ii) no other change is repeated (so that each possible change other than rounds is rung exactly once);
- (iii) from one change to the next, no bell moves more than one position in its order of ringing (positions 1 and  $n$  are not adjacent, unless  $n = 2$ );
- (iv) no bell rests in the same position for more than two successive changes;
- (v) each bell (except perhaps the treble) does the same amount of 'work' (hunting, plain hunting, dodging, etc.);
- (vi) The method employed is palindromic, or self-reversing

Conditions (iv) and (v) are occasionally relaxed in practice, but the other conditions are inviolable. *cf. [37], 1. Introduction*

The different conditions are based on different concepts that change ringing has to fulfill. The first is for musicality, the second for thoroughness and the third is based on the mechanical limits of bell ringing (see 2.4.2, Basic Ringing). The fourth and fifth rules are in place to keep things interesting for the ringers and the last rule follows from the fact that an extent is a cycle that can also be played in the opposite direction. [37]

The majority of bell towers home to change ringers have a number of bells  $n$  ranging between 3 and 12. The maximum number of bells ever to have been rung in change ringing seems to be 16. The odd-bell extents receive names based on the maximum number of pairs of bells that can be interchanged in one transition and the even-bell extents are named from smallest (4 bells, Minimus) to biggest (12 bells, Maximus) in Latin [38]. [1]

Table 1: Assuming that 30 changes can be rung per minute, we obtain in the rightmost column the time required to ring a given extent. *cf. [1], Table 1.1*

number of bells $n$	name	changes in extent $n! + 1$	time required to ring extent
3	Singles	7	14 seconds
4	Minimus	25	50 seconds
5	Doubles	121	4 minutes
6	Minor	721	24 minutes
7	Triples	5'041	2 hours 48 minutes
8	Major	40'321	22 hours 24 minutes
9	Caters	362'881	8 days 10 hours
10	Royal	3'628'801	84 days
11	Cinques	39'916'801	2 years 194 days
12	Maximus	479'001'601	30 years 138 days
16		20'922'789'888'001	1'326'914 years

So much time is required to ring an extent on over 7 bells, that a full extent on 8 bells has only been rung once without changing the ringers during the extent, which is quite a feat. Due to the large amount of time required to ring the extents, ringers of large numbers of bells focus on ringing peals instead.



**Definition 3.1.3.** Let  $n > 7$ . A **peal** is composed of at least 5000 successive changes satisfying conditions (i)-(iii) of Definition 1.2 Hence, a peal is nothing more than a partial  $n$ -bell extent. cf. [1], Definition 1.1.6

### 3.2 Excursion into Group Theory

The next step is to describe the extents themselves. We will use group theory to do so.

**Definition 3.2.1.** A **group** is an algebraic structure  $(G, *)$ , in which  $G$  is a set and  $*$  a binary operation  $G \times G \rightarrow G$ ,  $(a, b) \mapsto a * b$  satisfying the following properties:

- Associativity:  $(a * b) * c = a * (b * c)$  for all  $a, b, c \in G$ .
- Identity element: There exists an element  $e \in G$  such that for all  $a \in M$ :  $a * e = e * a = a$   
(The identity for groups under multiplication is 1, under addition it is 0 (cf. [39], Definition 1.1.)).
- Inverse element: For every element  $a \in G$ , there exists an inverse element  $a' \in G$   $a * a' = a' * a = e$ .

Special cases

- Commutative group (ABELian group): For all  $a, b \in G$ :  $a * b = b * a$
- Cyclic group: There exists an element  $a \in G$  such that every element of  $G$  can be expressed as a power of  $a$ .
- Finite group:  $G$  is a finite set. The number of elements is called the order of the group.

cf. [40], 2.4 Algebraic Structures, Groups

**Definition 3.2.2.** A **subgroup** is a subset  $H$  of a group  $G$  that is closed under the operation of  $G$  ( $*$ ), inverses, and contains the identity. It then becomes a group in its own right. Note that associativity is inherited from the parent group and the other two axioms are verified by definition. cf. [39], Definition 2.1

**Definition 3.2.3.** Let  $f : A \rightarrow B$  be a function from the domain  $A$  to the range  $B$ .

- **Injection:**  $\forall x_1, x_2 \in A : x_1 \neq x_2 \Rightarrow f(x_1) \neq f(x_2)$ .
- **Surjection:**  $\forall y \in B : \exists x \in A$  such that  $y = f(x)$ .
- **Bijection:**  $f$  is injective and surjective, so  $\forall y \in B : \exists! x \in A$  such that  $y = f(x)$ .

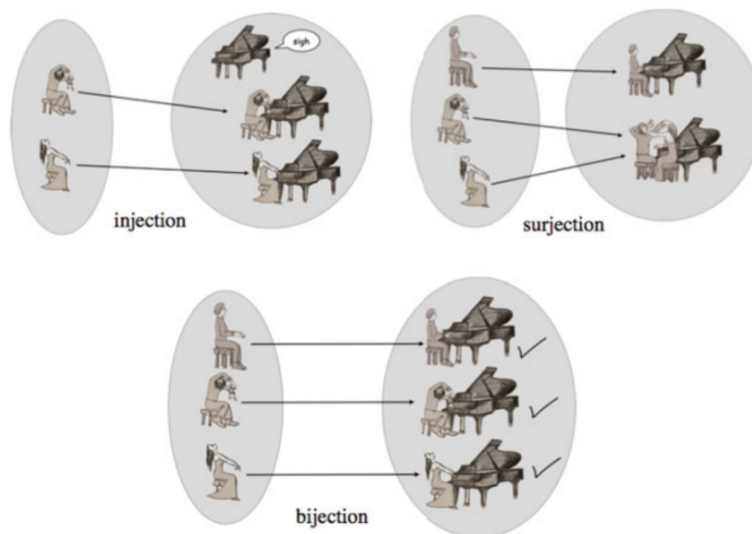


Figure 11: Illustration of three function types: injective, surjective and bijective cf. [41], Figure 4.6

**Definition 3.2.4.** Let  $\Omega$  be an arbitrary non-empty set; we shall often refer to its elements as **points**. A bijection of  $\Omega$  onto itself is called a **permutation** of  $\Omega$ . The set of all permutations of  $\Omega$  forms a group, under the composition of mappings, called the **symmetric group** on  $\Omega$ . We shall denote this group by  $Sym(\Omega)$ , and write  $S_n$  to denote the special group  $Sym(\Omega)$  when  $n$  is a positive integer and  $\Omega = 1, 2, \dots, n$ . A **permutation group** is just a subgroup of a symmetric group. If  $\Omega$  and  $\Omega'$  are two non-empty sets of the same cardinality (that is there is a bijection  $\alpha \mapsto \alpha'$  from  $\Omega$  onto  $\Omega'$ ) then the group  $Sym(\Omega)$  is isomorphic to the group  $Sym(\Omega')$  via the mapping  $x \mapsto x'$  defined by:

$$x'(\alpha') \mapsto \beta' \text{ when } x(\alpha) \mapsto \beta.$$

In particular,  $Sym(\Omega) \cong S_n$  whenever  $|\Omega| = n$ . cf. [42], section 1.2 Symmetric Groups

In change ringing, all permutations we examine are finite, even if the definition above theoretically includes both finite and infinite permutations.

**Definition 3.2.5.** There are two common ways in which permutations are written. First of all, the mapping  $x : \Omega \rightarrow \Omega$  may be written explicitly in the form

$$x = \begin{pmatrix} \alpha_1 & \alpha_2 & \dots & \alpha_n \\ \beta_1 & \beta_2 & \dots & \beta_n \end{pmatrix}$$

where the top row is some enumeration of the points of  $\Omega$  and  $\beta_i$  is the image of  $\alpha_i$  under  $x$  for each  $i$ . The other notation is to write  $x$  as a product of disjoint cycles. A permutation  $c \in Sym(\Omega)$  is called an **r-cycle** ( $r = 1, 2, \dots$ ) if for  $r$  distinct points  $\gamma_1, \gamma_2, \dots, \gamma_r$  of  $\Omega$ ,  $c$  maps  $\gamma_i$  onto  $\gamma_{i+1}$  ( $i = 1, \dots, r-1$ ), maps  $\gamma_r$  onto  $\gamma_1$ , and leaves all other points fixed. cf. [42], 1.2 Symmetric Groups

**Example 3.2.6.** The best way to illustrate these definitions is by looking at an example. In this example we will look at a permutation in the  $S_6$  group.  $S_6$  is the group of all permutations on the set  $\Omega = 1, 2, 3, 4, 5, 6$  and has the cardinality  $|S_6| = 6! = 720$ . We will be looking at a permutation  $\sigma$  that maps 1 to 2, 2 to 3, 3 to 1, 4 to 6 and 6 to 4, whereas 5 stays in its original position. This mapping can be written as

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 2 & 3 & 1 & 6 & 5 & 4 \end{pmatrix}$$

alternately it can be written in cycle notation as  $\sigma = (123)(46)$ . (Side note: this example is not a valid transition in change ringing as 3, 4 and 6 “jump” more than one spot.)

Cycle notation can be used to denote valid transitions in change ringing. For example a change on 4 bells, where the first switches with the second, will be denoted as below  $A = (12)$ .

To connect these concepts to change ringing we need to look at the definition of a change. As shown in definition 3.1.1 a changes correspond to the ringing of permutations on  $n$  bells, so an extent is just going through all the permutations in the group  $S_n$ , where only some swaps are allowed to be made. As shown in the definition 1.2 i. ii. no change can be repeated, with the exception of the rounds at the very beginning and end, so the members of an extent correspond to the members of  $S_n$ .

Another restriction put on the ringing, is the fact, that one bell is not able to move more than one position in one turn, as shown in def 1.2 iii. So for an example of  $n = 4$  bells, there are  $F_4 - 1 = 4$  possible transitions:  $A = (12)(34)$ ,  $B = (23)$ ,  $C = (34)$ ,  $D = (12)$ . With only these four different transitions one is able to make quite a large number of different extents.











**Definition 3.2.7.** Let  $n \geq 2$ . The number of possible transitions for  $n$  bells  $t(n)$  is given by





































































































$$t(n) = F_n - 1.$$

where  $F_n$  is the  $n$ th Fibonacci number with the initial condition  $F_0 = F_1 = 1$ . cf. [1], theorem 3.2.2

Table 2: Some extents on  $n = 4$  bells (Minimus) *cf.* [1], Table 2.4

name of the method	transition sequence $t$
Plain Bob	$(AB)^3AC$
Reverse Bob	$ABAD(AB)^2$
Double Bob	$ABADABAC$
Canterbury	$ABCD CBAB$
Reverse Canterbury	$DB(AB)^2DC$
Double Canterbury	$DBCDCBDC$
Single Court	$DB(AB)^2DB$
Reverse Court	$AB(CB)^2AB$
Double Court	$DB(CB)^2DB$
St. Nicolas	$DBADABDC$
Reverse St. Nicolas	$ABCD CBAC$

**Example 3.2.8.** Let there be 4 bells , ,  & , where  is the treble, the highest bell,  the second highest,  the second lowest and  the tenor or the lowest bell. The bells are also sized accordingly, with  being the smallest and  the largest. If we ring these bells according to a Reverse Canterbury Minimus method ( $DB(AB)^2DC$ ), the  $n! + 1 = 25$  changes will be:

				Rounds
				D = (12)
				B = (23)
				A = (12)(34)
				B = (23)
				A = (12)(34)
				B = (23)
				D = (12)
				C = (34)
				D = (12)
				B = (23)
				A = (12)(34)
				B = (23)
				A = (12)(34)
				B = (23)
				D = (12)
				C = (34)
				D = (12)
				B = (23)
				A = (12)(34)
				B = (23)
				A = (12)(34)
				B = (23)
				D = (12)
				C = (34)Rounds

It is important to note that the Reverse Canterbury Minimus method is not technically an extent according to definition 3.1.2. Condition (iv) of definition 3.1.2 is not fulfilled here, because the bell in the last position stays there for 3 changes and not the maximum of 2 from the definition. This is the case for many traditional change ringing methods.

### 3.3 Graphical Illustration

**Definition 3.3.1.** Let  $S \subseteq G$  be a subset of a group  $G$ . We say  $S$  is a **generating set** of this group if there does not exist a proper subgroup of  $G$  containing  $S$ . So every element of  $G$  can be expressed as a product of elements of  $S$  and their inverses. *cf. [1], Definition 3.2.3, and [43], Definition 3.0.0*

**Definition 3.3.2.** A **directed graph**  $\Gamma$  (sometimes also called **digraph**) consists of a vertex set  $V(\Gamma)$ , an edge set  $E(\Gamma) = V \times V$  and a function, which assigns an ordered pair of vertices  $(v_1, v_2)$  to each edge such that  $v_1$  is the **tail** and  $v_2$  is the **head** of this edge. *cf. [1], Definition 3.2.4*

**Definition 3.3.3.** Let  $G$  be a group with generating set  $S$ . The Cayley color graph of  $G$  with respect to  $S$ , denoted by  $C_S(G)$ , is a colored directed graph such that the following three conditions hold:

- (i) Every vertex of  $C_S(G)$  corresponds to an element of the group  $G$ .
- (ii) Every element  $s \in S$  is assigned to a color  $c_s$ .
- (iii)  $\forall g_1 \in G, \forall s \in S$  the vertex corresponding to  $g_1$  is connected with the vertex corresponding to  $g_2 = g_1 s$  by a directed edge of color  $c_s$ .

*cf. [1], Definition 3.2.5*

**Definition 3.3.4.** A path in a graph  $\Gamma$  that visits each vertex of  $V(\Gamma)$  exactly once is called a **Hamiltonian path**. If, in addition, the Hamiltonian path is a cycle, then we call it a **Hamiltonian cycle**. A graph that contains a Hamiltonian cycle is called a **Hamiltonian graph**. *cf. [1], Definition 3.2.11*

**Example 3.3.5.** This is the Cayley graph for  $n = 3$  bells. For three bells the only two possible changes are  $(12)$  and  $(23)$ , which makes a generating set  $S = \{A = (12); B = (23)\}$ . From any of the given vertices, there is exactly one Hamiltonian cycle in the graph starting out from that vertex, so one could theoretically start ringing from any of them and return to the original one. Even if there is only one Hamiltonian cycle, there are two ways to ring the extent, as you could ring it forwards or backwards, or in this case clockwise or anticlockwise. Although it does not matter where you start in this graph, for demonstration purposes, the vertex corresponding to rounds is marked in figure 12 in blue.

