

# DISCRETE MATHEMATICS

## for Computer Science

Alexander Golovnev, Alexander S. Kulikov, Vladimir V. Podolskii, Alexander Shen

## Welcome!

Thank you for downloading this book! It supplements the [Introduction to Discrete Mathematics for Computer Science](#) specialization at Coursera and contains many interactive puzzles, autograded quizzes, and code snippets. They are intended to help you to discover important ideas in discrete mathematics on your own, and to show you corresponding applications of these ideas in computer science.

This book contains material corresponding to the first two courses in the associated specialization at Coursera: [Mathematical Thinking in Computer Science](#) and [Combinatorics and Probability](#). Future editions will cover the additional three courses: Graph Theory, Number Theory and Cryptography, and Delivery Problem.

There are 300+ problems and 150+ code snippets in the book. Most of the problems come with solutions and 130+ of them are graded automatically (allowing you to get instant feedback). We're constantly working on extending and improving this book. Please ask questions, report typos, and suggest improvements through this [form](#). Check <https://leanpub.com/discrete-math> for updates of the book.

# Contents

|          |                             |          |
|----------|-----------------------------|----------|
| <b>0</b> | <b>About the Book</b>       | <b>7</b> |
| 0.1      | Active Learning             | 7        |
| 0.2      | Problem-based Learning      | 7        |
| 0.3      | Python Programming Language | 8        |
| 0.4      | Acknowledgments             | 9        |

## I

## Mathematical Thinking in Computer Science

|          |                                     |           |
|----------|-------------------------------------|-----------|
| <b>1</b> | <b>Proofs: Convincing Arguments</b> | <b>13</b> |
| 1.1      | Warm Up                             | 13        |
| 1.2      | Existence Proofs                    | 20        |
| <b>2</b> | <b>Finding an Example</b>           | <b>31</b> |
| 2.1      | How to Find an Example              | 31        |
| 2.2      | Optimality                          | 39        |
| 2.3      | Computer Search                     | 49        |
| <b>3</b> | <b>Recursion and Induction</b>      | <b>57</b> |
| 3.1      | Recursion                           | 57        |
| 3.2      | Induction                           | 75        |
| <b>4</b> | <b>Logic</b>                        | <b>93</b> |
| 4.1      | Examples and Counterexamples        | 93        |
| 4.2      | Logic                               | 97        |
| 4.3      | Reductio ad Absurdum                | 103       |

|          |                                 |            |
|----------|---------------------------------|------------|
| <b>5</b> | <b>Invariants .....</b>         | <b>109</b> |
| 5.1      | Double Counting                 | 109        |
| 5.2      | Searching for Invariants        | 111        |
| 5.3      | Termination                     | 112        |
| 5.4      | Even and Odd Numbers            | 115        |
| <b>6</b> | <b>Project: 15-Puzzle .....</b> | <b>121</b> |
| 6.1      | The Puzzle                      | 121        |
| 6.2      | Permutations and Transpositions | 122        |
| 6.3      | Why 15-puzzle Has No Solution   | 133        |
| 6.4      | When 15-puzzle Has a Solution   | 135        |
| 6.5      | Implementation                  | 142        |
| <b>7</b> | <b>Appendix .....</b>           | <b>149</b> |
| 7.1      | Cutting a Figure                | 149        |
| 7.2      | Using SAT-solvers               | 149        |
| 7.3      | Using ILP-solvers               | 153        |
| 7.4      | Visualizing Football Fans       | 156        |

## II

## Combinatorics and Probability

|           |                                    |            |
|-----------|------------------------------------|------------|
| <b>8</b>  | <b>Basic Counting .....</b>        | <b>161</b> |
| 8.1       | Starting to Count                  | 161        |
| 8.2       | Recursive Counting                 | 167        |
| 8.3       | Tuples and Permutations            | 173        |
| <b>9</b>  | <b>Binomial Coefficients .....</b> | <b>181</b> |
| 9.1       | Number of Games in a Tournament    | 181        |
| 9.2       | Combinations                       | 184        |
| 9.3       | Binomial Theorem                   | 190        |
| 9.4       | Practice Counting                  | 193        |
| <b>10</b> | <b>Advanced Counting .....</b>     | <b>197</b> |
| 10.1      | Review                             | 197        |
| 10.2      | Combinations with Repetitions      | 198        |
| 10.3      | Practice Counting                  | 202        |
| <b>11</b> | <b>Probability .....</b>           | <b>211</b> |
| 11.1      | What is Probability?               | 211        |
| 11.2      | Probability: Do's and Don'ts       | 227        |
| 11.3      | Conditional Probability            | 241        |
| 11.4      | Monty Hall Paradox                 | 253        |

|           |   |            |
|-----------|---|------------|
| <b>12</b> | <b>Random Variables .....</b>           | <b>259</b> |
| 12.1      | Random Variables and Their Expectations | 259        |
| 12.2      | Linearity of Expectation                | 268        |
| 12.3      | Expectation is Not All                  | 271        |
| 12.4      | Markov's Inequality                     | 273        |
| <b>13</b> | <b>Dice Games .....</b>                 | <b>275</b> |
| 13.1      | Dice Game Problem                       | 275        |
| 13.2      | Optimal Strategy                        | 278        |



# 0. About the Book

## 0.1 Active Learning

This book covers ideas and concepts in discrete mathematics which are needed in various branches of computer science. To make the learning process more efficient and enjoyable, we use the following *active learning components* implemented through our [Introduction to Discrete Mathematics for Computer Science specialization](#) at Coursera.

**Interactive puzzles** provide you with a fun way to “invent” the key ideas on your own. The puzzles are mobile-friendly, so you can play with them anywhere. The goal of every puzzle is to give you a clean and easy way to state problems where nothing distracts you from inventing a method for solving it. In turn, the corresponding method usually has a wide range of applications to various problems in computer science.

**Autograded quizzes** allow you to immediately check your understanding after learning a new concept or idea.

**Code snippets** are helpful in two ways: 1) they show you how ideas from discrete mathematics are used in programming, and 2) they serve as interactive examples and challenges: tweak the given piece of code, run it, and see what happens.

**Programming challenges** will help you to solidify your understanding. As Donald Knuth said, “I find that I don’t understand things unless I try to program them.”

## 0.2 Problem-based Learning

Throughout the book (and the associated specialization at Coursera) we follow a “try this before we explain everything” approach: we always ask you to solve a problem first, and then we explain how to solve it and introduce important ideas needed to solve it. We believe, this way you will get a deeper understanding and also develop a better appreciation for the beauty of the underlying ideas (not to mention the self-confidence that you get if you invent these ideas on your own!). Don’t be discouraged if you can’t solve all the problems. Just having attempted them is often enough to engage your mind, and make you more curious about the solution.

We use the following two basic types of questions in the book.

**Stop and think** questions invite you to slow down and contemplate the current material before continuing to the next topic. We *always* provide an answer to the corresponding question right after it. We strongly encourage you (as the name suggests) to stop and do your best to answer the question.

**Problems** usually require more effort to solve. We use some of them to warm you up and to develop your curiosity. Such problems are followed by detailed solutions. Some other problems are left for you as exercises.

Many questions in the book are *graded automatically* through Coursera. They are marked with:

Try it: [Coursera ↗](#), [external ↗](#).

Both these links are clickable: the first one opens the corresponding autograded puzzle at Coursera (this requires an active subscription to the specialization), the second one opens the corresponding interactive puzzle (and requires no subscription). At the same time, the book is self-contained: if you are unable to watch the videos and access the interactive puzzles at Coursera, just read the book and solve the problems on a piece of paper.

## 0.3 Python Programming Language

### 0.3.1 Why Programming?

*Why on earth do we start the book with discussing a programming language? After all, this is a math (rather than programming) book!*

That's true. But we believe that many pieces of code shown in this book will help you in many ways:

- They will show you a rich variety of applications of discrete math ideas in various branches of computer science.
- Code snippets can serve as interactive examples: you may want to tweak the given piece of code, run it, and see what happens.
- By trying to implement a particular idea, you are forced to understand every single detail of it.
- It is often easier to reason in terms of specific objects in programming rather than abstract mathematical concepts.

We have set up everything in a way that will allow you to run the code snippets used in this book even if you have never tried to write a program before. You don't even need to install or set up anything: everything can be run in the cloud, through your Internet browser. At the same time, we also provide instructions for those who would like to learn the basics of Python while learning discrete math.

### 0.3.2 Why Python?

*OK, let's do some programming while learning discrete math. But why Python instead of any other popular programming language?*

Let us convince you that Python is an excellent choice for our purposes.

**High-level language.** It is particularly easy to start using Python (even if you haven't programmed before). The syntax is reader friendly (and close to a natural language). The code is compact: most of the pieces of code in this book are less than ten lines long!

**Interactive mode.** It can be used in an interactive mode (also known as [REPL ↗](#), for read-eval-print loop). This allows you to talk to your computer using Python as a language: the computer then *reads* your input, *evaluates* it, and *prints* the result. This way, you work stuff out and get instant feedback from the machine.

**“Batteries included”.** The Python [standard library ↗](#) offers a wide range of facilities, and many external libraries are available as well. In particular, this will allow us to generate a random sequence, plot a function, and draw a graph in just one line of code!

This (partly) explains why Python is often used for software prototyping, and in such areas as machine learning, data science, and web development.

Of course, advantages always come at the cost of some disadvantages. The high-levelness of Python makes it less flexible in performance tuning. This is OK for us, as we will only be using simple snippets of code where optimizing is not an issue.

### 0.3.3 How to Catch Up with Python?

*OK, let's try! Where do I start?*

**Locally.** To install Python on your machine, go to the [Get Started](#) section of [python.org](#) and follow the instructions. If you are new to Python, we encourage you to install [PyCharm](#) to start working with Python: this (free of charge) professional IDE will make the process of writing and running your code smoother and more efficient.

**In the cloud.** Alternatively, you may run all our code snippets from your Internet browser, without installing or configuring anything on your machine. To do this, visit the [repository page](#) and click the badge “Open in Colab”. This will show you a list of notebooks that can be run in an interactive mode right in your browser (together with links to a tutorial on notebooks).

## 0.4 Acknowledgments

This book was greatly improved by the efforts of a large number of individuals whom we owe a debt of gratitude.

We thank the students of the Coursera specialization as well as the students of the Modern Software Engineering B.Sc. program at St. Petersburg State University for their continuous and valuable feedback. We also thank Jerry Allen, Huck Bennett, Marie Brodsky, Anuj Kumar Karmakar, and Terence Minerbrook for carefully reading an earlier draft of this book.

We are grateful to Anton Konev and Daria Borisyak for leading the development of interactive puzzles. We thank Vitaliy Polshkov for reviewing our Python code.



# Mathematical Thinking in Computer Science

|          |   |            |
|----------|---|------------|
| <b>1</b> | <b>Proofs: Convincing Arguments . . . . .</b> | <b>13</b>  |
| 1.1      | Warm Up                                       |            |
| 1.2      | Existence Proofs                              |            |
| <b>2</b> | <b>Finding an Example . . . . .</b>           | <b>31</b>  |
| 2.1      | How to Find an Example                        |            |
| 2.2      | Optimality                                    |            |
| 2.3      | Computer Search                               |            |
| <b>3</b> | <b>Recursion and Induction . . . . .</b>      | <b>57</b>  |
| 3.1      | Recursion                                     |            |
| 3.2      | Induction                                     |            |
| <b>4</b> | <b>Logic . . . . .</b>                        | <b>93</b>  |
| 4.1      | Examples and Counterexamples                  |            |
| 4.2      | Logic   |            |
| 4.3      | Reductio ad Absurdum                          |            |
| <b>5</b> | <b>Invariants . . . . .</b>                   | <b>109</b> |
| 5.1      | Double Counting                               |            |
| 5.2      | Searching for Invariants                      |            |
| 5.3      | Termination                                   |            |
| 5.4      | Even and Odd Numbers                          |            |
| <b>6</b> | <b>Project: 15-Puzzle . . . . .</b>           | <b>121</b> |
| 6.1      | The Puzzle                                    |            |
| 6.2      | Permutations and Transpositions               |            |
| 6.3      | Why 15-puzzle Has No Solution                 |            |
| 6.4      | When 15-puzzle Has a Solution                 |            |
| 6.5      | Implementation                                |            |
| <b>7</b> | <b>Appendix . . . . .</b>                     | <b>149</b> |
| 7.1      | Cutting a Figure                              |            |
| 7.2      | Using SAT-solvers                             |            |
| 7.3      | Using ILP-solvers                             |            |
| 7.4      | Visualizing Football Fans                     |            |



# 1. Proofs: Convincing Arguments

Why are some arguments convincing while others are not? What makes an argument convincing? How can you establish your argument in such a way that no room for doubt is left? How can mathematical thinking help us deal with this? In this chapter, we will start by digging into these questions. Our goal here is to learn by examples how to understand proofs, how to discover them on your own, how to explain them, and — last but not least — how to enjoy them: we will see how a small remark or a simple observation can turn a seemingly non-trivial question into one with an obvious answer.

## 1.1 Warm Up

### 1.1.1 Why Proofs?

Proofs are absolutely necessary in mathematics, computer science, programming, and many other areas. Once you have an algorithm for a problem, you need to prove that it is correct. “The program works for me, what else do I need?” is not an approach that would scale well. A library function may run billions of times while being used by thousands of diverse programs. If it returns a single wrong result, say, once in every million calls, it could be disastrous. Think of an air traffic controller. If in their entire career they receive information with just one wrong value, just one time, hundreds of people could perish.<sup>1</sup> This is why mathematical proofs must be rigorously demonstrated to provide correct conclusions.

That is why throughout the whole book we will be focusing on formal proofs. Here, “formal” does not mean “long” or “unclear”! We will encounter many short and elegant proofs that are formal and convincing. Using our carefully designed interactive puzzles, we will try to push you softly to discover some of the proofs on your own. Nothing compares to the sense of happiness and self-satisfaction of an “Aha!” moment when you find a solution to a mathematical problem!

### 1.1.2 Tiling a Chessboard

**Problem 1** Can a chessboard be tiled by domino tiles? Here, a chessboard is an  $8 \times 8$  square divided into 64 squares  $1 \times 1$  (see Figure 1.1), a domino tile is a  $1 \times 2$  (or  $2 \times 1$ ) rectangle, and

<sup>1</sup>For some specific examples, Google for “Bugs in the Space Program”, “Therac-25”, and “Toyota unintended acceleration”.

by saying “tiled” we mean that there are no overlaps or empty spaces. Try it (question 1): [Coursera](#), [external](#).

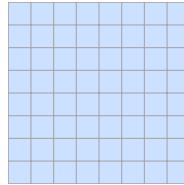


Figure 1.1: An  $8 \times 8$  chessboard. Can it be tiled with domino tiles?

Yes, there are many such tilings. Two of them are shown in Figure 1.2.

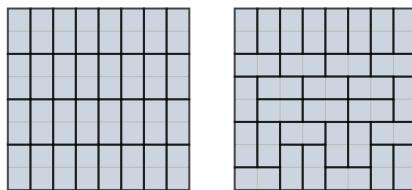


Figure 1.2: Two examples of tiling a chessboard.

**Stop and Think!** Do we need both these examples to solve Problem 1, or one is enough?

In fact, one example is enough: any such example shows that it is possible to tile a chessboard.

**Problem 2** Now consider the chessboard without one of the corners, see Figure 1.3. Can we tile it with domino tiles? Try it (question 2): [Coursera](#), [external](#).

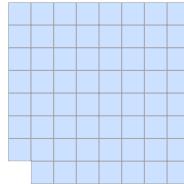


Figure 1.3: A chessboard without a corner. Can it be tiled with domino tiles?

Let's try. For example, let us use horizontal tiles, starting from the top row. Everything goes OK until we come to the bottom row, see Figure 1.4. This last row is problematic. We can put three tiles, but one cell is left uncovered.

**Stop and Think!** Can we say now that Problem 2 is solved and we proved that the required tiling does not exist?

No, we cannot: one attempt to tile the board was unsuccessful. But this *does not* mean that the task is impossible. We are not limited to using horizontal tiles only; we can try doing something more sophisticated. Let us try a spiral, see Figure 1.5.

**Stop and Think!** Can we say now that Problem 2 is solved and we have proven that the required tiling does not exist?

Of course not: we tried twice, but there are many more ways to try. What if some of them are successful? For a smaller board we may try all possibilities. Say we invent a systematic way to

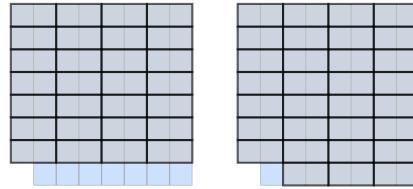


Figure 1.4: An unsuccessful attempt to tile a chessboard without a corner.

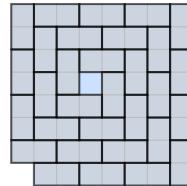


Figure 1.5: An unsuccessful attempt at a spiral tiling of a chessboard without a corner.

enumerate them, making sure that no possible tiling has been missed. However, here the space of possibilities is rather large.

**Stop and Think!** Can you find a tiling or some general reason why we will always fail?

More specifically:

**Stop and Think!** Imagine there is a tiling of an  $8 \times 8$  chessboard without a corner, by  $1 \times 2$  tiles. How many tiles are in this tiling?

The full board contains  $8 \times 8 = 64$  cells, so without a corner we have  $64 - 1 = 63$  cells. Each domino tile consists of two cells, so the answer is  $63/2 = 31.5$  tiles.

**Stop and Think!** The answer 31.5 is absurd: the number of tiles should be an integer. How is this even possible?

Recall the assumption we started with: “Imagine there is a tiling...”. If there was a tiling of 63-cell board with domino tiles, it would use  $63/2 = 31.5$  tiles. This, of course, is impossible — therefore, such a tiling does not exist. Thus, we get the proof we were looking for.

**Problem 3** Consider an  $8 \times 8$  chessboard without two adjacent corners, see Figure 1.6. Can it be tiled by domino tiles? Try it (question 3): [Coursera](#).

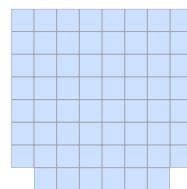


Figure 1.6: A chessboard without two adjacent corners. Can you tile it with domino tiles?

We already know that one should find the number of tiles needed: we have  $8 \times 8 - 2 = 62$  cells, so we need  $62/2 = 31$  tiles. We got an integer number, so we do not run into a problem and the tiling exists.

**Stop and Think!** Do you agree with this reasoning?

If you do, you are too fast. The argument shows only that *if* a tiling existed, it *would* consist of 31 tiles. But it does not show that a tiling exists. Informally speaking, we see only that some specific obstacle (non-integer number of tiles) does not prevent the existence of a tiling, but there may be other obstacles.

**Stop and Think!** Give a correct proof of the existence of a tiling for the board without two adjacent corners.

Here it is, see Figure 1.7.

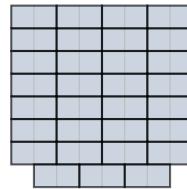


Figure 1.7: A tiling of a chessboard without two corners.

After training with these simple examples, we can tackle a more difficult question.

**Problem 4** Consider an  $8 \times 8$  chessboard without two opposite corners, see Figure 1.8. Can it be tiled by domino tiles? Try it (question 4): [Coursera ↗](#).

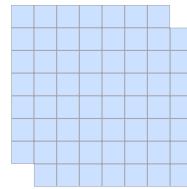


Figure 1.8: A chessboard without two opposite corners.

Again, the board contains  $8 \times 8 - 2 = 62$  cells, so  $62/2 = 31$  tiles would be needed. This is an integer number. But we already know that it does *not* mean that a tiling exists. Mathematicians would say that the even number of cells is a *necessary* condition for the existence of a tiling, but we do not know whether it is a *sufficient* condition.

**Stop and Think!** Can you complete the existence proof by constructing some tiling?

Let's try. One such attempt is shown in Figure 1.9. As you see, this attempt was not successful: two cells are not covered.

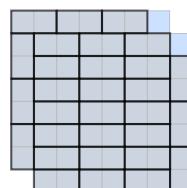


Figure 1.9: An unsuccessful attempt to tile a chessboard without two opposite corners.

The situation does not look hopeless: two cells are not covered, and the only problem is that they are not neighbors, so we cannot cover both by one tile. But maybe one can move some tiles to

make the non-covered cells neighboring? Or maybe we can just start anew and have better luck? Let us try, see Figure 1.10. Now the empty cells are in the same column, but they are not neighbors.

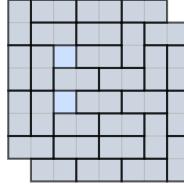


Figure 1.10: Another unsuccessful attempt to tile a board without two opposite corners.

If you play a bit more with this [puzzle ↗](#) (level 3), you will see that this is more than just bad luck: every time at least two uncovered cells remain. But why?

**Stop and Think!** How can we prove that such a tiling is not possible?

Here a new tool is needed. If you are a chess player or have seen a real chessboard, you may have noticed that our drawings ignore one important feature of a chessboard. It has cells of two colors, usually black and white. In our color scheme, we will distinguish light and dark cells, see Figure 1.11.

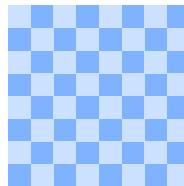


Figure 1.11: Chessboard coloring.

**Stop and Think!** How many dark and how many light cells are there on the chessboard?

Each row contains 4 light and 4 dark cells, so in total we have  $8 \times 4 = 32$  light and 32 dark cells.

Thus, we have the same number of light and dark cells. Could we see this without counting them? One could show that the number of chairs in a room is equal to the number of people in it by asking everyone to sit down: if each person is seated and no chairs are empty, these two numbers are equal. (Unless somebody sits on two chairs or some chair is shared.)

**Stop and Think!** Can you prove in a similar way that the number of dark cells is the same as the number of light cells, by pairing them into light-dark pairs?

Any tiling of a chessboard will work: looking at the chessboard, we see that neighboring cells are always of different colors, hence every tile is a dark-light pair (Figure 1.12).

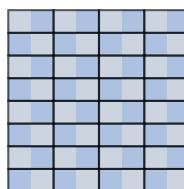


Figure 1.12: Pairing proves that the number of light cells is the same as the number of dark cells.

After this digression let us return to our board without opposite corners (Figure 1.13).

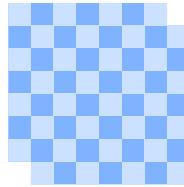


Figure 1.13: Looking again at the board without opposite corners.

**Stop and Think!** How many dark and how many light cells are on this board?

We do not need to count them again: two corner cells that are deleted are both dark. Thus, we have 30 dark cells and 32 light cells.

**Stop and Think!** Do you see why this board cannot be tiled by dominos?

Imagine that a tiling existed. It would use  $62/2 = 31$  tiles. Each tile covers one dark cell and one light cell. So in total all tiles would cover 31 dark cells and 31 light cells. And — *aha!* — our board has 30 dark and 32 light cells. This mismatch shows that tiling is impossible.

Let us repeat the argument in a slightly different way. If a board can be tiled, then the numbers of dark and light cells are the same (=the number of tiles), because each tile covers one dark and one light cell. Hence, if these two numbers (dark and light cells) are different, no tiling is possible.

In a more concise exposition, this argument could be compressed into one paragraph.

**Theorem 1.1.1** An  $8 \times 8$  chessboard without two opposite corners cannot be tiled by  $1 \times 2$  dominoes.

*Proof.* Consider the standard coloring of the chessboard with two colors (dark and light) where neighboring cells have opposite colors. It is easy to see that

- each tile contains one dark cell and one light cell;
- the board has 32 cells of one color and 30 cells of the other color (two deleted corners have the same color).

The first observation implies that any tileable region has the same number of dark and light cells, and then the second observation shows that our board is not tileable. ■

**Stop and Think!** We have seen a partial tiling of the board without two opposite corners where two cells remain uncovered. What are the colors of these cells? (Try to give an answer without looking at the picture of the tiling.)

We do not need to know the specific tiling: if we have 32 light and 30 dark cells, and the tiling pairs every light and dark cell with two remaining unpaired, they must be light colored cells.

We have seen that a chessboard with one deleted cell is not tileable for trivial reasons (the number of cells is not even). For two deleted cells we had two examples. The first one, without two neighboring corners, was tileable; the other one, without opposite corners, was not.

**Stop and Think!** Why can't the argument that we use to prove the non-tileability in the second case be applied to the first case? What is the difference?

Let us formulate this question in a more specific way. Consider a chessboard without two cells. We know that sometimes the rest is tileable (example: two adjacent corners) and sometimes it is not (example: two opposite corners). Try it (questions 5 and 6): [Coursera](#) ↗.

**Stop and Think!** Can you state a general rule that distinguishes between tileable and untileable boards without two cells?

The following problem provides an answer to this question.

**Problem 5** Prove that a chessboard without two cells can be tiled by domino tiles if and only if the deleted cells are of opposite colors.

The statement includes a strange expression “if and only if” (also known as “iff”). This mathematical jargon means that we have to prove two things:

- if we delete two cells of the opposite colors, then the rest is tileable (“if” part);
- if the board without two cells is tileable, then the deleted cells are of opposite colors (“only if” part).

**Stop and Think!** We have shown that any tileable region has the same number of dark and light cells, so the board without two cells of the same color is not tileable. What did we prove: the “if” part or the “only if” part?

It remains to prove that the board without one dark and one light cell is always tileable. To keep the suspense, we will not give the solution of this problem. Here is a diagram to think about, Figure 1.14. If you are old enough (some of the authors are), you may remember the computer game where a growing snake moves along itself eating food items placed in certain cells and increasing the length. In terms of this game, this picture shows a snake that has reached maximal possible length (includes all cells) so its head can be glued to its tail.

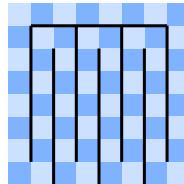


Figure 1.14: This “circular snake” helps us to prove that the board without two cells of opposite colors is tileable. Do you see how? For starters, tile the entire board by cutting the snake into domino tiles. What happens if you delete two cells from the snake?

**Problem 6** We want to cover the figure shown in Figure 1.15 by  $1 \times 2$  domino tiles. Is it possible (a) if we cover the highlighted cell by a horizontal tile? (b) if we cover the highlighted cell by a vertical tile? Try it (questions 1 and 2): [Coursera](#).

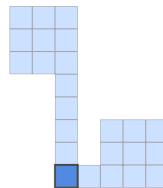


Figure 1.15: Board for Problem 6.

Now consider a  $5 \times 5$  board divided into 25 cells  $1 \times 1$  (Figure 1.16). It is not possible to tile it

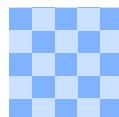


Figure 1.16: A  $5 \times 5$  board (Problem 7).

by  $1 \times 2$  tiles. (Why? Since the number of tiles needed for that, i.e.,  $25/2 = 12.5$ , is not an integer.)

However, if we delete one cell, there may be a chance to cover the remaining 24 cells by 12 tiles. For example, if you delete the left upper corner, you can tile the rest using vertical tiles in the first column and horizontal tiles elsewhere. Let us call a cell *good* if the rest ( $5 \times 5$  board without this cell) can be tiled by dominoes.

**Problem 7** Which of the cells are good? How many good cells are there? Try it (question 3): [Coursera](#).

**Problem 8** Can we tile an  $8 \times 8$  board by  $1 \times 3$  tiles? (They can be placed both horizontally and vertically.) Can we tile  $8 \times 8$  board without one corner by  $1 \times 3$  tiles?

**Problem 9** Can we tile a  $10 \times 10$  board by  $2 \times 2$  tiles? Can we tile this board by  $1 \times 4$  tiles (that can be placed horizontally or vertically)?

*Dark and light cells strike back.* We have a full characterization of tileability of a chessboard without two cells: if they are of the same color, then there is no tiling; otherwise, the tiling exists. But what if the board is missing more than two cells? Specifically, do you see a way to implement a program that is given a subset of the cells of the board (i.e., for each cell it is indicated whether it is present or not) and quickly checks whether this region is tileable? We'll learn how to do this later in the book, when we study matchings in graphs! Interestingly, the solution will be based on dark and light cells again. (Technically, one should look for a maximal matching between dark and light cells in the bipartite neighborhood graph; we will explain what all these words mean.)

## 1.2 Existence Proofs

In this section, we study *existence proofs*, i.e., proofs of *existential statements*. For example,

There exists an object which satisfies a particular property.

To prove this statement, we can provide an example of such an object: this is called a *constructive proof*.

There is another way to prove an existential statement: we can prove that such an object exists without providing an example of the object! Sounds counterintuitive, doesn't it? These are known as *non-constructive proofs* and we will see them in Section 1.2.4. For now, we will work with constructive proofs.

### 1.2.1 One Example Is Enough

We have seen constructive proofs of existential statements in the previous section.

**Stop and Think!** In the previous section we proved statements of two types: (a) some region can be tiled by dominos, and (b) some region cannot be tiled. Which of them are existential statements: (a) or (b)?

The existence of a tiling is (by definition) an existential statement. We proved it by showing an example of the required tiling. As we have said, one example is enough. This already constitutes a formal proof of existence. One does not need to explain how this example has been found, though this may be an interesting and non-trivial question (discussed later in Section 2.1).

**Problem 10** Is it possible to cut down the figure shown in Figure 1.17 into two congruent pieces, i.e., into two pieces of the same shape and size? (In other words, does there *exist* a way to cut the figure into two congruent pieces?)

For this problem, a solution is not difficult to find, see Figure 1.18.

**Stop and Think!** What if we want to cut down the same figure into three congruent pieces?

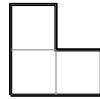


Figure 1.17: Can you cut this figure into two congruent pieces?

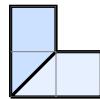


Figure 1.18: A solution to Problem 10.

This is even easier, since the figure consists of three squares. To make things more challenging, what if we pose the same question, but with four pieces?

**Problem 11** Prove that the same figure can be cut down into four congruent pieces. (Equivalently, prove that there *exists* a way to cut down the figure into four congruent pieces.)

The hint is that the small pieces could be of the same shape as the figure itself. The construction is given in Figure 1.19. Note that just this example is enough to solve the problem. It provides a complete proof, and we do not need to add anything else.

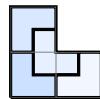


Figure 1.19: A solution to Problem 11.

Let's consider a somewhat more advanced problem of a similar flavor.

**Problem 12** Prove that the octagon shown in Figure 1.20 can be cut down into two congruent pieces (an octagon is a polygon with eight sides). Try it (question 2): [Coursera](#).

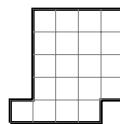


Figure 1.20: Can you cut this octagon into two congruent pieces?

This problem might look difficult at first, but there is again a simple solution. We can actually cut this figure along grid lines, see Figure 1.21. (To see that two pieces are the same, just move the left piece two cells to the right.)

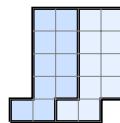


Figure 1.21: A solution to Problem 12.

How curious! There is another solution to this puzzle, see Figure 1.22.

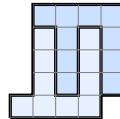


Figure 1.22: Another solution to Problem 12.

**Problem 13** Prove that this figure can be cut down into *three* congruent pieces.

In this problem we do *not* require the cuts to go along the grid lines. This allows us to solve the problem by extending the previous solution a bit, see Figure 1.23.

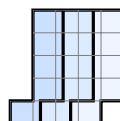


Figure 1.23: A solution to Problem 13.

If we required the cuts in this solution to go along the grid lines, our task would be impossible.

**Stop and Think!** Do you see why?

The reason is that the figure consists of 20 cells, and 20 is not divisible by 3.

We conclude with a more challenging problem of the same type.

**Problem 14** Prove that it is possible to cut the figure from Figure 1.24 into two congruent pieces.

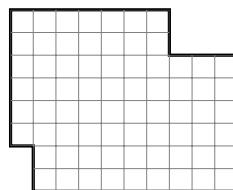


Figure 1.24: Can you cut this figure into two congruent pieces?

Finding a solution to this problem is difficult, so don't spend too much time on it. Knowing that a solution is possible and not knowing it is intolerable. For this reason, we provide a solution in Section 7.1.

Let us illustrate the “one example is enough” principle physically rather than mathematically. Imagine that you have three sticks and a length of string. Tie the string around the three sticks so that they form a free-standing structure in which the sticks do not touch. Could you make a free-standing structure with these restrictions? Put it another way: Does such a structure *exist*? It may surprise you but the answer is yes! Again, to prove this existential statement we only have to provide a single working example. See the [video in our course](#) or this [wikipedia page](#).

### 1.2.2 Existential Statements in Number Theory

Recall that an integer  $a$  is *divisible* by a positive integer  $b$  if  $k = a/b$  is an integer. In other terms,  $a$  is divisible by  $b$  if there *exists* an integer  $k$  such that  $a = kb$ .

In Python, to check whether  $a$  is divisible by  $b$ , one checks whether the *remainder* of  $a$  when divided by  $b$  is equal to zero. The remainder is found using the *modulo operator* — `%`. The following snippet shows that 237 is divisible by 3 and is not divisible by 7.

```
print(237 % 3)
print(237 % 7)
```

```
0
6
```

**Stop and Think!** Is 123 123 123 divisible by 123?

Yes, it is:  $123\ 123\ 123 = 123 \cdot 1\ 001\ 001$ , so we may take  $k = 1\ 001\ 001$  in the definition above. The ratio  $k$  can be easily found by a computer program (or just a calculator). You may also get the answer without any tools (even without pencil or paper) if you think about the long division procedure. (Be careful: a common error is to get  $k = 111$ .)

**Problem 15** Is there a positive integer that is divisible by 13 and ends with 15?

To prove that such a number exists, it is enough to give a single example. One such example is 715: it ends with 15 and it is divisible by 13 ( $715 = 13 \cdot 55$ ). This already proves the existence, and we don't even need to explain how we have found this integer. Still, the following three lines of code help to find all such integers in the range  $[0, 9999]$ .

```
for n in range(10 ** 4):
    if n % 13 == 0 and n % 100 == 15:
        print(n)
```

```
715
2015
3315
4615
5915
7215
8515
9815
```

This program checks all numbers in `range(10 ** 4)`. Here,  $10^{** 4}$  stands for  $10^4 = 10000$ . In Python, `range(N)` where  $N$  is some non-negative number is a *list*<sup>2</sup> (sequence) of  $N$  numbers  $0, 1, 2, \dots, N - 1$ . The `for`-loop goes over all of them in this order; the `if` operator checks whether they have the required properties. The last two digits of an integer  $n$  can be computed as  $n \% 100$ . In general,  $n \% m$  denotes the *remainder* when dividing  $n$  by  $m$ .<sup>3</sup>

**Problem 16** Is there an integer that is divisible by 15 and ends with 13?

In this case, a similar program will produce no output. This doesn't indicate that there is no such integer since we only checked the positive integers below  $10^4$  in the program. But this is indeed the case: such an integer must be divisible by 5 (since it is divisible by 15), but all integers divisible by 5 end with either 0 or 5.

**Problem 17** Find a two-digit (positive) integer that becomes 7 times smaller when its first (=leftmost) digit is removed.

Let's try. Consider all two-digit integers that are divisible by 7:

14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84, 91, 98.

We know that dividing the required integer by 7 should result in a single digit integer. This allows us to rule out all numbers starting from 70 from the list. We can then check manually that out of the remaining numbers the only one satisfying the required property is 35.

<sup>2</sup>Technically speaking, in version 3 of Python `range(N)` is no longer a list, but a so-called *generator*, but for us the difference is not so important.

<sup>3</sup>Imagine we have  $n$  identical books on the table and pack them into boxes that contain  $m$  books each. Then  $n \% m$  books remain unpacked.

The argument above is simple, but still some reasoning is needed. One could use a brute force search instead.

```
for n in range(10, 100):
    if n == 7 * int(str(n)[1:]):
        print(n)
```

35

This code goes through all integers in the range [10, 99]. In general, `range(a, b)` where  $a \leq b$  are integers, denotes a list that contains  $a, a+1, \dots, b-1$  (empty if  $a = b$ ). To remove the first digit of a number, we convert it to a string (by calling the `str()` function), then use slicing (`[1:]`) to remove the first symbol of the resulting string, and finally convert the resulting string back to an integer.

**Problem 18** Find an integer that becomes 57 times smaller when its first digit is removed.

Solving this problem on a piece of paper is already not that easy, but it is possible to adjust our code above (try it!) and get 7125. Indeed,  $7125 = 57 \cdot 125$  (check that this is correct!). Note that you don't need to include the program in the solution: an example is enough. The questions on why and how this number was picked are irrelevant, since this example already answers the question. On the other hand, checking that  $7125 = 57 \cdot 125$  is a part of the solution (even if this part is left to the reader, as we did above).

**Stop and Think!** Do you see a way to find this example by hand?

Here is one possible line of reasoning. Let  $c$  be the first digit of the unknown number  $x$ , let  $z$  be the number  $x$  without the first digit, and let  $k$  be the number of digits in  $z$ .

$$x = \boxed{c} \quad \boxed{z}$$

$\xleftarrow{k}$

Then,

$$x = c \underbrace{00\dots00}_{k \text{ zeros}} + z = c \cdot 10^k + z.$$

Since  $x$  should become 57 times smaller when its first digit is removed,

$$c \cdot 10^k + z = 57 \cdot z.$$

By moving  $z$  to the right hand side, we get

$$c \cdot 10^k = 56 \cdot z. \tag{1.1}$$

**Stop and Think!** Can you find the value of  $c$  by looking at this equation? Recall that  $c$  is the first digit, i.e., an integer between 1 and 9.

Note that the right hand side of (1.1) is divisible by 7. And the only case<sup>4</sup> when the left hand side is divisible by 7, too, is  $c = 7$ . Plugging this into (1.1), we simplify it to

$$10^k = 8z.$$

Again, this means that  $10^k$  should be divisible by 8. Hence  $k$  is certainly greater than 2 as both  $10 = 10^1$  and  $100 = 10^2$  are not divisible by 8. However, 8 divides  $10^3$ : in this case,  $z = 125$ . This leads us to the final answer:  $x = 7125$ .

<sup>4</sup>Here we are cheating a bit: this is not that easy to see. To prove this, one needs some tools from basic number theory that we will discuss later. The left hand side of (1.1) can be factored as  $c \cdot 2^k \cdot 5^k$ , and this product is divisible by a prime number 7, so one of the factors should be divisible by 7. The only possibility is  $c = 7$ . But even if we know nothing about prime numbers and factorization, it would be a natural idea to try  $c = 7$ ; recall that we do not need to explain how the example is found.

**Stop and Think!** Are there other examples of a number that becomes 57 times smaller when its first digit is removed?

Note that  $10^k$  is divisible by 8 for any  $k \geq 3$ : indeed, if  $k \geq 3$ , then  $10^k$  is divisible by  $10^3$ . E.g., for  $k = 4$ , we get  $z = 1250$  and  $x = 71250 = 57 \cdot 1250$ . This way, we get an infinite series of solutions (for all integer  $k \geq 3$ ). All these solutions are obtained simply by padding our first solution 7125 by zeros.

**Problem 19** Are there other examples besides the solutions we have found for all  $k \geq 3$ ?

**Problem 20** Is there an integer that becomes 37 times smaller when its first digit is removed?

**Problem 21** Is there an integer that becomes 58 times smaller when its first digit is removed?

**Problem 22** Are there positive integers  $a, b, c$  such that  $a^3 + b^3 = c^3$ ? Are there positive integers  $a, b, c, d$  such that  $a^4 + b^4 + c^4 = d^4$ ?

We will discuss the answer to the last problem later in the book!

In some cases, nobody knows whether a simple existential statement is true or false. For example, nobody knows if there is a positive odd perfect integer  $N$  or not. Here “odd” means that  $N$  is not divisible by 2, and “perfect” means that  $N$  is equal to the sum of all its positive divisors, including 1 but excluding  $N$  itself (e.g.,  $N = 28$  is perfect since  $1 + 2 + 4 + 7 + 14 = 28$ , but 28 is not odd).

### 1.2.3 Proofs of Non-existence

A witness can certify that you made a payment to Mr. X or visited Moscow. But it would be much more difficult to prove that you did *not* pay Mr. X or that you have never been to Moscow. The situation with an existential statement is similar: to prove it, one example of an object with the required property is enough. But you cannot disprove it by providing an example or several examples of objects that do not have the required property; some general reasoning is needed to show that objects with the required property do not exist. We have already seen this kind of reasoning when proving that some regions are not tileable. In this section, we’ll give more examples of this type. (We will give rigorous mathematical notation for existential and universal statements, and for their negations, in Section 4.2.)

Imagine that you go for a hike with a friend and want to split the weight evenly, that is, to split items you take into two groups of the same weight. Try it: [Coursera](#).

**Stop and Think!** Suppose we have three items with weights 1, 2, and 3. Can we split them into two groups of the same weight?

This question is almost trivial:

$$1 + 2 = 3$$

gives a required split (1 and 2 go in one group, 3 goes in the other one).

Instead of splitting, we could use signs of positive or negative value to generate a zero sum:

$$\pm 1 \pm 2 \pm 3 = 0.$$

The weights with plus and minus signs form two groups of equal weights. Our example can be represented now as

$$+1 + 2 - 3 = 0 \text{ or } -1 - 2 + 3 = 0.$$

Now, let’s consider a more interesting example.

**Stop and Think!** Can we split the weights 1, 2, 3, 4, 5, and 7 into two groups of equal weights? (Reformulation: choose signs in  $\pm 1 \pm 2 \pm 3 \pm 4 \pm 5 \pm 7$  to get 0.)

**Stop and Think!** If we split these weights evenly, what is the weight in each group?

This is easy: the total weight is

$$1 + 2 + 3 + 4 + 5 + 7 = 22,$$

and if we have two equal parts, each one is 11. Hence, we can reformulate the problem as follows: *form a group of weight 11*. Note that it is enough to find one such group: the rest will automatically have the same weight 11.

**Stop and Think!** Do you see such a group?

Indeed it is easy to find one:

$$4 + 7 = 11.$$

As we observed, all the remaining weights sum up to the same number:

$$1 + 2 + 3 + 5 = 11.$$

Thus, the problem is solved.

There is no guarantee that an even splitting is always possible. Let us, for example, substitute 7 by 6 (one item is now lighter):

**Stop and Think!** Can we split the weights 1, 2, 3, 4, 5, and 6 into two groups of equal weights?

Let us try the same approach and compute the total weight. Now it is one unit smaller: 21, and each part should have weight  $21/2 = 11.5$ . This cannot happen since all weights are integers. This means that a required splitting does not exist (the corresponding existential statement is provably false, as a mathematician would say).

**Stop and Think!** Can we split the weights 2, 4, 6, 8, 10, and 12 into two groups of equal weights?

This is also impossible. Do you see why? It is essentially the same problem as the previous one but we have multiplied all the weight units by 2.

Alternatively, we can say that the total weight is 42, so if we split the weight into two equal parts, the weight of each part is 21. But all weights are even, so it is impossible to obtain the odd sum of weights (not divisible by 2).

**Stop and Think!** Can we split the weights 1, 2, 3, 4, 5, and 17 into two groups of equal weights?

The sum of weights is now

$$1 + 2 + 3 + 4 + 5 + 17 = 32,$$

so each part (if a splitting exists) has weight  $32/2 = 16$ , and this is an integer. Still the splitting is impossible.

**Stop and Think!** Do you see why?

The obstacle is easy to see: each part should have weight 16, and one of the items is too heavy (weight 17) for that. We see that the splitting is also impossible, but the obstacle is of a different type.

It would be nice if our list of obstacles was complete. If so, we would be able to check whether the splitting exists by checking whether it is prevented by one of the known obstacles. However, this problem is not nearly so simple. For this problem, no one knows the complete list of obstacles and no one knows a quick way to tell (for a given set of weights) whether the splitting exists.

This problem belongs to the class of so called *NP-complete problems*. Roughly speaking, this means that the problem is (at least currently) infeasible. More precisely, the situation is as follows. We do not know a way to solve this problem (and other NP-complete problems) efficiently, but we also do not know a proof that this is impossible. Technically, it is called “the P vs. NP problem” and it is one of the most famous and important unsolved mathematical problems (with a \$1M prize from [Clay Mathematics Institute](#)).

You see that our simple weight splitting problem is very close to the current boundary of Computer Science knowledge. Or, it might be better to say that the boundary is very close to it. No current approach works for ten thousand weights that are 64-bit integers (which is usually not much for computers to deal with). The exhaustive search in the space of all splittings is not feasible, and no significantly better approaches are known.

#### 1.2.4 Non-constructive Proofs

Can we prove the existence of an object with some property without giving an example? This sounds counterintuitive, but sometimes it is possible.

**Stop and Think!** There exists an integer  $n$  such that

$$2^n + 3^n \leq 10^{1000} \quad \text{and} \quad 2^{n+1} + 3^{n+1} > 10^{1000}$$

Do you see why?

For small  $n$  (say, for  $n = 1$ ) the first inequality is true. On the other hand, for large  $n$  the second one is true. Consider, for example,  $n = 4000$ . For this  $n$ , the first term alone is large enough:  $2^{n+1} > 2^{4000} = 16^{1000} > 10^{1000}$ . But we need to find  $n$  such that *both* inequalities are true at the same time. We need to find  $n$  such that  $2^n + 3^n \leq 10^{1000}$ , but increasing  $n$  by 1, we reverse the inequality.

**Stop and Think!** Do you see why there exists  $n$  with this property?

Intuitively, if we were inside and now are outside, we had to cross the boundary at some point. Let us increase  $n$  (going from 1 to 4000) and just stop before  $2^n + 3^n$  becomes greater than  $10^{1000}$ . This will give us the required value of  $n$ . This finished the proof, but we can also find the value of  $n$  as follows.

```
for n in range(4000):
    if 2 ** n + 3 ** n > 10 ** 1000:
        print(n - 1)
        break
```

2095

Another classical example deals with irrational numbers. Recall that a real number  $x$  is called *rational* if it can be represented as  $\frac{a}{b}$  where  $a$  and  $b$  are integers. Otherwise, it is called *irrational*. E.g.,  $\sqrt{2}$  is irrational. Indeed, assume that  $\sqrt{2} = \frac{a}{b}$  for some integers  $a, b$ . Then,  $2 = \frac{a^2}{b^2}$ , and hence  $2b^2 = a^2$ . We may assume that at least one of  $a$  and  $b$  is odd (if both are even, keep canceling the factor 2 until one becomes odd). If  $a$  is odd, then  $a^2$  is odd and cannot be equal to  $2b^2$ . Therefore,  $a$  is even and  $b$  is odd (due to our assumption). But this is also not possible: if  $a = 2k$ , then  $2b^2 = a^2 = 4k^2$ , so  $b^2 = 2a^2$ , but  $b^2$  should be odd for odd  $b$ . The contradiction shows that  $\sqrt{2}$  is irrational.

**Problem 23** Prove that there exist two irrational numbers  $x$  and  $y$  such that  $x^y$  is rational.

To prove this, consider  $x = y = \sqrt{2}$ . If  $x^y$  is rational, then we are done. Otherwise,  $x^y = (\sqrt{2})^{\sqrt{2}}$



Figure 1.25: There are ten pigeons and nine holes, therefore there must be a hole having more than one pigeon. (Source: [Wikipedia](#).)

is irrational. But then, raising it to the  $\sqrt{2}$  power gives a rational number:

$$\left((\sqrt{2})^{\sqrt{2}}\right)^{\sqrt{2}} = (\sqrt{2})^{\sqrt{2} \cdot \sqrt{2}} = (\sqrt{2})^2 = 2.$$

Note that we again proved that there exist  $x$  and  $y$  satisfying the requirements of Problem 23 without explicitly specifying the values of  $x$  and  $y$ !

**Problem 24** Prove that either  $\sin^2(10^{1000}) \geq 1/2$  or  $\cos^2(10^{1000}) \geq 1/2$ . [Hint: check out basic trigonometric identities or the Pythagorean theorem.]

It is not so easy to find out which of these two inequalities holds. Indeed, the first step of computing these two values would be to reduce  $10^{1000}$  radians to the interval  $[0, 2\pi]$ . To do this, one needs to know the value of  $\pi$  with very high precision (about a thousand decimal digits). Still, we know for sure that one of these values is guaranteed to be at least  $1/2$ , even though we do not know which one.

The previous two examples are based on the so-called *law of excluded middle* that says that, for any statement, either it is true or its negation is true. For example, for any number  $x$ , either  $x = 5$  or  $x \neq 5$ . We do not need to know the value of  $x$  to be sure that one of these two statements is true, and at the same time without knowing the value of  $x$  we do not know which of these two statements is actually true.

Another useful method for proving existence non-constructively is *the pigeonhole principle*: if there are more pigeons than holes, and each pigeon occupies some hole, then some two pigeons share the same hole (see Figure 1.25). As simple as it is, it has many non-trivial applications.

**Problem 25** Prove that there exists a positive integer divisible by 57 that uses only digits 0 and 1.

Consider the numbers  $1, 11, 111, 1111, 11111, \dots$  and divide each of these numbers by 57. We get a sequence of remainders:  $1 \bmod 57 = 1$ ,  $11 \bmod 57 = 11$ ,  $111 \bmod 57 = 54$ ,  $1111 \bmod 57 = 28$ ,  $11111 \bmod 57 = 53$  and so on ( $a \bmod 57$  is the remainder of  $a$  when divided by 57). Since this sequence is infinite and there are only 57 possible remainders modulo 57, at some point we should see repetitions, i.e.,

$$\underbrace{111\dots111}_{m \text{ digits}} \bmod 57 = \underbrace{111\dots111}_{n \text{ digits}} \bmod 57$$

for  $m \neq n$ .

**Stop and Think!** Do you see how this helps?

If two numbers have the same remainder when divided by 57, then their difference is a multiple of 57. But the difference has decimal representation  $11\dots1100\dots00$ , so it is the desired example. (Assuming  $m > n$ , we have  $n$  zeros at the end and  $m - n$  ones before them.)

In fact, one can find a number of the form  $111\dots111$  that is divisible by 57, the trailing zeros

are not needed. We will see later in the book that trailing zeros do not help either, since 10 is coprime with 57 (i.e., the only positive number that divides 10 and 57 is 1).

**Problem 26** Using a Python program (if needed), find a number of the form 111...111 that is divisible by 57.

**Problem 27** As of December 2020, there are 66758 learners enrolled into our [Mathematical Thinking in Computer Science](#) course. Prove that there are 183 of them that share a birthday.

Often a non-constructive proof can be replaced by a constructive one. Sometimes a brute force search helps; sometimes another argument is needed. For Problem 23, one may note that

$$(\sqrt{2})^{2\log_2 3} = 2^{\log_2 3} = 3$$

and both  $\sqrt{2}$  and  $2\log_2 3$  are irrational (and 3 is rational).

**Problem 28** We discussed why  $\sqrt{2}$  is irrational above. Show that  $2\log_2 3$  is also irrational. [Hint: unique factoring is useful here.]

One may ask what actually happens with  $\sqrt{2}^{\sqrt{2}}$ . The [Gelfond – Schneider theorem](#) guarantees that this number is irrational (and even *transcendental* which means that it is not a root of a non-zero polynomial with integer coefficients).

We end this section with two examples where finding a constructive proof is difficult. Let us generate 30 seven-digit numbers.

```
from random import randint, seed
seed(14)

for i in range(30):
    print(randint(10 ** 6, 10 ** 7 - 1), end=' ')
    if i % 6 == 5:
        print()
```

```
2792285 9843470 5142886 5548558 5291201 5882438
2218459 8545410 6083294 8828556 7655079 7607029
2986415 5421072 4745783 6295863 7006791 5368826
7051040 9661551 3511608 3701978 5619271 3767372
1177927 2173344 3064039 6655032 1466196 2395998
```

The function `randint(a,b)` returns a random integer in the range  $a \dots b$ . We call it 30 times, using some additional tricks to have a nice printout. The second argument `end=' '` tells the `print` function that it should print a space character (instead of the newline character as it does by default). Then a call `print()` is used to get a newline character after groups of six numbers. (The call `seed()` initializes the pseudorandom generator used. This is done for reproducibility: it ensures that every time this program is called, the same 30 integers are generated. To get different 30 numbers, just change the argument of the `seed` function.)

**Problem 29** Prove that there exist two disjoint subsets of this set of 30 integers that have equal sum. (In other words, we can color some of the elements blue, and color some other elements red so that the sum of the red numbers is equal to the sum of the blue numbers.)

It looks counterintuitive: the numbers are rather long, and having two equal sums looks at first as a strange coincidence. Still, the following (non-constructive) argument proves our claim. For every subset  $A$  of this 30-element set, consider an integer  $S(A)$ , the sum of all elements in  $A$ . All  $S(A)$  are less than  $30 \cdot 10^7$  (about third of a billion). On the other hand, we have  $2^{30}$  subsets (each

can be described by a 30-bit string saying which of the numbers are included and which are not, and there are  $2^{30}$  binary strings of length 30). Now the crucial point (pigeonhole principle):

since  $2^{30} = 1024^3 > 10^9 > 30 \cdot 10^7$ , there exist different  $A$  and  $B$  such that  $S(A) = S(B)$ .

These  $A$  and  $B$  may include some common numbers, but these numbers can be deleted and still  $S(A) = S(B)$  (since we delete the same numbers). This is exactly what we claimed. (Note that  $A$  and  $B$  are both non-empty: since  $A \neq B$ , at least one of them is non-empty and has positive sum, and the other has the same sum.) Still, this non-constructive argument does not give any information about these two subsets.

Later in the book, we present a simple Python code that finds two such subsets using ILP-solvers.

Our last example is *factoring*: one can prove using Fermat's little theorem<sup>5</sup> that the number

```
4120234369866595438555313653325759481798
1169984432798284545562643387644556524842
6198098870423161841879261420247188869492
5609317763750334211309823974851509449091
0691026986103186270411488086697056490290
3653658867433731720813104105190864254793
282601391257624033946373269391
```

(it does not fit in a single line!) is *composite*, which means that it is a product of two smaller integers. It is an existential statement (there *exists* a factoring), but nobody has been able to find such a factoring. (There even was a \$70000 reward ↗ for doing this.)

Another common (and powerful!) way of proving something non-constructively is called the *probabilistic method*. Roughly, its main idea is the following: to prove that an object satisfying a particular property exists, take a *random* object and show that there is a *non-zero* chance that it satisfies the required property. From this we conclude that an object satisfying the property *exists*. We will see applications of this method later in the book.

### Summary

- A chessboard without two cells can be tiled by domino tiles if and only if the deleted cells are of opposite colors.
- Structure of a proof reflects the structure of the claim.
- One example suffices to prove an existential statement.
- A general argument is required to prove that an existential statement is false.
- Non-constructive proofs show that a certain object exists without giving an example.

---

<sup>5</sup>Fermat's little theorem states that  $(a^p - a)$  is divisible by  $p$  for any integer  $a$  and any prime  $p$ . We will discuss this theorem later.

## 2. Finding an Example

How can we know that an object with certain properties exists? In Section 1.2, we saw that it suffices to give an example of such an object, but finding an example might be a hard problem. One way to find an example is to go through all objects and check whether at least one of them meets the requirements. However, in many cases, the search space is enormous. A computer may help, but some reasoning that *narrow*s the search space is important both for computer search and for “bare hands” work. In this chapter, we will learn various techniques for showing that an object exists and that an object is optimal among all objects (say, the smallest or largest object that meets the requirements).

### 2.1 How to Find an Example

We start our journey with the famous engraving “Melencolia I” by Albrecht Dürer! Note the  $4 \times 4$  square in the upper right corner of the engraving containing all the numbers from 1 to 16 (see Figure 2.1). If you sum up all the numbers in each row, each column, and both diagonals; you will find all are equal. Such a square is called “magic square”.

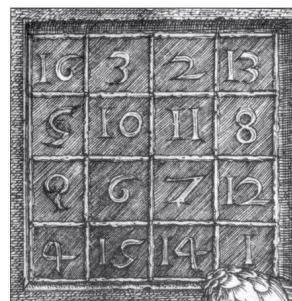


Figure 2.1: Melencolia I by Albrecht Dürer, and the magic square in its upper right corner. (Source: [Wikipedia](#).)

**Stop and Think!** – **Recover Dürer's engraving.** Some parts of Dürer's engraving are unclear. Can you use the magic square property to recover the first number in the second row? The first number in the third row?

We know that the sum in each row is the same, but *what* is this sum? Since all numbers from 1 to 16 appear in the table once, we know that the sum of all entries in the table is  $1 + 2 + \dots + 16 = 136$ . On the other hand, each row has the same sum, and there are 4 rows in the table. This tells us that the sum in each row is exactly 34. Hence, the first number in the second row is 5, and the first number in the third row is 9.

In general, an  $n \times n$  magic square is a square where each integer from 1 to  $n^2$  appears exactly once, and the sums of the numbers in each row, each column, and both diagonals are the same. For what values of  $n$ , do  $n \times n$  magic squares even *exist*? From Dürer's engraving we know that  $4 \times 4$  magic squares exist. What about  $n \times n$  magic squares for other values of  $n$ ?

When  $n = 1$ , the problem is trivial. Namely, we have a  $1 \times 1$  table with number 1 written in its only cell. Of course, all sums are the same in this table.

**Stop and Think!** –  **$2 \times 2$  magic squares.** Do  $2 \times 2$  magic squares exist?

No, there are no  $2 \times 2$  magic squares! Let us prove this. Consider a  $2 \times 2$  table with numbers  $a, b, c, d$ :

|     |     |
|-----|-----|
| $a$ | $b$ |
| $c$ | $d$ |

To satisfy the magic square requirement, the sums in the first row and in the first column must be the same:  $a + b = a + c$ , i.e.,  $b = c$ , which contradicts our assumption that all numbers in the table are distinct.

**Problem 30** –  **$3 \times 3$  magic squares.** Do  $3 \times 3$  magic squares exist? Try it: [Coursera](#).

Since there are too many different  $3 \times 3$  tables to check them all, we will first narrow the search space.

### 2.1.1 Narrowing the Search Space

We want to find a  $3 \times 3$  magic square. Should we just try all  $3 \times 3$  tables with numbers from 1 to 9?

**Stop and Think!** – **Number of  $3 \times 3$  tables.** How many  $3 \times 3$  tables with all numbers from 1 to 9 are there?

There are 9 choices for which number will go in the first cell, 8 choices for which number will go in the second cell, 7 choices for which number will go in the third cell, and so on. This tells us that the total number of such tables is  $9! = 1 \times 2 \times \dots \times 9 = 362\,880$ . The exclamation point here denotes the “factorial” of 9, i.e., the product of all integers from 1 to 9. We will discuss this in more detail and see more examples of such calculations later in the book.

While a computer can handle 362 880 cases, it is way too many for us. Let us try to decrease the number of cases we need to consider. Recall that the sums in each row, column, and diagonal are the same. Let us denote this sum by  $S$ .

**Stop and Think!** – **Sum in each row.** What is the value of  $S$ —the sum in each row, column, and diagonal?

The total sum of all entries in the table is  $1 + 2 + \dots + 9 = 45$ . And the sum of the three rows of the table, i.e.,  $3S$ , is 45 too. Thus, we have that  $S = 15$ . In fact, this simple observation already narrows the search space a lot.

**Stop and Think!** Imagine that we know the four numbers in the upper left corner of a magic square (see Figure 2.2), can we compute all other numbers in the table?

|       |       |       |
|-------|-------|-------|
| $a_1$ | $a_2$ | $a_3$ |
| $a_4$ | $a_5$ | $a_6$ |
| $a_7$ | $a_8$ | $a_9$ |

Figure 2.2: Given the four highlighted entries in the upper left corner of a magic square, we can compute the values of the remaining entries.

We know the sum  $S = 15$  in each line. Hence, given the numbers  $a_1, a_2, a_4, a_5$ , we can compute all the remaining numbers  $a_3, a_6, a_7, a_8$ , and  $a_9$ . Now, we only need to find out the numbers  $a_1, a_2, a_4$ , and  $a_5$ .

**Stop and Think!** How many choices for the values  $a_1, a_2, a_4$ , and  $a_5$  are there?

Let us count the number of choices: there are 9 choices for  $a_1$ ; then for each value of  $a_1$ , there are 8 choices for  $a_2$ ; then 7 choices for  $a_4$ ; finally, 6 choices for  $a_5$ . This gives us  $9 \times 8 \times 7 \times 6 = 3024$  choices. We do not need to consider all 362880 tables anymore, because we reduced the size of the search space by a factor of 120: now it suffices to check only 3024 tables. Let us not rush to check all those tables, but rather try to eliminate more cases.

**Problem 31 – Value of the central entry.** There is only one possible value of the central element  $a_5$  in a  $3 \times 3$  magic square. Can you find this value?

Consider the sums of all entries along the four lines shown in Figure 2.3. On one hand, these

|       |       |       |
|-------|-------|-------|
| $a_1$ | $a_2$ | $a_3$ |
| $a_4$ | $a_5$ | $a_6$ |
| $a_7$ | $a_8$ | $a_9$ |

Figure 2.3: Sums of the numbers along the four highlighted lines contain each entry of the table once, but the central entry  $a_5$  four times.

four sums together should give  $4S = 60$ . On the other hand, they include each entry  $a_i$  once, but  $a_5$  is counted four times. This gives us:

$$4S = 3S + 3a_5.$$

We conclude that  $a_5 = S/3 = 15/3 = 5$ . Now, we know the value of the central entry in every  $3 \times 3$  magic square, it is always 5, and we have only  $9 \times 8 \times 7 = 504$  tables remaining!

Now, we will see where one can place the number 1 in the table. Since our table is symmetric, there are only two possibilities: either 1 is in a corner, or 1 is in the middle of one of the lines. We consider these two cases.

**Case I. 1 is in a corner.** For example, let  $a_1 = 1$ , see Figure 2.4.

|       |       |       |
|-------|-------|-------|
| 1     | $a_2$ | $a_3$ |
| $a_4$ | 5     | $a_6$ |
| $a_7$ | $a_8$ | $a_9$ |

Figure 2.4: One is placed in a corner:  $a_1 = 1$ . In this case,  $a_9 = 9$ , and it is impossible to simultaneously satisfy  $a_2 + a_3 = a_4 + a_7 = 14$ .

From  $1 + 5 + a_9 = 15$ , we have that  $a_9 = 9$ . Also,  $a_2 + a_3 = a_4 + a_7 = 14$ . Can we find distinct numbers  $a_2, a_3, a_4, a_7$  from 1 to 9 satisfying this equation? Yes, there are two ways to get the

sum 14:  $9 + 5 = 8 + 6 = 14$ . But we have already placed 9 at the bottom right corner. This means that there is no way to satisfy the magic square requirement in the first column and the first row simultaneously. We conclude that 1 cannot be placed in a corner.

**Case II. 1 is in the middle.** For example,  $a_2 = 1$ , see Figure 2.5.

|       |       |       |
|-------|-------|-------|
| $a_1$ | 1     | $a_3$ |
| $a_4$ | 5     | $a_6$ |
| $a_7$ | $a_8$ | $a_9$ |

Figure 2.5: One is placed in the middle of a row:  $a_2 = 1$ .

By inspecting the middle column, we see that  $a_8$  must equal 9. Again, we have that  $a_1 + a_3 = 14$ , and we already know that this implies that one of these numbers is 8, and the other one is 6. Since  $a_1$  and  $a_3$  are symmetric so far, without loss of generality we can assume that  $a_1 = 8$  and  $a_3 = 6$  (see Figure 2.6).

|       |   |       |
|-------|---|-------|
| 8     | 1 | 6     |
| $a_4$ | 5 | $a_6$ |
| $a_7$ | 9 | $a_9$ |

Figure 2.6: One is placed in the middle of a row:  $a_2 = 1$ . We can compute the values of  $a_1, a_3, a_8$ .

Actually, we already have enough information about this magic square to uniquely identify all the remaining numbers in the table. From one of the diagonals, we have that  $8 + 5 + a_9 = 15$ , or,  $a_9 = 2$ . Similarly,  $a_7 = 4$ . Then  $a_4 = 3$  and  $a_6 = 7$  (see Figure 2.7).

|   |   |   |
|---|---|---|
| 8 | 1 | 6 |
| 3 | 5 | 7 |
| 4 | 9 | 2 |

Figure 2.7: One is placed in the middle of a row:  $a_2 = 1$ . We can compute all other entries.

It only remains to verify that we used each number exactly once, and that all rows, columns, and diagonals sum up to 15. This concludes our proof of the existence of  $3 \times 3$  magic squares. Note that we reduced the number of cases we needed to consider from 362880 to two!

We have proved that there exist  $3 \times 3$  magic squares. Actually it is possible to show that for every  $n > 2$ , there is at least one magic square of size  $n \times n$ . Below, we will see other techniques which often come handy when trying to find examples.

### 2.1.2 Using Known Results

We have constructed magic squares, but what can we say about their *multiplicative* versions? Let us say that a  $3 \times 3$  table with integers is a multiplicative magic square if the *products* of numbers in each row, column, and diagonal are the same.

**Stop and Think!** Is there a multiplicative magic square that contains each of the numbers from 1 to 9 exactly once?

No, there are no such squares. Indeed, wherever we place the number 5, there will be lines which contain 5, and which do not. The products in the former lines will be multiples of 5, while the products in the latter lines will not be multiples of 5, which contradicts the multiplicative

magic square requirement. (We explain this divisibility issue in much greater detail later in the book.)

Then, let us try to find magic squares with arbitrary integers. Is this easy? Yes, we can just write the same number in every cell of the table. This finally leads us to an interesting problem.

**Problem 32** Does there exist a multiplicative  $3 \times 3$  magic square with distinct integers without the restriction that those must be the numbers 1 through 9?

Should we try to repeat the search that we did in Section 2.1.1? There we had to use numbers from 1 to 9, and here we do not have such a restriction. Thus, the search space in our current problem appears to be infinite. As the section title suggests, we should try to use what we have learned until now. Can we take a  $3 \times 3$  magic square; and transform it into a multiplicative magic square? Is there an operation that turns sums into products? Yes, we can use exponentiation! Namely, pick your favorite integer  $c > 1$ , e.g.,  $c = 2$ . And notice that for every  $x, y$ :

$$c^x \cdot c^y = c^{x+y}.$$

**Stop and Think!** Do you see how to modify the magic square from Figure 2.7 into a multiplicative magic square?

We can take a magic square from the previous section, and replace every number  $x$  in it with  $2^x$  (see Figure 2.8). Previously we had that if three numbers  $x, y, z$  formed a line, then  $x + y + z = 15$ . In the new table with numbers  $2^x, 2^y, 2^z$ , we have the same property for their products:

$$2^x \cdot 2^y \cdot 2^z = 2^{x+y+z} = 2^{15}.$$

|   |   |   |
|---|---|---|
| 8 | 1 | 6 |
| 3 | 5 | 7 |
| 4 | 9 | 2 |

|       |       |       |
|-------|-------|-------|
| $2^8$ | $2^1$ | $2^6$ |
| $2^3$ | $2^5$ | $2^7$ |
| $2^4$ | $2^9$ | $2^2$ |

|     |     |     |
|-----|-----|-----|
| 256 | 2   | 64  |
| 8   | 32  | 128 |
| 16  | 512 | 4   |

(a)

(b)

(c)

Figure 2.8: (a) “Additive” magic square from Figure 2.7; (b) The corresponding multiplicative magic square; (c) The corresponding multiplicative magic square where we have computed the values of all entries.

This gives us a multiplicative magic square. Indeed, the product in each row, column, and diagonal is  $2^{15}$ , and all numbers in the table are distinct integers. Note that we did not need to go through many cases: we just showed how to use a magic square to build a multiplicative magic square.

**Problem 33** The largest number in our multiplicative magic square (Figure 2.8) is 512. Do you see a way to build a multiplicative magic square with distinct positive integers where the largest number is less than 300?

Notice that all numbers in Figure 2.8 are even, so we can divide all numbers in the table by 2 while preserving the multiplicative magic property (see Figure 2.9). (In other words, we can use the numbers from  $2^0$  to  $2^{14}$  instead of  $2^1$  to  $2^{15}$ .)

There is a construction of a multiplicative magic square where all numbers are smaller than 40. We start with two distinct “additive” magic squares with very small but not necessarily distinct entries (Figure 2.10 (a) and (c)). We transform one of them into a multiplicative magic square (with not necessarily distinct entries) by raising  $c = 2$  to the corresponding powers (Figure 2.10 (b)). Similarly, we transform the other one by raising  $c = 3$  to the corresponding powers

|     |     |    |
|-----|-----|----|
| 128 | 1   | 32 |
| 4   | 16  | 64 |
| 8   | 256 | 2  |

Figure 2.9: Another multiplicative magic square.

(Figure 2.10 (d)). In Figure 2.10 (b) and (d), we have two multiplicative magic squares with very small entries, but alas, the entries are not distinct. How can we combine two multiplicative magic squares into a new one? We can take two squares, and multiply entries located in the same positions to get another multiplicative square. It is easy to verify that the final multiplicative magic square (Figure 2.10 (e)) has 9 distinct numbers (and each of them is less than 40). Here it is important that even though each of the squares in (a) and (c) have some duplicate values, duplicates cannot be in the same locations in both (a) and (c).

|   |     |     |     |     |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |  |   |    |   |   |   |   |    |   |    |
|---|-----|-----|-----|-----|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|--|---|----|---|---|---|---|----|---|----|
| <table border="1"> <tr><td>1</td><td>2</td><td>0</td></tr> <tr><td>0</td><td>1</td><td>2</td></tr> <tr><td>2</td><td>0</td><td>1</td></tr> </table> | 1   | 2   | 0   | 0   | 1 | 2 | 2 | 0 | 1 | <table border="1"> <tr><td>2</td><td>4</td><td>1</td></tr> <tr><td>1</td><td>2</td><td>4</td></tr> <tr><td>4</td><td>1</td><td>2</td></tr> </table> | 2 | 4 | 1 | 1 | 2 | 4 | 4 | 1 | 2 | <table border="1"> <tr><td>0</td><td>2</td><td>1</td></tr> <tr><td>2</td><td>1</td><td>0</td></tr> <tr><td>1</td><td>0</td><td>2</td></tr> </table> | 0 | 2 | 1 | 2 | 1 | 0 | 1 | 0 | 2 | <table border="1"> <tr><td>1</td><td>9</td><td>3</td></tr> <tr><td>9</td><td>3</td><td>1</td></tr> <tr><td>3</td><td>1</td><td>9</td></tr> </table> | 1 | 9 | 3 | 9 | 3 | 1 | 3 | 1 | 9 | <table border="1"> <tr><td>2</td><td>36</td><td>3</td></tr> <tr><td>9</td><td>6</td><td>4</td></tr> <tr><td>12</td><td>1</td><td>18</td></tr> </table> | 2 | 36 | 3 | 9 | 6 | 4 | 12 | 1 | 18 |
| 1   | 2   | 0   |     |     |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |  |   |    |   |   |   |   |    |   |    |
| 0   | 1   | 2   |     |     |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |  |   |    |   |   |   |   |    |   |    |
| 2   | 0   | 1   |     |     |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |  |   |    |   |   |   |   |    |   |    |
| 2   | 4   | 1   |     |     |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |  |   |    |   |   |   |   |    |   |    |
| 1   | 2   | 4   |     |     |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |  |   |    |   |   |   |   |    |   |    |
| 4   | 1   | 2   |     |     |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |  |   |    |   |   |   |   |    |   |    |
| 0   | 2   | 1   |     |     |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |  |   |    |   |   |   |   |    |   |    |
| 2   | 1   | 0   |     |     |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |  |   |    |   |   |   |   |    |   |    |
| 1   | 0   | 2   |     |     |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |  |   |    |   |   |   |   |    |   |    |
| 1   | 9   | 3   |     |     |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |  |   |    |   |   |   |   |    |   |    |
| 9   | 3   | 1   |     |     |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |  |   |    |   |   |   |   |    |   |    |
| 3   | 1   | 9   |     |     |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |  |   |    |   |   |   |   |    |   |    |
| 2   | 36  | 3   |     |     |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |  |   |    |   |   |   |   |    |   |    |
| 9   | 6   | 4   |     |     |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |  |   |    |   |   |   |   |    |   |    |
| 12  | 1   | 18  |     |     |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |  |   |    |   |   |   |   |    |   |    |
| (a)   | (b) | (c) | (d) | (e) |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |  |   |    |   |   |   |   |    |   |    |

Figure 2.10: (a) “Additive” magic square with repetitions; (b) A multiplicative magic square with repetitions, constructed by raising  $c = 2$  to the powers that show up in the additive version from (a); (c) Another “additive” magic square with repetitions; (d) A multiplicative magic square with repetitions, constructed by raising  $c = 3$  to the powers that show up in the additive version from (c); (e) Product of (b) and (d) — multiplicative magic square with small distinct entries.

### 2.1.3 Identify More Properties

It often helps to identify more properties of an object we are looking for and use these properties to narrow the search space.

**Problem 34** Does there exist a six-digit integer that starts with 100 and is a multiple of 9127?

We are looking for a number that starts with 100 and has three more digits (from 000 to 999) and is divisible by 9127. It gives us 1000 candidate numbers, and we should just check whether one of them is a multiple of 9127. A computer can easily do this.

```
for i in range(100000, 101000):
    if i % 9127 == 0:
        print(i)
```

100397

On the other hand, we can find the answer without a computer. Recall that our number must be a multiple of 9127. It is an important property which significantly reduces the search space: Instead of checking all six-digit numbers starting with 100, let us only look at multiples of 9127. We want to find an integer  $k$  such that

$$100000 \leq 9127k \leq 100999.$$

For this we just need to divide both 100000 and 100999 by 9127:

$$10.95\dots \leq k \leq 11.06\dots$$

Thus, we take  $k = 11$ , and the answer to the problem is  $9127k = 100397$ . Our reasoning also shows that 100397 is the only number satisfying the given requirements.

**Problem 35** Does there exist a three-digit integer that has remainder 1 when divided by 2, 3, 4, 5, 6, 7?

Again, we can write a simple program that checks all three-digit numbers.

```
for n in range(100, 1000):
    if all(n % m == 1 for m in range(2, 8)):
        print(n)
```

```
421
841
```

Instead, we could easily find such numbers by revealing other properties that they have. Assume that  $N$  has remainder 1 when divided by 2, 3, 4, 5, 6, 7, then what can we say about the number  $N - 1$ ?  $N - 1$  must be a multiple of 2, 3, 4, 5, 6, and 7. That is,  $N - 1$  must be a multiple of the *least common multiple* of 2, 3, 4, 5, 6, 7 which is  $4 \times 3 \times 5 \times 7 = 420$ . (The least common multiple of a set of numbers is the smallest positive integer that is divisible by all of them — this is a topic we will cover in great detail later in the book) Thus,  $N - 1 = 420k$  for an integer  $k$ , and  $N = 420k + 1$ . We do not take  $k = 0$  or  $k > 2$ , because we are looking for a three-digit number  $N$ . On the other hand, both  $k = 1$  and  $k = 2$  give us examples of the desired object:  $N = 421$  and  $N = 841$ .

**Problem 36** Does there exist an integer  $n$  such that  $n^2$  starts with 31415?

The search space for this problem is seemingly infinite: there are infinitely many integers. Can we identify useful properties of  $n$  that would simplify the search? We have two cases: the remaining number of digits is odd or even. We'll show the process with an even number, and let you do the same for the odd case in Problem 37. Assume that  $n^2$  starts with 31415 followed by  $2k$  digits:

$$n^2 = 31415d_1d_2\dots d_{2k}.$$

Let us look at the square of  $(n/10)$ . Since  $(n/10)^2 = n^2/100$ , this number is just  $n^2$  with a decimal point preceding the last two digits:

$$(n/10)^2 = 31415d_1d_2\dots d_{2k-2}.d_{2k-1}d_{2k}.$$

Now instead of dividing  $n$  by 10, let us divide it by  $10^k$ .

$$(n/10^k)^2 = 31415.d_1d_2\dots d_{2k}.$$

This already gives us the first digits of  $n$ . Indeed, by taking the square root of both sides of the last equation, from  $\sqrt{31415} = 177.242771\dots$  and  $\sqrt{31416} = 177.245592\dots$ , we have that

$$177.242771 < (n/10^k) < 177.245593.$$

In particular, we can take  $n/10^k = 177.243$ , e.g.,  $n = 177243$  and  $k = 3$ . This gives us an example of such a number, because  $n^2 = 31415081049$ .

**Problem 37** Can you find an integer  $n$  starting with 5 such that  $n^2$  starts with 31415?

In order to solve this problem, we suggest repeating our solution but this time assume that  $n^2$  starts with 31415 followed by an *odd* number of digits.

#### 2.1.4 Solve a Smaller Problem

**Problem 38** Alice and Bob live in a country where only coins of values 7 and 13 florins are used. Luckily, Alice and Bob have infinitely many coins of each type. Find all integer values  $c$  such that Alice can pay Bob  $c$  florins.

Let us see an example first. If Alice wants to pay Bob 6 florins, she can give Bob a 13-florin coin, and get a 7-florin coin back. Thus, if  $c = 6$ , Alice can pay Bob  $c$  florins.

**Stop and Think!** How can we check all values of  $c$ ?

In such cases, a very useful technique is to identify important special cases.

**Stop and Think!** Is there a value of  $c$  such that if Alice can give  $c$  florins to Bob, then Alice can pay Bob *any* amount of money?

If Alice can pay  $c = 1$  florin, then she can pay any other amount, too! And it is not difficult to pay one florin. We already know that Bob can pay Alice 6 florins. After this, Alice just gives one 7-florin coin to Bob, and in total Alice paid Bob one florin. If we repeat this procedure  $c$  times, then Alice pays Bob  $c$  florins.

By identifying the most important special case ( $c = 1$ ), and finding an example for this case, we solved the problem for all values of  $c$ !

**Problem 39** Do both Alice and Bob need both kinds of coins? In other words, can Alice pay Bob  $c$  florins if Alice has only 7-florin coins and Bob has only 13-florin coins? Try it: [Coursera](#).

Consider another version of this problem.

**Problem 40** Alice and Bob have infinitely many coins of values 15 and 21 florins. Find all values  $c$ , such that Alice can pay Bob  $c$  florins.

Can Alice pay Bob  $c = 1$  florin? No, however many coins of values 15 and 21 they use, the amount that Alice pays Bob is always a multiple of three. Indeed, 15 and 21 are multiples of three, so any combination of these coins is also a multiple of three. This means that if  $c$  is not divisible by 3, then Alice *cannot* pay  $c$  florins.

**Stop and Think!** Assume that  $c$  is divisible by 3, can Alice pay  $c$  florins then?

Again, we do not need to check all such values of  $c$ , it is enough to check just one “main” value: if Alice can pay 3 florins, then she can pay any multiple of 3, too. We find one example for  $c = 3$ :

$$3 = (3 \times 15) - (2 \times 21)$$

and solve the problem. Alice can pay  $c$  florins *if and only if*  $c$  is a multiple of 3. Indeed, *if*  $c$  is a multiple of 3, then Alice can pay  $c$  florins by repeatedly paying 3 florins. On the other hand, Alice can pay  $c$  florins *only if*  $c$  is a multiple of 3 (recall that any combination of 15 and 21 is a multiple of 3).

**Problem 41** We have coupons for 10 free nights at the hotel A, 15 free nights at the hotel B, and 20 free nights at the hotel C. However, there is a restriction: one cannot spend two nights in a row at the same hotel. Can we use these coupons and stay at the hotels A, B, and C for  $10 + 15 + 20 = 45$  consecutive nights? Try it (question 1): [Coursera](#).

One way to solve this problem is to identify a smaller and simpler sub-problem. Solving it will lead us to a solution to the original, larger problem. 10, 15, and 20 are multiples of 5, and if we divide all these numbers by 5, then we get a smaller version of this problem.

**Stop and Think!** Now we have 2 coupons for A, 3 coupons for B, and 4 coupons for C. Say we find a solution for this smaller problem, couldn't we just join five copies of this solution to resolve our larger, original problem?

Almost: we want the first and the last hotels in our small solution to be different, so that when we glue them together we do not spend two consecutive nights at the same hotel.

Hence, we only need to find an example of a schedule where one spends 2, 3, and 4 nights at the hotels A, B and C, such that the first and the last hotels are distinct. This small problem is easy to solve: since C has the greatest number of coupons, let us start there, and suggest we spend every other night at C:

$$\cdot C \cdot C \cdot C \cdot C \cdot$$

And let us add *As* and *Bs* in between:

*A C A C B C B C B*

We have solved the smaller problem, and we can stack together five copies of this solution to solve the original problem.

**Problem 42** You have coupons for 10 free nights at the hotel *A*, 15 free nights at the hotel *B*, and 30 free nights at the hotel *C*. However, there is a restriction: you cannot spend two nights in a row at the same hotel. Can you use these coupons and stay at the hotels *A*, *B*, and *C* for  $10 + 15 + 30 = 55$  consecutive nights? Try it (question 2): [Coursera](#).

Note that even if one spends every other night at the hotel *C*, one still needs to fill 29 gaps between them. We cannot do this, because we only have 25 coupons for hotels *A* and *B*. Therefore, it is impossible to use your coupons for 55 consecutive nights. We identified an obstacle, we proved that there is no schedule for 55 nights, and, thus, there is no use in trying to find an example.

### Summary

- In order to prove that an object exists, it is enough to find one example of such an object.
- Check if you see any obstacles to the existence of such objects.
- Try to narrow the search space.
- Check whether known results let you simplify the problem.
- Identify more properties of the desired object to narrow the search space.
- Identify smaller problems which can lead to a solution for the original problem.

Try to use these techniques to solve the following problems Try it: [Coursera](#).

**Problem 43** Does there exist a 5-digit integer *N* such that  $N^2$  starts with 27182?

**Problem 44** You have 25 coupons for the hotel *A*, and 20 coupons for the hotel *B*. What is the maximum number of coupons for the hotel *C* that you can use if you are not allowed to spend two consecutive nights at the same hotel?

**Problem 45** There are fewer than 100 books on a shelf. If you pack them into boxes that contain 3 books each, then 2 books remain on the shelf. If you pack them into boxes that contain 4 books each, then 3 books remain on the shelf. Finally, if you pack them into boxes that contain 5 books each, then 4 books remain. How many books are there on the shelf?

**Problem 46** Some country uses only coins of values 12, 20, and 30 florins. A person living in this country wants to pay some amount to her neighbor. What is the smallest (positive) amount that can be paid if both the person and her neighbor have infinitely many coins of each type?

## 2.2 Optimality

In *optimality* problems we aim to find the maximum possible value of some parameter. To solve such a problem, we should find this optimal value *c* and prove two claims: (a) *c* can be achieved (there is a way to get *c*) and (b) bigger values are impossible (all possible values of the parameter are less than or equal to *c*). The part (a) is an existential statement (and one example is enough to prove it), the part (b) is a universal statement (some argument that covers all cases is needed).

### 2.2.1 Producing Chocolate

**Problem 47 – Producing chocolate.** A factory produces milk chocolate (\$10 profit per box) and dark chocolate (\$30 profit per box). The daily demands are 500 and 200 boxes for milk and dark chocolate, respectively. The factory's maximum capacity is 600 boxes of chocolate per day. (This means that the factory can produce any combination of milk and dark chocolate boxes, the only requirement is that the total number of boxes produced is at most 600 per day.) What

is the optimal production plan, that is, what is the maximum profit per day?

**Stop and Think!** Can the factory work at full capacity and still sell all the chocolate produced? What would be the best production plan if we had unlimited demand for both types of chocolate?

The total demand for both types is  $500 + 200 = 700$  boxes per day, and this exceeds the total capacity, so we can sell all the chocolate assuming we choose the types correctly (say, 450/150 for milk/dark chocolate). If demand is unlimited, it is more profitable to produce only dark chocolate (30 profit per box instead of 10), i.e., 600 boxes of dark chocolate (more than allowed by the actual demand).

If you have some business experience, probably at this point you would laugh and say: what a stupid problem — of course, one should produce as much of the more profitable dark chocolate as one can sell, that is, 200 boxes of dark chocolate, and use the rest of the production capacity (400 boxes per day) for milk chocolate. This would give daily profit  $200 \cdot 30 + 400 \cdot 10 = 6000 + 4000 = 10000$ . And your intuition would be right. Still, we need some simple examples to work with.

We claim that the maximum profit per day is \$10000 and want to prove it. What does it mean and what should we prove? We need to show two claims:

1. *Existential statement*: there exists a production plan achieving a profit of \$10000.

2. *Universal statement*: all plans give profit at most \$10000, so there is no better one.

To show the first claim, it is enough to show a production plan achieving \$10000: 400 boxes of milk chocolate and 200 boxes of dark chocolate per day. It is indeed a valid plan: no more than 500 boxes of milk chocolate, no more than 200 boxes of dark chocolate, and no more than 600 boxes in total. The resulting profit is

$$\$10 \times 400 + \$30 \times 200 = \$10000.$$

To prove the second claim, we use the following argument that looks strange at first, but is formally correct. Denote by  $M$  and  $D$  the number of boxes of milk and dark chocolate produced per day. From the problem statement, we know that

$$M \leq 500, \tag{2.1}$$

$$D \leq 200, \tag{2.2}$$

$$M+D \leq 600. \tag{2.3}$$

The profit is equal to

$$10M + 30D.$$

Now, let us multiply the inequality (2.2) by 20 and multiply the inequality (2.3) by 10:

$$20D \leq 4000,$$

$$10M + 10D \leq 6000.$$

By summing up these two inequalities, we show that the profit per day is always at most \$10000:

$$10M + 30D \leq 10000.$$

But why on earth did we decide to multiply the second inequality by 20, the third by 30, and then sum them up? Note also that we completely disregarded the first inequality. This is a good question that can be answered in several ways. First, we may use our mathematicians' rights to do whatever we want as far as the reasoning is correct; we do not need to explain how we came up with this reasoning. Second, we can say that there is a special theory, called *linear programming theory*, that recommends this course of action. This sounds intimidating, but in fact one can show how this theory works for our case using Figure 2.11. Do not worry if this is not clear yet: we just wanted to give an example of optimality proof.

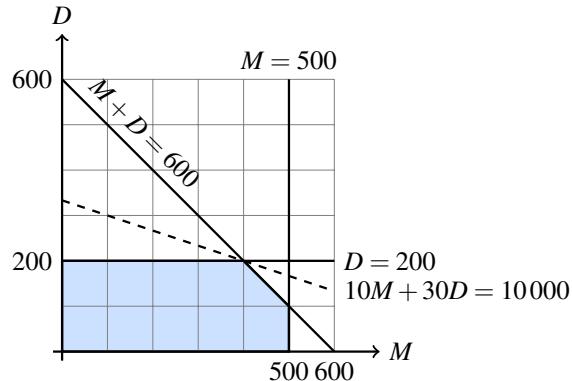


Figure 2.11: We plot all the restrictions on the pair  $(M, D)$ ; the region of possible production plans is highlighted. The dashed line indicates the production plans with a profit of \$10000 per day. We see that there is exactly one production plan that achieves this profit. The same would be true even in the case of unlimited demand for milk chocolate, so only the inequalities 2.2 and 2.3 matter. We combine them with suitable coefficients to get the dotted line: to have  $c_1(M+D) + c_2D = 30D + 10M$ , we use  $c_1 = 10$  and  $c_2 = 20$ .

### Summary

A proof that  $\alpha$  is optimal consists of two parts:

- there exists a solution achieving the value  $\alpha$ ;
- no solutions achieve a value better than  $\alpha$ .

#### 2.2.2 Maximum Number of Two-digit Integers

**Problem 48 – Maximum number of two-digit integers.** There are 90 cards with all two-digit numbers on them:

$$10, 11, 12, \dots, 98, 99.$$

A player can select some of these cards. For each card chosen, she gets \$1. However, if the player takes two cards that add up to 100 (say, 23 and 77), then she loses all the money. How much money can she gain in this game? Try it: [Coursera](#) ↗.

In mathematical language, the problem can be restated as follows: what is the maximum number of elements in a subset of  $\{10, 11, \dots, 99\}$  that does not contain two numbers  $x$  and  $y$  with  $x + y = 100$ ? Even more formal version: find

$$\max\{|S| : S \subseteq \{10, \dots, 99\} \text{ and } \forall x \neq y \in S, (x + y \neq 100)\}.$$

Don't be frightened by the notation: we will learn it later step by step.

**Stop and Think!** Why do we have exactly 90 two-digit numbers?

One could count them by tens: there are ten numbers that start with 1 (i.e., 10, 11, 12, ..., 18, 19), ten numbers starting with 2, ..., ten numbers starting with 9. Another way to count: there are 99 numbers from 1 to 99, we delete 9 numbers from 1 to 9, and we get  $99 - 9 = 90$ . In general, the range  $m \dots n$  (for integer  $m \leq n$ ) contains  $n - (m - 1) = n - m + 1$  numbers. (A common mistake here is to forget the “+1” correction.)

Let us return to the original problem. There are pairs of numbers that sum up to 100, e.g.,

$10 + 90 = 100$ . If the player takes 10, then she cannot take 90, and vice versa. Similarly, 11 and 89 cannot be taken simultaneously. In total, we have 40 such pairs:

$$(10, 90), (11, 89), \dots, (49, 51). \quad (2.4)$$

**Stop and Think!** What two-digit numbers are left without pairs?

The integers 91, 92, ..., 99 do not have pairs. Are there any others? Yes, there is one more number without a pair, 50. In total, we have ten numbers without pairs:

$$50, 91, 92, \dots, 99.$$

Hence, one definitely cannot take more than  $40 + 10 = 50$  numbers (one number from each pair, and all the unpaired numbers).

**Stop and Think!** Can you find a solution with exactly 50 numbers?

There are many way to provide such an example (we are free to choose one of the elements from each pair). Say, we may select the bigger element in each pair. Then we get

$$50, 51, \dots, 99.$$

Let us repeat this argument in a more concise and formal way.

**Theorem 2.2.1** The maximum number of two-digit integers such that no two of them sum up to 100 is 50.

*Proof.* Denote this maximum by  $M$ . Below, we show that  $M \geq 50$  and  $M \leq 50$ . This means that  $M = 50$ .

*Existential part.* There exists such a set of size 50: 50, 51, ..., 99. Indeed, the sum of any two numbers from this set is greater than 100. This shows that  $M \geq 50$ .

*Universal part.* On the other hand, any such set must exclude at least one number from each of the 40 pairs from (2.4). Hence, the size of the set is at most  $90 - 40 = 50$ . This implies that  $M \leq 50$ . ■

### 2.2.3 Rooks on a Chessboard

A chess rook moves vertically and horizontally, see Figure 2.12. Chess players say that the rook *attacks* all cells where it can move, i.e., all cells on the same vertical or horizontal line.

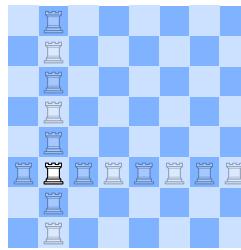


Figure 2.12: A chess rook moves vertically and horizontally.

**Stop and Think!** How many cells does a rook attack (not including its own cell)?

Each vertical (or horizontal) line contains 8 cells, so there are 7 attacked cells (in addition to the rook cell), for a total of 14 cells.

**Problem 49** What is the maximum number of rooks on a chessboard such that no two rooks attack each other? Try it: [Coursera ↗](#).

To answer this problem, let us reformulate the rules: two rooks attack each other if they are in the same row (horizontal line) or column (vertical line).

**Stop and Think!** Do you see why the number of rooks not attacking each other is at most 8?

The reason is obvious: each of 8 columns may contain at most one rook, so there are at most 8 rooks.

**Stop and Think!** In this argument we talked about columns and did not say anything about rows, is it OK?

Yes — it shows that the restriction “no two rook in one column” is enough to prove that there are at most 8 rooks.

**Stop and Think!** Can we say now that the problem is solved and we proved that the maximum number of rooks not attacking each other is 8?

Not yet, but we are close. It remains to show the first (existential) part of the proof: we need to provide an example with 8 rooks not attacking each other. There are many ways to do this, see Figure 2.13.

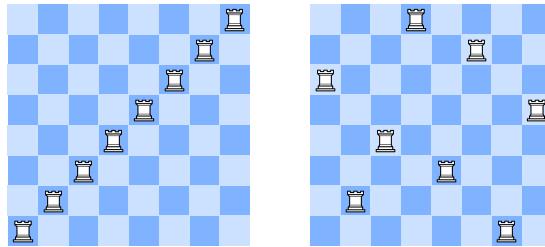


Figure 2.13: Two ways of placing eight rooks that do not attack each other. (Do we really need both of them? Of course not, one would be enough.)

We have seen two possible solutions to the non-attacking rooks problem. There are many others: can you say how many? Later in the book, we will learn a way to compute this number.

**Problem 50** Imagine a “chessboard” with 10 columns and 15 rows. What is the maximum number of rooks not attacking each other on this board?

#### 2.2.4 Knights on a Chessboard

Consider another chess piece: a *knight*. It moves two cells in one direction (vertical or horizontal) and one in the other (perpendicular) direction, see Figure 2.14.

**Stop and Think!** What is the maximum number of cells that a knight may attack?

The maximum number is 8 if the knight is far from the border. Some of the moves become impossible if the knight is close to the border. For example, if we move the knight one cell to the left, two of the moves become impossible and we get 6 attacked cells.

**Problem 51** List all the possible numbers of cells that a knight may attack.

You probably guessed what is our next question.

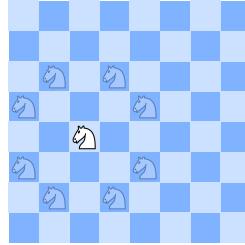


Figure 2.14: A chess knight makes an L-shape move.

**Problem 52** What is the maximum number of knights on a chessboard such that no two of them attack each other? Try it: [Coursera ↗](#).

This is a more difficult problem, and it is instructive to think about it before reading further. But the crucial observation is again related to dark and light coloring of the chessboard: *a knight never attacks the cells of the same color*.

**Stop and Think!** What are the consequences of this observation for our goal (maximum number of knights)?

Let us reformulate this observation: *two knights placed on cells of the same color cannot attack each other*. Therefore — aha! — we can safely fill all light cells (or, if you prefer, all dark cells) with knights (Figure 2.15).

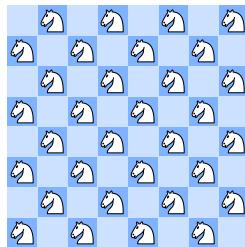


Figure 2.15: Each dark cell has a knight.

**Stop and Think!** How many knights did we place this way?

The number of knights, that is, the number of light cells, is 32 (half of the total number of cells; the board contains equal numbers of light and dark cells). We cannot add any more knights, since every remaining cell (every dark cell) is under attack by one or more knights. Thus, the maximum number of knights is 32 and the problem is solved.

**Stop and Think!** Do you see a mistake in this reasoning?

This mistake is often overlooked by beginners. We proved that we cannot *add* any knights to this configuration, this is OK. But maybe we can start anew and place knights in a different way where not all light cells are used, but more knights are placed due to the knights on dark cells — why not? Imagine, for example, that we have filled the third and sixth columns with knights (Figure 2.16). No more knights can be added — still, as we have seen, 16 is far from being maximum.

Similarly, there is no reason to be sure that 32 is the maximum number of knights. We need to prove that we cannot place more than 32 knights, and we do not have it yet.

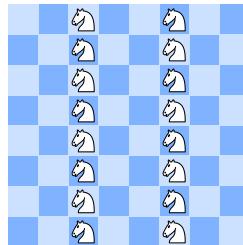


Figure 2.16: In this configuration, we have 16 knights and can't add more since all the cells are under attack. But this does not mean that 16 is the maximum number of knights on the board.

**Stop and Think!** Do you see such a proof? (Recall the problem with integers not summing up to 100: a similar argument may work, too.)

In Problem 48, we grouped the numbers in pairs in such a way that one cannot take both numbers from a pair. We can do a similar pairing of the cells. If two cells differ by a knight move, they both can't be used (Figure 2.17).

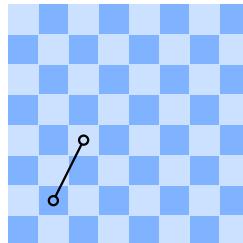


Figure 2.17: A pair of cells that can't be used together.

Thus, if we can group all 64 cells into 32 pairs connected by a knight move, then each pair could contain only one knight, and therefore the number of knights is at most 32.

**Stop and Think!** Can you construct such a pairing?

We can start with a smaller board of size  $2 \times 4$  (smaller than this boards do not work).

**Stop and Think!** Do you see how to group the 8 cells of a  $2 \times 4$  board in 4 pairs in such a way that in each pair the cells are connected by a knight move? Do you see how to use this pairing to construct a pairing for an  $8 \times 8$  board?

The required pairing for a  $2 \times 4$  board is shown in Figure 2.18.

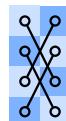


Figure 2.18: A  $2 \times 4$  board: four pairs

We can combine eight small boards to get a big one (Figure 2.19). Since only one cell from each pair can be used, this picture proves that we can't place more than 32 knights, and, thus, finishes the proof.

In this problem, we saw a difference between an optimal configuration with the maximum

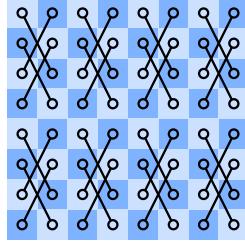


Figure 2.19: Combining eight  $2 \times 4$  boards to get a pairing for  $8 \times 8$  board.

number of knights (e.g., Figure 2.15) and a suboptimal configuration where no knights can be added (e.g., Figure 2.16).

### 2.2.5 Bishops on a Chessboard

A *bishop* attacks in every direction diagonally (Figure 2.20).

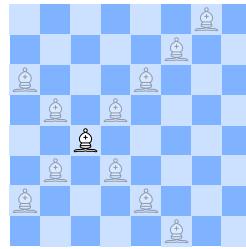


Figure 2.20: A bishop moves diagonally.

**Problem 53** How many bishops not attacking each other can be placed on a chessboard? Try it: Coursera [↗](#).

Now we have enough experience to present a solution to this problem in a rigorous but concise form. Each of the 15 diagonals (including the two trivial “diagonals”, the top left corner and the bottom right corner) shown in Figure 2.21(a) may contain only one bishop. In addition, only one of two trivial diagonals can be used since they are connected by a bishop’s move. Thus, the maximum number of bishops on a board is at most 14. On the other hand, one can place 14 bishops not attacking each other, see Figure 2.21(b).

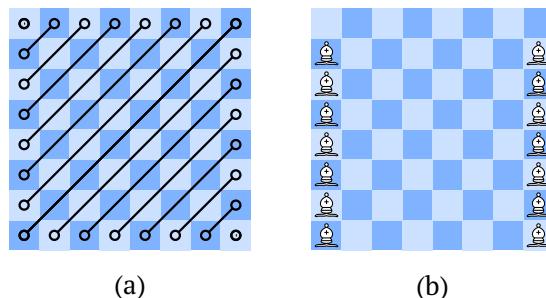


Figure 2.21: A solution to Problem 53.

There are two remaining chess figures, the royal ones. If you know how they move, you can try to find out the maximum number of *queens* that can be placed on a chessboard. Since the rook moves are available to queens, we cannot place more than eight queens; it turns out (see

Section 2.3) that it is indeed possible to put eight queens on a chessboard. Kings are less aggressive than queens (they can move only to one of the 8 neighboring cells), and for them, the problem is easier.

**Problem 54** Prove that one can place 16 kings not attacking each other, but no more.

*Hint:* divide the board into  $2 \times 2$  squares.

There is another type of interesting questions: what is the *minimum* number of kings that suffices to attack all the cells (that are not occupied by kings). It is also an optimization problem (here “optimal” means “minimum” rather than “maximum”). To solve it, you should specify some value  $n$  and

- show an example with  $n$  kings attacking all the remaining cells;
- prove that fewer than  $n$  kings are not enough (i.e., there exists a non-attacked cell without a king in it).

**Problem 55** Prove that the minimum number of kings attacking all cells is  $n = 9$ .

*Hint:* cut the board into  $3 \times 3$  pieces (this can be done for a  $9 \times 9$  board, but for an  $8 \times 8$  board some pieces will have to be smaller). Show that each piece can be “covered” by one king, and that one king should be present in each piece.

In the language of graph theory, this problem is called the *dominating set* problem, whereas the previous problems were examples of the *independent set* problem.

### 2.2.6 Subsets without $x$ and $2x$

In Problem 48, we were asked to select the maximum number of two-digit numbers such that no two of them sum up to 100. Now, let us consider a similar problem but with another restriction.

**Problem 56** What is the maximum number of integers among  $1, 2, \dots, 50$  that one can select, if one is not allowed to select simultaneously  $x$  and  $2x$ ? Try it: [Coursera](#) ↗.

**Stop and Think!** Can we again split the numbers into pairs of mutually incompatible numbers that can't be selected together?

Not exactly. We can't select 1 and 2 simultaneously, so they should form a pair. But we also can't select 2 and 4 simultaneously, so 2 should also be paired with 4. Then 4 should be used together with 8, and 8 should be paired with 16, etc. This way, we get a chain

$$1 - 2 - 4 - 8 - 16 - 32,$$

where the neighbors can't be used simultaneously. (The next number, 64, is outside our range.) In general,  $x$  cannot appear with  $x/2$  or  $2x$  (if present among  $1, 2, \dots, 50$ ).

**Stop and Think!** Can you list other pairs of numbers that can't be selected together?

The answer can be presented graphically (Figure 2.22).

Each node contains a number between 1 and 50. Mutual incompatible numbers are connected by edges. Each chain starts with some odd  $x$  (if  $x$  is even, then it follows  $x/2$  in a different chain) and continues with  $2x$ ,  $4x$ , etc. as long as possible (numbers do not exceed 50). For numbers 27, 29, ..., 49 the chains are trivial (since there is no node with  $x/2$  or  $2x$ ).

**Stop and Think!** How can we now reformulate our question in terms of the graph in Figure 2.22?

The integers  $1, 2, \dots, 50$  are nodes of the graph, and the restriction is that we should not select two nodes connected by an edge.

**Stop and Think!** What is the maximum number of nodes that can be selected according to this rule?

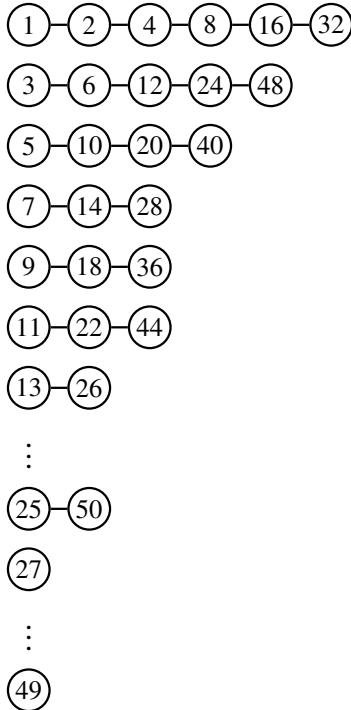


Figure 2.22: Chains of mutually incompatible numbers.

The crucial observation here is that the *choices in different chains are independent* (there are no edges between chains). Hence, we can choose the maximum number of nodes in each chain separately, and then combine our choices! In the first chain, we can choose three nodes (say, 1, 4, 16, or 2, 8, 32, or 1, 8, 32, etc.). In the second chain, we can also select three nodes, but uniquely: 3, 12, 48. Then we have chains with two nodes, starting with 5, 7, 9, 11, and all the remaining chains give only one node. In total, we get the sum  $3 + 3 + 2 + 2 + 2 + 2 + 1 + 1 + \dots + 1$ .

**Stop and Think!** How many terms do we have in this sum? How many of them are equal to 1?

Each chain gives us one term in the sum, so the number of terms is the number of chains. The chains start with all odd numbers from this range: 1, 3, 5, ..., 49. Since each odd number can be paired with the following even number, the number of odd numbers (and the number of chains) is  $50/2 = 25$ . There are 6 terms greater than 1 (recall  $3 + 3 + 2 + 2 + 2 + 2$ ), and, therefore, there are  $25 - 6 = 19$  terms that are equal to 1. Then the total sum is  $3 + 3 + 2 + 2 + 2 + 2 + 2 + 19 = 33$ . Our reasoning shows therefore that the maximum number of integers that can be selected according to the rules, is 33.

**Stop and Think!** Can you write an example of 33 numbers that can be selected according to these rules?

As we have discussed, there are many ways to do this. One of them: take all the numbers in the first column, in the third column and in the fifth column. This gives  $25 + 6 + 2 = 33$  numbers.

**Problem 57** How many different maximum solutions are there (subsets of  $\{1, \dots, 50\}$  of size 33 that do not include  $x$  and  $2x$  at the same time)?

We'll see how to do this using the product rule later in the book. However, if you're curious, the answer is 1 536!

**Problem 58** What is the maximum number of distinct integers from  $1, 2, \dots, n$  that can be selected if we are not allowed to select  $x$  and  $2x$  at the same time? In the previous problems, we had  $n = 50$ .

The answer to this problem is a bit more complicated:

$$\left\lfloor \frac{n+1}{2} \right\rfloor + \left\lfloor \frac{n/4+1}{2} \right\rfloor + \left\lfloor \frac{n/4^2+1}{2} \right\rfloor + \dots,$$

where  $\lfloor u \rfloor$  stands for the integer part of  $u$ , also known as the *floor* function (the maximum integer not exceeding  $u$ ), and the sum is taken until the terms become zeros. For  $n = 50$ , we get  $25 + 6 + 2 + 0 + 0 + \dots = 33$  (the answer that we already know). Try to prove this formula for all values of  $n$ . One way to prove this is to show that the first column has  $\lfloor \frac{n+1}{2} \rfloor$  numbers, the third column has  $\lfloor \frac{n/4+1}{2} \rfloor$  numbers, etc.

## 2.3 Computer Search

This section discusses more advanced techniques for using computers for constructing tricky combinatorial objects. If you have never written recursive programs before, we encourage you to skip this section for now and return here after you learn recursion in Section 3.1.

We have already seen examples of problems where a solution exists, but it is not at all easy to construct it by hand. A natural thing to do in such a case is to ask a machine to do the job. But if the search space is enormous, then one still needs to help the machine to narrow the search space. In this section, we will practice implementing efficient programs for this task. As usual, to introduce the underlying techniques, we use puzzles with simple rules (so that no problem-specific details distract you from understanding the main ideas). Still, the technique that we introduce, *backtracking*, finds applications in a wide range of real-life optimization problems — planning, scheduling, supply chain optimization, and logistics, to name a few.

Our working example for introducing the *backtracking* technique will be the classical  $n$  queens problem. Recall that a *chess queen* moves vertically, horizontally, or diagonally (see Figure 2.23).

**Problem 59 –  $n$  queens.** Is it possible to place  $n$  queens on an  $n \times n$  board such that no two of them attack each other? Try it: [Coursera](#).

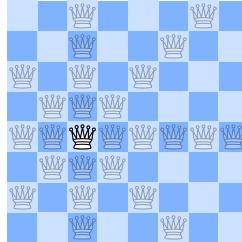


Figure 2.23: Moves of a chess queen.

It is not that difficult to find the answer for  $n \leq 5$  (try it!), but already for  $n = 8$  it is not easy to construct a solution by hand.

It is known that, for any  $n \geq 4$ , the answer is positive: one can place  $n$  non-attacking queens on an  $n \times n$  board for any integer  $n \geq 4$ . Moreover, there is an *explicit* solution for this task: there is a simple formula saying where to place the queens (the fact that the formula is simple means that it is easy to *state it* and does *not* mean that it is easy to *find it*!).

**Problem 60** Assume that  $n \geq 4$  is even and the remainder of  $n$  when divided by 6 is not equal to 2. Prove that the following placement is correct (i.e., no two queens attack each other): for every  $0 \leq i \leq n/2 - 1$ , place two queens on the cells  $(i, 2i+1)$  and  $(n/2+i, 2i)$ . An example of this placement for  $n = 12$  is shown in Figure 2.24.

This means that one does not need a computer to solve this problem. Still, the  $n$  queens problem is a popular benchmark for various combinatorial optimization programs. Implementing such

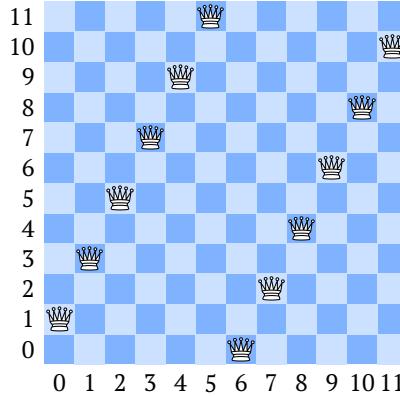


Figure 2.24: A correct placement of 12 queens according to the formula from Problem 60.

a program is also an interesting and instructive problem in algorithms. For this reason, in the rest of this section, we will be solving the following problem.

**Problem 61** Implement a program that, given  $n \geq 4$ , finds a correct placement of  $n$  queens on an  $n \times n$  board.

### 2.3.1 Brute Force

We start by implementing a brute force solution. Specifically, we will make the following two steps.

**Step 1.** Enumerate all possible placements of  $n$  queens where no two queens stay in the same row or column.

**Step 2.** Among all such placements, we select one where no two queens stay on the same diagonal.

**Stop and Think!** Consider a placement of  $n$  queens no two of which stay in the same row or column. How do we represent such a placement in a program?

To represent it, we may use a sequence  $r_0, r_1, \dots, r_{n-1}$  of  $n$  integers:  $r_i$  is the index of the row containing a queen in the  $i$ -th column. That is, a sequence  $r_0, r_1, \dots, r_{n-1}$  defines the following  $n$  cells on the board:

$$(0, r_0), (1, r_1), \dots, (n-1, r_{n-1}).$$

Since we know that no two queens stay in the same row, all  $r_i$ 's must be different. This, in turn, means that  $r_0, r_1, \dots, r_{n-1}$  is nothing else but a *permutation* of the numbers  $0, 1, \dots, n-1$ . In Figure 2.25, we give examples of such permutations.

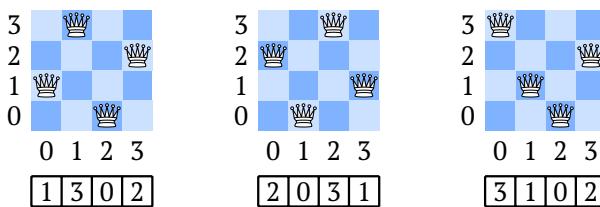


Figure 2.25: A placement of  $n$  queens in different rows and columns can be represented as a permutation.

A permutation is a fundamental object in discrete mathematics. Therefore, it is not at all surprising that Python has built-in methods for enumerating permutations. Using the `permutations()` function from the `itertools` library we implement step 1 in just one line of code!

```

from itertools import permutations

for permutation in permutations(range(4)):
    print(permutation)

(0, 1, 2, 3)
(0, 1, 3, 2)
(0, 2, 1, 3)
(0, 2, 3, 1)
(0, 3, 1, 2)
(0, 3, 2, 1)
(1, 0, 2, 3)
(1, 0, 3, 2)
(1, 2, 0, 3)
(1, 2, 3, 0)
(1, 3, 0, 2)
(1, 3, 2, 0)
(2, 0, 1, 3)
(2, 0, 3, 1)
(2, 1, 0, 3)
(2, 1, 3, 0)
(2, 3, 0, 1)
(2, 3, 1, 0)
(3, 0, 1, 2)
(3, 0, 2, 1)
(3, 1, 0, 2)
(3, 1, 2, 0)
(3, 2, 0, 1)
(3, 2, 1, 0)

```

We proceed to step 2. Looking at Figure 2.25, we see that not every permutation constitutes a solution to the  $n$  queens problem. E.g., the first two permutations of the figure are valid solutions, whereas the last one is not as there are two queens attacking each other.

**Stop and Think!** How do we check whether a permutation contains two queens on the same diagonal?

To do this, we need to check whether two cells  $(i_1, j_1)$  and  $(i_2, j_2)$  lie on the same diagonal. This happens if and only if

$$|i_1 - i_2| = |j_1 - j_2|,$$

i.e., if the horizontal shift is the same as the vertical shift (see Figure 2.26).

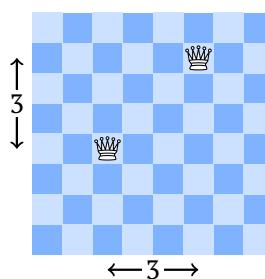


Figure 2.26: Two queens are on the same diagonal.

This observation allows us to implement the following simple procedure for checking whether the given permutation gives a correct solution.

```
from itertools import combinations

def is_solution(perm):
    pairs = combinations(range(len(perm)), 2)
    return all(abs(i1 - i2) != abs(perm[i1] - perm[i2])
              for i1, i2 in pairs)

print(is_solution([1, 3, 0, 2]))
print(is_solution([2, 0, 3, 1]))
print(is_solution([3, 1, 0, 2]))
```

True  
True  
False

In this code, we use another useful function from the `itertools` library: `combinations` allows us to enumerate all pairs  $(i_1, i_2)$  such that  $0 \leq i_1 < i_2 \leq n - 1$ .

Finally, by combining the two steps that we've implemented, we get a program for the  $n$  queens problem (basically, just six lines of code!). In the code, the function `filter` is used to disregard all permutations for which the function `is_solution` returns `False`. The function `next` just returns the first permutation satisfying our property. This first permutation is shown in Figure 2.27.

```
from itertools import combinations, permutations

def is_solution(perm):
    pairs = combinations(range(len(perm)), 2)
    return all(abs(i1 - i2) != abs(perm[i1] - perm[i2])
              for i1, i2 in pairs)

solution = next(filter(is_solution,
                       permutations(range(8))))
print(solution)
```

(0, 4, 7, 5, 2, 6, 1, 3)

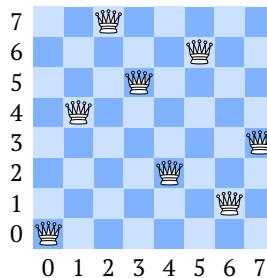


Figure 2.27: A solution to the 8 queens problem found by the program that we've implemented.

The resulting program is simple and short. At the same time, it is slow enough. Already for  $n = 13$  it starts taking noticeable time to complete. The reason is that it is not at all optimized: it just goes through all permutations and finds the first one that gives a correct solution. For this reason, it is called a *brute force* solution. Below, we show how to optimize it so that it works fast enough even for  $n = 20$ .

### 2.3.2 Backtracking

The main idea of the powerful *backtracking* method is the following:

- We construct a solution piece by piece: first, we place a queen in the first column; then, we place a queen in the second column, and so on.
- At each point, we check whether the given *partial solution* is correct or not: say, if the queens in the first two columns already stay on the same diagonal, there is no sense to further extend the current partial solution. In such a case, we *backtrack*.

This is best illustrated by an example. Figure 2.28 shows how one would try to place three queens on a  $3 \times 3$  board. We start with an empty board and try to place a queen in the leftmost column. We first place a queen in the bottom row. This leaves a single non-attacked cell in the middle column (all attacked cells on every board are crossed out). If one places a queen to this cell, then all cells in the rightmost column are attacked. At this point, we realize that we should backtrack and try to place a queen in the leftmost column in some other cell. If we use the middle cell, then all cells in the middle column are attacked, so we discard this partial solution and backtrack again. Finally, we try the top row in the leftmost column. This branch leads to a dead end similar to the one in the leftmost branch of the tree.

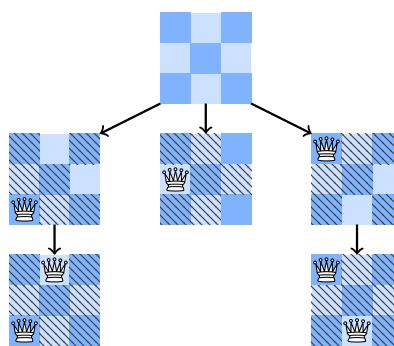


Figure 2.28: A backtracking approach to the 3 queens problem.

To implement the backtracking approach, we need to be able to enumerate partial permutations. This can be done as follows: given a partial permutation, extend it with an element that it does not contain and continue recursively.

```
def extend(perm, n):
    if len(perm) == n:
        print(perm)
        return

    for k in range(n):
        if k not in perm:
            extend(perm + [k], n)

extend(perm=[], n=3)
```

[0, 1, 2]  
[0, 2, 1]  
[1, 0, 2]  
[1, 2, 0]  
[2, 0, 1]  
[2, 1, 0]

As usual, it is instructive to add a few debug printing instructions to better understand the behavior of the code.

```
def extend(perm, n):
    if len(perm) == n:
        print(f'Final permutation: {perm}')
        return

    print(f'Extending partial permutation {perm}...')
    for k in range(n):
        if k not in perm:
            extend(perm + [k], n)

extend(perm=[], n=3)
```

```
Extending partial permutation []
Extending partial permutation [0]...
Extending partial permutation [0, 1]...
Final permutation: [0, 1, 2]
Extending partial permutation [0, 2]...
Final permutation: [0, 2, 1]
Extending partial permutation [1]...
Extending partial permutation [1, 0]...
Final permutation: [1, 0, 2]
Extending partial permutation [1, 2]...
Final permutation: [1, 2, 0]
Extending partial permutation [2]...
Extending partial permutation [2, 0]...
Final permutation: [2, 0, 1]
Extending partial permutation [2, 1]...
Final permutation: [2, 1, 0]
```

OK, we now know how to generate permutations recursively. We do this by starting from an empty permutation and extending it gradually. The next step is to detect partial permutations that cannot be extended to a solution (to the  $n$  queens problem) for sure. To do this, we just check whether the last element of the permutation (i.e., the rightmost queen) conflicts with any of the previous elements (i.e., is attacked by one of the previous queens).

```
def can_be_extended(perm):
    i = len(perm) - 1
    return all(i - j != abs(perm[i] - perm[j])
              for j in range(i))
```

Finally, we have everything to write down our backtracking program! For  $n = 16$ , it finds a solution immediately.

```
def can_be_extended(perm):
    i = len(perm) - 1
    return all(i - j != abs(perm[i] - perm[j])
              for j in range(i))

def extend(perm, n):
    if len(perm) == n:
        print(perm)
        exit()
```

```

for k in range(n):
    if k not in perm:
        if can_be_extended(perm + [k]):
            extend(perm + [k], n)

extend(perm=[], n=16)

```

[0, 2, 4, 1, 12, 8, 13, 11, 14, 5, 15, 6, 3, 10, 7, 9]

In Section 7.2, we show how to implement a program that works in the blink of an eye even for  $n = 100$ .

To practice implementing backtracking solutions, try to solve the following problem (it is not easy to solve it by hand!).

**Problem 62** In a  $5 \times 5$  grid, draw 16 diagonals that do not touch each other. (E.g., in  $3 \times 3$  and  $4 \times 4$  grids one can place 6 and 10 diagonals, respectively. See Figure 2.29.) Try it: [Coursera](#) ↗.

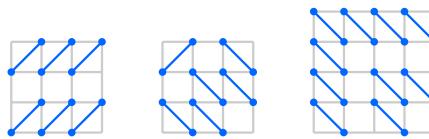


Figure 2.29: Placing many non-intersecting diagonals.

A backtracking approach for this problem may proceed as follows:

- Fill in the grid gradually (say, from bottom to top, from left to right).
- For each cell consider three possibilities:
  1. Diagonal from bottom left corner to top right corner.
  2. Diagonal from bottom right corner to top left corner.
  3. No diagonal.
- Each time when a new diagonal is placed, check whether it conflicts with other diagonals. If it does, backtrack.

We encourage you to post your solution and to check solutions by other learners at the [forum thread](#) ↗.

**Problem 63** Prove that one cannot place more than 16 non-intersecting diagonals on a  $5 \times 5$  grid.

This is another challenging problem. To see that 16 is optimal, note that a  $5 \times 5$  grid contains 36 nodes and each diagonal must use two of them. This already means that it is impossible to place more than 18 diagonals. To see that even 17 diagonals is impossible, see Figure 2.30. It focuses our attention on two layers of nodes: the external one (shown in blue) and the layer next to it (shown in black). If we manage to place 17 diagonals, then only two nodes will be unused.

Every blue node can be joined by a diagonal with either another blue node or a black node. At the same time, if we join two blue nodes (this is only possible for two blue nodes that are close to the corner), we block the corner blue node. Thus, even if we join every black node with some blue node, we will still have eight remaining blue nodes that can be joined only to each other. To leave only two of them unused, we need to join at least three pairs of them, but (as discussed above) this makes three blue nodes unused.

When you implement a backtracking program for the  $5 \times 5$  grid and make sure that it finds an answer, you might want to check the behavior of your program on  $n \times n$  grids for  $n > 5$  (and

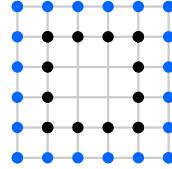


Figure 2.30: Two layers of nodes in the  $5 \times 5$  grid.

even for  $n \times m$  grids). Interestingly, already for  $n = 27$  the exact value of the optimum number of diagonals for the  $n \times n$  grid is not currently known! See the [corresponding sequence](#) in the On-line Encyclopedia of Integer Sequences.

Another way to generalize the diagonals problem is to consider diagonals of length  $\sqrt{5}$  in  $1 \times 2$  rectangles. For example, one can place 20 such diagonals on a  $6 \times 6$  grid, see Figure 2.31.

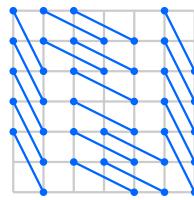


Figure 2.31: 20 non-intersecting  $1 \times 2$  diagonals on a  $6 \times 6$  grid.

**Problem 64** Prove that it is possible to place 21 non-intersecting diagonals of length  $\sqrt{5}$  in a  $6 \times 6$  grid.

In Section 7.3, we state the diagonals problems as Integer Linear Programming (ILP) problems, and use ILP-solvers to solve them efficiently even when  $n$  is as large as 15.

### Summary

- In some situations, an object exists but it is not easy to construct it by hand. In this case, one may want to use a computer to find such an object.
- If the search space is small enough, a naive brute force search might help to find the required object. However, in many cases the search space is enormous and one should use additional techniques to narrow the search space.
- One such technique is known as backtracking. Its main idea is to construct the required object step by step and to get a step back as soon as a conflict is detected.

## 3. Recursion and Induction

We will discover two powerful methods of defining objects, proving concepts, and implementing programs — *recursion* and *induction*. These two methods are regularly used in discrete mathematics and computer science. You will see them frequently in algorithms — for analyzing the correctness and running time of algorithms as well as for implementing efficient solutions. For some computational problems (for example, exploring networks), recursive solutions are the most natural ones.

The main idea of recursion and induction is to decompose a given problem into smaller problems of the same type. Recursion is often used for computing something whereas induction is used for proving things. Being able to see such decompositions into smaller steps is an important skill both in mathematics and programming. We will hone this skill by solving various problems.

### 3.1 Recursion

#### 3.1.1 Counting People

You are standing in a long queue waiting for the opening of a museum. How can you find the number of people before you? You are not allowed to get out of the queue and count. What can you do instead? This is easy: *just ask the person before you the same question* (how many people are before her), *and then add one to her answer*.

This sounds like a joke: we reduced the problem to the same problem for another person. However, imagine that the person before you will do the same and ask her predecessor, who will do the same, etc. Then a wave of questions will go to the start of the queue, and when it reaches the first person in the queue, she will say “zero”. When the reflected wave of answers comes back, you know the answer. See Figure 3.1 for example.

In programming, this is called “*recursion*”. Specifically, a program is called *recursive* if it calls itself, thus reducing an instance of a problem to some other instance of the same problem.

The following example illustrates the same idea by a Python program that counts the number of elements in a list. This is overkill of course, since you can just use the built-in function `len()` for that, but this is still an instructive example of recursion.

```
def length(lst):
    if not lst:
        return 0
```

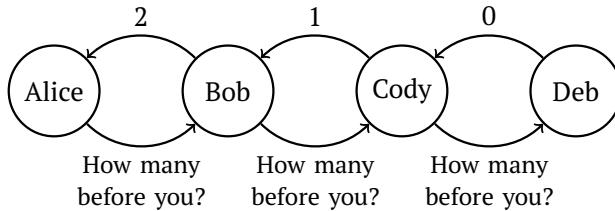


Figure 3.1: In order to find the number of people in the line before her, Alice asks Bob, “How many people are ahead of you?”. In turn, Bob asks Cody the same question, and Cody asks Deb. There are no more people before Deb and Deb tells this to Cody. Cody realizes that there is exactly one person before him in the line and says this to Bob. Finally, Bob tells Alice that there are two people before him and Alice concludes that she is the fourth in the line.

```

else:
    return 1 + length(lst[1:])

print(length([5, 3, 2, 1, 7]))

```

5

Here, `[5, 3, 2, 1, 7]` is a *list*, that is, a sequence of objects (shown in square brackets separated by commas). To find the length (the number of elements) in the list `lst`, we first check whether `lst` is *empty*. This is done by checking the [truth value](#) of `lst`: in Python, any empty collection is considered false. (One can also use condition `lst == []` instead.) If `lst` is empty, we return zero immediately. Otherwise, we delete the first element of `lst` by using slicing (every sequence in Python is 0-based; hence the slice `[1:]` just takes a subsequence starting with element number one, that is, the whole subsequence without the first element). Then, we *recursively* call the same function `length()` on this shorter list, and add one to its answer.

If you’ve never seen recursive programs before, this type of function call may confuse you. To better understand what is going on inside, it’s always good to add a few *debug printing* instructions. This debug output shows the forward and backward waves that we discussed above!

```

def length(lst):
    print(f'Computing the length of {lst}...')
    if not lst:
        print(f'The length of {lst} is 0')
        return 0
    else:
        shorter_lst = lst[1:]
        result = 1 + length(shorter_lst)
        print(f'The length of {lst} is {result}')
        return result

```

```
length([5, 3, 2, 1, 7])
```

```

Computing the length of [5, 3, 2, 1, 7]...
Computing the length of [3, 2, 1, 7]...
Computing the length of [2, 1, 7]...
Computing the length of [1, 7]...
Computing the length of [7]...
Computing the length of []...

```

```
The length of [] is 0
The length of [7] is 1
The length of [1, 7] is 2
The length of [2, 1, 7] is 3
The length of [3, 2, 1, 7] is 4
The length of [5, 3, 2, 1, 7] is 5
```

Here, the print command with [formatted string literals](#) is used; expressions within {} are printed in specified surroundings.

Finally, we also note that the same recursive function can be implemented as a one-liner.

```
def length(lst):
    return 1 + length(lst[1:]) if lst else 0

print(length(['Alice', 'Bob', 'Charlie']))
```

3

**Stop and Think!** Professor X suggests another solution to the queue problem: ask the person *behind* you and subtract 1. Will this approach work?

Of course not. The wave of questions will reach the last person in the queue who has nobody to ask. And if you imagine an infinite queue, the wave goes to infinity and never returns. When writing recursive programs, one should be careful to avoid this kind of situation. We discuss this issue below (Section 3.1.3)

### 3.1.2 Factorial: Iterations Versus Recursion

Now, we consider a classical example: computation of the factorial ( $n!$ ). For an integer  $n \geq 1$ , the factorial  $n!$  is defined as the product of the numbers  $1, 2, \dots, n$ . For example,  $2! = 1 \cdot 2 = 2$ ,  $3! = 1 \cdot 2 \cdot 3 = 6$ ,  $4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$ .

**Stop and Think!** What is the last digit of  $10! = 1 \cdot 2 \cdot 3 \cdots 9 \cdot 10$ ?

One can easily compute  $10!$  with a calculator. But it is easy to find the answer without using a computer (or even a pen and paper). Note that the last operation is a multiplication by 10. Therefore the product ends with 0 (regardless of what the previous operations were).

**Stop and Think!** What is the last digit of  $99! = 1 \cdot 2 \cdot 3 \cdots 98 \cdot 99$ ?

Again, we do not need to compute anything: this product includes a factor 10 and therefore ends with 0 (whatever the other factors are).

**Problem 65** What is the last *nonzero* digit of  $20!$  and how many zeros follow it? (No computer is needed to answer this question.)

Now we switch to a different problem: given a positive integer  $n$ , compute  $n!$ . It is easy to write a program that computes factorials:

```
def factorial(n):
    assert n > 0
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
```

```
print(factorial(10))
```

```
3628800
```

In this program, we first define a function `factorial(n)` that should compute (and return) the factorial of a positive integer  $n$ , its argument. Then we ask to print  $10!$ . How does the function `factorial` work? It checks first that the input is positive: `assert n > 0` will complain (and stop the program) if the assertion  $n > 0$  is false. Then it multiplies the result (initially 1) by the numbers  $2, 3, \dots, n$ . It is done as follows: `range(1, n + 1)` is (according to Python conventions) the list  $[1, 2, \dots, n]$ . The line `result *= i` (equivalent to: `result = result * i`) multiplies the result by  $i$ . After we do this for  $i=1, 2, \dots, n$ , the result is equal to  $n!$ .

**Stop and Think!** What is the behavior of the program for  $n = 1$ ?

If  $n = 1$ , the list `range(1, n + 1)` is `range(1, 2)`, that is,  $[1]$ . Hence, we multiply 1 by 1 and get 1. Indeed,  $1!$  is considered to be 1 (the product  $1 \cdot 2 \dots n$  contains only one factor 1 for  $n = 1$ ).

Factorials appear naturally when we count permutations. For example, three letters  $A, B, C$  can be arranged in 6 possible orderings:  $ABC, ACB, BAC, BCA, CAB, CBA$ . This number 6 equals  $3! = 3 \cdot 2 \cdot 1$ , and this is not a coincidence: four letters  $A, B, C, D$  can be arranged in  $24 = 4!$  possible orderings. Indeed, any of the four letters  $A, B, C, D$  can come first, so all sequences are partitioned into four groups by their first letter. After that first letter is fixed all that remains is to order the remaining letters in  $3!$  factorial ( $3! = 3 \cdot 2 \cdot 1 = 6$ ) possible ways. Each group consists of 6 orderings which share the same primary letter. We can describe this mathematically with the following equation,  $4! = 4 \cdot 3! = 24$ . We will discuss permutations in more detail later.

Why are we discussing the factorial function? Because this is a problem that lends itself perfectly to being computed (or defined) with recursion.

**Stop and Think!** If you know  $(n - 1)!$ , what is a simple way to compute  $n!$  using your knowledge?

Of course,  $n!$  has an additional factor  $n$  compared to  $(n - 1)!$ , so it is enough to multiply by  $n$ :

$$n! = (n - 1)! \cdot n. \quad (3.1)$$

Hence, if you want to compute  $n!$ , you may ask somebody to compute  $(n - 1)!$ , and then multiply the result by  $n$ . This is what the Python interpreter does when you run it on the following program:

```
def factorial(n):
    assert n > 0
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(10))
```

```
3628800
```

The program checks that  $n$  is positive. Then, it considers the special case when  $n = 1$ ; we are at our simplest case, and return 1. For  $n > 1$ , we return  $n * factorial(n - 1)$  according to the formula 3.1. Note that to compute `factorial(n - 1)`, the Python interpreter needs to follow the same procedure. Returning to our queue metaphor, we may say that we have a line of Python interpreters who ask each other about the factorials of  $n, n - 1, n - 2$  etc. until the last one computes  $1! = 1$  and then this answer propagates back, being multiplied by  $2, 3, \dots, n$  on the way.

**Stop and Think!** We have defined  $n!$  for  $n \geq 1$ . How should we define  $0!$  to keep 3.1 valid?

For  $n = 1$ , the equation (3.1) gives  $1! = 0! \cdot 1$ . We already agreed that  $1! = 1$ , so we let  $0! = 1$ , too.

**Stop and Think!** What changes are needed in the programs above (the non-recursive and recursive ones) to let them compute  $\text{factorial}(0)=1$ ?

First, we should replace  $n > 0$  by  $n \geq 0$  in the assertion. For the non-recursive program, this is enough: for  $n = 0$ , the list  $\text{range}(1, n + 1) = \text{range}(1, 1)$  is empty, no multiplication is performed, and  $\text{result}=1$  is returned. For the recursive program, we should start the recursion with `if n == 0` instead of `n == 1` (and return the same value 1).

We conclude with two compact versions of iterative and recursive programs for computing the factorial function.

```
from numpy import prod

def factorial(n):
    assert n >= 0
    return prod(range(1, n + 1))

def factorial(n):
    assert n >= 0
    return n * factorial(n - 1) if n else 1
```

### 3.1.3 Recurse with Care

Recursive programs are often easy to read and write. Still, one should be careful. In this section, we review some of the common pitfalls.

First, consider the following equation for factorial

$$n! = \frac{(n+1)!}{(n+1)}. \quad (3.2)$$

**Stop and Think!** Is this equation correct? Can we write a recursive program based on it?

While the equation is correct (for integer  $n \geq 0$ ), it is a bad idea to write a recursive program based on it. In order to compute  $\text{factorial}(10)$ , we need to compute  $\text{factorial}(11)$ , for that we need to compute  $\text{factorial}(12)$ , and we never come to anything known. Trying to run this program, we get an error message.

```
def factorial(n):
    assert n >= 0
    return factorial(n + 1) / (n + 1)

print(factorial(10))

Traceback (most recent call last):
  File "./fact-bad-recurse.py", line 6, in <module>
    print(factorial(10))
  File "./fact-bad-recurse.py", line 3, in factorial
```

```

        return factorial(n + 1) / (n + 1)
File "./fact-bad-recurse.py", line 3, in factorial
    return factorial(n + 1) / (n + 1)
File "./fact-bad-recurse.py", line 3, in factorial
    return factorial(n + 1) / (n + 1)
[Previous line repeated 995 more times]
File "./fact-bad-recurse.py", line 2, in factorial
    assert n >= 0
RecursionError: maximum recursion depth exceeded
in comparison

```

Here, the Python interpreter tells us that there are too many recursive calls in the chain — more than it can handle — and lists some part of this chain of calls. Not a surprise: what else could we expect?

A lesson that we've learned from this example: when using recursion, we should always prove that the chain of recursive calls is finite. This means that at some stage the answer is obtained directly, without calling the recursively defined function one more time. For the recursive factorial program, this was easy to see: since we start with an integer  $n \geq 1$  and then decrease it at every call, at some point we get 1 and no more recursive calls are needed.

This example resembles the weird way to find the length of the queue that we discussed above: ask the person behind you (rather than in front) and subtract 1. Such a recursive approach does not work as it never reaches a base case.

**Problem 66** For what integers  $n$  does the following program terminate?

```

def strange(n):
    if n == 10:
        return 9
    return n * strange(n + 1)

```

It immediately terminates for  $n = 10$ . This implies that it terminates also for  $n = 9$ , since the computation for  $n = 9$  calls  $\text{strange}(10)$ . Then, we see that it terminates for  $n = 8$ , then for  $n = 7$ , and so on. Hence, for every integer  $n \leq 10$ , the program terminates. (Negative values are also OK.) But  $\text{strange}(11)$  will call  $\text{strange}(12)$  that will call  $\text{strange}(13)$ , and so on. Thus, for  $n = 11$  the program never terminates (as well as for 12, 13, ...). You may try to run the program to see what kind of error it will produce.

**Problem 67** What do  $\text{strange}(1)$  and  $\text{strange}(0)$  return?

Sometimes it is difficult to guarantee that a program terminates. For example, nobody knows whether the following recursive program terminates for all integer  $n \geq 1$ , though the famous [Collatz conjecture](#) says that it does.

```

def collatz(n):
    assert n >= 1
    if n == 1:
        return 0
    elif n % 2 == 0:
        return collatz(n // 2)
    else:
        return collatz(3 * n + 1)

```

(Here, the construction `if...elif...else...` tells Python what to do in the three corresponding cases, and the `//` operator is used for integer division.)

You may argue at this point that we intentionally implemented a weird program `factorial(n)`

that does not terminate and produces an error. Still, it is easy to get the same effect inadvertently: it is enough to forget the base case and write something like the code shown below.

```
def factorial(n):
    return n * factorial(n - 1)
```

And it is indeed a common mistake in programming. In theory, this program makes an infinite number of recursive calls. In practice, Python will generate an error message like the one above. (Note that assert statements help to catch such bugs at early stages.)

Another sure way to get a non-terminating program is to call the function with the same argument, as in the following code.

```
def infinite(n):
    if n == 1:
        return 0
    return 1 + infinite(n)
```

**Stop and Think!** For what values of  $n$  does this function terminate (and return 0)?

It is easy to see that the program terminates for  $n = 1$ ; for any other  $n$ , we get an infinite chain of identical calls and the Python interpreter (sooner or later) complains.

**Problem 68** What is the value returned by the following program for an integer  $n \geq 0$ ?

```
def foo(n):
    assert n >= 0
    if n == 0:
        return 0
    else:
        return foo(n - 1) + 2
```

**Problem 69** Now consider the same question for the following program.

```
def foo(n):
    assert n >= 0
    if n == 0:
        return 0
    else:
        return foo(n - 1) + 2 * n - 1
```

There are applications of recursion not only in mathematical and programming worlds but also in real life. The “chicken-and-egg problem” is usually stated as follows:

which came first: the chicken or the egg?

To get an egg, you need a chicken, and to get a chicken, you need an egg. If you implement a function `chicken()` that always calls `egg()` and vice versa, none of the calls would terminate for obvious reasons.

If a decompression software (line `unzip` program) is distributed in a compressed form, to get it running we first need to get it running. This is a real story: the documentation even acknowledged this problem by saying “This is a bit of chicken and egg problem”.

If you have a movable web camera, you can point it at the screen and see a chain of images: a screen, an image of a screen on the screen, etc. Something similar happens when sharing screens in teleconferencing software. It is often possible to see the delay: a change in the picture visibly

propagates through the chain of images. You could use double mirrors to create the [same ↗ effect](#) – but the speed of light is too great for the human eye to perceive this *physical* recursion.

An infinite loop appears in the advice “To understand recursion, you must first understand recursion”. The Google search engine also makes fun of this problem (Figure 3.2): if you click on the “Did you mean” suggestion, you will get exactly the same page.

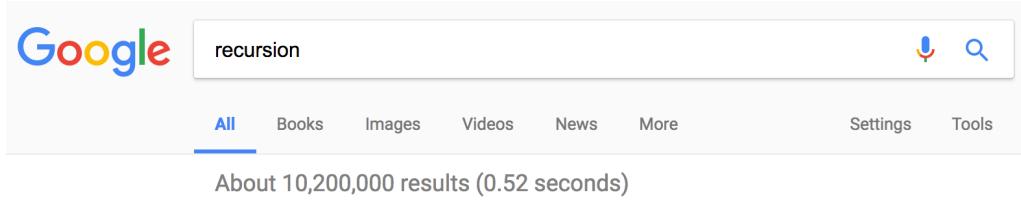


Figure 3.2: Google and recursion.

There is a different (and more esoteric) type of recursion: a program that does not *call* itself but *reproduces* itself (prints its own text). There is an important tool in computability theory, the *recursion theorem*; it guarantees that every programming language has such a program. A genetic code is similar: it contains information on how to build a cell that contains the same genetic code.

### 3.1.4 Coins

**Problem 70** Prove that any monetary amount starting from 8 can be paid using coins in denominations of 3 and 5.

This is a mathematical statement; the corresponding programming problem would be: Given an integer  $n \geq 8$ , find a list of 3s and 5s whose sum is  $n$ .

As usual, let us start by looking at some simple cases.

$$\begin{aligned} 8 &= 3 + 5, \\ 9 &= 3 + 3 + 3, \\ 10 &= 5 + 5, \\ 11 &= 5 + 3 + 3. \end{aligned}$$

These examples do not show us an obvious pattern, so it is hard to derive a rule of thumb with what we have found. Can we move on from 8, 9, 10, and 11 to a sum of 12 with the restrictions we have in place?

**Stop and Think!** Assume that we know how to pay  $n$ . How can we pay  $n + 3$ ?

The question looks obvious: of course, we should pay one more coin of value 3 (add  $+3$  to the sum).

**Stop and Think!** Do you see how this remark can be used?

We know how to pay  $9 = 3 + 3 + 3$ ; adding one more 3, we find how to pay  $12 = 9 + 3 = 3 + 3 + 3 + 3$ , so we can continue our table by one more line. Knowing how to pay  $10 = 5 + 5$ , we find how to pay  $13 = 10 + 3 = 5 + 5 + 3$ ; knowing how to pay  $11 = 5 + 3 + 3$ , we know how to pay  $14 = 11 + 3 = 5 + 3 + 3 + 3$ . And we can continue further: knowing how to pay 12 (see above), we can pay  $12 + 3 = 15$ , and so on. This will cover all integers starting from 8, so we have proven what we wanted.

**Problem 71** How can we convert this proof into a program that takes as input some  $n \geq 8$  and prints the corresponding sum of 3 and 5 summands?

The program follows our previous argument “in the backward direction”, reducing the problem for a given  $n$  to the same problem for  $n - 3$ .

Following programming best practices, we choose a meaningful parameter name: `amount`. First, we check that the `amount` is at least 8. Then, we consider three special cases (8, 9, and 10) where the answer can be given immediately. This answer is a list of integers (of length 2 or 3, depending on the case). If these lines are not applicable, then `amount ≥ 11`, and `amount - 3 ≥ 8`. Then, we use the same program recursively and get a list of integers 3 and 5 with the sum `amount - 3`. It remains to append the integer 3 to the list `coins`. This is done (surprise!) by the line `coins.append(3)`, and the resulting list is returned.

Now, we test this program for some specific amount, say, for 200. (The answer is long and was artificially split into few lines while typesetting the book; the computer output was one line.) We could just say `print(coins)`, but this will give the output in the form `[3, 5, 3, 3, 3, ..., 3]`. We want to get a better-looking output with plus signs (just for fun), and it is done by some Python tricks. The function `str` converts an integer to its decimal representation (the integer 3 is not the same as the string 3 in Python), and `map(str, coins)` applies `str` to every element of the list `coins`, so we get a list of strings. Now, we need to convert this list to one long string that puts + between the integers. For that, we use the `join` function for the string + with the list of strings as its argument. This way, we get the string we want (the sum expression as a string).

Below, we also show a more compact solution implementing the same recursive idea.

```
base_cases = {8: [3, 5], 9: [3, 3, 3], 10: [5, 5]}

def change(amount):
    assert amount >= 8
    if amount <= 10:
        return base_cases[amount]
    return change(amount - 3) + [3]
```

**Stop and Think!** Your friend does not like recursion and says : “Look, I can write an easier and shorter non-recursive program, it just pays 5 until the rest is payable by 3-coins, that is, the rest is a multiple of 3. Isn’t it shorter and better?” What will you say?

```
def change(amount):
    assert amount >= 8
    coins = []
    while amount % 3 != 0:
        coins.append(5)
        amount -= 5
    while amount != 0:
        coins.append(3)
        amount -= 3
    return coins

print(change(13))
```

```
[5, 5, 3]
```

Indeed, this program should work correctly for all payable amounts: if we can pay the required amount by 3- and 5-coins, it means exactly that after paying several 5-coins the rest should be payable by 3-coins. Thus, this program works correctly. But to see this, *we need a separate proof that every amount starting from 8 is payable*, while our previous program implicitly provided such a proof by itself. Moreover, being able to decompose a problem into smaller problems of the same type and to implement recursive solutions is an important skill in programming.

**Stop and Think!** Is the solution of our problem unique? Can we have two different ways to pay some amount?

If we look at our examples, we see that there is no other way<sup>1</sup> to pay 8, 9, 10, or 11. But this is not always the case: for example, we can pay 30 as 10 coins of value 3 or as 6 coins of value 5.

**Stop and Think!** Which way of paying 30 will be given by our program?

The call `change(30)` will call recursively `change(27)` and then add 3; the call `change(27)` will call `change(24)` and add 3, etc. Finally, we will come (check this yourself!) to `change(9)=[3,3,3]`. Thus, we will not use 5-coins at all.

**Problem 72** What is the smallest amount that can be paid in two different ways?

**Problem 73** As we have seen in our example, our program for large inputs uses mostly 3-coins. Say we want to use more 5-coins and fewer 3-coins when there is a choice. What changes should be implemented in the program?

**Problem 74** Is it true that our program always gives an *optimal* solution in the sense that it uses as few coins as possible? Can you give a counterexample or a proof that it is always the case?

**Stop and Think!** Why do we start our problem with  $n = 8$ ?

For the simple reason that 7 cannot be paid.

---

<sup>1</sup>In fact, we should be more accurate here and say explicitly that we do not take into account the order of elements, and consider  $11 = 5 + 3 + 3$  and  $11 = 3 + 5 + 3$  as the same answer. Only the numbers of 3-coins and 5-coins matter, not their ordering.

**Stop and Think!** Which of the amounts 0, 1, 2, 3, 4, 5, 6 can be paid?

Clearly, 0, 3, and 5 can be paid with zero or one coin, and 6 can be paid by two 3-coins. It is easy to see that other amount from 0 to 6 cannot be paid.

We considered coins of value 3 and 5, but the same question could be asked for different sets of values. For example, we may consider coins of values 5 and 7. It turns out that the situation here is similar: there are only finitely many positive integer amounts that cannot be paid (including, 1, 2, 3, 4, 6, 8, 9, and some others), and starting from some point, any amount is payable. You can try to investigate this situation solving the following problems.

**Problem 75** What is the maximum integer amount that *cannot* be paid with 5- and 7-coins?

Try it: [Coursera](#).

**Problem 76** Write a function `change(amount)` that for any integer `amount` starting from 24 returns a list consisting of numbers 5 and 7 only, such that their sum is equal to `amount`. For example, `change(28)` may return `[7, 7, 7, 7]`, while `change(49)` may return `[7, 7, 7, 7, 7, 7]` or `[5, 5, 5, 5, 5, 7, 7]` or `[7, 5, 5, 5, 5, 5, 5, 7]`. Try it: [Coursera](#).

### 3.1.5 Hanoi Towers

In this section, we consider the classical Hanoi Towers puzzle (though it has [nothing to do with Hanoi](#)). There is a tower made of  $n$  disks of decreasing sizes put on a rod, and two other rods (see Figure 3.3). The goal is to move the entire tower to a different rod if you are allowed to move only one disk at a time (this disk should be the upper one on some rod), and a larger disk should never be placed on top of a smaller one.

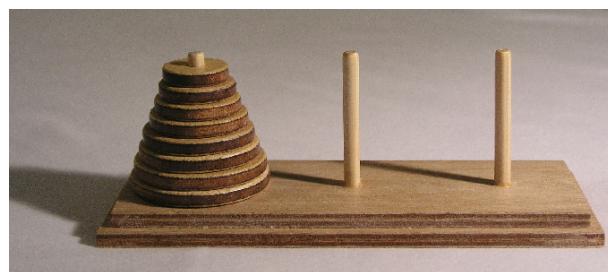
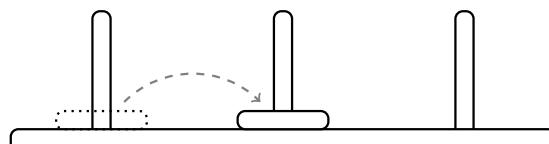


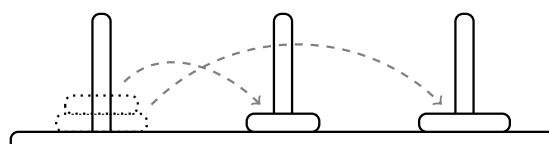
Figure 3.3: The Hanoi Towers puzzle. (Source: [Wikipedia](#)).

**Stop and Think!** Can you solve this puzzle for  $n = 1, 2, 3$ ? Try it: [Coursera](#).

For  $n = 1$ , this is easy: we move the (only) disk to another rod. (Any disk could be placed on an empty rod according to the rules).



For  $n = 2$ , the first move is essentially unique: we can move only the small disk to one of the empty rods; by that we “free” the large disk and can move it to another rod.

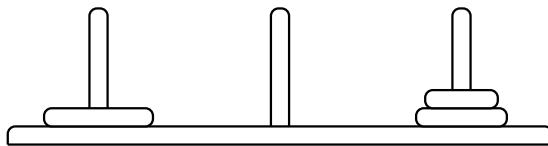


It remains to move the small disk on top of the large disk to get the solution for  $n = 2$ .

For  $n = 3$ , some planning is useful. At some point we need to move the largest disk.

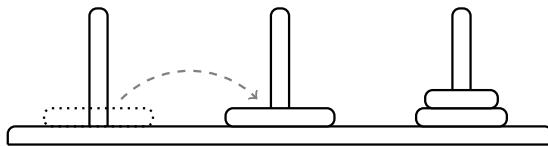
**Stop and Think!** When is it possible to move the largest disk? How should we prepare this move?

First, the largest disk should be free to move, that is, it should be the only disk on the first rod. Second, we need an empty rod to place the largest disk. Assume, for example, that this is the middle rod. Therefore, the two smaller disks should be placed on the third rod (in a correct order). Hence, we need to achieve the following configuration.



For that, we need to move two smaller disks from the first rod to the third one. And — aha! — this is exactly what we did for  $n = 2$ : when we move smaller disks, the largest one is not an obstacle (any disk can be placed on top of it).

After these preparations, we move the largest disk to the middle rod.



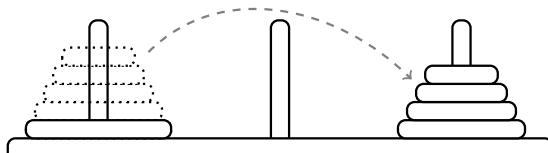
**Stop and Think!** Do you see how to complete the argument?

It remains to move the two smaller disks on top of the largest one, so again we come to the  $n = 2$  case, and we already know how to deal with it.

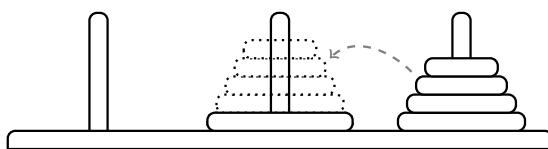
Now, we are ready to consider the general case.

**Problem 77** Prove that for every integer  $n \geq 1$  the Hanoi towers puzzle with  $n$  disks can be solved.

In fact, we have already discussed the key idea for  $n = 3$ : at some point we have to move the largest disk, so we need to prepare this move. To move the largest disk, we need to free it (leave it as the only disk on the rod). Also, we can move the largest disk only if the destination is an empty rod. These two observations imply that  $n - 1$  other disks should be on the third rod, and the position just before the move of the largest disk should be as follows:



In this picture the dotted arrow indicates the effect of the preparation moves: all the disks except for the largest one should be moved to the third rod. Then, the largest disk is moved to the second rod, and it remains to bring the other disks there:



The dotted arrow indicates what should be done before and after the move of the largest disk. Fortunately, we already know (recursion!) that this is possible and recursively call the solution for  $n - 1$  disks. Problem 77 is solved. We can now implement it in Python.

```
def hanoi_towers(n, from_rod, to_rod):
    if n == 1:
        print(f'Move disk from {from_rod} to {to_rod}')
    else:
        unused_rod = 6 - from_rod - to_rod
        hanoi_towers(n - 1, from_rod, unused_rod)
        print(f'Move disk from {from_rod} to {to_rod}')
        hanoi_towers(n - 1, unused_rod, to_rod)

hanoi_towers(3, 1, 2)
```

Move disk from 1 to 2  
 Move disk from 1 to 3  
 Move disk from 2 to 3  
 Move disk from 1 to 2  
 Move disk from 3 to 1  
 Move disk from 3 to 2  
 Move disk from 1 to 2

The function `hanoi_towers` takes three arguments: the number  $n$  of disks and two integers (from the range  $[1, 3]$ ) `from_rod` and `to_rod` specifying from which rod to which rod one should move the disks. The base case is  $n = 1$ : we then just move the only disk. Otherwise, we first move  $n - 1$  top disks to the unused rod, then move the largest disk, and finally move  $n - 1$  disks on top of the largest disk. (The sum of the numbers of three different rods is equal to  $1 + 2 + 3 = 6$ . Hence, to get the number of the unused rod, one needs to subtract (`from_rod + to_rod`) from 6.)

**Stop and Think!** We have seen the solution for  $n = 1$  (one move) and for  $n = 2$  (three moves). How many moves do we make for  $n = 3$ ,  $n = 4$ , and  $n = 5$ ? Can you suggest a general formula for arbitrary  $n$ ? Try it: [Coursera](#).

For  $n = 3$ , we moved the largest disk (one move) and made some moves before that (preparations) and after that. In both cases, we essentially solved the problem for  $n = 2$ . So, if we denote the number of moves for three disks as  $T_3$  and for two disks as  $T_2$ , we get  $T_3 = T_2 + 1 + T_2$ , or  $T_3 = 2T_2 + 1 = 2 \cdot 3 + 1 = 7$ . In general, we have  $T_n = 2T_{n-1} + 1$ , where  $T_n$  stands for the number of moves used by our procedure. So  $T_4 = 2 \cdot 7 + 1 = 15$ ,  $T_5 = 2 \cdot 15 + 1 = 31$ ,  $T_6 = 2 \cdot 31 + 1 = 63$ ,  $T_7 = 2 \cdot 63 + 1 = 127$ ,  $T_8 = 2 \cdot 127 + 1 = 255$ , etc.

This way, we may compute all  $T_n$ , but probably you have already noticed something.

**Stop and Think!** Do the values of  $T_n$  (or, maybe,  $T_n + 1$ ) look familiar?

You see that  $T_n$  are powers of two minus one:  $T_n = 2^n - 1$ . This happens at least for the values  $n \leq 8$ , and it looks plausible that this is a general rule.

**Stop and Think!** Can you prove that  $T_n = 2^n - 1$  for all  $n \geq 1$ ? In other words, can you prove that  $T_n + 1 = 2^n$ ?

Let us look at the *recurrent* formula  $T_n = 2T_{n-1} + 1$ . Let us rewrite it for  $T_n + 1$ :

$$T_n + 1 = (2T_{n-1} + 1) + 1 = 2T_{n-1} + 2 = 2(T_{n-1} + 1).$$

This means that  $T_n + 1$  increases by a factor of 2 as  $n$  increases by 1, exactly as the powers of two do. Hence, our observation (that  $T_n + 1$  is a power of two) remains true as  $n$  increases by 1. Therefore, being true for  $n = 1$  (as we have checked), it remains true for all larger  $n$ .