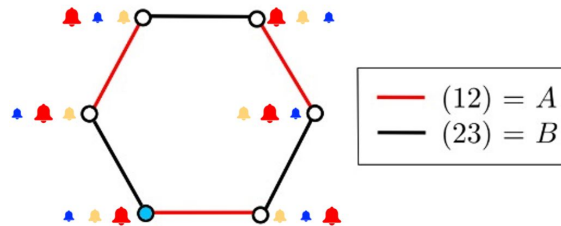


Figure 12: The 3-bell Cayley graph  $C_S(S_3)$ , where rounds are at the bottom left vertex and marked in blue. *cf. [1], Figure 3.1*

**Theorem 3.3.6.** *The number of transitions  $t$  needed to complete a Hamiltonian cycle in the digraph of a symmetric group  $S_n$  is equal to the number of vertices  $|S_n| = n!$  that need to be visited in the cycle.*

*Proof.* For every vertex that is visited in a given Hamiltonian cycle, there has to be a transition to said vertex. As no vertex is left out, the number of transitions  $t$  needed to complete the cycle corresponds to the number of vertices there are, which is the order of the Symmetric group  $|S_n| = n!$ .  $\square$

### 3.3.1 Examples for 4 Bells

For 4 bells the Cayley graph with the order of the bells on each vertex is shown in figure 13.

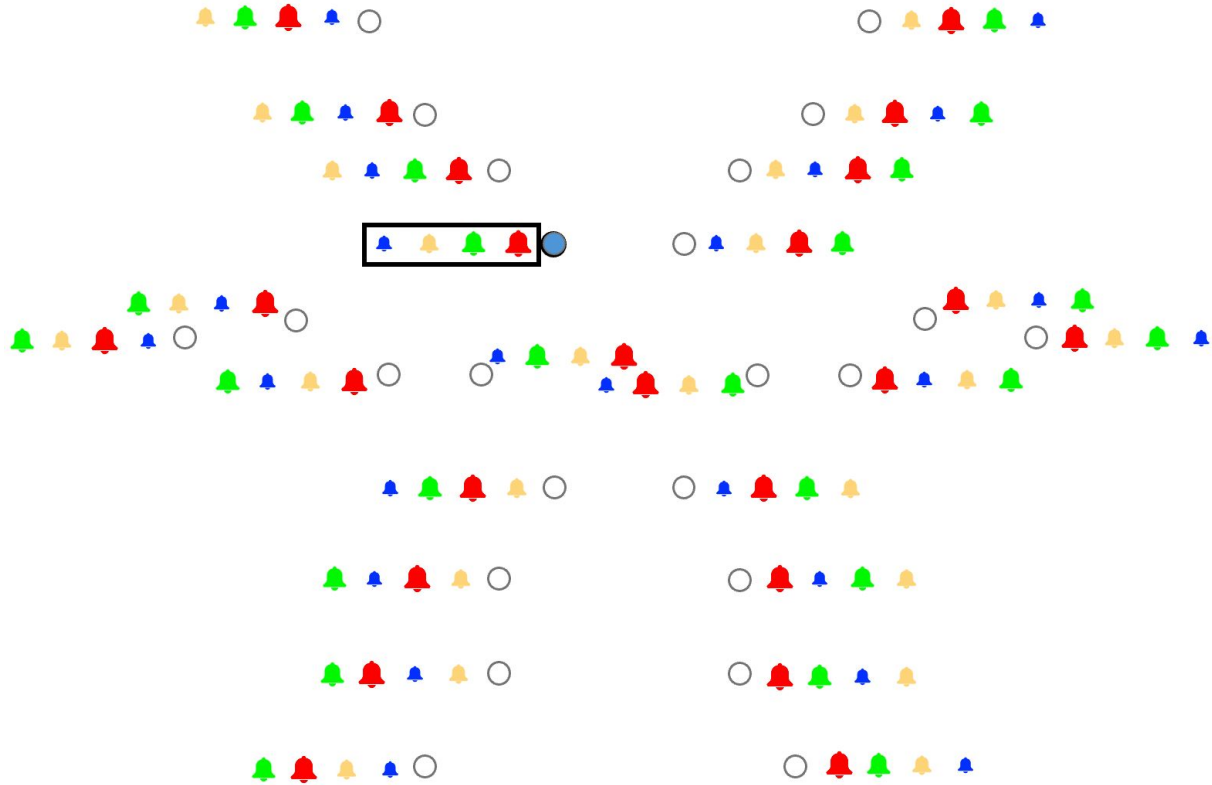


Figure 13: Vertices of the Cayley graph of  $C_S(S_4)$ , showing the change corresponding to each vertex. Rounds are marked in blue and framed with a box.

Figure 14 shows the Cayley graph for  $S_4$  with the generating set  $S = \{A = (12)(34), B = (23), C = (34), D = (12)\}$ . This graph shows all the possible transitions from one change to the next on four bells.

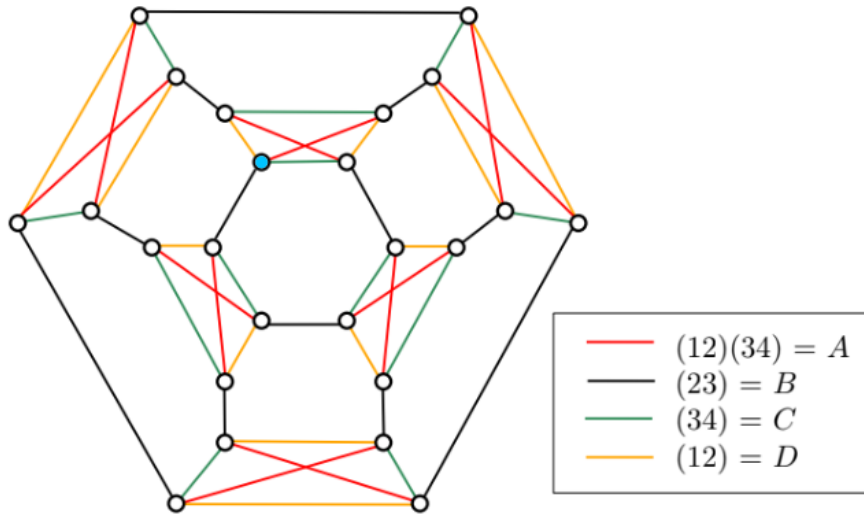
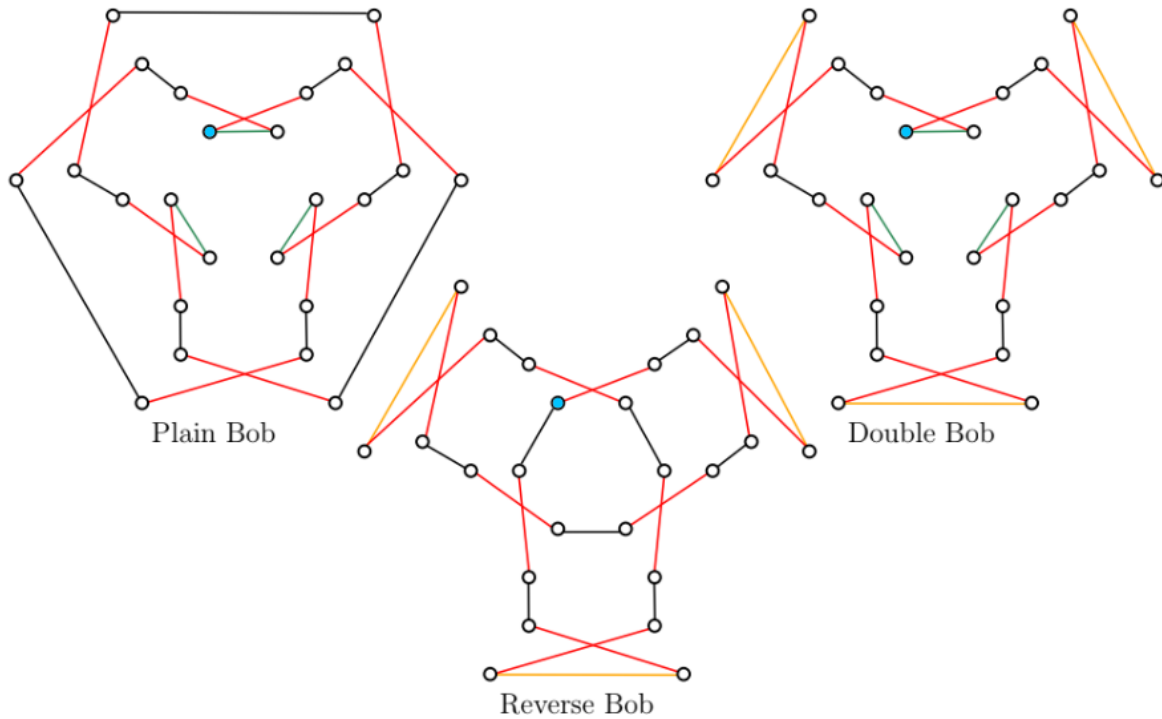


Figure 14: The 4-bell Cayley graph  $C_S(S_4)$  cf. [1], Figure 3.2

Within this graph one can now find Hamiltonian cycles as in  $C_S(S_3)$ , only that in this case, not all the possible transitions are always used in one given cycle. For example we can look at the extents shown in Table 2.



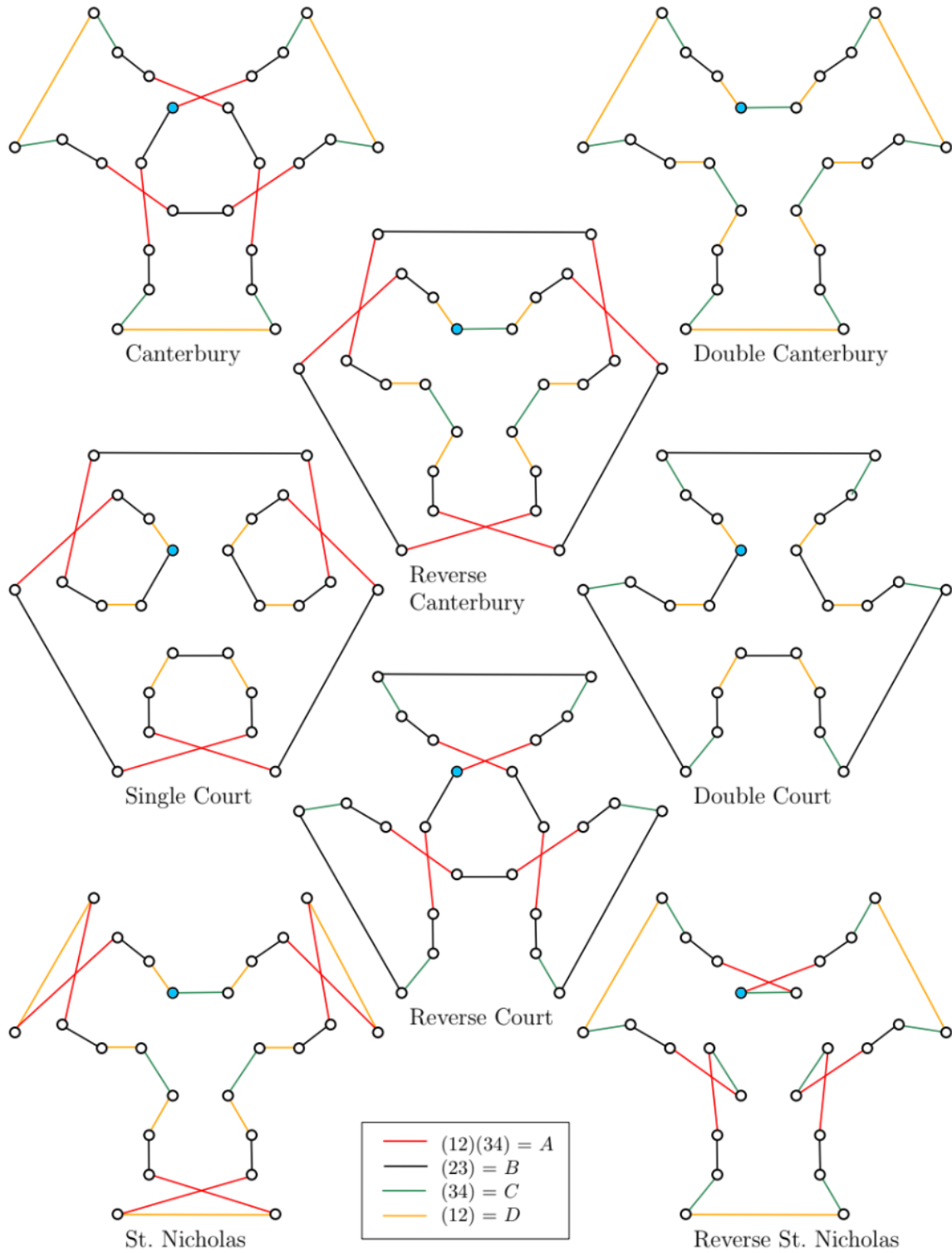


Figure 15: Here, the upper-left vertex of the inner hexagon of  $C_5(S_4)$  is labeled by rounds in dark sky blue, so it is the start for all Hamiltonian cycles. The extents shown here are: Plain Bob Minimus  $((AB)^3AC)^3$ , Reverse Bob Minimus  $((ABAD(AB)^2)^3)$ , Double Bob Minimus  $((ABADABAC)^3)$ , Canterbury Minimus  $((ABCD CBAB)^3)$ , Reverse Canterbury Minimus  $((DB(AB)^2DC)^3)$  (as seen in example 3.2.8)6, Double Canterbury Minimus  $((DBCDCBDC)^3)$ , Single Court Minimus  $((DB(AB)^2DB)^3)$ , Reverse Court Minimus  $((AB(CB)^2AB)^3)$ , Double Court Minimus  $((DB(CB)^2DB)^3)$ , St. Nicholas Minimus  $((DBADABDC)^3)$  and to Reverse St. Nicholas Minimus  $((ABCD CBAC)^3)$ . cf. [1], Figure 3.4