This type of argument is called *mathematical induction* and will be considered in Section 3.2.

**Stop and Think!** Can we now summarize our findings and say that we have proven that one needs  $2^n - 1$  steps to solve the Hanoi Towers puzzle with  $n$  disks?

Not quite: the word “needs” can be understood in many ways. Our argument shows that *our recursive solution* requires  $2^n - 1$  steps for  $n$  disks. But there may be other solutions. For example, one can easily imagine a solution that requires more steps: just make some additional moves back and forth with the smallest disk (this is always allowed). Maybe then we could say that *the minimal number of moves for  $n$  disks is  $2^n - 1$* ? It is true but some additional reasoning is necessary. We must show that no faster solution exists.

**Stop and Think!** Can you modify our argument to prove this?

We denoted by  $T_n$  the number of moves *in our solution*. Let us now change the notation and denote by  $T_n$  the *minimal number of moves* needed to solve the puzzle with  $n$  disks. Again, we want to show that  $T_n = 2T_{n-1} + 1$ . To do this, it suffices to show that  $T_n \leq 2T_{n-1} + 1$  and  $T_n \geq 2T_{n-1} + 1$ .

- $T_n \leq 2T_{n-1} + 1$ : for that, we need to show a solution for  $n$  disks using at most  $2T_{n-1} + 1$  steps, if there exists a solution for  $n - 1$  disks using  $T_{n-1}$  steps. This is easy: we just use the existing solution for  $n - 1$  steps twice (before and after the move of the largest disk).
- $T_n \geq 2T_{n-1} + 1$ : here, we need to show that *every* solution (rather than just our solution) for  $n$  disks requires at least  $2T_{n-1} + 1$  steps. In fact, our analysis contains all ingredients for that, we just need to place the emphasis correctly. Imagine there is *some* solution for  $n$  disks. This solution has to move the largest disk at some point. It may do it several times, but let us consider the first time when this happens. As we have discussed, just before the move one of the other rods is empty and the third one has all the smaller disks (in the only allowed order). The largest disk has not moved, so we conclude that the first part (before the first move of the largest disk) is a solution for  $n - 1$  disks, and therefore takes at least  $T_{n-1}$  steps. A similar argument shows that the part after the *last* move of the largest disk is also a solution for  $n - 1$  disks and requires at least  $T_{n-1}$  steps.<sup>2</sup> In total, we need at least  $2T_{n-1} + 1$  steps (or even more if the largest disk moved several times).

### 3.1.6 Binary Search

Binary search is an extremely efficient searching procedure that is frequently used in programming and computer science. If you haven’t heard about this idea before, we will discover it together by solving the following puzzle.

**Problem 78 – Guess a number.** In the “Guess a Number” game, your opponent has an integer  $1 \leq x \leq n$  in mind. You ask questions of the form “Is  $x = y$ ?” Your opponent replies either “yes”, or “ $x < y$ ” (that is, “my number is smaller than your guess”), or “ $x > y$ ” (that is, “my number is larger than your guess”). Your goal is to get the “yes” answer by asking at most  $k$  questions.

Let  $n = 3$  and  $k = 2$ : your goal is to guess  $1 \leq x \leq 3$  by asking at most two questions. How can you do this? Try it (question 1): [Coursera](#) ↗.

If you ask “Is  $x = 1$ ?” and get the “yes” answer, then you are done. But what if the opponent replies “ $x > 1$ ”? You conclude that  $x$  is equal to either 2 or 3, but you only have one question left. Similarly, if you ask “Is  $x = 3$ ?", the opponent may reply “ $x < 3$ ” and you will not be able to get the desired “yes” response by asking just one more question.

Let’s see what happens if your first question is “Is  $x = 2$ ”. If the opponent replies that  $x = 2$ , then you are done. If she replies that  $x < 2$ , then you already know that  $x = 1$ . Hence, you just ask “Is  $x = 1$ ?” as your second question and get the desired “yes” response. If the opponent replies that  $x > 2$ , your next question “Is  $x = 3$ ?” will get the “yes” response.

**Problem 79** Guess an integer  $1 \leq x \leq 7$  by asking at most three questions.

You may have already guessed that you are going to start by asking “Is  $x = 4$ ?” The reason is that in both cases,  $x < 4$  and  $x > 4$ , we *reduce the size of the search space from 7 to 3*:

<sup>2</sup>One can also reverse the time and see that the last part becomes the first part in the reversed video, and our problem is time-symmetric.

- if  $x < 4$ , then  $x$  is equal to either 1, 2, or 3;
- if  $x > 4$ , then  $x$  is equal to either 5, 6, or 7.

This, in turn, means that in both cases you can invoke the solution to Problem 78. Recursion! The resulting protocol of questions is shown in Figure 3.4.

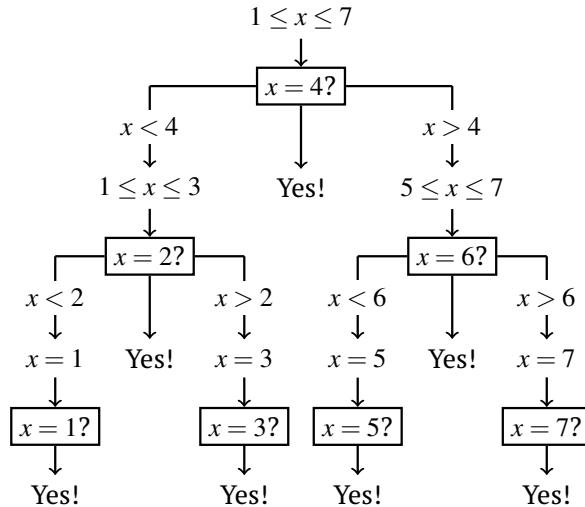


Figure 3.4: Guessing an integer  $1 \leq x \leq 7$  by asking at most three questions.

Interestingly enough, this strategy allows us to guess an integer  $1 \leq x \leq 2097151$  (over two million!) in just 21 questions.

**Problem 80** Guess an integer  $1 \leq x \leq 2097151$  by asking at most 21 questions. Try it (question 4): [Coursera](#).

The following code mimics the guessing process. The function `query` “knows” an integer  $x$ . A call to `query(y)` tells us whether  $x = y$ , or  $x > y$ , or  $x < y$ . The function `guess()` finds the number  $x$  by calling `query()`. It is called with two parameters, `lower` and `upper`, such that

$$\text{lower} \leq x \leq \text{upper},$$

that is,  $x$  lies in the segment  $[\text{lower}, \text{upper}]$ . It first computes the `middle` point of the segment  $[\text{lower}, \text{upper}]$  and then calls `query(middle)`. If  $x < \text{middle}$ , then it continues with the interval  $[\text{lower}, \text{middle} - 1]$ . If  $x > \text{middle}$ , then it continues with the interval  $[\text{middle} + 1, \text{upper}]$ .

```

def query(y):
    x = 1618235
    if x == y:
        return 'equal'
    elif x < y:
        return 'smaller'
    else:
        return 'greater'

def guess(lower, upper):
    middle = (lower + upper) // 2
    answer = query(middle)
    print(f'Is x={middle}? It is {answer}.')
    if answer == 'equal':
        return
    elif answer == 'smaller':
        guess(lower, middle - 1)
    else:
        guess(middle + 1, upper)
  
```

```

        guess(lower, middle - 1)
    else:
        assert answer == 'greater'
        guess(middle + 1, upper)

guess(1, 2097151)

```

```

Is x=1048576? It is greater.
Is x=1572864? It is greater.
Is x=1835008? It is smaller.
Is x=1703936? It is smaller.
Is x=1638400? It is smaller.
Is x=1605632? It is greater.
Is x=1622016? It is smaller.
Is x=1613824? It is greater.
Is x=1617920? It is greater.
Is x=1619968? It is smaller.
Is x=1618944? It is smaller.
Is x=1618432? It is smaller.
Is x=1618176? It is greater.
Is x=1618304? It is smaller.
Is x=1618240? It is smaller.
Is x=1618208? It is greater.
Is x=1618224? It is greater.
Is x=1618232? It is greater.
Is x=1618236? It is smaller.
Is x=1618234? It is greater.
Is x=1618235? It is equal.

```

Try changing the value of  $x$  and run this code to see the sequence of questions (but make sure that  $x$  lies in the segment that `guess` is called with).

In general, our strategy for guessing an integer  $1 \leq x \leq n$  will require about  $\log_2 n$  questions. Recall that  $\log_2 n$  is equal to  $b$  if  $2^b = n$ . This means that if we keep dividing  $n$  by 2 until we get 1, we will make about  $\log_2 n$  divisions. What is important here is that  $\log_2 n$  is a *slowly growing* function: say, if  $n \leq 10^9$ , then  $\log_2 n < 30$ .

```

from math import log2

def divide_till_one(n):
    divisions = 0
    while n > 1:
        n = n // 2
        divisions += 1
    return divisions

for d in range(1, 10):
    n = 10 ** d
    print(f'{n} {log2(n)} {divide_till_one(n)}')

```

```

10 3.321928094887362 3
100 6.643856189774724 6

```

```
1000 9.965784284662087 9
10000 13.287712379549449 13
100000 16.609640474436812 16
1000000 19.931568569324174 19
10000000 23.253496664211536 23
100000000 26.575424759098897 26
1000000000 29.897352853986263 29
```

The binary search method helps in daily life as follows. Imagine you are watching a recording of a concert where an orchestra plays a symphony of Mozart, and then a symphony of Shostakovich. But there is no timing: it is not said at what time Shostakovich starts, and you want to find this out. You can distinguish between Mozart and Shostakovich, but you don't have time to watch the entire recording (or even Mozart's part).

**Stop and Think!** What would you do?

Probably you would play the recording at the middle point. If this is Mozart, then you know that you should look for the change in the second half; if this is Shostakovich, you should find the change moment in the first half. This way, you reduce the question for the entire recording to the same question for one of its halves: recursion again!

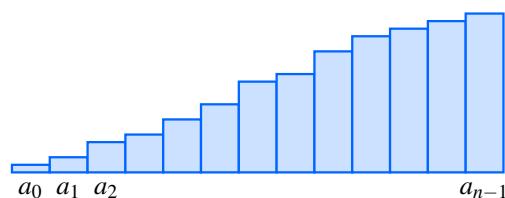
Perhaps the most important application of binary search is *searching sorted data*. Searching is a fundamental problem: given a sequence and an element  $x$ , we would like to check whether  $x$  is present in this sequence. For example, 3 is present in the sequence (7, 2, 5, 6, 11, 3, 2, 9) and 4 is not present in this sequence. Given the importance of the search problem, it is not surprising that Python has built-in methods for this.

```
print(3 in [7, 2, 5, 6, 11, 3, 2, 9])
print(4 in [7, 2, 5, 6, 11, 3, 2, 9])
```

```
True
False
```

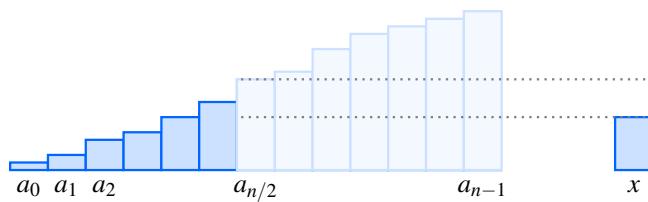
What is going on under the hood when one calls this `in` method? As you would expect, Python just scans the given sequence from left to right and compares every element to  $x$ . This simple method is called *linear scan*. It makes up to  $n$  comparisons on a sequence of length  $n$ . If the sequence does not contain  $x$ , we *have to* scan all the elements: if we skip an element, we can't be sure that it is not equal to  $x$ .

Things change drastically if the given data is *sorted*. Namely, assume that we are searching for an integer  $x$  in a sorted sequence  $(a_0, a_1, \dots, a_{n-1})$ .

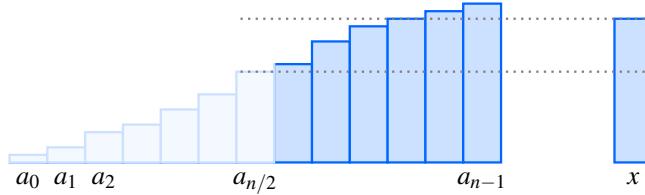


It turns out that in this case about  $\log_2 n$  comparisons are enough! This is a great speedup: the linear scan of a sorted array with a billion elements will take a billion comparisons, but binary search makes at most  $\log_2 10^9 < 30$  comparisons!

The idea is again to try to half the search space. To do this, we compare  $x$  with  $a_{n/2}$ . If  $x = a_{n/2}$ , then we are done. If  $x < a_{n/2}$ , then we can discard the right half of the sequence as  $x$  just cannot appear there. Indeed, since the sequence is sorted,



Similarly, if  $x > a_{n/2}$ , we discard the left half of the sequence as all its elements are certainly smaller than  $x$ :



This leads us to the following (recursive!) code.

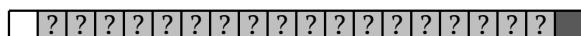
```
def binary_search(a, x):
    print(f'Searching {x} in {a}')
```

```
Searching 8 in [1, 2, 3, 3, 5, 6, 6, 8, 9, 9, 9]
Searching 8 in [6, 8, 9, 9, 9]
Searching 8 in [6, 8]
Found!
```

**Problem 81** In the array of cells, the first (leftmost) one is white, and the last one is black. All others are either black or white. Here is an example:



However, you don't see the colors of the cells: for you, this array looks like this:



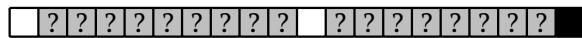
You may point to any cell and ask a question “What is its color?” Find a white-black pair, that is, a white cell that is directly followed by a black cell. Try it: [Coursera](#).

After solving this puzzle, you will see that binary search could help even when the data is not sorted!

But how do we know that there is a white-black pair? This is true because when moving from the first white cell to the last black cell, the color of a cell should switch from white to black at least once. Note two important things about this proof:

1. It does not rely on the fact that we have 20 cells. In other words, *it works for any number of cells*.
2. It proves that there exists a white-black pair, but it doesn't give an *efficient method* for finding this pair (by revealing the color of a few cells). In particular, if we start checking the color of the cells one by one from the left, then we will eventually find a white-black pair, but in the worst case it will require us to reveal the color of all cells.

Thus, we know that a white-black pair exists, but we still need to figure out an efficient method for finding it. Inspired by our previous examples, let's reveal the color of the middle cell. Assume, for example, that it is white.



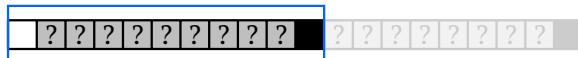
**Stop and Think!** Do you see how to proceed?

At first sight, it does not help us much. If we start revealing the color of its neighbor cells, all of them may appear to be white. Instead, let's focus on the right part of the sequence.



Do you see? It is the same problem again! The leftmost cell is white, the rightmost one is black. Hence, it must contain a pair of adjacent cells of opposite colors. And its length is twice smaller.

If the middle cell turns out to be black, we also have the same subproblem (though of length 11, but not 10).



Hence, in any case, we decrease the size by a factor of (almost) two: from a starting array of length  $n$ , we get an array of length  $\lceil \frac{n+1}{2} \rceil$ . This way, we will need to reveal the color of at most five cells before we find what we are looking for. Indeed, the size of the array will shrink as follows:

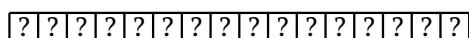
$$20 \rightarrow 11 \rightarrow 6 \rightarrow 4 \rightarrow 3 \rightarrow 2.$$

When the size of the current sequence is equal to two we already know the required pair since the leftmost cell in our sequence is white and the rightmost one is black.

We conclude this section with a puzzle that can be solved by binary search again. An element of an integer sequence is called a *local maximum* if it is not smaller than all its neighbors. For example, all local maximums of the following sequence are highlighted.



**Problem 82 – Local maximum.** Consider an integer sequence with 16 unknown elements.



Find (any) local maximum by revealing at most seven of its elements. Try it: [Coursera](#) ↗.

This is a more challenging problem than the previous ones. We encourage you to discuss it with other learners at [this forum thread](#) ↗.

## 3.2 Induction

### 3.2.1 Why Induction?

You are reviewing your friend's Python code. The code contains a function that, given a positive integer  $n$ , computes the sum of the first  $n$  positive integers:  $1 + 2 + \dots + n$ .

```
def sum_of_integers(n):
    assert n > 0
    return sum(range(1, n + 1))
```

You suggest your friend to improve the code by using the *formula for arithmetic series*: for every positive integer  $n$ ,

$$\sum_{i=1}^n i = 1 + 2 + \dots + n = \frac{n(n+1)}{2}. \quad (3.3)$$

A code that uses a formula  $n(n+1)/2$  instead of summing up all integers from 1 to  $n$  is more efficient in practice: already for  $n = 10^9$ , computing the sum takes noticeable time, whereas computing  $n(n+1)/2$  takes almost nothing.

**Stop and Think!** How could you convince your friend that (3.3) is true for every positive integer  $n$ ?

One possibility is to check that it holds for all  $1 \leq n \leq 100$ .

```
print(all(sum(range(1, n + 1)) == n * (n + 1) // 2
         for n in range(1, 101)))
```

True

This code snippet ensures that the formula above is true for all  $1 \leq n \leq 100$ . But it only convinces us that the formula holds *sometimes*. Moreover, we have no possibility to check every positive integer  $n$  for a simple reason: there are infinitely many of them! Thus, to be sure that the formula holds for *all values of  $n$* , we need a rigorous mathematical proof.

This is reminiscent of software testing. Whereas one often can run a program on several tests, it is usually impossible to cover all cases with tests. This is why it is important to *prove* correctness of a program (for all input values).

*Mathematical induction* is a general way of proving statements of the form

for all integer  $n \geq c$ , the statement  $A(n)$  is true.

Here,  $A(n)$  is a statement (that might be true or false) that depends on  $n$ . In our warm-up example above,  $A(n)$  asserts that  $1 + 2 + \dots + n$  is equal to  $n(n+1)/2$ . In the next section, we introduce the method of mathematical induction and give many examples.

In general, checking a few first values of  $n$  to verify the correctness of  $A(n)$  is a bad idea. Later in the book, we'll encounter several examples where  $A(n)$  is true in the beginning, but eventually it becomes false. Moreover, there are examples of simple statements  $A(n)$  that are true for all  $1 \leq n \leq 10^{80}$ , but are false in general! That is, there exists  $n$  such that  $A(n)$  is false, but the minimum such  $n$  is so huge that it cannot be found by a brute force search.

### 3.2.2 What is Induction?

A mathematical induction proof of the fact that all of the statements  $A(1), A(2), A(3), \dots$  are true, consists of two parts.

**The base case:** First, we prove that  $A(1)$  is true.

**The induction step:** Then, we prove that for every  $n \geq 1$ , the statement  $A(n)$  implies the statement  $A(n+1)$  (that is, if  $A(n)$  is true, then  $A(n+1)$  is true).

Mathematical induction assures that once we (i) proved the base case and (ii) proved the induction step, we have actually proved *all* statements  $A(n)$  for  $n \geq 1$ . See Figure 3.5: (a) the base case ensures that the first statement is true; (b) the induction step lets us move from any statement to the following one; (c) from this, we conclude that all the statements are true.

Thus, a typical induction proof consists of two parts. The first part (usually the easiest one) is to prove the base case. Of course, there is nothing special about  $n = 1$ , and we could start at

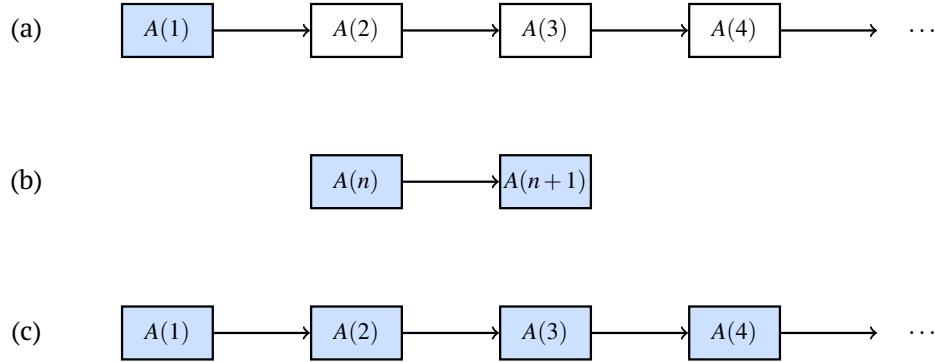


Figure 3.5: Proving that  $A(n)$  is true for all integer  $n \geq 1$  by induction: (a) the base case; (b) the induction step; (c) the resulting picture.

any other number  $n = c$  (say,  $c = 0$ , or  $c = 15$ , or  $c = -3$ ). Either way, we will prove all statements  $A(c), A(c+1), A(c+2), \dots$

The second part of an induction proof is to prove the induction step. Here, we assume that we already know that the statement  $A(n)$  is true. This assumption is called the *induction hypothesis*. Assuming the induction hypothesis, we prove the statement  $A(n+1)$ , and this finishes the proof. Sometimes it comes in handy to use a stronger induction hypothesis: assume that all the previous statements  $A(1), \dots, A(n)$  are true, and use them to prove correctness of  $A(n+1)$ . This variation of induction proofs is often called *strong induction*.

Let us recall a strong induction proof that we have seen. Consider the following statement  $A(n)$ :  $n$  can be represented as  $3k + 5l$  where  $k$  and  $l$  are non-negative integers. Problem 70 states that  $A(n)$  is true for all  $n \geq 8$ . Indeed, for  $n = 8, 9, 10$ ,  $A(n)$  is true:  $8 = 3 \cdot 1 + 5 \cdot 1$ ,  $9 = 3 \cdot 3 + 5 \cdot 0$ ,  $10 = 3 \cdot 0 + 5 \cdot 2$ . This is the base case. For the induction step, assume that  $A(8), A(9), \dots, A(n)$  are all true for some  $n \geq 11$ . Since  $n \geq 11$ ,  $n - 3 \geq 8$  and hence  $A(n-3)$  is true. Therefore,  $n - 3 = 3 \cdot k + 5 \cdot l$  for some  $k, l$ . Thus,  $n = 3k + 5l + 3 = 3(k+1) + 5l$ .

Mathematical induction can be visualized as a domino effect: Given an infinitely long chain of dominoes, each separated by a small distance, it suffices to push the first domino to knock down all dominoes in the chain (see Figure 3.6). In fact, this holds due to mathematical induction! Indeed, for every  $n \geq 1$ , let  $A(n)$  be the proposition that the domino number  $n$  falls down. Then, we ensure  $A(1)$  by pushing the first domino. Since the dominoes are too close to one another,  $A(n)$  implies  $A(n+1)$ : each domino falls after it is knocked over by the previous one.



Figure 3.6: The domino effect: pushing the first domino knocks down all other dominoes. (Source: [Wikipedia](#).)

**Problem 83 – arithmetic series.** Using mathematical induction, prove the formula (3.3): for every integer  $n \geq 1$ , the sum of all integers from 1 to  $n$  is  $n(n+1)/2$ . In other words,

$$\sum_{i=1}^n i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}.$$

The base case of  $n = 1$  is easy to check:  $1 = \frac{1 \cdot 2}{2}$ . The induction step from  $n$  to  $n+1$  for every  $n \geq 1$  can be proven as follows:

$$\begin{aligned} \sum_{i=1}^{n+1} i &= 1 + 2 + \cdots + n + (n+1) \\ &= \left( \sum_{i=1}^n i \right) + (n+1) \\ &= \frac{n(n+1)}{2} + (n+1) \quad (\text{by ind. hypothesis}) \\ &= \frac{n(n+1)}{2} + \frac{2(n+1)}{2} \\ &= \frac{(n+1)(n+2)}{2}. \end{aligned}$$

While we finished the proof of this beautiful formula, this proof did not reveal a way of arriving at this formula. There is no general recipe for finding such expressions, but the following tricks are often handy.

**Arithmetic trick.** When looking at the sum  $1 + 2 + 3 + \cdots + (n-2) + (n-1) + n$ , one may want to pair up the first and the last numbers, then the second and the second to last numbers, and so on. Indeed, each such pair has a sum  $(n+1)$ , and the number of pairs is  $n/2$ . A simple visualization of this trick is to write all the numbers from 1 to  $n$  in a row, and then write them back one more time in a row below it. We will have  $n$  columns each with sum  $(n+1)$ . This trick was allegedly discovered by “the greatest mathematician since antiquity” Carl Gauss (see more on [Wikipedia](#) ☰).

$$\begin{array}{ccccccccccccc} 1 & + & 2 & + & \cdots & + & (n-1) & + & n \\ n & + & (n-1) & + & \cdots & + & 2 & + & 1 \\ \hline (n+1) & + & (n+1) & + & \cdots & + & (n+1) & + & (n+1) \end{array}$$

**Geometric trick.** Think of each number  $i$  as a strip of length  $i$  and width 1. Then the total area of these strips is half the area of an  $n \times n$  square plus  $n$  little triangles of area  $1/2$ : total of  $\frac{n^2}{2} + n \cdot \frac{1}{2}$ , see Figure 3.7. Another way to visualize this formula is to take one vertical and one horizontal strip of length  $i$  for every  $i$ . You can fill in an  $(n+1) \times n$  rectangle using these strips (see Figure 3.7).

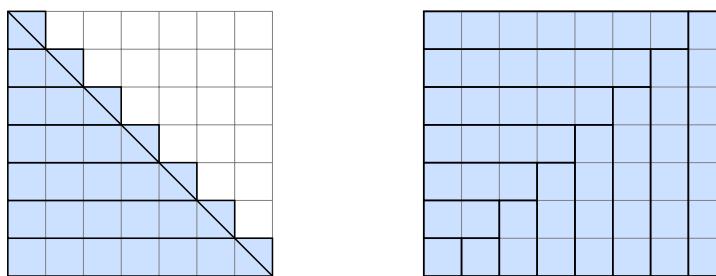
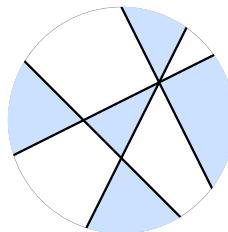


Figure 3.7: Geometric proofs of the arithmetic series formula.

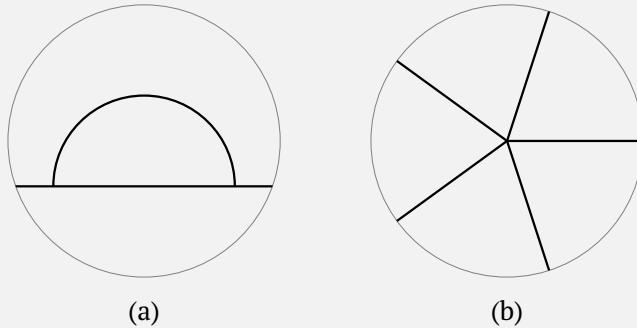


Figure 3.8: Johann Carl Friedrich Gauss (1777–1855). (Source: [Wikipedia](#).)

**Problem 84** The plane is partitioned into several regions by straight (infinitely long) lines. Prove that the plane can be colored in just two colors so that neighboring regions have different colors.



Note that it is crucial that all lines in this problem are *straight* and *infinitely long*. Indeed, the figure below shows that if the lines are not straight (a), or if the lines are not infinitely long in both directions (b), then more than two colors may be needed.



In fact, for regions bounded by any (for example, curvy and finite) lines, one can always use just four colors. This is the famous [four color theorem](#), which to this day only has proofs involving a large computer search!

Let us apply mathematical induction to Problem 84. We want to show that regions formed by  $n$  lines can be colored in two colors so that neighboring regions have different colors. We prove this by induction on  $n$ . For the base case  $n = 1$ , we can take a coloring where the two regions have different colors. For the induction step from  $n$  to  $n + 1$ , let us assume that we have a proper coloring of the plane with  $n$  lines, and that we are adding one more line  $L$ , see Figure 3.9. Now, all regions on one side of  $L$  keep their colors, while all regions on the other side of  $L$  switch their colors. Consider two neighboring regions  $R_1$  and  $R_2$ . If their common border is not a part of  $L$ , then they had different colors in the old coloring, and they still have different colors in the new coloring. On the other hand, if their common border belongs to  $L$ , then they used to be one region and had the same color, but one of the regions switched color.

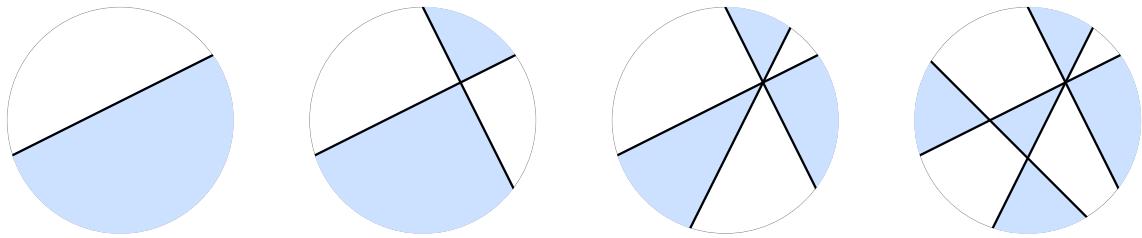


Figure 3.9: An example of the inductive coloring from the proof. After adding a new line  $L$ , we keep colors of all regions on one side of  $L$ , and switch colors of all regions on the other side of  $L$ .

**Stop and Think!** A simple interest deposit is a kind of deposit where you earn  $x\%$  of the *initial* deposit each period (day, month or year). A compound interest deposit is a deposit where you earn  $x\%$  of *what you already have* each period (day, month or year). Will you get \$1 000 000 faster starting with \$1 000 and earning 2% every day with compound interest or with simple interest?

While simple interest will earn you \$20 every day, compound interest will give you \$20 on the first day, \$20.4 on the second day, \$20.808 on the third day, and more and more every following day.

In the case of compound interest, if you start with some amount of money, then after  $n$  days, this amount is multiplied by  $1.02^n$ . In the case of simple interest, the money is multiplied by  $(1 + n \cdot 0.02)$ . The following code demonstrates the (huge) difference between these two cases for large values of  $n$ .

```
import matplotlib.pyplot as plt
import numpy as np

plt.xlabel('$n$')
plt.ylabel('Money ($)')
x = np.arange(200)
plt.plot(x, 1.02 ** x, label='$1.02^n$')
plt.plot(x, 1 + 0.02 * x, label='1+0.2^n')
plt.legend(loc='upper left')
plt.savefig('bernoulli.png')
```

This holds in general: compound interest is always at least as profitable as simple interest. This is known as Bernoulli's inequality, and we will prove it using mathematical induction.

**Theorem 3.2.1 – Bernoulli's inequality.** For every  $x \geq -1$  and every integer  $n \geq 0$ , it holds that

$$(1+x)^n \geq 1+xn. \quad (3.4)$$

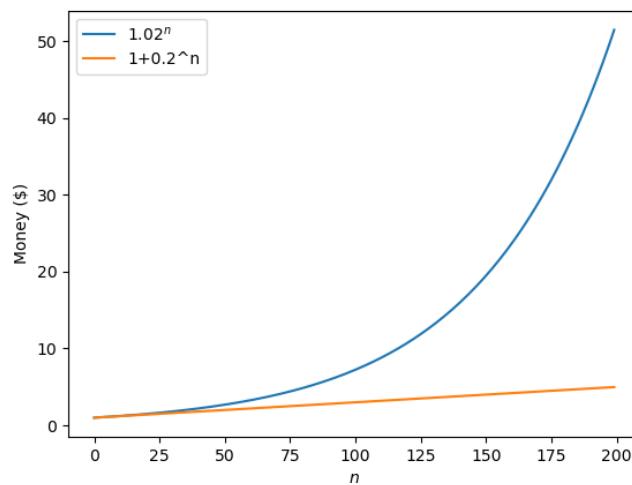
The base case  $n = 0$  holds trivially:  $(1+x)^0 = 1 = 1+x \cdot 0$ . Now we prove the induction step from  $n$  to  $(n+1)$  for every  $n \geq 0$ :

$$\begin{aligned} (1+x)^{n+1} &= (1+x)^n \cdot (1+x) \\ &\geq (1+xn) \cdot (1+x) \quad (\text{by ind. hypothesis}) \\ &= 1+x(n+1)+x^2n \\ &\geq 1+x(n+1), \end{aligned}$$

where we used both the induction hypothesis for  $n$ , and the fact that an inequality can be multiplied by a non-negative value  $(1+x)$ .



Figure 3.10: Jacob Bernoulli (1655–1705). (Source: Wikipedia [↗](#)).



It is instructive to see how exponential functions (such as  $1.02^n$ ) grow, and to see how fast they reach large values. For example, how many days of 2% compound interest does it take to get from \$1 000 to \$1 000 000?

```
def days_to_target(starting_amount, earn_percent,
                  target_amount):
    day = 1
    amount = starting_amount
    daily_factor = (1 + earn_percent / 100.0)
    while amount < target_amount:
        day += 1
        amount = amount * daily_factor
    return day

def print_example(starting_amount, earn_percent,
                  target_amount):
    days = days_to_target(starting_amount, earn_percent,
                          target_amount)
    print(f"If you start with ${starting_amount} "
          f"and earn {earn_percent}% a day,"
          f"\nyou will have more than ${target_amount} "
          f"on day {days}!")
```

```
print_example(1000, 2, 1000000)
```

If you start with \$1000 and earn 2% a day,  
you will have more than \$1000000 on day 350!

Or how much money will I have after a year?

```
def how_much_money(starting_amount, earn_percent, day):
    daily_factor = 1 + (earn_percent / 100.0)
    return starting_amount * (daily_factor ** (day - 1))

def print_example(starting_amount, earn_percent, day):
    money = int(how_much_money(starting_amount,
                                earn_percent, day))

    print(f"If you start with ${starting_amount} "
          f"and earn {earn_percent}% a day,"
          f"\non day {day} you will have "
          f"more than ${money}!")
```

```
print_example(1000, 2, 365)
```

If you start with \$1000 and earn 2% a day,  
on day 365 you will have more than \$1350400!

We encourage you to play with various exponential functions in this Python notebook ( Try it: [Coursera](#) ).

The inequality of arithmetic and geometric means states that for any pair of non-negative numbers  $x$  and  $y$ , their arithmetic mean is no less than their geometric mean:

$$\frac{x+y}{2} \geq \sqrt{xy}.$$

This follows from the observation that the square of any number is non-negative.

$$0 \leq (\sqrt{x} - \sqrt{y})^2 = x + y - 2\sqrt{xy},$$

which, after rearranging its terms, implies the original inequality.

In fact, the inequality between the arithmetic and geometric means holds for arbitrarily many positive numbers.

**Theorem 3.2.2** (inequality of arithmetic and geometric means) For any  $x_1, \dots, x_n > 0$

$$\frac{x_1 + x_2 + \dots + x_n}{n} \geq \sqrt[n]{x_1 x_2 \dots x_n}. \quad (3.5)$$

For simplicity, first we scale all  $x_i$  so that  $x_1 x_2 \dots x_n = 1$ . Namely, assume that the geometric mean of these numbers is  $c = \sqrt[n]{x_1 x_2 \dots x_n}$ . Then let us divide each  $x_i$  by  $c$ . Note that both the arithmetic and the geometric mean decrease by a factor of  $c$ , therefore, the ratio between the two means does not change. Now, we indeed have  $x_1 x_2 \dots x_n = 1$  without loss of generality.

It remains to show that  $x_1 x_2 \dots x_n = 1$  implies

$$x_1 + x_2 + \dots + x_n \geq n.$$

We will prove this inequality by induction on  $n$ . The base case of  $n = 1$  is trivial:  $x_1 \geq 1$ . For the induction step from  $n \geq 1$  to  $(n+1)$  we assume that we have  $x_1, \dots, x_{n+1} > 0$  whose product equals 1. In particular, this means that at least one of the numbers is  $\leq 1$  (indeed, if all numbers are  $> 1$ , then their product is  $> 1$ ). Similarly, at least one of the numbers is  $\geq 1$ . Let us assume that  $x_1 \leq 1$  and  $x_2 \geq 1$ . Then  $(x_1 - 1)(x_2 - 1) \leq 0$ , which implies that  $x_1 + x_2 \geq x_1 x_2 + 1$ . In particular we have that

$$x_1 + x_2 + \dots + x_n + x_{n+1} \geq 1 + (x_1 x_2) + x_3 + \dots + x_{n+1}.$$

Now, consider  $n$  numbers

$$(x_1 x_2), x_3, \dots, x_{n+1}.$$

Since their product is 1, we can apply the induction hypothesis to them:  $x_1 x_2 + x_3 + \dots + x_{n+1} \geq n$ . We conclude that

$$x_1 + x_2 + \dots + x_{n+1} \geq 1 + (x_1 x_2) + x_3 + \dots + x_n \geq 1 + n.$$

**Problem 85** Prove that for every  $n \geq 1$ ,

$$\sum_{i=1}^{n-1} i \cdot (n-i) = \frac{(n-1)n(n+1)}{6}.$$

The induction base case is  $n = 1$ , where both sides of the formula equal 0. For the induction step, assume that this formula holds for  $n \geq 1$ , and let us prove this statement for  $n+1$ , simply by plugging in  $(n+1)$  everywhere that we previously saw  $n$ .

$$1 \cdot n + 2 \cdot (n-1) + \dots + n \cdot 1 = \frac{n(n+1)(n+2)}{6}.$$

Indeed,

$$\begin{aligned} & 1 \cdot n + 2 \cdot (n-1) + \dots + n \cdot 1 \\ &= 1 \cdot (n-1) + 2 \cdot (n-2) + \dots + (n-1) \cdot 1 \\ &+ 1 + 2 + \dots + (n-1) + n \\ &= \frac{(n-1)n(n+1)}{6} \quad (\text{by ind. hypothesis}) \\ &+ 1 + 2 + \dots + (n-1) + n \\ &= \frac{(n-1)n(n+1)}{6} + \frac{n(n+1)}{2} \quad (\text{by Problem 83}) \\ &= \frac{n(n+1)(n+2)}{6}. \end{aligned}$$

Practice proofs by induction on the following problems.

**Problem 86** Prove that the following equalities hold for every  $n > 0$ .

1.  $1 + 3 + 5 + \dots + (2n-1) = n^2$  (for a geometric proof of this formula see Figure 3.11);
2.  $1 \cdot 2^1 + 2 \cdot 2^2 + 3 \cdot 2^3 + \dots + n \cdot 2^n = (n-1) \cdot 2^{n+1} + 2$ ;
3.  $1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{2^n} \geq \frac{n}{2} + 1$ .

### 3.2.3 Where to Start Induction?

Sometimes we prove statements which do not necessarily hold for all  $n \geq 1$ , but hold only for  $n \geq c$  for some number  $c$ . To do this, we can just start our induction base at  $n = c$  rather than at  $n = 1$ .

**Problem 87** Prove that  $2^n \geq n^3$  for all  $n \geq 10$ .

The following plots show that this statement is not even true for  $n < 10$ .

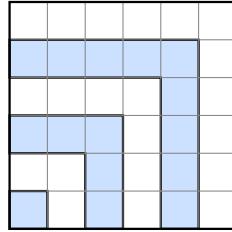
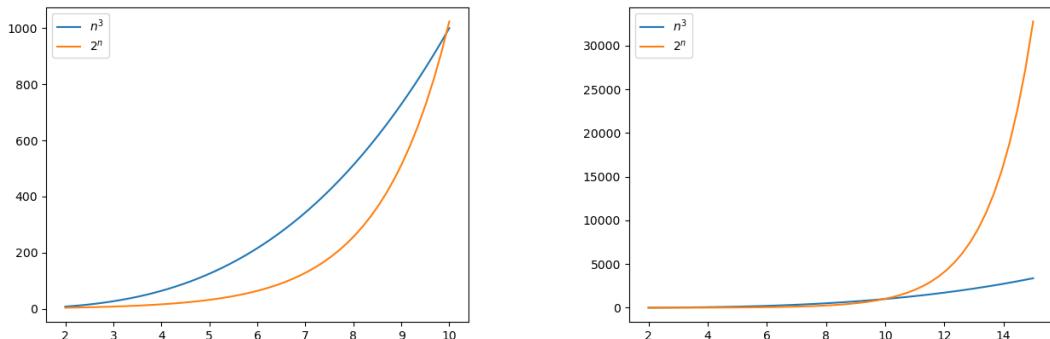


Figure 3.11: As it often happens with induction proofs, the induction proof convinces us that the formula is correct, but gives no clues as to where the formula is actually coming from. For this particular case, there is again a neat geometric visualization of the formula.

```
import matplotlib.pyplot as plt
import numpy as np

for m in [10, 15]:
    plt.clf()
    n = np.linspace(2, m)
    plt.plot(n, n ** 3, label='$n^3$')
    plt.plot(n, 2 ** n, label='$2^n$')
    plt.legend(loc='upper left')
    plt.savefig(f'plotn3vs2n{m}.png')
```

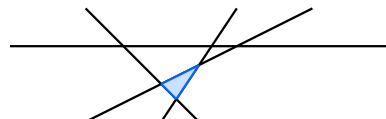


We can prove the statement from Problem 87 for all  $n \geq 10$  using mathematical induction. The base case of  $n = 10$  is easy to check:  $2^n = 1024 > 1000 = n^3$ . For the step from  $n$  to  $n + 1$ , the left-hand side is multiplied by 2, but the right-hand side is multiplied by

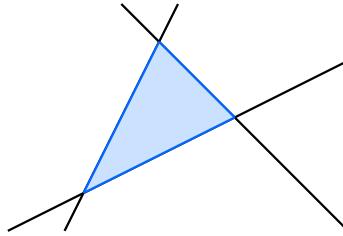
$$\frac{(n+1)^3}{n^3} = \left(1 + \frac{1}{n}\right)^3.$$

For  $n \geq 10$ , this expression is bounded from above by  $1.1^3 = 1.331 < 2$ . Thus, for every  $n \geq 10$ , we multiply the greater left side by a larger number, and have that  $2^n \geq n^3$ .

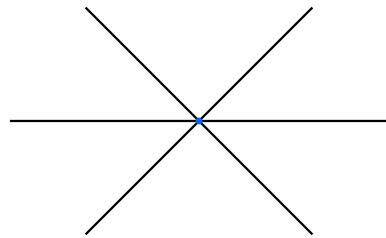
**Problem 88** Several straight lines (at least three) cut a plane into pieces. Each line intersects with every other line, and all the intersection points are different. Prove that there is at least one triangular piece.



We will prove this statement by induction on the number of lines. The base case is when there are only three lines. Since the lines intersect at three different points, they must form a triangle.

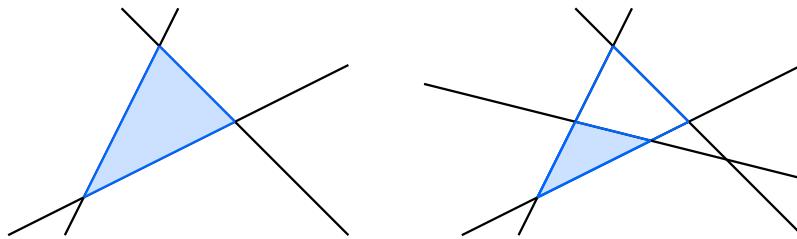


Indeed, the only case when three lines do not form a triangle — when they intersect at one point — is forbidden.

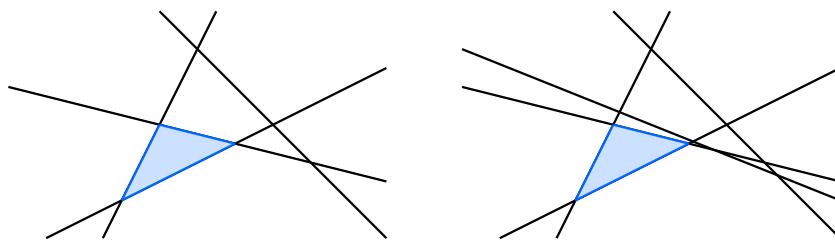


In order to prove the induction step, we start with  $n \geq 3$  lines (which, by the induction hypothesis, already form a triangle  $T$ ) and add one more line  $L$ . Consider two cases:

**Case 1:  $L$  intersects  $T$ .** In this case,  $L$  must intersect two sides of  $T$  ( $L$  cannot intersect  $T$  in a vertex, because no triple of lines intersects at one point). When a line intersects two sides of a triangle, it creates a new triangle.



**Case 2:  $L$  does not intersect  $T$ .** In this case,  $T$  remains intact, so the new picture still has the triangle  $T$ .



#### 3.2.4 Proving Stronger Statements May Be Easier!

Recall from Section 3.1.5 that the number of moves in the Hanoi Towers puzzle with  $n$  discs is given by the recurrent formula

$$T_n = \begin{cases} 2T_{n-1} + 1 & \text{for } n > 1; \\ 1 & \text{for } n = 1. \end{cases}$$

A simple induction argument shows that  $T_n = 2^n - 1$  for every  $n \geq 1$ . The base case holds as  $T_1 = 1 = 2^1 - 1$ . The induction step follows from

$$T_n = 2T_{n-1} + 1 = 2 \cdot (2^{n-1} - 1) + 1 = 2^n - 1.$$

We have an induction proof of the statement  $T_n = 2^n - 1$ . It must be even easier to prove a weaker statement  $T_n \leq 2^n$ . Let us try to prove this by induction again. The base case for  $n = 1$  trivially holds. However, in the induction step, we can only say that

$$T_n = 2T_{n-1} + 1 \leq 2 \cdot 2^{n-1} + 1 = 2^n + 1,$$

which is not sufficient for our goals!

**Stop and Think!** How is it even possible that we can inductively prove that  $T_n = 2^n - 1$ , but cannot prove a weaker statement that  $T_n < 2^n$ ?

While this might seem strange at first, sometimes for an induction proof one needs to strengthen the statement. The trick here is that this also allows one to use a *stronger induction hypothesis*. And, having a stronger hypothesis, one has more tools to prove stronger statements.

**Problem 89** Prove that

$$\begin{aligned} 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \cdots + \frac{1}{99} - \frac{1}{100} \\ = \frac{1}{51} + \frac{1}{52} + \cdots + \frac{1}{100} \end{aligned}$$

We will solve a more general problem:

**Problem 90** Prove that for every  $k \geq 1$ :

$$\begin{aligned} 1 - \frac{1}{2} + \cdots + \frac{1}{2k-1} - \frac{1}{2k} \\ = \frac{1}{k+1} + \frac{1}{k+2} + \cdots + \frac{1}{2k} \end{aligned}$$

This will imply Problem 89 by setting  $k = 50$ . The induction base case  $k = 1$  is easy to verify:

$$1 - \frac{1}{2} = \frac{1}{2}.$$

For the induction step from  $k \geq 1$  to  $k+1$ , it suffices to show that the sum on the left and the sum on right change by the same amount when going from  $k$  to  $k+1$ . The sum on the left simply increases by  $\frac{1}{2k+1} - \frac{1}{2k+2}$ , while the sum on the right increases by

$$\begin{aligned} & \frac{1}{k+2} + \frac{1}{k+3} + \cdots + \frac{1}{2k} + \frac{1}{2k+1} + \frac{1}{2(k+1)} \\ & - \left( \frac{1}{k+1} + \frac{1}{k+2} + \frac{1}{k+3} + \cdots + \frac{1}{2k} \right) \\ & = \frac{1}{2k+1} + \frac{1}{2(k+1)} - \frac{1}{k+1} \\ & = \frac{1}{2k+1} - \frac{1}{2(k+1)}. \end{aligned}$$

Thus, the expressions on the left-hand side and on the right-hand side are the same initially for  $k = 1$ , and each time we increment  $k$  these two expressions change by the same value. Therefore, they stay the same for all values of  $k \geq 1$ .

**Problem 91** Prove that for every  $n \geq 1$  it holds that

$$\sum_{i=1}^n \frac{1}{i(i+1)} < 1.$$

First, let us see whether this sum is less than 1 for small values of  $n$ .

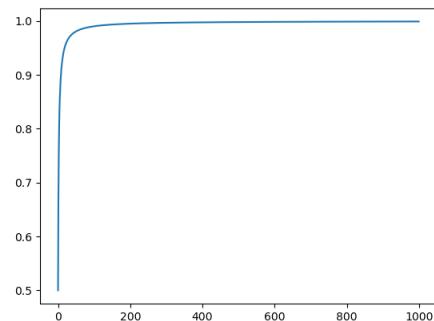
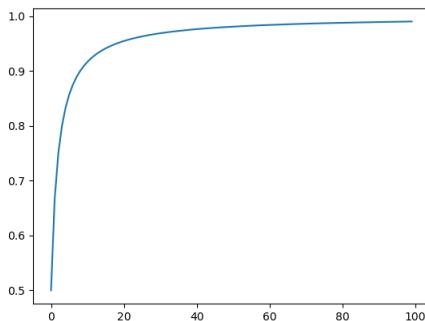
```

from itertools import accumulate
import matplotlib.pyplot as plt

n = 1000
sums = [*accumulate(1 / (i * (i + 1)))
         for i in range(1, n + 1)]

for k in (n // 10, n):
    plt.clf()
    plt.plot(sums[:k])
    plt.savefig(f'sum{k}.png')

```



Now, let us prove an even stronger statement: not only is this sum less than one, but it actually equals  $1 - \frac{1}{n+1}$ . We will use mathematical induction to prove this stronger statement. When  $n = 1$ ,

$$\sum_{i=1}^n \frac{1}{i(i+1)} = \frac{1}{2} = 1 - \frac{1}{n+1}.$$

Now if we assume that this holds for some  $n \geq 1$ , then for  $n+1$  we have

$$\begin{aligned}
& \sum_{i=1}^{n+1} \frac{1}{i(i+1)} \\
&= \sum_{i=1}^n \frac{1}{i(i+1)} + \frac{1}{(n+1)(n+2)} && \text{(by ind. hypothesis)} \\
&= 1 - \frac{1}{n+1} + \frac{1}{(n+1)(n+2)} \\
&= 1 - \frac{1}{n+2}
\end{aligned}$$

### 3.2.5 What Can Go Wrong with Induction?

**Problem 92** The Fibonacci sequence

$$F_0 = 0, F_1 = 1, F_2 = 1, F_3 = 2, F_4 = 3, F_5 = 5, F_6 = 8, \dots$$

is defined as:  $F_0 = 1$ ,  $F_1 = 1$ , and  $F_n = F_{n-2} + F_{n-1}$  for all  $n \geq 2$ . Find a mistake in the following proof of the statement " $F_n \geq 2^{n/2}$  for all  $n \geq 6$ ".

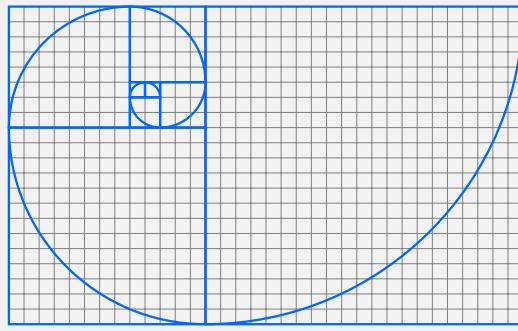
The proof is by induction. The base case  $n = 6$  holds trivially:  $F_6 = 8 = 2^{6/2}$ . The

induction step

$$\begin{aligned}
 F_n &= F_{n-2} + F_{n-1} \\
 &\geq 2^{\frac{n-2}{2}} + 2^{\frac{n-1}{2}} && \text{(by ind. hypothesis)} \\
 &= 2^{\frac{n-2}{2}} \cdot (1 + \sqrt{2}) \\
 &\geq 2^{\frac{n-2}{2}} \cdot 2 \\
 &= 2^{\frac{n}{2}}.
 \end{aligned}$$

This sequence is one of the most popular integer sequences with applications not only in mathematics and computer science, but also in music and even nature! Read more on [Wikipedia](#) or the [Online Encyclopedia of Integer Sequences](#).

The picture below shows a tiling of a rectangle by squares with side lengths 1, 1, 2, 3, 5, 8, 13, 21 and a spiral through their corners.



The growth factor of the spiral is the *golden ratio*:

$$\phi = \frac{1 + \sqrt{5}}{2} = 1.6180339887\dots$$

that also reflects the growth rate of  $F_n$ . The exact value of  $F_n$  is given by Binet's formula:

$$F_n = \frac{\phi^n - \psi^n}{\sqrt{5}},$$

where  $\psi = 1 - \phi = -\phi^{-1}$ . Interestingly, the formula involves irrational numbers though  $F_n$  is an integer. A similar closed-form expression can be derived for any sequence defined recursively (with constant coefficients).

The issue with this solution is quite subtle: we prove the base case only for one value of  $n$ , but we use it as if we proved it for at least two values of  $n$ . It is easier to see this for specific values of  $n$ . In the base case, we only proved that  $F_n \geq 2^{n/2}$  for  $n = 6$ . When we prove this inequality for  $n = 7$ , the induction step assumes that the inequality holds for  $n = 6$  and  $n = 5$ . Which is not true! We have only verified the inequality for  $n = 6$ . (If it held for  $n = 5$  too, then the whole induction proof would be correct.)

This teaches us an important lesson: If the induction step assumes the correctness of the statement for several smaller values of  $n$ , then we have to examine several values of  $n$  in the base case. For example, if the induction step assumes that the statement holds for  $n$  and  $n - 1$ , and proves the correctness of the statement for  $n + 1$ , then the induction base must cover at least two consecutive values of  $n$ .

While this proof is flawed, the statement is actually correct:  $F_n \geq 2^{n/2}$  for all  $n \geq 6$ . One way to prove this is to check the base cases  $n = 6, n = 7$ , and then repeat the previous proof of the induction step. This flaw in the proof may seem insignificant, but in fact it is very dangerous. For example, the same (flawed) argument could be used to prove the statement that " $F_n$  is even" for all  $n \geq 6$ :

This holds for  $n = 6$ , and if  $F_{n-2}$  and  $F_{n-1}$  are even, then  $F_n = F_{n-2} + F_{n-1}$  would be even too.

This statement is not even correct! We will see more examples of such flaws below.

**Problem 93** Find a mistake in the following proof of the statement “Any  $n \geq 1$  points on the plane lie on a line”.

Of course, for any point there is a line containing it, so the base case  $n = 1$  holds. For the induction step from  $n$  to  $n + 1$ , consider points  $p_1, p_2, \dots, p_{n+1}$ . By the induction hypothesis, there is a line  $L_1$  containing the points  $p_1, p_2, \dots, p_n$ . Similarly, there is a line  $L_2$  containing the points  $p_2, \dots, p_{n+1}$ . Since  $L_1$  and  $L_2$  both contain all the points  $p_2, p_3, \dots, p_n$ ,  $L_1$  and  $L_2$  are the same line. Therefore, all  $n + 1$  points lie on one line.

It is not true that all points lie on a line. For example, consider three points of a (non-degenerate) triangle, there is no line passing through all three of them. Again, let us carefully examine what assumptions the proof of the induction step requires. We find a line  $L_1$  passing through the first  $n$  points, and we can indeed find such a line by the induction hypothesis. Similarly, we can indeed find  $L_2$ . If these two lines passed through at least two (distinct) points, then they would indeed coincide. They have  $n - 1$  points in common:  $p_2, \dots, p_n$ . But for  $n = 2$ , this gives just a single common point. Hence, for  $n = 2$  we cannot conclude that  $L_1$  and  $L_2$  are the same line, and cannot finish the proof of the induction step. This induction proof failed only because we could not finish the proof of the case  $n = 2$ . While the induction step would indeed work for  $n > 2$ , there is no base case to base this proof on.

**Problem 94** Find a mistake in the following proof of the statement “All horses are of the same color”.

We prove this statement by induction on  $n$  — the number of horses. The base case of  $n = 1$  is indeed trivial. Assuming that the induction hypothesis holds for any set of  $n$  horses, we prove the statement for  $n + 1$  horses. Let us consider the first  $n$  horses. By the induction hypothesis they all are of the same color. Now let us take the last  $n$  horses, similarly, they are of the same color. But then the  $n - 1$  horses in the middle are of the same color as the first  $n$  horses and the last  $n$  horses. This implies that all  $n + 1$  horses are of the same color.

The next problem is similar to the previous one. Try to carefully examine what assumptions the induction step of the proof makes, and find a mistake in this proof.

**Problem 95** Find a mistake in the following proof of the statement “For any integer  $n \geq 0$ ,  $5n = 0$ ”.

The base case holds trivially:  $n = 0$  implies that  $5n = 0$ . For the induction step, we assume that the statement holds for all numbers 0 through  $n$ , and we want to prove the statement for  $n + 1$ . First, we write  $n + 1 = i + j$  where  $i, j \leq n$ . Then

$$5(n + 1) = 5(i + j) = 5i + 5j = 0 + 0 = 0.$$

Again, this proof “almost works”, the induction step only fails for the case of  $n = 1$ . The induction step assumes that  $n + 1$  can be written as a sum  $n + 1 = i + j$  where  $i, j \leq n$ . But this is not true for  $n = 0$ : no sum of numbers  $i, j \leq 0$  would give  $n + 1 = 1$ .

See other examples of horse paradox in [Wikipedia](#).

### Summary

- Mathematical induction is used to prove that some statements  $A(i)$  hold for all values of  $i$ .
- An induction proof consists of two parts: the base case and the induction step.
- The base case assures that  $A(i)$  holds for some (not necessarily small) values of  $i$ .
- The base case and the induction step must be consistent: if the induction step uses  $A(n)$

and  $A(n - 1)$ , then the base case should cover at least 2 consecutive values of  $n$ .

- The induction step for proving  $A(n + 1)$  can use  $A(n)$  or even all  $A(i)$  for  $i \leq n$ . The latter is called strong induction.
- Sometimes proving a stronger statement by induction may be easier.

Try to use these techniques to solve the following problems Try it: [Coursera](#) ↗..

**Problem 96** Which of the following is the sum of all integers from 5 to  $n$  for  $n \geq 5$ ?

- $\frac{(n+5)(n-6)}{2}$
- $\frac{(n+5)(n+1)}{2}$
- $\frac{(n+5)(n-5)}{2}$
- $\frac{(n+5)(n-4)}{2}$
- $\frac{n(n+1)}{2}$

**Problem 97** Which of the following is  $1 + 3 + 5 + \dots + (2n - 3) + (2n - 1)$  (the sum of all odd numbers from 1 to  $2n - 1$ )?

- $n^2$
- $n(n + 1)$
- $n(2n - 1)$
- $3n - 2$

**Problem 98** What is the value of the sum  $\frac{1}{1 \cdot 2} + \frac{1}{2 \cdot 3} + \frac{1}{3 \cdot 4} + \dots + \frac{1}{99 \cdot 100}$ ?

**Problem 99** You have \$1 000 on day 1, and every day you earn 10% of what you already have, so that on day 2 you have  $\$1\,000 + 10\% \cdot \$1\,000 = \$1\,100$ , and on day 3 you have  $\$1\,100 + 10\% \cdot \$1\,100 = \$1\,210$ . When will you have more than \$1 000 000? Feel free to use the code for exponential functions and Bernoulli's inequality that we saw earlier.

**Problem 100** Function  $T(n)$  is defined by  $T(0) = a$ ,  $T(n + 1) = T(n) + b$ . Which of the following is a correct formula for computing  $T(n)$ ?

- $T(n) = a + nb$
- $T(n) = a + (n + 1)b$
- $T(n) = ab^n$
- $T(n) = a + (n - 1)b$
- $T(n) = b + na$

**Problem 101** Find a mistake in the following proof of the statement “For any integer  $n \geq 0$  and any positive integer  $a$ ,  $a^n = 1$ ”.

We are proving this by induction on  $n$ . When  $n = 0$ ,  $a^n = a^0 = 1$ . For the induction step from all numbers 0 through  $n$  to the number  $n + 1$ , we have

$$a^{n+1} = \frac{a^n \cdot a^n}{a^{n-1}} = \frac{1 \cdot 1}{1} = 1,$$

where by the induction hypothesis  $a^n = 1$  and  $a^{n-1} = 1$ .

**Problem 102** Function  $T(n)$  is defined by  $T(0) = a$ ,  $T(n + 1) = b \cdot T(n)$ . Which of the following is a correct formula for computing  $T(n)$ ?

- $T(n) = b + na$
- $T(n) = ab^n$
- $T(n) = a + bn$
- $T(n) = a^n b$

**Problem 103** Function  $T(n)$  is defined as  $T(0) = 0$ ,  $T(2n) = T(n) + 1$ ,  $T(2n+1) = T(n) + 1$ . Which of the following is a correct formula for computing  $T(n)$ ?

- $T(n) = k$ , where  $k$  is the smallest non-negative integer such that  $2^k > n$
- $T(n) = 2^n$
- $T(n) = 2n$
- $T(n) = n$



## 4. Logic

Mathematical logic plays a crucial and indispensable role in creating convincing arguments. We use the rules and language of mathematical logic while writing code, reasoning and making decisions, and using computer programs. In this chapter, we'll learn the basics of mathematical logic, and we'll practice tricky and seemingly counterintuitive, yet logical aspects of mathematical logic. This will help us to write readable and precise code, and to formulate our thoughts rigorously and concisely.

### 4.1 Examples and Counterexamples

In Sections 1.2 and 2.1, we saw that in order to prove existential statements (e.g., “White horses exist”) it suffices to find just one *example* (one white horse). Similarly, *counterexamples* are *examples* showing that a statement is false. Given a statement like “All swans are white”, it is sufficient to provide one *counterexample* (one black swan) to prove that the statement is wrong.



Figure 4.1: One black swan *falsifies* the statement “All swans are white”. (Source: [Wikipedia ↗](#).)

**Problem 104** For which of these statements is one *example* enough to prove them? For which of them is one *counterexample* enough to disprove them?

- All crocodiles are green.
- White lions exist.

- No tiger is green.
- Some cats are white.

One brown crocodile would be a *counterexample* to the first statement. A white lion would be enough to prove the correctness of the second statement. One green tiger would falsify the third statement (such a tiger would be a *counterexample* to this statement). Finally, an *example* of a white cat will prove the last statement.

Let us see a few more examples of such problems. We start with problems in geometry.

**Stop and Think!** All rectangles are squares. Is this statement true?

Clearly, this is not true. A rectangle with side length 1 and 2 is a *counterexample* as it is not a square (squares have four equal sides). Is the opposite statement true?

**Stop and Think!** All squares are rectangles. Is it true?

Yes, this is true, a square by definition is a rectangle with equal sides. Hence, this statement is true, and no counterexample can be found here.

Now, we switch to problems in number theory.

**Problem 105** Does there exist a power of 2 that starts with 65?

It is not difficult to find such a number, either with a computer or even manually. After checking several powers of two, one finds such an *example*:  $2^{16} = 65536$ .

```
for n in range(100):
    if int(str(2 ** n)[:2]) == 65:
        print(f'2**{n}={2 ** n}')
```

```
2**16=65536
```

Here, we use `int(str(2 ** n)[:2])` to compute the first two digits of  $2^n$ . Indeed, the function `str` converts the value of  $2^n$  into a string, then the slice `[:2]` takes the first two characters of this string, and finally the function `int` converts the result back into an integer.

**Problem 106** Does there exist a power of 2 that starts with 17?

In fact, for every integer  $n > 0$ , there is a power of 2 starting with  $n$  (this follows from the fact that  $\log_{10} 2$  is irrational).

**Problem 107** For every integer  $n > 1$ , the number  $n^2 + n + 41$  is prime (is not a product of two smaller positive integers). Is this statement true? Try it: [Coursera](#).

For  $n = 2, 3, 4$ ,  $n^2 + n + 41$  is equal to 47, 53, 61, respectively, which are all primes. In fact, for all  $1 < n < 40$  it holds that  $n^2 + n + 41$  is prime, but  $n = 40$  and  $n = 41$  are *counterexamples*. Indeed, for  $n = 41$ , each of the terms of the sum  $n^2 + n + 41$  is divisible by 41, which makes this sum a product of 41 and another integer. Again, we could find these counterexamples by a simple Python program.

```
def is_prime(n):
    assert n > 0
    if n == 1:
        return False

    for d in range(2, n):
        if n % d == 0:
            return False

    return True
```

```
for n in range(1, 100):
    if not is_prime(n ** 2 + n + 41):
        print(n)
        exit()
```

40

Here, the function `is_prime` checks whether the given positive integer  $n$  is prime simply by checking all of its potential divisors  $2 \leq d < n$  (recall that  $n = 1$  is not prime by convention). We then go through all  $1 \leq n < 100$  and check if  $n^2 + n + 41$  is prime. If it is not, we print  $n$  and stop.

Here's a shorter version of this program.

```
def is_prime(n):
    return n != 1 and all(n % d != 0 for d in range(2, n))

print(next(n for n in range(2, 100) if
           not is_prime(n * n - n + 41)))
```

**Stop and Think!** Do there exist positive integers  $a, b, c$  such that  $a^2 + b^2 = c^2$ ?

Yes, there are (infinitely) many triples of such numbers. One can recall the Pythagorean theorem and a right triangle with side lengths 3, 4, and 5, which gives us the following *example*:

$$3^2 + 4^2 = 9 + 16 = 25 = 5^2.$$

Alternatively, one could write a simple Python program to find such examples.

```
from itertools import combinations

for a, b, c in combinations(range(1, 20), 3):
    if a ** 2 + b ** 2 == c ** 2:
        print(f'{a}**2 + {b}**2 = {c}**2')

3**2 + 4**2 = 5**2
5**2 + 12**2 = 13**2
6**2 + 8**2 = 10**2
8**2 + 15**2 = 17**2
9**2 + 12**2 = 15**2
```

This code enumerates all triples  $1 \leq a \leq b \leq c < 20$  (by using the `combinations` ↗ function from the `itertools` module). For each such triple, it checks whether  $a^2 + b^2 = c^2$  (recall the Python notation `a ** n` for  $a^n$ ).

Now, we change this problem slightly, and use cubes instead of the squares of three numbers.

**Problem 108** Do there exist positive integers  $a, b, c$  such that  $a^3 + b^3 = c^3$ ?

Here, just one example of numbers satisfying the equation would also be sufficient. However, such triples of numbers *do not* exist! One of the most famous results in mathematics, [Fermat's Last Theorem](#) ↗, says that for any integer  $n > 2$  there are no positive integers  $a, b, c$  satisfying

$$a^n + b^n = c^n.$$

While Fermat's Last Theorem was conjectured by Pierre de Fermat in 1637, it was proven only in 1994 by Andrew Wiles! The special case of Fermat's Last Theorem with  $n = 3$  asserts that  $a^3 + b^3 = c^3$  has no solutions in positive integers.



Figure 4.2: Pierre de Fermat (1601–1665). (Source: [Wikipedia](#).)

**Problem 109 – Sum of powers conjecture.** In 1769, Euler conjectured that for any integer  $n > 2$ , one needs at least  $n$  summands to represent the  $n$ th power of a positive integer as a sum of the  $n$ th powers of positive integers. In particular, equations

$$a^4 + b^4 + c^4 = d^4 \text{ and } a^5 + b^5 + c^5 + d^5 = e^5$$

do not have solutions in positive integers. (Hence, Fermat's Last Theorem is a special case of this conjecture for  $n = 2$ .) How can we disprove this conjecture? [Warning: this is rather hard, and serious tools like elliptic curves or a supercomputer were used to find a counterexample for  $n = 4$ . For  $n = 5$ , this is easier and can be done if you optimize the search a bit.]



Figure 4.3: Leonhard Euler (1707–1783). (Source: [Wikipedia](#).)

To disprove Euler's conjecture, it is again enough to give a single counterexample. For example, for  $n = 4$  one may take the following integers:

$$a = 95800, b = 217519, c = 414560, d = 422481.$$

The following code verifies that these numbers indeed form a counterexample. (It is also known that this is the only counterexample where all numbers are less than one million.)

```
print(95800 ** 4 + 217519 ** 4 + 414560 ** 4)
print(422481 ** 4)
```

```
31858749840007945920321
31858749840007945920321
```

For  $n = 5$ , the minimal counterexample has smaller numbers, see Figure 4.4.

**Problem 110 – Fermat numbers.** Fermat conjectured that for every integer  $n \geq 0$ , the number  $2^{2^n} + 1$  is prime. For example,

$$2^{2^0} + 1 = 3, \quad 2^{2^1} + 1 = 5, \quad 2^{2^2} + 1 = 17$$

**COUNTEREXAMPLE TO EULER'S CONJECTURE  
ON SUMS OF LIKE POWERS**

BY L. J. LANDER AND T. R. PARKIN

Communicated by J. D. Swift, June 27, 1966

A direct search on the CDC 6600 yielded

$$27^5 + 84^5 + 110^5 + 133^5 = 144^5$$

as the smallest instance in which four fifth powers sum to a fifth power. This is a counterexample to a conjecture by Euler [1] that at least  $n$   $n$ th powers are required to sum to an  $n$ th power,  $n > 2$ .

REFERENCE

1. L. E. Dickson, *History of the theory of numbers*, Vol. 2, Chelsea, New York, 1952, p. 648.

Figure 4.4: One of the shortest research papers.

are primes. Find the minimal counterexample to this conjecture.

Euler found this counterexample in 1732 — definitely by hand! Feel free to use modern technologies (Python, in particular) for this task!

There is no guarantee that a minimal counterexample to an existential statement about numbers contains relatively small numbers. Let's see two notable examples.

Is it possible to represent 42 as a sum of three (not necessarily positive) cubes? In other words, are there integers  $x, y, z$  such that  $42 = x^3 + y^3 + z^3$ ? An affirmative answer to this question was [found ↗](#) just recently (in 2019!). It took over a million of machine hours to find it, but checking the result is trivial.

```
x = -80538738812075974
y = 80435758145817515
z = 12602123297335631

print(x ** 3 + y ** 3 + z ** 3)
```

42

Another example: how long are the smallest positive integers  $a, b, c$  satisfying

$$\frac{a}{b+c} + \frac{b}{c+a} + \frac{c}{a+b} = 4?$$

About [eighty digits long ↗](#)!

## 4.2 Logic

A *proposition* is a statement that is either true or false. For example, “I'm a human” and “25 is a multiple of 7” are propositions (the first one is true, and the second one is false), while “You should fly to the Moon” is not a proposition. Using the language of mathematical logic, we create compound propositions from simple ones (“ $i \geq 0$  and  $i \leq n$  and  $a[i]$  divides 100, or  $i$  is a prime”). We use such propositions as conditions in `if`-statements and `while`-loops, and, thus, “give directions” to our computer programs.

```
print(3 < 5 and 7 < 5)
print(3 < 5 and not(7 < 5))
print(2+2 == 5 or 2+2 == 4)
```

```
False
True
True
```

Moreover, mathematical logic allows us to make rigorous unambiguous statements, and to simplify and formalize complicated proof arguments. We usually start with simple statements, and create from them more sophisticated ones using logical operations. The four main operations used in mathematical logic are *Negation*, *Logical AND*, *Logical OR*, and *If-Then* (also known as *Implication*).

### Negation

Negation is the simplest logical operation. If we have a statement “All swans are white”, then its negation is “Not all swans are white” or “There are non-white swans”. If a statement  $S$  is “ $2+2=4$ ”, then the negation of  $S$ , denoted by  $\neg S$ , is  $2+2 \neq 4$ . If a statement  $T$  is “ $2+2=5$ ”, then  $\neg T$  — the negation of  $T$  — is  $2+2 \neq 5$ . Note that we can define all four of these statements even though only two of them ( $S$  and  $\neg T$ ) are true. In general, a negated statement is true if and only if the original statement is false.

### Conjunction (AND)

The Logical AND (denoted by  $\wedge$ ) of two statements is true if and only if *both* of the statements are true. For example, the statement “ $2 \cdot 2 = 4$  AND  $2 + 2 = 4$ ” is true precisely because both of the statements “ $2 \cdot 2 = 4$ ” and “ $2 + 2 = 4$ ” are true. “ $5^2 > 10$  AND there are positive integers  $a, b, c$  such that  $a^2 + b^2 = c^2$ ” is true because both parts are true. The following statement is false because one of its parts is false: “There exists a power of 2 that starts with 65 AND  $2 + 2 = 5$ ”.

**Problem 111** Is the following proposition true: “All rectangles are squares, and there is no life outside Earth.”?

We do not know for sure if there is life outside Earth, but luckily we do not need to know this to solve our problem! Indeed, not all rectangles are squares (think of a rectangle with sides 1 and 2), therefore the AND of the two statements is false.

Again, this has a natural counterpart in programming and is known as [lazy evaluation](#). To give an example, consider the following code snippet:

```
def foo():
    print('Foo!')
    return True

if 2 + 2 == 5 and foo():
    print('True')
else:
    print('False')
```

```
False
```

Here, the interpreter knows that the whole `if`-condition evaluates to `False` after evaluating its first part. This is why it does not even bother to call the `foo()` function (so “Foo!” is not printed). Try changing the order of the two statements in the `if`-statement and see what happens!

### Disjunction (OR)

Similarly, the Logical OR (denoted by  $\vee$ ) of two statements is true if and only if *at least one* of them is true. For example, the statement “ $2 + 2 = 5$  OR  $2 + 2 = 4$ ” is true. And the statement “All squares are rectangles OR all rectangles are squares” is true because all squares are indeed rectangles. The statement “ $2 + 2 = 5$  OR 15 is even” is false because both its parts are false.

**Implication (If-Then)**

Finally, the fourth basic logical operation is Implication, or If-Then, denoted by  $\implies$ . Let  $S$  be the statement “If there is an elephant in the refrigerator, then Alice pays Bob \$100”. There are four cases:

- There is indeed an elephant in the refrigerator, and Alice pays Bob \$100. In this case Alice kept her promise, and  $S$  is true.
- There is no elephant in the refrigerator, and Alice does not pay Bob \$100. Alice didn’t promise Bob anything in the case when there is no elephant, so  $S$  is true!
- There is an elephant in the refrigerator, but Alice does not pay Bob. In this case Alice didn’t keep her promise, so  $S$  is certainly false.
- There is no elephant, but Alice still decides to pay Bob \$100. Since  $S$  does not say what Alice has to do if there is no elephant in the refrigerator, she may as well pay Bob. Thus, in this case  $S$  is true, too.

Formally, the statement  $T \implies U$  (which we read “ $T$  implies  $U$ ” or “If  $T$  then  $U$ ”) is true whenever  $U$  is true or  $T$  is false. In other words,  $T \implies U$  is false if and only if  $T$  is true but  $U$  is false. Here are several examples of this:

- “If  $n = 6$ , then  $n$  is even” is true for all  $n$ , because  $n = 6$  indeed *implies* that  $n$  is even. In other words, the only way for this If-Then statement to be false is if  $n = 6$  and  $n$  is not even, which is impossible.
- “If  $n = 5$ , then  $n$  is even” is false for some  $n$ , because for  $n = 5$ ,  $n$  is *not* even.
- “If  $1 = 2$ , then I am an elephant” is true! Indeed, if the If-part of the statement is false, then the If-Then statement is automatically true, because If-Then statements do not put any restrictions on the case when the If-part is not satisfied.
- “If pigs can fly, then  $2 \times 2 = 4$ ” is true. When the hypothesis (“if pigs can fly”) is false, the whole statement is true.

**Problem 112** Is the following statement true: “If pigs can fly, then  $2 \times 2 = 5$ ”?

Again, regardless of the conclusion (“ $2 \times 2 = 5$ ”), if the hypothesis is false, then the implication statement is true. Another way to view this is the following. Consider the statement “If you get caught in the rain, then you’ll get wet”. This statement doesn’t say anything about the case when you don’t get caught in the rain. If you do not get caught in the rain, you may or may not get wet. Thus, if the hypothesis is not satisfied (you don’t get caught in the rain), then the statement is true.

**Problem 113** Is the following statement true: “If all squares are rectangles, then  $2=3$ ”?

**Negations of Operations**

We can apply logical operations to statements containing other operations. Suppose the statement  $S$  is “ $2 + 2 = 4$ ” and the statement  $T$  is “ $2 \cdot 2 = 4$ ”. Now let the statement  $U$  be  $U = S \wedge T$ , that is,  $S$  AND  $T$ . What does the negation of  $U$  mean? Simple formulas for negations of AND- and OR-statements are known as *De Morgan’s laws*. An AND is true if *both* statements are true, so its negation is true if *at least one* of the statements is false. In other words, the negation of an AND is an OR of negations:

$$\neg(S \wedge T) = \neg S \vee \neg T.$$

**Problem 114** What is the negation of the statement “Anna speaks French and German”?

The negation of an AND-statement is an OR of negations. Thus, the negation of this statement will be “Anna does not speak French or Anna doesn’t speak German”.

Similarly, the negation of an OR is an AND of negations:

$$\neg(S \vee T) = \neg S \wedge \neg T.$$



Figure 4.5: Augustus De Morgan (1806–1871). (Source: [Wikipedia](#).)

**Problem 115** What is the negation of the statement “Either he will do it on time, or he won’t get paid”?

We negate each part of this statement, and then connect them with an AND: “He will not do it on time and will still get paid”.