### 3.4 Finding Hamiltonian Cycles

The Hamiltonian cycles problem is well known to graph theorists. According to *Finding Hamiltonian Cycles* by Sridher Kaminani “To find a Hamiltonian cycle in a general graph is an NP-Complete problem and no deterministic polynomial-time algorithm has been discovered to find a Hamiltonian cycle in a general graph” [44]. In short, there is no efficient general algorithm to find a Hamiltonian cycle in a general graph. But there are some approaches to find Hamiltonian cycles. One of these is the *Steinhaus-Johnson-Trotter* algorithm. It generates permutations by using *adjacent transpositions*, so basically swapping two places in the permutation. [44][45]

**Example 3.4.1.** To illustrate and explain the Steinhaus-Johnson-Trotter algorithm this example will find all the permutations of  $S_3$  starting with 1, 2, 3. For this we will be using the concept of mobile numbers. A number  $n$  in the permutation is mobile, iff  $n$  is greater than the number it is pointed to. For the permutations, the largest mobile number  $k$  will move in the direction of its arrow and any arrows above numbers  $> k$  will be flipped to point the other direction. This swap of the arrow will be denoted as:  $\overleftarrow{n}$ . For example, the in the first row of the example below both 2 and 3 are both mobile numbers, because 2 is pointed at 1 and  $2 > 1$  and 3 is pointed at 2 and  $3 > 2$ . In this case the number that moves  $k = \overleftarrow{3}$ , because it is the larger of the two numbers. After the third line, where  $k = \overleftarrow{2}$ , the arrow over 3 switches directions, because all the arrows over the numbers  $n > k$  switch directions. [46]

$$\begin{array}{ll}
 \overleftarrow{1} \overleftarrow{2} \overleftarrow{3} & \text{mobile numbers: } 2 < 3; k = 3 \quad (1) \\
 \overleftarrow{1} \overleftarrow{3} \overleftarrow{2} & \leftarrow 3: \text{mobile number: } 3; k = 3 \quad (2) \\
 \overleftarrow{3} \overleftarrow{1} \overleftarrow{2} & \leftarrow 3: \text{mobile number: } 2; k = 2 \quad (3) \\
 \overrightarrow{3} \overleftarrow{2} \overleftarrow{1} & \leftarrow 2; \overleftarrow{3}: \text{mobile number: } 3; k = 3 \quad (4) \\
 \overleftarrow{2} \overrightarrow{3} \overleftarrow{1} & 3 \rightarrow: \text{mobile number: } 3; k = 3 \quad (5) \\
 \overleftarrow{2} \overleftarrow{1} \overrightarrow{3} & 3 \rightarrow: \text{no mobile numbers} \rightarrow \text{all permutations have been found} \quad (6)
 \end{array}$$

In this way, the Steinhaus-Johnson-Trotter algorithm can be used to find all permutations of any set. For the full descriptions of the algorithm by S. M. Johnson and H. F. Trotter see references [47] and [48] respectively. To come back to change ringing, this algorithm is also called the method of *plain changes*. Note that steps (1) through (6) result in the sequence shown in figure 12. The method of plain changes for  $n$  elements is the same as the Plain Bob method on  $n$  bells. The Steinhaus-Johnson-Trotter algorithm, first published in the 20<sup>th</sup> century, was originally based on the change ringing method of plain changes, which can be traced back to a transcription of a manuscript by Peter Mundy from 1653 in Morris’ *The History and Art of Change Ringing* [3]. Shortly after this text had been published, *Tintinnalogia* [32] was also published containing a long description of plain changes. [45] [49]



## 4 Programming

The goal of this part of the project was to computationally generate valid change ringing sequences.

### 4.1 Beginning to Program

I chose to use Python for this project because it is one of the easier programming languages to learn and use. I used the integrated development environment Thonny [50].

To start I wanted to simply be able to print a list of items.

```
1 z = [1, 2, 3] #define a list
2 print(z)
```

Output:

```
[1, 2, 3]
```

This is done by first defining a list (line 1) and printing it with the `print()` function on line 2.

After this the next important thing was to be able to swap two items in a list.

```
1     def swapPositions(z, pos1, pos2):
2
3     # Storing the two elements as a pair in a tuple variable as get
4     get = z[pos1], z[pos2]
5
6     # unpacking those elements
7     z[pos2], z[pos1] = get
8
9     return z
10
11 z = [1, 2, 3] #define a list
12 pos1, pos2 = 0, 1
13
14 print(z)
15 print(swapPositions(z, pos1, pos2))
```

This is done by defining a new function that swaps two elements. The function `swapPositions(z, pos1, pos2)` takes inputs `z`, the list, and `pos1` and `pos2`, the positions of the two items being swapped. For the positions the numbering starts on the left with 0 and goes up to  $n - 1$ , where  $n$  is the number of elements in the list. It is also possible to use negative positions, where the position -1 is equal to position  $n - 1$  and position 0 is position  $-n$ . The function above swaps the elements in the two given positions and returns the updated list `z`. [51][52]

In the end this code prints:

```
[1, 2, 3]
[2, 1, 3]
```

## 4.2 Programming Changes “by Hand”

After defining a function to swap the positions of two bells, I could program full extents by going through and programming each transition individually.

### 4.2.1 Change on 3 Bells

For example, here is a program that prints a full extent for  $n = 3$  bells.

```

1 def swapPositions(z, pos1, pos2):
2
3     get = z[pos1], z[pos2]
4
5     z[pos2], z[pos1] = get
6
7     return z
8
9 z = [1, 2, 3] #define a list
10 pos1, pos2 = 0, 1
11
12 print(z)
13 print(swapPositions(z, pos1, pos2))
14 print(swapPositions(z, pos1+1, pos2+1))
15 print(swapPositions(z, pos1, pos2))
16 print(swapPositions(z, pos1+1, pos2+1))
17 print(swapPositions(z, pos1, pos2))
18 print(swapPositions(z, pos1+1, pos2+1))
19 #Full extent on 3 bells

```

Output:

```

[1, 2, 3]
[2, 1, 3]
[2, 3, 1]
[3, 2, 1]
[3, 1, 2]
[1, 3, 2]
[1, 2, 3]

```

This output corresponds to a counterclockwise extent, as illustrated in figure 12, or the reverse of example 3.4.1.

### 4.2.2 Change on 4 Bells (Reverse Canterbury)

The following code implements the Reverse Canterbury Minimus method (see table 2).

```

1 def swapPositions(z, pos1, pos2):
2
3     # Storing the two elements
4     # as a pair in a tuple variable get
5     get = z[pos1], z[pos2]
6
7     # unpacking those elements
8     z[pos2], z[pos1] = get
9
10    return z
11
12 def swap0123():
13     swapPositions(z, pos1, pos2-1)
14     print(swapPositions(z, pos2, pos2+1))
15
16 def swap12():

```

```

17     print(swapPositions(z, pos1+1, pos2))
18
19 def swap23():
20     print(swapPositions(z, pos2, pos2+1))
21
22 def swap01():
23     print(swapPositions(z, pos1, pos2-1))
24
25 z1 = [1, 2, 3, 4]
26 z = [1, 2, 3, 4] #define a list
27 pos1, pos2 = 0, 2
28
29 print(z)
30 for i in range(10):
31     swap01()
32     swap12()
33     swap0123()
34     swap12()
35     swap0123()
36     swap12()
37     swap01()
38     swap23()
39     if z != z1:
40         continue
41     else:
42         break
43 print('done.')
```

Shell:

```

[1, 2, 3, 4]
[2, 1, 3, 4]
[2, 3, 1, 4]
[3, 2, 4, 1]
[3, 4, 2, 1]
[4, 3, 1, 2]
[4, 1, 3, 2]
[1, 4, 3, 2]
[1, 4, 2, 3]
[4, 1, 2, 3]
[4, 2, 1, 3]
[2, 4, 3, 1]
[2, 3, 4, 1]
[3, 2, 1, 4]
[3, 1, 2, 4]
[1, 3, 2, 4]
[1, 3, 4, 2]
[3, 1, 4, 2]
[3, 4, 1, 2]
[4, 3, 2, 1]
[4, 2, 3, 1]
[2, 4, 1, 3]
[2, 1, 4, 3]
[1, 2, 4, 3]
[1, 2, 3, 4]
done.
```

This program is different from the previous program in two important ways. First, for an extent on 4 bells, instead of using `print(swapPositions(z, pos1, pos2))` for the main code, I defined new function for each possible transition. Second, I used a `for` loop to create the full extent [53]. The `for` loop checks

whether  $z$ , the current change, is the same as  $z1$ , which is defined to be rounds and is not changed by the swap function. If  $z$  is not equal ( $\neq$ ) to  $z1$ , the loop will repeat. When it is equal to  $z1$ , the loop will break and print `done` to show that the program is finished.

### 4.3 Random sequences

Next, I wanted to create a sequence of an arbitrary length, which starting with rounds and given the ringable transitions (swaps) for 4 bells, could create a possible sequence. Here, the transitions are chosen randomly and therefore the goal is not to create an extent, but simply to ring changes until rounds are reached. The program does ensure that the same transitions do not happen twice in a row. The resulting sequence will be of arbitrary length  $\geq 4$ : The shortest possible sequence would be one of 4 changes with the transition sequence being *ACD* or anagrams of it.

At this point the full code is getting long, so I will review only the relevant new functionalities. The full code can be found in appendix A.3.3.

#### 4.3.1 Function to Swap Items: `swap(pos)`

```

10 def swap(pos):
11     if pos == 3:
12         swapPositions(z, pos-3, pos-2)
13         print(swapPositions(z, pos-1, pos))
14         already_done.append(z.copy())
15     else:
16         print(swapPositions(z, pos+1, pos))
17         already_done.append(z.copy())

```

The definitions for the function `swapPositions` is taken from the previous code. Here, however, rather than creating 4 different functions for the 4 transitions, this swap function takes a single input between 0 and 3 and implements the transition accordingly. The inputs 0, 1 and 2 swap the bells in the corresponding position of the list with the one to its right. The first item in a list in python is 0, so `swap(0)` would swap the first item in the list with the second one. The special case of `swap(3)` swaps both the first two and the last two bells, because that is the fourth possible transition with 4 bells. The `swap` functions also adds the new change to the list of permutations that have already been “rung”.

#### 4.3.2 Choose Random Swap: `randomswap(possible_swaps)`

```

19 all_possible_swaps = [0, 1, 2, 3]
20
21 def randomswap(possible_swaps):
22     selected_swap1 = random.choice(possible_swaps)
23     #print("selected swap = ",selected_swap1)
24     swap(selected_swap1)
25     Transition.append(selected_swap1)
26     next_swaps = all_possible_swaps.copy()
27     next_swaps.remove(selected_swap1)
28     return next_swaps

```

This is the function, which will choose at random what transition to make given the possible transitions defined in the previous section. The function takes as an input the possible swaps that could be made at this point in the sequence (`possible_swaps`), so that swaps do not repeat. The print function on line 23 was used to see whether the correct bells were being swapped; The `#` symbol is used to indicate a comment, so this code is not executed.

#### 4.3.3 List to Word: convert(s)

```

30 def convert(s):
31     str1 = ""
32     return(str1.join(s))

```

The function `convert(s)`, takes a list `s` of letters and returns a “word”. For example for `s = ['h', 'e', 'l', 'l', 'o']`, if it is put into `print(convert(s))` the output will be `hello`. This function is used to make the output more compact. [54]

#### 4.3.4 Main Loop

```

34 z = [1, 2, 3, 4]
35 z1 = [1, 2, 3, 4]
36 already_done = []
37 Transition = []
38
39 print(z)
40 next = randomswap(all_possible_swaps)
41 for i in range(100):
42     next = randomswap(next)
43     if z != z1:
44         continue
45     else:
46         print("we're back at the start")
47         break

```

The main loop starts by initializing `z`, the current change, which starts at rounds; `z1`, rounds, which is used as above to check whether the sequence is finished; and `already_done` and `Transition`, which are the lists which will collect the changes and the transitions, respectively, in one run of the program.

To start the process of changing, `z` is first printed, to signify the start of each extent on 4 bells with rounds. For the first change, the program chooses the transition from `all_possible_swaps`. The value it returns is stored as `next`, which is the list of the possible transitions to the next change. After this initial swap comes the main loop to create the changes. It is a `for` loop that will repeat until the sequence comes back to rounds, or reaches a length of 100 (an arbitrary value much longer than the length of a valid extent).

#### 4.3.5 Checking Whether a Sequence is an Extent

```

49 print(already_done)
50 already_done.sort()
51 print("sorted:", already_done)
52 print("length:", len(already_done))
53
54 for i in range(len(Transition)): # for same notation as campanology
55     if Transition[i] == 0:
56         Transition[i] = 'D'
57     if Transition[i] == 2:
58         Transition[i] = 'C'
59     if Transition[i] == 1:
60         Transition[i] = 'B'
61     if Transition[i] == 3:
62         Transition[i] = 'A'
63 print("A = (12)(34), B = (23), C = (34), D = (12)")
64 print("change:", convert(Transition))
65
66
67 perm = permutations([1, 2, 3, 4])
68 perm_list_tuple = list(perm)
69 perm_list_list = []

```

```

70 for tup in perm_list_tuple:
71     perm_list_list.append(list(tup))
72 print("all permutations:", perm_list_list)
73 print("how many permutations?", len(perm_list_list))
74
75 if perm_list_list == already_done:
76     print('The lists are the same')
77 else:
78     print('The lists are not quite the same')

```

It is extremely unlikely in this case that the main loop will generate a valid extent, but since this is relevant to my project I wrote some code to do this check in various ways:

- A simple check is the length of the list. A valid extent would have a length of 24, though not all lists of length 24 are extents. (line 52)
- The list of transitions is also converted to letters according to the notation used in *Campanology - Ringing the changes* [1], in order to facilitate the comparison of sequences. (lines 54-64)
- I also generate a list of all permutations using the `permutations` function of the `itertools` library and compare it to the sorted list of permutations generated by the main loop. For a valid extent, these two lists would be the same. (lines 67-78)

#### 4.3.6 Output

Here is an example output of one run of this program. Note that because the transitions are randomly selected, every run of this program will be different. While each of the transitions is valid, this result is not a valid extent, because some of the changes are repeated.

```

[1, 2, 3, 4]
we're back at the start
[[1, 3, 2, 4], [3, 1, 4, 2], [1, 3, 4, 2], [1, 4, 3, 2], [4, 1, 3, 2], [1, 4, 2, 3],
[1, 2, 4, 3], [2, 1, 3, 4], [2, 3, 1, 4], [3, 2, 1, 4], [3, 1, 2, 4], [3, 1, 4, 2],
[3, 4, 1, 2], [3, 4, 2, 1], [3, 2, 4, 1], [3, 2, 1, 4], [2, 3, 4, 1], [2, 4, 3, 1],
[2, 4, 1, 3], [4, 2, 3, 1], [4, 3, 2, 1], [3, 4, 1, 2], [4, 3, 1, 2], [4, 3, 2, 1],
[3, 4, 1, 2], [3, 1, 4, 2], [1, 3, 4, 2], [1, 3, 2, 4], [1, 2, 3, 4]]
sorted: [[1, 2, 3, 4], [1, 2, 4, 3], [1, 3, 2, 4], [1, 3, 2, 4], [1, 3, 4, 2],
[1, 3, 4, 2], [1, 4, 2, 3], [1, 4, 3, 2], [2, 1, 3, 4], [2, 3, 1, 4], [2, 3, 4, 1],
[2, 4, 1, 3], [2, 4, 3, 1], [3, 1, 2, 4], [3, 1, 4, 2], [3, 1, 4, 2], [3, 1, 4, 2],
[3, 2, 1, 4], [3, 2, 1, 4], [3, 2, 4, 1], [3, 4, 1, 2], [3, 4, 1, 2], [3, 4, 1, 2],
[3, 4, 2, 1], [4, 1, 3, 2], [4, 2, 3, 1], [4, 3, 1, 2], [4, 3, 2, 1], [4, 3, 2, 1]]
length: 29
A = (12)(34), B = (23), C = (34), D = (12)
change: BADBDABABDBCBCBCABCABADCBDCB
all permutations: [[1, 2, 3, 4], [1, 2, 4, 3], [1, 3, 2, 4], [1, 3, 4, 2],
[1, 4, 2, 3], [1, 4, 3, 2], [2, 1, 3, 4], [2, 1, 4, 3], [2, 3, 1, 4], [2, 3, 4, 1],
[2, 4, 1, 3], [2, 4, 3, 1], [3, 1, 2, 4], [3, 1, 4, 2], [3, 2, 1, 4], [3, 2, 4, 1],
[3, 4, 1, 2], [3, 4, 2, 1], [4, 1, 2, 3], [4, 1, 3, 2], [4, 2, 1, 3], [4, 2, 3, 1],
[4, 3, 1, 2], [4, 3, 2, 1]]
how many permutations? 24
The lists are not quite the same

```

## 4.4 Program to Find Valid Extents on 4 Bells

This last section of the programming part describes my final program, which includes an audible representation of the changes. There are many steps between this program and the one before, but I will not discuss them here. Some of them are in the appendix, where you can find several intermediate programs.

The primary goal of this program is to find valid extents on 4 bells. It does this by making a transition and checking it against the criteria from definition 3.1.1. After finding all valid extents, it chooses one at random and uses recordings of the bells of Maschwanden to play the extent. The program was ultimately run from the penguin terminal on my device, because I could not find out how to get Thonny to support the audio.

For this program I will also not describe the full program, but rather the new functionalities not included in the previous descriptions.

In this part of the program, I reached the limits of my programming capabilities and my father helped me write a program that uses recursion to find the valid extents.

### 4.4.1 Function to Test Rows: `test_rows`

The function `test_rows` is designed to test whether a new change can be added to the previous changes so that the sequence still fulfills conditions (ii) and (iv) of definition 3.1.2.

```

23 def test_rows(previous_rows: list[list[int]], next_row: list[int]) -> bool:
24     if len(previous_rows) != 24 and next_row in previous_rows:
25         return False
26
27     for i in range(len(next_row)):
28         if (previous_rows[-2][i] == previous_rows[-1][i] and
29             previous_rows[-2][i] == next_row[i]):
30             return False
31
32     else: return True

```

The first thing it checks is if the row it is trying to add (`next_row`) is already in the list of previous rows, so (ii) of definition 3.1.2 is fulfilled. Also the length of `previous_rows` has to be unequal to 24, because rounds will be repeated in this special case at the end of an extent. (Line 40 in the definition of `extend_rows` will check if the last change is rounds, which will filter out any false endings.) Secondly, this code tests, for each of the positions of the next row, whether the bell rung in that position has changed since the last two rows to fulfill (iv) of definition 3.1.2. If these two conditions from definition 3.1.2 are fulfilled, the function returns true.

### 4.4.2 Function to Build Extents: `extend_rows`

The recursive function `extend_rows` is the key to the search for valid extents among the  $4 \cdot 3^{23} = 376,572,715,308$  potentially valid sequences. It takes three arguments: `rows`, the list of “changes” that have been “rung” so far, `positions`, the list of positions (known in previous versions of the program as swaps), that have been completed, and `avail_pos` the potential swaps that could happen next (see for comparison `possible_swaps` in section 4.3.2). It returns a list of the lists of changes and a list of the lists of swaps, as will be discussed further below.

```

36 def extend_rows(rows: list[list[int]], positions: list[int], avail_pos:
37     list[int]) -> tuple[list[list[int]], list[list[int]]]:
38     result: list[list[list[int]]] = []
39     result_pos: list[list[int]] = []
40     if len(rows) == 25 and rows[-1] == [1, 2, 3, 4]:
41         # print(f'found it {rows}')
42         return [rows], [positions]
43     if len(rows) >= 25:
44         print(f'overdone {rows}')
45         return result, result_pos

```

```

46     for pos in avail_pos:
47         # print(f'testing {rows} move {pos}')
48         new_row = swap(rows[-1], pos)
49         if test_rows(rows, new_row):
50             # print(f'{len(rows)} - extending {rows} with {new_row}.')
51             new_rows = rows.copy()
52             new_rows.append(new_row)
53             new_positions = positions.copy()
54             new_positions.append(pos)
55             all_possible_pos.copy()
56             new_avail_pos = all_possible_pos.copy()
57             new_avail_pos.remove(pos)
58             new_result, new_result_pos = extend_rows(new_rows, new_positions, new_avail_pos)
59             if new_result:
60                 result.extend(new_result)
61                 result_pos.extend(new_result_pos)
62
63     return result, result_pos

```

“Recursive” means that a function calls itself. A common example of a recursive function is one that makes a list of all the files in a particular folder on a computer and all its sub-folders: It looks at the items in the folder one at a time. When it encounters files it adds them to a list, and as it encounters sub-folders it calls itself to open the folder and search further. [55]

In the case of `extend_rows`, the function builds potential extents one change at a time, and what the function returns depends on whether a valid extent has been found. It returns different things in different cases:

- If the length of a sequence is the length of an extent, 25, and the last element of the sequence, `rows[-1]` is [1,2,3,4] or rounds, then a valid extent has been found and the inputs `rows` and `positions` are returned (lines 40-42).
- If the sequence has grown longer than a valid extent or if the length of `rows` is 25 but `rows[-1]` is not rounds, empty lists are returned (lines 43-35).
- If neither of the two conditions above are met, the program loops through the next available transitions (line 46). If `test_rows`, described in section 4.4.1 above, returns `True`, indicating that the transition is valid, the function extends the sequence of rows by that one change and calls itself again based on the extended sequences and new list of possible transitions. If the recursive call to `extend_rows` returns a valid result, the result is added to the list of the lists of valid extents.
- If no valid extents are found, line 63 will return an empty list. If valid extents have been found, line 63 will return a list of valid extents.

The final return of this function will contain all of the valid extents that can be built from a single stem of two changes (rounds and the result of one possible transition) as well as the lists of the transitions needed to ring these changes.



#### 4.4.3 Main Loop to Build Extents

```

94 final_result: list[list[list[int]]] = []
95 final_result_pos: list[list[int]] = []
96 for pos in all_possible_pos:
97     #print(f'Starting with move {pos}')
98     new_row = swap(z, pos)
99     new_avail_pos = all_possible_pos.copy()
100     new_avail_pos.remove(pos)
101     new_result, new_result_pos = extend_rows([z, new_row], [pos], new_avail_pos)
102     if new_result:
103         final_result.extend(new_result)
104         final_result_pos.extend(new_result_pos)
105
106 print("A = (12)(34), B = (23), C = (34), D = (12)")
107 for i in range(len(final_result)):
108     print(f'{convert(Translate_pos_alphabet(final_result_pos[i]))}')
109     print(f'{final_result[i]}')
110 print("number of valid extents:", len(final_result))

```

Now that `extend_rows` has been defined, it can be used in the main code that begins the recursion. First, `final_result` and `final_result_pos` are defined as empty lists of lists of extents and transitions. These lists will be filled with the lists of valid extents and transitions returned by `extend_rows`. Next, the program loops through all the possible transitions and calls `extend_rows` for each of the short stems beginning with rounds and continuing with each of the four possible transitions. `extend_rows` will, as described above, return all of the valid extents that can be built from a single stem. These results are added to the lists `final_result` and `final_result_pos`.

The main loop finishes by printing the valid extents as lists of transitions to ring the extent and as lists of changes, and finishes by printing the number of valid extents found.

#### 4.4.4 Output

The main loop finds 24 valid extents for  $n = 4$  bells:

```

1  DABABABADABABABADABABABA
2  DABACABADABACABADABACABA
3  BADABABABADABABABADABABA
4  BADABACABADABACABADABACA
5  BABADABABABADABABABADABA
6  BABABADABABABADABABABADA
7  BABABACABABABACABABABACA
8  BABACABABABACABABABACABA
9  BACABADABACABADABACABADA
10 BACABABABACABABABACABABA
11 CABADABACABADABACABADABA
12 CABABABACABABABACABABABA
13 ADABABABADABABABADABABAB
14 ADABACABADABACABADABACAB
15 ABADABABABADABABABADABAB
16 ABADABACABADABACABADABAC
17 ABABADABABABADABABABADAB
18 ABABABADABABABADABABABAD
19 ABABABACABABABACABABABAC
20 ABABACABABABACABABABACAB
21 ABACABADABACABADABACABAD
22 ABACABABABACABABABACABAB
23 ACABADABACABADABACABADAB
24 ACABABABACABABABACABABAB

```

Interestingly, all 24 valid extents consist of three repeated sequences of transitions. The attentive reader

will recognize three methods from table 2: Plain Bob  $(AB)^3AC$  (line 19), Reverse Bob  $ABAD(AB)^2$  (line 15) and the Double Bob  $ABADABAC$  (line 16). The remaining 21 valid extents turn out to be variations of these three patterns. For example the extent on line 12  $C(AB)^3A$  is a variant of Plain Bob circularly shifted by one position. The Cayley graphs of the eight variants of the Plain Bob Minimus are shown in figure 16. As noted in section 3.2 the remaining methods in table 2 are actually not valid extents according to condition (iv) of definition 3.1.2.

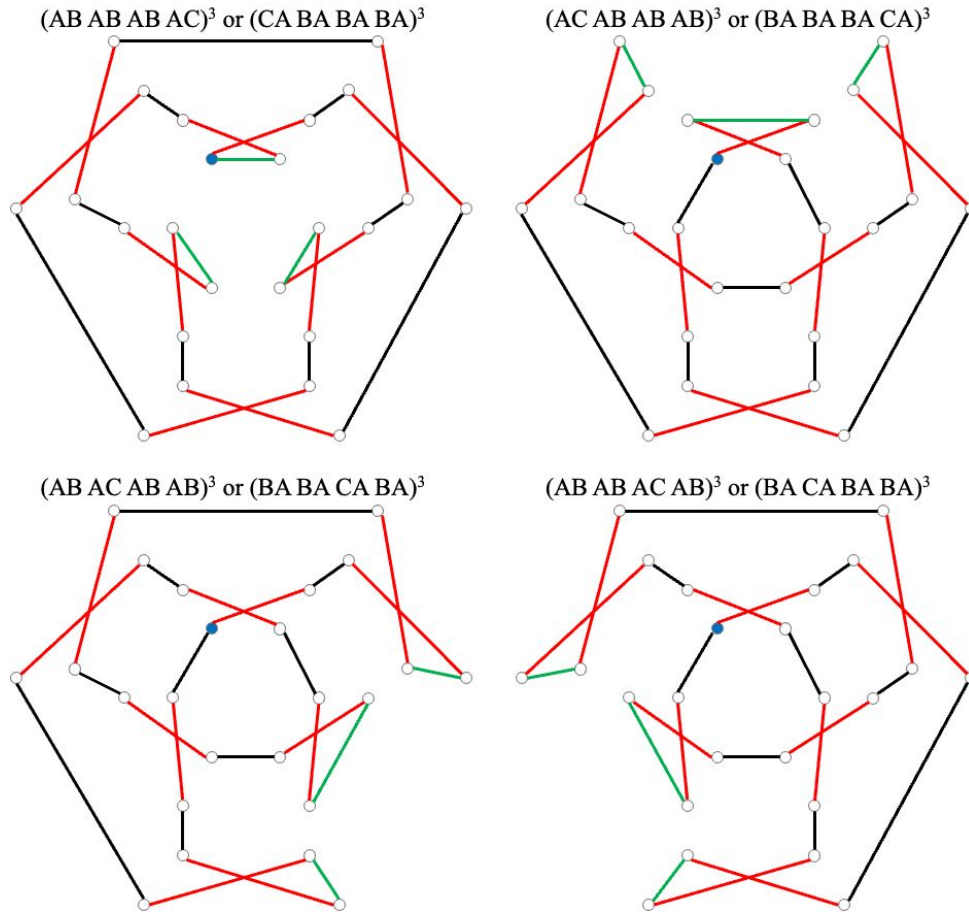


Figure 16: The Cayley graphs of the eight variants of the Plain Bob Minimus

#### 4.4.5 Audio

The final task for the programming part was to create an audio representation of an extent.

```

3  from pygame import mixer
4  from pygame import time

112 choice = random.randrange(len(final_result))
113
114 print("randomly chose:", choice)
115 print(f'{convert(Translate_pos_alphabet(final_result_pos[choice]))}')
116 print(f'{final_result[choice]}')
117 chosen = final_result[choice]
118
119 mixer.init()
120 soundfiles = ["Treble1.mp3", "Bell2.mp3", "Bell3.mp3", "Tenor4.mp3"]
121 sounds = []
122 for s in soundfiles:
123     sounds.append(mixer.Sound(s))
124
125 def output(bell_output):
126     for i, bell in enumerate(bell_output):
127         print(bell, end = " ")
128         mixer.Channel(i).play(sounds[bell - 1])
129         time.wait(1000)
130     print("\n", end = "")
131
132 for i in chosen:
133     output(i)
134 print("we done")
135 time.wait(1000)
136 mixer.quit()

```

Lines 112-117 choose and print a result to play. One of the extents is chosen and later this one will be played with audio recordings of bells. The actual choice it makes is the choice of a position in `final_result`. `choice` (line 112) is a random integer within the length of `final_result` chosen with `random.randrange(len(final_result))`. The function `random.randrange(stop)` chooses an integer in the range from 0 to the number given for `stop` [56]. After choosing the extent to play, the program prints the position in the list of results of this extent along with the extent itself and its transitions.