Since the only way to falsify the statement  $T \implies U$  is to have  $T$  true and  $U$  false, the negation of that statement is

$$\neg(T \implies U) = T \wedge \neg U.$$

Let  $S$  be “If there is an elephant in the refrigerator, then Alice pays Bob \$100”. Then  $\neg S$  is “There is an elephant in the refrigerator and Alice does *not* pay Bob \$100”.

**Problem 116** Use De Morgan’s law to simplify the following code:

```
if not ((x <= 5) and (y >= 7)):
```

By De Morgan’s law, the negation of an AND is an OR of negations:

```
if ((x > 5) or (y < 7)):
```

**Problem 117** Use De Morgan’s law to simplify the following code:

```
if not ((x > 9) or (y > 3)):
```

### Quantifiers

Another powerful and expressive tool in creating logical statements is quantifiers. The *universal quantifier* claims that the statement holds for all instances. The following are examples of statements with universal quantifiers:

- All swans are white.
- All integers ending with the digit 2 are even.
- For all integers  $n$ ,  $2 \cdot n$  is even.
- Fermat’s Last Theorem: for all  $n \geq 3$ , the equation  $a^n + b^n = c^n$  does not have solutions with positive integers  $a, b$ , and  $c$ .

Similarly, the *existential quantifier* claims that the statement holds for at least one instance:

- There exist black swans.
- There is a way to get exactly 12 cents with 4-cents and 5-cents coins.
- There exist positive integers satisfying  $a^4 + b^4 + c^4 = d^4$ .
- There is a power of two starting with 65.

**Problem 118** Which of the following statements have universal or existential quantifiers?

- Some cats are black.
- All crocodiles are green.
- There exist white lions.
- No tiger is green.

The first and the third statements merely claim that there exist black cats and white lions, so these are examples of existential statements. The second statement says that all crocodiles are green, and, thus, is a universal statement. Finally, the last statement claims that all tigers are not green, which is a universal statement too.

The universal quantifier is a mathematical analogue of the function `all` in Python that outputs True if all elements in the list are True. Similarly, the existential quantifier is a mathematical formalization of the function `any` that outputs True if at least one of the elements in the list is True. As an example, the following program verifies the following two statements with universal quantifiers: “All integers in {6, 2, 4} are even” and “All integers in {2, 7, 6} are even”.

```
a = [6, 2, 4]
print(all((i % 2 == 0) for i in a))
a = [2, 7, 6]
print(all((i % 2 == 0) for i in a))
```

```
True
False
```

Similarly, the following code evaluates the statements with the existential quantifier “There is an even integer in {1, 7, 9}” and “There is an even integer in {9, 2, 3}”.

```
a = [1, 7, 9]
print(all((i % 2 == 0) for i in a))
a = [9, 2, 3]
print(any((i % 2 == 0) for i in a))
```

```
False
True
```

Most mathematical statements include several alternating quantifiers. A typical theorem may look like this: “There exists an integer  $m$  such that for all integers  $n > m$  the equation  $a^n + b^n = c^n$  has no solutions with positive integers  $a, b$ , and  $c$ .”

The negation of a universal quantifier is an existential quantifier; and the negation of an existential quantifier is a universal quantifier. For example, the negation of “For all  $n$ ,  $A(n)$  is true” is the statement “There exists  $n$ , such that  $A(n)$  is false”. In order to negate the previous theorem, we consequently negate each quantifier in it: “For each integer  $m$ , there exists an integer  $n > m$  such that the equation  $a^n + b^n = c^n$  has a solution with positive integers  $a, b$ , and  $c$ ”.

Mathematicians would say that disproving an existential statement “there exists  $x$  with property  $P$ ” means proving its *negation*, that is, proving that there is no  $x$  with property  $P$ . In other words, we need to show that *all*  $x$  do not have the property  $P$ ; statements of this form (“for all  $x \dots$ ”) are called *universal* statements. There is even special notation for this: the statement “there exists  $x$  with property  $P$ ” is written as  $\exists x P(x)$ , and “all  $x$  do not have property  $P$ ” is written as  $\forall x \neg P(x)$ . The *quantifiers*  $\exists$  and  $\forall$  are read as “there exists” and “for all”, and the *negation* sign  $\neg$  is read as “not”. Using this notation, one can say that  $\neg \exists x P(x)$  is equivalent to  $\forall x \neg P(x)$ .

```

def is_divisible_by_3(x):
    return x % 3 == 0

lst = [5, 17, 6, 10]

print(not any([is_divisible_by_3(x) for x in lst]))
print(all([not is_divisible_by_3(x) for x in lst]))

```

False  
False

In this example, the function `is_divisible_by_3(x)` returns True or False when `x` is (or is not) a multiple of 3. Here, True and False are *Boolean values*. We may want to check whether a list `lst` of integers contains a multiple of 3. For that, we use a special Python construction. First, `[is_divisible_by_3(x) for x in lst]` is the list of Boolean values `is_divisible_by_3(x)` for all elements `x` in the list `lst`. In our example, `lst=[5, 17, 6, 10]`, so this list equals `[False, False, True, False]` (only 6 is divisible by 3). Then, `any(S)` checks whether there exists a True value in `S`, and not reverses the answer. This way, we evaluate the statement “there is no element `x` in the list `lst` that is divisible by 3”. In the next line, we evaluate the statement “all elements of `lst` are not divisible by 3” and get the same answer `False`: there is no good element in the list if and only if all elements are bad.

**Problem 119** Alice says that all white lions weigh more than 100kg. What statement could Bob say to contradict her?

In order to negate this universal quantifier, we use the existential quantifier: “There exists at least one white lion whose weight does not exceed 100kg”.

**Problem 120** Alice says that all elephants are tall and heavy. What statement could Bob say to contradict her?

First, we change the universal quantifier to an existential quantifier, then we negate the AND-statement “elephants are tall and heavy”. Recall that the negation of an AND is an OR of negations. This gives us the following solution: “There exists at least one elephant that is short or light.”

We have seen *De Morgan’s laws*: the negation of an AND is an OR, the negation of an existential statement is a universal statement, the negation of an example is a counterexample. We will use this for proving statements via *Reduction ad Absurdum* in Section 4.3.

### Summary

- One example is enough to prove an existential statement.
- One counterexample is enough to refute a universal statement.
- Negation, AND, OR, and Implication (If-Then) are basic logical operations.
- The negation of an AND is an OR of the negations, and vice versa.
- The negation of a universal statement is an existential statement, and vice versa.

Try to use these techniques to solve the following problems Try it: [Coursera](#).

**Problem 121** Alice (a mathematician specialized in mathematical logic) says that every recursive ordinal is constructive. Bob wants to refute Alice’s claim by giving a counterexample. Which of the following could he provide? (Note that in order to answer this question you do not need to know what an ordinal is, or what it means for it to be constructive or recursive.)

- A constructive ordinal that is not recursive.
- A non-constructive ordinal that is recursive.
- A non-constructive ordinal that is not recursive.

**Problem 122** Which of the following statements are equivalent?

- All recursive ordinals are constructive.
- All constructive ordinals are recursive.
- All non-recursive ordinals are non-constructive.
- All non-constructive ordinals are non-recursive.

**Problem 123** Alice claims that every student in a group knows both French and German. Which of the following sentences asserts that Alice's statement is wrong (no more, no less)?

- All students know at most one of these two languages.
- There is a student who does not know French and does not know German.
- There is a student who does not know French or does not know German (or both).
- Every student who knows French does not know German.

**Problem 124** Alice claims that every student in a group knows French, German, or both. Which of the following sentences asserts that Alice's statement is wrong (no more, no less)?

- All students know at most one of these two languages.
- There is a student who does not know French and does not know German.
- There is a student who does not know French or does not know German (or both).
- Every student who knows French does not know German.

**Problem 125** Alice says that in every region there is a town where all inhabitants are happy. Bob wants to say that Alice is wrong. Which of the following sentences should Bob say?

- There is a region where in all towns at least one inhabitant is unhappy.
- There is a region where there is a town where all inhabitants are happy.
- In every region in all towns all inhabitants are happy.
- In every region there is a town where at least one inhabitant is unhappy.

**Problem 126** Let  $x$  be a number. Consider the following five statements:  $x > 10$ ,  $x > 20$ ,  $x > 30$ ,  $x > 40$ ,  $x > 50$ . Three of these statements are true and two are false. Which ones are true?

**Problem 127** Consider the following statements about a number  $x$ :

1.  $x$  is a multiple of 2.
2.  $x$  is a multiple of 3.
3.  $x$  is a multiple of 6.

Is it possible that for some integer  $x$

- None of the statements are true?
- Exactly one of the statements is true?
- Exactly two of the statements are true?
- All three statements are true?

## 4.3 Reductio ad Absurdum

A natural way to show that a statement is true is to show that its opposite is impossible. This approach is known as *reductio ad absurdum* or *proof by contradiction*. This basic idea is used constantly in combination with other methods.

Reductio ad absurdum is known from ancient times. A variation of it was used in the *Socratic method* of argument (described by Plato around 400 BC). Socrates, in dialogues with his students, revealed contradictions in their beliefs by asking them questions one by one and showing that their beliefs are inconsistent.

**Problem 128** There are boys and girls in a class. Some of them study French, while others study German. Prove that there are a boy and a girl studying different languages. Try it: [Coursera](#).

Let us start by observing that the problem statement assumes that the following conditions hold:

1. There is at least one boy in the class.
2. There is at least one girl in the class.
3. At least one of the students studies French.

4. At least one of the students studies German.

All four conditions are essential: say, if there were no girls in the class, one would not be able to find a girl and a boy studying different languages. The task would also be impossible if one of these languages was not studied by anyone in the class.

To solve the problem, assume that *one cannot find a boy and a girl studying different languages*. This is the opposite of the statement we need to prove. We are going to derive a contradiction from this assumption. This will imply that the statement holds.

To do this, take any girl (there must be at least one!), call her Alice, and assume that she studies French (if she studies German, the argument is analogous). According to our assumption, this must mean that all of the boys in the class study French too. Since there is at least one boy and he studies French, all of the girls must also be studying French, contradicting the assumption that somebody studies German. We assumed that there are no boy and girl studying different languages and reached a contradiction, this implies that our assumption was wrong, that is, there are a boy and a girl studying different languages.

### 4.3.1 Balls in Boxes

**Problem 129** Is it possible to place 30 balls in ten boxes so that all boxes contain a different number of balls? Try it: [Coursera](#).

Perhaps you see an obstacle: in a sense, we have too few balls. To make this formal, assume that the challenge is possible. Let us place the boxes in order of increasing number of balls and let us denote these numbers by  $b_0, \dots, b_9$  (this 0-based indexing, used in programming constantly, will prove useful in this problem):

$$b_0 < b_1 < \dots < b_9.$$

What do we know about  $b_0$ ? Almost nothing:  $b_0$  is non-negative (that is,  $b_0 \geq 0$ ), and that is all we can say. However, for  $b_1$  we can already say something non-trivial: it is greater than  $b_0$ , hence  $b_1 \geq 1$ . Next,  $b_2 \geq 2$ , since  $b_2 > b_1 \geq 1$ . Proceeding in the same fashion, we conclude that  $b_i \geq i$  for every  $i = 0, 1, \dots, 9$ . This means that

$$b_0 + b_1 + \dots + b_9 \geq 0 + 1 + \dots + 9 = 45,$$

contradicting the fact that we have 30 balls. We have found that our task with ten boxes is only possible when we have 45 or more balls to place.

### 4.3.2 Numbers in Tables

**Problem 130** In a sequence of 10 cells, the leftmost one has the number 1 and the rightmost one has the number 30.

|   |  |  |  |  |  |  |  |  |    |
|---|--|--|--|--|--|--|--|--|----|
| 1 |  |  |  |  |  |  |  |  | 30 |
|---|--|--|--|--|--|--|--|--|----|

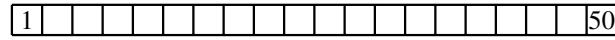
Is it possible to fill the other cells with integers in such a way that consecutive numbers differ by at most 3? Try it: [Coursera](#).

By trying to do this, you'll notice that, in a sense, the numbers grow too slowly to get to 30 by the end. To make this argument formal, note that the second integer must be at most 4. From this, the third one should not exceed 7. Continuing in the same manner, we arrive at the following picture.

|   |          |          |           |           |           |           |           |           |    |
|---|----------|----------|-----------|-----------|-----------|-----------|-----------|-----------|----|
| 1 | $\leq 4$ | $\leq 7$ | $\leq 10$ | $\leq 13$ | $\leq 16$ | $\leq 19$ | $\leq 22$ | $\leq 25$ | 30 |
|---|----------|----------|-----------|-----------|-----------|-----------|-----------|-----------|----|

This reveals that our goal cannot be achieved: the ninth integer is at most 25, hence, it differs from the last integer by more than 3.

**Problem 131** There is a sequence of 20 cells with 1 and 50 at the ends.



We would like to place integers into the cells so that the neighboring cells differ by at most  $k$ . What is the minimum  $k$  that such a construction exists? Try it: [Coursera](#).

Consider a more challenging problem of a similar flavor.

**Problem 132** Is it possible to place the integers  $1, 2, \dots, 64$  on a chessboard in such a way that neighboring cells (sharing a side) differ by at most 4? Try it: [Coursera](#).

In this case, the sequence of cells is not linear, but two-dimensional. Still, one can extend the idea discussed above! Consider the two cells containing 1 and 64. A simple, but crucial observation: *one can reach 64 from 1 in at most 14 moves*. Indeed, at most seven horizontal moves are enough to reach the desired column and at most seven vertical moves are enough to reach the desired row. See Figure 4.6 for an example.

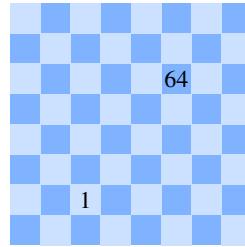


Figure 4.6: In this case, one can reach 64 from 1 in just seven moves: three moves to the right and four moves up. In general, at most 14 moves are required.

And this shows that our task is impossible! Indeed, if we focus on this path from 1 to 64, then the first integer must be at most 5, the second one should be at most 9, and so on. Since there are at most 14 moves on the path, the integer next to 64 on this path should be at most 53 violating the required property. Figure 4.7 shows an unsuccessful attempt to place integers in an extreme case: even if a path from 1 to 64 consists of 14 moves and we always place the largest possible integer (starting from 1), the final integer is too small (at most 53) to differ from 64 by at most 4.

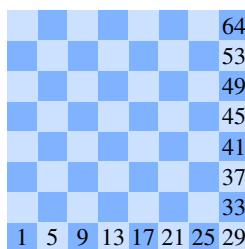


Figure 4.7: An attempt to place integers on a shortest path from 1 to 64. All the neighboring integers differ by four except for the last pair:  $64 - 53 = 11 > 4$ .

### 4.3.3 Pigeonhole Principle

Recall the pigeonhole principle that we discussed in Section 1.2.4: if there are ten pigeons and nine holes, and each pigeon occupies some hole, then some two pigeons share a hole. We used it to give non-constructive proofs of existence (that is, to prove that an object satisfying some property exists without constructing such an object explicitly). It turns out that it can also be used in reductio ad absurdum type arguments to show impossibility (or non-existence). In fact, the pigeonhole

principle itself can be proved using reductio ad absurdum: assume (for the sake of contradiction) that no two pigeons share the same hole; hence, each hole contains at most one pigeon and the number of pigeons is at most nine; since there are ten pigeons, we conclude that our initial assumption is wrong.

**Problem 133 – Antimagic square.** Is it possible to fill a  $3 \times 3$  grid with the integers  $-1, 0, 1$  so that the sums of each row, each column, and both diagonals are different? Figure 4.8 gives an example of an unsuccessful attempt. Try it: [Coursera](#).

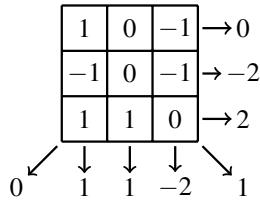


Figure 4.8: An unsuccessful attempt to construct an antimagic square: the sums of the first two columns are the same.

Figure 4.8 suggests the reason why this is impossible: we want eight integers (the sums of three rows, three columns, and two diagonals) to be different, but there are only seven possibilities for the sums:  $-3, -2, -1, 0, 1, 2, 3$ .

**Problem 134** There are  $n$  individuals at a party, and some of them have shaken hands. Prove that there are two individuals who have participated in the same number of handshakes. Figure 4.9 gives an example.

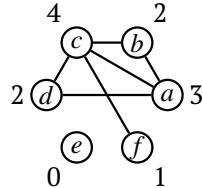


Figure 4.9: There are  $n = 6$  people at a party:  $a, b, c, d, e, f$ . We join two individuals if they shook hands. Near each individual, her number of handshakes is shown. The individuals  $b$  and  $d$  have shaken the same number of hands.

We have  $n$  integers (the number of handshakes for each of  $n$  individuals) and each of these integers may take any of the values  $0, 1, \dots, n - 1$ . The pigeonhole principle does not seem to be applicable here: it is perfectly fine if each of the integers  $0, 1, \dots, n - 1$  appears once among our  $n$  numbers.

But a closer look at the numbers reveals that some two of these numbers cannot appear together.

**Stop and Think!** What are these two integers?

These are the integers  $0$  and  $n - 1$ : the integer  $0$  corresponds to an individual who shook no hands, whereas the integer  $n - 1$  corresponds to an individual who shook all other hands. But these two cases cannot happen simultaneously! Indeed, if an individual shook all other hands, then everybody else shook at least one hand.

This allows us to conclude that there are at most  $n - 1$  different values among our  $n$  integers. The pigeonhole principle implies immediately that some two of these integers are equal.

**Summary**

- Proof by contradiction (or reductio ad absurdum) is a basic argument: to prove that a statement is true, one assumes that negation is true and derives a contradiction.
- This is one of the most popular proof arguments, and it is usually combined with other proof ideas.

**Problem 135** There is a box with socks of five different colors in a dark room. We may take a number of socks from the box, but we don't see their colors. What is the minimum number of socks we need to take to ensure that we get a pair of socks of the same color? Try it: [Coursera ↗](#).

**Problem 136** There are  $2n$  pigeons sitting in  $n$  holes. They are trying to minimise the number of pigeons in the most occupied pigeonhole. What is the minimum value they can achieve? Try it: [Coursera ↗](#).

**Problem 137** There are  $2n + 1$  pigeons sitting in  $n$  holes. They are trying to minimise the number of pigeons in the most occupied pigeonhole. What is the minimum value they can achieve? Try it: [Coursera ↗](#).



## 5. Invariants

“There are things that never change.” Apart from being just a philosophical statement, this phrase turns out to be an important idea in discrete mathematics and computer science. A property that is preserved during a process is called an *invariant*. Invariants are widely used in analyzing the behavior of algorithms, programs, and other processes. In this chapter, we will develop the important skill of finding the right invariant for a problem.

### 5.1 Double Counting

Looking at the same object from various angles can help not only in your everyday life, but also in analyzing mathematical structures.

**Problem 138** Fill a  $3 \times 5$  table with integers so that the sum of each row is equal to 20 and the sum of each column is equal to 10. Try it: [Coursera](#).

Let us see whether we are lucky enough to do this on the first try. To do this, let us fill the first two rows with arbitrary numbers so that the sum in each of them is equal to 20.

|   |   |   |   |   |      |
|---|---|---|---|---|------|
| 2 | 4 | 7 | 3 | 4 | → 20 |
| 5 | 5 | 1 | 6 | 3 | → 20 |
|   |   |   |   |   |      |

Since we want the sum in every column to be equal to 10, we are forced to fill in the last row as follows.

|    |    |    |    |    |      |
|----|----|----|----|----|------|
| 2  | 4  | 7  | 3  | 4  | → 20 |
| 5  | 5  | 1  | 6  | 3  | → 20 |
| 3  | 1  | 2  | 1  | 3  |      |
| ↓  | ↓  | ↓  | ↓  | ↓  |      |
| 10 | 10 | 10 | 10 | 10 |      |

This makes the sum of the last row equal 10. That is, from the eight constraints (three for rows and five for columns) we were able to satisfy all but one. Thus, our first attempt is unsuccessful, but this does *not* mean that the task is impossible: perhaps, if we started with some other numbers in the first two rows, we would be able to fill in the table as needed.

**Stop and Think!** Try filling the first two rows with various numbers so that the sums of the first and the second rows are 20. Then fill the third row so that the sum of each column is 10. What is the sum of the last row in the resulting table?

A closer look at the table reveals that it is not a coincidence that the sum in the last row is 10. Indeed, if the sum of every column is equal to 10, then the sum of all elements in the table is equal to 50. Since the sum of the first two rows is equal to 20, the sum of the last row is  $50 - 20 - 20 = 10$ . And this is why a filling of the table with the required properties does not exist.

Our invariant in this problem is the sum of all elements in the table. And the trick that we used is known as *double counting*: if you count the elements of a set in two different ways, the two results should be the same. (In other contexts, double counting could also refer to a wrong argument where the same object is counted more than once.)

**Problem 139** In a group of 20 students everyone has solved three problems from the homework assignment, and each problem was solved by two students. What is the number of problems in the assignment? Try it: [Coursera ↗](#).

Imagine a summary sheet where the teacher marks which of the problems each student has solved.

|         | 1 | 2 | 3 | 4 | 5 | ... | $k$ |
|---------|---|---|---|---|---|-----|-----|
| Alice   | ✓ | ✓ |   |   | ✓ |     |     |
| Bob     |   | ✓ | ✓ | ✓ |   |     |     |
| Charlie | ✓ |   | ✓ |   | ✓ |     |     |
| :       |   |   |   |   |   |     |     |

In this example, Alice solved problems 1, 2, and 5, whereas Bob solved problems 2, 3, and 4. The number of rows in this table is equal to 20 (the number of students). The number of columns is the (unknown) number of problems and is denoted by  $k$ .

What is the number of marks in the table? On one hand, it is equal to  $20 \cdot 3 = 60$  as each student has solved three problems. On the other hand, it is equal to  $2k$  as each problem has been solved by two students. From  $60 = 2k$  we conclude that  $k = 30$ .

**Problem 140** Everybody in the class got the same assignment. It turned out that every student solved more than half of all problems. Is it possible that at the same time each problem was solved by fewer than half of the students? Try it: [Coursera ↗](#).

**Problem 141** In a group of 27 students every girl knows four boys and every boy knows five girls. Find the number of boys in the group. Try it: [Coursera ↗](#).

To solve this problem, you may want to draw a  $b \times g$  table where  $b$  and  $g$  are the (unknown) numbers of boys and girls, respectively. In each cell of this table, put a mark if the corresponding boy and girl know each other. Then, compute the number of marks in two different ways.

**Problem 142** Four chess players play a series of tournaments. In each tournament, the player that finishes in the first place receives \$4 000 of prize money, the player that finishes in the second place receives \$2 000, and the player that finishes in the third place receives \$1 000. The player in last does not receive anything. After the series, the total prizes won by each player are \$13 000, \$10 000, \$7 000 and \$5 000 respectively. How many tournaments have they played? Try it: [Coursera ↗](#).

**Problem 143** Prove that for any integer  $n \geq 0$ ,

$$1 + 3 + 3^2 + \cdots + 3^n = \frac{3^{n+1} - 1}{2}.$$

Perhaps the easiest way to prove this formula is by means of mathematical induction, but we will prove it by double counting here. Denote the sum by  $S_n$ . How can we get  $S_{n+1}$  from  $S_n$ ? On one

hand, it suffices to add  $3^{n+1}$ :  $S_{n+1} = S_n + 3^{n+1}$ . On the other hand, one can multiply  $S_n$  by 3 and add 1:  $S_{n+1} = 3 \cdot S_n + 1$ . Hence,  $S_n + 3^{n+1} = 3 \cdot S_n + 1$ . Thus,  $S_n = \frac{3^{n+1}-1}{2}$ .

**Problem 144 – sum of geometric series.** Use the trick above to derive the formula for the sum of geometric series: for any  $a \neq 1$  and any integer  $n \geq 0$ ,

$$1 + a + a^2 + \cdots + a^n = \frac{a^{n+1} - 1}{a - 1}.$$

## 5.2 Searching for Invariants

In the previous section, we considered quantities that were invariant for straightforward reasons: no matter how one counts the elements of a set, the result is the same. In this section, we will consider problems where discovering the right invariant is a more challenging task.

**Problem 145** There are two cups: cup 1 contains coffee and cup 2 contains milk. We take a spoon of coffee from cup 1 and pour it into cup 2. After that, we take one spoon of the drink in cup 2 and pour it into cup 1. What is larger, the amount of milk in cup 1 or the amount of coffee in cup 2? Try it: [Coursera](#).

It might seem that one needs to know the volumes of both cups as well as the volume of the spoon to answer the question, but it is not the case! What is important here is that the amount of drink in both cups remains the same (this is our invariant!). This means that the amount of coffee missing in cup 1 is the same as the amount of milk added to cup 1. And the amount of coffee in cup 2 is the amount of coffee missing in cup 1. Hence, the two quantities from the problem statement are equal.

**Problem 146** There are two equally sized cups: cup 1 contains coffee and cup 2 contains milk. Both cups are half full (we are optimists). Your favorite drink is  $1/3$  coffee and  $2/3$  milk. Can you get such a drink in cup 1 by transferring (any amount of) liquid between the two cups? Any amount of your favorite drink would work — the right proportion is what matters. Try it: [Coursera](#).

We are going to show that, at any point of time, *at least half of the drink in cup 1 is coffee* (yes, this is our invariant!). Since the total amounts of coffee and milk are equal (and will remain equal after any number of pourings), this is the same as saying that *at least half of the drink in cup 2 is milk*. And this implies that we cannot achieve our goal: no matter what we do, cup 1 will always contain at least as much coffee as milk.

To show that this property holds, we start by noting that it clearly holds in the very beginning: cup 1 contains coffee only whereas cup 2 contains milk only. Now, consider the two allowed operations. When we pour drink from cup 1 to cup 2, the proportion of milk and coffee in cup 1 stays the same. Hence, cup 1 still contains at least as much coffee as milk. And when we pour drink from cup 2 to cup 1, the proportion in cup 2 does not change. This means that cup 2 still contains at least as much milk as coffee and this, in turn, means that cup 1 contains at least as much coffee as milk.

**Problem 147** Bob is debugging his code. When he starts, he has only one bug. But once he fixes one bug, three new bugs appear. Last night Bob fixed 15 bugs. How many pending bugs does Bob have now? Try it: [Coursera](#).

Let us model a few first steps of this (not so complicated) process.

1. Initially, Bob has one pending bug. He fixes it and introduces three new bugs.
2. Bob has three pending bugs. He fixes one and gets three new bugs.
3. Bob has five pending bugs. He fixes one of them, but this gives rise to three fresh bugs.
4. Bob has seven pending bugs...

We notice that the number of pending bugs increases by two at each step. The corresponding invariant is:

$$\#pending = 1 + 2 \cdot \#fixed.$$

Hence, the answer to this problem is  $1 + 2 \cdot 15 = 31$ .

It is also straightforward to model this process in Python.

```
number_of_pending_bugs = 1

for _ in range(15):
    number_of_pending_bugs -= 1
    number_of_pending_bugs += 3

print(number_of_pending_bugs)
```

31

**Problem 148** Consider the following two-part problem.

1. You have 10\$ in each of ten different banks. You would like to merge all accounts into one. Each day, you pick two banks where you still have money left and move all the money from one bank to the other. How many days do you need to move all your money to one account?
2. Suppose now that you are charged 1\$ each time you merge two accounts. How many dollars will you have left when you move all your money to the same account?

Try it: [Coursera](#).

### 5.3 Termination

In this section, we use invariants to show that two natural processes terminate quickly. Later we show how to use similar ideas and invariants for estimating the running time of algorithms.

**Problem 149** There are two football teams in a town. Each citizen supports one of the teams. If among someone's friends there are more fans of the other team than of her own, she switches to support the other team. Each day, one such citizen switches. Is it possible that this switching process goes on forever? (For the sake of this problem, we assume that friendship is always mutual and that both the population and the friendship networks do not change.) Try it: [Coursera](#).

Figure 5.1 shows a toy example with seven citizens. Two individuals are friends if they are

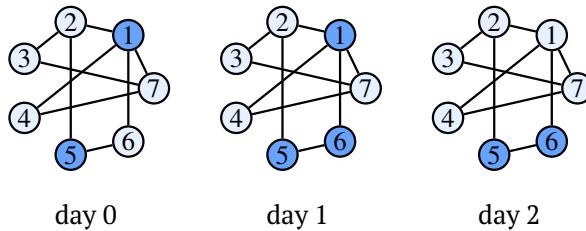


Figure 5.1: A toy example of switching teams: on the first day, individual 6 switches; on the second day, individual 1 switches.

connected by a segment in the picture (for example, individuals 3 and 7 are friends whereas individuals 2 and 6 are not). Initially (day 0), individuals 1 and 5 support the first team (call it dark) and all others support the second team (call it light). On the first day, individual 6 realizes that all of her friends (individuals 1 and 5) support the dark team and starts to support it, too. On the second day, individual 1 switches to support the light team as he has three friends supporting the light team (individuals 2, 4, and 7) whereas only one of his friends (individual 6) supports his favorite team. Then, no more further switches are possible and the process stops.

It seems plausible that for some larger size of the population and some other friendship connections the process can go forever. Below, we show that in fact this is impossible. To do this,

we find a quantity that decreases each time an individual switches. We want this quantity to be restricted to non-negative integers so that it cannot decrease for infinitely long.

**Stop and Think!** Do you see such a quantity?

Let us call a friendship connection *shaky* if the two friends support different teams. Figure 5.2 repeats Figure 5.1 with shaky connections shown dotted. As you see, the number of shaky connec-

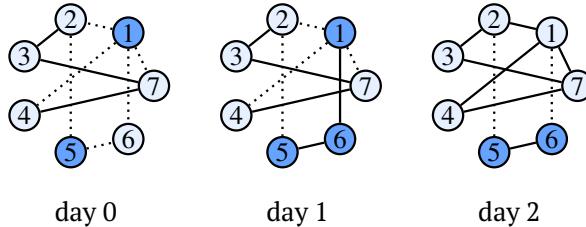


Figure 5.2: Population from Figure 5.1 with shaky connections shown dotted.

tions drops after each day: initially (day 0), it is equal to six, on the first day it decreases to four, on the second day it is equal to two. It turns out that this is not a coincidence: for any population and any friendship network, the number of shaky connections decreases whenever an individual switches.

To show this, consider an individual that is about to switch. We know the reason why she is going to switch: the number of her friends supporting the opposite team is greater than the number of her friends supporting her favorite team. In other words, before the switch, she has more shaky connections than she will have after it. See Figure 5.3.

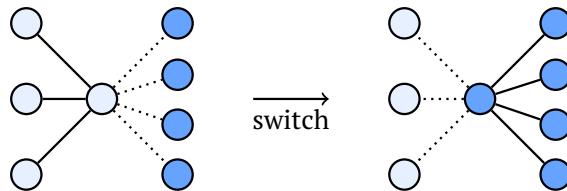


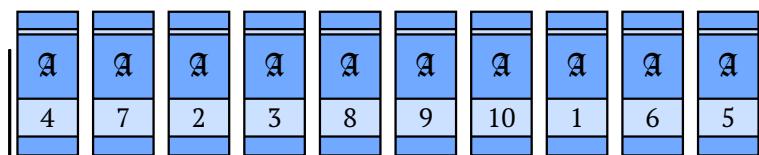
Figure 5.3: When an individual switches, the number of shaky connections decreases.

To complete this argument, it remains to note that the number of shaky connections is a non-negative integer. This means that it cannot decrease for infinitely long and this, in turn, allows us to conclude that the switching process cannot continue forever.

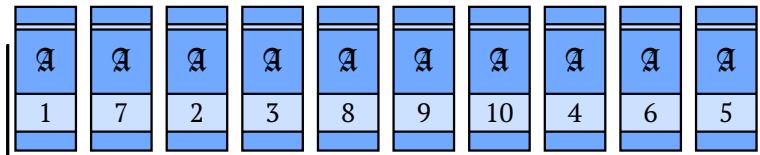
In Section 7.4, we provide a program that visualizes the switching process for any initial population. We encourage you to play around with its parameters (size of population, number of friends) and see what happens.

In the following problem, we will use invariants twice — to show that something can be done efficiently and to show that there also exists a limit to how efficiently it can be done.

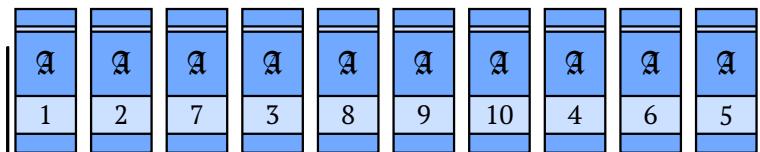
**Problem 150** King Arthur has a shelf of his works consisting of 10 volumes, numbered 1, 2, ..., 10. Over the years of use, the volumes got disordered. Arthur hires Merlin to sort the collection, but he doesn't want more than two volumes to leave the shelf at once. The volumes are heavy, so Merlin can only switch two volumes on the shelf in a day. How many days does Merlin need to surely sort the volumes regardless of their initial position? Try it: [Coursera](#).



Let us show that nine days are enough for any initial permutation of the volumes. On the first day, Merlin finds the position  $j$  of volume 1 (in the example above,  $j = 8$ ). If  $j \neq 1$ , he exchanges the volumes at the positions 1 and  $j$ .



On the second day, he finds the position  $j$  of volume 2. Note that  $j$  cannot be equal to 1, since the first position is occupied by volume 1. If  $j \neq 2$ , Merlin exchanges the books at positions 2 and  $j$ .



Proceeding in the same fashion, on day  $i$ , Merlin ensures that volume  $i$  moves to position  $i$ . This way, he maintains the following *invariant*: after  $i$  days, the first  $i$  volumes occupy their intended positions. It remains to note that by the end of the ninth day, volume 10 must be at position 10: volumes 1, 2, ..., 9 occupy positions 1, 2, ..., 9, hence 10 is the only available position for volume 10.

It is particularly easy to implement this strategy in Python. The code below uses 0-based indexing for days, books, and positions.

```
def sort_books(books):
    day = 0

    for i in range(len(books)):
        j = books.index(i)
        if j != i:
            books[i], books[j] = books[j], books[i]
            print(f'After day {day}: {books}')
            day += 1
```

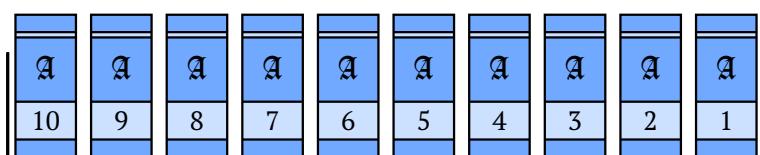
```
sort_books([0, 5, 8, 1, 2, 3, 7, 4, 9, 6])
```

```
After day 0: [0, 1, 8, 5, 2, 3, 7, 4, 9, 6]
After day 1: [0, 1, 2, 5, 8, 3, 7, 4, 9, 6]
After day 2: [0, 1, 2, 3, 8, 5, 7, 4, 9, 6]
After day 3: [0, 1, 2, 3, 4, 5, 7, 8, 9, 6]
After day 4: [0, 1, 2, 3, 4, 5, 6, 8, 9, 7]
After day 5: [0, 1, 2, 3, 4, 5, 6, 7, 9, 8]
After day 6: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

This example reveals the following property: at least some permutations (orderings) can be sorted in fewer than nine days.

**Stop and Think!** Are there permutations of ten books that cannot be sorted in fewer than nine days?

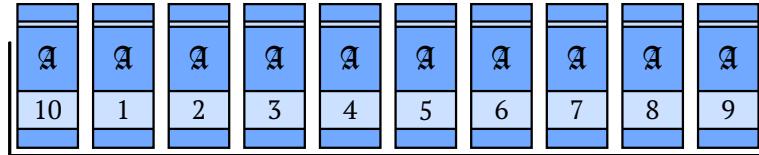
A natural first guess would be to consider the volumes sorted in the reverse order.



However, it can be sorted in just five days:

$$10 \leftrightarrow 1, \quad 9 \leftrightarrow 2, \quad 8 \leftrightarrow 3, \quad 7 \leftrightarrow 4, \quad 6 \leftrightarrow 5.$$

Consider the following permutation.



Try to sort it manually or to run the code above on it (don't forget to subtract one from the volume numbers, as the code assumes 0-based indexing). Note that if you (and the function `sort_books`) are not able to sort it in fewer than nine days, this *does not* mean that it is impossible. Indeed, Merlin's strategy (implemented above) is to place book  $i$  correctly on day  $i$ , for  $i = 1, 2, \dots, 9$ . However, there are many other strategies: say, on the first day, we may want to place book 7 correctly, then book 2, and so on. What if some of these strategies allow for sorting the above sequence in fewer than nine days? Below, we exploit invariants to prove formally that no such strategy exists.

Consider the following integer measure of a permutation  $\pi$ :  $\text{inv}(\pi)$  is the number of elements located to the right of their intended positions. The following example highlights such elements.

$$\begin{aligned}\text{inv}([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]) &= 0, \\ \text{inv}([10, 9, 8, 7, 6, 5, 4, 3, 2, 1]) &= 5, \\ \text{inv}([3, 1, 4, 8, 6, 10, 5, 9, 2, 7]) &= 4, \\ \text{inv}([10, 1, 2, 3, 4, 5, 6, 7, 8, 9]) &= 9.\end{aligned}$$

Roughly speaking, all highlighted elements should move to the left and all the remaining elements should either move to the right or stay unchanged. Since each exchange moves one element to the left and one element to the right, the number of days required should be at least the number of highlighted elements. We make this formal below.

We claim that a permutation  $\pi$  cannot be sorted in fewer than  $\text{inv}(\pi)$  days. Indeed, when  $\pi$  is sorted,  $\text{inv}(\pi) = 0$  (when all volumes occupy their intended places, no volume stands to the right of its intended position). Each day, one book moves rightward and one book moves leftward. Only the latter book may contribute to reducing the value of  $\text{inv}(\pi)$ , so the value of  $\text{inv}(\pi)$  drops by at most one.

All of this allows us to conclude that

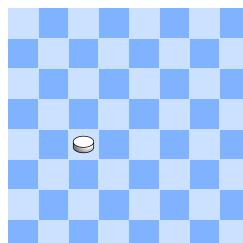
$$\pi = [10, 1, 2, 3, 4, 5, 6, 7, 8, 9]$$

cannot be sorted in fewer than nine days.

## 5.4 Even and Odd Numbers

Recall that integers divisible by 2 are called *even* (examples: 6,  $-8$ , 0), while all others are called *odd* (examples:  $-3$ , 7, 19). This basic property of integers, called *parity*, finds applications in many problems.

**Problem 151** A piece on a chessboard can move up, down, left, or right. Can it return to the original position after 17 moves? What about 18 moves? Try it: [Coursera](#).



The first thing to note is that the piece can return to its original position in principle: just move one step (say, to the right) and then move back. This means that returning back in 18 steps is also possible: repeat the previous strategy nine times.

For the case of 17 steps the same argument does not work: 17 is an odd number. As usual, this *does not* mean that this is impossible in general: it is still not excluded that there exists some other strategy.

You may notice that with each move, the color of the current cell changes. If the piece starts on a light cell, in one step it will be on a dark cell, in two steps it will be on a light cell again, and so on. Hence, after 17 steps it will be on a dark cell for sure. And this is why it cannot return back to the original cell in 17 steps.

Part of the reason why parity of numbers is so useful is that parity behaves nicely under arithmetic operations. This behavior is simple and natural. We summarize it in the following problem.

**Problem 152** Consider the following operations.

- Suppose  $a$  and  $b$  are even. Is their sum  $a + b$  even or odd?
- Suppose  $a$  is even and  $b$  is odd. Is their sum  $a + b$  even or odd?
- Suppose  $a$  and  $b$  are odd. Is their sum  $a + b$  even or odd?
- Suppose  $a$  and  $b$  are even. Is their product  $a \cdot b$  even or odd?
- Suppose  $a$  is even and  $b$  is odd. Is their product  $a \cdot b$  even or odd?
- Suppose  $a$  and  $b$  are odd. Is their product  $a \cdot b$  even or odd?

Try it: [Coursera](#) ↗

Note that the color of a cell on a blackboard also depends on the parity of a particular integer. To see this, let us number the rows and the columns of the board. Then, a cell is dark if and only if the sum of its row and column numbers is even (see Figure 5.4).

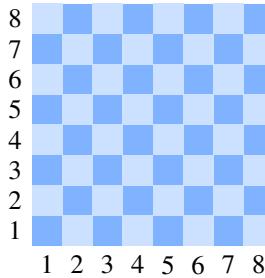


Figure 5.4: The cell  $(2, 3)$  is light, since  $2 + 3 = 5$  is odd. The cell  $(3, 5)$  is dark, since  $3 + 5$  is even.

Now, let's consider a more challenging problem.

**Problem 153** Is it possible to place signs in the expression

$$\pm 1 \pm 2 \pm 3 \pm 4 \pm 5 \pm 6 \pm 7 \pm 8 \pm 9$$

to get 100 as a result? What about 2? Try it: [Coursera](#) ↗.

Clearly, one gets the largest number by using all + signs. The value in this case is

$$1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 = 45.$$

Hence, one definitely cannot get 100.

What about getting 2 as a result? Note that our expression contains four even numbers and five odd numbers. This means that no matter how one puts the signs, the result will be odd. Hence, it is impossible to get 2.

Now, it is natural to ask the following question.

**Problem 154** What integers can be obtained by placing signs in the expression

$$\pm 1 \pm 2 \pm 3 \pm 4 \pm 5 \pm 6 \pm 7 \pm 8 \pm 9?$$

We've discussed already that if an integer can be obtained, then it must be between  $-45$  and  $45$ , and it must be odd. Note however that we haven't proved that every such number can be obtained: perhaps, there are other obstacles. It turns out that there are no other obstacles: every odd integer from  $-45$  to  $45$  can be obtained. This requires a proof, of course.

**Problem 155** Prove that every odd integer from  $-45$  to  $45$  can be obtained by placing signs in the expression

$$\pm 1 \pm 2 \pm 3 \pm 4 \pm 5 \pm 6 \pm 7 \pm 8 \pm 9.$$

As a hint, consider the following code that tries to represent the given value by placing signs in the expression  $1 \pm 2 \pm \dots \pm n$ .

```
def represent(n, value):
    if n == 0 and value == 0:
        return []
    total = sum(range(1, n + 1))

    if abs(value) > total or (total - value) % 2 != 0:
        return False

    if value >= 0:
        return represent(n - 1, value - n) + [n]
    else:
        return represent(n - 1, value + n) + [-n]

for v in (7, 15, 22, 33, 40, 47):
    print(v, end=' ')
    result = represent(9, v)
    if not result:
        print(' is not representable')
    else:
        print('=', end=' ')
        for i in result:
            if i < 0:
                print(f'{i}', end=' ')
            else:
                print(f'+{i}', end=' ')
        print()
```

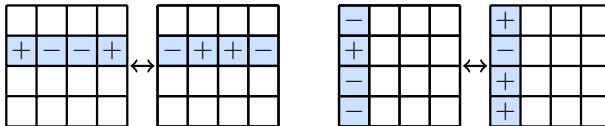
```
7=+1-2-3+4+5-6+7-8+9
15=-1-2+3+4-5+6-7+8+9
22 is not representable
33=+1-2+3-4+5+6+7+8+9
40 is not representable
47 is not representable
```

This code proceeds recursively. This suggests that you may want to solve the above problem as follows: first, generalize the statement (so that it states something about every positive integer  $n$  rather than just  $n = 9$ ), then prove the generalized statement by induction on  $n$  (recall Section 3.2.4).

**Problem 156** All cells of a  $4 \times 4$  table contain plus signs except for the top left cell that contains a minus sign.

|   |   |   |   |
|---|---|---|---|
| - | + | + | + |
| + | + | + | + |
| + | + | + | + |
| + | + | + | + |

In each step, one is allowed to switch all of the signs in any row or column.



Is it possible to switch all signs to '+'? Try it: [Coursera](#).

As you may guess, the parity is going to help here. In particular, the parity of the number of minuses never changes. Indeed, if we switch all signs in a row containing  $k$  minuses, we get  $4 - k$  minuses. Hence, the number of minuses changes by  $k - (4 - k) = 2k - 4$ , an even number. (In the first example in the problem statement, the number of minuses does not change, in the second one, it changes by 2.) Hence, the parity of the number of minuses does not change. We conclude that it is impossible to get a configuration with no minuses.

After this warm up, consider a more challenging version of the problem.

**Problem 157** Is it possible to switch all signs to '+' in the following table, where you once again can only flip rows or columns in their entirety?

|   |   |   |   |
|---|---|---|---|
| - | + | + | - |
| + | + | + | + |
| + | + | + | + |
| - | + | + | - |

Try it: [Coursera](#).

The previous argument does not seem to work here as the number of minuses is even in both the initial and the final configurations.

**Stop and Think!** It turns out that the previous argument (tracking the parity of the number of minuses) can, in fact, be applied here. As usual, we encourage you to try to figure this out before reading further.

The trick is to focus on the top left  $2 \times 2$  subtable.

|   |   |   |   |
|---|---|---|---|
| - | + | + | - |
| + | + | + | + |
| + | + | + | + |
| - | + | + | - |

In the original configuration, this subtable contains a single minus sign whereas in the end we want it to contain no minus signs. This is impossible by the previous argument! Indeed, when we switch, say, a row in the  $4 \times 4$  table, we also switch a row in this  $2 \times 2$  subtable or do nothing (if the switched row is one of the bottom two rows). Hence, the parity of the number of minuses is preserved in the  $2 \times 2$  subtable, and this is why the challenge is impossible.

After discussing two versions of this puzzle, it is natural to ask the following general question.

**Problem 158** Call a configuration *solvable*, if it is possible to switch all its signs to pluses. What configurations are solvable?

We've already seen one obstacle: if some  $2 \times 2$  subtable of a configuration contains an odd number of minuses, then it is unsolvable. It turns out that it is the only obstacle!

**Problem 159** Prove that if every  $2 \times 2$  subtable of a configuration contains an even number of minuses, then one can switch all signs to pluses.

To do this, prove two things:

- for any initial configuration, it is possible to obtain pluses only in the top row and left column;

|   |   |   |   |
|---|---|---|---|
| + | + | + | + |
| + |   |   |   |
| + |   |   |   |
| + |   |   |   |

- if every  $2 \times 2$  subtable of the initial configuration contains an even number of minuses, then, the previous step results in the required (all-pluses) configuration.

**Problem 160** Consider the standard coloring of an  $8 \times 8$  chessboard. In one step, one is allowed to pick a  $2 \times 2$  subboard and switch the colors of all its cells. Is it possible to get a board containing only a single cell of a particular color? Try it: [Coursera](#).

#### Summary

- Invariants are important tools for proving impossibility, termination, and various bounds.
- Invariants may take many forms: numbers, “parity”, equations, inequalities.
- To prove impossibility, one finds a quantity that never changes during a process.
- To prove that a process terminates in a number of steps, one usually finds a quantity that decreases at every step.
- Double counting is a method that uses the sum invariant.



## 6. Project: 15-Puzzle

In this chapter, we consider the well known 15-puzzle where one needs to place 15 square pieces in a square box. It turns out that whether this puzzle can be solved or not is determined by mathematics: it is solvable if and only if the corresponding permutation is even. To understand what this means and why this is true, we will learn basic properties of even and odd permutations — an important tool in algebra and discrete mathematics. We will implement a number of simple methods for dealing with permutations. We will then use them as building blocks for a program that solves any configuration of this game in the blink of an eye!

### 6.1 The Puzzle



Figure 6.1: A photo of a 15-puzzle from [Wikipedia](#). You can see pins that prevent the pieces from being taken out from the box.

Take a square  $4 \times 4$  box and put 15 square pieces numbered  $1, 2, \dots, 15$  (Figure 6.1) in it. Each piece is of size  $1 \times 1$ , so together the pieces occupy 15 places (cells) out of 16, and one cell remains empty. We can move the pieces as follows: a piece neighboring the empty cell moves into this cell, and its old position is vacated. The goal of this game is to rearrange the cells in some other order. For example, one may try to apply a sequence of moves transforming the left configuration from Figure 6.2 into the right configuration from Figure 6.2.

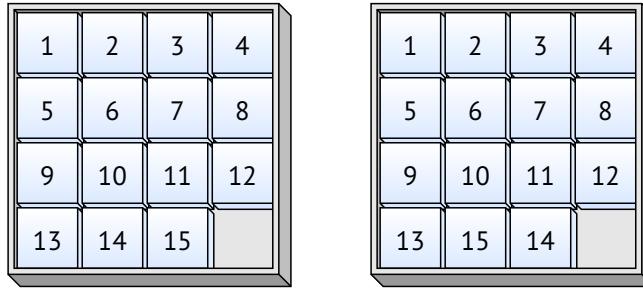


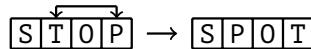
Figure 6.2: Can you transform the left configuration into the right one by several moves (not taking the pieces out)? The [Wikipedia](#) article says that once it was a \$1000 question.

Equivalently, we may go backwards and ask you to restore the initial configuration (left) from the configuration where 14 and 15 are exchanged (right). This is an equivalent task (just reverse the sequence of the moves like in a video played backwards), that is more convenient in practice since you do not need to memorize the target configuration. Try it: [Coursera](#).

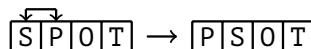
In fact, a sequence of moves required in Figure 6.2 does not exist at all! Thus, those who promised to pay for a solution took no risk. But how can we prove this? This will be done later in this chapter, but first we should consider a more general object, *permutations*.

## 6.2 Permutations and Transpositions

Let us start with a sequence of letters (string), say, STOP. We can *exchange* two letters, for example, T and P. Then we get a string SPOT.



Now, we can exchange some other pair, for example, S and P, and get PSOT.



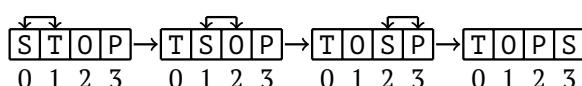
An exchange of two letters is called a *transposition*.

**Stop and Think!** Can you make one more transposition to get POST?

This is easy: exchange S and 0.

**Stop and Think!** Can you convert STOP to TOPS by a sequence of transpositions (pair exchanges)?

Here is one such sequence.



We number the positions from left to right as 0, 1, 2, 3. Then, each transposition can be conveniently specified by two integers — indices of the swapped symbols. In particular, the transpositions applied above are

$(0,1)$ ,  $(1,2)$ ,  $(2,3)$ .

The code below applies a sequence of transpositions to a string.

```

def apply(lst, trans):
    first, second = trans
    lst[first], lst[second] = lst[second], lst[first]

word = ['S', 'T', 'O', 'P']
transpositions = [(0, 1), (1, 2), (2, 3)]

for transposition in transpositions:
    apply(word, transposition)

print(word)

```

['T', 'O', 'P', 'S']

Here, we represent a sequence of objects to be permuted (letters in a string) as a Python list. For example, the string STOP is represented as a list of four elements 'S', 'T', 'O', and 'P' (one could use a string 'STOP', but strings are immutable in Python). This list is stored in the variable word, so, for example, word[2] is 'O' (recall that the indexing is 0-based).

The first argument of the function apply is a list, and the second argument is a pair of integers — the positions to be exchanged (in Python terms, pairs are “tuples” made of two elements). The line first, second = trans unpacks the pair trans and copies the two integers that constitute this pair into variables first and second. Note that we do not check whether first and second are different and the call apply(lst, (0, 0)) will not cause any error (as long as lst is not empty).

**Stop and Think!** What does the call apply(lst, (0, 0)) do?

In fact, nothing: the line lst[0], lst[0] = lst[0], lst[0] is perfectly legal in Python and doesn’t change the values in lst (the new value lst[0] coincides with the old value).

However, when we talk about transpositions, we do *not* allow such “degenerate” transpositions: the *identity permutation* where all objects remain at their places, is *not* a transposition.

Finally, the for loop applies sequentially all the transpositions from the list transpositions to word; then we print the new (permuted) value of word.

**Stop and Think!** What happens with STOP after a sequence of transpositions [(1,3),(0,1)]?

It is exactly the sequence of transpositions we started with: STOP → SPOT → PSOT.

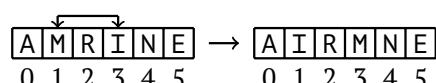
Let us emphasize again that we encode a transposition by a pair of *positions* of exchanged letters (and not by the pair of letters).

**Problem 161** Find a sequence of transpositions that converts MARINE to AIRMEN. The positions are numbered 0, 1, 2, 3, 4, 5 from left to right. Try it: [Coursera](#).

A systematic way to find the required sequence of transpositions is to fill positions correctly one by one, first making position 0 correct, then position 1 correct, etc. In our example, we first put A into position 0, i.e., exchange letters at positions 0 and 1.



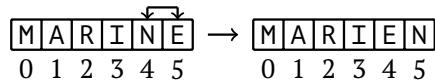
Then we fill position 1 with letter I by applying the transposition (1,3):



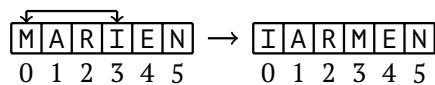
Now we are lucky: positions 2 and 3 are already filled correctly by R and M. But position 4 contains N instead of E, so we need to apply transposition (4, 5). After that, position 4 contains E as required. Moreover, the last position 5 is also filled correctly: N was moved there from position 4, so nothing else is needed.

**Stop and Think!** Is the solution unique? Are there other sequences of transpositions that also transform MARINE to AIRMEN?

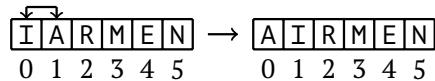
Sure, there are other sequences. For example, we could go from right to left and first make the *last* position correct:



Then we make position 3 correct:



Position 2 is already correct, so we make position 1 correct:



**Stop and Think!** Can you represent this sequence of transpositions in our Python notation?

Answer: `[(4,5),(0,3),(0,1)]`. There are other equivalent answers; for example, `[(5,4),(0,3),(1,0)]` represents the same sequence of transpositions, since the ordering in pairs does not matter.

There are many other sequences of transpositions that convert MARINE to AIRMEN. For example, one could append some transposition twice at the end; such two transpositions cancel out.

**Problem 162** Prove that *any* permutation can be obtained by a sequence of transpositions.

In fact, our approach to the AIRMEN example was general enough and it will work for every permutation. We fill all the positions one by one in some order (in our example, from left to right), until all the positions are filled correctly.

More formally, we prove by induction that the first  $k$  positions can be filled correctly. For  $k = 0$ , this statement is (vacuously) true. Induction step: assume that positions  $0, 1, \dots, k - 1$  (there are  $k$  of them) are already filled correctly. We need to fill the next position, i.e., position  $k$ , by a correct letter. It may happen that it is already filled correctly. Then we have nothing to do at this step. If position  $k$  is occupied by a wrong letter, we exchange this wrong letter with the letter we want to see at position  $k$ . After this transposition, the position  $k$  is filled correctly. (End of the induction step.)

Mathematicians would say that in the *permutation group* every permutation is a *product* of some transpositions.

**Stop and Think!** Is this argument convincing? If it is, could you tell what happens at the last step when we need to fill the last position correctly? And why don't we destroy any of the previous positions  $0, 1, \dots, k - 1$  while filling the  $k$ -th position with a correct letter?

The first question is easy. At the last step, all the positions, except for the last one, are filled by correct letters. So only one letter and only one position remain, and there is no choice.<sup>1</sup>

<sup>1</sup>If a group of people get back their passports after a passport control, and everybody except one person have checked that they got the correct passport, then this last person does not need to worry either.

The second question: assume that positions  $0, 1, \dots, k - 1$  are filled correctly. We need to put the correct letter in the  $k$ -th position if it is not already there. Can we destroy positions  $0, 1, \dots, k - 1$  while doing this? No, because they are filled by the letters whose target positions are  $0, 1, \dots, k - 1$ , and we consider a letter whose target position is  $k$ , and its current position is also greater than  $k - 1$ .

**Problem 163** Prove that any permutation of  $n$  objects can be obtained by at most  $n - 1$  transpositions.

**Problem 164** Give an example of a permutation of 4 objects that cannot be obtained by 2 or fewer transpositions. Give an example of a permutation of 10 objects that cannot be obtained by 8 or fewer transpositions.

Note that in this problem it is not enough to provide an example. We have to prove that *this permutation has the required property* (cannot be obtained by a small number of transpositions). For the first part of the problem (2 or fewer transpositions) it is easy to check all the possibilities, but for the second part (8 or fewer transpositions) some general argument is needed, and this is not so easy.

**Problem 165** Implement a Python function

```
transform(first, second)
```

that takes two permutations of  $\{0, 1, \dots, n - 1\}$  (i.e., two lists containing each integer from  $\{0, 1, \dots, n - 1\}$  exactly once, in any order) and returns a list of transpositions that transforms the first list into the second one.

A solution to this problem is shown below.

```
def transform(first, second):
    assert len(first) == len(second)
    n = len(first)
    assert set(first) == set(range(n))
    assert set(second) == set(range(n))

    transpositions = []
    current = list(first)

    for i in range(n):
        if current[i] != second[i]:
            idx = current.index(second[i])
            assert idx != i
            transpositions.append((i, idx))
            current[i], current[idx] = \
                current[idx], current[i]

    assert current[i] == second[i]

    return transpositions

print(transform([3, 4, 0, 2, 1], [0, 1, 3, 4, 2]))
[(0, 2), (1, 4), (3, 4)]
```

First, we check that both input lists are indeed permutations of  $0, 1, \dots, n - 1$  for some  $n$ . Then, we prepare the `transpositions` list that is empty initially; the transpositions will be added to it one by one. We also make a copy of the list `first` and call it `current` (note that it is essential to call the method `list()` for this).

Then, for  $i = 0, 1, \dots, n - 1$ , we fill the  $i$ -th position of `current` correctly. Our goal is to move `second[i]` there. It may be already there; if not, we use the built-in method `index` to find the position where `second[i]` appears in `current`. This is the position that needs to be exchanged with  $i$  (and it should be different from  $i$  due to our `if`-check). We add the corresponding transposition (i.e., pair) to the list `transpositions`, and apply it to `current`. After that, we check that our goal is achieved: `current[i]` is equal to `second[i]`. (In fact one could check more:  $i$  first elements are the same, so `current[:i+1]` should be equal to `second[:i+1]`.)

**Problem 166** Replace the line “`for i in range(n):`” by “`for i in range(n-1):`”. Is this algorithm correct too?

### 6.2.1 Counting Permutations and Transpositions

Using two letters A and B (once each), we can get two strings: AB and BA.

**Stop and Think!** How many strings can we get using letters A, B, and C (once each)?

The first letter can be A, B, or C. First we list strings starting with A, then with B, then with C. For each first letter, we have (as we discussed above) two possibilities for the remaining two letters:

ABC, ACB; BAC, BCA; CAB, CBA.

So the answer is 6.

**Problem 167** How many different strings can be made using 4 letters (once each)? Can you answer the same question for  $n$  letters?

You may want to run the following code to list all permutations of four letters.

```
from itertools import permutations
print(list(permutations('ABCD', 4)))
```

Now we are interested in the *number of transpositions needed to obtain a given permutation*. Our goal is the following result that will be used in the analysis of 15-puzzle:

**Theorem 6.2.1** One cannot return to the original ordering after an *odd* number of transpositions.

In other words, if we have  $n$  different objects at  $n$  positions, and after  $k$  transpositions we come back (all objects are again where they were initially), then  $k$  is even (a multiple of 2, i.e.,  $k = 0, 2, 4$ , etc.). Recall that a “transposition” means that two *different* objects exchange their positions.

Mathematicians would say that the *identity* permutation (each element remains in its place) is not a product of an odd number of transpositions.

**Stop and Think!** Can you prove this result for small number of objects, e.g., for  $n = 1, 2$ , or 3?

For  $n = 1$  (one object), there are no transpositions at all, since for a transposition we need two different objects. So our statement is *vacuously* true: no transpositions means no odd sequences of transpositions.

**Stop and Think!** What does Theorem 6.2.1 say for  $n = 2$ ?

For  $n = 2$ , there is only one transposition (no choice: we have only one pair of different objects that can be exchanged). Applying this transposition several times, we go back and forth:

$$\text{AB} \rightarrow \text{BA} \rightarrow \text{AB} \rightarrow \text{BA} \rightarrow \dots$$

Thus, an odd number of transpositions (1, 3, 5, etc.) is equivalent to one transposition, and an even number of transpositions (0, 2, 4, etc.) returns us to the original order. Hence, Theorem 6.2.1 is proven for  $n = 2$ .

**Stop and Think!** Can you prove Theorem 6.2.1 for  $n = 3$ ?

We have six possible orderings, and three pairs of positions that can form a transposition, namely,  $(0, 1)$ ,  $(0, 2)$ , and  $(1, 2)$ . Each transposition changes the ordering of letters, and we have to prove that we can return to the original ordering only after an *even* number of transpositions. This looks a bit messy; can we phrase this in a more structured way?

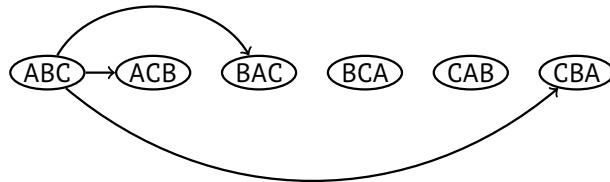
Let us start by listing all the six orderings (using letters A, B, and C):



This picture does not help much, since we do not show the effect of transpositions. Imagine that we start with ABC and apply one transposition. We may get BAC after  $(0, 1)$ , CBA after  $(0, 2)$ , and ACB after  $(1, 2)$ .

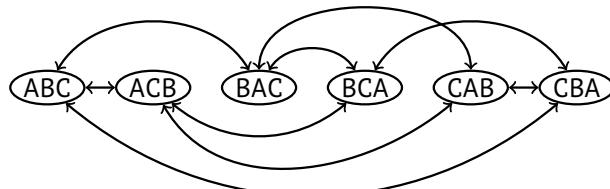
**Stop and Think!** How would you visualize this information?

Probably the easiest way is to add arrows that show all three possibilities (we do not show the transpositions, only their result):



**Stop and Think!** Since we need to study the effect of multiple transpositions, we should add arrows starting from other orderings too. Can you do this?

This requires some patience, but not much: for each vertex (three-letter string), we have three outgoing edges that correspond to three possible transpositions:



**Stop and Think!** In the picture above, all edges are bidirectional — can you explain why this happens? How many edges does each vertex have? Why?

If a string  $Y$  can be obtained from  $X$  by some transposition, and we apply the same transposition to  $Y$ , we come back to  $X$  (two exchanges of the same objects cancel each other). So, an arrow  $X \rightarrow Y$  is accompanied by an arrow  $Y \rightarrow X$ . Note also that we have three transpositions  $(0, 1)$ ,  $(0, 2)$ , and  $(1, 2)$ , and they transform each string  $X$  to three other strings (all different), so we have three bidirectional edges adjacent to each vertex.

Now, having this picture (a *transition graph*), let us recall what we need to prove. We need to prove that we cannot return to the original string after an odd number of transpositions. In terms of this picture: *we cannot come back to the initial vertex after an odd number of moves*. (Mathematicians would say that this graph doesn't have odd cycles.) It is not yet clear why this picture helps at all...

Instead of giving up, let us look at the picture more closely. Where can we go from, say, ABC in one, two, or three steps? For one step, we have three possibilities (connected to ABC by edges):

one step:    ACB, BAC, CBA.

How can we find vertices that can be reached in two steps? We look at the vertices reachable in one step, and make one more step. We need to check three neighbors for each of three vertices reachable in one step. In fact (check this carefully!), these three neighbors are the same for all three vertices, and we get the answer:

two steps:    ABC, BCA, CAB.

**Stop and Think!** How can we find vertices that can be reached from ABC in three steps?

To do this, one should look at all neighbors of all vertices reachable in two steps. It gives

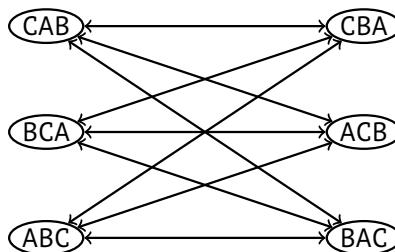
three steps:    ACB, BAC, CBA.

**Stop and Think!** How can we find vertices reachable from ABC in four steps, in five steps, and so on?

In fact, we already have all the necessary information. Looking at our data, we observe that the lists for one and three steps are the same. Hence, taking all neighbors, we get the same lists for two and four steps, and our sequence starts repeating itself. Then we get the same lists for three and five steps, etc. This gives us a general answer:

1, 3, 5, 7, ... steps:    ACB, BAC, CBA  
 2, 4, 6, 8, ... steps:    ABC, BCA, CAB.

This behavior becomes even more clear if we reposition the vertices of our graph keeping the same edge connections:



(Check that we do not cheat and connections are the same!)

**Stop and Think!** Do you see why we come back to the original configuration only after an even number of moves?

Each move brings us from the left part to the right part or back, hence to return to the same part (left or right), we need an even number of moves. Thus — finally! — we have proved Theorem 6.2.1 for  $n = 3$ .

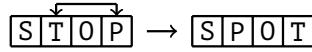
**Problem 168** Prove Theorem 6.2.1 for  $n = 4$  using the same idea.

It is boring but not that difficult: we need to draw a similar graph that classifies 24 possible strings (made of ABCD by permutations) into left and right part in such a way that every edge connects left and right part (no edges between two left or two right vertices). But this approach doesn't generalize to a general argument that can be used for every  $n$ .

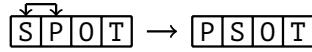
We will soon prove the general result, but some preparation is needed: we should look at *neighbor transpositions*.

### 6.2.2 Neighbor Transpositions

Recall what we started with: permuting letters in a string by exchanging pairs of letters. We called this exchange a *transposition*. If a transposition exchanges neighbor letters, we call it a *neighbor transposition*. For example, our first example,



is not a neighbor transposition: we exchange T and P that are not neighbors (there is O between them). On the other hand, our second example



is a neighbor transposition: the letters S and P that are exchanged are neighbors.

**Stop and Think!** We denoted a transposition by a pair of numbers  $(i, j)$  of exchanged positions. When  $(i, j)$  is a neighbor transposition?

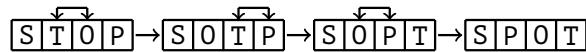
This is an easy question — just to check that we are on the same page:  $i$  and  $j$  should differ by 1; one can write this condition as  $|i - j| = 1$ . Note that we require  $i \neq j$  for every transposition (we cannot exchange a letter with itself).

**Stop and Think!** We proved that every permutation can be obtained by a sequence of transpositions. Is it true that every permutation can be obtained by a sequence of *neighbor* transpositions?

The answer is yes, and we will see why soon. But let us first look at our example.

**Problem 169** Represent the permutation STOP → SPOT as a sequence of neighbor transpositions.

If two people in a queue are not neighbors, they can still exchange their positions by moving along the queue and back. Here the letter T moves to the right (exchanging with O), then it makes an exchange with P, and then P moves back to the left:



In this example, we had one letter between the letters T and P that we needed to exchange. The same trick works for cases with several letters in between, see Figure 6.3.

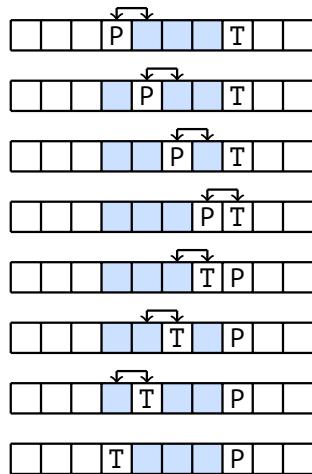


Figure 6.3: Representing an arbitrary transposition as a sequence of neighbor transpositions. The intermediate elements are shaded: P and T jump over them.

**Problem 170** How many neighbor transpositions are used in this construction if there are  $m$  letters between the pair of letters that we want to exchange?

There are three stages:

- T crosses  $m$  intermediate letters by  $m$  neighbor exchanges (=transpositions) and becomes a neighbor of P (in Figure 6.3,  $m = 3$ );

- neighbor transposition of T and P;
- P moves to the left crossing the same  $m$  letters ( $m$  neighbor exchanges).

In total, we have  $2m + 1$  neighbor exchanges. Note that this number is odd; this fact will be used later. Now we only need to know that a non-neighbor transposition can be replaced by a sequence of neighbor transpositions.

**Problem 171** Prove that every permutation can be obtained as a sequence of *neighbor* transpositions.

Indeed, we know already that every permutation can be obtained by a sequence of transpositions; it remains to replace each non-neighbor transposition by a sequence of neighbor transpositions.

Another argument: let us show that any order can be reduced to any other by neighbor transpositions. Let us imagine  $n$  people in a queue that have different weights, and choose the weights in such a way that the desired ordering is weight ordering (maximal weight is assigned to the first person in the desired ordering, etc.). Then, if an ordering is incorrect, there is a place where a heavier person stands just behind a lighter one. Exchange them (they are neighbors) and repeat this procedure until the ordering is correct. This procedure is called *bubble sort*.

One needs to prove that this process terminates. For that we note that the center of gravity of the people moves forward at each step and there are only finitely many orderings.

**Problem 172** Find a sequence of *neighbor* transpositions that converts MARINE to AIRMEN.  
Try it: [Coursera](#).

**Problem 173** Implement a Python function

```
transform(first, second)
```

that takes two permutations of  $\{0, 1, \dots, n - 1\}$  and returns a list of *neighbor* transpositions that transforms the first list into the second one.

Here is a function that converts *one* transposition into a sequence of neighbor transpositions.

```
def convert(transposition):
    i, j = transposition
    assert i != j
    i, j = min(i, j), max(i, j)

    return [(s, s + 1) for s in range(i, j)] + \
           [(s, s + 1) for s in
            reversed(range(i, j - 1))]

for transposition in [(2, 3), (2, 5), (5, 2)]:
    print(transposition, '->', convert(transposition))

(2, 3) -> [(2, 3)]
(2, 5) -> [(2, 3), (3, 4), (4, 5), (3, 4), (2, 3)]
(5, 2) -> [(2, 3), (3, 4), (4, 5), (3, 4), (2, 3)]
```

### 6.2.3 Proof of the Main Theorem

Theorem 6.2.1 claims that we cannot return to the initial ordering after an odd number of transpositions. In other words, if we return to the initial ordering after  $k$  transpositions, then  $k$  is even.

We've proved this result for permutations of  $n$  elements for  $n \leq 3$ , but had no general argument. In this section, we give a proof for arbitrary  $n$  using the notion of a neighbor transposition.

We start by proving the special case of the theorem that allows only *neighbor* transpositions: *If we return to the initial ordering after  $k$  neighbor transpositions, then  $k$  is even.*

**Stop and Think!** Do you see why this is true?

To visualize the statement, imagine a queue with  $n$  people. At each step, two neighbors in the queue exchange their positions while others are standing still. After  $k$  steps, all people return to their initial positions. We need to prove that  $k$  is even.

A crucial idea: let us look at some pair of people in the queue, say, Alice and Bob, and consider only the exchanges that involve both Alice and Bob. At these steps, Alice and Bob exchange their places, so we call them “ $A$ – $B$  steps”. During these steps, Alice and Bob are neighbors (both before and after the exchange).

**Stop and Think!** Do you see why the number of  $A$ – $B$  steps is even if the original ordering is restored at the end?

To see why, note that Alice may be either before Bob or after Bob in the queue. There could be people between them; we just look who is closer to the beginning of the queue, Alice or Bob. In the first case, we write  $A < B$ , in the second case we write  $A > B$ . When Alice and Bob exchange their positions (=during an  $A$ – $B$  step), the situation changes:  $A < B$  becomes  $A > B$  and vice versa. When some other people exchange their positions, or even when Alice or Bob exchanges with somebody else,  $A < B$  remains  $A < B$  and  $A > B$  remains  $A > B$ . (Do you see why?) But we know that at the end, Alice and Bob are where they were initially, hence the mutual ordering is the same at the beginning and at the end. Therefore, the total number of flips back and forth is even.

**Stop and Think!** Do you see why the total number of steps is even?

Let us classify all the exchanges according to their participants: for each pair  $X, Y$  of (different) people we consider  $X$ – $Y$  exchanges. As we have seen, each class (for fixed  $X$  and  $Y$ ) consists of an even number of exchanges. Thus, the total number of exchanges (=neighbor transpositions), being the sum of these even numbers, is even.

We proved theorem 6.2.1 for the special case of *neighbor* transpositions.

**Stop and Think!** Do you see how the general case can be reduced to this special case?

Recall how we replace one non-neighbor transposition by  $2m + 1$  neighbor transpositions (where  $m$  is the number of elements in between). This replacement increases the total number of transpositions by  $2m$ , i.e., by an even number. So if the number of transpositions was even (or odd), it remains even (respectively, odd). Now assume that a sequence of transpositions returns us to the original ordering. Replacing non-neighbor transpositions by neighbor ones, we change the number of transpositions by some even number. And our special-case theorem says that after the replacement the number of transposition is even. Therefore, it was even before the replacement too. Theorem 6.2.1 is proven.

**Problem 174** Consider the following sequence of transpositions:

TASTE → ATSTE → ATSET → AESTT → TESTA → TASTE

We returned to the original configuration after 5 transpositions, and we have shown that the number of transpositions must be even. How is this possible?

We have two letters  $T$ , and in fact these letters in the final ordering changed their places, so the resulting permutation is the  $(0, 3)$  transposition, not the identity permutation. To return to the truly original ordering, we should add this  $(0, 3)$  transposition, thus getting 6 transpositions (an even number, so everything is OK).

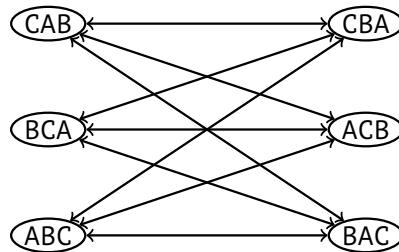
We should be careful and always apply permutations to distinct objects to avoid situations like this one.

We conclude by an important remark. The notion of neighbor transpositions makes sense if the permuted objects form a line (like characters in a string, or elements in an array, or people

in a queue). The argument we used to prove Theorem 6.2.1 is valid only for this case. However, the statement of this theorem does not use this line ordering, and the theorem is valid for any  $n$  objects positioned in  $n$  different places. Indeed, we can create an imaginary line (curve) that goes through all the positions. In other words, we may number the positions by integers  $1, 2, \dots, n$  and consider the exchanges between positions  $i$  and  $i + 1$  as neighbor transpositions. (We will return to this discussion when applying our theory to 15-puzzle, in Section 6.3.)

#### 6.2.4 Even and Odd Permutations

Recall the classification of permutations of  $n$  objects (different letters) for  $n = 3$ :



On the left, we have the original string ABC and all other strings that one gets after an even number of transpositions; on the right, we have strings that are obtained by an odd number of transpositions. The permutations that correspond to the left part (that are obtained by an even number of transpositions) are called *even*, and the permutations that correspond to the right part (that are obtained by an odd number of transpositions) are called *odd*.

The use of words *even* and *odd* for permutations may be confusing at first: this has nothing to do with the number of objects ( $n$ ) being even or odd.

To make this classification well defined, we need to prove the following:

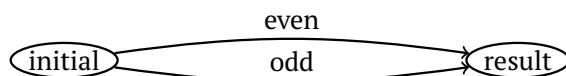
**Theorem 6.2.2** If some permutation is obtained by an even number of transpositions, it cannot be obtained by an odd number of transpositions.

This theorem shows that these two classes of permutations (obtained by even or odd number of transpositions) are disjoint. Hence, we may call them *even* and *odd* permutations and no permutation can be even and odd at the same time.

**Stop and Think!** Is it true that every permutation is either even or odd?

Yes: we know that every permutation can be obtained by a sequence of transpositions, and this sequence may include even or odd number of transpositions. The problem would appear if one sequence of transpositions is even (i.e., consists of an even number of transpositions) while another is odd, and we need to prove Theorem 6.2.2 to exclude this case.

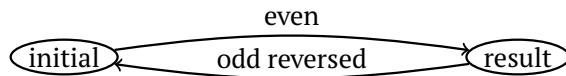
Imagine that the claim is false and there are two sequences of transpositions, one even and the other odd, that lead to the same ordering:



Each arrow denotes a sequence of transpositions; one contains an even number of transpositions, and the other contains an odd number of transpositions.

**Stop and Think!** Do you see a reason why this cannot happen?

Let us apply first the even sequence of transpositions, and then the odd one, but *in the reversed order*. In other words, append to the video of the even sequence the video of the odd one going backwards.



Since both sequences (even and odd) end at the same position, we get a consistent picture (no sudden changes in the middle of the combined video). In other words, we now have a sequence of transpositions that starts and ends in the same position, and the total number of transpositions is even + odd, which is odd. This is impossible by Theorem 6.2.1. Theorem 6.2.2 is proven.

### 6.2.5 Finding the Parity of a Permutation

Given a permutation, how can we find whether it is even or odd? How could one implement a function that checks whether a given permutation is even or odd? For convenience, let us assume that  $n$  objects that are permuted are integers from `range(n)`, i.e.,  $0, 1, 2, \dots, n - 1$ .

**Stop and Think!** Is the permutation  $[4, 0, 1, 2, 3]$  even or odd?

To answer this question, we need to know how many transpositions are needed to get this ordering from  $[0, 1, 2, 3, 4]$ . We know how to represent a permutation as a sequence of transpositions (placing correct values one by one). For this example, we could do the following.

$$[0, 1, 2, 3, 4] \rightarrow [4, 1, 2, 3, 0] \rightarrow [4, 0, 2, 3, 1] \rightarrow [4, 0, 1, 3, 2] \rightarrow [4, 0, 1, 2, 3].$$

We used four transpositions, so the permutation is even.

**Problem 175** Implement a program that gets a list of length  $n$  that consists of the integers  $0, 1, \dots, n - 1$  in some order, and returns `True` if and only if the corresponding permutation is even. Try it: [Coursera](#) ↗.

To solve this problem, one may use the program that computes a sequence of transpositions that transform one list into the other one.

Programmers could prefer another approach: we can go backwards (the number of transpositions is the same) and sort the given array by an exchange sorting algorithm (at each step, two elements are exchanged). Since our initial configuration is sorted, returning to the initial configuration means sorting. Then, the permutation is even if and only if the number of exchanges (made by the sorting algorithm) is even.

What are the advantages of this approach? The experts in algorithms design would say that our original approach requires about  $n^2$  steps for a permutation of length  $n$ : we need to put  $n$  objects in place, and each object should be found in the list (which requires up to  $n$  comparisons, even if this is masked by a Python `index` operation).

On the other hand, there are sorting algorithms that are faster (require about  $n \log n$  operations), and they can be adapted to provide a sequence of transpositions. This gives a faster way to check whether a permutation is even or odd. Still, this is not the optimal approach: there are methods where the number of operations is proportional to  $n$ .

**Problem 176** Find an algorithm that determines whether a permutation of  $\{1, \dots, n\}$  is even or odd using  $O(n)$  operations.

Hint: one can decompose a permutation into cycles; another possibility is to keep the permutation and its inverse to avoid wasting time on index operations.

## 6.3 Why 15-puzzle Has No Solution

We are ready to return to the 15-puzzle and show that no sequence of game moves can exchange 14 and 15 (transform the left configuration into the right one shown in Figure 6.4).

For the proof, let us replace the empty cell by a special dummy piece 0, see Figure 6.5 (a). After this, pieces cannot move. If in the original game we move, say, 12 down, in this new representation we exchange 12 and 0, see Figure 6.5(b)–(c). Every move in the original game is now a transposition in the new game (exchanging the moving piece with 0-piece).

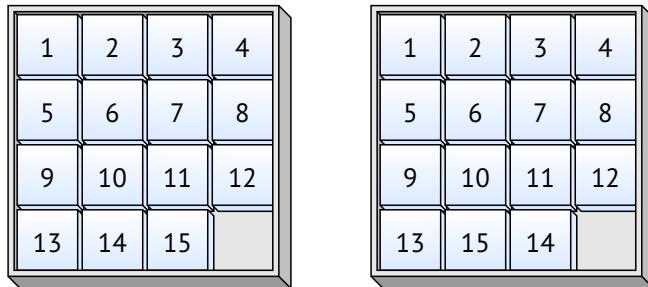


Figure 6.4: One cannot go from the left configuration to the right one, and soon we will see why.

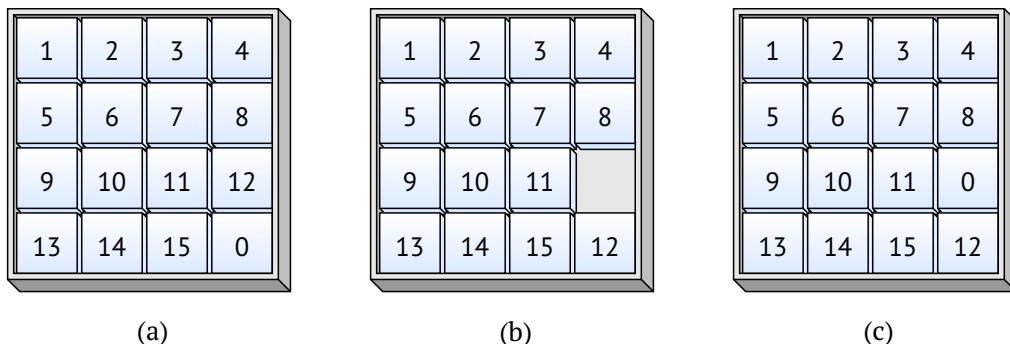


Figure 6.5: Empty cell replaced by 0-piece (a). A move in the original game (b) is now an exchange with the 0-piece (c).

**Stop and Think!** If the original 15-puzzle (where one needs to exchange 14 and 15) were solvable, the solution would require an odd number of moves. Do you see why?

This is a direct corollary of what we know about even and odd permutations. The exchange of 14 and 15 corresponds to a transposition in the new representation (as permutations of 16 pieces, including the dummy piece), so the required permutation is odd. This means that if this permutation is obtained by a sequence of moves, and each move is a transposition, then the sequence should contain an odd number of moves (otherwise we would get an even permutation).

Here, as we have discussed above, the permuted objects do not form a line. This does not invalidate the distinction between even and odd permutation. But if you prefer to deal with a list, write down the puzzle configuration as a list, in a reading order. The move shown in Figure 6.5 will then transform the list

[1,2,3,4,5,6,7,8,9,10,11,0,13,14,15,12]

into

[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,0]:

On the other hand, a different argument shows that any solution of the 15-puzzle should have an even number of moves. For that, we do not need to look at the numbers; only the position of the empty cell is important.

**Stop and Think!** Do you see why the number of moves in the 15-puzzle must be even?

If we ignore the pieces and look only at the position of the empty cell, we see that at each move the empty cell makes one step in some direction (left, right, up or down). At the end of the game,

the empty cell should return to its original place, the right lower corner. Therefore, the number of steps left must be equal to the number of steps right (left = right); also the number of steps up must be equal to the number of steps down (up = down). Thus, the total number (left + right + up + down) is even (being equal to  $2 \cdot \text{left} + 2 \cdot \text{up}$ ). In fact, this question was already considered in Problem 151 where we proved that a piece on a chessboard cannot return to its original position after an odd number of moves.

**Stop and Think!** Now we have two arguments: one is based on permutations and shows that the number of moves in a solution of the 15-puzzle must be odd; the other one uses the empty cell position and shows that the number of moves must be even. Is mathematics inconsistent?

Of course not: we proved only that *if a solution to this puzzle existed*, it would require an even number of moves — and, at the same time, an odd number of moves. This is a contradiction, but this contradiction proves only that there is no solution. Thus, we have proven that the puzzle (exchanging 14 and 15 by legal game moves) is unsolvable.

## 6.4 When 15-puzzle Has a Solution

In this section, we consider a general question: what configurations in the 15-puzzle are solvable? The answer is provided by the following criterion.

**Theorem 6.4.1** Consider a configuration  $X$  in the 15-puzzle where the empty cell is located in the bottom right corner. The configuration  $X$  can be achieved from the initial configuration if and only if  $X$  is an even permutation of the initial configuration.

The statement of this theorem needs some clarification. The configuration  $X$  can be considered as a permutation of 15 pieces. Or we may add a dummy piece 0 and consider  $X$  as a permutation of 16 pieces, including the dummy one (which happens to stay in the bottom right corner). Which of these two permutations is used in Theorem 6.4.1?

The answer is that *this does not matter*. If  $X$  is even or odd as a permutation of 15 pieces, this means that  $X$  can be obtained by an even (resp. odd) number of transpositions (pair exchanges between these 15 pieces). The same transpositions can be considered as transpositions for the full board (with a dummy piece) that give  $X$ . In general, an even/odd permutation remains even/odd when we add one object that does not move.

How can we prove Theorem 6.4.1? In one direction, the proof repeats the argument we have already seen. If a sequence of moves transforms the original configuration into some other configuration that has the empty cell in the bottom right corner, then this sequence contains an even number of moves (since the empty cell returns to its place). Therefore, we get an even permutation of 16 pieces (including the dummy one) and, as we have mentioned, an even permutation of  $1, 2, \dots, 15$ .

The other direction is more difficult and some preparations are needed. Let us start with a simple question (just for a warm-up).

### 6.4.1 Moving the Empty Cell

**Problem 177** Prove that you can move the empty cell wherever you want: for every configuration  $C$  (empty cell may be anywhere) and for every desired position  $p$  of the empty cell on the game field, there exists a sequence of moves that bring the empty cell to the position  $p$  (starting from configuration  $C$ ).

To see why this is true, we ignore all the labels, and follow the position of the empty cell only. As we mentioned, we may move the empty cell left, right, up, and down. Of course, physically we move not the empty cell but the piece that is now placed in the cell where the empty cell is moving to.

Using these moves (left, right, up, and down) we can bring the “hole” (empty cell) wherever we want.

### 6.4.2 Moving a Given Piece to the Top Left Corner

Our next task will be more complicated. Imagine that you have chosen some piece and you want to bring it to, say, the top left corner. (Initially this piece can be anywhere, and the position of the hole may also be arbitrary.)

**Problem 178** Prove that this is always possible.

If you've played this game a few minutes, this is perhaps already clear to you: there is a lot of movement freedom if we are interested only in one piece and do not care about the others. However, how could we prove this rigorously for any configuration? Or how can we implement a function that computes the required sequence of moves?

A tool that is useful here is a *cyclic move*. Let us explain what a cyclic move is using a metaphor. Put 16 chairs around a table, as it is often done for meetings. Imagine that one of 16 persons did not come for the meeting. So there is a "hole": one chair is empty, and 15 others are occupied by the participants. Assume that the only allowed move is when someone moves to the empty chair near her (so that others are not disturbed). Figure 6.6 visualizes a single move.



Figure 6.6: There are 16 chairs around a table, one of them is vacant. A person next to the vacant chair moves to this empty chair counterclockwise. As a result, the vacant chair (a "hole") moves clockwise.

**Stop and Think!** How can we bring any participant to any chair using only allowed moves?

This is easy: if we start a wave of movements in some direction (I move right to the empty chair, my left neighbor moves right to my place, her left neighbor moves to her place, etc.), we get a full cycle where everybody just moved one chair to the right. Repeating this cycle, we can move any given participant to any given place.

**Stop and Think!** How does this remark help us? Recall that we need to bring a given piece in a 15-puzzle configuration to the top left corner.

When we speak about a cyclic movement, the cycle does not need to be a physical circle. We can draw an imaginary cycle (a "snake") on the board, see Figure 6.7 (a).

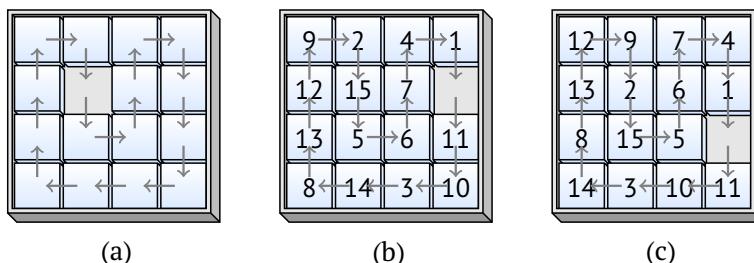


Figure 6.7: (a) A cycle through all the cells of the board. (b) A particular configuration. (c) The result of a cyclic move applied to this configuration. To perform the cyclic move, one first moves 1 down (and the hole moves up), then 4 moves into the new hole (a cell vacated by 1), etc.

**Stop and Think!** How many moves are needed to complete one full cycle?

Since every piece should move one step along the cycle, we need 15 moves.

**Stop and Think!** How many cycles are still needed (after the first one, Figure 6.7 (c)) to bring 3 to the top left corner?

Looking at the arrows, we see that after the first cycle (that moved 3 left), we need four more cycles to put 3 in the top left corner.

Of course, the same *U*-shaped cycle can be used for any configuration and any position of the hole. This way, we are able to move any given piece to the left top corner, as requested.

**Stop and Think!** What piece will be to the right of the hole after four more cycles, when 3 is in the top left corner?

To get the answer, we can draw all the pieces along the cycle, starting from 3 and keeping the cycle ordering. We get the following configuration (Figure 6.8; check the cycle ordering). Thus, to the right of the hole we have 1.

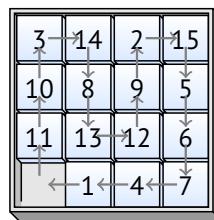


Figure 6.8: After five cycles, 3 is in the top left corner and the cycle ordering remains the same.

#### 6.4.3 Moving One More Piece

It is time to think where we are, comparing what we wanted to achieve and what we have achieved. Our goal was to implement an arbitrary permutation, i.e., to put all 15 pieces in the right places — assuming that the permutation is even. What we have achieved so far is more modest: we know how to fill the top left corner with a given piece. Still, it is a step in the right direction. After the top left corner is filled with a correct piece, we may decide that we never move that piece again, and fill correctly some other cell, for example, the cell below the top left corner.

**Problem 179** Prove that it is always possible: if we glue the top left corner piece and never move it, we still can put any of the remaining pieces to the cell below the top left corner.

How can we do this? In fact, we have just decreased the size of our field by deleting the top left corner. We can try to use the same approach: form a cycle that goes through 15 remaining cells, and move the pieces along this cycle.

**Stop and Think!** Can you draw such a cycle?

Trying to do this, you will see that one cell remains outside the cycle.<sup>2</sup> For example, Figure 6.9 shows such a cycle that avoids the top left corner (as we wanted), the left bottom corner (inconvenient but unavoidable), and goes through all other cells.

Having a cycle that covers all cells of the reduced field except one, we may use this cycle as before. This solves our problem assuming that (a) the cell we want to bring below the top left corner, is not in the bottom left corner and (b) the hole is not in the bottom left corner.

<sup>2</sup>There is a simple reason why this is unavoidable: moving along the cycle, we return to the same position. So, as we have seen, the number of moves (the length of the cycle) is even.

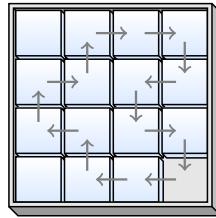


Figure 6.9: Using this cycle, we can move any piece (with one exception: left bottom corner) to the cell below the top left corner. Note the the cycle must include a hole to make the cyclic movement possible. In our picture, the hole is shown in the right bottom corner, but any other place is possible (except the bottom left corner that is not in the cycle).

**Stop and Think!** Do you see why conditions (a) and (b) are important?

If (a) is not true, the cyclic move won't help. If (b) is not true, the cyclic move is impossible.

**Problem 180** Consider Figure 6.10. How many cyclic moves (shown in the figure) are needed to bring the piece 9 in the cell below the top left corner?

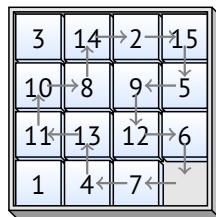


Figure 6.10: The top left corner is occupied by 3 forever; we want to bring 9 in the cell below the top left corner.

Now, we need to consider the exceptional cases when conditions (a) or (b) are not fulfilled.

**Stop and Think!** What should we do if (b) is false, i.e., the hole is in the bottom left corner?

We could move the hole right or up (by moving the neighbor pieces). Any of the two neighbor pieces would move the hole, but we should be careful. Recall that we wanted to move some piece below the top left corner, so we should not place this piece in the bottom left corner. Since we have a choice between two neighbors, this is always possible.

**Stop and Think!** What should we do if (a) is false, i.e., if the current position of the piece we want to bring below the top left corner, is the bottom left corner?

For example, this is the case for piece 1 in Figure 6.10.

Since we already know how to use cycles, we may just use another cycle that covers the bottom left corner (and the cell below the top left corner). One such cycle is shown in Figure 6.11.

#### 6.4.4 Moving the Third Piece

Where are we now? A bit closer to our goal: we know how to fill *two* positions by the pieces we want: the top left corner and the cell below it. We can continue this process and fill the cell on the right of the top left corner, using the same approach.

For that, we need to cover the board without two cells (top left corner and the cell below it) by a cycle.

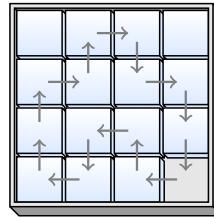


Figure 6.11: Using this cycle, we can move the piece from the bottom left corner (as well as others, except for the top right corner) to the place below the top left corner. Again, we assume that the cell outside the cycle (top right corner) is not a hole; if it is, we fill it by its neighbor.

**Problem 181** Construct such a cycle.

One possible solution is shown in Figure 6.12.

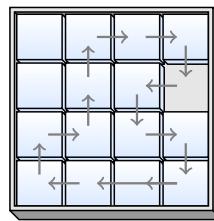


Figure 6.12: This cycle allows us to put any of the remaining pieces to the right of the top left piece, not touching the piece in the top left corner and the piece below it.

One could say that we are 20%-ready: to get an arbitrary permutation, we need to fill 15 places, and now we know how to fill three places (20% of 15).

**Problem 182** Show how one can fill the cell in the second row and second column, using the same technique.

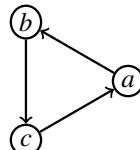
However, this 20% or even 26% ( $4/15 = 0.2666\dots$ ) are too optimistic: the more pieces we bring to the desired positions (and do not want to move again), the less freedom we have for the cycle, and the more difficult the task is. For example, after we fill three places in the top row, we cannot use the cycle technique for the remaining (fourth) piece: there is no cycle, since this cell now has only one free neighbor. So we are stuck and some new idea is needed.

In the next section, we introduce new tools (3-cycles and conjugacy) that will allow us to move forward.

#### 6.4.5 Decomposition into 3-cycles

For a while, we forget about 15-puzzle and consider permutations in general.

A 3-cycle is a permutation where all the objects except three of them keep their places, and these three move in a cycle:



It is easy to see that a 3-cycle can be obtained by two transpositions and therefore is an even permutation.

**Problem 183** Find two transpositions that (being performed sequentially) give the 3-cycle  $a \rightarrow b \rightarrow c \rightarrow a$ .

If we apply sequentially several 3-cycles, each of them can be replaced by two transpositions, therefore we get an even permutation. The following theorem shows that *every even permutation can be obtained this way*.

**Theorem 6.4.2** Every even permutation of  $n$  objects can be represented as a sequence of 3-cycles.

Note that this theorem is vacuously true for  $n = 1$  or  $n = 2$ : there are no even permutations except for the identity permutation (everybody stays in their place) that is a result of an empty sequence of 3-cycles.<sup>3</sup>

*Proof sketch.* Earlier, we proved that every permutation can be decomposed into transpositions — filling the places one after another by desired objects. We can do the same with 3-cycles instead of transpositions (2-cycles), if there are at least 3 objects left (we need a third spare object to form a cycle). Hence, compared to our previous argument for transpositions, we need one spare object, and we have to stop when only two positions remain unfilled. The good case is when the two remaining objects are already in the right places. The bad case is when they are not, and an additional transposition is needed to get the required permutation. But in this bad case the required permutation is obtained by several 3-cycles and one transposition, and therefore is odd, contrary to our assumption. ■

**Problem 184** Implement a Python function

```
transform_by_3cycles(first, second)
```

that takes two lists that are permutations of  $\{0, 1, \dots, n - 1\}$  and returns a list of 3-cycles needed to get the second list from the first one. We assume that the permutation that transforms the first list into the second, is even. Each 3-cycle is represented by a triple of positions:  $(i, j, k)$  moves the element from position  $i$  (respectively,  $j, k$ ) to position  $j$  (respectively  $k, i$ ).

```
def transform_by_3_cycles(first, second):
    assert len(first) == len(second)
    n = len(first)
    assert set(first) == set(range(n))
    assert set(second) == set(range(n))

    cycles = []
    current = list(first)

    for i in range(n - 2):
        if current[i] != second[i]:
            idx = current.index(second[i])
            spare = i + 1 if idx != i + 1 else i + 2
            assert i != idx and i != spare and \
                   idx != spare
            cycles.append((idx, i, spare))
            current[i], current[idx], current[spare] = \
                current[idx], current[spare], current[i]

    assert current[i] == second[i]

    return cycles
```

<sup>3</sup>3-cycles don't exist for  $n = 1$  or  $n = 2$ , but this does not prevent us from considering an *empty* sequence of 3-cycles.

```

print(transform_by_3_cycles(
    [3, 4, 0, 2, 1, 5],
    [0, 5, 1, 4, 3, 2]
))
[(2, 0, 1), (5, 1, 2), (4, 2, 3), (5, 3, 4)]

```

In this code snippet, we check that `first` and `second` lists consist of  $0, 1, \dots, n - 1$ , but do *not* check that the permutation required to transform the first list into the second one is even. The structure of the program is the same as before (for transpositions), but we need a spare element. We use `i + 1` if possible (and if not, use `i + 2`; one of these two elements should be different from `idx`, and both are different from `i`). The `for` loop omits two last elements ( $n - 2$  and  $n - 1$ ), since there are no spare elements for them. If the permutation was even, they will go in the correct order automatically.

So far we developed a theory about even permutations and 3-cycles, but how does it help us to deal with 15-puzzle? Recall that we have to prove Theorem 6.4.1 for every even permutation of 15 pieces (here we do not add the dummy piece 0 and assume that the right bottom corner is empty). This permutation can be obtained as a sequence of 3-cycles. And if we can perform any 3-cycle according to the puzzle rules, we can combine the corresponding sequences of puzzle moves to achieve any even permutation. Therefore,

*it is enough to prove Theorem 6.4.1 for a 3-cycle.*

We do this in the next section.

#### 6.4.6 How to Get an Arbitrary 3-cycle

Let us first recall what we have proven in Sections 6.4.2–6.4.4:

*in the 15-puzzle, we can put three arbitrary pieces  $a, b, c$  in the upper left corner*

as shown in Figure 6.13 (a).

From Section 6.4.1, we know how to move the empty cell ("hole") in any direction, so we can put it near them, as shown in Figure 6.13 (b).

**Stop and Think!** Why don't we destroy the  $a$ - $b$ - $c$  corner while moving the hole?

The problem may appear if we move the hole along a path that crosses the  $a$ - $b$ - $c$  corner. But this can be easily avoided (take a path avoiding the corner: the board without the corner is connected).<sup>4</sup>

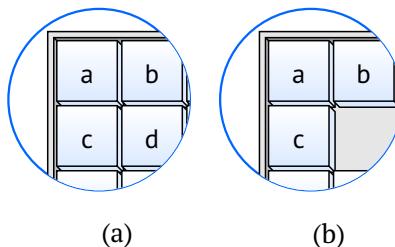


Figure 6.13: Three pieces  $a, b, c$  moved to the top left corner (a) with a hole near them (b).

How does this help us? Recall our cycle trick: now we can apply it to these four cells and organize a circular movement inside the  $2 \times 2$  square (Figure 6.14).

Recall that our goal was to implement a 3-cycle for any triple of positions; we have achieved this for a very special case of three positions near the top left corner. Again, what we achieved here is a very small part of what we wanted.

<sup>4</sup>One can also use the cyclic move along a suitable cycle; see the discussion in the next section.

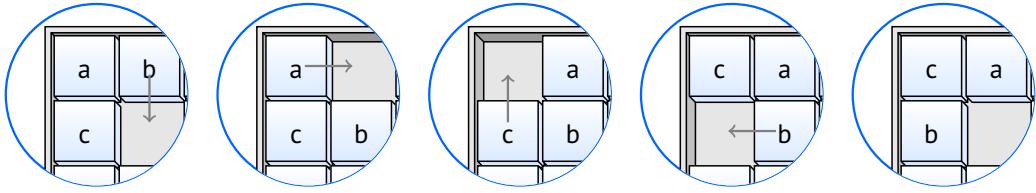


Figure 6.14: A 3-cycle performed inside a  $2 \times 2$  square in the top left corner.

Still, we can combine our two observations to achieve the goal. Here is the key idea. To implement a 3-cycle that involves arbitrary three pieces  $a, b, c$ , one should:

- Step 1.** Bring three pieces  $a, b, c$  to the top left corner and put the hole near them (as we explained above).
- Step 2.** Perform that 3-cycle in the top left corner.
- Step 3.** Perform the reversed sequence of moves made during Step 1.

Mathematicians call this trick *conjugation*.

If we skipped Step 2, then Step 3 would restore the original configuration: video for Step 3 would be just a reversed video for Step 1. But after Step 2,  $c$  stands in place of  $a$ ;  $b$  stands in place of  $c$ , and  $c$  stands in place of  $b$ . Hence, performing all three steps, we obtain a 3-cycle that involves  $a, b$ , and  $c$ . Since this is possible for arbitrary 3 pieces, we can implement all 3-cycles, and therefore (Theorem 6.4.2) every even permutation.

## 6.5 Implementation

We have shown that a configuration in the 15-puzzle is solvable if and only if the corresponding permutation of 15 objects is even. This gives rise to a natural programming challenge: given a configuration (a list of positions of all pieces), find a sequence of allowed moves that transforms this position into the standard one.

We have to agree on the input and output formats. For the input, we represent the empty cell by 0, and all the pieces (including this 0-piece) are listed according to their labels in the “reading order”, so the standard configuration is represented as

$$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0],$$

whereas the impossible configuration we discussed is represented as

$$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 15, 14, 0].$$

Hence, every configuration is now represented by a permutation of  $0, 1, \dots, 15$ , and we assume (as the statement of Theorem 6.4.1 does) that in the initial configuration the last number (bottom right cell) is 0.

For the output format, we have to decide how the game moves are encoded. The move is uniquely determined by the piece that is moved (since there is only one empty cell, the direction of the move is determined uniquely, if the move is possible at all). Then the sequence of moves is just a list of numbers that are written on the moved pieces, in the same order as the moves happen. For example, you may check that for the position

$$[1, 2, 3, 4, 5, 6, 7, 8, 13, 9, 11, 12, 10, 14, 15, 0]$$

one of the solutions is

$$[15, 14, 10, 13, 9, 10, 14, 15].$$

**Problem 185** Implement a Python function

```
solution(configuration)
```

that gets a solvable configuration with the empty piece in the bottom right corner and outputs a sequence of moves that transforms it to a standard position. Try it: [Coursera](#).

An approach to this problem suggested by the discussion above is the following. Given an initial position,

- construct a sequence of 3-cycles needed to transform configuration into the standard position (recall Problem 184);
- decompose each 3-cycle  $(a, b, c)$  into a sequence of valid game moves as follows:
  - move  $a$  to the top left corner;
  - move  $c$  below  $a$ ;
  - move  $b$  to the right of  $a$ ;
  - move the empty cell (piece 0) below  $b$ ;
  - make four moves to mimic the 3-cycle that involves pieces  $a, b, c$ ;
  - repeat the moves from the first four steps in the reversed order.

(As the empty cell is in the bottom right corner initially,  $a, b, c \neq 0$  during this process.)

We encourage you to implement this plan. It requires some patience and care, as we have found ourselves when trying to provide a reference solution. We reproduce this reference solution here in full. But (please!) try to solve the problem yourself before reading the solution. First, you may find a cleaner and nicer solution. Second, it will be much easier to understand (criticize, appreciate) our solution if you have your own experience.

First, we should be prepared to debug the program. For that it is convenient to print the configuration in a readable form.<sup>5</sup> Recall that configuration is represented as a list of length 16 that includes all the pieces' numbers in the reading order. This list should be a permutation of `range(16)`; as we agreed, 0 stands for the empty cell. (The function prints an additional space character before one-digit piece numbers.)

```
def fancy_print(configuration):
    assert len(configuration) == 16
    print('-----')
    for i in range(len(configuration)):
        if configuration[i] < 10:
            print(' ', end=' ')
        print(configuration[i], end=' ')
        if i % 4 == 3:
            print()
```

We number the board cells in the reading order; 0 is the top left corner, 1 is the cell to the right of the top left corner, ..., 4 is the cell below the top left corner, ..., 15 is the bottom right corner. But we also need 2D coordinates of the cells in the form `(row, column)`. Both `row` and `column` are integers in  $0 \dots 3$ ; for example, the top left corner is  $(0, 0)$ , the cell on the right of it is  $(0, 1)$ , and the bottom right corner is  $(3, 3)$  (Figure 6.15).

The following function converts 1D position to its 2D representation and returns a pair of integers:

```
def to_2d_index(index):
    assert 0 <= index < 16
    return index // 4, index % 4
```

---

<sup>5</sup>In our solution, this function is not used since it is needed only for debugging; we provide it for your convenience and to illustrate the input encoding.

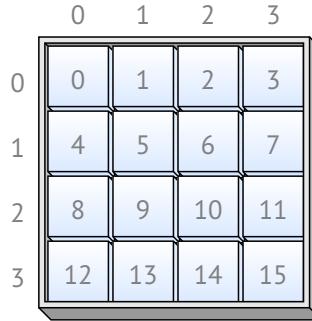


Figure 6.15: Indexing of pieces as well as row and columns of the board.

The puzzle moves are represented as transpositions (a piece is exchanged with an empty cell near it).<sup>6</sup> Each transposition is encoded as a pair of 1D coordinates of cells that are exchanged. The following function checks that the cells numbers are valid and different, the cells are neighbors, and one of the cells is empty in the configuration. Then (the last line) it performs the exchange.

```
def apply_transposition(configuration, transposition):
    i, j = transposition
    assert i in range(16) and j in range(16)
    assert i != j
    i2d, j2d = to_2d_index(i), to_2d_index(j)
    assert abs(i2d[0] - j2d[0]) + \
        abs(i2d[1] - j2d[1]) == 1
    assert configuration[i] == 0 or \
        configuration[j] == 0
    configuration[i], configuration[j] = \
        configuration[j], configuration[i]
```

The sequences of moves are represented by lists of transpositions. Each lists contains pairs (transpositions) that should be performed sequentially:

```
def apply_transpositions(configuration, transpositions):
    for transposition in transpositions:
        apply_transposition(configuration, transposition)
```

The key role in the solution is played by sequence of moves performing a cyclic shift along some cycle. We represent a cycle as a list of 1D positions. The start position is arbitrary; for example, [0, 1, 5, 4] and [1, 5, 4, 0] represent the same cycle (while [0, 4, 5, 1] represents another cycle, with the opposite orientation). A cyclic shift along  $[a, b, \dots, y, z]$  is

$$a \leftarrow b \leftarrow \dots \leftarrow y \leftarrow z \leftarrow a$$

(piece moves from  $b$ -position to  $a$ -position, ..., from  $z$  to  $y$ , from  $a$  to  $z$ ).

The following function applies the cyclic shift along the given path to the given configuration and returns the sequence of moves (transpositions) needed. First, we need to change the representation of the path in such a way that it starts with the empty cell. For that we search for the empty cell and then use slices and concatenation. We have to perform the transpositions sequentially:  $a$  (that is now a hole) is exchanged with  $b$ , then  $b$  is exchanged with  $c$ , etc. The list of transposition is prepared, then applied to configuration and then returned.

<sup>6</sup>This is not the required output format, so we will have to reencode the solution later.

```
def cyclic_shift(configuration, path):
    start = 0
    while configuration[path[start]] != 0:
        start = start+1
    rotated = path[start:]+path[:start]
    transpositions = []
    for i in range(len(rotated)-1):
        transpositions += [(rotated[i], rotated[i+1])]
    apply_transpositions(configuration, transpositions)
    return (transpositions)
```

The argument explained in the previous section used some carefully chosen cycles; in our program they are represented by the constants:

```
cycle1 = [0, 4, 8, 12, 13, 14, 15, 11, 7, 3, 2, 6, 10, 9, 5, 1]
cycle2a = [1, 5, 4, 8, 9, 13, 14, 15, 11, 10, 6, 7, 3, 2]
cycle2b = [1, 5, 4, 8, 12, 13, 9, 10, 14, 15, 11, 7, 6, 2]
cycle3 = [1, 5, 9, 8, 12, 13, 14, 15, 11, 10, 6, 7, 3, 2]
cycle4 = [2, 6, 5, 9, 8, 12, 13, 14, 15, 11, 7, 3]
```

Here, `cycle1` corresponds to the cycle shown in Figure 6.7. (Please do not trust us and check this carefully!) Arrows on the picture indicate the direction of the pieces' movements. In our program, the positions are listed in the reversed ordering (pieces are moved from right to left in the cycle:  $0 \leftarrow 4 \leftarrow 8 \leftarrow \dots$ ). The next `cycle2a` corresponds to the cycle of Figure 6.9; the alternative cycle (Figure 6.10) is encoded as `cycle2b`. The constant `cycle3` corresponds to Figure 6.12 (used to bring the third piece to the position on the right of the top left corner). The last constant `cycle4` was not explained before; it is used to bring the empty cell to the position (1, 1), see below.

The following function is the central part of our solution. It performs a 3-cycle that involves three pieces labeled `a`, `b`, `c`: the piece `a` is moved to the place where `b` was, etc. Here, the participants of the cycle are specified by their names (what is written on the piece), not their positions (where the piece is). The initial configuration is `cfg`; the sequence of transpositions (as pairs) is applied to `cfg` and returned.

```
def do_3_cycle(cfg, a, b, c):
    assert a in range(16) and b in range(16) and c in range(16)
    assert a != b and a != c and b != c
    assert a != 0 and b != 0 and c != 0
    transpositions = []
    # move a to top left corner (position 0)
    while cfg[0] != a:
        transpositions += cyclic_shift(cfg, cycle1)
    # move c below a
    if cfg[12] != c:
        # make sure the hole is not in the bottom left corner
        if cfg[12] == 0:
            transposition = (12, 8 if cfg[8] != c else 13)
            transpositions += [transposition]
            apply_transposition(cfg, transposition)
        while cfg[4] != c:
            transpositions += cyclic_shift(cfg, cycle2a)
    else:
        assert cfg[12] == c
        if cfg[3] == 0:
            transposition = (3, 7) # we know that cfg[7] != c
            transpositions += [transposition]
            apply_transposition(cfg, transposition)
```

```

while cfg[4] != c:
    transpositions += cyclic_shift(cfg, cycle2b)
# move b to the right of a
while cfg[1] != b:
    transpositions += cyclic_shift(cfg, cycle3)
# move the empty cell to the 2x2 block
# ensure the hole is not at 10:
if cfg[10] == 0:
    transposition = (10, 11)
    transpositions += [transposition]
    apply_transposition(cfg, transposition)
while cfg[5] != 0: # use one more cycle to move the hole
    transpositions += cyclic_shift(cfg, cycle4)
# now everything is ready for the conjugation
assert cfg[0] == a and cfg[1] == b and cfg[4] == c and cfg[5] == 0
abccycle_and_reverse = [(1, 5), (0, 1), (0, 4),
                        (4, 5)]+list(reversed(transpositions))
apply_transpositions(cfg, abccycle_and_reverse)
return transpositions + abccycle_and_reverse

```

This code consists of several parts. First we check that  $a$ ,  $b$ , and  $c$  denote valid pieces (not the empty cell) and are different, and prepare the empty list `transpositions` where the moves will be placed.

Then, we compute a sequence of transpositions needed to place the piece labeled  $a$  to the top left corner. For that we perform the cyclic shift along `cycle1` until the label `cfg[0]` (the contents of the top left corner) becomes  $a$ . Next, we need to put  $c$  below  $a$ . This requires considering two cases.

To use `cycle2a`, we need to know that  $c$  is covered by this cycle, i.e.,  $c$  is not in the avoided cell 12 (`cfg[12] != c`). Therefore, the `if` construction is used. But this is not all. The other exceptional case happens if the empty cell is outside the cycle (i.e., the bottom left corner is empty, `cfg[12] == 0`). In this case, we need to exchange the bottom left corner (position 12) with one of its neighbors (positions 8 or 13; we need to chose a neighbor that is different from  $c$ , to keep  $c$  in the cycle).

If  $c$  is in the bottom left corner, we use `cycle2b` instead of `cycle2a`. Again we need to ensure that the hole is in the cycle and not in the top right corner (i.e., that `cfg[3] != 0`). If `cfg[3] == 0`, we move the hole down to position 7. Here we are sure that  $c$  is not at position 7, being at position 12, so the conditional expression is not needed. After these preparations, we perform cyclic shifts along `cycle2b` until  $c$  moves in the desired position 4 (below the top left corner).

Then, we move  $b$  to the position 1 not touching  $a$  and  $c$ ; here, we may use `cycle3` without hesitation.

The next step is to move the hole to the position 5 (to complete the  $2 \times 2$  square as shown in Figure 6.13). In our argument above, we just noted that we can move the hole in any direction (and avoid three cells labeled  $a$ ,  $b$ ,  $c$ ), but to make the program shorter (though using more moves) we use the cyclic shift along `cycle4` (Figure 6.16). The only precaution needed: if the hole is outside `cycle4`, at position 10, we move it elsewhere (say, to position 11 by exchanging 10 and 11).

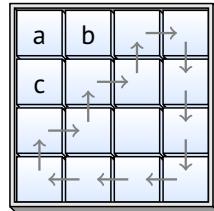


Figure 6.16: To move the hole at position (1,1), we use `cycle4`.

After that we have  $2 \times 2$  square in the top left corner as shown in Figure 6.13 (b). It remains to perform the cyclic shift of a, b, c, as shown in Figure 6.14, and then play back the movements we have done at the first stage. The remaining transpositions are placed in `abccycle_and_reverse`, applied to `cfg` and then returned together with the `transpositions` list.

Now, we are able to perform any 3-cycle, and it remains to use them to put all pieces at the required places. Recall that we are able to do this for all pieces except the last two, and the correct ordering of the last two is guaranteed since we assume that our puzzle is solvable (the required permutation is even). First, we compute the required sequence of transpositions (and postpone the conversion into the output format):

```
def transpositions_solution(configuration):
    standard = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 0]
    transpositions = []
    current = list(configuration)
    for i in range(13): # two last pieces not processed
        if current[i] != standard[i]:
            idx = current.index(standard[i])
            assert idx > i
            # position idx should be moved to position i
            spare = i + 1 if i+1 != idx else i+2
            a, b, c = current[idx], current[i], current[spare]
            cycle_transpositions = do_3_cycle(current, a, b, c)
            transpositions += cycle_transpositions
        assert current[i] == standard[i]
    return transpositions
```

First, we copy the given configuration into `current`, and make the `transpositions` list (to be returned later) empty. Then, we fill correctly `current[0]`, ..., `current[12]`; the two last pieces `current[13]` and `current[14]` should be correctly filled (by 14 and 15 respectively), as explained above. To fill `current[i]` correctly (if it is not happened yet), we find the place `idx` where the desired piece happens to be now. It should be outside the part that is already filled or is to be filled (`idx > i`). Then, we find the spare position that is greater than `i` but is different from `idx`. For that we use `i+1` if it is different from `idx`, or the next position `i+2`. Finally, we find the labels `a`, `b`, `c` appearing in 3-cycle, compute the transpositions needed for this 3-cycle, apply them, and add them to the answer list.

The last step is to convert the answer (sequence of transpositions) to the required format, the list of labels for all moving pieces.

```
def solution(configuration):
    transpositions = transpositions_solution(configuration)
    answer = []
    current = list(configuration)
    for trans in transpositions:
        i, j = trans
        label = current[i] if current[i] != 0 else current[j]
        answer.append(label)
        apply_transposition(current, trans)
    return(answer)
```

We compute the required sequence of transpositions, copy the given configuration to `current`, and then convert every transposition `trans` to the required format. It exchanges positions `i` and `j`, we see which of them is not a hole, and append the corresponding label to the answer list.

**Stop and Think!** We did not check that the given permutation is even. What will our program return if it is odd (and the puzzle is not solvable)?

All the pieces except the two last ones will still be placed at their “standard” places, but the

remaining two pieces 14 and 15 will go in the wrong order (since the input permutation was odd). Thus, we will get the classical unsolvable position with 14 and 15 exchanged.

The approach described above leads to an efficient solution in practice, in the sense that it is reasonably fast. At the same time we have no guarantee that the number of moves is optimal (in most cases it uses a lot of moves). Finding the smallest number of moves is a more challenging problem. Google for “A\* search 15-puzzle” to find out the details of one particular approach. In general, for boards of arbitrary size (instead of the  $4 \times 4$  board), the problem of finding the optimal number of moves is [NP-complete](#) meaning that it is unlikely that it can be solved efficiently.

## 7. Appendix

### 7.1 Cutting a Figure

See Figure 7.1 for a solution to Problem 14. This is yet another example of a situation where a solution is easy to check, but not so easy to find.

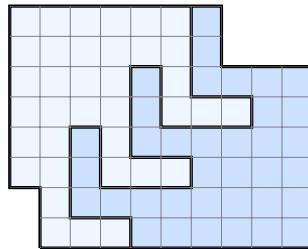


Figure 7.1: Solution to Problem 14.

### 7.2 Using SAT-solvers

In this section, we revisit Problem 61: implement an efficient program for placing  $n$  non-attacking queens on an  $n \times n$  board. We've seen that implementing a program that works in the blink of an eye, say, for  $n = 20$  is a non-trivial task. Here, we will implement such a program! To do this, we are going to use the so called *SAT-solvers* — programs for solving the *Boolean satisfiability problem*.

The *satisfiability* problem, commonly abbreviated as SAT, is one of the most important computational problems, with millions of theoretical and practical applications. One good reason for its wide applicability is that it provides a convenient *language* for stating various restrictions.

We introduce this language using a toy example. Assume that we have two 0/1-variables  $x_1$  and  $x_2$ . We might ask then whether it is possible to assign values to  $x_1$  and  $x_2$  to satisfy the following constraints:

- $(x_1 = 0 \text{ or } x_2 = 1)$ ;
- $(x_1 = 1 \text{ or } x_2 = 1)$ ;
- $(x_1 = 1 \text{ or } x_2 = 0)$ .

There are three constraints here and each of them says that *at least one* of the equalities must hold (it is perfectly fine if more than one inequality hold). A sequence of such constraints is called a *formula in conjunctive normal form* and each constraint is called a *clause*. Hence, in a sense, we are asking whether it is possible to *satisfy* the given clauses.

**Stop and Think!** Is it possible to satisfy the three clauses listed above?

Yes, it is not difficult to see that this is possible. To do this, assign  $x_1 = 1$  and  $x_2 = 1$ . Then, the first clause is satisfied since it contains  $x_2 = 1$ , the second clause is satisfied as both its equalities hold, and the third one is satisfied by  $x_1 = 1$ .

We will specify our formula in a program as follows:

`[-1, 2], [1, 2], [1, -2].`

This is just a list of clauses. If a clause contains  $x_i = 1$ , we include  $i$ . If a clause contains  $x_i = 0$ , we include  $-i$ .

**Stop and Think!** Is it possible to satisfy the following list of clauses?

`[-1, 2], [1, 2], [1, -2], [-1, -2].`

In fact, it is impossible. To see this, it is instructive to view each clause as *forbidding* a particular assignment to the variables it involves. For example, the clause `[-1, -2]` says that either  $x_1 = 0$  or  $x_2 = 0$  must hold. In other words, it forbids the assignment  $x_1 = 1, x_2 = 1$ . Then, it is not difficult to see that each of the four potential assignments to  $x_1, x_2$  is forbidden by one of the clauses.

In general, a clause may have an arbitrary length. In particular, it must not contain all the variables.

**Stop and Think!** Which of the following formulas (over variables  $x_1, x_2, x_3$ ) are satisfiable?

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

In Python, one can use the [pycosat](#) module for this task (this library is a Python wrapper for a particular SAT-solver called picosat).

```
from pycosat import solve

print(solve([[1, -2, -3], [3, -1], [2, 3]]))
print(solve([[-2, 3], [2, 1], [-3], [3, -1]]))

[-1, 2, -3]
UNSAT
```

This code shows that the first formula has a satisfying assignment  $x_1 = 0, x_2 = 1, x_3 = 0$ , whereas the second formula is unsatisfiable.

SAT is a difficult problem from theoretical point of view: we still don't know algorithms that are *provably* efficient. At the same time, somewhat surprisingly, state-of-the-art SAT-solvers are extremely efficient in practice. We will make use of this remarkable fact below.

After this short introduction to the Boolean satisfiability problem and SAT-solvers, we are ready to implement a program for the  $n$ -queens problem that will appear to be quite efficient! (The catch is that it will use an efficient SAT-solver as a black box.)

To do this, we will use  $n^2$  0/1-variables: for  $0 \leq i, j < n$ ,  $x_{ij} = 1$  if and only if a queen is placed into the cell  $(i, j)$  of the board. To ensure that these variables encode a valid placement of  $n$  queens, we pose the following constraints:

- For each  $0 \leq i < n$ , the  $i$ -th row contains at least one queen:

$$(x_{i1} = 1 \text{ or } x_{i2} = 1 \text{ or } \dots \text{ or } x_{i(n-1)} = 1).$$

This ensures that there *at least*  $n$  queens on the board.

- For each  $0 \leq i < n$ , the  $i$ -th row contains at most one queen. To state this, we use many clauses: for all  $0 \leq j_1 \neq j_2 < n$ , we add a clause

$$(x_{ij_1} = 0 \text{ or } x_{ij_2} = 0).$$

This ensures, that there are *at most*  $n$  queens. Together with the previous item, this gives that there are *exactly*  $n$  queens.

- For each  $0 \leq j < n$ , the  $j$ -th column contains at most one queen: for all  $0 \leq i_1 \neq i_2 < n$ , we add a clause

$$(x_{i_1j} = 0 \text{ or } x_{i_2j} = 0).$$

- Finally, we need to ensure that no two queens stay on the same diagonal. We do this naively: we just range over all pairs of cells  $(i_1, j_1)$  and  $(i_2, j_2)$  and if they stay on the same diagonal (that is, if  $|i_1 - i_2| = |j_1 - j_2|$ ), we add a clause

$$(x_{i_1j_1} = 0 \text{ or } x_{i_2j_2} = 0).$$

The following code implements this reasoning. It has one additional idea: we need to index variables by integers rather than pairs of integers. This is what the function `varnum` is responsible for: for every pair  $0 \leq i, j < n$  it gives a unique integer in the range  $\{1, 2, \dots, n^2\}$ .

```
from itertools import combinations, product
from pycosat import solve

n = 10
clauses = []

# converts a pair of integers into a unique integer
def varnum(i, j):
    assert i in range(n) and j in range(n)
    return i * n + j + 1

# each row contains at least one queen
for i in range(n):
    clauses.append([varnum(i, j) for j in range(n)])

# each row contains at most one queen
for i in range(n):
    for j1, j2 in combinations(range(n), 2):
        clauses.append([-varnum(i, j1), -varnum(i, j2)])

# each column contains at most one queen
for j in range(n):
    for i1, i2 in combinations(range(n), 2):
        clauses.append([-varnum(i1, j), -varnum(i2, j)])

# no two queens stay on the same diagonal
for i1, j1, i2, j2 in product(range(n), repeat=4):
    if i1 == i2:
        continue

    if abs(i1 - i2) == abs(j1 - j2):
        clauses.append([-varnum(i1, j1),
                        -varnum(i2, j2)])
```

```

assignment = solve(clauses)
for i, j in product(range(n), repeat=2):
    if assignment[varnum(i, j) - 1] > 0:
        print(j, end=' ')

```

This code works fast enough even for  $n = 50$ . To make it more efficient (so that it works fast, say, for  $n = 100$ ) you may want to improve the last for loop: instead of ranging through all pairs of cells, range through all cells, and for each cell, range through all other cells that stay on the same diagonal.

**Problem 186** Implement a program that solves any [Sudoku puzzle](#) using SAT-solvers.

We provide a solution below. It uses the following variables (to reduce to SAT):  $x_{ijk} = 1$  iff the cell  $(i, j)$  contains the digit  $k$ .

```

from itertools import combinations, product
import pycosat

def varnum(row, column, digit):
    assert row in range(1, 10) and column in range(1, 10)
    assert digit in range(1, 10)
    return 100 * row + 10 * column + digit

def exactly_one_of(literals):
    clauses = [[l for l in literals]]
    for pair in combinations(literals, 2):
        clauses.append([-l for l in pair])
    return clauses

def one_digit_in_every_cell():
    clauses = []
    for row, column in product(range(1, 10), repeat=2):
        clauses += exactly_one_of([varnum(row, column, digit)
                                   for digit in range(1, 10)])
    return clauses

def one_digit_in_every_row():
    clauses = []
    for row, digit in product(range(1, 10), repeat=2):
        clauses += exactly_one_of([varnum(row, column, digit)
                                   for column in range(1, 10)])
    return clauses

def one_digit_in_every_column():
    clauses = []
    for column, digit in product(range(1, 10), repeat=2):
        clauses += exactly_one_of([varnum(row, column, digit)
                                   for row in range(1, 10)])
    return clauses

def one_digit_in_every_block():
    clauses = []

```

```

for row, column in product([1, 4, 7], repeat=2):
    for digit in range(1, 10):
        clauses += exactly_one_of(
            [varnum(row + a, column + b, digit)
             for (a, b) in product(range(3), repeat=2)])
    )
return clauses

def solve_puzzle(puzzle):
    assert len(puzzle) == 9
    assert all(len(row) == 9 for row in puzzle)

    clauses = []
    clauses += one_digit_in_every_cell()
    clauses += one_digit_in_every_row()
    clauses += one_digit_in_every_column()
    clauses += one_digit_in_every_block()

    for row, column in product(range(1, 10), repeat=2):
        if puzzle[row - 1][column - 1] != "*":
            digit = int(puzzle[row - 1][column - 1])
            assert digit in range(1, 10)
            clauses += [[varnum(row, column, digit)]]

    solution = pycosat.solve(clauses)
    if isinstance(solution, str):
        print("No solution")
        exit()

    assert isinstance(solution, list)

    for row in range(1, 10):
        for column in range(1, 10):
            for digit in range(1, 10):
                if varnum(row, column, digit) in solution:
                    print(digit, end="")
            print()

difficult_puzzle = [
    "8*****",
    "***36***",
    "**7**9*2**",
    "**5***7***",
    "****457**",
    "***1***3*",
    "***1***68",
    "***85***1*",
    "***9***4**"
]
solve_puzzle(difficult_puzzle)

```

### 7.3 Using ILP-solvers

*Integer linear programming*, or ILP, is another difficult algorithmic problem. Its input consists of a list of linear inequalities over integer variables and the goal is to optimize (i.e., find either the

minimum value or the maximum value) an objective linear function.

Again, we illustrate this by a toy example. Let's revisit Problem 47 where the goal is to find an optimal chocolate production plan. In mathematical terms, the goal is to find the maximum value of  $10M + 30D$  where  $M, D$  are integers satisfying the following three inequalities (see (2.1)–(2.3)):

$$M \leq 500, \quad D \leq 200, \quad M + D \leq 600.$$

What we currently know about ILP is similar to SAT: ILP-solvers are quite efficient in practice, though we don't know how to prove that they *always* find an optimum solution quickly.

We use our working chocolate example to introduce the input format of the `mip` library.

```
from mip import *

model = Model(solver_name='cbc')
model.verbose = False

m = model.add_var(var_type=INTEGER)
d = model.add_var(var_type=INTEGER)

model += (m <= 500)
model += (d <= 200)
model += (m + d <= 600)

model.objective = maximize(10 * m + 30 * d)
model.optimize()

print(model.objective_value)
```

10000.0

Let's see another interesting application of ILP-solvers. In Problem 62, the goal is to place 16 non-intersecting diagonals in a  $5 \times 5$  grid. This can be done as follows. The grid has 36 nodes: let's index them with pairs of integers  $(0,0), (0,1), \dots, (5,5)$ . For every two nodes  $(i,j)$  and  $(k,l)$  that are opposite corners of the same cell we introduce a 0/1-variable  $x_{ijkl}$ :  $x_{ijkl} = 1$  if we draw a diagonal joining these two cells,  $x_{ijkl} = 0$  otherwise. Then, if two diagonals intersect, we just write down an inequality stating that the sum of the corresponding two  $x$ 's is at most one. To do this, we employ the `intersects` method from the `shapely` library. Finally, we tell the solver that we would like to maximize the number of diagonals.

```
from itertools import combinations, product
from mip.model import *
from shapely.geometry import LineString

n = 5

model = Model(solver_name='cbc')
model.verbose = False

segments = []
for i, j in product(range(n + 1), repeat=2):
    for delta_i, delta_j in product((-1, 1), repeat=2):
        if i + delta_i in range(n + 1) and \
           j + delta_j in range(n + 1):
            segments[i, j, i + delta_i, j + delta_j] = \
                model.add_var(var_type=mip.BINARY)
```

```

for s1, s2 in combinations(segments, 2):
    if LineString([s1[:2], s1[2:]]).intersects(
        LineString([s2[:2], s2[2:]])):
        model += segments[s1] + segments[s2] <= 1

model.objective = maximize(xsum(segments.values()))
model.optimize()

print(model.objective_value)
for s in segments:
    if abs(segments[s].x) > 1e-6:
        print(s)

16.0
(0, 1, 1, 0)
(0, 2, 1, 3)
(0, 4, 1, 5)
(1, 1, 2, 0)
(1, 2, 2, 3)
(1, 4, 0, 3)
(2, 1, 3, 0)
(3, 1, 2, 2)
(3, 2, 4, 3)
(3, 3, 2, 4)
(3, 4, 2, 5)
(3, 5, 4, 4)
(4, 2, 5, 3)
(5, 1, 4, 0)
(5, 2, 4, 1)
(5, 4, 4, 5)

```

**Problem 187** Tweak the code above to solve Problem 64 where the goal is to draw 21 non-intersecting diagonals of length  $\sqrt{5}$  in a  $6 \times 6$  grid.

In our final example, we show that ILP-solvers are also suitable for *existence* (or *decision*) problems rather than just *optimization* problems. Recall that in Section 1.2.4, we proved that in the following list of thirty integers it is possible to color some numbers red and some others blue such that the sums of red and blue numbers coincide (see Problem 29).

```

from random import randint, seed

seed(14)

for i in range(30):
    print(randint(10 ** 6, 10 ** 7 - 1), end=' ')
    if i % 6 == 5:
        print()

```

```

2792285 9843470 5142886 5548558 5291201 5882438
2218459 8545410 6083294 8828556 7655079 7607029
2986415 5421072 4745783 6295863 7006791 5368826
7051040 9661551 3511608 3701978 5619271 3767372
1177927 2173344 3064039 6655032 1466196 2395998

```

We proved this *non-constructively*: we showed that such a coloring exists without providing it explicitly. Now, we are going to use ILP-solvers to find this coloring!

Let  $S = \{a_1, \dots, a_{30}\}$  be the considered set of integers. To find two non-empty and disjoint subsets  $X, Y \subseteq S$  with equal sums, we use sixty 0/1-variables:  $x_1, \dots, x_{30}$  and  $y_1, \dots, y_{30}$ . These are *indicator* variables:  $x_i = 1$  if and only if  $a_i \in X$  (and similarly for  $y_i$  and  $Y$ ). This is how one can write all the required constraints:

- $X$  and  $Y$  are non-empty (in set theoretic notation:  $X \neq \emptyset, Y \neq \emptyset$ ):

$$x_1 + x_2 + \dots + x_{30} \geq 1 \text{ and } y_1 + y_2 + \dots + y_{30} \geq 1.$$

- $X$  and  $Y$  are disjoint (in set theoretic notation:  $X \cap Y = \emptyset$ ): for any  $i = 1, 2, \dots, 30$ ,

$$x_i + y_i \leq 1.$$

- the sums of  $X$  and  $Y$  are equal (in set theoretic notation:  $\sum_{a \in X} a = \sum_{b \in Y} b$ ):

$$a_1(x_1 - y_1) + a_2(x_2 - y_2) + \dots + a_{30}(x_{30} - y_{30}) = 0.$$

```
from mip import *
from random import randint, seed

seed(14)

n = 30
numbers = [randint(10 ** 6, 10 ** 7 - 1)
           for _ in range(n)]

model = Model(solver_name=CBC)
model.verbose = False

x = [model.add_var(var_type=BINARY) for _ in range(n)]
y = [model.add_var(var_type=BINARY) for _ in range(n)]
model += xsum(x) >= 1
model += xsum(y) >= 1

for i in range(n):
    model += (x[i] + y[i]) <= 1

model += (xsum(numbers[i] * (x[i] - y[i]))
          for i in range(n)) == 0

model.optimize()

one = [numbers[i] for i in range(n) if x[i].x > 0.1]
two = [numbers[i] for i in range(n) if y[i].x > 0.1]

assert sum(one) == sum(two)

print(*one)
print(*two)
```

```
5548558 5368826 7051040 9661551 3511608 2173344 3064039
2792285 8545410 8828556 7607029 2986415 5619271
```

## 7.4 Visualizing Football Fans

The code below produces a visualization for Problem 149. It generates a random population with 31 people where every person has 18 random friends and supports a random team. It then starts

the switching process and produces a new picture every time when somebody switches. Finally, all the pictures are compiled into a video (for this to work, you need [ImageMagick](#) installed). Try running the code with various parameters and see what happens!

```
import networkx as nx
from os import system
from random import randint, seed

def prepare_population(number_of_people,
                      number_of_friends):
    seed(17)
    graph = nx.random_regular_graph(
        number_of_friends, number_of_people)

    population = nx.nx_agraph.to_agraph(graph)
    population.layout(prog='circo')

    for person in population.nodes():
        population.get_node(person).attr['style'] = \
            'filled'
        set_team(population, person, randint(0, 1))

    for u, v in population.edges():
        population.get_edge(u, v).attr['penwidth'] = 2

    return population

def set_team(population, person, team):
    assert team in range(2)
    population.get_node(person).attr['team'] = team

def get_team(population, person):
    return int(population.get_node(person).attr['team'])

def draw_population(population, idx):
    team_colors = ['deepskyblue', 'lightblue']
    for person in population.nodes():
        team = get_team(population, person)
        population.get_node(person).attr['fillcolor'] = \
            team_colors[team]

    for u, v in population.edges():
        if get_team(population, u) != \
           get_team(population, v):
            population.get_edge(u, v).attr['style'] = \
                'dotted'
        else:
            population.get_edge(u, v).attr['style'] = \
                'solid'

    population.draw(f'pic{idx:03}.png')
```

```
if __name__ == '__main__':
    population = prepare_population(31, 18)

    has_changed = True
    idx = 0
    while has_changed:
        draw_population(population, idx)
        has_changed = False

        for person in population.nodes():
            team = get_team(population, person)
            same = [u for u in
                    population.neighbors(person)
                    if get_team(population, u) == team]
            other = [u for u in
                    population.neighbors(person)
                    if get_team(population, u) != team]

            if len(other) > len(same):
                set_team(population, person, 1 - team)
                has_changed = True
                idx += 1
                break

    system('ffmpeg -r 1 -i pic%03d.png '
           '-pattern_type glob -pix_fmt yuv420p -r 1 '
           'fans.mp4')
    # system('rm pic*.png')
```

# Combinatorics and Probability

|           |   |     |
|-----------|---|-----|
| <b>8</b>  | <b>Basic Counting .....</b>             | 161 |
| 8.1       | Starting to Count                       |     |
| 8.2       | Recursive Counting                      |     |
| 8.3       | Tuples and Permutations                 |     |
| <b>9</b>  | <b>Binomial Coefficients .....</b>      | 181 |
| 9.1       | Number of Games in a Tournament         |     |
| 9.2       | Combinations                            |     |
| 9.3       | Binomial Theorem                        |     |
| 9.4       | Practice Counting                       |     |
| <b>10</b> | <b>Advanced Counting .....</b>          | 197 |
| 10.1      | Review                                  |     |
| 10.2      | Combinations with Repetitions           |     |
| 10.3      | Practice Counting                       |     |
| <b>11</b> | <b>Probability .....</b>                | 211 |
| 11.1      | What is Probability?                    |     |
| 11.2      | Probability: Do's and Don'ts            |     |
| 11.3      | Conditional Probability                 |     |
| 11.4      | Monty Hall Paradox                      |     |
| <b>12</b> | <b>Random Variables .....</b>           | 259 |
| 12.1      | Random Variables and Their Expectations |     |
| 12.2      | Linearity of Expectation                |     |
| 12.3      | Expectation is Not All                  |     |
| 12.4      | Markov's Inequality                     |     |
| <b>13</b> | <b>Dice Games .....</b>                 | 275 |
| 13.1      | Dice Game Problem                       |     |
| 13.2      | Optimal Strategy                        |     |



## 8. Basic Counting

Counting is one of the basic mathematically related tasks we encounter on a daily basis. The main question here is the following: can we count the number of objects without listing all of them? This question arises naturally in various scenarios both in real life and in Computer Science. What is the number of phone numbers or license plates? What is the number of combinations one needs to brute force in order to crack a password? Is there a way to tell that an algorithm will run in a reasonable time before implementing and running it? All these questions are addressed by a mathematical field called *combinatorics*.

In this chapter, we consider the basic building blocks of combinatorics. All of them are easy to understand and at the same time are powerful enough to handle various non-trivial questions. To help you to develop an intuition, we consider short Python code snippets for generating the objects to be counted.

### 8.1 Starting to Count

#### 8.1.1 Warm Up

To whet your appetite, we start by stating a few counting problems. In the following sections, we will introduce the basic combinatorial toolbox and solve these problems together.

**Problem 188** A license plate in Portugal contains a block of two letters followed by a block of two digits followed by another block of two letters, see Figure 8.1. What is the number of license plates?



Figure 8.1: Portuguese license plate contains a block of two letters (AA), a block of two digits (00), and another block of two letters (AA). (Source: [Wikipedia](#).)

**Problem 189** You are creating an account at a web service. The service asks you to set up a strong password:

- it must contain either eight or nine symbols;
- each symbol must be either a Latin letter (lower or upper case) or a digit;
- it must contain at least one digit.

For example, A2Jp5eLJg, 7m5a5th6, and discr7ete are strong passwords, whereas qwer-tyui, math17, discr-ete, and ghBTrehY are not. What is the number of strong passwords?

### 8.1.2 Sum Rule

If your bag contains apples and oranges, the total number of fruits in the bag is the sum of the number of apples and the number of oranges. This trivial remark is called sometimes the *sum rule* and can appear in different situations. The total length of a playlist is the sum of the lengths for the individual tracks (if there are no pauses). If a program consists of two consecutive blocks and the two blocks make  $n$  and  $k$  steps, respectively, then the whole program makes  $n + k$  steps. For example, the following Python code prints the word “Hi!” eight times (since the two `for` loops perform three and five iterations).

```
for _ in range(3):
    print('Hi!')
for _ in range(5):
    print('Hi!')
```

If at some moment the format of the license plates changes and temporarily both old and new license plates are valid, then the total amount of possible valid plates is the sum of the number of old-style plates and new-style ones.

In fact, we have already used the sum rule implicitly in the book. Namely, in Problem 151, we considered a piece on a chessboard that, in one move, can move either one cell to the left or one cell to the right. Our goal was to prove that if after several moves the piece returned to its original position, then the total number of moves is even. We argued as follows. The total number of moves is the sum of two numbers: left moves and right moves. As the piece returned to the original position, it made as many right moves as left moves. If this number is  $k$ , the total number of moves is  $k + k = 2k$  and therefore is even.

**Stop and Think!** In one move, a piece on a chessboard moves either one cell up or one cell to the right. How many moves are needed to get from the bottom left corner to the position shown in Figure 8.2 (that is, from the cell (1, 1) to the cell (4, 6))?

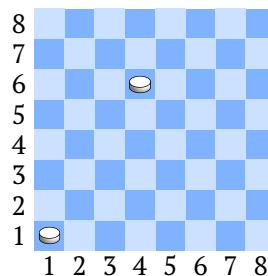


Figure 8.2: How many moves are needed to get from the cell (1, 1) to the cell (4, 6)?

By a *path* let us denote a sequence of moves from (1, 1) to (4, 6) where every move is either up or to the right. Our goal is then to find the number of moves in a path. Three examples are given in Figure 8.3. Each of the three paths contains eight moves. This is not a coincidence: any other path also consists of exactly eight moves. And there is a straightforward explanation for this: to get

to the cell  $(4, 6)$  from the cell  $(1, 1)$ , one needs to reach the fourth column and the sixth row. To do this, one needs three moves to the right and five moves up.

Here we again apply the sum rule — not to apples and oranges, but to moves to the right and moves up.

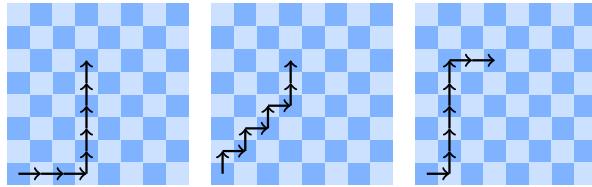


Figure 8.3: There are many paths from  $(1, 1)$  to  $(4, 6)$ . All of them contain exactly eight moves.

Figure 8.3 shows three paths from  $(1, 1)$  to  $(4, 6)$ . What is the total number of paths? This is a more difficult question (compared to the counting of moves), but we will learn soon how to find this number!

This simple reasoning can be summarized as follows.

**Sum rule** If there are  $n$  objects of the first type and there are  $k$  objects of the second type, then there are  $n + k$  objects of one of two types.

**Problem 190** What is the number of integers from 1 to 10 that are divisible by 2 or 3? Try it: Coursera ↗

Well, this is easy. There are five integers divisible by 2 (2, 4, 6, 8, 10) and three integers divisible by 3 (3, 6, 9). Hence, by the sum rule, the answer is  $5 + 3 = 8$ .

**Stop and Think!** Do you see a flaw in this argument?

Let's just find these numbers.

```
print([n for n in range(1, 11) if n % 2 == 0 or n % 3 == 0])
[2, 3, 4, 6, 8, 9, 10]
```

Thus, the correct answer is seven. Looking back at our argument, we see that it is the number 6 that causes problems: it is divisible by both 2 and 3 and because of that we counted it twice. The lesson learned here is: *when applying the sum rule, one should ensure that no object belongs to both classes*.

Mathematicians formulate the sum rule as follows: the cardinality of the union of two disjoint sets is the sum of their cardinalities. As you may guess, the sets are formed by apples and oranges, the union is what happens when we put them in the same bag, and disjointness means that we do not count the same object twice (as we first did in our last example).

Philosophers would add that the sum rule is not a *rule*, but a *definition*: the definition of addition operation.

### 8.1.3 Convenient Language: Sets

In this section, we introduce the *set language*. This is one of the most important notation, used almost everywhere in mathematics. Needless to say, Python has a built-in type [set](#) ↗ that supports most of the standard set operations. Below, we give the definitions and complement them with examples and Python code. We'll see later that the common rules of combinatorics are stated cleanly and compactly using this language.

A *set* is a collection of elements in which each element appears at most once. A natural way to specify a set is to list its elements, separated by commas, in curly brackets. Below, we show

two examples of sets containing integers, though in general a set may contain any objects (not just integers).

$$A = \{5, 2, 8, 17\}, \quad B = \{3, 17, 2, 19, 6\}.$$

Sets are *unordered* meaning that the order of elements are not important:  $\{17, 2, 5, 8\}$  is the same set as  $A$ . Also,  $\{2, 5, 2, 17, 8, 5\}$  is the same set as  $A$ , too, as each element appears at most one in a set.

The *size*, or *cardinality*, of a set is the number of elements in it. It is denoted by vertical bars:

$$|A| = 4, \quad |B| = 5, \quad |\{7, 2, 7\}| = 2.$$

A set may have an infinite size. One such example is the set of all integers:

$$\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}.$$

Also, there is a unique *empty set*, denoted by  $\emptyset$  (or  $\varnothing$ ): this set contains no elements and its size is equal to zero. We use the  $\in$  symbol to indicate that an element *belongs* to a set (or *lies* in a set):

$$2 \in A, 3 \notin A, 2 \in B, 5 \notin B.$$

When an element belongs to a set, we also say that this set *contains* this element.

The basic set operations are *union*, denoted by  $\cup$ , and *intersection*, denoted by  $\cap$ : the union of two sets consists of all elements from these two sets (if some element appears in both sets, it still appears just once in the union, as a set never contains duplicates), the intersection of two sets contains only elements that belong to *both* of the sets.

$$A \cup B = \{5, 2, 8, 17, 3, 19, 6\}, \quad A \cap B = \{17, 2\}.$$

**Stop and Think!** Practice finding the intersection and the union of two sets.

- Find the intersection and the union of two sets  $A = \{1, 2, 4, 6, 7\}$  and  $B = \{1, 2, 3, 5, 7\}$ .
- Let  $A = \{1, 2, 4, 6, 7\}$  and  $B$  be the set of all even positive integers. Find the size  $|A \cap B|$  of the intersection of  $A$  and  $B$ .

Try it: [Coursera](#) ↗.

Using these two operations, one may construct new sets out of the existing ones. Another way of getting a new set is through a *set builder notation*:

$$C = \{x \in A : x \text{ is odd}\} = \{5, 17\},$$

$$D = \{x \in B : x \notin A\} = \{3, 19, 6\}.$$

Two sets are called *disjoint* if they share no elements (that is, if their intersection is the empty set). We say that one set is a subset of another, denoted by the symbol  $\subseteq$ , if all elements of the former set belong to the latter set as well:

$$C \subseteq A, \quad C \not\subseteq B.$$

The *difference*  $X \setminus Y$  (also denoted as  $X - Y$ ) is the set of those elements of  $X$  that do not belong to  $Y$ :

$$X \setminus Y = \{x \in X : x \notin Y\}.$$

For example,

$$A \setminus B = \{5, 8\}, \quad B \setminus A = \{3, 19, 6\}.$$

Python has a built-in type [set](#) ↗ that supports most of the standard set operations.

```

a = {5, 2, 8, 17, 2}
b = {3, 17, 2, 19, 6, 17}

print(f'Duplicates are removed automatically:')
print(f'a={a}')
print(f'b={b}')
print()

print(f'Size of {a} is {len(a)}')
print(f'Size of {b} is {len(b)}')
print()

print(f'2 belongs to {a}: {2 in a}')
print(f'5 belongs to {a}: {3 in a}')
print(f'2 belongs to {b}: {2 in b}')
print(f'5 belongs to {b}: {5 in b}')
print()

print(f'Union of {a} and {b}: {a.union(b)}')
print(f'Intersection of {a} and {b}: {a.intersection(b)}')
print()

print(f'Set building:')
print(f'Set of odd numbers of {a} is {set(x for x in a if x % 2 == 1)}')
print(f'Set of numbers from {b} that do not belong to {a}:\n'
      f' {set(x for x in b if x not in a)}')
print()

print(f'Difference of {a} and {b}: {a-b}')

```

```

Duplicates are removed automatically:
a={8, 17, 2, 5}
b={17, 2, 3, 19, 6}

Size of {8, 17, 2, 5} is 4
Size of {17, 2, 3, 19, 6} is 5

2 belongs to {8, 17, 2, 5}: True
5 belongs to {8, 17, 2, 5}: False
2 belongs to {17, 2, 3, 19, 6}: True
5 belongs to {17, 2, 3, 19, 6}: False

Union of {8, 17, 2, 5} and {17, 2, 3, 19, 6}: {2, 3, 5, 6, 8, 17, 19}
Intersection of {8, 17, 2, 5} and {17, 2, 3, 19, 6}: {17, 2}

Set building:
Set of odd numbers of {8, 17, 2, 5} is {17, 5}
Set of numbers from {17, 2, 3, 19, 6} that do not belong to {8, 17, 2, 5}:
{19, 3, 6}

Difference of {8, 17, 2, 5} and {17, 2, 3, 19, 6}: {8, 5}

```

Even if Python allows this, it is a good taste not to list the same element twice in the curly brackets. But sometimes you have no other option if you do not know *a priori* whether the elements are the same or not.

**Problem 191** What is the cardinality of the set  $\{x, -x, x^2\}$  for different values of real number  $x$ ? When is it maximal and when is it minimal?

### 8.1.4 Generalized Sum Rule

After introducing the set language, we can restate the sum rule in this language.

**Sum rule** The size of the union of two *disjoint* sets is equal to the sum of their sizes: if a set  $A$  contains  $k$  elements, a set  $B$  contains  $n$  elements and these sets do not have common elements, then the set  $A \cup B$  has  $n + k$  elements.

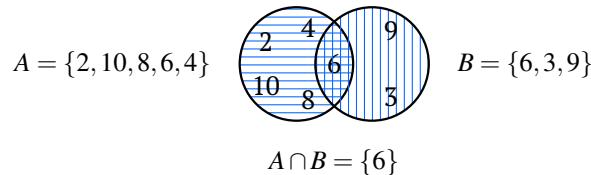
The following toy example shows two disjoint sets (the first one contains two strings, the second one contains three integers) and their union.

```
first_set = {'e', 'u'}
second_set = {8, 2, 5}

print(first_set | second_set)
{2, 5, 'u', 'e', 8}
```

Note that the union operation is also denoted by `|` in Python.

Recall from our previous discussion that it is essential that  $A$  and  $B$  are *disjoint*. For example, one cannot apply the sum rule to the following two sets:  $A$  is the set of all integers from 1 to 10 divisible by 2 and  $B$  is the set of all integers from 1 to 10 divisible by 3. The reason is that these two sets are not disjoint:  $6 \in A$  and  $6 \in B$ . One could picture this as follows:



If we count all elements from  $A$  and then count all elements from  $B$ , then 6 will be counted twice. This suggests a way to generalize the sum rule.

**Sum rule** The size of the union of two sets is equal to the sum of their sizes minus the size of their intersection:

$$|A \cup B| = |A| + |B| - |A \cap B|. \quad (8.1)$$

In particular, if  $A$  and  $B$  are disjoint, then  $|A \cap B| = 0$  and we get our initial version of the formula:  $|A \cup B| = |A| + |B|$ .

**Problem 192** How many integers from 1 to 1000 are divisible by 2 or by 3? Try it: [Coursera](#) ↗

**Problem 193** There are 25 students in the class. Each student should take at least one of the two foreign languages, French and German, and students are allowed to take both courses. It is known that 17 students signed up for the French language course and 14 students signed up for German language course. How many students signed up for both courses? Try it: [Coursera](#) ↗

The size of the union of three sets can be found using a similar formula:

$$\begin{aligned} |A \cup B \cup C| &= |A| + |B| + |C| \\ &\quad - |A \cap B| - |A \cap C| - |B \cap C| \\ &\quad + |A \cap B \cap C|. \end{aligned}$$

The two formulas above are special cases of the [inclusion-exclusion principle](#) ↗ that shows how to find the union of  $n$  sets  $A_1, \dots, A_n$ :

$$\left| \bigcup_{i=1}^n A_i \right| = \sum_{\emptyset \neq I \subseteq [n]} (-1)^{|I|+1} \left| \bigcap_{i \in I} A_i \right|.$$

(As usual with the gray boxes in this book, don't worry if not everything is clear to you here.)

**Problem 194** Prove the inclusion-exclusion principle for three sets.

To see why

$$\begin{aligned}|A \cup B \cup C| &= |A| + |B| + |C| \\&\quad - |A \cap B| - |A \cap C| - |B \cap C| \\&\quad + |A \cap B \cap C|,\end{aligned}$$

let us see how many times some element  $x$  is taken into account in the left hand and right hand size. Left hand side ignores elements that are not in  $A \cup B \cup C$ , i.e., are neither in  $A$ , nor in  $B$ , nor in  $C$ . The right hand side also ignores them (for obvious reasons).

But there are also elements  $x$  that belong to one of the sets  $A, B, C$ , or to exactly two of these sets, or even to all three of them. All three types of elements are counted once in the left hand side. But what about the right side?

If  $x$  belongs to exactly one of the sets, say,  $A$ , it will be counted only when we consider  $|A|$  (since  $B$  and  $C$  do not contain  $x$ ; neither do the intersections).

If  $x$  belongs to exactly two of the sets, say  $A$  and  $B$ , it will be counted twice in the first line (for  $A$  and  $B$ ), but then discounted back in the second line (once, for  $A \cap B$ ), and will not be counted in the third line.

Finally, if  $x$  belongs to all three  $A, B, C$ , then it will be counted 3 times, discounted three times and again counted once, and we get what we need:  $3 - 3 + 1 = 1$ .

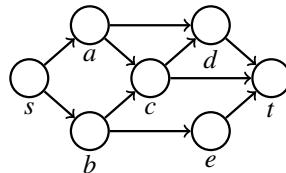
**Stop and Think!** Can you do a similar analysis for four sets?

For the general case we need a more general argument; one of the options is to use identities for binomial coefficients that we consider in Section 9.2.

## 8.2 Recursive Counting

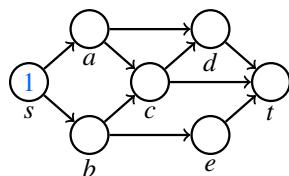
### 8.2.1 Number of Paths

**Stop and Think!** There are many ways of getting from  $s$  to  $t$  in the network below: for example,  $s \rightarrow b \rightarrow e \rightarrow t$  and  $s \rightarrow a \rightarrow c \rightarrow d \rightarrow t$ . What is the total number of ways?

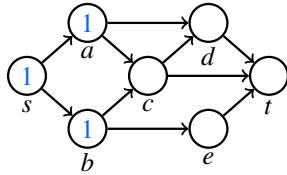


Try it: Coursera [↗](#).

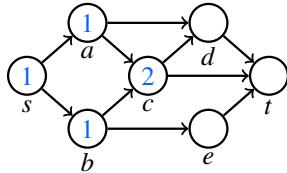
Since we start from  $s$ , there is a unique way to get to  $s$ . Let's write this down:



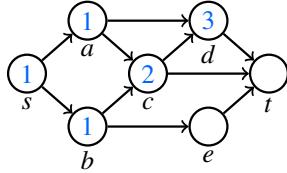
For  $a$  and  $b$ , there is also just a single path.



Since there is only one path to  $a$  and only one path to  $b$ , the number of paths to  $c$  is  $1 + 1 = 2$  ( $s \rightarrow a \rightarrow c$  and  $s \rightarrow b \rightarrow c$ ).

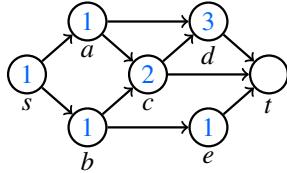


Similarly, to get to  $d$  one needs to get to either  $a$  or  $c$ . There is one path to get to  $a$  and two paths to get to  $c$ . Hence, the number of paths to get to  $d$  is  $1 + 2 = 3$  ( $s \rightarrow a \rightarrow d$ ,  $s \rightarrow a \rightarrow c \rightarrow d$ , and  $s \rightarrow b \rightarrow c \rightarrow d$ ).

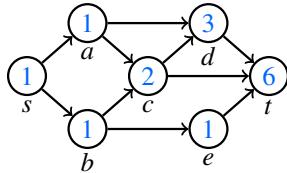


What we use here (and above) is the sum rule: any path to  $d$  goes to  $d$  from either  $a$  or  $c$ . Thanks to our recursive approach, these two numbers of paths are already counted.

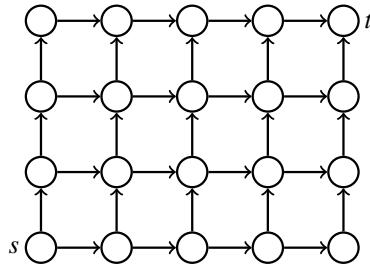
The number of paths ending in  $e$  is equal to 1 as  $e$  can be reached from  $b$  only.

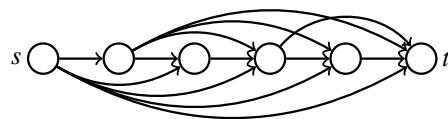


Finally, there are three possibilities to get to  $t$ : from  $c$ ,  $d$ , or  $e$ . Hence, the number of ways to get to  $t$  is equal to  $2 + 3 + 1 = 6$ .



**Problem 195** Find the number of ways to get from  $s$  to  $t$  in the following networks.





Try it: [Coursera](#).

Later in the book, we will learn how to work with *graphs*. Built-in libraries for manipulating graphs allow to enumerate paths in just few lines of code. As usual, what is nice about the code below is that it is interactive: change the graph, run the code, and see what happens (note that it also draws the graph!).

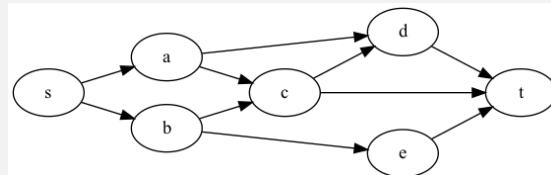
```
from networkx import DiGraph, all_simple_paths, nx_agraph

edges = [('s', 'a'), ('s', 'b'), ('a', 'c'), ('a', 'd'), ('b', 'c'),
          ('b', 'e'), ('c', 'd'), ('c', 't'), ('d', 't'), ('e', 't')]

graph = DiGraph(edges)
for path in all_simple_paths(graph, source='s', target='t'):
    print(*path, sep='->')

agraph = nx_agraph.to_agraph(graph)
agraph.graph_attr.update(rankdir='LR')
agraph.draw('network.png', prog='dot')

s->a->c->d->t
s->a->c->t
s->a->d->t
s->b->c->d->t
s->b->c->t
s->b->e->t
```



To run the code, one needs to install the modules [networkx](#) and [pygraphviz](#). The first one is responsible for manipulating graphs, the second one is responsible for drawing graphs nicely.

### 8.2.2 Product Rule

**Stop and Think!** Let  $A = \{e, u\}$  be a set of two letters and  $B = \{8, 2, 5\}$  be a set of three digits. What is the number of ways to select one element from  $A$  and one element from  $B$ ?

In Python, one can model this process with two nested `for` loops: we first select a letter from  $A$  and then select a digit from  $B$ .