For the audio output, I used the modules `mixer` and `time` from the Pygame library, which was designed for programming games in Python [57]. The `mixer` module is used to load sound objects and control the playback of the sounds. In the program, it is first initialized on line 119 and in the following line, the names of the sound files are given. Lines 120 and 123 load the sound files for the 4 bells into the `mixer` using the `mixer.Sound(s)` function. The audio files are saved in the same folder as the code and so that they can be easily retrieved by the program.

The function `output(bell_output)` plays and prints a single change from the extent. The sounds are played on different channels, so they can overlap. After a sound is played the program waits 1 second (1000 milliseconds) before playing the next bell.

The main loop of the program simply loops through the changes and calls the `output` function to play each change. At the end it waits so the last bell to sound has time to fade away before the mixer module is closed.



Figure 17: Link to a demonstration of the program in action

## 5 Discussion and Conclusions

For me, the best part of this project was its interdisciplinary nature. In the course of this one project, I read texts from the 17<sup>th</sup> century, learned the basics of group theory and programming in Python, and recorded bells in a Swiss bell tower.

The history of bells and bell ringing is fascinating and I have only been able to give an overview here. In the course of my research, I learned many interesting things which did not make it into this report, for example, how the change ringers have almost their own language for saying where a bell has to go next. This is briefly touched on in section 2.4.3, in which the methods are described, but only a few words could be explained there. I would particularly like to learn more about the methods used in change ringing and how people compose them. My biggest regret in this project is that I was not able to attend a change ringing session in person.

While working on the mathematical part of this project, I found that it was difficult to find definitions that I could understand to use in this paper, because the sources I used either had definitions that were too detailed or they went over the concepts only in passing, never really defining them. I needed to consult a number of sources before I felt comfortable presenting the mathematical background in this paper.

I would also like to learn more about the symmetries of the Cayley graphs in section 3.3 and how to use Cayley graphs to find extents. The outputs of the final program were made into Cayley graphs (see figure 4.4.4) and it is very interesting to see the fact that they are all symmetric in some sort of way. In the future I would like to make Cayley graphs for other methods and larger numbers of bells to see, what their graphs reveal.

My programming goals evolved over the course of the project. Originally, I wanted to write a program which could create an extent or peal for  $n$  bells. This could have been done by implementing the Steinhaus-Johnson-Trotter algorithm described in section 3.4. Instead, I decided that a more intriguing problem was to look at all the ringable sequences for a given number of bells and find which of these were valid extents. Even for four bells, there are three possible transitions from any given change, so the number of potential extents is  $4 \cdot 3^{23} = 376,572,715,308$ . In the end I explored these by using a recursive function that effectively tested whether a given sequence was valid while it was being built, cutting off any sequences that were never going to generate valid extents before going through all 24 transitions. But extending my function even to five bells, which require  $5! = 120$  transitions to form an extent, makes the computation too large to run in a reasonable amount of time. This would be an interesting problem to revisit once I know more about programming. Nevertheless, I am proud of what I was able to accomplish given that I started with very little knowledge of Python. In particular it was satisfying that audio output of my program sounded like real recordings of change ringing.

If I had had more time, I would have edited my program again to make it more legible. At the moment, there are not very many comments on what the program is doing and the variables could have better names. Ideally, the code would be readable on its own, without explanatory text.

The interaction between the historical, mathematical, and computational parts of this project were also very interesting. How did bell ringers in the 17<sup>th</sup> century solve a combinatorics problem practically? After I had written my program, I noticed that the three Bob methods are really the only true extents for 4 bells, but that because they each have eight different but equivalent variations there are 24 valid extents for 4 bells. This brings up other interesting questions that I did not have time to pursue: Is there a mathematical reason that all valid extents on 4 bells are repeated sequences of eight transitions? Is there a reason that the number of valid extents was the same as the number of possible changes?

In conclusion change ringing is a fascinating subject with many different aspects and I have enjoyed exploring it in this project.

## 6 References

The visited on date of the references is given in the format month/day/year.

- [1] FABIA WEBER. “Campanology - Ringing the changes”. In: *ETH Department of Mathematics* (Aug. 2017).
- [2] GINTAUTAS ŽALĖNAS. “Cum signo campanae. The origin of the bells in Europe.” In: (Aug. 2013), pp. 65–92. ISSN: 1822-4555. URL: [https://www.academia.edu/22415048/Cum\\_signo\\_campanae\\_The\\_origin\\_of\\_the\\_bells\\_in\\_Europe](https://www.academia.edu/22415048/Cum_signo_campanae_The_origin_of_the_bells_in_Europe) (visited on 09/05/2024).
- [3] ERNEST MORRIS. *‘The History and Art of Change-Ringing’*. London: Chapman & Hall Ltd., 1931. URL: <https://www.whittingsociety.org.uk/old-ringing-books/morris-history-and-art-of-change-ringing.html> (visited on 04/25/2024).
- [4] THOMAS LESLIE PAPILLON. *Church Bells and Bell-Ringing*. London: “The Guardian”, 1909. URL: <https://www.whittingsociety.org.uk/old-ringing-books/papillon-church-bells-and-bell-ringing-01.pdf> (visited on 06/29/2024).
- [5] GRACE CHAPMAN. *BBC Four - Still Ringing After All These Years: A Short History of Bells*. Dec. 2011. URL: <https://www.youtube.com/watch?v=OR32AT1QaF8> (visited on 07/11/2024).
- [6] BELLRINGINGFILM. *The Craft of Bellringing*. Sept. 2014. URL: <https://www.youtube.com/watch?v=yLMiK-TMyPI> (visited on 04/30/2024).
- [7] UW ETHNOMUSICOLOGY ARCHIVES. *Idiophones - UW Ethnomusicology Archives - Library Guides at University of Washington Libraries*. Aug. 2024. URL: <https://guides.lib.uw.edu/archives/idiophones> (visited on 09/05/2024).
- [8] THE NATIONAL BELL FESTIVAL INC. *Bell History — National Bell Festival*. URL: <https://www.bells.org/bell-history> (visited on 09/03/2024).
- [9] MATTHEW ERNEST. *ASK A PRIEST*. 2019. URL: <https://archwaysmag.org/when-and-why-do-the-altar-servers-ring-a-bell-at-mass> (visited on 09/05/2024).
- [10] MEDIEVAL HISTORIES. *Early Medieval Irish Hand-Bells*. Apr. 2017. URL: <https://www.medieval.eu/early-medieval-irish-hand-bells/> (visited on 09/05/2024).
- [11] JOHN H. ARNOLD and CAROLINE GOODSON. “Resounding Community: The History and Meaning of Medieval Church Bells”. en. In: *Viator* 43.1 (Jan. 2012), pp. 99–130. ISSN: 0083-5897, 2031-0234. DOI: 10.1484/J.VIATOR.1.102544. URL: <https://www.brepolsonline.net/doi/10.1484/J.VIATOR.1.102544> (visited on 09/05/2024).
- [12] ONLINE ETYMOLOGY DICTIONARY. *clock — Search Online Etymology Dictionary*. Aug. 2020. URL: <https://www.etymonline.com/search?q=clock> (visited on 09/05/2024).
- [13] MERRIAM-WEBSTER DICTIONARY. *Definition of SANCTUS BELL*. URL: <https://www.merriam-webster.com/dictionary/Sanctus+bell> (visited on 09/15/2024).
- [14] MERRIAM-WEBSTER DICTIONARY. *Definition of SANCTUS*. URL: <https://www.merriam-webster.com/dictionary/Sanctus> (visited on 09/15/2024).
- [15] MERRIAM-WEBSTER DICTIONARY. *Definition of THANE*. URL: <https://www.merriam-webster.com/dictionary/thane> (visited on 07/18/2024).
- [16] THE NATIONAL BELL FESTIVAL INC. *Meet St. Dunstan, Patron Saint of Bell Ringers — National Bell Festival*. URL: <https://www.bells.org/blog/meet-st-dunstan-patron-saint-bell-ringers> (visited on 07/17/2024).
- [17] THE NATIONAL BELL FESTIVAL INC. *Learn about the Different Parts of a Bell — National Bell Festival*. URL: <https://www.bells.org/activity/learn-about-different-parts-bell> (visited on 09/29/2024).
- [18] MERRIAM-WEBSTER DICTIONARY. *Definition of TRANSUBSTANTIATION*. July 2024. URL: <https://www.merriam-webster.com/dictionary/transubstantiation> (visited on 09/15/2024).
- [19] STADT ZÜRICH. *Städtische Läuteordnung*. June 1969. URL: <https://www.stadt-zuerich.ch/content/dam/stzh/portal/Deutsch/AmtlicheSammlung/Erlasse/713/420/713.420%20St%C3%A4dtische%20L%C3%A4uteordnung%20V1.pdf> (visited on 10/15/2024).
- [20] *Schweiz aktuell - In Maschwanden läuten die Glocken weiterhin morgens um 04:45 - Play SRF*. June 2019. URL: <https://www.srf.ch/play/tv/schweiz-aktuell/video/in-maschwanden-laeuten-die-glocken-weiterhin-morgens-um-0445?urn=urn:srf:video:dde6cbd9-1aa8-4f65-94d5-f5646d12b970> (visited on 10/15/2024).
- [21] JOHN ROBERT NICHOLS. *Bells Thro’ the Ages The Founders’ Craft and Ringers’ Art*. London: Chapman & Hall Ltd., 1928.
- [22] DEUTSCHE GLOCKENSPIELVEREINIGUNG E.V. *Was ist ein Carillon?* Oct. 2024. URL: <https://glockenspieler.de/was-ist-ein-carillon> (visited on 10/11/2024).

- [23] MATTHIAS WALTER. “Das Walliser Carillon”. In: (2021). URL: <https://www.vs.ch/documents/249470/14131302/Das+Walliser+Carillon.pdf/7532b8a4-e9d0-74d1-6bb7-36b2e2b2ab6a?t=1648198350562> (visited on 10/11/2024).
- [24] Bible Gateway passage: Psalm 22:22 - New International Version. en. URL: <https://www.biblegateway.com/passage/?search=Psalm%2022%3A22&version=NIV> (visited on 10/15/2024).
- [25] MARK CARTWRIGHT. *English Reformation - World History Encyclopedia*. July 2020. URL: [https://www.worldhistory.org/English\\_Reformation/](https://www.worldhistory.org/English_Reformation/) (visited on 09/21/2024).
- [26] KERSTEN KNIPP. *When church bells were transformed into weapons of war - DW - 09/21/2018*. Sept. 2018. URL: <https://www.dw.com/en/when-church-bells-were-transformed-into-weapons-of-war/a-45576884> (visited on 10/10/2024).
- [27] STEPHEN J. THORNE. *The seizing of Europe’s bells*. Nov. 2018. URL: <https://legionmagazine.com/the-seizing-of-europes-bells/> (visited on 10/10/2024).
- [28] GERMANISCHES NATIONALMUSEUM. *Glockenlager im Hamburger Hafen*. URL: <https://www.gnm.de/objekte/glockenlager-im-hamburger-hafen> (visited on 10/10/2024).
- [29] THE NATIONAL BELL FESTIVAL INC. *When Nazis Took All the Bells — National Bell Festival*. URL: <https://www.bells.org/blog/when-nazis-took-all-bells> (visited on 10/10/2024).
- [30] *The History of Ringing*. URL: <https://cccbr.org.uk/the-history-of-ringing/> (visited on 09/22/2024).
- [31] REV. H. EARLE BULWER. *Glossary of Technical Terms*. 1901. URL: <https://www.whittingsociety.org.uk/old-ringing-books/bulwer-glossary-ringing-terms.html> (visited on 10/11/2024).
- [32] RICHARD DUCKWORTH. *Tintinnalogia*. en. Last Modified: 2024-07-08T12:03:03.780600+00:00. 1668. URL: <https://www.gutenberg.org/files/18567/18567-h/18567-h.htm> (visited on 08/02/2024).
- [33] FABIAN STEDMAN. *Campanalogia*. Last Modified: 2024-07-30T11:45:50.723876+00:00. 1677. URL: <https://www.gutenberg.org/cache/epub/73423/pg73423-images.html> (visited on 09/29/2024).
- [34] *Ancient Society of College Youths*. URL: <https://www.ascy.org.uk/> (visited on 10/13/2024).
- [35] *The Central Council of Church Bell Ringers*. URL: <https://cccbr.org.uk/> (visited on 10/13/2024).
- [36] R. MURRAY SCHAFER. *The soundscape: our sonic environment and the tuning of the world*. Rochester, Vermont: Destiny Books [u.a.], 2006. ISBN: 978-0-89281-455-8.
- [37] ARTHUR T. WHITE. “Ringing the changes”. In: *Mathematical Proceedings of the Cambridge Philosophical Society* 94.2 (Sept. 1983), pp. 203–215. ISSN: 0305-0041, 1469-8064. DOI: 10.1017/S0305004100061053. URL: [https://www.cambridge.org/core/product/identifier/S0305004100061053/type/journal\\_article](https://www.cambridge.org/core/product/identifier/S0305004100061053/type/journal_article) (visited on 06/30/2024).
- [38] JEFF LADD. *12: Extents — St Georges Bells - France’s First Set of English Change-Ringing Bells - Vernet-les-Bains - Pyrénées Orientales*. Aug. 2017. URL: <https://vernetbells.com/en/lessons/12-extents> (visited on 08/22/2024).
- [39] JAEYI SONG and SOPHIA HOU. *Group Theory*. 2022. URL: <https://math.mit.edu/research/highschool/primes/circle/documents/2022/Sophia%20%20Jaeyi.pdf> (visited on 09/02/2024).
- [40] WERNER DURANDI et al. *Formulae, Tables and Concepts:: a concise handbook of mathematics - physics - chemistry*. Accepted: 2017-06-11T14:47:20Z. Orell Füssli, 2014. ISBN: 978-3-280-04084-3. URL: <https://www.research-collection.ethz.ch/handle/20.500.11850/94744> (visited on 09/07/2024).
- [41] GUERINO MAZZOLA, MARIA MANNONE, and YAN PANG. *Cool Math for Hot Music*. Computational Music Science. Cham: Springer International Publishing, 2016. ISBN: 978-3-319-42935-9 978-3-319-42937-3. DOI: 10.1007/978-3-319-42937-3. URL: <http://link.springer.com/10.1007/978-3-319-42937-3> (visited on 09/07/2024).
- [42] JOHN D. DIXON and BRIAN MORTIMER. *Permutation Groups*. Springer-Verlag New York, Inc., 1996. ISBN: 0-387-94599-7. URL: <https://link.springer.com/book/10.1007/978-1-4612-0731-3>.
- [43] TOM DENTON. *3.1: Generating Sets*. Nov. 2013. URL: [https://math.libretexts.org/Bookshelves/Abstract\\_and\\_Geometric\\_Algebra/Introduction\\_to\\_Algebraic\\_Structures\\_\(Denton\)/03%3A\\_Groups\\_II/3.01%3A\\_Generating\\_Sets](https://math.libretexts.org/Bookshelves/Abstract_and_Geometric_Algebra/Introduction_to_Algebraic_Structures_(Denton)/03%3A_Groups_II/3.01%3A_Generating_Sets) (visited on 09/08/2024).
- [44] SRIDHER KAMINANI. “Finding Hamiltonian Cycles”. In: (Aug. 2005). URL: <https://core.ac.uk/download/pdf/43618678.pdf>.
- [45] TORSTEN MÜTZE. “Combinatorial Gray Codes—An Updated Survey”. In: (July 2024). URL: <https://arxiv.org/pdf/2202.01280>.
- [46] LAM DUONG. *Johnson Trotter Algorithm - Generate Permutations*. Mar. 2020. URL: [https://www.youtube.com/watch?v=LBuERmz\\_BCM](https://www.youtube.com/watch?v=LBuERmz_BCM) (visited on 10/09/2024).
- [47] SELMER M JOHNSON. “Generation of Permutations by Adjacent Transposition”. In: 17 (1963), pp. 282–285. URL: <https://api.semanticscholar.org/CorpusID:119440379> (visited on 10/09/2024).

- [48] HALE F. TROTTER. “Algorithm 115: Perm”. In: *Communications of the ACM* 5.8 (Aug. 1962), pp. 434–435. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/368637.368660. URL: <https://dl.acm.org/doi/10.1145/368637.368660> (visited on 10/09/2024).
- [49] DONALD E. KNUTH. *The Art of Computer Programming*. Vol. 4. 2005. ISBN: 0-201-85393-0.
- [50] *Thonny, Python IDE for beginners*. URL: <https://thonny.org/> (visited on 09/08/2024).
- [51] MICHAEL GALARNYK. *Python Lists and List Manipulation Tutorial*. URL: <https://builtin.com/data-science/python-list> (visited on 08/03/2024).
- [52] GEEKSFORGEEKS. *Python Program to Swap Two Elements in a List*. Section: Python. Nov. 2018. URL: <https://www.geeksforgeeks.org/python-program-to-swap-two-elements-in-a-list/> (visited on 08/03/2024).
- [53] PYTHONTM. *ForLoop - Python Wiki*. Jan. 2022. URL: <https://wiki.python.org/moin/ForLoop> (visited on 08/03/2024).
- [54] GEEKSFORGEEKS. *Python — Convert a list of characters into a string*. Section: Python. Dec. 2017. URL: <https://www.geeksforgeeks.org/python-convert-list-characters-string/> (visited on 08/10/2024).
- [55] CSROCKS. *What Is Recursion - Recursion Explained In 3 Minutes*. July 2017. URL: [https://www.youtube.com/watch?v=YZc0\\_jRhvx8](https://www.youtube.com/watch?v=YZc0_jRhvx8) (visited on 10/08/2024).
- [56] DANIEL BURRUECO. *random.randrange — Interactive Chaos*. Feb. 2021. URL: <https://interactivechaos.com/en/python/function/randomrandrange> (visited on 10/09/2024).
- [57] *about - pygame wiki*. URL: <https://www.pygame.org/wiki/about> (visited on 10/09/2024).
- [58] MALLI. *Convert List of Tuples to List of Lists in Python*. Feb. 2023. URL: <https://sparkbyexamples.com/python/convert-list-of-tuples-to-list-of-lists-in-python/> (visited on 08/10/2024).
- [59] GEEKSFORGEEKS. *Generate all permutation of a set in Python*. Section: Algorithms. Jan. 2016. URL: <https://www.geeksforgeeks.org/generate-all-the-permutation-of-a-list-in-python/> (visited on 10/08/2024).
- [60] *How can I randomly select an item from a list in Python? — Better Stack Community*. URL: <https://betterstack.com/community/questions/python-how-to-randomly-select-list-item/> (visited on 08/10/2024).
- [61] *Get the size (length, number of items) of a list in Python — note.nkmm.me*. Aug. 2023. URL: <https://note.nkmm.me/en/python-list-len/> (visited on 08/10/2024).
- [62] GEEKSFORGEEKS. *How to Replace Values in a List in Python?* Section: Python. Nov. 2021. URL: <https://www.geeksforgeeks.org/how-to-replace-values-in-a-list-in-python/> (visited on 08/10/2024).
- [63] GEEKSFORGEEKS. *Python — Check if two lists are identical*. Section: Python. Nov. 2018. URL: <https://www.geeksforgeeks.org/python-check-if-two-lists-are-identical/> (visited on 08/05/2024).

## List of Tables

1	Assuming that 30 changes can be rung per minute, we obtain in the rightmost column the time required to ring a given extent. cf. [1], Table 1.1 . . . . .	11
2	Some extents on $n = 4$ bells (Minimus) cf. [1], Table 2.4 . . . . .	14

## List of Figures

1	Bell from Herculaneum, 1st century AD cf. [2], figure 10 . . . . .	2
2	The Cloc ind Édachta, the bell of St. Patrick, said to be the oldest preserved bell from Ireland [10] . . . . .	3
3	The parts of a church bell [17] . . . . .	4
4	The largest in the belltower of Maschwanden, Switzerland Inscription: ICH WILL DEINEN NAMEN MEINEN BRÜDERN VERKÜNDEN INMITTEN DER GERMEINDE WILL ICH DICH LOBEN PS. 22. 23. (“I will declare your name to my people; in the assembly I will praise you.” [24]) . . . . .	5
5	Glockenfriedhof in Hamburg’s harbor [28] . . . . .	6
6	The evolution of methods of hanging bells and up and down strokes on the different mounts [30] . . . . .	7
7	Bell hung on a full wheel in a wooden frame [3] . . . . .	7
8	The two strokes, that can be used to ring an upset bell [3] . . . . .	8

9	Blue bell plain hunting . . . . .	9
10	Green and red bell dodging . . . . .	9
11	Illustration of three function types: injective, surjective and bijective <i>cf.</i> [41], <i>Figure 4.6</i> .	12
12	The 3-bell Cayley graph $C_S(S_3)$ , where rounds are at the bottom left vertex and marked in blue. <i>cf.</i> [1], <i>Figure 3.1</i> . . . . .	15
13	Verticies of the Cayley graph of $C_S(S_4)$ , showing the change corresponding to each vertex. Rounds are marked in blue and framed with a box. . . . .	16
14	The 4-bell Cayley graph $C_S(S_4)$ <i>cf.</i> [1], <i>Figure 3.2</i> . . . . .	17
15	Here, the upper-left vertex of the inner hexagon of $C_S(S_4)$ is labeled by rounds in dark sky blue, so it is the start for all Hamiltonian cycles. The extents shown here are: Plain Bob Minimus ( $((AB)^3AC)^3$ ), Reverse Bob Minimus ( $((ABAD(AB)^2)^3)$ ), Double Bob Minimus ( $((ABADABAC)^3)$ ), Canterbury Minimus ( $((ABCD CBAB)^3)$ ), Reverse Canterbury Minimus ( $((DB(AB)^2DC)^3)$ (as seen in example 3.2.8)6, Double Canterbury Minimus ( $((DBCD CBDC)^3)$ ), Single Court Minimus ( $((DB(AB)^2DB)^3)$ ), Reverse Court Minimus ( $((AB(CB)^2AB)^3)$ ), Double Court Minimus ( $((DB(CB)^2DB)^3)$ ), St. Nicholas Minimus ( $((DBADABDC)^3)$ and to Reverse St. Nicholas Minimus ( $((ABCD CBAC)^3)$ . <i>cf.</i> [1], <i>Figure 3.4</i> . . . . .	18
16	The Cayley graphs of the eight variants of the Plain Bob Minimus . . . . .	29
17	Link to a demonstration of the program in action . . . . .	30

## Acknowledgments

Special thanks to Gregory Bloch for the idea for this project and to my supervisor Mr. Pfenninger for patiently answering my many questions. I would also like to thank Silvia Bachmann and the Kirchenkommission of Maschwanden, for letting me record the bells of the church in Maschwanden. I am very grateful to my father, Daniel Dulay, for helping me with the programming, and to my mother, Susannah Bloch, for proofreading and correcting the final paper. Thank you also to my other proofreaders Sheri Bloch, Sombo Em and Daniel Holzner.

## Einhaltung rechtlicher Vorgaben

Ich habe die Arbeit selbstständig und unter Aufsicht meines Betreuers/meiner Betreuerin verfasst und keine anderen als die angegebenen Hilfsmittel verwendet.

Abschnitte, für deren Erstellung KI-Programme (bspw. ChatGPT) zum Einsatz kamen, habe ich allesamt offengelegt und mit einer entsprechenden Fussnote versehen. Meine Arbeit wird gegebenenfalls einer Prüfung bezüglich KI-Einsatz unterzogen; im Rahmen dieser Prüfung wird festgestellt, ob die Arbeit neben den angegebenen Stellen weitere von einer KI verfasste Elemente enthält.

Ich nehme darüber hinaus zur Kenntnis, dass meine Arbeit zur Überprüfung der korrekten und vollständigen Angabe der Quellen mit Hilfe einer Software (eines Plagiaterkennungstools) geprüft wird. Zu meinem eigenen Schutz wird diese Software auch dazu verwendet, später eingereichte Arbeiten mit meiner Arbeit elektronisch zu vergleichen und damit Abschriften und eine Verletzung meines Urheberrechts zu verhindern. Ich erkläre mich damit einverstanden, dass die Schulleitung bei Verdacht auf Urheberrechtsverletzung meine Arbeit zu Prüfzwecken herausgibt..

Datum, Unterschrift



## A Appendix: Full Process of Programming

### A.1 Starting With Changes on 3 Bells

#### A.1.1 Printing a List

```

1  z = [1, 2, 3] #define a list
2  print(z)

[51]
```

#### A.1.2 Swapping the Positions of Two Items in a List

```

1  def swapPositions(z, pos1, pos2):
2
3      # Storing the two elements as a pair in a tuple variable as get
4      get = z[pos1], z[pos2]
5
6      # unpacking those elements
7      z[pos2], z[pos1] = get
8
9      return z
10
11 z = [1, 2, 3]
12 pos1, pos2 = 0, 1
13
14 print(z)
15 print(swapPositions(z, pos1, pos2))
```

#### A.1.3 Extent on 3 Bells

```

1  def swapPositions(z, pos1, pos2):
2      get = z[pos1], z[pos2]
3      z[pos2], z[pos1] = get
4      return z
5
6  z = [1, 2, 3]
7  pos1, pos2 = 0, 1
8
9  print(z)
10 print(swapPositions(z, pos1, pos2))
11 print(swapPositions(z, pos1+1, pos2+1))
12 print(swapPositions(z, pos1, pos2))
13 print(swapPositions(z, pos1+1, pos2+1))
14 print(swapPositions(z, pos1, pos2))
15 print(swapPositions(z, pos1+1, pos2+1))
16 #Full change on 3 bells
```

#### A.1.4 Extent on 3 Bells With Loop

```

1  def swapPositions(z, pos1, pos2):
2      get = z[pos1], z[pos2]
3      z[pos2], z[pos1] = get
4      return z
5
6  def swap01():
7      print(swapPositions(z, pos1, pos2))
8
9  def swap12():
10     print(swapPositions(z, pos1+1, pos2+1))
11
12 z1 = [1, 2, 3]
13 z = [1, 2, 3]
14 pos1, pos2 = 0, 1
15
16 print(z)
17 for i in range(10):
18     swap01()
19     swap12()
20     if z != z1:
21         continue
22     else:
```

```
23         break
```

[53]