```
for letter in {'e', 'u'}:
    for digit in {8, 2, 5}:
        print(letter, digit)
```

```
e 8
e 2
```

```
e 5
u 8
u 2
u 5
```

An alternative and more compact way to achieve this is to use the `product` function from the `itertools` module.

```
from itertools import product

for t in product({'e', 'u'}, {8, 2, 5}):
    print(*t)
```

```
e 8
e 2
e 5
u 8
u 2
u 5
```

As both code snippets reveal, the answer is 6. This is what the product rule in combinatorics is responsible for.

**Product rule** If there are  $k$  objects of the first type and there are  $n$  objects of the second type, then there are  $k \times n$  ways to select one object of the first type and one object of the second type.

In set theory language, it is stated as follows. For sets  $A$  and  $B$ , their product is the set

$$A \times B = \{(a, b) : a \in A, b \in B\}$$

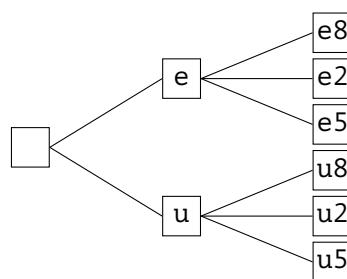
of all pairs whose first element is drawn from  $A$  and second element is drawn from  $B$ . The product rule says that the size of the product of two finite sets is equal to the product of their sizes:

$$|A \times B| = |A| \cdot |B|.$$

The following table for our previous toy example explains the reason behind the rule: one could organize all the pairs into a table whose rows are indexed by the objects of the first type and whose columns are indexed by the objects of the second type.

|   |    |    |    |
|---|----|----|----|
|   | 8  | 2  | 5  |
| e | e8 | e2 | e5 |
| u | u8 | u2 | u5 |

It is also instructive to view the process of selecting a pair by going along the following tree. Starting at the root (the leftmost node), we select one of two letters (e or u) and arrive at a node on the next level. There, we select one of three digits (8, 2, or 5). Hence, the number of *leaves* in this tree is equal to six.



**Problem 196** Apply the sum and product rules to find out the number of times the word Hi! is printed by each of the following three code snippets.

```
for _ in range(5):
    for _ in range(7):
        print('Hi!')
```

```
for _ in range(3):
    for _ in range(4):
        for _ in range(5):
            print('Hi!')
```

```
for _ in range(2):
    for _ in range(6):
        print('Hi!')

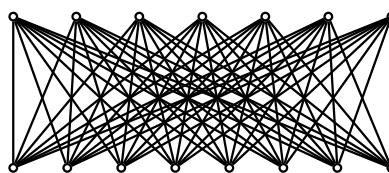
for _ in range(3):
    for _ in range(5):
        print('Hi!')
```

Try it: [Coursera](#).

**Problem 197** There are five sport teams in a tournament, each consisting of 8 players. We would like to choose one of the players from each of the teams to lead an opening ceremony. How many ways do we have to do it? Try it: [Coursera](#).

**Problem 198** You are picking a t-shirt in a store. A t-shirt comes in 5 possible colors, 7 possible sizes, and 2 neck types (crew neck and v-neck). How many possible variants of a t-shirt are there? Try it: [Coursera](#).

**Problem 199** What is the number of segments in the picture below? (Each segment joins two circles and every top circle is connected to every bottom circle.)



Try it: [Coursera](#).

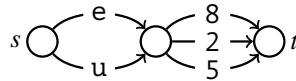
### 8.2.3 Back to Recursive Counting

**Stop and Think!** Find the number of ways to get from  $s$  to  $t$ .



(Different arrows mean different paths even if they connect the same pair of vertices.)

The answer is six: one can arrive at the middle point in two ways and, for each of these ways, there are three ways to continue to  $t$ . What we use here is the product rule! This rule has a natural interpretation in terms of the number of paths in an appropriate network. Let us visualize this using our previous toy example.



Here, we label the left pair of arrows and the right triple of arrows by the elements of the sets  $A = \{e, u\}$  and  $B = \{8, 2, 5\}$ , respectively. Then,  $s - t$  paths in the network above are in *one-to-one correspondence* with pairs  $(a, b)$  where  $a \in A$  and  $b \in B$ :

- every pair  $(a, b)$  defines an  $s - t$  path: first, follow an arrow labeled by  $a$ , then continue by arrow labeled by  $b$ ;
- every  $s - t$  path defines a pair  $(a, b)$ : the first arrow of the path gives a letter  $a$ , the second one gives a digit  $b$ .

Mathematicians also use the name *bijection* for one-to-one correspondences. They would say that we constructed a bijection between the set of paths and the set of pairs. This bijection, for example, maps the uppermost path (that selects the upper arrow on both steps) to the pair  $(e, 8)$ . It guarantees that the number of paths is the same as the number of pairs.

We've already seen the trick with one-to-one correspondence: in the beginning of the book, we discussed that to convince yourself that the number of people in a room is equal to the number of chairs, you ask everybody to take a seat. Then, if each person is seated and no chair is empty, the two numbers are equal.

**Bijection rule** A bijection between two sets  $A$  and  $B$  proves that the cardinalities of  $A$  and  $B$  are the same.

Sometimes this rule allows us to see that two sets have the same number of elements without finding that number.

**Problem 200** Consider 10 points on a circle. One can draw many triangles whose vertices are chosen among these points. Also one can draw many 7-gons whose vertices are chosen among these points. We will learn later how to count these triangles and 7-gons, but the bijection principle guarantees that the numbers of triangles and 7-gons are the same. Do you see why? (Figure 8.4 gives an example and a hint.)

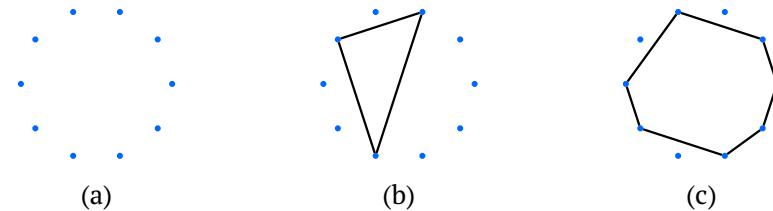
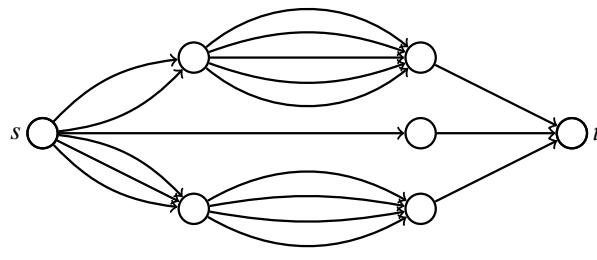


Figure 8.4: (a) Ten points on a circle. (b) A triangle. (c) A 7-gon.

Let us finish this section with one more simple exercise.

**Problem 201** Practice applying the product rule (and also the sum rule) by finding the number of ways to get from  $s$  to  $t$ .



Try it: [Coursera](#).

## 8.3 Tuples and Permutations

### 8.3.1 Tuples and Generalized Product Rule

**Stop and Think!** What is the number of 5-symbol passwords where each symbol is one of the 26 lower case Latin letters? Try it: [Coursera](#).

Let us start by showing how to generate (and count) all such passwords in Python.

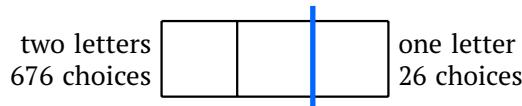
```
from itertools import product

for number, password in enumerate(
    product('abcdefghijklmnopqrstuvwxyz', repeat=5)
):
    print(number, ''.join(password))
```

0 aaaaa  
1 aaaab  
...  
11881373 zzzzz  
11881374 zzzzy  
11881375 zzzzz

This program finds the answer (11 881 376), but it already takes a few seconds to complete. Listing all passwords of length, say, 20 is completely impractical. Below, we discuss how to find the number of passwords without listing all of them. Along the way, we will generalize the product rule.

Let us apply a trick that helps in many scenarios: solve a simplified version of a problem first. Namely, what is the number of passwords of length one (that is, consisting of a single letter)? This is not difficult: the number of ways to choose the single letter is 26, hence the answer is 26. Now, what is the number of passwords of length 2? There are 26 choices for the first letter and there 26 choices for the second letter. By the product rule, the answer is  $26 \cdot 26 = 676$ . So far so good! The next case to consider is passwords of length three. A crucial observation is that the product rule helps to count such passwords too. Indeed, any password of length three consists of a prefix of length two and one additional symbol. Moreover, we already know the number of choices for each of these two parts.



Thus, the number of 3-symbol passwords is

$$676 \cdot 26 = 26 \cdot 26 \cdot 26 = 26^3 = 17576.$$

Proceeding in the same fashion, we find out that the number of 5-symbol passwords is

$$26^5 = 11\,881\,376.$$

A 5-symbol password is an example of a tuple. Formally, a *tuple* is just a sequence of objects. A *k-tuple* consists of exactly  $k$  objects. Tuples are usually written in parentheses (round brackets), both in mathematics and in Python.

```
password = ('b', 'z', 'a', 'd', 'z')
print(type(password))
```

```
<class 'tuple'>
```

To simplify notation, we will avoid commas and parentheses in some cases: for example, a 3-tuple  $(a, c, a)$  consisting of three letters will be denoted by  $aca$ . In Python, one can use the asterisk (\*) for such unpacking.

```
tuple = ('a', 'c', 'a')
print(tuple)
print(*tuple)
print(*tuple, sep=' ')
```

```
('a', 'c', 'a')
a c a
aca
```

For finite sets  $A_1, \dots, A_k$ , by their product we denote the set of all  $k$ -tuples whose first element is drawn from  $A_1$ , second element is drawn from  $A_2$ , and so on. In set theory language:

$$A_1 \times \cdots \times A_k = \{(a_1, \dots, a_k) : a_1 \in A_1, \dots, a_k \in A_k\}.$$

Then, the generalized product rule says that the size of the product of sets is equal to the product of their sizes.

**Product rule** For finite sets  $A_1, \dots, A_k$ ,

$$|A_1 \times \cdots \times A_k| = |A_1| \times \cdots \times |A_k|.$$

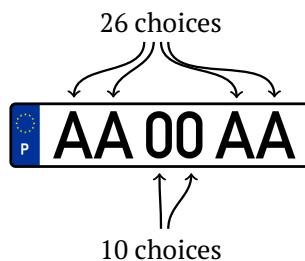
An important special case is when all  $A_1, \dots, A_k$  are the same set  $A$  of size  $n$ . The product  $A \times \cdots \times A$  is denoted by  $A^k$ :  $A^k$  is the number of tuples  $(a_1, \dots, a_k)$  where  $a_i \in A$  for every  $i = 1, \dots, k$ . The product rule then says that the number of  $k$ -tuples is equal to  $n^k$ :

$$|A^k| = |A|^k = n^k.$$

For example, the set of all 5-symbol passwords is  $L^5$ , where  $L$  is the set of all lower case Latin letters. Thus,  $|L^5| = |L|^5 = 26^5$ .

**Problem 202** Linguists are studying texts in some ancient language with 25 letters. They want to compute how often different 3-letter combinations occur in texts in this language. To do this, they first want to list all 3-letter combinations. How many combinations will they have? Try it: [Coursera](#).

Using the product rule, it is straightforward to solve Problem 188 that asks to compute the number of license plates. Recall that it consists of a block of two letters followed by a block of two digits followed by a block of two letters.



Every license plate is a tuple from a set

$$L \times L \times D \times D \times L \times L,$$

where  $D = \{0, 1, \dots, 9\}$  is the set of ten digits and  $L$  is the Latin alphabet (26 letters). Hence, the answer is  $26^2 \cdot 10^2 \cdot 26^2 = 45697600$ . This is a lot, but still not necessarily enough for the whole

country. What happens when the country runs out of license plates? A simple solution is just to change a format. Actually, the current format of the license plate in Portugal is the forth recent years. One of the previous format, for example, consisted of a block of two digits followed by a block of two letters followed by a block of two digits.

**Problem 203** Suppose a license plate format contains a block of two digits followed by a block of two letters (26 choices) followed by a block of two digits. What is the number of different license plates? Try it: [Coursera](#).

### 8.3.2 Tuples with Restrictions

In this subsection, we'll see a less straightforward application of the product rule.

**Problem 204** How many integers from 0 to 9999 have exactly one digit 7?

As usual, let us start by showing how to find the answer using Python.

```
from itertools import product

print(len([n for n in range(10000)
           if str(n).count('7') == 1]))
print(len([t for t in product(range(10), repeat=4)
           if t.count(7) == 1]))
```

```
2916
2916
```

This code counts the required number in two ways. First, we just go through all integers from 0 to 9999 and check whether the current integer has exactly one digit 7. To check this, we convert an integer to a string and then use the built-in count method to find the number of occurrences of a string “7” in it. The second approach goes through all 4-tuples of digits and, for each tuple, uses the count function again.

Why these two computations give the same answer? This happens because there is a bijection between integers from 0 to 9999 and 4-tuples of digits:

- to get a number from a tuple, just concatenate the digits of the tuple:  $(6, 7, 8, 7) \rightarrow 6787$ ;
- to get a tuple from a number, first add a few leading zeros to the number so that it contains exactly four digits and then convert the four digits to a tuple:  $23 \rightarrow 0023 \rightarrow (0, 0, 2, 3)$ .

So, starting from now, we consider 4-tuples of digits (instead of the corresponding numbers from 0 to 9999).

What is the number of 4-tuples of digits that have a single digit 7? Every such tuple falls into one of the following four categories:

- 4-tuples having 7 at the first position;
- 4-tuples having 7 at the second position;
- 4-tuples having 7 at the third position;
- 4-tuples having 7 at the fourth position.

All these four sets are disjoint (since there must be exactly one occurrence of 7), hence it is enough to count the number of tuples in each of these four sets. Let us start with the first set. Every 4-tuple in this set starts with the digit 7. For each of the second, third, and fourth digits there are nine choices: all ten digits except for 7. The product rule implies that the size of the first set is  $9^3$ . It is not difficult to see that the size of each of the remaining three sets is exactly the same. Thus, the answer to the initial problem is

$$4 \cdot 9^3 = 4 \cdot 729 = 2916.$$

Now, let's change the problem a bit.

**Problem 205** How many integers from 0 to 9999 have *at least* one digit 7?

To count such integers, one could again consider 4-tuples containing 7 at various positions. But since we now allow more than a single 7, the corresponding sets will not be disjoint: say, a tuple  $(7, 3, 7, 2)$  belongs to the first and the third set. Hence, to apply the sum rule we will need to compute the size of various intersections of our four sets.

Fortunately, there is a neat trick that allows to find the required number easily: every 4-tuple either contains at least one 7 or has no 7's at all. Hence, the required number is equal to  $10000$  (the total number of 4-tuples) minus the number of 4-tuples avoiding the digit 7. The last number is equal to  $9^4$  by the product rule: there are nine choices (all ten digits except for 7) for each of the four positions. Thus, the answer is

$$10^4 - 9^4 = 3439.$$

Let's ensure that the answer matches the one found by Python.

```
print(len([n for n in range(10000) if '7' in str(n)]))
```

3439

The complement trick that we've just used is also used frequently when counting. We state it as a rule.

**Complement rule** The number of objects satisfying a certain property is equal to the total number of objects minus the number of objects that do not satisfy the property.

In set theory language it is stated as follows: if  $A \subseteq U$ , then

$$|A| = |U| - |U \setminus A|.$$

Here,  $A$  is a set of all objects satisfying some property and  $U$  is a set of all objects (that we consider for our problem). Hence,  $U \setminus A$  is the set of all objects that do not satisfy the property. What the complement rule says is that each object either satisfies the property or does not satisfy it:

$$|A| + |U \setminus A| = |U|.$$

We are now fully equipped to solve Problem 189: count the number of strong passwords, where a strong password is a string of length eight or nine that consists of lower and upper case Latin letters as well as digits and contains at least one digit. We will count the number of strong passwords of length eight and nine separately and then apply the sum rule. Let us focus on strong passwords of length eight. The total number of strings of length eight consisting of Latin letters and digits is  $(26 \cdot 2 + 10)^8$ , by the product rule: each of eight positions may be either a Latin letter ( $26 \cdot 2$  choices for lower and upper case) or a digit (10 choices). The number of strings of length eight that do not contain a digit is  $(26 \cdot 2)^8$ . Hence, by the complement rule, the number of strings of length eight that contain at least one digit is

$$(26 \cdot 2 + 10)^8 - (26 \cdot 2)^8 = 62^8 - 52^8.$$

Similarly, the number of strong passwords of length nine is

$$(26 \cdot 2 + 10)^9 - (26 \cdot 2)^9 = 62^9 - 52^9.$$

Finally, by the sum rule, the number of strong passwords is

$$62^8 - 52^8 + 62^9 - 52^9 = 10922061039681280.$$

### 8.3.3 Permutations

**Stop and Think!** What is the number of 2-tuples consisting of letters a, b, and c, where no letter appears more than once?

Without the restriction (that no letter appears more than once) the answer would be  $3^2 = 9$ : there are two positions and three choices per position. In this toy example, it is not difficult to list all tuples with repeated letters: (a, a), (b, b), (c, c). Hence, the answer is  $9 - 3 = 6$ .

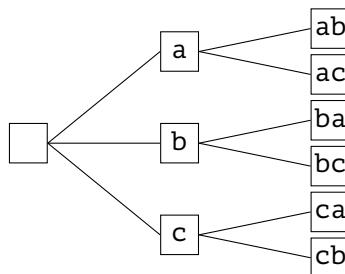
Such tuples are called *permutations* and the built-in method `permutations` can be used for enumerating them.

```
from itertools import permutations

for p in permutations('abc', 2):
    print(p)
```

```
('a', 'b')
('a', 'c')
('b', 'a')
('b', 'c')
('c', 'a')
('c', 'b')
```

Let's draw a process tree again. Here, after selecting the first letter on the first stage, we are not allowed to select the same letter on the second stage.



**Stop and Think!** In how many ways can the first, the second, and the third place be awarded to five contestants?

This is again a question about permutations! Let us denote the contestants by a, b, c, d, and e. Then, all possible first three places can be found as follows. (To make the output more compact, we omit parentheses when printing a permutation: instead of (b, a, d) we print just b a d. To achieve this, we use an asterisk \* for unpacking a permutation.)

```
from itertools import permutations

for p in permutations('abcde', 3):
    print(*p)
```

|       |       |       |       |       |
|-------|-------|-------|-------|-------|
| a b c | b a c | c a b | d a b | e a b |
| a b d | b a d | c a d | d a c | e a c |
| a b e | b a e | c a e | d a e | e a d |
| a c b | b c a | c b a | d b a | e b a |
| a c d | b c d | c b d | d b c | e b c |
| a c e | b c e | c b e | d b e | e b d |
| a d b | b d a | c d a | d c a | e c a |
| a d c | b d c | c d b | d c b | e c b |
| a d e | b d e | c d e | d c e | e c d |
| a e b | b e a | c e a | d e a | e d a |
| a e c | b e c | c e b | d e b | e d b |
| a e d | b e d | c e d | d e c | e d c |

**Stop and Think!** In how many orders can we place three books on a shelf?

Permutations again! Denote the books by a, b, and c and use the `permutations` method to list all the possibilities.

```
from itertools import permutations
for p in permutations('abc', 3):
    print(*p, sep='')
```

```
abc
acb
bac
bca
cab
cba
```

After developing an intuition on these toy examples, let us answer a general question. By a *k-permutation* over *n* symbols we mean a *k*-tuple where each element is one of the *n* symbols and no symbol is repeated.

**Problem 206** What is the number of *k*-permutations over *n* symbols?

The product rule will help us again. There are *n* choices for the first symbol. For the second symbol, there are *n* – 1 choices: we can take any of the *n* symbols except for the one taken for the first position. For the third symbol there are *n* – 2 choices: all but the two taken for the first two positions. And so on. For the *k*-th position there are *n* – (*k* – 1) choices. The product rule implies that the number of *k*-permutations is a product of *k* factors:

$$n \cdot (n - 1) \cdots (n - k + 1).$$

**Stop and Think!** Do you see why the last factor is  $(n - k + 1)$ ?

Indeed, the first factor is *n*, the second factor is  $(n - 1)$ , and so on: the *i*-th factor is  $(n - i + 1)$ , and the last (*k*-th) factor is  $(n - k + 1)$ .

This product can be expressed in compact way using factorials. Recall that for a positive integer *m*, the factorial *m*! is defined as the product of all integers from 1 to *m*:

$$m! = 1 \cdot 2 \cdots m.$$

Then,

$$n \cdot (n - 1) \cdots (n - k + 1) = \frac{n!}{(n - k)!}. \quad (8.2)$$

Indeed, if we expand  $n!$  and  $(n-k)!$  as the products of integers, all integers from 1 to  $(n-k)$  will cancel out. What is left is exactly the product of integers from  $(n-k+1)$  to  $n$ .

For example, the number of 2-permutations over letters a, b, and c is

$$\frac{3!}{(3-2)!} = \frac{3!}{1!} = \frac{1 \cdot 2 \cdot 3}{1} = 6.$$

The number of ways to award the first, second, and third place to five contestants is

$$\frac{5!}{(5-3)!} = \frac{5!}{2!} = \frac{1 \cdot 2 \cdot 3 \cdot 4 \cdot 5}{1 \cdot 2} = 60.$$

Note that the formula  $\frac{n!}{(n-k)!}$  does not make any sense for  $k > n$ . But in this case the original problem of counting  $k$ -permutations over  $n$  symbols does not make sense, too: there are just no such permutations. At the same time, for  $k = n$  the problem of counting  $k$ -permutation makes perfect sense as we've seen above with the books example. In this case, the formula becomes  $\frac{n!}{0!}$ . Though we defined the factorial function  $m!$  for positive integers  $m$  only, it can also be defined for  $m = 0$ : we set  $0! = 1$ . In this case,  $\frac{n!}{0!} = n!$  and this is exactly the number of ways to arrange  $n$  objects. For example, the number of ways to arrange three books on a shelf is  $3! = 1 \cdot 2 \cdot 3 = 6$ .

**Problem 207** How many 3-digit numbers are there that have digits 1, 2, and 3 (each of them exactly once)? Try it: [Coursera](#).

**Problem 208** A board of 15 people has to pick a chairman of the board, a vice-chairman and a secretary among themselves. All three should be different persons. How many ways do they have to do it? Try it: [Coursera](#).

**Problem 209** There are five different tables in the class and five students. Each table can be occupied by only one student. Their studying year consists of 200 days. As a small prank on their teacher students would like to sit in a new way every day, so there are no two days during their studying year such that all students are occupying the same tables. They would like to see whether this is possible. How many ways are there for them to sit in the class? Try it: [Coursera](#).

**Problem 210** How many integer numbers between 0 and 9999 are there that have one digit 1 and at the same time one digit 3? Try it: [Coursera](#).

### Summary

- The basic rules that are used almost everywhere in combinatorics:

**Sum rule:** if there are  $n$  objects of the first type and  $k$  objects of the second type, then there  $n+k$  objects in total. In set theory language: for disjoint sets  $A$  and  $B$ ,  $|A \cup B| = |A| + |B|$ .

**Product rule:** if there are  $n$  objects of the first type and  $k$  objects of the second type, then there are  $nk$  pairs of objects, where the first element of a pair is of the first type and the second element of the pair is of the second type. In set theory language: for sets  $A$  and  $B$ ,  $|A \times B| = |A| \cdot |B|$ .

**Bijection rule:** one can count the number of objects in a set by observing that the objects of this set are in one-to-one correspondence with another set. In set theory language: if there exists a bijection between sets  $A$  and  $B$ , then  $|A| = |B|$ .

**Complement rule:** instead of counting the number of objects satisfying a property, it may be easier to count the number of objects that do not satisfy the property. In set theory language: if  $A \subseteq U$ , then  $|A| = |U| - |U \setminus A|$ .

- Assume that we have a set of  $n$  symbols and we form a sequence of length  $k$  of these symbols. Such a sequence is called a *k-tuple*. The number of  $k$ -tuples (made from  $n$  symbols) is  $n^k$ . A  $k$ -tuple where no symbol is repeated is called a *k-permutation*. The number of  $k$ -permutations (from  $n$  symbols, where  $n \geq k$ ) is  $\frac{n!}{(n-k)!}$ .



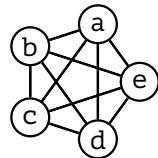
## 9. Binomial Coefficients

In how many ways one can select a team of five students out of ten students? What is the number of non-negative integers with at most five digits whose digits are decreasing? In how many ways one can get from the bottom left cell to the top right cell of a  $6 \times 6$  grid, each time going either up or to the right? And why all these three numbers are equal? We'll figure this out in this chapter!

### 9.1 Number of Games in a Tournament

**Stop and Think!** Five teams played a tournament: each team played once with each other. What was the number of games?

One way to find the answer is to draw each team, say, as a circle and to connect each pair of teams by a segment.

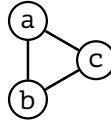


Then, the number of games is the number of segments in the picture. Whereas this approach works well for the toy example, it does not scale (try drawing, say, 50 teams and connecting every pair of them!). We need a more generic approach.

Let's see. Each team played four games (with each of the other four teams). In other words, we have five choices for the first team and four choices for the second team. By the product rule, there answer is  $5 \times 4 = 20$ .

**Stop and Think!** Do you see a flaw in this argument?

Looking at the picture, we see only 10 games (segments), not 20. Why? Let us look at even smaller example. For three teams (instead of five), our argument gives  $3 \times 2 = 6$  games. And for three teams it is particularly easy to see that this is *not* the right answer: there are three games, since during each game, one of the teams takes a rest.



Going back to the initial five teams, let us try to list all the games using Python.

```
from itertools import permutations

for p in permutations('abcde', 2):
    print(*p)
```

|   |   |
|---|---|
| a | b |
| a | c |
| a | d |
| a | e |

|   |   |
|---|---|
| b | a |
| b | c |
| b | d |
| b | e |

|   |   |
|---|---|
| c | a |
| c | b |
| c | d |
| c | e |

|   |   |
|---|---|
| d | a |
| d | b |
| d | c |
| d | e |

|   |   |
|---|---|
| e | a |
| e | b |
| e | c |
| e | d |

It seems that there are 20 games, but we already know that this is not the right answer. A close look at this list reveals the reason: each game is listed twice! For example, the game between the teams b and d is listed as bd and as db. The same happens for every other game. Thus, the number of games is  $(5 \times 4)/2 = 10$ .

An important message from our previous example: when counting, make sure that each object is counted once. If *each* object is counted  $k$  times, divide the resulting count by  $k$ .

Let us now state and prove formally the obtained result.

**Theorem 9.1.1** The number of games in a tournament with  $n$  teams (each pair of teams played each other exactly once) is

$$\frac{n(n-1)}{2}. \quad (9.1)$$

*Proof.* There are  $n$  choices for the first team in a game and  $(n-1)$  choices for the second team. Each game is counted twice: the game between teams  $i$  and  $j$  is counted as  $ij$  and as  $ji$ . Thus, the total number of games is  $n(n-1)/2$ . ■

Let us also derive the same formula using *recursion*. Denote by  $T(n)$  the number of games in a tournament with  $n$  teams. For example,  $T(1) = 0$  and  $T(2) = 1$ . We claim that for any integer  $n \geq 2$ ,

$$T(n) = (n-1) + T(n-1). \quad (9.2)$$

To prove this, we exploit the *method of distinguished element*. Fix some team — say, the first one. Then any game of the tournament either involves the fixed team or does not involve it. The first number is equal to  $(n-1)$ : the fixed team plays with each of the other  $(n-1)$  teams. The second number is equal to  $T(n-1)$ : just by definition, this is the number of games between all teams except for the fixed one.

The equation (9.2) is called a *recurrence relation*. It is straightforward to turn it into a recursive procedure.

```
def t(n):
    return (n - 1) + t(n - 1) if n > 1 else 0

print(t(5))
```

To convert the recurrence relation (9.2) into a formula (9.1), one may unwind it, that is, apply the same equation to  $T(n-1)$ ,  $T(n-2)$ , and so on:

$$\begin{aligned} T(n) &= (n-1) + T(n-1) \\ &= (n-1) + (n-2) + T(n-2) \\ &= (n-1) + (n-2) + (n-3) + T(n-3) \\ &= \dots \\ &= (n-1) + (n-2) + \dots + 2 + 1 + 0. \end{aligned}$$

This is arithmetic series! By the formula (3.3), it is equal to  $n(n-1)/2$ .

To conclude the section, we show how one can list all the games (in a tournament with five teams) in Python. To do this, one uses the combinations method.

```
from itertools import combinations

for game in combinations('abcde', 2):
    print(*game)
```

a b  
a c

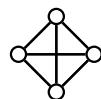
a d  
a e

b c  
b d

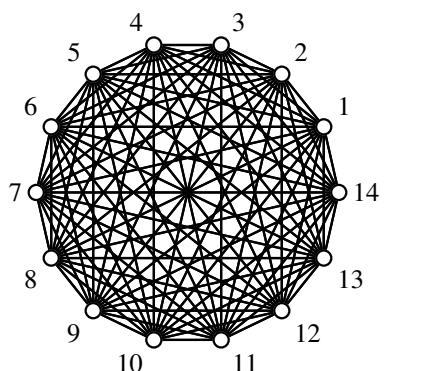
b e  
c d

c e  
d e

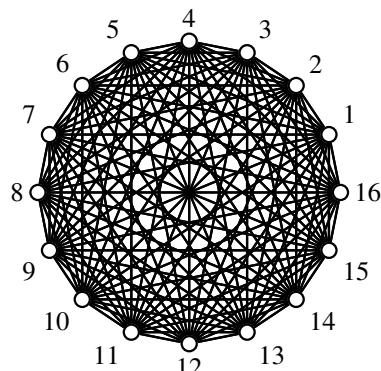
**Problem 211** What is the number of segments in the picture below? (Each segment joins two circles.) Try it: [Coursera](#).



**Problem 212** What is the number of segments in Figure 9.1(a)? (Each segment joins two circles and all pairs are connected.) Try it: [Coursera](#).



(a)



(b)

Figure 9.1: Two polygons: (a) with 14 vertices and (b) with 16 vertices.

**Problem 213** A polygonal *diagonal* is a line segment connecting two nonadjacent vertices of a polygon. (In other words, each segment connecting two vertices is either a diagonal or a side of the polygon, but not both.) Compute the number of diagonals of the polygon shown in Figure 9.1(b). Try it: [Coursera](#).

## 9.2 Combinations

**Stop and Think!** You are organizing a car journey. You have five friends, but there are only three vacant places in your car. What is the number of ways of taking three of your five friends to the journey?

In mathematical notation, what we need to find here is the number of ways of choosing 3 elements out of a set of size 5.

There are five choices for the first friend, four choices for the second friend, and three choices for the third friend. In total, there are  $5 \times 4 \times 3 = 60$  choices. But each group of three friends is counted  $3! = 6$  times: a group  $\{a, b, c\}$  is counted as  $(a, b, c)$ ,  $(a, c, b)$ ,  $(b, a, c)$ ,  $(b, c, a)$ ,  $(c, a, b)$ ,  $(c, b, a)$ . Thus, the answer is  $(5 \times 4 \times 3)/3! = 10$ .

In the argument above, we indicated a group of three friends as a *set* (in curly brackets) to emphasize that the order does not matter for us. Formally,  $\{a, b, c\}$  is a set and the order of elements is not relevant for sets:  $\{a, b, c\}$  and  $\{b, c, a\}$  are the same sets! On the other hand, parentheses (or round brackets) indicate that we are dealing with *tuples*, and for tuples the order of elements does matter:  $(a, b, c)$  and  $(b, c, a)$  are different tuples. This is reflected in Python.

```
s1 = {'a', 'b', 'c'}
s2 = {'b', 'c', 'a'}
print(type(s1))
print(s1 == s2)

t1 = ('a', 'b', 'c')
t2 = ('b', 'c', 'a')
print(type(t1))
print(t1 == t2)

<class 'set'>
True
<class 'tuple'>
False
```

As we've seen already, in some cases we count subsets, in some other cases we count tuples. Say, in the problem above, we have five friends and we want to take three of them to a journey. In this case, the order among three friends does not matter (the invitation does not specify their places in the car) and hence we are counting subsets. If, on the other hand, we are organizing a tournament among these five friends and would like to award gold, silver, and bronze medals for three winners, then the order (among three winners) does matter and we are dealing with tuples.

Going back to our journey example, this is how one can list all groups of three friends in Python.

```
from itertools import combinations

for group in combinations('abcde', 3):
    print(*group)
```

|   |   |   |
|---|---|---|
| a | b | c |
| a | b | d |

|   |   |   |
|---|---|---|
| a | b | e |
| a | c | d |

|   |   |   |
|---|---|---|
| a | c | e |
| a | d | e |

|   |   |   |
|---|---|---|
| b | c | d |
| b | c | e |

|   |   |   |
|---|---|---|
| b | d | e |
| c | d | e |

The code shows that one can use the method `combinations` to count subsets. And this is exactly how they are called in mathematics.

For a set  $S$ , its  $k$ -*combination* is a subset of  $S$  of size  $k$ . The number of  $k$ -combinations of an  $n$  element set is denoted by  $\binom{n}{k}$  and is pronounced as “ $n$  choose  $k$ ”. This number arises so frequently that you may even compute it using Google, see Figure 9.2. Of course, Python also provides a convenient way to compute it.

```
from math import comb
print(comb(5, 3))
10
```

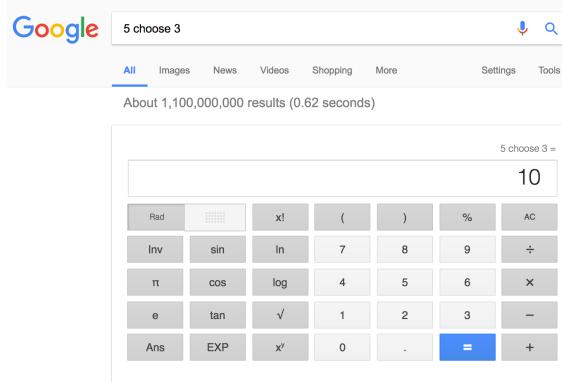


Figure 9.2: Computing  $\binom{5}{3}$  in Google.

Before deriving a formula for  $\binom{n}{k}$ , let us go through our argument for  $\binom{5}{3}$  once again. By definition,  $\binom{5}{3}$  is the number of 3-element subsets of a set of size 5. As usual, for the sake of illustration, let us denote these five elements by  $a, b, c, d, e$ . We start by listing all 3-permutations of these five letters. There are

$$\frac{5!}{(5-3)!} = 5 \cdot 4 \cdot 3 = 60$$

of them: there are five choices for the first element, four choices for the second element, and three choices for the third element (recall formula (8.2) from Section 8.3.3). Then, there are six 3-permutations corresponding to the same 3-combination: for example, for a 3-combination  $\{a, c, e\}$  the corresponding six 3-permutations are

$ace, aec, cae,cea, eac, eca.$

Hence, the number of 3-combinations is equal to the number of 3-permutations divided by  $3!$ :

$$\binom{5}{3} = \frac{5!}{(5-3)! \cdot 3!}. \quad (9.3)$$

Figure 9.3 visualizes the proof. And we are ready to generalize this proof.

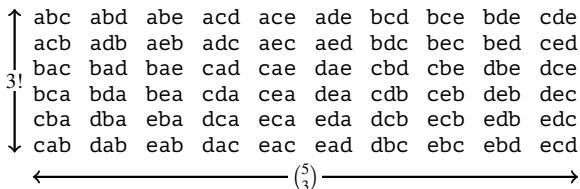


Figure 9.3: A visual proof of a formula  $\binom{5}{3} = \frac{5!}{(5-3)! \cdot 3!}$ .

**Theorem 9.2.1** The number of  $k$ -combinations of an  $n$ -element set is equal to

$$\frac{n!}{k!(n-k)!}. \quad (9.4)$$

*Proof.* There are  $n$  choices of the first element,  $(n-1)$  choices of the second element,  $\dots$ ,  $(n-k+1)$  choices of the  $k$ -th element. This gives  $n(n-1)\cdots(n-k+1) = \frac{n!}{(n-k)!}$ , but this is the number of  $k$ -permutations rather than the number of  $k$ -combinations. This way, each  $k$ -combination is counted  $k!$  times. This finally gives  $\frac{n!}{k!(n-k)!}$ . ■

The formula (9.4) assumes that  $0 \leq k \leq n$ . In particular, it works for two extreme cases:  $k=0$  and  $k=n$ . Recall that a common convention is that  $0!=1$ . By plugging this into (9.4), one gets  $\binom{n}{0}=1$  and  $\binom{n}{n}=1$ . This makes perfect sense: the empty set is the only 0-combination and the entire universe set is the only  $n$ -combination.

**Problem 214** In how many ways can one select a team of five students out of ten students?  
Try it: [Coursera](#).

**Problem 215** In how many ways can one partition ten students into two teams of size five?  
Try it: [Coursera](#).

### 9.2.1 Pascal's Triangle

The number of ways of forming a team of  $k$  students out of  $n$  students is  $\binom{n}{k}$ . Let us apply the method of distinguished element for this counting problem. To do this, fix one of the students and call her Alice. Then, there are two types of teams:

1. Teams with Alice. There are  $\binom{n-1}{k-1}$  of them: we already have Alice in a team and need to choose  $k-1$  other players from the remaining  $n-1$  students.
2. Teams without Alice. There are  $\binom{n-1}{k}$  of them: we need to select  $k$  players out of  $n-1$  students.

This way, we arrive at a useful identity known as *Pascal's rule*:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}. \quad (9.5)$$



Figure 9.4: Blaise Pascal (1623–1662). (Source: [Wikipedia](#).)

Pascal's rule can also be proved using the formula  $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ :

$$\begin{aligned}
& \binom{n-1}{k-1} + \binom{n-1}{k} \\
&= \frac{(n-1)!}{(k-1)!(n-k)!} + \frac{(n-1)!}{k!(n-1-k)!} \\
&= (n-1)! \left( \frac{1}{(k-1)!(n-k)!} + \frac{1}{k!(n-1-k)!} \right) \\
&= (n-1)! \left( \frac{k}{k!(n-k)!} + \frac{n-k}{k!(n-k)!} \right) \\
&= (n-1)! \frac{n}{k!(n-k)!} \\
&= \frac{n!}{k!(n-k)!} \\
&= \binom{n}{k}.
\end{aligned}$$

Let us “pack” the values of  $\binom{n}{k}$  into an infinite structure known as *Pascal's triangle* and shown in Figure 9.5. It turns out that Pascal's triangle has many remarkable properties. We discuss them below.

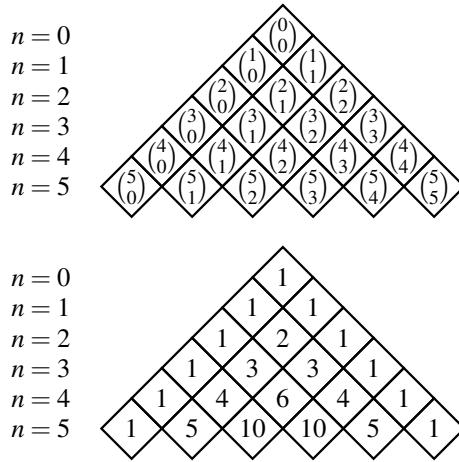


Figure 9.5: Pascal's triangle: the  $n$ -th row contains the values of  $\binom{n}{0}, \binom{n}{1}, \dots, \binom{n}{n}$ . Only the first six rows of the triangle are shown.

### Top Neighbors

Pascal's rule (9.5) has a natural interpretation in terms of Pascal's triangle: the value of each cell (that is not on the border) is equal to the sum of the values of its two top neighbors. See Figure 9.6.

### Computing the Number of Combinations

This neighboring property leads to the following natural way of computing  $\binom{n}{k}$ : start with an empty triangle, fill in all border cells by ones; then, going from top to bottom, compute the value of every cell as the sum of its two top neighbors. See Figure 9.7

The following code mimics filling in Pascal's triangle by values. We use a [dictionary ↗](#) `cells` to keep to store the values: it allows to use pairs  $(n, k)$  as indices. The `for` loop computes the values of the triangle row by row. In each row, we initialize the first and the last cells by ones. We compute the value of each other cell as the sum of the values of its two neighboring cells.

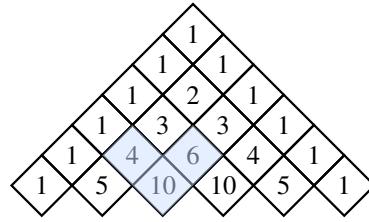


Figure 9.6: The value of each cell in Pascal's triangle is equal to the sum of the values of its two top neighbors.

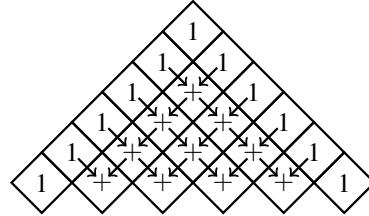


Figure 9.7: Filling in Pascal's triangle.

```
cells = dict() # cells[n, k] will keep n choose k

for n in range(8):
    cells[n, 0] = 1
    cells[n, n] = 1

    for k in range(1, n):
        cells[n, k] = cells[n - 1, k - 1] + cells[n - 1, k]

print(cells[7, 4])
```

35

Such a way of computing binomial coefficients for large values of  $n$  and  $k$  might be preferable to computing through the formula  $\frac{n!}{k!(n-k)!}$ . The reason is that the code above only uses the summation operation whereas the factorial function grows extremely fast and this can cause an integer overflow in some programming languages. (Another possibility is to use a recurrence  $\binom{n}{k} = \frac{n}{k} \binom{n-1}{k-1}$ . It requires a smaller number of arithmetic operations, but the operations themselves are multiplications and divisions rather than additions.)

### Symmetries

Pascal's triangle is symmetric with respect to the middle vertical line, see Figure 9.8.

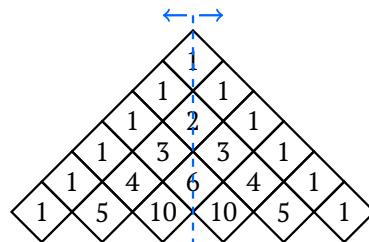


Figure 9.8: Pascal's triangle is symmetric:  $\binom{n}{k} = \binom{n}{n-k}$ .

As a formula, this property is stated as follows:

$$\binom{n}{k} = \binom{n}{n-k}. \quad (9.6)$$

It can be proved algebraically in just one line:

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

There is also a natural combinatorial proof of the identity (9.6). The number of ways of selecting a team of size  $k$  out of  $n$  students is  $\binom{n}{k}$ . The number of ways of selecting a team of size  $n-k$  out of  $n$  students is  $\binom{n}{n-k}$ . These two numbers are equal as there is a bijection between the corresponding two sets: when you select a team of  $k$  students you get a team of  $(n-k)$  remaining students (and vice versa).

The Pascal triangle is symmetric just because the initial filling (1s on the sides) and the rule to fill all other places are symmetric. If we fill the triangle on a glass board, then people on the other side of the glass will also see the correct filling.

### Row Sums

Let us compute the sum of every row in Pascal's triangle, see Figure 9.9.

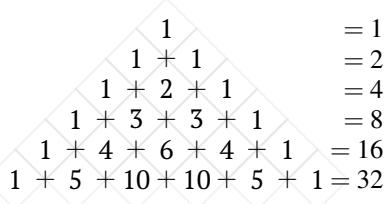


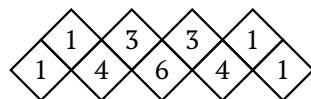
Figure 9.9: Row sums of Pascal's triangle.

**Stop and Think!** By looking at Figure 9.9, can you guess a formula for the sum of  $n$ -th row?

You have probably guessed that the answer is  $2^n$ . More formally, the sum of all the numbers in the  $n$ -th row of Pascal's triangle is equal to  $2^n$ :

$$\sum_{k=0}^n \binom{n}{k} = \binom{n}{0} + \binom{n}{1} + \dots + \binom{n}{n-1} + \binom{n}{n} = 2^n. \quad (9.7)$$

This can be proved by induction on  $n$ . The base case  $n = 0$  clearly holds. For the induction step, assume that  $n > 0$  and show that the sum of each row is twice the sum of the previous row. This follows from the neighboring property of Pascal's triangle. Instead of proving this formally, let us consider a toy example. These are the third and fourth rows:



Every non-border element of the fourth row is equal to the sum of its two top neighbors from the third row.

$$\begin{array}{ccccc} 1 & 4 & 6 & 4 & 1 \\ 1 & (1+3) & (3+3) & (3+1) & 1 \end{array}$$

By regrouping the terms, we see that every element from the third row appears exactly twice:

$$(1+1) \quad (3+3) \quad (3+3) \quad (1+1)$$

For this reason, the sum of the fourth row is twice the sum of the third row.

A combinatorial proof of the equation (9.7) proceeds as follows. The binomial coefficient  $\binom{n}{k}$  is the number of  $k$ -subsets of a set of size  $n$ . Hence, the sum of  $\binom{n}{k}$  for all  $k$  (from 0 to  $n$ ) is just the number of all subsets of an  $n$  element set. This number is equal to  $2^n$  by the product rule: each of the  $n$  elements is either included or not.

### Alternating Row Sums

Now, let us compute alternating row sums in Pascal's triangle: in every row, we take every other element with a minus sign. See Figure 9.10.

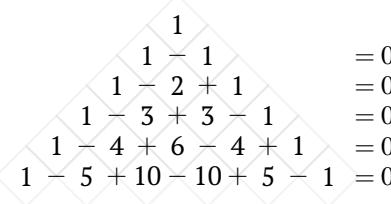


Figure 9.10: Alternating rows sums in Pascal's triangle.

As you see, we get zero for every row except for the 0-th row: for every  $n > 0$ ,

$$\sum_{k=0}^n (-1)^k \binom{n}{k} = 0. \quad (9.8)$$

For odd  $n$ , this follows immediately from the symmetry property. In general, it can be derived from the neighboring property again: when the alternating sum of the row is expressed through the elements of the previous sum, everything cancels out.

Let us prove the identity (9.8) combinatorially. By rearranging the terms, we get the following identity:

$$\binom{n}{1} + \binom{n}{3} + \dots = \binom{n}{0} + \binom{n}{2} + \dots \quad (9.9)$$

It has a natural combinatorial meaning: for  $n > 0$ , the number of odd size subsets of an  $n$ -element set is the same as the number of even size subsets. To prove that these two numbers are equal, we construct the bijection between odd size subsets and even size subsets. We do this by the method of distinguished element. Fix any element  $x$  of the underlying  $n$ -element set (can do this since  $n > 0$ ). Partition all subsets into pairs  $(A, B)$  where  $A$  and  $B$  “differ by  $x$ ” only: that is, either  $A = B \cup \{x\}$  or  $B = A \cup \{x\}$ . It remains to note that one of  $A, B$  has odd size and the other one has even size (as their sizes differ by one).

Figure 9.11 shows an example illustrating the bijection between odd size subsets and even size subsets of a set  $S = \{a, b, c, d\}$ .

Finally, since the sum of both sides of (9.9) is  $2^n$ , we conclude that

$$\binom{n}{1} + \binom{n}{3} + \dots = \binom{n}{0} + \binom{n}{2} + \dots = 2^{n-1}. \quad (9.10)$$

## 9.3 Binomial Theorem

Recall the well known identities:

$$\begin{aligned} (a+b)^2 &= a^2 + 2ab + b^2, \\ (a+b)^3 &= a^3 + 3a^2b + 3ab^2 + b^3, \\ (a+b)^4 &= a^4 + 4a^3b + 6a^2b^2 + 4ab^3 + b^4. \end{aligned}$$

| even size subsets | odd size subsets |
|-------------------|------------------|
| $\emptyset$       | $\{a\}$          |
| $\{a, b\}$        | $\{b\}$          |
| $\{a, c\}$        | $\{c\}$          |
| $\{a, d\}$        | $\{d\}$          |
| $\{b, c\}$        | $\{a, b, c\}$    |
| $\{b, d\}$        | $\{a, b, d\}$    |
| $\{c, d\}$        | $\{a, c, d\}$    |
| $\{a, b, c, d\}$  | $\{b, c, d\}$    |

Figure 9.11: A bijection between odd size subsets and even size subsets of a set  $S = \{a, b, c, d\}$ : every two paired elements differ by  $a$ .

They can be proved just by expanding the left hand side. The monomials in the right hand side are given in the order of increasing degree of the formal variable  $b$ . These coefficients are  $(1, 2, 1)$ ,  $(1, 3, 3, 1)$ , and  $(1, 4, 6, 4, 1)$ . These are the second, third, and fourth rows of the Pascal's triangle! This is not a coincidence, of course, and this can be generalized to every integer degree  $n$ .

**Theorem 9.3.1 – Binomial theorem.** For every positive integer  $n$ ,

$$(a+b)^n = \sum_{k=0}^n \binom{n}{k} a^{n-k} b^k. \quad (9.11)$$

A less formal statement of the theorem: if one expands  $(a+b)^n$  and counts the number of occurrences of the monomial  $a^{n-k}b^k$ , one gets  $\binom{n}{k}$ . And this is exactly the reason why  $\binom{n}{k}$  is called a *binomial coefficient*:  $a+b$  is a *binomial*.

*Proof.* When expanding the product

$$(a+b)(a+b) \cdots (a+b)$$

we select either  $a$  or  $b$  from each of the  $n$  terms. Hence, the monomial  $a^{n-k}b^k$  appears each time when we select  $b$  exactly  $k$  times. The number of ways to do this is  $\binom{n}{k}$ . ■

One can also prove the theorem by induction on  $n$ , using the neighboring property of Pascal's triangle. Let us illustrate this on a toy example again.

$$\begin{aligned} (a+b)^4 &= (a+b)^3(a+b) \\ &= (a^3 + 3a^2b + 3ab^2 + b^3)(a+b) \\ &= a^4 + 3a^3b + 3a^2b^2 + ab^3 + \\ &\quad + a^3b + 3a^2b^2 + 3ab^3 + b^4 \\ &= a^4 + 4a^3b + 6a^2b^2 + 4ab^3 + b^4. \end{aligned}$$

The binomial theorem allows to expand quickly not only simple binomials like  $(a+b)$ , but also arbitrary linear combinations of two variables:

$$\begin{aligned} (2a-b)^4 &= ((2a)+(-b))^4 = (2a)^4 + 4(2a)^3(-b) + 6(2a)^2(-b)^2 + 4(2a)(-b)^3 + (-b)^4 \\ &= 16a^4 - 32a^3b + 24a^2b^2 - 8ab^3 + b^4. \end{aligned}$$

As Pascal's triangle, the binomial theorem has many remarkable consequences. We list some of them below.

**Stop and Think!** What do we get from the binomial theorem when  $a = b = 1$ ?

By plugging  $a = b = 1$  into the binomial theorem, one shows that the sum of the  $n$ -th row of Pascal's triangle is equal to  $2^n$ :

$$2^n = (1+1)^n = \sum_{k=0}^n \binom{n}{k} 1^{n-k} 1^k = \sum_{k=0}^n \binom{n}{k}.$$

**Stop and Think!** Now plug  $a = 1$  and  $b = -1$  into the binomial theorem. What do you get?

By plugging  $a = 1$  and  $b = -1$ , one proves that the alternating sum of the  $n$ -th row (for  $n > 0$ ) is equal to 0:

$$0 = (1 + (-1))^n = \sum_{k=0}^n \binom{n}{k} 1^{n-k} (-1)^k = \sum_{k=0}^n (-1)^k \binom{n}{k}.$$

In both examples we get some results that we already knew. Let us try something more exotic, e.g.,  $a = 1, b = 2$ . We get the following identity:

$$3^n = \sum_{k=0}^n \binom{n}{k} 2^k = \binom{n}{0} + \binom{n}{1} 2 + \binom{n}{2} 2^2 + \cdots + \binom{n}{n} 2^n.$$

**Stop and Think!** Can you imagine a combinatorial proof of this identity, without referring to the binomial theorem?

Indeed such a proof exists. What is the number of  $n$ -tuples over a ternary alphabet  $\{a, b, c\}$ ? There are two ways to answer this question.

- By the product rule, it is equal to  $3^n$ : there are three choices for each of  $n$  positions.
- Let us count the number of  $n$ -tuples containing  $k$  occurrences of the letters  $a$  and  $b$  for each  $k = 0, 1, \dots, n$ . This number is equal to  $\binom{n}{k} 2^k$ : there are  $\binom{n}{k}$  choices for the positions of these two letters and for each of these  $k$  positions we choose either  $a$  or  $b$  (all the remaining positions are filled by the letter  $c$ ). Then we should sum up these quantities for all  $k$ ; this way, we get the right hand side of the identity.

Since we compute the same number in two ways, we should get the same answers. This gives a combinatorial proof of the identity.

If you studied calculus, you may wish to compute the derivatives of the both sides of the binomial theorem. Let us fix  $b = 1$ , so we get only one variable  $a$ :

$$(1+a)^n = \sum_{k=0}^n \binom{n}{k} a^k,$$

Taking the derivatives, we get

$$n(1+a)^{n-1} = \sum_{k=1}^n k \binom{n}{k} a^{k-1}.$$

Now we may plug  $a = 1$  (*after* taking the derivative, not *before*, as we did earlier) and get the identity

$$\sum_{k=1}^n k \binom{n}{k} = n 2^{n-1}.$$

**Stop and Think!** What is the combinatorial meaning of this identity? How can we prove it without differentiation?

We need to invent some interpretation for the product  $k \binom{n}{k}$ . We know that  $\binom{n}{k}$  is the number of ways to select a  $k$ -people team having  $n$  candidates. Multiplying by  $k$ , we get the number of ways to select a  $k$ -people team and make one of the players a captain. Indeed, after a  $k$ -team is fixed, there are  $k$  ways to select a captain. Therefore, the sum

$$\sum_{k=1}^n k \binom{n}{k}$$

is the number of ways to select a team (of any size  $k$ ) where one of the players is nominated a captain. (Note that for  $k = 0$  this is not possible, and the product  $0 \binom{n}{0}$  is 0.)

It remains to explain why this can be done in  $n2^{n-1}$  ways.

**Stop and Think!** Do you see why it is true? (Hint: select the captain first.)

Indeed, we can select the captain (of the future team of any size) in  $n$  ways. After the captain is chosen, we need to decide who else is in the team, i.e., to say “yes” or “no” to each of the  $n - 1$  remaining players. This can be done in  $2^{n-1}$  ways, so in total we have  $n2^{n-1}$ .

**Problem 216** Prove the inclusion-exclusion principle

$$\left| \bigcup_{i=1}^n A_i \right| = \sum_{\emptyset \neq I \subseteq [n]} (-1)^{|I|+1} \left| \bigcap_{i \in I} A_i \right|$$

using the properties of binomial coefficients.

Let us recall how we proved this for  $n = 3$ . We took an arbitrary  $x$  and checked how many times  $x$  was counted in the left hand and right hand sides. If  $x$  is not in the union of  $A_i$ , then it is not counted at all. So let us assume that it is an element of  $k$  sets among  $A_i$ , where  $1 \leq k \leq n$ .

The right hand side starts with  $|A_1| + \dots + |A_n|$ , and here  $x$  is counted  $k$  times. Then we subtract all  $|A_u \cap A_v|$  where  $\{u, v\}$  is a 2-element subset of  $\{1, 2, \dots, n\}$ . When  $x$  is discounted? This happens if both  $A_u$  and  $A_v$  contain  $x$ , i.e., are chosen among  $k$  sets  $A_i$  that contain  $x$ . This can be done in  $\binom{k}{2}$  ways.

At the next step we add all  $|A_u \cap A_v \cap A_w|$ , and  $x$  is counted again  $\binom{k}{3}$  ways, etc. The total multiplicity of  $x$  in the right hand side will be

$$\binom{k}{1} - \binom{k}{2} + \binom{k}{3} - \dots;$$

we replace  $k$  by  $\binom{k}{1}$  for uniformity. And this sum equals 1 since

$$\binom{k}{0} - \binom{k}{1} + \binom{k}{2} - \binom{k}{3} + \dots = 0$$

and  $\binom{k}{0} = 1$ .

If this argument looks too cryptic, don't worry; you may wish to recall the case  $n = 3$  and then try to see what happens for  $n = 4$ .

## 9.4 Practice Counting

We conclude the chapter by practicing our tools and applying what we've learned to various counting problems.

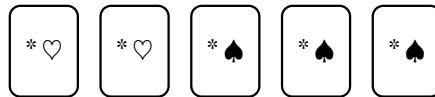
**Stop and Think!** What is the number of 5-card hands dealt off of a standard 52-card deck?



The answer is  $\binom{52}{5}$ , as we need to select five elements out of a set of size 52:

$$\binom{52}{5} = \frac{52!}{5!47!} = \frac{52 \times 51 \times 50 \times 49 \times 48}{5 \times 4 \times 3 \times 2 \times 1} = 2598960.$$

**Problem 217** What is the number of 5-card hands with two hearts and three spades?



We need to select two out of 13 hearts and three out of 13 spades. By the product rule, the answer is

$$\binom{13}{2} \binom{13}{3} = 22308.$$

**Problem 218** What is the number of four-digit integers whose digits are decreasing?

Interestingly, the answer here is again given by a binomial coefficient. The bijection rule helps to realize this: we claim that there is a bijection between four-digit integers whose digits are decreasing and subsets of size four of ten digits. Indeed, given four distinct digits, one sorts them in the decreasing order, concatenates them, and gets a four-digit number with decreasing digits. Conversely, a four-digit integer whose digits are decreasing gives a set of four digits.<sup>1</sup> Thus, the answer is

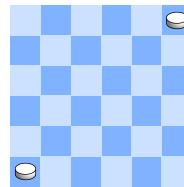
$$\binom{10}{4} = 210.$$

Let's confirm this with Python.

```
from itertools import product
print(len([d for d in product(range(10), repeat=4)
           if d[0] > d[1] > d[2] > d[3]]))
```

210

**Problem 219** What is the number of paths from the bottom left corner to the top right corner of a  $6 \times 6$  grid, each time going either up or to the right?

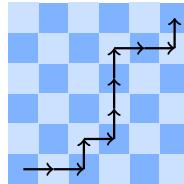


Any path to the top right corner consists of ten moves: five moves up and five moves to the right. One can specify such a path as a sequence of ten letters:

RRURUUURRU

Here, R corresponds to a move to the right whereas U is a move up. For example, the sequence above defines the following path:

<sup>1</sup>Note that a four-digit number with decreasing digits cannot have leading zeros, so we do not care about numbers less than 1000. If we spoke about four-digit numbers with *increasing* digits, we should specify whether, say, 123 = 0123 is considered as one of them.



To get such a sequence, one selects five (out of ten) positions containing the letter U (all the remaining positions contain R). Thus, the answer is  $\binom{10}{5} = 252$ .

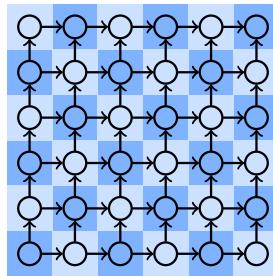
More generally, the number of paths from  $(0,0)$  to  $(k,l)$  is

$$\binom{k+l}{k}.$$

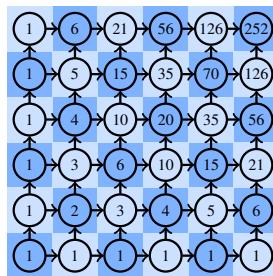
Indeed, any such path consists of  $k+l$  moves. Out of them, one needs to select  $k$  moves to the right. Note that instead of selecting  $k$  moves to the right, one could also select  $l$  moves up. This would give the answer  $\binom{k+l}{l}$ . This is the same number as  $\binom{k+l}{k}$  by equation 9.6:

$$\binom{k+l}{k} = \binom{k+l}{l}.$$

Another natural way to solve the problem is to use the recursive counting approach (recall Section 8.2). To do this, associate the following network with our grid. The network reflects the fact that from each cell one can move either up or to the right.



As we start in the bottom left corner, we fill in this corner with 1. Then, we fill all other cells using the sum rule. In the resulting picture, every cell contains the number of ways to arrive at this cell from the bottom left corner.



**Stop and Think!** Do these numbers look familiar?

All these numbers are binomial coefficients from Pascal's triangle! The reason is that this table is filled according to the same rule Pascal's triangle is filled: the value of each (non-border) cell is equal to the value of its two neighbors (recall Figure 9.7). This means that there is a natural “physical meaning” of the values in Pascal's triangle: the value of a cell is the number of ways to reach this cell from the top cell of the triangle, see Figure 9.12.

**Problem 220** What is the number of 6-card hands with three hearts and three spades? Try it: Coursera [↗](#).

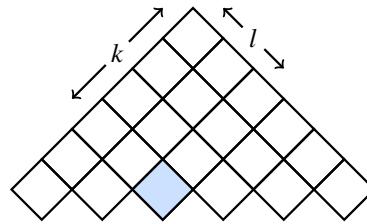


Figure 9.12: The number of ways to reach the highlighted cell is  $\binom{k+l}{k} = \binom{k+l}{l}$ .

**Problem 221** What is the number of bit strings (that is, strings consisting of 0's and 1's) of length 6 where the number of 0's is equal to the number of 1's? For example, there are two such strings of length two: 01 and 10. Try it: [Coursera](#).

**Problem 222** What is the number of sequences of six digits where the number of even digits is equal to the number of odd digits? For example, there are 50 such sequences of length two:

01, 03, 05, 07, 09, 10, 12, 14, 16, 18, ..., 90, 92, 94, 96, 98.

Try it: [Coursera](#).

**Problem 223** In how many ways one can get from the bottom left cell to top right cell of a  $9 \times 9$  grid, if each move is either two cells up or three cells to the right? Try it: [Coursera](#).

**Problem 224** In how many ways one can get from the bottom left cell to the top right cell of a  $13 \times 13$  grid, if each move is either two cells up or three cells to the right? Try it: [Coursera](#).

### Summary

- The binomial coefficient  $\binom{n}{k}$  has many natural meanings:
  - $\binom{n}{k}$  is the number of  $k$ -subsets (called combinations) of an  $n$ -element set.
  - $\binom{n}{k}$  is the coefficient of  $a^{n-k}b^k$  in  $(a+b)^n$ .
  - $\binom{n}{k}$  is the number of ways to get from the cell  $(0,0)$  to the cell  $(n-k, k)$  each time going either up or right.
- There are many interesting identities involving binomial coefficients. Below we list a few of them. Each of them can be proved either combinatorially or algebraically.
  - Pascal's rule:  $\binom{n}{k} = \binom{n}{n-k}$ .
  - Pascal's triangle row sum:  $\sum_{k=0}^n \binom{n}{k} = 2^n$ .
  - Alternating row sum:  $\sum_{k=0}^n (-1)^k \binom{n}{k} = 2^n$  (for  $n > 0$ ).
  - Ternary tuples:  $\sum_{k=0}^n 2^k \binom{n}{k} = 3^n$ .
  - Teams with a captain:  $\sum_{k=1}^n k \binom{n}{k} = n2^{n-1}$ .

# 10. Advanced Counting

In how many ways one can distribute ten candies to four kids? What is the number of credit card PINs with non-increasing digits? What is the number of non-negative integer solutions of an equation  $x_1 + x_2 + x_3 + x_4 = 10$ ? It turns out that this is essentially the same problem. The corresponding selection scheme is known as combinations with repetitions. In this chapter, we derive a formula for counting such objects using a powerful stars-and-bars method. We'll practice applying this method to many problems that have nothing in common on the first sight.

## 10.1 Review

Below, we review four standard types of selections. To visualize them, we use a toy example of selecting  $k = 2$  objects out of a set  $\{a, b, c\}$  of size  $n = 3$ . We distinguish ordered and unordered selections. In mathematics, the ordered ones are written in parentheses (that is, round brackets) whereas the unordered ones are written in braces (that is, curly brackets): we treat  $(c, a)$  and  $(a, c)$  as different, however we consider  $\{c, a\}$  and  $\{a, c\}$  as being the same. We also distinguish selections with and without repetitions:  $(c, c)$  contains a repeating element, whereas  $(c, a)$  does not.

**Tuples** are ordered selections with repetitions. Also known as *words*. The number of tuples is  $n^k$  (see Section 8.3.3).

```
from itertools import product

for t in product('abc', repeat=2):
    print(*t, sep=' ', end=' ')
```

aa ab ac ba bb bc ca cb cc

**Permutations** are ordered selections without repetitions. The number of permutations is  $\frac{n!}{(n-k)!}$  (see Section 8.3.3).

```
from itertools import permutations

for t in permutations('abc', 2):
    print(*t, sep=' ', end=' ')
```

```
ab ac ba bc ca cb
```

**Combinations** are unordered selections without repetitions. Also known as *sets*. The number of combinations is  $\frac{n!}{k!(n-k)!}$  and this quantity appears so frequently that there is a notation for it:  $\binom{n}{k}$  (see Section 9.2).

```
from itertools import combinations

for t in combinations('abc', 2):
    print(*t, sep=' ', end=' ')
```

```
ab ac bc
```

**Combinations with repetitions** are unordered selections with repetitions. Also known as *multisets*. Below, we will derive a formula for the number of combinations with repetitions.

```
from itertools import combinations_with_replacement

for t in combinations_with_replacement('abc', 2):
    print(*t, sep=' ', end=' ')
```

```
aa ab ac bb bc cc
```

## 10.2 Combinations with Repetitions

**Problem 225** We have an unlimited supply of tomatoes, bell peppers, and lettuce. We would like to make a salad out of four units among these three ingredients (we do not have to use all ingredients). How many different salads can we make? Try it: [Coursera](#).

Here, we are dealing with combinations with repetitions: the order of the ingredients does not matter and each ingredient may be taken more than once. To develop intuition, let's try to list all possible salads. To ensure that we don't miss anything, let's list all possibilities systematically. To do this, consider all salads containing  $t$  tomatoes separately.

There is one salad consisting of  $t = 4$  tomatoes (shown in red).

There are two salads containing  $t = 3$  tomatoes: the remaining unit may be either a bell pepper (shown in yellow) or a lettuce (shown in green).

There are three salads containing  $t = 2$  tomatoes.

There are four salads containing just one tomato ( $t = 1$ ).

Finally, there are five salads containing no tomatoes at all ( $t = 0$ ). Such a salad contains  $b$  bell papers where  $b$  may take five different values: 0, 1, 2, 3, or 4. Then, each of the remaining  $4 - b$  units is lettuce.

```
* * * *
* * * *
* * * *
* * * *
* * * *
```

Thus, there are  $1 + 2 + 3 + 4 + 5 = 15$  possibilities for a salad.

One can list all possibilities using Python.

```
from itertools import combinations_with_replacement
for salad in combinations_with_replacement('TBL', 4):
    print(*salad)
```

```
T T T T
T T T B
T T T L
T T B B
T T B L
T T L L
T B B B
T B B L
T B L L
T L L L
B B B B
B B B L
B B L L
B L L L
L L L L
```

Interestingly, this code also lists salads in a systematic way: in every salad, tomatoes are followed by bell peppers that are in turn followed by lettuce. This gives us a hint: to describe a salad, it suffices to specify three numbers — the number of tomatoes, bell peppers, and units of lettuce. As we agreed on the order of ingredients (tomatoes, then bell peppers, then lettuce), one can visualize the ingredients using *bars* instead of colors.

```
* | * | * *
```

Here, everything that goes before the first bar is tomatoes, everything between the first and the second bars is bell peppers, and everything after the last bar is lettuce.

\* | \* | \* \*

tomatoes    bell peppers    lettuce

This leads to the following salad.

```
* * * *
```

Moreover, every salad can be described using this *stars-and-bars* notation!

|         |             |
|---------|-------------|
| * * * * | * * * *     |
| * * * * | * * *   *   |
| * * * * | * * *     * |
| * * * * | * *   **    |
| * * * * | * *   *   * |
| * * * * | * *     **  |
| * * * * | *   * * *   |
| * * * * | *   * *   * |
| * * * * | *   *   **  |
| * * * * | *     * * * |
| * * * * | * * * *     |
| * * * * | * * *   *   |
| * * * * | * *   * *   |
| * * * * | *   * * *   |
| * * * * | * * * *     |

This example shows, in particular, that even if some ingredient is missing in a salad, this is perfectly fine for the corresponding stars-and-bars encoding. For example, there are no bell papers in the salad below. For this reason, the second bar goes immediately after the first bar.

\* \* \* \*      \* \* \* | | \*

Also, there are no tomatoes in the following salad, and this is why its code starts with a bar.

\* \* \* \*      | \* | \* \* \*

Our toy example reveals that every salad can be encoded as a sequence of four stars and two bars. Moreover, every sequence of four stars and two bars gives a salad. Thus, *there is a bijection between salads and sequences of four stars and two bars*. What is great about this is that we already know how to count the number of such sequences. Indeed, there are six positions in a sequence. Two of them contain bars, whereas all other positions contain stars.

To visualize this, let's index the positions in a stars-and-bars sequence from 0 to 5 and show the indices of two bars.

| 0       | 1           | 2   | 3 | 4 | 5 |
|---------|-------------|-----|---|---|---|
| * * * * | * * * *     | 4,5 |   |   |   |
| * * * * | * * *   *   | 3,5 |   |   |   |
| * * * * | * * *     * | 3,4 |   |   |   |
| * * * * | * *   * *   | 2,5 |   |   |   |
| * * * * | * *   *   * | 2,4 |   |   |   |
| * * * * | * *     * * | 2,3 |   |   |   |
| * * * * | *   * * *   | 1,5 |   |   |   |
| * * * * | *   * *   * | 1,4 |   |   |   |
| * * * * | *   *   * * | 1,3 |   |   |   |
| * * * * | *     * * * | 1,2 |   |   |   |
| * * * * | * * * *     | 0,5 |   |   |   |
| * * * * | * * *   *   | 0,4 |   |   |   |
| * * * * | * *   * *   | 0,3 |   |   |   |
| * * * * | *   * * *   | 0,2 |   |   |   |
| * * * * | * * * *     | 0,1 |   |   |   |

Thus, the number of such sequences is equal to the number of possibilities to select two out of six positions:

$$\binom{6}{2} = \frac{6!}{2! \cdot 4!} = \frac{5 \cdot 6}{1 \cdot 2} = 15.$$

This observation also makes it easy to enumerate all stars-and-bars sequences in Python.

```
from itertools import combinations

for bars_indices in combinations(range(6), 2):
    sequence = ['*'] * 6
    for i in bars_indices:
        sequence[i] = '|'
    print(*sequence, ' ', *bars_indices)
```

```
| | * * * * 0 1
| * | * * * * 0 2
| * * | * * * 0 3
| * * * | * * 0 4
| * * * * | * 0 5
* | | * * * * 1 2
* | * | * * * 1 3
* | * * | * * 1 4
* | * * * | * 1 5
* * | | * * * 2 3
* * | * | * * 2 4
* * | * * * | 2 5
* * * | | * * 3 4
* * * | * * | 3 5
* * * * | | * 4 5
```

Before deriving a general formula for the number of combinations with repetitions, let's practice applying the stars-and-bars method for a slightly more general salad problem where listing all possibilities by hand is already not that easy.

**Problem 226** In addition to tomatoes, bell peppers, and lettuce, we now have an unlimited supply of eggplant. How many different salads can we make out of seven units among these four ingredients?

Finding the answer using Python is straightforward.