## A.2 Programmed Extents on 4 Bells

### A.2.1 Plain Bob

```
1  def swapPositions(z, pos1, pos2):
2      get = z[pos1], z[pos2]
3      z[pos2], z[pos1] = get
4      return z
5
6  def swap0123():
7      swapPositions(z, pos1, pos2-1)
8      print(swapPositions(z, pos2, pos2+1))
9
10 def swap12():
11     print(swapPositions(z, pos1+1, pos2))
12
13 def swap23():
14     print(swapPositions(z, pos2, pos2+1))
15
16 def swap01():
17     print(swapPositions(z, pos1, pos2-1))
18
19 z1 = [1, 2, 3, 4]
20 z = [1, 2, 3, 4]
21 pos1, pos2 = 0, 2
22
23 print(z)
24 for i in range(10):
25     for i in range(3):
26         swap0123()
27         swap12()
28         swap0123()
29         swap23()
30         if z != z1:
31             continue
32         else:
33             break
34 print('done.')
```

### A.2.2 Reverse Canterbury

```
1  def swapPositions(z, pos1, pos2):
2      get = z[pos1], z[pos2]
3      z[pos2], z[pos1] = get
4      return z
5
6  def swap0123():
7      swapPositions(z, pos1, pos2-1)
8      print(swapPositions(z, pos2, pos2+1))
9
10 def swap12():
11     print(swapPositions(z, pos1+1, pos2))
12
13 def swap23():
14     print(swapPositions(z, pos2, pos2+1))
15
16 def swap01():
17     print(swapPositions(z, pos1, pos2-1))
18
19 z1 = [1, 2, 3, 4]
20 z = [1, 2, 3, 4]
21 pos1, pos2 = 0, 2
22
23 print(z)
24 for i in range(10):
25     swap01()
26     swap12()
27     swap0123()
28     swap12()
```

```

29     swap0123()
30     swap12()
31     swap01()
32     swap23()
33     if z != z1:
34         continue
35     else:
36         break
37 print('done.')
38

```

### A.2.3 Double Court

```

1  def swapPositions(z, pos1, pos2):
2      get = z[pos1], z[pos2]
3      z[pos2], z[pos1] = get
4      return z
5
6  def swap0123():
7      swapPositions(z, pos1, pos2-1)
8      print(swapPositions(z, pos2, pos2+1))
9
10 def swap12():
11     print(swapPositions(z, pos1+1, pos2))
12
13 def swap23():
14     print(swapPositions(z, pos2, pos2+1))
15
16 def swap01():
17     print(swapPositions(z, pos1, pos2-1))
18
19 z1 = [1, 2, 3, 4]
20 z = [1, 2, 3, 4]
21 pos1, pos2 = 0, 2
22
23 print(z)
24 for i in range(10):
25     swap01()
26     swap12()
27     for n in range(2):
28         swap23()
29         swap12()
30     swap01()
31     swap12()
32     if z != z1:
33         continue
34     else:
35         break
36 print('done.')

```

## A.3 Generating a Random Set of Changes

### A.3.1 Create Change on 3 Bells

```

1  def swapPositions(z, pos1, pos2):
2      get = z[pos1], z[pos2]
3      z[pos2], z[pos1] = get
4      return z
5
6  def swap01():
7      print(swapPositions(z, pos1, pos2))
8      already_done.append(z.copy())
9
10 def swap12():
11     print(swapPositions(z, pos1+1, pos2+1))
12     already_done.append(z.copy())
13
14 z1 = [1, 2, 3]
15 z = [1, 2, 3] #define a list
16 pos1, pos2 = 0, 1
17 already_done = []
18
19 print(z)

```

```

20 for i in range(10):
21     swap01()
22     swap12()
23     if z != z1:
24         continue
25     else:
26         break
27
28 print('we done')
29
30 print(already_done)
31 already_done.sort() #sort list so it's the same to compare?
32 print(already_done)
33 print('mmm')
34
35 from itertools import permutations
36
37 perm = permutations([1, 2, 3])
38 perm_list_tuple = list(perm)
39 perm_list_list = []
40 for tup in perm_list_tuple:
41     perm_list_list.append(list(tup))
42 print(perm_list_list)
43
44 if perm_list_list == already_done:
45     print('weee')
46 else:
47     print('not quite')
48
[58][59]

```

### A.3.2 Creating a Random Set of Changes on 3 Bells

```

1 import random
2 from itertools import permutations
3
4
5 def swapPositions(z, pos1, pos2):
6     get = z[pos1], z[pos2]
7     z[pos2], z[pos1] = get
8     return z
9
10 def swap(pos):
11     print(swapPositions(z, pos+1, pos))
12     already_done.append(z.copy())
13
14 all_possible_swaps = [0, 1]
15
16 def randomswap1(possible_swaps):
17     selected_swap1 = random.choice(possible_swaps)
18     print("selected swap = ",selected_swap1)
19     swap(selected_swap1)
20     next_swaps = all_possible_swaps.copy()
21     next_swaps.remove(selected_swap1)
22     return next_swaps
23
24 z1 = [1, 2, 3]
25 z = [1, 2, 3]
26 already_done = []
27
28 print(z)
29 next = randomswap1(all_possible_swaps)
30 for i in range(44):
31     next = randomswap1(next)
32     if z != z1:
33         continue
34     else:
35         break
36
37 print('we done')
38

```

```

39 print(already_done)
40 already_done.sort()
41 print("sorted:", already_done)
42
43
44 perm = permutations([1, 2, 3])
45 perm_list_tuple = list(perm)
46 perm_list_list = []
47 for tup in perm_list_tuple:
48     perm_list_list.append(list(tup))
49 print(perm_list_list)
50
51 if perm_list_list == already_done:
52     print('the same')
53 else:
54     print('not quite')

```

[60]

### A.3.3 Random Changes on 4 Bells of a Random Length

```

1  from itertools import permutations
2  import random
3
4  def swapPositions(z, pos1, pos2):
5      get = z[pos1], z[pos2]
6      z[pos2], z[pos1] = get
7
8      return z
9
10 def swap(pos):
11     if pos == 3:
12         swapPositions(z, pos-3, pos-2)
13         print(swapPositions(z, pos-1, pos))
14         already_done.append(z.copy())
15     else:
16         print(swapPositions(z, pos+1, pos))
17         already_done.append(z.copy())
18
19 all_possible_swaps = [0, 1, 2, 3]
20
21 def randomswap(possible_swaps):
22     selected_swap1 = random.choice(possible_swaps)
23     #print("selected swap = ",selected_swap1)
24     swap(selected_swap1)
25     Transition.append(selected_swap1)
26     next_swaps = all_possible_swaps.copy()
27     next_swaps.remove(selected_swap1)
28     return next_swaps
29
30 def convert(s):
31     str1 = ""
32     return(str1.join(s))
33
34 z1 = [1, 2, 3, 4]
35 z = [1, 2, 3, 4]
36 already_done = []
37 Transition = []
38
39 print(z)
40 next = randomswap(all_possible_swaps)
41 for i in range(100):
42     next = randomswap(next)
43     if z != z1:
44         continue
45     else:
46         print("we're back at the start")
47         break
48
49 print(already_done)
50 already_done.sort()
51 print("sorted:", already_done)
52 print("length:", len(already_done))

```

```

53
54 for i in range(len(Transition)): # for same notation as campanology
55     if Transition[i] == 0:
56         Transition[i] = 'D'
57     if Transition[i] == 2:
58         Transition[i] = 'C'
59     if Transition[i] == 1:
60         Transition[i] = 'B'
61     if Transition[i] == 3:
62         Transition[i] = 'A'
63 print("A = (12)(34), B = (23), C = (34), D = (12)")
64 print("change:", convert(Transition))
65
66
67 perm = permutations([1, 2, 3, 4])
68 perm_list_tuple = list(perm)
69 perm_list_list = []
70 for tup in perm_list_tuple:
71     perm_list_list.append(list(tup))
72 print("all permutations:", perm_list_list)
73 print("how many permutations?", len(perm_list_list))
74
75 if perm_list_list == already_done:
76     print('The lists are the same')
77 else:
78     print('The lists are not quite the same')

```

[61][62]

---

Example shell<sup>7</sup>:

```

1  [1, 2, 3, 4]
2  we're back at the start
3  [[1, 3, 2, 4], [3, 1, 4, 2], [1, 3, 4, 2], [1, 4, 3, 2], [4, 1, 3, 2], [1, 4, 2, 3],
4  [1, 2, 4, 3], [2, 1, 3, 4], [2, 3, 1, 4], [3, 2, 1, 4], [3, 1, 2, 4], [3, 1, 4, 2],
5  [3, 4, 1, 2], [3, 4, 2, 1], [3, 2, 4, 1], [3, 2, 1, 4], [2, 3, 4, 1], [2, 4, 3, 1],
6  [2, 4, 1, 3], [4, 2, 3, 1], [4, 3, 2, 1], [3, 4, 1, 2], [4, 3, 1, 2], [4, 3, 2, 1],
7  [3, 4, 1, 2], [3, 1, 4, 2], [1, 3, 4, 2], [1, 3, 2, 4], [1, 2, 3, 4]]
8  sorted: [[1, 2, 3, 4], [1, 2, 4, 3], [1, 3, 2, 4], [1, 3, 2, 4], [1, 3, 4, 2],
9  [1, 3, 4, 2], [1, 4, 2, 3], [1, 4, 3, 2], [2, 1, 3, 4], [2, 3, 1, 4], [2, 3, 4, 1],
10 [2, 4, 1, 3], [2, 4, 3, 1], [3, 1, 2, 4], [3, 1, 4, 2], [3, 1, 4, 2], [3, 1, 4, 2],
11 [3, 2, 1, 4], [3, 2, 1, 4], [3, 2, 4, 1], [3, 4, 1, 2], [3, 4, 1, 2], [3, 4, 1, 2],
12 [3, 4, 2, 1], [4, 1, 3, 2], [4, 2, 3, 1], [4, 3, 1, 2], [4, 3, 2, 1], [4, 3, 2, 1]]
13 length: 29
14 A = (12)(34), B = (23), C = (34), D = (12)
15 change: BADBDABABDBCBCBCBACABADCBADCB
16 all permutations: [[1, 2, 3, 4], [1, 2, 4, 3], [1, 3, 2, 4], [1, 3, 4, 2],
17 [1, 4, 2, 3], [1, 4, 3, 2], [2, 1, 3, 4], [2, 1, 4, 3], [2, 3, 1, 4], [2, 3, 4, 1],
18 [2, 4, 1, 3], [2, 4, 3, 1], [3, 1, 2, 4], [3, 1, 4, 2], [3, 2, 1, 4], [3, 2, 4, 1],
19 [3, 4, 1, 2], [3, 4, 2, 1], [4, 1, 2, 3], [4, 1, 3, 2], [4, 2, 1, 3], [4, 2, 3, 1],
20 [4, 3, 1, 2], [4, 3, 2, 1]]
21 how many permutations? 24
22 The lists are not quite the same

```

## A.4 Extents on 4 Bells

### A.4.1 Program Finds All Possible Extents on 4 Bells

```

1  #!/usr/bin/python3
2
3  from itertools import permutations
4  import random
5
6  # def convert(s):
7  #     str1 = ""
8  #     return(str1.join(s))
9
10 def swapPositions(r, pos1, pos2):
11     get = r[pos1], r[pos2]
12     r[pos2], r[pos1] = get

```

---

<sup>7</sup>For the example shell the prints on lines 13 and 16 were taken out leaving just the list `already_done` to be printed to show, what permutations it went through.

```

13     return r
14
15 def swap(row, pos):
16     new_row = row.copy()
17     if pos == 3:
18         swapPositions(new_row, pos-3, pos-2)
19         swapPositions(new_row, pos-1, pos)
20     # already_done.append(new_row.copy())
21     else:
22         swapPositions(new_row, pos+1, pos)
23     # already_done.append(row.copy())
24     return new_row
25
26 #def pos_change_check(previous_rows: list[list[int]], next_row: list[int], pos)
27
28
29 def test_rows(previous_rows: list[list[int]], next_row: list[int]) -> bool:
30     if len(previous_rows) != 24 and next_row in previous_rows:
31         return False
32
33     for i in range(len(next_row)):
34         if (previous_rows[-2][i] == previous_rows[-1][i] and
35             previous_rows[-2][i] == next_row[i]):
36             return False
37     else: return True
38
39 all_possible_pos = [0, 1, 2, 3]
40
41 def intersection(lst1, lst2):
42     lst3 = [value for value in lst1 if value in lst2]
43     return lst3
44
45 def extend_rows(rows: list[list[int]], positions: list[int], avail_pos:
46     list[int]) -> tuple[list[list[list[int]]], list[list[int]]]:
47     result: list[list[list[int]]] = []
48     result_pos: list[list[int]] = []
49     if len(rows) == 25 and rows[-1] == [1, 2, 3, 4]:
50         # print(f'found it {rows}')
51         return [rows], [positions]
52     if len(rows) >= 25:
53         # print(f'overdone {rows}')
54         return result, result_pos
55     for pos in avail_pos:
56         # print(f'testing {rows} move {pos}')
57         new_row = swap(rows[-1], pos)
58         if test_rows(rows, new_row):
59             # print(f'{len(rows)} - extending {rows} with {new_row}.')
60             new_rows = rows.copy()
61             new_rows.append(new_row)
62             new_positions = positions.copy()
63             new_positions.append(pos)
64             all_possible_pos.copy()
65             new_avail_pos = all_possible_pos.copy()
66             new_avail_pos.remove(pos)
67             new_result, new_result_pos = extend_rows(new_rows, new_positions, new_avail_pos)
68             if new_result:
69                 result.extend(new_result)
70                 result_pos.extend(new_result_pos)
71
72     return result, result_pos
73
74 def Translate_pos_alphabet(pos_list: list[int]) -> list[str]:
75     alpha_pos_list: list[str] = []
76     alpha_pos_map = {0: 'D', 1: 'B', 2: 'C', 3: 'A'}
77     for pos in pos_list:
78         alpha_pos_list.append(alpha_pos_map[pos])
79     return alpha_pos_list
80
81
82 z1 = [1, 2, 3, 4]
83 z = [1, 2, 3, 4]
84 Len_int_ad = [0]
85 already_done = []