```
from itertools import combinations_with_replacement

print(len(list(combinations_with_replacement('TBLE', 7))))
```

120

Now, let's confirm it mathematically. Since we select seven out of four ingredients (with repetitions, of course), a salad is described using seven stars and three bars:



Thus, one needs to select three positions (for bars) out of ten:

$$\binom{10}{3} = \frac{8 \cdot 9 \cdot 10}{1 \cdot 2 \cdot 3} = 120.$$

We are now ready to generalize our result. Recall that a  $k$ -combination with repetitions is a collection of  $k$  objects where any object may appear more than once.

**Theorem 10.2.1** The number of  $k$ -combinations with repetitions selected out of  $n$  objects is

$$\binom{k+n-1}{n-1}. \quad (10.1)$$

*Proof.* In short, a  $k$ -combination with repetitions is specified by  $k$  stars and  $n-1$  bars, hence the answer is  $\binom{k+n-1}{n-1}$ .

A more formal argument goes as follows. Assume that the universe (the set one selects from) is  $\{u_1, \dots, u_n\}$ . To describe a  $k$ -combination with repetitions, one specifies how many times each of  $u_1, \dots, u_n$  appears in it. This can be done as follows. First, write down  $k$  stars as placeholders for  $k$  elements in a combination. The first few stars will denote the number of times  $u_1$  occurs in a combination. Place the first bar after them. What goes next is a number of copies of  $u_2$ . Place the second bar after the corresponding number of stars. And so on. In total, we need  $n-1$  bars to split all stars into  $n$  groups. This way, we get a sequence of  $k$  stars and  $n-1$  bars. To specify such a sequence, one needs to select positions for  $n-1$  bars out of all  $k+n-1$  positions. Thus, the total number of  $k$ -combinations with repetitions is  $\binom{k+n-1}{n-1}$ .

Note that instead of selecting  $n-1$  positions for bars, one could as well select  $k$  positions for stars (all the remaining positions are filled with bars). This leads to the answer  $\binom{k+n-1}{k}$  which is equal to  $\binom{k+n-1}{n-1}$  by Pascal's rule. ■

Figure 10.1 gives a summary of the four selection schemes. In the next section, we'll practice applying the corresponding formulas in various settings.

|           | with repetitions  | without repetitions  |
|-----------|---|--|
| ordered   | <p><i>also known as:</i><br/> <math>k</math>-tuples (<math>k</math>-words)<br/> <i>number:</i> <math>n^k</math><br/> <i>example:</i><br/> <math>(a, a), (a, b), (a, c),</math><br/> <math>(b, a), (b, b), (b, c),</math><br/> <math>(c, a), (c, b), (c, c)</math></p>             | <p><i>also known as:</i><br/> <math>k</math>-permutations<br/> <i>number:</i> <math>\frac{n!}{(n-k)!}</math><br/> <i>example:</i><br/> <math>(a, b), (a, c), (b, a),</math><br/> <math>(b, c), (c, a), (c, b)</math></p> |
| unordered | <p><i>also known as:</i><br/> <math>k</math>-combinations with repetitions (<math>k</math>-multisets)<br/> <i>number:</i> <math>\binom{k+n-1}{n-1}</math><br/> <i>example:</i><br/> <math>\{a, a\}, \{a, b\}, \{a, c\},</math><br/> <math>\{b, b\}, \{b, c\}, \{c, c\}</math></p> | <p><i>also known as:</i><br/> <math>k</math>-combinations (<math>k</math>-sets)<br/> <i>number:</i> <math>\binom{n}{k}</math><br/> <i>example:</i><br/> <math>\{a, b\}, \{a, c\}, \{b, c\}</math></p>                    |

Figure 10.1: Summary of four standard selection schemes: in each case, we select  $k$  objects from a set of size  $n$ . In the examples, we show all selections of  $k = 2$  objects from a set  $\{a, b, c\}$  of size  $n = 3$ .

## 10.3 Practice Counting

In this section, we'll solve together a number of problems. We'll see that the four standard selection schemes arise naturally in various settings.

### 10.3.1 Distributing Assignments

**Problem 227** There are nine assignments and four people. How many ways are there to give a single assignment to every person (so that different people receive different assignments)? Try it: [Coursera](#) ↗

Give the first person any of nine assignments, give the second person any of the remaining eight assignments, and so on. Hence, by the product rule, the answer is

$$9 \cdot 8 \cdot 7 \cdot 6 = 3024.$$

What we count here is 4-permutations (see Section 8.3.3). The following code shows how to enumerate all possibilities in Python.

```
from itertools import permutations

for i, d in enumerate(permutations(range(9), 4)):
    print(i, d)

0 (0, 1, 2, 3)
1 (0, 1, 2, 4)
2 (0, 1, 2, 5)
3 (0, 1, 2, 6)
4 (0, 1, 2, 7)
5 (0, 1, 2, 8)
6 (0, 1, 3, 2)
7 (0, 1, 3, 4)
...
3021 (8, 7, 6, 3)
3022 (8, 7, 6, 4)
3023 (8, 7, 6, 5)
```

Here, the assignments are indexed from 0 to 8. A distribution like (0, 1, 2, 6) means that the four persons get assignments 0, 1, 2, 6, respectively.

**Problem 228** Now, we would like to distribute all nine assignments to four people (no assignment should be given to two people; every person may receive any number of assignments from 0 to 9). How many ways are there to do it? Try it: [Coursera](#).

Let's try to repeat the previous argument. As now every person may receive any number of assignments, let's give the first person a subset of assignments. The second person should receive a subset of the remaining assignments. Counting the number of such choices that depend on each other is not so easy. It turns out that another point of view helps to count the required number easily. Namely, *instead of giving assignments to people, let's assign people to assignments*: the first assignment can be given to any of four people; the second assignment can also be given to any of four people, and so on. By the product rule the number of choices is

$$4^9 = 262\,144.$$

Thus, our choices are 9-tuples (see Section 8.3.3).

The Python code below enumerates all possibilities. As in our solution above, the code assigns people to assignments. The four persons are indexed as 0, 1, 2, 3. The first shown distribution gives all nine assignments to person 0. The next one gives the first eight assignments to person 0 and gives the last assignment to person 1. And so on.

```
from itertools import product

for i, d in enumerate(product(range(4), repeat=9)):
    print(i, d)

0 (0, 0, 0, 0, 0, 0, 0, 0, 0)
1 (0, 0, 0, 0, 0, 0, 0, 0, 1)
2 (0, 0, 0, 0, 0, 0, 0, 0, 2)
3 (0, 0, 0, 0, 0, 0, 0, 0, 3)
```

```

4 (0, 0, 0, 0, 0, 0, 0, 1, 0)
5 (0, 0, 0, 0, 0, 0, 0, 1, 1)
6 (0, 0, 0, 0, 0, 0, 0, 1, 2)
7 (0, 0, 0, 0, 0, 0, 0, 1, 3)
...
262142 (3, 3, 3, 3, 3, 3, 3, 3, 2)
262143 (3, 3, 3, 3, 3, 3, 3, 3, 3)

```

### 10.3.2 Distributing Candies

**Problem 229** In how many ways one can distribute 15 candies to seven kids? Try it: [Coursera](#).

For every of  $k = 15$  candies, we select one of  $n = 7$  kids that receives this candy. Any kid may be selected any number of times (including zero number of times). Thus, we are dealing with  $k$ -combinations with repetitions selected out of  $n$  objects. The formula (10.1) gives the answer:

$$\binom{k+n-1}{n-1} = \binom{15+7-1}{7-1} = \binom{21}{6} = 54\,264.$$

As usual, keeping a formula in mind is not as important as being able to derive this formula. To support this, let's repeat the stars-and-bars argument in terms of candies and kids. Let's lay out the fifteen candies in a row:

\* \* \* \* \* \* \* \* \* \* \*

Now, let's ask the first kid to pick the first few candies from the left. Say, she decided to pick two candies. We specify it using a bar as follows.

\* \* | \* \* \* \* \* \* \* \* \*

Then, we ask the second kid to take a number of candies after the first bar. If he takes four candies, we indicate this as follows.

\* \* | \* \* \* \* | \* \* \* \* \* \*

Continuing in the same fashion, we arrive at a picture like the one below.

\* \* | \* \* \* \* | \* \* | | \* \* \* \* | \* | \*

In this example, the seven kids get the following numbers of candies: 2, 4, 2, 0, 4, 1, 2. Thus, every distribution of fifteen candies to seven kids is specified by a sequence of fifteen stars and six bars. The number of such sequences is  $\binom{21}{6}$ .

**Problem 230** Now, we would like to be more fair and to ensure that no kid is left without a candy. How many ways are there to distribute 15 candies to seven kids under this restriction? Try it: [Coursera](#).

There is a neat trick that allows to reduce this setting to the previous one. To satisfy the restriction, let's give a candy to every kid. Then, what is left is to distribute the remaining  $k = 15 - 7 = 8$  candies to  $n = 7$  kids. The number of possibilities is

$$\binom{8+7-1}{7-1} = \binom{14}{6} = 3\,003.$$

It is interesting to compare the resulting number 3003 with the answer 54264 to the previous problem: we see that the vast majority of ways to distribute the candies leave some kid without candies.

Note that the same trick allows us to solve the problem under more general restrictions. Say, we require that the second kid gets at least three candies whereas the fourth kid gets at least five

candies. To count the number of such distributions, we start by giving three candies to the second kid and five candies to the fourth kid. Then, we count the number of ways of distributing the remaining candies (without any restrictions).

Now, let's use the stars-and-bars method to solve Problem 230. Our restriction (no kid is left without a candy) translates to the stars-and-bars sequences as follows: every bar should be surrounded by two stars. In other words, a sequence cannot start or end with a bar and no bar should follow a bar. This can be visualized as follows. Initially, we have a sequence of fifteen stars.

\* \* \* \* \* \* \* \* \* \* \* \*

There are fourteen possible positions for the bars:

\* ↓ \* ↓ \* ↓ \* ↓ \* ↓ \* ↓ \* ↓ \* ↓ \* ↓ \* ↓ \*

Out of these fourteen possibilities, we need to select six positions. Thus, the answer is

$$\binom{14}{6} = 3003.$$

More generally, the two results that we've just proved can be stated as follows. We would like to count the number of integer solutions to the equation

$$x_1 + \cdots + x_n = k.$$

1. Under the restriction that  $x_i \geq 0$  for all  $i$ , the number of solutions is  $\binom{k+n-1}{n-1}$ .
  2. Under the restriction that  $x_i \geq 1$  for all  $i$ , the number of solutions is  $\binom{k-1}{n-1}$ .
- Clearly, this is the same candy distribution problem:  $x_i$  is just the number of candies that the  $i$ -th kid gets.

### 10.3.3 Counting Credit Card PINs

A typical problem in cryptography is estimating the total number of passwords: if this number is low, then a password can be cracked by a brute-force attack that systematically checks all possibilities. For example, if a password consists of three lower case Latin letters and an attacker knows this, she can find it by trying at most  $26^3 = 17576$  possibilities. On modern computers, this takes less than a second. In this section, we consider a toy problem of this flavor: counting the number of credit card PINs. A PIN (personal identification number) is a sequence of four digits. The total number of PINs is easy to find: by the product rule, it is  $10^4 = 10000$  as each of the four positions may contain any of ten digits. What we are going to count is the number of PINs with various restrictions.

**Problem 231** What is the number of PINs with sum of digits equal to 9? Try it: [Coursera](#) ↗.

To describe a PIN with sum of digits equal to nine, we need to specify how to distribute nine ones to four positions. To do this, one draws nine stars and splits them into four parts by three bars. For example, a sequence

\* \* \* | \* \* | | \* \* \* \*

gives a PIN 3204. In other words, we are dealing here with combinations with repetitions: we select  $k = 9$  objects out of  $n = 4$  possibilities. The answer is

$$\binom{9+4-1}{4-1} = \binom{12}{3} = 220.$$

Let's confirm our result.

```
from itertools import product

pins = product(range(10), repeat=4)
print(len([pin for pin in pins if sum(pin) == 9]))
```

220

**Problem 232** What is the number of PINs with sum of digits equal to 10? Try it: [Coursera ↗](#).

Well, this is easy: the answer is

$$\binom{10+4-1}{4-1} = \binom{13}{3} = 286.$$

Again, let's confirm this.

```
from itertools import product

pins = product(range(10), repeat=4)
print(len([pin for pin in pins if sum(pin) == 10]))
```

282

**Stop and Think!** Surprisingly, the answer is not 286. What is wrong?

Our answer is off by four. The reason is that we counted four illegal PINs using combinations with repetitions, namely those where all ten ones are assigned to the same position:

$$(10,0,0,0), (0,10,0,0), (0,0,10,0), (0,0,0,10).$$

These are the only illegal PINs: if not all ones are assigned to the same position, then all four positions contain a number from 0 to 9, that is, a valid digit. Thus, the answer is

$$286 - 4 = 282.$$

This coincides with the result we've found experimentally.

**Problem 233** What is the number of PINs with non-increasing digits? Try it: [Coursera ↗](#).

A PIN with non-increasing digits is uniquely determined by its multiset of digits as illustrated by the following example:

$$7441 \rightarrow \{7,4,4,1\}, \quad \{2,9,3,9\} \rightarrow 9932.$$

Thus, we need to select  $k = 4$  digits out of  $n = 10$  possibilities and repetitions are allowed. The answer is

$$\binom{4+10-1}{10-1} = \binom{13}{9} = 715.$$

```
from itertools import product

pins = product(range(10), repeat=4)
print(len([pin for pin in pins
           if all(pin[i] <= pin[i + 1] for i in range(3))]))
```

715

#### 10.3.4 Working Groups

**Problem 234** How many ways are there to split 12 students into six groups of two people? Try it: Coursera [↗](#).

There are  $\binom{12}{2}$  ways to choose the first group. From the remaining ten students, there are  $\binom{10}{2}$  ways to select the second group. Continuing in the same fashion, we arrive at the following answer:

$$\binom{12}{2} \cdot \binom{10}{2} \cdot \binom{8}{2} \cdot \binom{6}{2} \cdot \binom{4}{2} \cdot \binom{2}{2}.$$

**Stop and Think!** Are we done?

Not yet, in fact. To see why, assume for the moment that we are dealing with four (instead of twelve) students. Call them  $A, B, C, D$ . According to our argument, the answer is

$$\binom{4}{2} \cdot \binom{2}{2} = 6 \cdot 1 = 6.$$

Let's check this step by step. There are  $\binom{4}{2} = 6$  ways to select the first group. One can easily check that at this point everything is correct by enumerating all possibilities.

$$\begin{aligned} &\{A, B\} \\ &\{A, C\} \\ &\{A, D\} \\ &\{B, C\} \\ &\{B, D\} \\ &\{C, D\} \end{aligned}$$

The next step is to select the second group. In this example, there is nothing to be done: the remaining two students form the second group. This aligns well with our reasoning: on the second step, we multiply by  $\binom{2}{2} = 1$ . Thus, we can already list all the ways to split four students into two groups.

$$\begin{aligned} &\{A, B\}, \{C, D\} \\ &\{A, C\}, \{B, D\} \\ &\{A, D\}, \{B, C\} \\ &\{B, C\}, \{A, D\} \\ &\{B, D\}, \{A, C\} \\ &\{C, D\}, \{A, B\} \end{aligned}$$

**Stop and Think!** Do you see anything suspicious here?

A close look reveals that every splitting into two groups is counted twice. For example, a pairing “ $A$  with  $C$ ,  $B$  with  $D$ ” is listed twice:

$$\begin{aligned} &\{A, B\}, \{C, D\} \\ &\{A, C\}, \{B, D\} \\ &\{A, D\}, \{B, C\} \\ &\{B, C\}, \{A, D\} \\ &\{B, D\}, \{A, C\} \\ &\{C, D\}, \{A, B\} \end{aligned}$$

Indeed, for every pairing there are two permutations of the two groups as  $2! = 2$ . Thus, the number of pairing of four students is

$$\frac{\binom{4}{2} \cdot \binom{2}{2}}{2!} = \frac{6 \cdot 2}{2} = 3.$$

Going back to our original problem with twelve students, we realize that the order of groups does not matter. Hence, every pairing is counted 6! times when we select groups one by one. Thus, the answer is

$$\begin{aligned} & \frac{\binom{12}{2} \cdot \binom{10}{2} \cdot \binom{8}{2} \cdot \binom{6}{2} \cdot \binom{4}{2} \cdot \binom{2}{2}}{6!} = \\ & = \frac{\frac{12 \cdot 11}{2} \cdot \frac{10 \cdot 9}{2} \cdot \frac{8 \cdot 7}{2} \cdot \frac{6 \cdot 5}{2} \cdot \frac{4 \cdot 3}{2} \cdot \frac{2 \cdot 1}{2}}{6!} = \\ & = \frac{12!}{2^6 \cdot 6!} = \\ & = 10395. \end{aligned}$$

**Problem 235** Try arriving at a formula  $\frac{12!}{2^6 \cdot 6!}$  in a different way.

**Problem 236** Alice has seven different textbooks. She would like to lend three books to Bob for a weekend. How many ways does she have to do it? Try it: [Coursera](#).

**Problem 237** Alice has seven textbooks and Bob has five textbooks. All textbooks are different. They would like to exchange three books: Alice gives three of her books to Bob and Bob gives three of his books to Alice. How many ways do they have to do it? Try it: [Coursera](#).

**Problem 238** Five married couples are planning a barbecue. They need to pick three couples who will be responsible for bringing food. How many ways do they have to do it? Try it: [Coursera](#).

**Problem 239** Five married couples are planning a barbecue. They need to hold a meeting dedicated to the planning. The meeting should consist of five people, one from each couple. How many possible ways do they have to pick people for the meeting? Try it: [Coursera](#).

**Problem 240** Five married couples gathered for a barbecue. They need to pick three people among them who will be responsible for preparing the table. But they do not want to pick two people from the same couple for this (this would not be fair). How many ways do they have to pick three people satisfying this requirement? Try it: [Coursera](#).

**Problem 241** In a six number lottery one is trying to guess an unordered subset of six numbers among 44 numbers without repetitions. To do this, one picks six numbers out of 44. How many ways are there to do this? Try it: [Coursera](#).

**Problem 242** In a six number lottery one is trying to guess an unordered subset of six numbers among 44 numbers without repetitions. What is the number of ways to guess correctly exactly three numbers? Try it: [Coursera](#).

### Summary

- When we need to select  $k$  items out of  $n$  objects such that the order of the items does not matter and every item may be selected any number of times, we are dealing with  $k$ -combinations with repetitions.
- A  $k$ -combination with repetitions is specified by a sequence with  $k$  stars and  $n - 1$  bars.
- The number of  $k$ -combinations with repetitions selected out of  $n$  objects is  $\binom{k+n-1}{n-1}$ .
- Typical applications:
  - $k$ -multisets of a set of size  $n$ .

- Non-negative integer solutions of  $x_1 + \cdots + x_n = k$ .
- Ways to distribute  $k$  candies to  $n$  kids.



# 11. Probability

The word “probability” is used frequently in the everyday life. We often say that something is probable, or that one outcome looks more probable than another one. However, not always we can speak about probability as some number: for that, a mathematical model is needed. What is this mathematical model (probability space)? How to compute probabilities (if the model is given)? How to judge whether the model is adequate? What is conditional probability and Bayes’ theorem? How our plausible reasoning can be interpreted in terms of Bayes’ theorem? In this chapter, we cover these questions using simple examples of probability spaces and real life situations.

## 11.1 What is Probability?

### 11.1.1 The Paradox of Probability Theory

When two people toss a coin after the dinner to decide who will pay the bill, they do it because none of them (according to probability theory) can predict whether heads or tails will appear. This sounds strange: we would expect that a scientific theory predicts something. Then we can compare the predictions with the observations and reject or accept the theory. But what can we do with the theory that says that something is *unpredictable*?

Let us toss a coin several times:

H H T T H T H T T T H T T T T H T H H T T T T H H

Among 28 tosses we have 17 tails and 11 heads. Not exactly the same number of heads and tails, but not too far either. The *frequency of tails*, the ratio

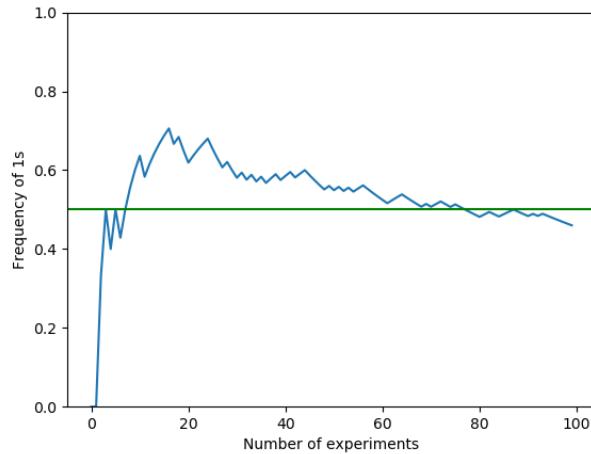
$$\frac{\text{number of tails}}{\text{number of experiments}},$$

is  $17/28 \approx 61\%$ .

**Stop and Think!** What is the frequency of heads in this sequence?

Heads and tails together make 100%, so the frequency of heads is about  $100 - 61 = 39$  percents.

Let us continue the experiments and draw the graph for the first 100 coin tosses. It shows the frequency of tails (vertical axis) after 1, 2, 3, ..., 100 experiments (horizontal axis).



The first two frequencies on the graph are zeros (since the sequence starts with  $\text{H}\text{H}$ ), and this gives a small horizontal interval on the curve. For three experiments, the frequency is  $1/3$  (one tail out of three). For four experiments, the frequency is  $1/2$  (two tails out of four), and we get a point on the green horizontal line (frequency =  $1/2$ ) that corresponds to the equal number of heads and tails.

**Stop and Think!** Looking at the graph and at the sequence of heads and tails, do you see when the frequency of tails is maximal during this series of experiments?

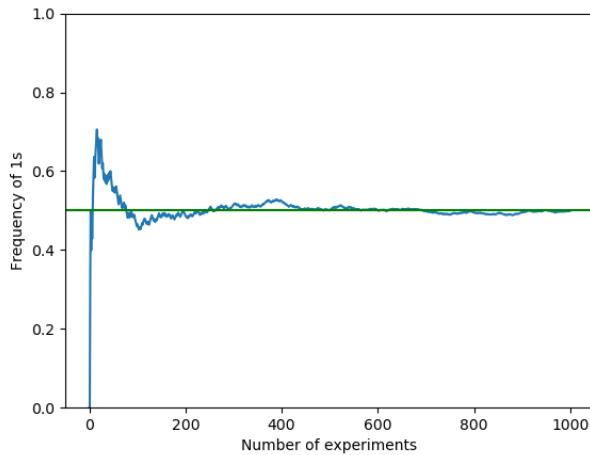
The graph shows that the maximal point is somewhere between 10 and 20 experiments. To find exactly the number of tosses that gives the maximal frequency, let us compare our sequence

$\text{H}\text{H}\text{T}\text{T}\text{H}\text{T}\text{H}\text{T}\text{T}\text{T}\text{T}\text{H}\text{T}\text{T}\text{T}\text{T}\text{T}\text{T}\text{H}\text{T}\text{H}\text{H}\text{T}\text{H}\text{H}\text{T}\text{T}\text{T}\text{T}\text{T}\text{H}\text{H}\text{H}$

with the curve. When heads appears, the frequency of tails ( $=\#\text{tails}/\#\text{experiments}$ ) decreases (the numerator remains the same, the denominator increases), and the frequency of heads increases. When tails appears, things move in the other direction: the frequency of tails increases and the frequency of heads decreases. Hence, the intervals where the frequency goes up/down correspond to groups of tails/heads in the sequence. Comparing the curve and the sequence, we see that the maximal point goes after a group of five consecutive tails. The number of experiments for this point is 17.

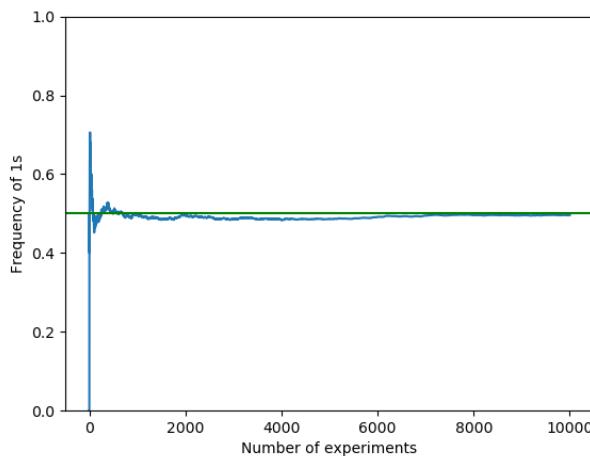
How can we describe the behavior of the frequency graph? The frequency starts below 50%, then becomes slightly higher than 50%, and then decreases. What does it mean? Does the coin becomes less “taily” with time?

To answer this question, let us look at the graph for the first 1000 experiments:

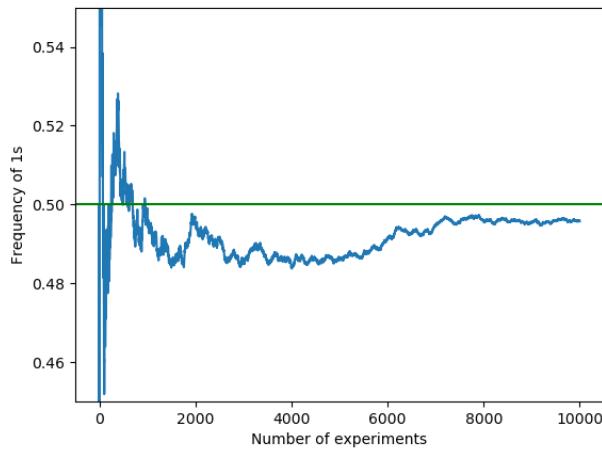


**Stop and Think!** Can you see here the first 100 experiments from the previous graph?

Yes, they are here, just compressed horizontally ten times (we have 1000 experiments instead of 100). We see that the decrease of frequency in the first 100 experiments was temporary; later the frequency started to increase again. And this is what we get for 10000 experiments:



Now, the frequency is close to 50%. To see its oscillations better, we can change the vertical scale:



Our experiment exhibits a typical behavior for random phenomena: in a long sequence of trials, the frequency of “success” of some kind (tails, in our case) becomes close to some value and oscillates slightly around this value. When people observe such a behavior, they call this value the *probability* of the event. Using this language, we can say that the probability to get a tail in our experiment seems to be close to  $1/2$ .

### 11.1.2 Random and Pseudorandom Numbers

As you might have guessed, we have not tossed a coin 10000 times in order to show a plot above. Instead, we used a special USB device that uses a random physical process (thermal noise amplification) to generate random bits (see Figure 11.1). We interpret 0 as heads and 1 as tails:

$$\textcircled{H} = 0 \quad \text{and} \quad \textcircled{T} = 1.$$



Figure 11.1: A hardware random bit generator BitBabbler.

We don’t expect you to have this device. How can you then repeat our experiments? First, you may toss a physical coin instead. Second, you may purchase this device<sup>1</sup> or some other hardware bit generator (there are many, and some are inexpensive, in \$100 range). Third, you may use sites like [random.org](http://random.org) that will give you several thousands of random bits for free. Fourth, you may download the file with bits that we have generated using our device and used for the graph above.

Perhaps, the easiest way to generate random bits is to use the `randint` functions from the `random` module in Python.

---

<sup>1</sup>This is not a paid advertisement!

```
from random import randint

print(*[randint(0, 1) for _ in range(30)], sep='')

101010010000111100100100111000
```

If you run the same code again, you get fresh random bits.

```
000010000111110110100011110001

010110110001000100011001010110
```

At the same time, the resulting bits are not “truly random”, for the following reason. Let us add a call `seed(29)` to our previous code.

```
from random import randint, seed

seed(29)
print(*[randint(0, 1) for _ in range(30)], sep='')

011011100010101110100110111011
```

Then, each time you run this code, you get exactly the same thirty bits! For this reason, the generated bits are called *pseudorandom* and an algorithm for generating them is called a *pseudorandom generator*. The function `seed` initializes the generator: try to feed it with another integer to get other thirty bits. This behavior of pseudorandom generators is useful in practice: like in our coin tossing example above, in many experiments we need to generate a bunch of randomly looking numbers. At the same time, in many cases, we would like the outcomes of the experiment to be *reproducible*. Randomness and reproducibility, in a sense, contradict each other. You would not call a coin *random*, if for each 30 tosses of this coin the results were always the same, right? For this reason, one should be careful with using pseudorandom numbers in serious applications like cryptography. Still, pseudorandom number generators are good enough for this introduction and we use them a lot in the book.

Finally, this is how to generate plots of the frequency of tails like the ones shown above.

```
import matplotlib.pyplot as plt
from random import randint, seed

# toss a coin many times
seed(19)
num_experiments = 1000
tosses = [randint(0, 1) for _ in range(num_experiments)]
frequencies = [sum(tosses[:i]) / i for i in range(1, num_experiments)]

# plot
plt.plot(frequencies)
plt.ylabel(ylabel='Frequency of 1s')
plt.xlabel(xlabel='Number of experiments')
plt.ylim(bottom=0, top=1)
plt.axhline(y=0.5, color='green')
plt.savefig('freq.png')
```

### 11.1.3 Galton Board

Another device to illustrate the behavior of random phenomena was invented by Sir Francis Galton (1822–1911). It is called “Galton board” or “bean machine” (Figure 11.2). Initially the



Figure 11.2: Galton board (source: [Wikipedia](#); on the same page there is a video showing the machine in action).

beans are at the top (after the board is rotated upside down), and then they go to the bottom through layers of pegs. Bumping into a peg, the bean goes left or right randomly; this depends on the small variations in its position, velocity, etc. Combined effect of all layers determines the box where the bean ends up.

**Stop and Think!** In the picture, the boxes at the left and at the right contain less beans compared to central boxes. Why?

To get into the leftmost box, the bean should go to the left at all layers. It is like getting tails many times in a row and happens rarely.

**Stop and Think!** Is this argument convincing? It is true that you need to go left on all ten layers (LLLLLLLLLL) to get in the leftmost box, and if you go left and right (say, LRLRLRLRLR), you end at the center box. But to get LRLRLRLRLR is as difficult as to get LLLLLLLLLL. Then, why does the center box contain much more beans?

Indeed, the bean has as many chances to follow the path LLLLLLLLLL as the path LRLRLRLRLR (assuming that at each level half of the beans goes left and half goes right). But there are many ways to get in the central box (e.g., RLRLRLRLRL, as well as many other sequences, will lead to the same central box) and only one way to get in the leftmost box.

**Problem 243** How many ways lead to the central box? In other words, how many sequences of ten letters L/R contain five L's and five R's?

To specify such a sequence, we select five out of ten positions that contains the letter *L*, hence the answer is

$$\binom{10}{5} = \frac{10!}{5! \cdot 5!} = 252$$

(recall Section 9.2).

**Stop and Think!** Look at the photo of Galton board (Figure 11.2) and find the box where the bean goes if it always goes to the left. It is not the leftmost box, but the next one. How then some beans got into the leftmost box?

The answer becomes clear if you watch the video. The beans fall rather fast and then jump again after bumping into the obstacles; they do not follow our theory (half of the beans go left and half go right at each layer). This is probably the reason why the shape of the curve differs from the theoretical predictions (as we will see below).

We see that the device does not follow exactly our model. What should we do? If our goal was to study the behavior of the real Galton board, we would need to find a better mathematical model for the real device. But our goal is to illustrate the properties of our probabilistic model using its physical implementation — so one should find instead a board that fits better our assumptions (even splitting of the beans at each splitting point). This could be done if we let the bean go through the vertical corridors before reaching the next splitting point. We discuss this below.

Before analyzing the bean machine mathematically, let us make a computer experiment. We use fresh random bits to decide where each bean should go at each layer. The machine on the photo has ten layers, so we use ten random bits to simulate the path of a single bean. The box where the bean ends up is determined by the number of 1s (“tails”) among these ten bits. This leads to 11 boxes ( $0, 1, \dots, 10$  tails). As we have mentioned, the actual machine on the photo has two more boxes (the leftmost and the rightmost boxes are accessible to naughty beans only).

```
import matplotlib.pyplot as plt
from random import randint, seed

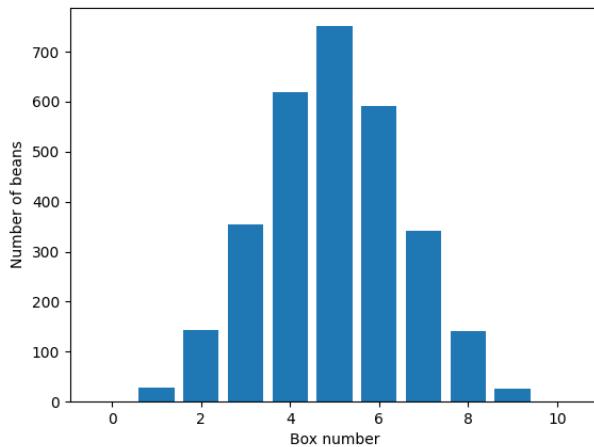
seed(29)

num_layers = 10
num_beans = 3000

def get_random_box():
    return sum([randint(0, 1) for _ in range(num_layers)])

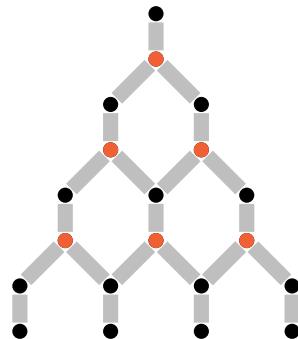
counts = [0] * (num_layers + 1)
for _ in range(num_beans):
    counts[get_random_box()] += 1

plt.bar(range(num_layers + 1), counts)
plt.ylabel(ylabel='Number of beans')
plt.xlabel(xlabel='Box number')
plt.savefig('galton.png')
```



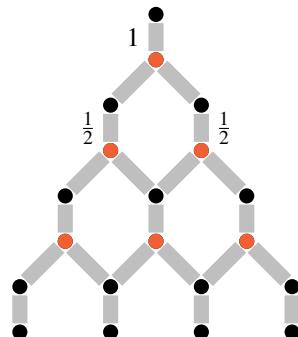
Comparing our model to the real thing, we see that the leftmost and rightmost boxes are almost empty (unlike in the real experiment; recall that we should ignore two extreme boxes that do not exist in our simulation). This can be explained by the same jumping effect in the real machine.

To predict the shape of the curve without simulation, one can analyze the fraction of beans that (on average) goes to each bin. Here is one of the possible ways to perform this analysis.



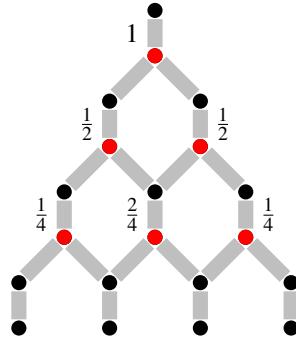
We present the board as a network of possible ways for a bean to go from top to bottom. Red points mark the places where the bean have to select whether to go left or right (and half of them goes in each direction).

Let 1 be the total flow of beans. After the first choice, we have  $1/2$  of the flow on the left and  $1/2$  of the flow on the right.



**Stop and Think!** Can you compute the flow for the next layer?

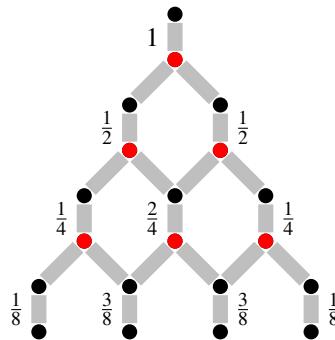
Each  $1/2$ -flow is again split into two equal parts, each of them is  $1/4$ . But then the two middle roads combine, giving  $1/4 + 1/4 = 2/4$  flow. (We keep the notation  $2/4$  instead of  $1/2$  to have fractions with the same denominators.)



**Stop and Think!** What flows do we get on the next layer?

Again, all flows are divided by 2 (denominators become twice bigger) and then some pairs are combined. We get

$$\frac{1}{8}, \quad \frac{1}{8} + \frac{2}{8} = \frac{3}{8}, \quad \frac{2}{8} + \frac{1}{8} = \frac{3}{8}, \quad \frac{1}{8}.$$



**Problem 244** Compute the flows for one more level.

**Stop and Think!** Where have you seen this process?

The numerators of the fractions form Pascal's triangle (see Section 9.2.1)! The denominators are powers of 2. The sum of each row is now 1 (as it was on the first level); recall that the sum of  $n$ -th row of Pascal's triangle is  $2^n$ .

One can explain the same fact in a different way: each bean makes  $n$  choices, and this choice can be recorded as the sequence of letters L and R. There are  $2^n$  possible sequences, and all of them should appear equally often. Hence, the number of beans in the  $k$ -th box is proportional to the number of  $n$ -letter sequences that have  $k$  letters R, and this number is  $\binom{n}{k}$ , as we have learned.

It is instructive to compare simulation results with our mathematical analysis. We computed the fraction of beans that should end in each box if the flow of beans is divided evenly at every splitting point. The program below adds red dots that show the result of this computation; the function `comb(num_layers, i)` from the `math` module computes the binomial coefficients (numbers from Pascal's triangle). One could see that the blue bars (simulation results) are quite close to them.

```

from math import comb
import matplotlib.pyplot as plt
from random import randint, seed

seed(29)

num_layers = 10
num_beans = 3000

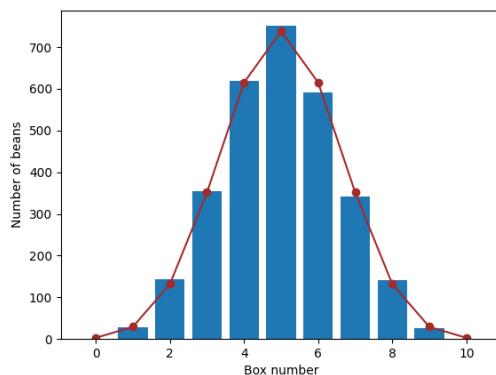
def get_random_box():
    return sum([randint(0, 1) for _ in range(num_layers)])

counts = [0] * (num_layers + 1)
for _ in range(num_beans):
    counts[get_random_box()] += 1

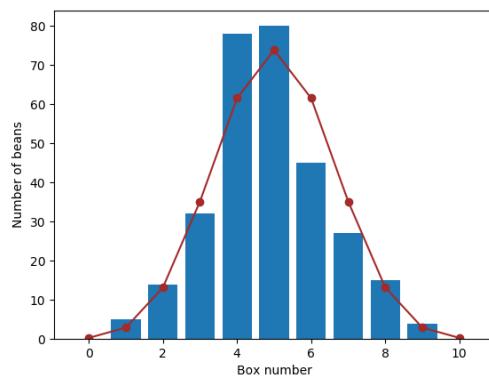
theory_counts = [
    num_beans * comb(num_layers, i) / 2 ** num_layers
    for i in range(num_layers + 1)
]

plt.bar(range(num_layers + 1), counts)
plt.plot(theory_counts, 'o-', color='brown')
plt.ylabel(ylabel='Number of beans')
plt.xlabel(xlabel='Box number')
plt.savefig('galton10-3000-2.png')

```

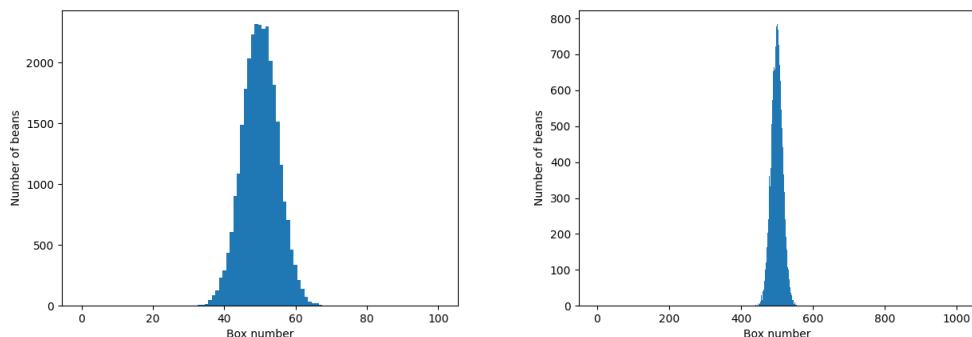


Play around with this code: change the number of layers or the number of beans. The plot below corresponds to 300 beans (instead of 3000 beans) and ten layers.



As you see, for 300 beans the blue bars are not so close (to the brown theoretical curve) as for the case of 3000 beans. This is the same effect that we have seen for coin tossing: for smaller number of experiments, random fluctuation become more visible.

This effect (the more experiments, the smaller fluctuations) can be illustrated in a different way by the same program. Recall that the destination of a bean is determined by the frequency of tails in the sequence of random bits. If this frequency is exactly  $1/2$ , then the bean ends up in the central box. As the deviation from  $1/2$  increases, the destination box becomes less central. We know that the frequency deviations become less significant as the number of coin tosses increases. Thus, we may expect that for a larger number of layers we get a more “concentrated” picture. And it is indeed the case, as the followings picture for 100 and 1000 layers show.



**Problem 245** What is the fraction of beans that end up near the center (bins 40–60 among 0–100) for the ideal Galton board with 100 layers? The same question for bins 400–600 and 1000 layers. Try it: Coursera ↗.

The following simple code finds the answer.

```
from math import comb

def middle_weight(n):
    middle_boxes = range(int(.4 * n), int(.6 * n) + 1)
    return sum(comb(n, k) for k in middle_boxes) / 2 ** n

print(middle_weight(100))
print(middle_weight(1000))
```

```
0.9647997997822951
0.999999998198318
```

Running the program, we get about 96.48% for  $n = 100$  and about 0.999999998 for  $n = 1000$ . The last number is so close to 1 that we can be practically sure that for 1000 (fair) coin tosses we will have frequency of tails between 40% and 60%.

**Stop and Think!** Do these results agree with our simulation?

Indeed, on the left picture we see some (small) fraction of beans that are either on the left of the 40-th box or on the right of the 60-th box (it should be about 3.5% according to the computations above). And on the right picture, the graph is clearly between the 40-th and 60-th boxes.

#### 11.1.4 Rolling Dice

Another toy example often used in the courses of probability theory (and sometimes for games) is a *dice*: six sides of a cube are marked 1, 2, 3, 4, 5, 6 (Figure 11.3):



We assume that in a series of experiments each number appears (on the top of the dice) with almost the same frequency, approximately  $1/6$ . We say that the probability of each outcome 1, 2, 3, 4, 5, 6 is  $1/6$ .

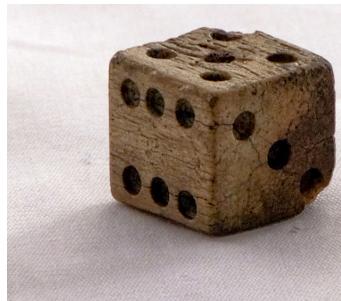


Figure 11.3: An old dice giving 5 (source: [Wikipedia](#)).

**Stop and Think!** Under this assumption, what is the fraction of experiments when 3 or a larger number appears?

There are four ways to succeed (3, 4, 5, 6) among six possible outcomes that appear (according to our assumption) equally often. Hence, the fraction of successes should be close to  $4/6 = 2/3$ .

**Problem 246** What is the probability to get (when rolling a dice) a multiple of 2? A multiple of 3? A multiple of 4?

Mathematicians say that the probability space for a dice consists of six outcomes, and each has probability  $1/6$ . (Whether a real dice follows this mathematical model, is a question for the dice manufacturers, not mathematicians.) Asking about probability of some event, we count the outcomes when this event happens, and divide the count by the total number of all outcomes. We will discuss the probability spaces below in more details.

Now, imagine two dice, say, a red one and a blue one. Rolling them, we see two numbers: a red number and a blue number (both from 1 to 6).

**Stop and Think!** How many different outcomes can we observe in this experiment?

In other terms, in how many ways may we select a red number (from 1 to 6) and a blue number (also from 1 to 6)? This is an exercise for the product rule (recall Section 8.2.2): six red options should be multiplied by six blue options, and we get 36 possible outcomes.

We can list them as a table:

|     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|
| • • | • • | • • | • • | • • | • • |
| • • | • • | • • | • • | • • | • • |
| • • | • • | • • | • • | • • | • • |
| • • | • • | • • | • • | • • | • • |
| • • | • • | • • | • • | • • | • • |
| • • | • • | • • | • • | • • | • • |

We assume again that all 36 outcomes appear equally often. Or, in mathematical terms, we consider a model where the probability space consists of 36 equiprobable outcomes having the same probability 1/36.

We say that the red and blue dice are *independent*. This means that for each red number  $r$  all the blue numbers are combined with  $r$  with equal probabilities.

If some startup starts to produce “intelligent” dice with a processor, accelerometer, and micro-motor inside, and provides a firmware that guarantees that red and blue dice always fall on the same face, these intelligent dice won’t be independent. (Still, for each of them all numbers 1, 2, 3, 4, 5, 6 may well be equiprobable!)

Now, consider an *event* in our setting (“a subset of the probability space”, mathematicians would say). For example, consider the event “the red number is bigger than the blue number”. This event may happen or not depending on the outcome. For example, for the outcome  $\begin{array}{|c|c|} \hline \text{red} & \text{blue} \\ \hline 5 & 1 \\ \hline \end{array}$  the event happens (since  $5 > 1$ ), and for  $\begin{array}{|c|c|} \hline \text{red} & \text{blue} \\ \hline 1 & 5 \\ \hline \end{array}$  the event does not happen (the numbers are equal). Sometimes the outcomes for which the event happens, are called “favorable” — but keep in mind that we may as well consider the probability of an undesirable event, and in this case we would like to avoid “favorable” outcomes.

If we roll two dice many times, what will be the fraction of the favorable outcomes among all experiments? Assuming that all outcomes are equiprobable, we should just count the fraction of favorable outcomes:

$$\text{probability} = \frac{\# \text{ favorable outcomes}}{\# \text{ all outcomes}} \quad (\text{for equiprobable outcomes})$$

and get the answer:  $p = \frac{15}{36}$  (Figure 11.4).

|     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|
| • • | • • | • • | • • | • • | • • |
| • • | • • | • • | • • | • • | • • |
| • • | • • | • • | • • | • • | • • |
| • • | • • | • • | • • | • • | • • |
| • • | • • | • • | • • | • • | • • |
| • • | • • | • • | • • | • • | • • |

Figure 11.4: There are 15 (out of 36) outcomes where the red number is larger than the blue number. Hence, the probability of the event “the red number is bigger than the blue number” is 15/36.

- Problem 247**
- Consider the event “red and blue numbers are different”. How many favorable outcomes (outcomes when this happens) do we have for this event (out of 36)? Do you see any connection with our previous computation (that gave 15 as an answer)?
  - In the same setting, consider the sum of two numbers (on two dice). What is the most

probable value of this sum?

- What is more probable while rolling two dice: to get at least one six, or to have no sixes?

Now, let us consider some variations of our setting.

**Stop and Think!** Red and blue dice are rolled sequentially, not simultaneously. We still have 36 outcomes. But are they still equiprobable?

A pure mathematician could note that the question should be asked differently (“does the model where all 36 pairs are equiprobable work well in the real life?”), and addressed to practical people. And then practical people would say that it is indeed the case: it is not important whether we roll two dice sequentially or in parallel.

A more tricky question: assume that we roll two identical dice (instead of red and blue ones). Then, we have fewer distinguishable outcomes: 23 and 32 are now the same outcome.

**Stop and Think!** How many outcomes do we have now? Are they equiprobable?

To count the outcomes, we may note that we had 36 pairs. Six of them were “doubles” where red and blue numbers are the same, and  $30 = 36 - 6$  have two different numbers. Now, these 30 numbers are combined in pairs, hence we get  $30/2 = 15$  unordered pairs, and  $6 + 15 = 21$  outcomes in total.

About equiprobability: pure mathematicians would say that it is our choice whether to consider them as equiprobable or not. But more practical people will soon find that the colors of dice do not influence the results. Since each of the pairs and appeared with probability  $1/36$ , now the combined unordered pair that consists of 2 and 3 will have twice bigger probability  $1/18$  (while each “double” outcome will have still probability  $1/36$ ).

### 11.1.5 Repeated Coin Tossing

**Stop and Think!** Toss a coin  $n$  times. What are the possible outcomes? How many of them are there?

We have already seen this example when discussing the Galton board. Interpreting as 0 and as 1, the outcomes are  $n$ -bit sequences, and there are  $2^n$  of them. We assume that all outcomes are equiprobable, and this model fits well the experimental data.

This equiprobability is not so obvious. Toss a coin ten times. Which of the following two outcomes is more probable?

Different people may have different intuition here, but the experiments show that both events happen with the same probability,  $2^{-10}$ .

**Problem 248** What is the probability that in a sequence of  $n$  coin tosses the results of the first toss and the last toss coincide?

The answer is  $1/2$ . Indeed, all  $2^n$  outcomes are divided into four groups:

Here, “...” stands for arbitrary sequence of  $(n-2)$  heads or tails, hence each group contains  $2^{n-2}$  outcomes. The favorable outcomes are in the first and in the last group, thus we have  $2 \cdot 2^{n-2} = 2^{n-1}$  of them, and the probability is  $2^{n-1}/2^n = 1/2$ .

**Problem 249** What is the probability (for  $n$  coin tossing) of the event “the number of heads is even”?

This is a more tricky problem, but we have already seen it! We need to count the number of sequences of length  $n$  consisting of and that have an even number of .

This number is equal

to

$$\binom{n}{0} + \binom{n}{2} + \binom{n}{4} + \dots$$

By (9.10), it is equal to  $2^{n-1}$ , and hence the probability is equal to  $1/2$ .

**Problem 250** Toss a coin 21 times. Consider the event: “among the first 10 tosses, there are at least as many tails as among the last 11 tosses”. What is the probability of this event?

If you do not see a better way, you may just write a program to count all the favorable outcomes (out of  $2^{21}$ , still a feasible task for modern computers). However, this problem has a nice solution that does not require any computations and works for  $n$  and  $n+1$  instead of 10 and 11. Do you see it? (A hint: consider an event “the number of *tails* among the first  $n$  tosses is at least the number of *heads* in the last  $n+1$  tosses”.)

**Stop and Think!** Professor says: “ $n$  people form a queue in a random order”. What kind of probability space does she have in mind?

As we know, there are  $n!$  permutations of  $n$  objects, and “queue” is just another name for a permutation: we have to choose the first person in the queue (out of  $n$ ), then the second one (out of  $n-1$  remaining), etc., and get  $n(n-1)(n-2) \cdots 1 = n!$  possibilities (recall Section 8.3.3). Usually, when considering random permutations, people consider all  $n!$  permutations as equiprobable.

**Problem 251** Six people, including Alice, Bob, and Charlie, form a queue in a random order.

1. What is the probability of the event “Alice is the first in the queue”?
2. What is the probability of the event “Alice precedes Bob in the queue”? (This means that Alice is closer to the start of the queue than Bob; there may be other people between them, or before/after them)
3. What is the probability of the event “Bob is between Alice and Charlie in the queue”? (Alice may be before or after Charlie, and Alice and Charlie may not be next to Bob.)
4. What is the probability of the event “Alice and Bob are neighbors in the queue”? (This means that Alice stays right before Bob or just after Bob).

Try it: [Coursera](#) ↗.

Again, if you don’t see better approaches, it is possible to write a program that lists all  $6! = 720$  permutations and counts the favorable outcomes. But there are easier ways to solve the problem using symmetries. For example, the probability for Alice to be the first in the queue is the same as for any other participant. In other words, all possible queues are divided into six groups (Alice is first, Bob is first, Charlie is first, and so on for all six people). The groups are of the same size (since the names do not really matter), so each of them consists of  $120 = 720/6$  outcomes.

**Stop and Think!** Can you use a similar idea to answer questions 2 and 3?

For question 2, we may note that all the outcomes (=queues) are divided into two classes: Alice can precede Bob, or Bob can precede Alice. (This cannot happen at the same time as one of them should be closer to the starting point.) Again, these two groups are of the same size, since the names do not matter, so each of them contains  $6!/2$  outcomes and the probability is  $1/2$ .

A similar argument, though a bit more complicated, works for question 3. If three people (Alice, Bob, and Charlie) are in the queue (of six people), there is exactly one of them who is between two others. This can be Alice, Bo,b or Charlie, so we get three groups that are of the same size (for symmetry reasons: names do not matter). Thus, each group has probability  $1/3$ .

Now let us consider the question 4. We have to count all queues (permutations) where Alice and Bob are neighbors. Let us combine them temporarily in one group. Then there are five “persons” in the queue (this combined group and four others), so there are  $5!$  possibilities. Each of these possibilities corresponds to two queues in the original sense: Alice is before Bob or after Bob in the two-people group. Thus, we get  $2 \cdot 5!$  queues where Alice and Bob are neighbors, and the probability of this event is  $2 \cdot 5!/6! = 2 \cdot 5!/6 \cdot 5! = 2/6 = 1/3$ .

Finally, this is how one could compute the answers in Python.

```
from itertools import permutations

queues = list(permutations('ABCDEF'))

for condition in (
    lambda q: q[0] == 'A',
    lambda q: q.index('A') < q.index('B'),
    lambda q: (q.index('A') - q.index('B')) *
               (q.index('B') - q.index('C')) < 0,
    lambda q: abs(q.index('A') - q.index('B')) == 1
):
    favorable = [q for q in queues if condition(q)]
print(len(favorable) / len(queues))

0.1666666666666666
0.5
0.6666666666666666
0.3333333333333333
```

### 11.1.6 Probability Theory is the Same as Counting?

Looking at our examples, one may think that probability theory is just counting in disguise: if all outcomes are equiprobable, we need just to count favorable ones using the tools we discussed earlier.

This is true, but it is not the whole truth. First, the intuition of chances, probabilities, and independence may be useful in purely combinatorial (counting) problems. (For example, recall the symmetry arguments in Problem 251.) Second, we may consider probability spaces where outcomes are not equiprobable (and we will do it in the next sections). Also, one could consider probability spaces that consist of infinitely many outcomes. One can speak about “random point in a square” where individual outcomes are points, and define probabilities in such an infinite setting – but this is outside the scope of our course.

### 11.1.7 Natural Sciences and Mathematics

Sometimes people say: “Heads and tails are equiprobable according to mathematical probability theory”. This is a simplification. Mathematics does not deal directly with the real world, and if someone made a coin that has heads on both sides (or a biased coin where one side is heavier and more likely to appear than the other one), she would not disprove the probability theory. For mathematicians, the equiprobability of heads and tails is an *assumption*, not a *claim* – just a model that they consider. Whether this model describes The Real World, is another question (and not a mathematical one).

**Stop and Think!** Alice and Bob make an experiment: they toss two coins simultaneously. This experiment is repeated, and they want to know how often one should expect one head and one tail (and not two heads, and not two tails). Alice believes that it will happen with probability  $1/2$ . Indeed, she says, there are four cases:

1. first = head, second = head;
2. first = head, second = tail;
3. first = tail; second = head;
4. first = tail; second = tail.

We have one head and one tail in two of four cases, hence the probability (says Alice) is  $2/4 = 1/2$ .

Bob disagrees. He notes that coins are identical, and we cannot distinguish between the second and third cases. Hence, essentially, we have three cases, not four:

1. two heads;

2. one head and one tail;
3. two tails.

So, Bob concludes that the probability is  $1/3$ .  
Who of the two is mathematically correct?

In fact, the question is ill-posed. From the pure mathematician's point of view, there are just two different models. Which of these two models describes the real world better is a separate question. You may notice that real coins from your pocket behave more like Alice thinks. If you don't believe us, toss two identical coins thousand times (or ten thousand times, it is still possible to do in one day, if you wake up early enough) and see what happens.

Quantum physicists know that for microscopic objects Bob may be right: some of them are "really indistinguishable" (physicists call this case the [Bose–Einstein statistics](#) ↗, and three outcomes listed by Bob can indeed be equiprobable.

## 11.2 Probability: Do's and Don'ts

Now you have an idea what people may have in mind when speaking about probabilities of events, at least in some cases. However, we need to consider more examples of what the probability theory can do (and what it cannot, too).

### 11.2.1 Not Equiprobable Outcomes

Equiprobable outcomes and perfectly symmetric objects do not appear often in the real life. If the probability theory dealt only with toy examples about coins and dice, it could hardly be so useful. And even coins and dice are not perfectly symmetric. If one side of the coin is heavier, heads could appear more often than tails. And this is not a problem, if the frequency of each side (head/tail) comes close to some limit value when the number of experiments grows. For example, it may happen that for some real coin the probability of having a tail is close to 50.07% for a long series of experiments.

For a coin we have two numbers (probabilities of heads and tails) that sum to 1. For a dice we have six outcomes, and we may assume that for large number of experiments their frequencies come close to six numbers  $p_1, \dots, p_6$ , whose sum is still 100%. For a perfect dice all  $p_i$  are equal to  $1/6$ .

**Stop and Think!** Is it possible that the probability to get  $\square$  on a non-symmetric dice is smaller than  $1/6$ , and still the outcome  $\square$  is the most probable one?

Of course, not: if  $\square$  is the most probable outcome, then  $p_2, \dots, p_6 \leq p_1$ , where  $p_i$ 's are the probabilities of the six outcomes. And if  $p_1 < 1/6$ , then all  $p_i$  are strictly less than  $1/6$  and their sum is less than 1 (whereas the sum of frequencies should be equal to 1).

We are ready to give a formula definition.

**Definition 11.2.1 – Probability space.** A *probability space* is a finite set  $U$  of *outcomes* and a function  $\Pr: U \rightarrow [0, 1]$  such that  $\sum_{u \in U} \Pr[u] = 1$ . For an outcome  $u \in U$ , the value  $\Pr[u]$  is called the *probability* of  $u$ .

For example, the probability space of a dice is the following:

|          |           |           |           |           |           |           |
|----------|-----------|-----------|-----------|-----------|-----------|-----------|
| $u$      | $\square$ | $\square$ | $\square$ | $\square$ | $\square$ | $\square$ |
| $\Pr[u]$ | $1/6$     | $1/6$     | $1/6$     | $1/6$     | $1/6$     | $1/6$     |

**Stop and Think!** If a probability space consists of  $k$  equiprobable outcomes (having equal probabilities), what is the probability of each outcome?

Clearly,  $1/k$ : if  $k$  equal numbers have sum 1, each of them is  $1/k$ .

**Stop and Think!** Consider the following (not necessarily symmetric) dice:

|          |       |       |       |       |       |       |
|----------|-------|-------|-------|-------|-------|-------|
| $u$      | •     | □     | ▢     | ▢▢    | ▢▢▢   | ▢▢▢▢  |
| $\Pr[u]$ | $p_1$ | $p_2$ | $p_3$ | $p_4$ | $p_5$ | $p_6$ |

What will be the fraction of experiments (in a long series) when an *even* number appears?

There are three favorable outcomes: □, ▢, and ▢▢. The fraction of tosses that give □ is  $p_2$ , for ▢ it is  $p_4$ , for ▢▢ it is  $p_6$ . Thus, the fraction of all favorable outcomes is  $p_2 + p_4 + p_6$ .

More formally, for  $n$  coin tosses, there will be about  $p_2n$  tosses with an outcome □, about  $p_4n$  tosses with an outcome ▢, and about  $p_6n$  tosses with an outcome ▢▢. These groups of experiments are disjoint (each experiment has only one outcome), so the sum rule says that there is approximately  $p_2n + p_4n + p_6n$  experiments with even outcome, and their fraction is about

$$\frac{p_2n + p_4n + p_6n}{n} = p_2 + p_4 + p_6.$$

In plain English: the frequency of the event is the sum of the frequencies of individual outcomes that form this event. Mathematicians translate this observation into a definition as follows.

**Definition 11.2.2 – Probability of an event.** Let  $(U, \Pr)$  be a probability space. An *event*  $E$  is a subset of  $U$ . Then, the *probability* of  $E$ , denoted  $\Pr[E]$ , is the sum of probabilities of all outcomes in  $E$ :

$$\Pr[E] = \sum_{u \in E} \Pr[u].$$

**Stop and Think!** For the dice we considered, what is the probability to get a number greater than 4?

The corresponding event is  $E = \{\square, \square\square\}$ . Thus, by definition,

$$\Pr[E] = \Pr[\square] + \Pr[\square\square] = p_5 + p_6.$$

**Stop and Think!** Can an event have probability greater than 1? Can an event have probability equal to 1? Can an event have probability equal to zero?

The first answer is negative: if the sum of all probabilities (for all outcomes) is 1, and then we restrict it and consider the sum of probabilities of *some* outcomes, we can only make the probability smaller, not bigger. It is tempting to say that for the same reason the probability of an event is always strictly between 0 and 1, but there are two exceptional cases. First, we may consider an event that contains all outcomes (always happens), then its probability is 1 (the sum of all outcomes' probabilities). Or, symmetrically, we may consider an empty event that never happens; its probability is 0. But this is not all: it is not forbidden to have outcomes with probability 0 ("legally possible, but practically impossible"). Then these "ghost" outcomes do not change the probability. For example, an event that contains only ghost outcomes has probability 0, and its negation (complement) has probability 1.

In the following problems, we try to make a fair bet, even if the dice we are given is not symmetric.

**Problem 252** 1. You are at a restaurant with your friend. You want to decide who pays, but you only have the following asymmetric dice.

|          |        |       |       |        |        |        |
|----------|--------|-------|-------|--------|--------|--------|
| $u$      | •      | □     | ▢     | ▢▢     | ▢▢▢    | ▢▢▢▢   |
| $\Pr[u]$ | $1/10$ | $2/5$ | $1/5$ | $1/10$ | $1/10$ | $1/10$ |

You would like to roll this dice once to decide who pays. How can you ensure that you and your friend have equal chances?

2. Assume now that you are at a restaurant with two of your friends. How can you ensure equal chances for three of you by rolling the same dice once?
3. Now you want to design a (six-face) dice that can be used for two, three, or four people (rolling the dice once, we can provide equal chances for all eaters). Is this possible to assign probabilities  $p_1, \dots, p_6$  to six outcomes that ensure this property?
4. Finally, is it possible to design such a dice if the number of eaters can be as large as five? Try it: [Coursera](#).

In this problem, we have to group the outcomes in two (or three, or four, or five) groups in such a way that the probability of each group (the sum of probabilities in the group, see our definition) will be  $1/2$  (or  $1/3$ , or  $1/4$ , or  $1/5$ ). For two people, the sum in each group should be  $1/2$ .

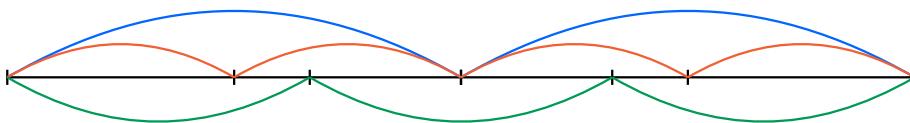
**Stop and Think!** Can you combine the outcomes to get such a sum?

For three people, you have the same numbers, but have to combine them in groups with sum  $1/3$ .

**Stop and Think!** Is it possible?

Feel free to use Python to list all subsets (of six outcomes) whose probabilities are equal to  $1/3$ .

The third question is somewhat more difficult: your goal is to *design* a dice. In this case, you cannot enumerate all possibilities using a computer. Instead, use the following picture as a hint.



For the last question, you may want to first focus on the case of five eaters. What are our options if we want to split six outcomes into five groups?

**Problem 253** Mathematician and his friend eat in the restaurant. After the dinner, the mathematician says: “Look, I have only one coin, and it is old and obviously biased. So we cannot use it to decide fairly who should pay. Indeed, … [15-minute lecture on probability theory skipped]. So, let me pay for you, darling!” The friend laughs and answers: “My dear, your theoretical analysis is perfectly correct, but does not take into account that we may toss the coin several times. We can just [...]. However, being a theoretician, you don’t have enough money, so I am paying anyway”. What was the friend’s suggestion for a fair bet? In other words, how can you use an unfair coin (that is,  $\Pr[\text{H}] \neq \Pr[\text{T}]$ ) to make a fair bet?

### 11.2.2 More About Finite Spaces

We say that two events are *mutually exclusive*, if they cannot happen simultaneously, i.e., there are no outcomes that are favorable for both events. For example, for a dice, the events “even digit” and “odd digit” are mutually exclusive, because no digit can be even and odd at the same time. In our case, these events are not only mutually exclusive, but also cover the entire probability space: one of them always happens (if you are not even, you are odd). But this is not required: the events “at least 4” and “at most 2” are also mutually exclusive (you cannot get a number that is at least 4 and at most 2) at the same time. But it is possible that both events did not happen.

**Stop and Think!** For what outcomes is this possible?

There is only one outcome avoiding both events, namely,  $\square$ .

In mathematical terms, mutual exclusivity is defined as follows.

**Definition 11.2.3 – Mutually exclusive events.** Two events  $A, B \subseteq U$  are called *mutually exclusive* if they are disjoint:  $A \cap B = \emptyset$ .

Recall (see Section 8.1.4) that it is particularly easy to compute the size of the union of two

disjoint sets  $A, B$ :

$$|A \cup B| = |A| + |B|.$$

The following formula represents a weighted version of this identity. For two mutually disjoint events  $A, B \subseteq U$  in a probability space  $(U, \Pr)$ ,

$$\Pr[A \text{ or } B] = \Pr[A] + \Pr[B].$$

Here, “ $A$  or  $B$ ” is an event that happens if at least one of the events  $A$  or  $B$  happens (we know that both of them cannot happen simultaneously). Mathematically speaking, it is just the union of the sets  $A$  and  $B$ , and may be denoted as  $A \cup B$ . The probability of the event “ $A$  or  $B$ ” is the sum of the probabilities of all its outcomes. This sum has two types of summands: outcomes from  $A$  and outcomes from  $B$ . We can regroup the terms: first compute the sum for  $A$ , and then for  $B$ , and then add the results. This is exactly what our claim says.

**Stop and Think!** Consider two events:

$$\begin{aligned} A &= \text{“tossing a coin gives T”}, \\ B &= \text{“rolling a dice gives a number smaller than 5”}. \end{aligned}$$

They are clearly disjoint: a dice is not a coin and tail is not a number. Their probabilities are  $\Pr[A] = 1/2$  and  $\Pr[B] = 2/3$ , so  $\Pr[A \text{ or } B] = 1/2 + 2/3 = 7/6$ . How is it possible that the probability becomes greater than 1?

Mathematicians would say that these two events are defined in different probability spaces (two sides for the coin and six faces for the dice), and the notion of disjoint events makes sense for the events on the same space only. Thus, we cannot use the results about disjoint events. More common-sense answer recalls the idea of probability as a frequency in the long series of experiments. We know that in a long series of coin tossing, we get (approximately) half tails, and in a long series of dice rolling, we get (approximately) two thirds of digits not exceeding 4. But those are *different* series of experiments, and it does not make sense to add these fractions ( $1/2$  and  $2/3$ ) as it were fractions in the same series.

We have already noted that mutually exclusive events  $A$  and  $B$  may or may not cover the entire probability space  $U$ . In the first case, every outcome belongs either to  $A$  or to  $B$ . In other words, an outcome is favorable for  $B$  if and only if it is *not* favorable for  $A$ . People say in this case that “ $B$  is a negation of  $A$ ”, or “ $B = \text{not } A$ ”. Sometimes, a negation of  $A$  is also denoted as follows:  $U \setminus A$ ,  $\neg A$ ,  $\overline{A}$ .

**Stop and Think!** Assume that  $B$  is a negation of  $A$ . Does it mean that  $A$  is a negation of  $B$ ?

Yes, according to our definition: both statements mean that no outcome belongs to both  $A$  and  $B$  and no outcome is outside both  $A$  and  $B$ .

**Stop and Think!** Assume that  $B$  is a negation of  $A$ , and  $C$  is a negation of  $B$ . What can we say about  $A$  and  $C$ ?

In this case, they are the same event, of course.

Clearly, if  $B$  is a negation of  $A$ , then  $\Pr[B] = 1 - \Pr[A]$ . In other words,

$$\Pr[\text{not } A] = 1 - \Pr[A].$$

To prove this, one can refer formally to our previous result: if  $B$  is a negation of  $A$ , then  $A$  and  $B$  are mutually exclusive, and the event “ $A$  or  $B$ ” covers the entire space (happens always) and therefore has probability 1. Hence,

$$1 = \Pr[A \text{ or } B] = \Pr[A] + \Pr[B] = 1,$$

and thus,  $\Pr[B] = 1 - \Pr[A]$ .

A more intuitive explanation: if in a long series, an event  $A$  happens, say, in 43% of the cases, then it *does not happen* in the remaining 57% of the cases. Those are exactly the cases when  $B = \text{not } A$  happens, and  $0.57 = 1 - 0.43$ .

**Stop and Think!** If events  $A$  and  $B$  are mutually exclusive, we know that  $\Pr[A \text{ or } B] = \Pr[A] + \Pr[B]$ . Now imagine that  $A$  and  $B$  are *not* mutually exclusive. What can be said about  $\Pr[A \text{ or } B]$ ? Can it be smaller than  $\Pr[A] + \Pr[B]$ ? Can it be greater than  $\Pr[A] + \Pr[B]$ ?

Let us consider an example. We roll a dice.  $A$  happens if an even number (i.e.,  $\square$ ,  $\square$ , or  $\square$ ) appears,  $B$  happens when a multiple of 3 (i.e.,  $\square$  or  $\square$ ) appears. Then,  $A$  and  $B$  are not mutually exclusive:  $\square$  belongs to both events. The probability of  $A$  is  $3/6 = 1/2$  (three out of six), the probability of  $B$  is  $2/6 = 1/3$  (two out of six). The probability of “ $A$  or  $B$ ” is  $4/6 = 2/3$ : favorable outcomes are  $\square$ ,  $\square$ ,  $\square$ , or  $\square$  (four out of six). And this is smaller than the sum  $\Pr[A] + \Pr[B] = 1/2 + 1/3 = 5/6$ . In other words: we have three outcomes for  $A$  and two for  $B$ , and one outcome is common, so instead of  $3 + 2 = 5$  we get only four favorable outcomes for “ $A$  or  $B$ ” out of 6.

The same happens in general. We add up all the probabilities for  $A$ -outcomes, and get  $\Pr[A]$ . Then, we add up all the probabilities of  $B$ -outcomes and get  $\Pr[B]$ . If we add these two sums, we get the sum of probabilities for all outcomes that are favorable for  $A$  or  $B$ , but some outcomes (those that are both in  $A$  and  $B$ ) are counted twice. So, if they exist (and have non-zero probabilities, a pedantic reader would add), the probability of “ $A$  or  $B$ ” is smaller than  $\Pr[A] + \Pr[B]$ , but it cannot be larger.

The same argument gives us the following result:

$$\Pr[A \text{ or } B] = \Pr[A] + \Pr[B] - \Pr[A \text{ and } B].$$

Here, the event “ $A$  and  $B$ ” consists of the outcomes that are favorable both for  $A$  and  $B$ . Their probabilities are counted twice in  $\Pr[A] + \Pr[B]$ , hence we need to subtract  $\Pr[A \text{ and } B]$  to compensate for that.

Again, the formula above is a weighted version of the formula (8.1):

$$|A \cup B| = |A| + |B| - |A \cap B|.$$

**Problem 254** It is known that two events  $A$  and  $B$  in some probability space have probabilities 0.7 and 0.8. What is the *minimal* possible probability of an event “ $A$  and  $B$ ” (the intersection of both events)? What is the *maximal* probability of this event? Try it: [Coursera](#) ↗.

**Problem 255** Assume that we know  $\Pr[A]$ ,  $\Pr[B]$ ,  $\Pr[C]$ , and also  $\Pr[A \text{ and } B]$ ,  $\Pr[A \text{ and } C]$ ,  $\Pr[B \text{ and } C]$ , and, finally,  $\Pr[A \text{ and } B \text{ and } C]$ . How can we compute  $\Pr[A \text{ or } B \text{ or } C]$ ?

A hint: recall the formula for the size of the union of three sets (the inclusion-exclusion principle).

**Problem 256** There are ten events; each has probability at least  $1/3$ . Prove that there are cases when at least four of the events happen simultaneously. In other words, there are outcomes that are favorable for at least four of these ten events.

### 11.2.3 Sequential Choice

Imagine the following experiment. There are two boxes and six balls labeled 0, 1, 2, 3, 4, 5. The balls 0 and 1 are placed in one box and the balls 2, 3, 4, 5 are placed in the other box. You choose randomly one of the boxes (they look identical), take a random ball from this box, and read a number on it.

In this experiment every ball may be chosen. But do they have the same chances?

**Stop and Think!** Which ball has more chances to be chosen: 1 or 4?

It seems that ball 1 has better chances. It is chosen if (a) we choose the first box; (b) we choose ball 1 from the first box (and not ball 0). The ball 4 is chosen if (a) we choose the second box; (b) we choose ball 4 from this box (and not balls 2, 3, 4). The chances for 1 and 4 to pass the first stage (a) are the same, since the boxes look identical. But at the second stage (b) the chances for 1 to be chosen are better (since it has less competitors).

Here is the program that simulates this process.

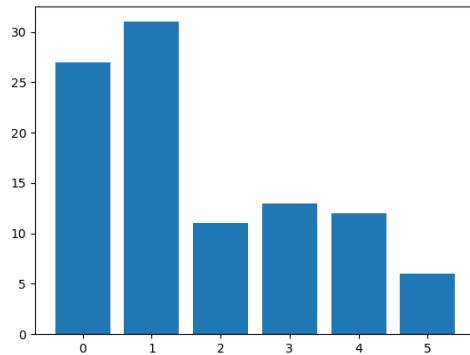
```
import matplotlib.pyplot as plt
from random import randint, seed

seed(27)