```

```

86 Transition = []
87
88 print("possible permutations:")
89 perm = permutations([1, 2, 3, 4])
90 perm_list_tuple = list(perm)
91 perm_list_list = []
92 for tup in perm_list_tuple:
93     perm_list_list.append(list(tup))
94 print("all permutations:", perm_list_list)
95 print("how many permutations?", len(perm_list_list))
96
97
98 # Main loop to build extents.
99 final_result: list[list[list[int]]] = []
100 final_result_pos: list[list[int]] = []
101 for pos in all_possible_pos:
102     print(f'Starting with move {pos}')
103     new_row = swap(z, pos)
104     new_avail_pos = all_possible_pos.copy()
105     new_avail_pos.remove(pos)
106     new_result, new_result_pos = extend_rows([z, new_row], [pos], new_avail_pos)
107     if new_result:
108         final_result.extend(new_result)
109         final_result_pos.extend(new_result_pos)
110
111 print("A = (12)(34), B = (23), C = (34), D = (12)")
112 for i in range(len(final_result)):
113     print(f'{Translate_pos_alphabet(final_result_pos[i])}')
114     print(f'{final_result[i]}')

```

[52][54][63]

#### A.4.2 Program Finds Extents on 4 Bells With Sounds

```

1  from itertools import permutations
2  import random
3  from pygame import mixer
4  from pygame import time
5
6  def swapPositions(r, pos1, pos2):
7      get = r[pos1], r[pos2]
8      r[pos2], r[pos1] = get
9
10     return r
11
12 def swap(row, pos):
13     new_row = row.copy()
14     if pos == 3:
15         swapPositions(new_row, pos-3, pos-2)
16         swapPositions(new_row, pos-1, pos)
17     #     already_done.append(new_row.copy())
18     else:
19         swapPositions(new_row, pos+1, pos)
20     #     already_done.append(row.copy())
21     return new_row
22
23 def test_rows(previous_rows: list[list[int]], next_row: list[int]) -> bool:
24     if len(previous_rows) != 24 and next_row in previous_rows:
25         return False
26
27     for i in range(len(next_row)):
28         if (previous_rows[-2][i] == previous_rows[-1][i] and
29             previous_rows[-2][i] == next_row[i]):
30             return False
31
32     else: return True
33
34 all_possible_pos = [0, 1, 2, 3]
35
36 def extend_rows(rows: list[list[int]], positions: list[int], avail_pos:
37     list[int]) -> tuple[list[list[int]], list[list[int]]]:
38     result: list[list[list[int]]] = []
39     result_pos: list[list[int]] = []

```



```

40     if len(rows) == 25 and rows[-1] == [1, 2, 3, 4]:
41         # print(f'found it {rows}')
42         return [rows], [positions]
43     if len(rows) >= 25:
44         print(f'overdone {rows}')
45         return result, result_pos
46     for pos in avail_pos:
47         # print(f'testing {rows} move {pos}')
48         new_row = swap(rows[-1], pos)
49         if test_rows(rows, new_row):
50             # print(f'{len(rows)} - extending {rows} with {new_row}.')
51             new_rows = rows.copy()
52             new_rows.append(new_row)
53             new_positions = positions.copy()
54             new_positions.append(pos)
55             all_possible_pos.copy()
56             new_avail_pos = all_possible_pos.copy()
57             new_avail_pos.remove(pos)
58             new_result, new_result_pos = extend_rows(new_rows, new_positions, new_avail_pos)
59             if new_result:
60                 result.extend(new_result)
61                 result_pos.extend(new_result_pos)
62
63     return result, result_pos
64
65 def convert(s):
66     str1 = ""
67     return(str1.join(s))
68
69 def Translate_pos_alphabet(pos_list: list[int]) -> list[str]:
70     alpha_pos_list: list[str] = []
71     alpha_pos_map = {0: 'D', 1: 'B', 2: 'C', 3: 'A'}
72     for pos in pos_list:
73         alpha_pos_list.append(alpha_pos_map[pos])
74     return alpha_pos_list
75
76
77 z1 = [1, 2, 3, 4]
78 z = [1, 2, 3, 4]
79 Len_int_ad = [0]
80 already_done = []
81 Transition = []
82
83 print("possible permutations:")
84 perm = permutations([1, 2, 3, 4])
85 perm_list_tuple = list(perm)
86 perm_list_list = []
87 for tup in perm_list_tuple:
88     perm_list_list.append(list(tup))
89 print("all permutations:", perm_list_list)
90 print("how many permutations?", len(perm_list_list))
91
92
93 # Main loop to build extents.
94 final_result: list[list[list[int]]] = []
95 final_result_pos: list[list[int]] = []
96 for pos in all_possible_pos:
97     #print(f'Starting with move {pos}')
98     new_row = swap(z, pos)
99     new_avail_pos = all_possible_pos.copy()
100    new_avail_pos.remove(pos)
101    new_result, new_result_pos = extend_rows([z, new_row], [pos], new_avail_pos)
102    if new_result:
103        final_result.extend(new_result)
104        final_result_pos.extend(new_result_pos)
105
106 print("A = (12)(34), B = (23), C = (34), D = (12)")
107 #for i in range(len(final_result)):
108 #    print(f'{Translate_pos_alphabet(final_result_pos[i])}')
109 #    print(f'{final_result[i]}')
110 print("number of valid extents:", len(final_result))
111
112 choice = random.randrange(len(final_result))

```

```

113
114 print("randomly chose:", choice)
115 print(f'{{convert(Translate_pos_alphabet(final_result_pos[choice]))}}')
116 print(f'{{final_result[choice]}}')
117 chosen = final_result[choice]
118
119 mixer.init()
120 soundfiles = ["Treble1.mp3", "Bell12.mp3", "Bell13.mp3", "Tenor4.mp3"]
121 sounds = []
122 for s in soundfiles:
123     sounds.append(mixer.Sound(s))
124
125 def output(bell_output):
126     for i, bell in enumerate(bell_output):
127         print(bell, end = " ")
128         mixer.Channel(i).play(sounds[bell - 1])
129         time.wait(1000)
130     print("\n", end = "")
131
132 for i in chosen:
133     output(i)
134 print("we done")
135 time.wait(1000)
136 mixer.quit()

```

### A.4.3 Output Printing All Extents

For demonstration purposes, the shell from the program without the audio, but all of the extents and their transitions:

```

possible permutations:
all permutations: [[1, 2, 3, 4], [1, 2, 4, 3], [1, 3, 2, 4], [1, 3, 4, 2], [1, 4, 2, 3], [1, 4, 3, 2],
[2, 1, 3, 4], [2, 1, 4, 3], [2, 3, 1, 4], [2, 3, 4, 1], [2, 4, 1, 3], [2, 4, 3, 1], [3, 1, 2, 4], [3, 1, 4, 2],
[3, 2, 1, 4], [3, 2, 4, 1], [3, 4, 1, 2], [3, 4, 2, 1], [4, 1, 2, 3], [4, 1, 3, 2], [4, 2, 1, 3], [4, 2, 3, 1],
[4, 3, 1, 2], [4, 3, 2, 1]]
how many permutations? 24
A = (12)(34), B = (23), C = (34), D = (12)
DABABABADABABABADABABABA
[[1, 2, 3, 4], [2, 1, 3, 4], [1, 2, 4, 3], [1, 4, 2, 3], [4, 1, 3, 2], [4, 3, 1, 2], [3, 4, 2, 1], [3, 2, 4, 1],
[2, 3, 1, 4], [3, 2, 1, 4], [2, 3, 4, 1], [2, 4, 3, 1], [4, 2, 1, 3], [4, 1, 2, 3], [1, 4, 3, 2], [1, 3, 4, 2],
[3, 1, 2, 4], [1, 3, 2, 4], [3, 1, 4, 2], [3, 4, 1, 2], [4, 3, 2, 1], [4, 2, 3, 1], [2, 4, 1, 3], [2, 1, 4, 3],
[1, 2, 3, 4]]
DABACABADABACABADABACABA
[[1, 2, 3, 4], [2, 1, 3, 4], [1, 2, 4, 3], [1, 4, 2, 3], [4, 1, 3, 2], [4, 1, 2, 3], [1, 4, 3, 2], [1, 3, 4, 2],
[3, 1, 2, 4], [1, 3, 2, 4], [3, 1, 4, 2], [3, 4, 1, 2], [4, 3, 2, 1], [4, 3, 1, 2], [3, 4, 2, 1], [3, 2, 4, 1],
[2, 3, 1, 4], [3, 2, 1, 4], [2, 3, 4, 1], [2, 4, 3, 1], [4, 2, 1, 3], [4, 2, 3, 1], [2, 4, 1, 3], [2, 1, 4, 3],
[1, 2, 3, 4]]
BADABABABADABABABADABABA
[[1, 2, 3, 4], [1, 3, 2, 4], [3, 1, 4, 2], [1, 3, 4, 2], [3, 1, 2, 4], [3, 2, 1, 4], [2, 3, 4, 1], [2, 4, 3, 1],
[4, 2, 1, 3], [4, 1, 2, 3], [1, 4, 3, 2], [4, 1, 3, 2], [1, 4, 2, 3], [1, 2, 4, 3], [2, 1, 3, 4], [2, 3, 1, 4],
[3, 2, 4, 1], [3, 4, 2, 1], [4, 3, 1, 2], [3, 4, 1, 2], [4, 3, 2, 1], [4, 2, 3, 1], [2, 4, 1, 3], [2, 1, 4, 3],
[1, 2, 3, 4]]
BADABACABADABACABADABACA
[[1, 2, 3, 4], [1, 3, 2, 4], [3, 1, 4, 2], [1, 3, 4, 2], [3, 1, 2, 4], [3, 2, 1, 4], [2, 3, 4, 1], [2, 3, 1, 4],
[3, 2, 4, 1], [3, 4, 2, 1], [4, 3, 1, 2], [3, 4, 1, 2], [4, 3, 2, 1], [4, 2, 3, 1], [2, 4, 1, 3], [2, 4, 3, 1],
[4, 2, 1, 3], [4, 1, 2, 3], [1, 4, 3, 2], [4, 1, 3, 2], [1, 4, 2, 3], [1, 2, 4, 3], [2, 1, 3, 4], [2, 1, 4, 3],
[1, 2, 3, 4]]
BABADABABABADABABABADABA
[[1, 2, 3, 4], [1, 3, 2, 4], [3, 1, 4, 2], [3, 4, 1, 2], [4, 3, 2, 1], [3, 4, 2, 1], [4, 3, 1, 2], [4, 1, 3, 2],
[1, 4, 2, 3], [1, 2, 4, 3], [2, 1, 3, 4], [2, 3, 1, 4], [3, 2, 4, 1], [2, 3, 4, 1], [3, 2, 1, 4], [3, 1, 2, 4],
[1, 3, 4, 2], [1, 4, 3, 2], [4, 1, 2, 3], [4, 2, 1, 3], [2, 4, 3, 1], [4, 2, 3, 1], [2, 4, 1, 3], [2, 1, 4, 3],
[1, 2, 3, 4]]
BABABADABABABADABABABADA
[[1, 2, 3, 4], [1, 3, 2, 4], [3, 1, 4, 2], [3, 4, 1, 2], [4, 3, 2, 1], [4, 2, 3, 1], [2, 4, 1, 3], [4, 2, 1, 3],
[2, 4, 3, 1], [2, 3, 4, 1], [3, 2, 1, 4], [3, 1, 2, 4], [1, 3, 4, 2], [1, 4, 3, 2], [4, 1, 2, 3], [1, 4, 2, 3],
[4, 1, 3, 2], [4, 3, 1, 2], [3, 4, 2, 1], [3, 2, 4, 1], [2, 3, 1, 4], [2, 1, 3, 4], [1, 2, 4, 3], [2, 1, 4, 3],
[1, 2, 3, 4]]
BABABACABABABACABABABACA
[[1, 2, 3, 4], [1, 3, 2, 4], [3, 1, 4, 2], [3, 4, 1, 2], [4, 3, 2, 1], [4, 2, 3, 1], [2, 4, 1, 3], [2, 4, 3, 1],
[4, 2, 1, 3], [4, 1, 2, 3], [1, 4, 3, 2], [1, 3, 4, 2], [3, 1, 2, 4], [3, 2, 1, 4], [2, 3, 4, 1], [2, 3, 1, 4],
[3, 2, 4, 1], [3, 4, 2, 1], [4, 3, 1, 2], [4, 1, 3, 2], [1, 4, 2, 3], [1, 2, 4, 3], [2, 1, 3, 4], [2, 1, 4, 3],
[1, 2, 3, 4]]
BABACABABABACABABABACABA

```

[illegible]

```
[1, 2, 3, 4]]
ACABADABACABADABACABADAB
[[1, 2, 3, 4], [2, 1, 4, 3], [2, 1, 3, 4], [1, 2, 4, 3], [1, 4, 2, 3], [4, 1, 3, 2], [1, 4, 3, 2], [4, 1, 2, 3],
[4, 2, 1, 3], [2, 4, 3, 1], [2, 4, 1, 3], [4, 2, 3, 1], [4, 3, 2, 1], [3, 4, 1, 2], [4, 3, 1, 2], [3, 4, 2, 1],
[3, 2, 4, 1], [2, 3, 1, 4], [2, 3, 4, 1], [3, 2, 1, 4], [3, 1, 2, 4], [1, 3, 4, 2], [3, 1, 4, 2], [1, 3, 2, 4],
[1, 2, 3, 4]]
ACABABABACABABABACABABAB
[[1, 2, 3, 4], [2, 1, 4, 3], [2, 1, 3, 4], [1, 2, 4, 3], [1, 4, 2, 3], [4, 1, 3, 2], [4, 3, 1, 2], [3, 4, 2, 1],
[3, 2, 4, 1], [2, 3, 1, 4], [2, 3, 4, 1], [3, 2, 1, 4], [3, 1, 2, 4], [1, 3, 4, 2], [1, 4, 3, 2], [4, 1, 2, 3],
[4, 2, 1, 3], [2, 4, 3, 1], [2, 4, 1, 3], [4, 2, 3, 1], [4, 3, 2, 1], [3, 4, 1, 2], [3, 1, 4, 2], [1, 3, 2, 4],
[1, 2, 3, 4]]
number of valid extents: 24
```