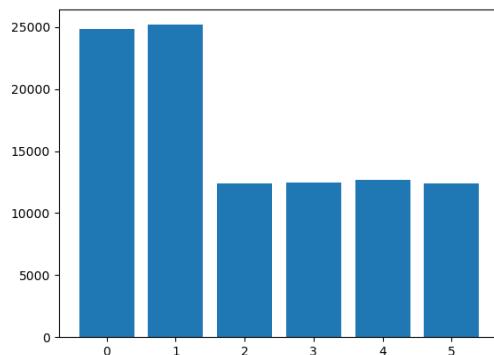
num_experiments = 100
count = [0] * 6

for _ in range(num_experiments):
    box = randint(0, 1)
    if box == 0:
        ball = randint(0, 1)
    else:
        ball = randint(2, 5)
    count[ball] += 1

plt.bar(range(6), count)
plt.savefig('twourns.png')
```



Probably you are not surprised that balls 0 and 1 appear more often than any other ball. Some fluctuations are visible (ball 0 appeared less times than 1 though nothing in our setting explains this; ball 5 is also a bit unlucky). We may increase the number of experiments; here are the results for  $N = 100\,000$ :

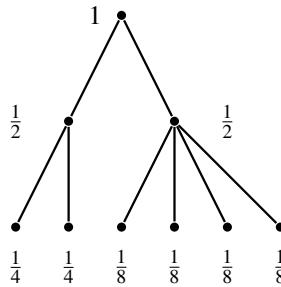


You may note that now the fluctuations are much less visible; the frequency of balls 0 and 1 is close to  $0.25 = 1/4$ , and the frequency of balls 2, 3, 4, 5 is close to  $0.125 = 1/8$ . So the combined

frequency of balls from the first box (0 and 1) is close to  $1/2$ , as well as the combined frequency of balls from the second box, and in each box the balls have approximately the same frequency. That corresponds to our description of the experiments (boxes are equiprobable, and for each box all the balls in it are equiprobable).

**Problem 257** Implement the random choice correctly, using random bits from the file `bittbabbler.txt`. (A lucky coincidence is that here we need to make equiprobable choice with two or four alternatives, and this is easy to do having a fair coin.) Does the behavior change?

Our setting can be called “sequential choice”: we made a sequence of choices (and the rules of next choice depend on the previous choices we made). The following diagram illustrates the process:



Top vertex indicates the box choice (half of the experiments go each way). Then, *depending on box chosen*, we make a random 2-way or random 4-way choice. The first half of the experiments is divided in two parts ( $1/2 = 1/4 + 1/4$ ), and the second half is divided in fourth parts ( $1/2 = 1/8 + 1/8 + 1/8 + 1/8$ ). Of course, these proportions are only approximate for a large number of experiments.

This diagram resembles the Galton board; the difference is that now the paths of choices are not combined (we cannot come in the same place in different ways), so we do not need to add the fractions.

**Stop and Think!** What is the probability to get a ball whose number is divisible by 3 (i.e., ball 0 or ball 3)?

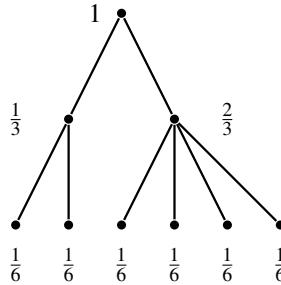
As we know now the probabilities of each outcome, we can just add them and get  $1/4 + 1/8 = 3/8$ .

**Problem 258** What is the probability to get a ball with odd number in this experiment?

The choices we make in such a scheme do not need to be equiprobable.

**Problem 259** Imagine that at the first stage we choose the first box (with two balls) with probability  $1/3$  (in one third of all experiments), and the second box (with four balls) with probability  $2/3$ . What will be the probabilities of all the outcomes?

It is easy to count:  $1/3$  of the experiments choose the first box, and then they are divided evenly:  $1/3 = 1/6 + 1/6$ . The rest ( $2/3$ ) are divided evenly between four outcomes:  $2/3 = 1/6 + 1/6 + 1/6 + 1/6$ . We see that all outcomes have probability  $1/6$ .



In this example the outcomes turn out to be equiprobable, though the setting has no symmetries: just a coincidence.

### 11.2.4 Mathematics for Prisoners

Consider a puzzle that illustrates the sequential choice scheme.

**Problem 260** A king decides to give a prisoner a chance of being set free. The king places two boxes, 15 white, and 15 black balls on the table. The prisoner can distribute the balls between two boxes in any way with the only constraint that no box is empty. After this, the king randomly picks one of the boxes with probability 1/2 each, and picks a random ball from that box (each ball in the box is picked with equal probability). If the ball happens to be white, then the prisoner is freed. Distribute the balls between the two boxes to maximize prisoner's chances. Try it: [Coursera](#).

**Stop and Think!** What are the chances of the prisoner if all black balls are put in one box, and all white balls in the second box?

In this case the fate of the prisoner depends on the choice of the box, and the chance to get a box with all white balls is 1/2.

**Problem 261** Assume that in the first box we have 10 white and 5 black balls, and the remaining balls (5 white and 10 black ones) are placed in the second box. What are the chances of the prisoner?

In the first box the white balls form a majority (2 : 1), and in the second box the black balls form a majority (1 : 2). In half of the cases the king selects the first box, and then this half is splitted in 2 : 1 proportion. For large number  $n$  of experiments, we have about  $n/2$  cases when the first box is chosen, and then these  $n/2$  cases are redistributed in 2 : 1 proportion, so we get  $(2/3)(n/2) = n/3$  cases when a white ball from the first box is chosen, and  $(1/3)(n/2) = n/6$  cases when a black ball from the first box is chosen. The same result (but reversed) is for the second box.

|        |       |                |
|--------|-------|----------------|
| first  | white | $\frac{1}{3}n$ |
| first  | black | $\frac{1}{6}n$ |
| second | white | $\frac{1}{6}n$ |
| second | black | $\frac{1}{3}n$ |

So the total number of experiments when a white ball is chosen (the prisoner is set free) is

$$\frac{1}{3}n + \frac{1}{6}n = \frac{1}{2}n,$$

so we get the same probability 50% to be set free as in the case when all balls of the same color go in the same box.

**Stop and Think!** Let us forget about the colors and remember only that there are 15 balls in each box. What are the chances of an individual ball to be chosen? (What is the probability distribution on the individual balls?)

The answer can be found without any computations. For symmetry reasons the chances of each ball in a given box are the same. Also, because of the symmetry between boxes, the chances for the balls in one box and the balls in the other one are the same. So every ball has the same chance to be chosen, i.e., 1/30.

If we now recall that there are 15 white balls, we conclude that the chances to get a white ball are  $15/30 = 1/2$ , without any computations.

**Stop and Think!** Can we conclude from the previous argument that the prisoner cannot improve her chances by redistributing the balls between boxes, since the number of white balls is fixed at 15? (Of course, we do not consider different versions of cheating, like hiding some black ball.)

Indeed, if each ball is selected with the same probability, it does not matter which of the balls are white and which are black, only the total number of white and black balls matters, and these numbers are fixed at 15. But this is a big *if*: if boxes contain different number of balls, the balls are not equiprobable. Imagine you are in the position of one of the balls and want to be selected. At the first stage you hope that your *box* will be selected. This happens with probability  $1/2$  (in approximately half of the experiments), and it does not matter how many other balls are in the same box. But at the second stage (if you were lucky at the first stage) you compete with the other balls in the box, so it is better to have fewer of them. Best case: you are alone in the box, then you are guaranteed to pass the second stage.

**Stop and Think!** You decide to consider the other extreme and put 1 ball in one box and 29 balls in the other box (14 of one color, and 15 balls of the other color). Which color should you choose for this lone ball?

The answer is obvious: the lone white ball is selected in 50% cases, and there are also white balls in the other boxes, so we get white ball in the majority of the experiments. And if we put black ball instead of white, we get a black ball (i.e., fail) in the majority of the experiments. So the first option is better (it make us free with probability greater than 50%, and the other option makes us free with probability less than 50%).

**Problem 262** Compute the probability for the prisoner to be freed if she places one white ball in the first box, and places 14 remaining white balls and 15 black balls in the other box.

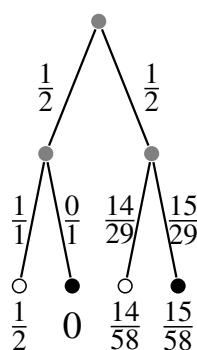
We have already seen how to compute this probability. For  $n$  experiments, we have (approximately)  $n/2$  experiments where the first box is chosen (and the prisoner is freed). For the other half of the experiments the second box is chosen. The proportion white : black for this box is 14 : 15, we have there 14 white balls out of 29, so the fraction of experiments where the white ball is taken, among all experiments where the second box is chosen, is  $14/29$ . In total we have

$$\frac{n}{2} + \frac{14}{29} \cdot \frac{n}{2} = \left( \frac{1}{2} + \frac{14}{29} \cdot \frac{1}{2} \right) \cdot n.$$

So the probability to become free is

$$\frac{1}{2} + \frac{14}{29} \cdot \frac{1}{2} = \frac{43}{58} \approx 0.741\dots$$

The selection process is shown as a tree: each branching point splits the experiments in some proportion (fractions are written on the edges); the final probabilities are written near the bottom row.



Rough estimate: the second box has almost as many white balls as the black ones; if not this small imbalance between black and white, we would have  $\frac{3}{4} = 0.75$ , and due to this disbalance we have slightly smaller chance to be freed.

We see that a clever redistribution of white and black balls increases our chances from 50% to more than 74%.

**Stop and Think!** The strategy described above gives probability 0.741... instead of 0.5. So we have found the optimal strategy for the prisoner, yes?

In fact, the answer is “yes and no”. Yes, because indeed our solution is optimal for the prisoner. No, because we did not prove this optimality. We considered only two extreme cases:

- each box contains 15 balls. Then the answer is  $1/2$  for any distribution of white balls.
- one box contains 1 ball and the other contains 29 balls. Then it is better to make the lone ball white; this give probability 0.741... to be freed.

But, may be, some other strategy is optimal?

**Problem 263** Can you prove that the strategy we described is optimal among all possibilities?

The rigorous argument we will provide is not that difficult and quite intuitive, but still requires some care.

To prove that answer  $43/58 = 0.741\dots$  is optimal, consider any other distribution of balls. We will show that it gives worse (or the same result) by gradually improving it, until we reach  $43/58$ . If we have improvement (or no change) at every step, then the original probability does not exceed  $43/58$ . So let us look on the arbitrary candidate solution.

*First optimization step.* One of the boxes (call it  $A$ ) is smaller, or may be both are the same (then it does not matter which of the boxes is called  $A$ ). Then every ball in  $A$  has the same or bigger probability to be chosen than every ball in the other box  $B$  (since the same  $1/2$  is divided between smaller number of balls or the same number of balls). If not all the balls in  $A$  are white, we can optimize our chances by exchanging black ball in  $A$  and white ball in  $B$ . (Why  $B$  contains at least one white ball?). We repeat this process until all balls in  $A$  are white (keeping the number of balls in both boxes and changing only the proportions).

*Second optimization step.* Now all balls in one of the boxes ( $A$ ) are white. If there is only one ball, we are done. If not, we can move one white ball from  $A$  to the other box. It does not change anything if box  $A$  is chosen (since all balls there are white anyway). But this move improves our chances if the other box is chosen: if we add a white ball, we increase the fraction of white balls and therefore our chances. In the same way we may move other white balls from  $A$  to the other box (except for the last one), and get our solution as promised.

Does this argument convince you? Please do not worry if not (and revisit it at the later stage).

### 11.2.5 Not All Questions Make Sense

We have learned a bit about probabilities, and it is time for the warnings. First, one should distinguish between mathematical notion of probability and the use of words “probability”, “probable” (and similar ones) in natural language. If you read something like “it is quite probable that, in the 12th century, alchemists obtained phosphorus through the distillation of urine”, it is perfectly OK, but one should not ask whether the probability of this event is greater than  $2/3$  or not. What kind of experiment could we have in mind? We cannot repeat the history  $n$  times and count the successes.

If you meet a friend and he asks you what is the probability that he has a dollar bill in his pocket, this question does not make much sense. What could it mean? You meet him regularly; should you compute the fraction of occasions when he has a dollar? Or you should consider only the occasions when he asks this question? Or you should consider some (vague) set of “people like him” and see how many of them have a dollar bill in their pockets? If you meet him here and now, he either has a dollar bill or not, so there are no fractions.

This all looks trivial, but people regularly make errors like that. One of these errors is related to probabilistic prime testing. A *prime* number is a positive integer greater than 1 that cannot be decomposed into a product for two such integers. The numbers that are not prime are called *composite*. For example, 7 is prime while 6 is composite (because  $6 = 2 \cdot 3$ ). One can check whether some  $n$  is prime by trying all integers  $2, 3, \dots, n-1$  and see whether  $n$  is a multiple of one of them.

In fact, it is enough to check only the numbers not exceeding  $\sqrt{n}$ . But it is still an infeasible task when you want to test, say, a 1000-digit number.

**Problem 264** Explain why we do not need to check numbers greater than  $\sqrt{n}$  while testing  $n$  for primality. [Hint: consider the smallest factor of  $n$ .]

Fortunately, mathematicians developed more efficient algorithms for primality testing. One of them, the Miller–Rabin primality test (see, e.g., [https://en.wikipedia.org/wiki/Miller-Rabin\\_primality\\_test](https://en.wikipedia.org/wiki/Miller-Rabin_primality_test)), is often used in practice. This algorithm uses randomization. This means that the answer (“prime” or “composite”) depends not only on the number  $x$  that we test but also on the random bits  $r_1, r_2, \dots$  used in the computation. We may imagine that algorithm tosses a coin to get these bits. So, for a given  $x$ , algorithm may give correct or incorrect answer depending on the bits used. And usually the answer of Miller–Rabin test is correct.

Large prime numbers (hundreds and thousands of digits) are used for cryptography, so primality tests are often used in practice. There are several quite efficient tests, and Miller–Rabin test is one of them. More recently, deterministic polynomial-time primality tests were discovered (by Manindra Agrawal, Neeraj Kayal, and Nitin Saxena, see [https://en.wikipedia.org/wiki/AKS\\_primality\\_test](https://en.wikipedia.org/wiki/AKS_primality_test)), but they are much less efficient in practice.

More precisely, Miller–Rabin algorithm has two properties:

- if  $x$  is prime, the algorithm says “prime” for all possible values of random bits (with probability 1);
- if  $x$  is composite, the algorithm says “composite” with probability close to 1 (for most values of the random bits).

The error probability (fraction of random bit sequences that lead to the wrong answer “prime”) depends on the parameters of the algorithm that could be adjusted. Smaller error probability requires more computation time, but it is quite feasible to get error probability, say,  $10^{-15}$ .

**Stop and Think!** We tested some number  $n$  using this algorithm and got answer “composite”. Can we be sure that the number is composite?

Yes, because the first property excludes answer “composite” for prime numbers.

However, the answer “prime” does not guarantee that the number is prime. It is possible that in reality the number  $x$  is composite, but the coin tossing in the algorithm produced (bad luck!) some rare sequence of bits that lead to a wrong answer “prime”.

All these statements are perfectly OK, the algorithms is widely used in practice and errors (appearing with small probability) were never a real problem. Still one should be careful: if you read something like “If  $n$  is positive and `IsPrime` returns `True`, then  $n$  is prime with a very high probability”, this makes no sense. Imagine that you ran this algorithm for some specific number  $n$  and the function returned `True`. Can you say that “the number [such-and-such] is prime with probability more than 99%”? No, because this specific number is either prime or composite, there are no probabilities here.

Another potentially dangerous situation are the infinite probability spaces and experiments with infinitely many possible outcomes.

**Stop and Think!** Somebody asks you: “Take a random non-negative integer; what is the probability that it will be a multiple of 3”? What will you answer?

It is tempting to say 1/3: for every number  $3n$  that is a multiple of 3, the next two numbers  $3n+1$  and  $3n+2$  are not, and then again we have a multiple of 3 (namely,  $3n+3 = 3(n+1)$ ). So all non-negative integers can be split into triples, and each triple contains exactly one multiple of 3. Still one should think more about the question before providing such and answer.

What kind of experiment do we have in mind here? We cannot make a dice with infinitely many sides labeled 0, 1, 2, …, so we need to imagine some other experiment. What could it be? One could, for example, toss a coin and count the number of heads before the first tail. This number can be indeed any non-negative number. But different numbers appear with different frequencies. For

example, in half of the cases the first coin shows us tail, so the number of heads before the first tail is 0, which is a multiple of 3 (since  $0 = 3 \cdot 0$ ) These cases alone guarantee that multiples of 3 will appear at least in 50% experiments, and not in  $1/3$  of them as our answer predicts.

**Problem 265** In what fraction of experiments we will get 1? will get 2? will get  $k$ ? In what fraction of experiments we will get a multiple of 3?

The answers can be computed by pencil and paper, but one can also get some hints from the following simulation.

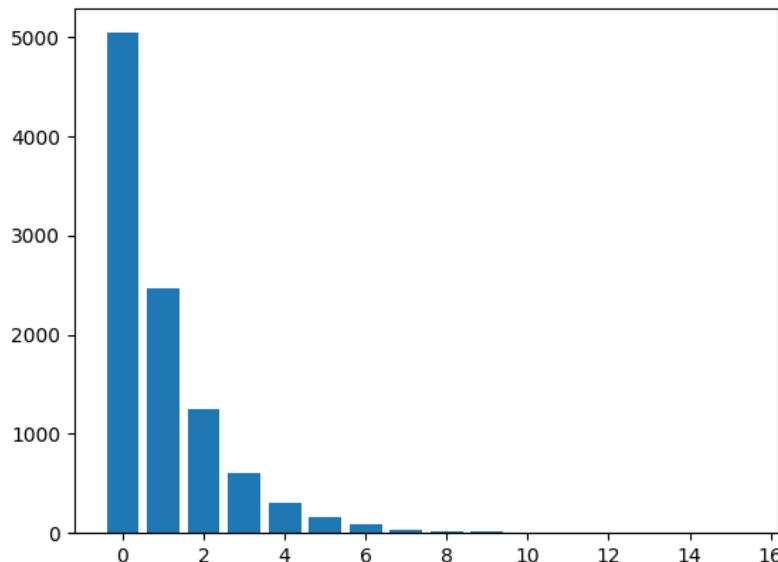
```
from random import randint, seed
import matplotlib.pyplot as plt

seed(12)

num_experiments, num_counts = 10000, 16
count = [0] * num_counts
for _ in range(num_experiments):
    trials = 0
    while randint(0, 1) != 1:
        trials += 1
    count[trials] += 1

plt.bar(range(num_counts), count)
plt.savefig('randint.png')
```

In this program we simulate  $N = 10000$  experiments. Each of them generates some integer (called `trials` in the program), namely, the number of zeros obtained before `randint(0, 1)` gives 1. Recall that this function is simulating a random integer in  $\{0, 1\}$ , i.e., simulates a coin tossing. When finally 1 appears and `trials` is determined, we update the array `count` that keeps frequencies of different integers. In fact, our program gives an error if 16 or more zeros appear before the first one, since we for simplicity artificially fixed the length of array `count` at 16. Still, with 10000 experiments this is quite unlikely and did not happen when we ran the program.



As the graph shows, the frequency of integer decreases (as expected) and rather fast. According to our prediction, about half of the cases produce 1 (about 5000 out of 10000).

Still the answer  $1/3$  can be interpreted in the following way: if we fix some very large number  $N$  and choose randomly a number in  $0 \dots N - 1$  (tossing an imaginary dice with  $N$  equiprobable faces), then the probability of each candidate will be  $1/N$ , we have about  $N/3$  multiples of 3 among candidates (with small boundary effects), so the probability to get a multiple of 3 is close to  $1/3$ .

However, a quite similar question “what is the probability that a random non-negative integer has even number of digits” cannot be interpreted in a similar way: most numbers among  $0 \dots 9999$  have even number of digits (about 90% have four digits), while most numbers among  $0 \dots 99999$  have odd number of digits (about 90% have five digits), etc.

Here is one more example of a dangerous question related to infinite probability space. Take three random points on a plane. What is a probability that they form (are vertices of) an acute triangle? (Acute triangle has all angles less than  $90^\circ$ ; right triangle has one angle  $90^\circ$ , and obtuse triangle has one angle greater than  $90^\circ$ .)

Again, before trying to provide a numerical answer, one should think carefully about the question. What kind of an experiment should we imagine here? We can take a sheet of paper, and put it under rain, and wait until three drops fall on the sheet. Then we construct a triangle with these vertices and check whether it is acute or not. Repeating the experiment many times, we may count the fraction of cases where an acute triangle appears.

This is a perfectly reasonable experiment. However, the answer *depends of the shape of the sheet we use*. If the paper is a rectangle with one short side and one long side, in most of the cases we will get an obtuse triangle. (Can you see why?) Even for standard A4 paper and square paper the answers look to be different, as the following simulation shows.

```
from random import random, seed
import matplotlib.pyplot as plt

seed(37)

def distance_squared(a, b):
    return (a[0] - b[0]) ** 2 + (a[1] - b[1]) ** 2

def acute(a, b, c):
    sides_squared = [
        distance_squared(a, b),
        distance_squared(a, c),
        distance_squared(b, c)
    ]
    sides_squared = sorted(sides_squared)
    return sides_squared[0] + sides_squared[1] > sides_squared[2]

def random_point(width, height):
    return [random() * width, random() * height]

def frequency_acute(width, height, num_experiments):
    experiments = [1 if acute(
        random_point(width, height),
        random_point(width, height),
        random_point(width, height)) else 0
```

```

        for _ in range(num_experiments)
    ]

    return [sum(experiments[:i]) / i for i in range(1, num_experiments)]

plt.plot(frequency_acute(100, 100, 10000), color='green')
plt.plot(frequency_acute(210, 297, 10000), color='red')
plt.plot(frequency_acute(100, 500, 10000), color='blue')
plt.ylabel(ylabel='Frequency of acute triangles')
plt.xlabel(xlabel='Number of experiments')
plt.ylim(bottom=0, top=1)
plt.savefig('freq_acute.png')

```

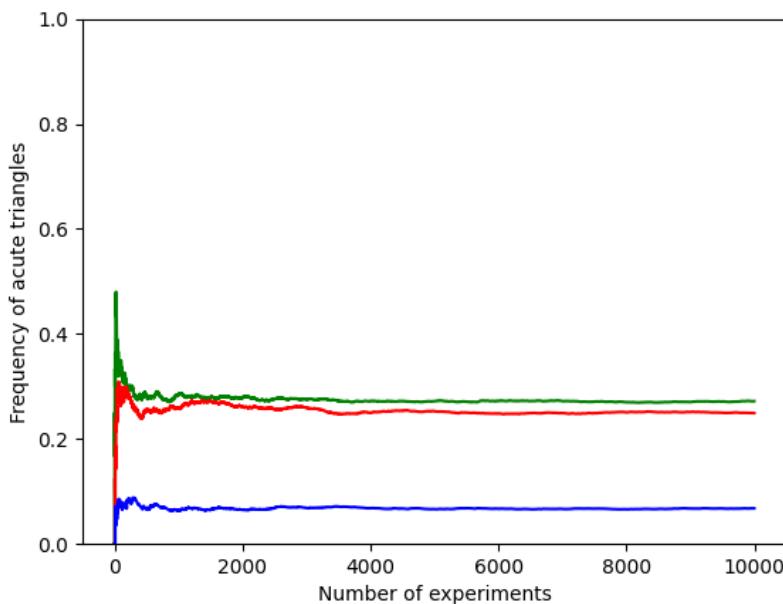
The function `distance_squared(a, b)` get two points (lists of length two) `a` and `b` and returns the square of distance between points with these coordinates (Pythagoras' theorem).

The function `acute(a, b, c)` determines whether the triangle with vertices `a`, `b`, `c` is acute: for that the square of the longest side should be smaller than the sum of squares of two other sides (for the right triangle we have equality thanks to Pythagoras). To check this we first sort the list `s` of squared sides.

The function `random()` is Python primitive that imitates choosing a random real between 0 and 1. It is used in the function `random_point(width, height)` to produce a “random” point in a rectangle of the given width and height.

Finally, the function `frequency_acute(width, height, num_experiments)` makes `num_experiments` experiments (each chooses a triangle and checks whether it is acute) and returns the array of frequencies of acute triangles (for every prefix of the sequence of experiments).

Finally, we draw these frequencies for three cases: for a square  $100 \times 100$  (green), for A4 paper  $210 \times 297$  (red), and for a rectangle  $100 \times 500$  (blue), each time producing 10000 triangles. As we have expected, the frequency of acute triangles is maximal for the square and minimal for a tall rectangle.



## 11.3 Conditional Probability

Now, we arrive at the central topic of probability theory (that distinguish it from combinatorics): the notions of *conditional probability* and *independence*.

### 11.3.1 What Is Conditional Probability

Let us start with some statistical data (taken from <https://www.census.gov/quickfacts/fact/table/US/PST045219>). The data says that about 15% of the people in the US are 65 years old or older, and about 44% of these group (people that are at least 65 years old) are male. (Women have bigger life expectancy, so it is not a surprise that there are more women than men among older people; as we see, the proportion is close to 4 : 3.)

**Stop and Think!** How to compute the fraction of males of age 65 or more among the entire US population, using this data?

For that you should take 44% of 15%, or multiply  $0.44 \cdot 0.15 = 0.066 = 6.6\%$ . Indeed, let  $n$  be the total number of people in the US. Then there are  $m = 0.15n$  people of age 65 or older, and among them there are

$$0.44m = 0.44 \cdot (0.15n) = (0.44 \cdot 0.15)n = 0.066n$$

men (of age 65 or older), i.e., 6.6% of the total population.

**Stop and Think!** Can you compute the fraction of females of age 65 or older among the total US population? Can you compute the fraction of males of age less than 65 among the entire US population?

The first question is easy: females form  $100\% - 44\% = 56\%$  of the people of age 65 or older, so they form

$$0.56 \cdot 0.14 = 0.0784 = 7.84\%$$

of the total population. But the second question *cannot be answered based on the data we have*. Can you see why? We know the fraction of males among the older people, but have no information about the fraction of males among the younger people.

Of course, the answer can be easily found in the published statistics, but we speak about conclusions from our data. Indeed, imagine that many young people (below 65) change their sex. This will not change both our data (the fraction of people that are 65 years or older, and the fraction of males among this group). Still the fraction of males among the young people may change.

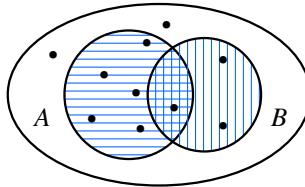
Now let us repeat the same simple computation in the language of probability theory. First, we say that *a random American is 65 years or older with probability 0.15*. If you imagine some mystical procedure that selects a random American person in such a way that everyone has the same chance to be selected, then the probability of choosing someone of age 65 or older is 0.15.

This procedure hardly can be implemented in reality: if you call a random phone number, you will miss people who do not have phones, etc. Pollsters do all kind of tricks to approximate such a procedure.

The fraction of males among the people of age 65 or older is called a *conditional probability of being male under the condition of being 65 year old or older*. Then our computation can be written as

$$\Pr[\text{male and (65 or older)}] = \Pr[65 \text{ or older}] \cdot \Pr[\text{male} \mid 65 \text{ or older}].$$

In general, consider a probability space made of equiprobable outcomes and two events  $A$  and  $B$  in this space. Then the conditional probability  $\Pr[A \mid B]$  (read as “conditional probability of  $A$  given  $B$ ”, or “probability of  $A$  under the condition  $B$ ”) is the fraction of  $A$ -elements *among B-elements*.



We can say that we restrict our probability space and consider outcomes only in  $B$ , and count the fraction of  $A$ -elements there.

**Stop and Think!** What is  $\Pr[A | B]$  for this picture?

It is  $1/3$ : one of three points from  $B$  belongs also to  $A$ . (Note that we do not need to count points outside  $B$ .)

**Problem 266** For the same picture: What is  $\Pr[B | A]$ ? What is  $\Pr[B | \text{not } A]$ ?

We have the same formula:

$$\Pr[A \text{ and } B] = \Pr[B] \cdot \Pr[A | B];$$

it says that the fraction of  $A \cap B$  in the entire space is the fraction of  $B$  in the entire space times the *fraction of  $A$ -elements among  $B$ -elements*. The second factor is the conditional probability of  $A$  with condition  $B$ .

For arbitrary probability spaces this formula becomes a definition. Assume that  $A$  and  $B$  are two events in the same probability space and  $\Pr[B] \neq 0$ . Then the conditional probability  $\Pr[A | B]$  is defined as follows:

$$\Pr[A | B] = \frac{\Pr[A \text{ and } B]}{\Pr[B]}.$$

**Stop and Think!** Recall our demographic example. What is the conditional probability of being female under the condition “age at least 65”? What is the conditional probability of being at least 65 under the condition of being male?

The first question is easy to answer: as we have mentioned, if 44% of people of age at least 65 are male, then  $100\% - 44\% = 56\%$  of the same group are female:  $\Pr[\text{female} | 65 \text{ or older}] = 0.56$ .

To answer the second question, we need more data. By definition, we have to find the fraction

$$\frac{\Pr[\text{male and (65 or older)}]}{\Pr[\text{male}]}$$

The numerator of this fraction is known, but we do not know its denominator: our data does not say how often males appear among the people below 65, so the fraction of males in the entire population cannot be derived from the data.

The next simple problem is just to check that we are on the same page.

**Problem 267** We roll a dice with 6 equiprobable sides and get a random number  $x$  in  $1, 2, 3, 4, 5, 6$ . Find the conditional probability

$$\Pr[(x \text{ is even}) | (x \geq 3)].$$

Here the condition consists of four outcomes  $3, 4, 5, 6$ , and two of these outcomes (4 and 6) are even. So the conditional probability in question is  $2/4 = 1/2$ .

Another explanation:  $\Pr[x \text{ is even and } x \geq 3]$  is  $2/6 = 1/3$ , and  $\Pr[x \geq 3]$  is  $4/6 = 2/3$ , so

$$\Pr[x \text{ is even} | (x \geq 3)] = \frac{\Pr[x \text{ is even and } x \geq 3]}{\Pr[x \geq 3]} = \frac{1/3}{2/3} = \frac{1}{2}.$$

**Problem 268**

1. We roll a dice with 6 equiprobable outcomes 1, 2, 3, 4, 5, 6. Find the conditional probability

$$\Pr[\text{outcome is odd} \mid \text{outcome is even}].$$

2. Six people form a queue, including  $A$  and  $B$  (all  $6!$  orderings are equiprobable). What is the probability that  $B$  is earlier in the queue than  $A$  under the condition that  $A$  is not the first?

3. We roll two dice (all 36 outcomes are equiprobable). What is the conditional probability to have 1 on the first dice under the condition that the sum of two numbers is 6?

4. Recall the puzzle about the prisoner and the king. Imagine that the prisoner puts 1 white ball in the first box, and 14 white and 15 black balls in the other box. Then the king chooses a random box and then chooses a random ball inside this box. What is the conditional probability of the event “the king chooses the second box” under the condition “the ball chosen by the king was white”?

Try it: [Coursera ↗](#).

The following question is similar to the last question of the preceding problem and can be solved in a similar way. However, this case is more symmetric, and there is another argument that solves the question without pencil and paper; can you see it?

**Problem 269** There are two boxes; the first one contains 10 white balls and 5 black balls; the other one contains 10 black balls and 5 white balls. The king randomly selects a box (with equal probabilities) and then randomly takes a ball from this box (with equal probabilities). What is the probability that the king selected the first box under the condition that the ball he selected is white? Try it: [Coursera ↗](#).

**11.3.2 How Reliable Is The Test?**

We often see some statistical data in the news. For example, during the COVID-19 pandemic we all read the news about the vaccine efficacy and were delighted to see that for some vaccines it exceeded 90% in the efficacy studies. However, not everybody can say clearly what these 90% mean. It does not mean, for example, that 90% vaccinated people will not contract the disease. Obviously, the chance to get ill depends not only on the properties of the vaccine but also on the epidemiological situation: in a country where only few people have the disease, even non-vaccinated people will have better chances to avoid it. What this 90% really means is that

during the Phase 3 efficacy studies, the fraction of people who contracted the disease during the test period among the vaccinated participants is ten times less than the fraction of people who contracted the disease in the same period among non-vaccinated participants.

One needs not only to know the statistical data, but understand their real meaning, and the notion of conditional probability is important here. Our intuition is not always right if not trained. Let us consider the following (artificial) example.

Imagine a disease (medical condition) that 1% of the population has. There is also a test for this disease that is not completely reliable and can give false positives and false negatives. “False positive” means that the test detects the disease in a healthy person; “false negative” means that an existing disease is not detected by the test. Assume that the test has 10% false positives rate and 10% false negatives rate, i.e., 10% of healthy people have positive test results, and 10% of ill people have negative test results. So, whatever you are ill or not, with 90% probability the test will give the correct answer.

Imagine now that you made a test and are informed that your test is positive. Of course you should be worried, but how much? What are the chances that you are really ill? Many people will probably say something like “Look, the test is 90% reliable, so I am most probably ill, even if the disease is quite rare”.

**Stop and Think!** Do you agree with this reasoning?

Let us reformulate this question in the language of probability theory. There are two events: “have a disease” ( $D$ ) and “have a positive test” ( $T$ ), and we assume the following:

- $\Pr[D] = 0.01$  (1% of population is ill)
- $\Pr[T \mid \text{not } D] = 0.1$  (10% false positives rate)
- $\Pr[\text{not } T \mid D] = 0.1$  (10% false negatives rate)

**Stop and Think!** Can you find the fraction of ill people among people with positive test, i.e.,  $\Pr[D \mid T]$ ?

Let us analyze the situation step by step. There is 1% of people who are ill, and 90% of them are correctly diagnosed, so there are 0.9% people who are test-positive and ill.

On the other hand, there are 99% of people who are healthy, and 10% of them are incorrectly diagnosed (are test-positive). So there are 9.9% of people that are test-positive and healthy.

So we counted all test-positive people (both ill and healthy) and get 0.9% and 9.9% of the entire population. So there are much more healthy people among the test-positive, in proportion  $9.9 : 0.9 = 11 : 1$ . So, contrary to our non-trained intuition, *11/12 of the test-positive people are healthy*. Only one of twelve test-positive people is really ill.

**Problem 270** What fraction of the test-negative people are healthy? In our notation, what is  $\Pr[\text{not } D \mid \text{not } T]$ ?

Let us repeat the computation we made, using the notation of probability theory.

- Since  $\Pr[\text{not } T \mid D] = 0.1$ , we have  $\Pr[T \mid D] = 0.9$ . (If 10% of the ill people have negative test, then 90% of ill people have positive test.)
- $\Pr[T \text{ and } D] = \Pr[D] \cdot \Pr[T \mid D] = 0.01 \cdot 0.9 = 0.009$ .
- Since  $\Pr[D] = 0.01$ , we have  $\Pr[\text{not } D] = 0.99$ .
- $\Pr[T \text{ and } (\text{not } D)] = \Pr[\text{not } D] \cdot \Pr[T \mid (\text{not } D)] = 0.99 \cdot 0.1 = 0.099$ .
- $\Pr[T] = \Pr[T \text{ and } D] + \Pr[T \text{ and } (\text{not } D)] = 0.009 + 0.099 = 0.108$ .
- $\Pr[D \mid T] = \Pr[D \text{ and } T] / \Pr[T] = 0.009 / 0.108 = 9 / 108 = 1 / 12 \approx 8.3\%$

This reasoning is a special case of some general rule that is solemnly called *The Law of Total Probability* (while being quite simple). Imaging that our probability space is split into mutually exclusive cases, i.e., there some events  $B_1, \dots, B_n$  such that

- $X = B_1 \cup B_2 \cup \dots \cup B_n$  (every outcome belongs to some  $B_i$ );
- $B_i \cap B_j = \emptyset$  for all  $i \neq j$  (no outcome belongs to two different events).

Then every event  $A$  is also split into  $n$  mutually exclusive parts “ $A$  and  $B_1$ ”, “ $A$  and  $B_2$ ”, …, “ $A$  and  $B_n$ ” and therefore

$$\Pr[A] = \Pr[A \text{ and } B_1] + \dots + \Pr[A \text{ and } B_n].$$

Each summand can be rewritten in terms of conditional probabilities, and we get

$$\Pr[A] = \Pr[B_1] \cdot \Pr[A \mid B_1] + \dots + \Pr[B_n] \cdot \Pr[A \mid B_n]$$

(the law of total probability).

We applied this law for two events:  $B_1$  was  $D$  and  $B_2$  was “ $\text{not } D$ ”.

### 11.3.3 Bayes' Theorem

In this section we consider a simple mathematical result that is often mentioned in discussions about foundations of probability theory: the Bayes theorem. (There is even a school of thought called “Bayesianism”.) Let us start with a real life example.

Imagine you got an e-mail with the subject line

Subject: MESSAGE FROM BANK OF AMERICA

and address line:

From: <info4@water.ocn.ne.jp>

(here .jp is the domain of Japan — and not America). This looks as a scam message, but what are the reasons for that? There are three factors that support this conclusion:

- many scam messages use foreign e-mail address;
- foreign e-mail address is rather unusual in general;
- scam messages are quite frequent nowadays.

All three factors are important. If scam messages almost never used foreign address, then the foreign address would be a reason to trust the message. If almost every message used foreign address, then the foreign address will not tell anything about the message. Finally, if scam message were extremely rare, we would rather believe that some valid message just has a foreign address for some reason.

Now let us see how this reasoning is reflected in Bayes' theorem. Consider two events; we call them  $H$  and  $E$  (for “hypothesis” and “evidence”), but they are just two arbitrary events in the same probability space. Assume that both events  $E$  and  $H$  have non-zero probabilities. Then the following *Bayes formula* holds:

$$\Pr[H | E] = \frac{\Pr[H \text{ and } E]}{\Pr[E]} = \frac{\Pr[E | H] \Pr[H]}{\Pr[E]}.$$

The first equality is the definition of the conditional probability  $\Pr[H | E]$ . The second equality follows from the definition of the conditional probability  $\Pr[E | H]$ .

So the Bayes formula (or theorem) is the direct consequence of the definition of conditional probability. Let us compare this formula with our example. We have

$H$  = message is a scam,

and

$E$  = message uses a foreign address

We are interested in  $\Pr[H | E]$ , the probability that the message is a scam under the condition that it has foreign address. The formula says that  $\Pr[H | E]$  differs from  $\Pr[H]$  by a factor  $\Pr[E | H] / \Pr[E]$ . This factor increases or decreases  $\Pr[H | E]$  compared to  $\Pr[H]$  (the latter is called *a priori* probability of the hypothesis) if it is greater than 1 or less than 1.

If  $\Pr[E | H]$  is much bigger than  $\Pr[E]$  (the evidence happens much more often when  $H$  happens than in general), then the factor is large, and the probability of the hypothesis increases significantly.

If  $\Pr[E | H]$  is much smaller than  $\Pr[E]$  (if  $H$  is true, this makes  $E$  much less likely), then the evidence is really an evidence *against* the hypothesis, so  $\Pr[H | E]$  is much smaller than  $\Pr[H]$ .

Finally, if  $\Pr[E | H]$  is equal to  $\Pr[E]$  (or very close to it), then evidence does not say much about the hypothesis, and  $\Pr[H | E]$  is equal (or very close) to  $\Pr[H]$ .

**Stop and Think!** In our preliminary analysis we mentioned three factors that are taken into account when we decide that the message is probably a scam. Can you see which terms in the Bayes formula correspond to these factors?

Let us recall these factors and find the corresponding terms:

- “many scam messages use foreign e-mail address”:  $\Pr[E | H]$  is rather big;
- “foreign e-mail address is rather unusual in general”:  $\Pr[E]$  is rather small;
- “scam messages are quite frequent nowadays”:  $\Pr[H]$  is significant.

Take-home message from Mr. Bayes:

*if condition B increases [decreases] the probability of A, then, symmetrically,  
condition A increases [resp. decreases] the probability of B by the same factor.*

As usual, the history is much more complicated: Bayes applied this rule to a special case with continuous parameter, not just discovering a (simple) formula that is now called after him, and other people used the same formula independently of Bayes.

**Problem 271** The conditional probability  $\Pr[B | A]$  is  $4/5$ ; the conditional probability  $\Pr[B | \text{not } A]$  is  $2/5$ , and the unconditional probability of  $B$  is  $1/2$ . What is the probability of  $A$ ? Try it: [Coursera](#).

This problem may look difficult at first. You may wish to start with the following question: *in which proportion you should mix 80% solution and 40% solution to get 50% solution?*

#### 11.3.4 Conditional Probability: A Paradox

“Mary has two children; one of them is a girl. What is the probability that the other one is also a girl?” This question is often used to illustrate troubles people have with conditional probability. We consider two different ways to find this probability that give different results and discuss which of them is correct, but first let us switch from real life to coin tossing (and avoid biological questions whether having a girl increases your chances that your next child is a girl, too — and about twins).

**Stop and Think!** Mary tossed the fair coin twice. At least one of the outcomes is a tail. What is the probability that she gets two tails?

We know that one of the coin tosses gave a tail. We may assume without loss of generality that it was the first one. (If it was the second one, the argument is symmetric.) Then there are two cases: the second coin tossing gives head (the sequence TH), or tail (TT). In one of the two cases we have two tails, so the probability is one out of two ( $1/2$ ).

**Stop and Think!** Does this argument sounds convincing?

If so, let us consider another computation that gives a different answer. Our probability space for two coins consists of four equiprobable outcomes (HH, HT, TH, TT). The event “at least one tail” consists of three of them (all except HH) and has probability  $3/4$ . The event “both tails” is a part of the previous event, consists only of one outcome (TT) and has probability  $1/4$ . So the conditional probability by definition is

$$\Pr[\text{two tails} | \text{at least one tail}] = \frac{1/4}{3/4} = 1/3.$$

Formally the definition requires the probability of the event “(at least one tail) and (two tails)” in the numerator, but this is just the event “two tails”.

**Stop and Think!** Which of the two different answers ( $1/2$  or  $1/3$ ) is correct?

It is not just a theoretical question. Imagine we asked Mary to repeat the experiment (two coin tossings) thousand times and write the result of each experiment (a sequence of two letters) on a separate paper card. We get thousand cards. Then we go through the cards and keep only the cards where one of the letter is T. Going through this smaller pile, we find the fraction of cards where both letters are T.

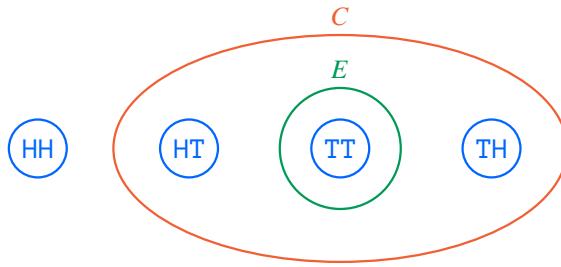
**Stop and Think!** What do you expect: will this fraction will be close to  $1/2$  or  $1/3$  (or something else)?

We expect that all four types of cards (HH, HT, TH, TT) appear equally often, so there are about 250 cards of each type (with some small fluctuations). After we delete HH-cards, we have about 750 cards of three types HT, TH, TT. Among them are about 250 cards of type TT, so the fraction should be close to  $1/3$ .

In more formal way the same reasoning can be explained as follows. First, we should parse the problem statement.

Mary tossed a fair coin twice. At least one of the outcomes is a tail. What is the probability that she got two tails?

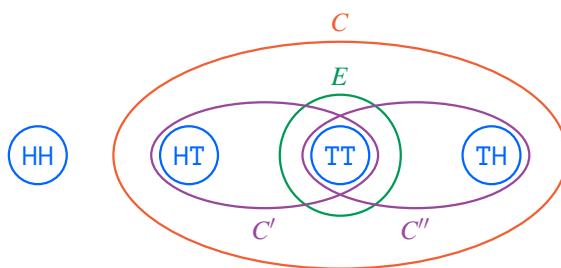
Here the part “Mary tossed a fair coin twice” is the description of the probability space. The part “At least one of the outcomes is a tail” is a condition (denoted by  $C$  in the picture). The part “she got two tails” is the event whose (conditional) probability we compute; we denote it by  $E$ .



Looking at this picture, we see that

$$\Pr[E | C] = \frac{1}{3}.$$

Moreover, we can see how we got the answer  $1/2$ .



When we say “one outcome is a tail; we may assume that the first one is a tail”, we replace the condition  $C$  by  $C'$ , and this changes the conditional probability:

$$\Pr[E | C'] = \frac{1}{2},$$

and the same for the symmetric condition  $C''$ : we also have  $\Pr[E | C''] = 1/2$ . And indeed, if  $C$  happens then either  $C'$  or  $C''$  happens, but this does not mean that we may replace  $C$  by  $C'$  or  $C''$  when computing the conditional probability.

In fact, some part of the blame goes to the teacher who asked the question. When she says “Mary has two daughters”, does she mean some specific person named Mary? No, because in this case the statements are either true or false, there is no randomness. And if the teacher does not have a specific Mary in mind, what does she mean? All girls with the same name that have two children? something else? The past tense in “Mary tossed a fair coin twice” is also confusing.

There is a long tradition to present the condition as some “information” about the situation. We may consider the phrase “Mary tossed the fair coin twice, at least one outcome is a tail” as a description of the probability space (with three outcomes), and then ask for the (unconditional) probability of the event “two tails” in this space ( $1/3$ ). This gives the same result, but one should be careful.

Take-home message: *one should first parse the problem statement trying to extract the description of probability space, the condition, and the event.* If you are in trouble doing this, think about a possible experiment that corresponds to the question — or try to simulate this experiment by a computer program, this could help.

Let us consider an example. Here is a puzzle cited in a documentary about math circles (“Mathematical Circles Around the World”, 2020).

**Problem 272** There are 2 normal apples and 4 poisoned apples. You die if you eat two poisoned apples. You have eaten 2 apples and you are still alive. What is your chance to survive if you will eat the 3rd apple?

First, we need to guess what setting the problem’s authors had in mind. Probably they consider a box with 6 apples (2 normal and 4 poisoned). You die if you eat *at least* two poisoned apples. You

have eaten 2 apples *from this box* and you are still alive. What is your chance to survive if you will eat the 3rd apple *from the same box*?

This is a question about conditional probability (formulated in a usual way).

- Outcomes are ordered sequences of three apples (in the order they are eaten) from the box.
- The condition is formed by all sequences where there is at most 1 poisoned apple among the first two apples.
- The event is formed by all sequences that contain at most 1 poisoned apple.

Now one can write a simple program that considers all outcomes and counts those who satisfy the condition and belong to the event. However, one can also do it almost without computations.

Our event  $E$  (at most one bad apple among three chosen) is a part of the condition  $C$  (at most one bad apple among the first two), so in this case the conditional probability is  $\Pr[E]/\Pr[C]$  (the intersection of  $E$  and  $C$  is  $E$ ). Let us compute the probabilities:

- To have at most one bad apple in the 3-apple sequence we have to eat both good apples and one bad. This can be done (if we ignore the ordering, which gives additional factor  $6 = 3!$ ) in four ways (depending on the bad apple) out of  $\binom{6}{3} = 20$  (again we ignore ordering), so  $\Pr[E] = 1/5$ .
- To compute the probability of  $C$ , we compute the probability of “not  $C$ ”, this is a bit easier. “Not  $C$ ” means that we ate two bad apples; they can be chosen among 4 bad apples in  $\binom{4}{2} = 6$  ways, while the total number of possible choices of two apples is  $\binom{6}{2} = 15$ , so “not  $C$ ” has probability  $6/15 = 2/5$ , and  $C$  has probability  $1 - (2/5) = 3/5$ . (Again the additional factors related to ordering and the third apple are ignored, since they are the same.)

In this way we get the answer

$$\Pr[E \mid C] = \Pr[E \text{ and } C] / \Pr[C] = \Pr[E] / \Pr[C] = \frac{1}{3}.$$

Another way to find the probability of having the first two apples bad (“not  $C$ ”): the probability that the first apple is bad is  $4/6$  (four bad apples out of six), and the conditional probability that the second apple is bad *under the condition that the first apple is bad* is  $3/5$  (three bad apples out of five), so the answer is  $(4/6) \cdot (3/5) = 2/5$ .

**Problem 273** Mary tosses a fair coin three times. What is conditional probability of the event “all tails” under the condition “at least two tails”? Try it: [Coursera](#).

### 11.3.5 Past and Future

Does the past influence the future? If you ask this as a philosophic question, the answer would be obviously positive. But let us ask a specific question of this type.

**Stop and Think!** We roll a dice and get six. Does it increase or decrease the chance to get six next time?

In terms of conditional probability the question can be reformulated as follows: is the conditional probability of the event “the second result is 6” under the condition “the first result is 6” greater or smaller than the unconditional probability of the event “the second result is 6”?

In our mathematical model of dice rolling the answer is easy to compute. The probability space consists of all pairs  $11, 12, \dots, 16, 21, 22, \dots, 26, \dots, 61, 62, \dots, 66$  (36 equiprobable outcomes). The condition  $C =$  “the second result is 6” consists of six outcomes  $16, 26, \dots, 66$ , the event  $E =$  “the first result is 6” consists of six outcomes  $61, 62, \dots, 66$ . Both have probability  $1/6$ . The event “ $E$  and  $C$ ” contains only one outcome 66. The conditional probability is then

$$\Pr[E \mid C] = \frac{\Pr[E \text{ and } C]}{\Pr[C]} = \frac{1/36}{1/6} = \frac{1}{6},$$

i.e., the same as the unconditional probability of  $E$ .

However, it is a property of our mathematical model, not of a real dice. Does the Nature follow our model? A famous American novelist Edgar Poe wrote in his “Marie Rogêt” story (1842):

“Nothing, for example, is more difficult than to convince the merely general reader that the fact of sixes having been thrown twice in succession by a player at dice, is sufficient cause for betting the largest odds that sixes will not be thrown in the third attempt. A suggestion to this effect is usually rejected by the intellect at once. It does not appear that the two throws which have been completed, and which lie now absolutely in the Past, can have influence upon the throw which exists only in the Future. The chance for throwing sixes seems to be precisely as it was at any ordinary time—that is to say, subject only to the influence of the various other throws which may be made by the dice. And this is a reflection which appears so exceedingly obvious that attempts to controvert it are received more frequently with a derisive smile than with anything like respectful attention. The error here involved—a gross error redolent of mischief—I cannot pretend to expose within the limits assigned me at present; and with the philosophical it needs no exposure. It may be sufficient here to say that it forms one of an infinite series of mistakes which arise in the path of Reason through her propensity for seeking truth in detail.”

It seems that Edgar Poe claims that

- people (“merely general readers”) believe that the outcomes of past dice rolling do not influence the probabilities for the next one;
- by thinking so, they make “a gross error redolent of mischief”.

It is hard to say whether Poe was serious (he was known for some practical jokes), but modern science agrees with the viewpoint that he criticizes.

The error made by Poe is similar to the reasoning of a crazy mathematician from a joke who computed the probability of having two bombs on a plane, found that this probability is negligible and then decided to take a bomb with him so he can be sure that there is no other bomb.

**Stop and Think!** You see that a coin tossing produced 92 heads in a row and you want to make a bet for the next result. Will you bet on head or tail?

You may see such a situation in Tom Stoppard’s play “Rosencrantz and Guildenstern Are Dead” (and a movie based on it).

Edgar Poe could say that since there were more tails than heads, the coin should be fair and compensate this bias by producing a head. Mathematician would say that for a fair coin the past performance is not indicative of future results, and head and tail are equiprobable. Guildenstern (in Stoppard’s play) becomes frightened: “...If we postulate, and we just have, that within un-, sub- or supernatural forces the probability is that the law of probability will not operate as a factor, then we must accept that the probability of the first part will not operate as a factor, in which case the law of probability will operate as a factor within un-, sub- or supernatural forces. And since it obviously hasn’t been doing so, we can take it that we are not held within un-, sub- or supernatural forces after all; in all probability, that is.” (One should be forgiven for being slightly hysterical when seeing 92 heads in a row.) A statistician would say simply that it is reasonable to reject the hypothesis of a fair coin and look more carefully on the specific experimental setting (may be, for example, the coin has two heads and no tail).

The last example illustrates the boundary between probability theory and mathematical statistics. Roughly speaking, probability theory studies the consequences of a given model (probability distribution), while statistics studies how one should choose a statistical model that fits well the experimental data.

### 11.3.6 Independence

The notion of independence is central in probability theory. We define the mathematical notion of independence: two events are independent if one of them, added as a condition, does not change the probability of the other one. More formally: two events  $A$  and  $B$  in the same probability space, having non-zero probabilities, are independent if  $\Pr[A | B] = \Pr[A]$ .

**Stop and Think!** Our terminology suggests that the independence relation is symmetrical: if  $A$  and  $B$  are independent, then  $B$  and  $A$  are independent. Is this really the case?

In other words, assume that  $\Pr[A | B] = \Pr[A]$ . Does it mean that  $\Pr[B | A] = \Pr[B]$ ? To see why it does, recall the definition of conditional probability:

$$\Pr[A | B] = \frac{\Pr[A \text{ and } B]}{\Pr[B]}.$$

This fraction equals  $\Pr[A]$  if

$$\Pr[A \text{ and } B] = \Pr[A] \cdot \Pr[B],$$

and this requirement is symmetrical. So we can safely speak about “two independent events  $A$  and  $B$ ” and do not care in which order they are listed.

**Stop and Think!** Why did we require the events to have non-zero probabilities?

This was needed to speak about conditional probabilities: if  $\Pr[B] = 0$ , then the fraction  $\Pr[A | B] = \Pr[A \text{ and } B] / \Pr[B]$  is not defined.

If one of the events  $A, B$  has zero probability (or both), then their intersection has zero probability, and the product  $\Pr[A] \cdot \Pr[B]$  also equals zero, so the displayed equation is true. It is convenient to extend our definition and say that the events are independent in this case. So we arrive to our final definition:

Events  $A$  and  $B$  in the same probability space are independent if

$$\Pr[A \text{ and } B] = \Pr[A] \cdot \Pr[B].$$

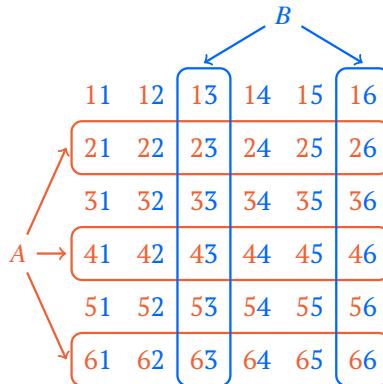
This equation is often called a *product rule*, but the name is a bit misleading, since it is a *definition*, not a statement.

A basic example of independence: rolling of two dices (say, red and blue one).

|          |    |    |    |    |    |          |
|----------|----|----|----|----|----|----------|
|          |    |    |    |    |    | <i>B</i> |
| 11       | 12 | 13 | 14 | 15 | 16 |          |
| 21       | 22 | 23 | 24 | 25 | 26 |          |
| 31       | 32 | 33 | 34 | 35 | 36 |          |
| 41       | 42 | 43 | 44 | 45 | 46 |          |
| 51       | 52 | 53 | 54 | 55 | 56 |          |
| <i>A</i> | 61 | 62 | 63 | 64 | 65 | 66       |

The event  $A$  is “6 on the first dice”, and  $B$  is “6 on the second dice”. They are shown in red/blue since the numbers on the first/second dice are shown in the same color. Probability of  $A$  and  $B$  are  $1/6$  (one column or row out of six), and the probability of the intersection “ $A$  and  $B$ ” is  $1/36$ . So the events are independent according to our definition.

Let us consider other events in the same space, say, let  $A$  be “the number on the first dice is even” and let  $B$  be “the number on the second dice is a multiple of three”. The picture changes:



Now  $\Pr[A] = 1/2$  (three rows out of six are one half of all outcomes),  $\Pr[B] = 1/3$  (two columns out of six). Finally,  $\Pr[A \text{ and } B] = 6/36 = 1/6$ , since the intersection of the red and blue zone, i.e., of events  $A$  and  $B$ , contains 6 outcomes out of 36. The product  $\Pr[A] \cdot \Pr[B]$  equals  $1/6$ , so the events are independent according to our definition.

**Stop and Think!** Can you see why these events are independent without computations?

It is convenient to use the version of the definition in terms of conditional probabilities and show that, say,

$$\Pr[B | A] = \Pr[B].$$

The conditional probability  $\Pr[B | A]$  is the fraction of  $B$ -outcomes among  $A$ -rows (shown in red). Each row contains the same number of  $B$ -outcomes (two in our case), so we can look at the fraction of  $B$ -outcomes in one row, in any group of rows it will be the same. In particular, it will be the same in  $A$  and in all rows — in other words,  $\Pr[B | A]$  is equal to  $\Pr[B]$ .

This argument can be applied to arbitrary events  $A$  and  $B$  that speak about first and second dice. This corresponds to the intuitive idea that the first and second dice are independent, one does not influence the other one. Still, as we have said, one should distinguish the mathematical notion of independence and the common sense meaning of the word “independence”.

**Stop and Think!** Consider an experiment with one dice only (6 equiprobable outcomes), and two events  $A$  = “the number is even” and  $B$  = “the number is divisible by three”. Are they independent?

Both events refer to the same dice, so there are no intuitive reasons for these events to be independent. Still they are: the probability of  $A$  is  $3/6 = 1/2$ , the probability of  $B$  is  $2/6 = 1/3$ , and they happen together only if we get 6, so  $\Pr[A \text{ and } B] = 1/6 = (1/2) \cdot (1/3)$ . Just a kind of coincidence.

**Stop and Think!** On the other hand, imagine that students in a class take two mathematical tests conducted by different (and independent) educational testing services. Would you expect that the events “pass the first test” and “pass the second test” are independent?

According to our definition, it would mean that the fraction of people who passed both tests equals the product of two fractions for both tests. Or, in terms of conditional probabilities, that the fraction of people that passed the second test, is the same among the people who passed the first test and in the entire group.

And there are no reason to expect this (the test results, we hope, are not produced by coin tossing and reflect some abilities of the people tested). Normally the people who passed the first test are better in math than average, so in this group the fraction of people who passed the second test could be bigger.

If two events are not independent, i.e.,  $\Pr[A \text{ and } B] \neq \Pr[A] \cdot \Pr[B]$ , there are two possibilities:

$$\Pr[A \text{ and } B] > \Pr[A] \cdot \Pr[B],$$

or

$$\Pr[A \text{ and } B] < \Pr[A] \cdot \Pr[B],$$

In the first case the condition  $B$  increases the probability of  $A$ , and vice versa; the events may be called *positively correlated*. In the second case one event, being used as a condition, decreases the probability of the second event, so people say that the events are *negatively correlated*.

If the events are not independent, then adding one as a condition changes the probability of the other one by some factor. Bayes' theorem says that this factor is the same: condition  $A$  increases/decreases the probability of  $B$  by the same factor as  $B$  increases/decreases the probability of  $A$ .

There is an old saying that “correlation is not causation”. Surely being ill and seeing a doctor are positive correlated events (there are more chances the you will go to a doctor if you are ill), but this does not mean the going to a doctor makes you ill. If you read in the Internet that (say) “scientists have shown that drinking [some beverage] increases your life expectancy”, it may well happen that scientists did really collected the data and found that people who drink this beverage live longer in average – but just because old people for some reason like this beverage very much. A bit more extreme example: the number of anniversary parties is strongly correlated with lifespan, but still you should not worry about your health if you forgot to celebrate your birthday.

Let us now look at some questions about independence, just to check that this notion is well understood.

- Problem 274**
1. Two events  $A$  and  $B$  are independent,  $\Pr[A] = p$ ,  $\Pr[B] = q$ . What is the probability of the event “ $A$  and  $B$ ”?
  2. (Continued) What is the probability of the event “ $A$  or  $B$ ”?
  3. (Continued) What is the probability of the event “neither  $A$  nor  $B$ ”?
  4. Two events  $A$  and  $B$  are independent. Does this imply that the events “not  $A$ ” and  $B$  are independent?
  5. (Continued) Does this imply that the events “not  $A$ ” and “not  $B$ ” are independent?
  6. Events  $A$  and  $C$  are independent. Events  $B$  and  $C$  are independent. Does this imply that events “ $A$  and  $B$ ” and  $C$  are independent?

Try it: [Coursera](#) ↗.

Here are some hints (but please try to solve these problems first!) For Problem 4 one may note that the probability of the event “not  $A$ ” (with or without condition) is 1 minus the probability of the event  $A$ . And if the probability of  $B$  is zero, then it is independent with any other event. Problem 5 can be solved by referring to problem 4 twice. Finally, for Problem 6 the probability space with two coins can be used. One may take  $A$  = “head on the first coin”,  $B$  = “head on the second coin” and  $C$  = “the same bit for both coins”. Here all events have probability  $1/2$ , any pair is independent, but still the event “ $A$  and  $B$ ” has different probabilities with and without condition  $C$  ( $1/2$  and  $1/4$ ).

Independence is a tricky notion.

- Stop and Think!** Alice tosses a fair coin. Bob rolls a symmetric dice. Are the events “head” (on Alice’s coin) and “six” (on Bob’s dice) independent?

The tricky part is that we do not say what are the probability spaces explicitly, and let the reader make an educated guess. If Alice were alone, this guess would be that her space has two (equiprobable) outcomes 0 and 1. The same for Bob (six equiprobable outcomes). Then the answer to the question would be “the question makes no sense, since these two events are defined in different spaces”.

However, if we consider the coin tossing and dice rolling as one experiment, this experiment has 12 outcomes, and our experience says that these outcomes will appear equally often in practice. (Unless the coin and dice are too hi-tech and include embedded electronics.) Then the events “head” and “six” are defined on the same space and they indeed are independent. This would be a

natural interpretation of this question if it is asked by a normal person, not a pedantic mathematics professor.

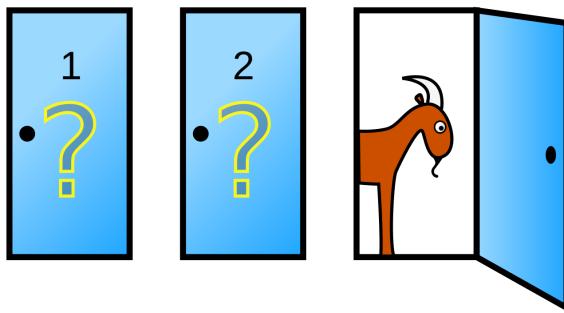
## 11.4 Monty Hall Paradox

In this section we consider a well known question that was a topic of many heated debates. It is called “Monty Hall Paradox” (named after the host of a TV show where a similar situation appeared), and illustrates many of the topics we have discussed.

### 11.4.1 The Show

The original formulation of the question that made it popular, was:

Suppose you’re on a game show, and you’re given the choice of three doors: Behind one door is a car; behind the others, goats. You pick a door, say No. 1, and the host, who knows what’s behind the doors, opens another door, say No. 3, which has a goat. He then says to you, “Do you want to pick door No. 2?” Is it to your advantage to switch your choice?



(See [https://en.wikipedia.org/wiki/Monty\\_Hall\\_problem](https://en.wikipedia.org/wiki/Monty_Hall_problem); the illustration is taken from the same article.)

**Stop and Think!** Before reading further, read carefully the question. Do you understand completely the rules of the game? Is there enough information to answer the question, and if yes, what is the answer?

Let us describe the process step by step, trying not to miss any details.

- The show involves Mme Host, Mr. Guest, and three identical doors.
- The prize is behind one of the doors, chosen randomly by Host; Guest does not know where the prize is (of course).
- Guest may choose any door he wants.
- Host selects one of the two other doors that do not have a prize behind it. If Guest makes an incorrect guess, there is only one other door without prize, so Host has no choice. If Guest makes a correct guess, then both other doors may be chosen by Host; to make the choice, she tosses a coin. The door is opened (and there is no prize behind).
- Guest may either stick to his choice, or change it. After he makes his decision, the door is opened and he either wins (gets the prize) if the final guess is correct or loses if not.

Now we give three arguments that lead to three different answers.

*To keep the guess:* Opening the door without prize does not provide any new information: we know in advance that Host will choose the door without prize. So there are no reasons to change the door, and Guest should stick to his original choice.

*To make a new random choice between two remaining doors:* After the door without prize is opened, there are two doors that can hide the prize, so we essentially have a problem with two doors, and the only thing Guest can do is to make a random guess.

*To change the guess:* The first door has a prize behind it with probability  $1/3$ , so the other (unopened) door has a prize with probability  $2/3$ , so Guest should change the door to get (twice) better chances.

**Stop and Think!** Now, after the detailed description of the setting and three different arguments, which one looks convincing to you?

What can be said against these arguments?

*First argument.* Yes, it is true that the opening of the other door does not prove anything, the prize still can be behind two different doors. Still we get some new information: we know the door with no prize, and this information can be valuable, even if indecisive.

*Second argument.* First, the natural idea that one should simulate the random process to predict its result, is *not* true. If an asymmetric coin has 70% probability of heads and 30% probability of tails, we should not imitate this coin making the guesses. We will get a better result if we always say “head”. Second, it is quite unclear why the probabilities to see the car behind both doors are the same.

**Problem 275** Compute the probability of success if we make a random 70 : 30 choice trying to guess the outcome of a 70 : 30 coin.

*Third argument.* The probability  $1/3$  to make a correct guess was computed for the first experiment, without opening any doors. Now we make a new experiment — the door is opened, but Guest keeps the guess. May be the probability in this new experiment is different?

**Stop and Think!** After hearing all this arguments, what is your opinion? If you have decided what is the correct answer, can you explain what are the errors in these arguments?

#### 11.4.2 Computer simulation

One way to think about this problem is to write a computer simulation. This is useful for two reasons. First, writing a program, we need to understand all the details of the problem setting and may notice something important. Second, we can look at the results and empirically check our hypotheses.

Here is the program that simulates the show.

```
from random import randint, seed

seed(96)

def initial_choice():
    return randint(1, 3)

def keep(choice, opened):
    return choice

def randomize(choice, opened):
    rooms = [1, 2, 3]
    rooms.remove(opened)
    return rooms[randint(0, 1)]

def change(choice, opened):
    return 6 - choice - opened

def success_frequency(strategy, num_experiments):
    num_wins = 0
    for _ in range(num_experiments):
```

```

car = randint(1, 3)
choice = initial_choice()

if choice != car:
    opened = 6 - choice - car # remaining
else: # car = choice
    rooms = [1, 2, 3]
    rooms.remove(car)
    opened = rooms[randint(0, 1)]

new_choice = strategy(choice, opened)

if new_choice == car:
    num_wins += 1

return num_wins / num_experiments

print(success_frequency(keep, 100000))
print(success_frequency(randomize, 100000))
print(success_frequency(change, 100000))

```

The simulation happens (`num_experiment` times) in the function `success_frequency`. First, Host chooses a door with a car (variable `car`; doors are numbered 1,2,3). Then, Guest's choice is simulated by function `initial_choice` that chooses a random door. (This function is fixed, since it is more or less clear that at that stage Guest cannot do anything better.)

The next conditional operator simulates the behavior of Host. If Guest's choice was incorrect, Host opens the third door (technical hack: given two different numbers  $u, v$  between 1 and 3, we may compute the third number as  $6 - u - v$ , since the sum of three numbers is 6). If Guest's choice was correct, Host has a choice between two rooms. We compute the list of these rooms by deleting the third room from the list `[1, 2, 3]`, and then choose a random one between two remaining rooms.

Then we simulate Guest's second choice calling the function `strategy`. This function is given two arguments: the first choice made by Guest (since Guest needs to remember the choice made) and the number of the opened door. It returns the new choice (may be the same as the previous one).

Finally we see whether the (second) guess is correct, and count the experiments where it was correct. The frequency of successes is then returned.

Three possible strategies are described in the program (`keep`, `randomize`, and `change`). Then we simulate all three (10000 times each) and print the frequencies of successes.

We can show you the output of this program, and you will see which argument was correct. But it looks like cheating (explaining the known answer by some plausible reasoning is easier than finding the answer when it is unknown). So we give in the next section some explanations (and we are not cheating here: they were prepared before running the simulation). If they does not look convincing to you, then run the simulation as the last resort.

#### 11.4.3 Our Position

Now we revisit all three arguments and explain which one is correct. In our postmodern politically correct world one probably should say instead that “we state our position” and modestly accept that other people may have different opinions that also have the same right to exist, so we named this section “Our Position”, not “Our Solution” (We are kidding: of course, like in many other cases, this question *does* have one correct answer — though it depends on subtle details in the problem setting.)

To see the situation clearly, let us think not about a unique experiment but about many

repetitions — as if the show is broadcast every day. First, we imagine that Guest always chooses the first (“keep”) strategy, and find the fraction of days when Guest wins. Then we do the same for two other strategies, and see which one is better.

“*Keep strategy*”. Let us imagine a series of  $N$  shows. The host puts the car behind a random door, so there are  $N/3$  shows when the first door is chosen,  $N/3$  shows when the second door is chosen, etc. (Of course, we do not expect exactly  $N/3$ , as usual, but for brevity we do not say it every time.) In each of the groups the guest can make one of three choice, dividing each group into three equal subgroup of size  $N/9$ . In each group one of the subgroups will let Guest win (the number is guessed correctly), together these three subgroups give  $N/3$  cases when Guest wins. Note that the rest of the show does not matter if Guest does not change the door. So the probability to win the show using the first strategy is  $1/3$ .

“*Change strategy*”. Here we can use the preceding analysis. The outcome of the show depends on the random choices made by players and on the strategy used by Guest (note that Host’s strategy is fixed). If all the random choices are the same, but Guest uses *Change* instead of *Keep*, the outcome of the show will reverse. The *Change* strategy will win if and only if the *Keep* strategy loses, so the *Change* strategy wins in  $2/3$  of all cases, the probability of success is  $2/3$ .

“*Randomize strategy*” can be analyzed as follows. Let us pause all  $N$  shows before the last choice of Guest. In each show there are two doors (excluding the one that is already opened), and the prize is behind one of them. The part of the show after this point is simple: Guest chooses randomly one of two doors, and gets the prize if he chooses the correct one. Since his choice is made randomly (with equal probabilities and independently from all previous choices, to be pedantic), in half of the cases Guest is lucky. The probability of success is therefore  $1/2$ .

Now the subtle point: why *the original setting of a problem*, without the clarifications we provided, is not enough to decide which strategy is optimal. Let us read the original setting again:

Suppose you’re on a game show, and you’re given the choice of three doors: Behind one door is a car; behind the others, goats. You pick a door, say No. 1, and the host, who knows what’s behind the doors, opens another door, say No. 3, which has a goat. He then says to you, “Do you want to pick door No. 2?” Is it to your advantage to switch your choice?

Note that the story is about one specific emission of our show and says nothing about the show in general. Reading this description, we do not know what are the instructions given to Host.

Imagine, for example, that the show organizers want to save the money and give Host the following instructions: “If Guest chooses an incorrect door as his first choice, open this door and give him the goat. Otherwise, if his choice is correct, open one of two other doors (randomly), show him the goat and ask whether he wants to change his mind”. These instructions are consistent with the problem description (all what is said there could well happen if Host follows these modified instructions).

Still, for this behavior of the Host, the *Keep* strategy is obviously the best: Host suggests to change the door only if the first choice is correct.

**Problem 276** What are the winning probabilities of all three strategies in this new setting?

**Problem 277** Change the simulation program to take the new setting into account and check your answer for the preceding problem.

This example shows that complete description of the experiment is needed and shortened statement of a problem may be deceptive. The same can happen for the two tails paradox we discussed earlier.

**Problem 278** Mary has the following instructions: toss a fair coin twice. If the first bit is a tail, say “at least one of the outcomes is a tail”, and if the first bit is a head, say “at least one of the outcomes is a head”. Mary followed the instructions and said “at least one of the outcomes is a tail”. What is the probability that she got two tails?

In the following problem Monty Hall behaves in a strange but perfectly legal way. You may try to use our approach to find a correct answer (or modify the simulation program and guess it).

**Problem 279** Assume that Host has the following instructions:

Put a prize behind a random door. Let Guest choose a door.

(1) If Guest chooses an incorrect door (with no prize), roll a dice (in such a way that Guest does not see it and does not know whether this happened);

(a) with probability  $1/3$  (outcomes 1 and 2) open that door (with no prize behind) and end the show;

(b) with probability  $2/3$  (outcomes 3, 4, 5, 6) open the other door with no prize and ask Guest whether he wants to change the door.

(2) if Guest chooses the correct door (with a prize), open one of the two other doors (making a random choice) and ask Guest whether he wants to change the door.

What is the probability for the guest to get a prize if he uses *Change* strategy? The same question for *Keep* strategy.

Try it: [Coursera ↗](#).



## 12. Random Variables

In the previous chapter, we discussed how to compute probabilities of random events: say, the probability of getting three heads out of seven coin tosses or the probability of winning a car if you switch the door after you see a goat behind the opened door. At the same time, an outcome of many random experiments is not just an event, but a number (a measurement of something). For example, one may want to estimate gambling winnings, or an income of a random person in a group of people, or the number of steps of a randomized algorithm. In this chapter, we discuss random variables, a mathematical model that allows to compute quantitative characteristics of random experiments.

### 12.1 Random Variables and Their Expectations

#### 12.1.1 Random Variables

In the previous chapter, we considered probability of *events* in random experiments. We consider random experiments with a finite number of possible outcomes:  $U = \{u_1, \dots, u_n\}$ . The probabilities  $p_1 = \Pr[u_1], \dots, p_n = \Pr[u_n]$  of these outcomes satisfy two natural restrictions:  $0 \leq p_i \leq 1$  (for all  $i = 1, \dots, n$ ) and  $p_1 + \dots + p_n = 1$ . In a long series of  $N$  experiments, we expect the outcome  $u_i$  to appear about  $p_i N$  times. Then, an *event*  $E$  is just a subset of the set of outcomes:  $E \subseteq U$ . The probability of an event is defined as the sum of probabilities of all outcomes from this event:

$$\Pr[E] = \sum_{u \in E} \Pr[u] = \sum_{u_i \in E} p_i.$$

For example, if a random experiments consists of tossing a coin two times, then the set  $U$  of outcomes is

$$U = \{\text{HH}, \text{HT}, \text{TH}, \text{TT}\}.$$

Assuming that a coin is fair, the probability of each of the four outcomes is equal to  $1/4$ . Then, the probability of an event  $E$  = “exactly one heads” is equal to  $1/2$ . Indeed,  $E = \{\text{HT}, \text{TH}\}$  and hence

$$\Pr[E] = \Pr[\text{HT}] + \Pr[\text{TH}] = \frac{1}{4} + \frac{1}{4} = \frac{1}{2}.$$

In programming terms, whether an outcome  $u_i$  of the random experiment belongs to a set  $E$  is a *Boolean* questions (that is, a yes or no question). However, in many cases, we are interested

in numerical characteristics of random experiments. For example, if we toss a coin several times, we might want to know the *number* of heads we get. In a gambling game, rather than checking whether you won, you usually would like to know *how much* you won. *Random variable* is a mathematical model for studying numerical characteristics of random experiments.

Formally, a random variable is a function  $\alpha: U \rightarrow \mathbb{R}$  that assigns a numerical value to every outcome of a random experiment. Note that there is nothing random in a random variable  $\alpha$  itself. However, when we see a random outcome  $u$  of the experiment, we can compute the value  $\alpha(u)$ . In the following, we use Greek letters to denote random variables. This makes it easier to distinguish them from objects of other types: outcomes, probabilities, events.

**Stop and Think!** To define a random variable, we assign a number to each outcome of the experiment. Does this look familiar? Review the previous chapter to see whether we have done this before.

It is indeed not something new, we have done this before! For example, when we throw a dice, formally speaking the outcome is the side of the dice that turns out to be on the top:

$$U = \{\square, \square, \square, \square, \square, \square\}.$$

But for our convenience, we introduced numbers on the sides of the dice: this makes it easier to name a particular side. Thus, we used a random variable  $\alpha: U \rightarrow \{1, 2, 3, 4, 5, 6\}$ .

|             |           |           |           |           |           |           |           |
|-------------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
| $u$         | $\square$ |
| $\Pr[u]$    | 1/6       | 1/6       | 1/6       | 1/6       | 1/6       | 1/6       | 1/6       |
| $\alpha(u)$ | 1         | 2         | 3         | 4         | 5         | 6         |           |

Consider another example: we toss a coin three times. The set  $U$  of all outcomes of the experiment is the following:

$$U = \{\text{HHH}, \text{HHT}, \text{HTH}, \text{HTT}, \text{THH}, \text{THT}, \text{TTH}, \text{TTT}\}.$$

We assume that all eight events are equiprobable: the probability of each of them is equal to 1/8. Now, define the following two random variables:  $\alpha$  is the number of heads;  $\beta$  is equal to 1 if the results of the three tosses are the same, and is equal to 0 otherwise.

|             |     |     |     |     |     |     |     |     |
|-------------|-----|-----|-----|-----|-----|-----|-----|-----|
| $u$         | HHH | HHT | HTH | HTT | THH | THT | TTH | TTT |
| $\Pr[u]$    | 1/8 | 1/8 | 1/8 | 1/8 | 1/8 | 1/8 | 1/8 | 1/8 |
| $\alpha(u)$ | 3   | 2   | 2   | 1   | 2   | 1   | 1   | 0   |
| $\beta(u)$  | 1   | 0   | 0   | 0   | 0   | 0   | 0   | 1   |

One can define events in terms of random variables. For example, “2 |  $\alpha$ ” is a shortcut for an event “the number of heads is even”. Examples:

$$\Pr[2 | \alpha] = \Pr[\{\text{HHT}, \text{HTH}, \text{THH}, \text{TTT}\}] = 1/4,$$

$$\Pr[\alpha = 3] = \Pr[\{\text{TTT}\}] = 1/8,$$

$$\Pr[\beta = 2] = \Pr[\emptyset] = 0,$$

$$\Pr[\beta \geq \alpha] = \Pr[\{\text{TTT}\}] = 1/8.$$

Finally, since  $\alpha(u)$  is a number, one can apply a function  $f: \mathbb{R} \rightarrow \mathbb{R}$  to it. This gives a new random variable  $f(\alpha)$ . For example, let  $f(x) = x^2 - 3$ . Then, the random variable  $f(\alpha): U \rightarrow \mathbb{R}$  assigns the following values to all outcomes.

|             |     |     |     |     |     |     |     |     |
|-------------|-----|-----|-----|-----|-----|-----|-----|-----|
| $u$         | HHH | HHT | HTH | HTT | THH | THT | TTH | TTT |
| $\alpha(u)$ | 3   | 2   | 2   | 1   | 2   | 1   | 1   | 0   |
| $f(\alpha)$ | 6   | 1   | 1   | -2  | 1   | -2  | -2  | -3  |

Instead of  $f(\alpha)$ , we will write just  $\alpha^2 - 3$ .

**Problem 280** A random experiment consists of throwing a dice two times. Which of the following are events and which are random variables?

1. The sum of numbers on both dice.
2. The number in the first throw is greater than the number in the second throw.
3. The difference between numbers on the first throw and the second throw.
4. The difference between numbers in the first and the second throw is positive.
5. The product of numbers on both dice is even.
6. The number on the first dice.

Try it: [Coursera](#).

**Problem 281** Toss a coin two times. Consider a random variable that is equal to the number of ‘heads’ in these throws. What is the probability of the event that this random variable has value 1? Try it: [Coursera](#).

One of the simplest and the most frequently used random variables is an *indicator random variable*. For an event  $A \subseteq U$ , an indicator  $\chi_A : U \rightarrow \{0, 1\}$  is defined as follows:

$$\chi_A(u) = \begin{cases} 1 & \text{if } u \in A, \\ 0 & \text{if } u \notin A. \end{cases}$$

Instead of a formula above, it is common to use the following shortcut:  $\chi_A(u) = [u \in A]$ . In plain English:  $\chi_A(u)$  is equal to 1 for those (and only those)  $u$  that belong to  $A$ .

### 12.1.2 Average

*Average* is the most frequently used number characteristic of a sequence of numbers. You have seen it many times: average salary, average temperature, average height, average cost. To compute it, one sums up the numbers and divides by the number of them.

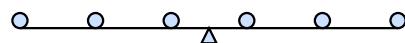
```
def average(numbers):
    return sum(numbers) / len(numbers)

print(average([1, 2, 3, 4, 5, 6]))
print(average([7, 7, 7]))
print(average([9.5, 10.5]))
```

```
3.5
7.0
10.0
```

This code shows, in particular, that the average of six numbers on dice sides is 3.5 (it is not an integer, though all of 1,2,3,4,5,6 are integers).

There is a natural physical interpretation of the average of a sequence of numbers: if we treat the numbers as points on a line, then their average is the *center of mass* of these points. To visualize this, imagine six balls of the same weight lying on a rod at points 1,2,3,4,5,6. To make this construction stable, one puts a stand under this rod at point 3.5.



Python has a built-in method `mean` for computing the average: *mean* and *arithmetic mean* are other names of average.

```
from statistics import mean

print(mean([1, 2, 3, 4, 5, 6]))
print(mean([7, 7, 7]))
print(mean([9.5, 10.5]))
```

```
3.5
7
10.0
```

**Problem 282** HR management of a company has decided to optimize the average performance of employees by applying the following strategy: while there is a person whose efficiency is less than the average efficiency, fire this person. What will be a result of such strategy?

To develop intuition, let us make an experiment. In the code below, there are six employees whose efficiency is 6, 9, 5, 2, 4, 3. The simulation follows the strategy described above: at every iteration, it computes the minimum and the mean of the current list of numbers. If the former is smaller than the latter, the minimum value is removed.

```
from statistics import mean

def improve_average(numbers):
    minimal, average = min(numbers), mean(numbers)
    while minimal < average:
        print(f'The mean of {numbers} is {average:.2f}')
        print(f'Remove {minimal}')
        numbers.remove(minimal)
        minimal, average = min(numbers), mean(numbers)

improve_average([6, 9, 5, 2, 4, 3])
```

```
The mean of [6, 9, 5, 2, 4, 3] is 4.83
Remove 2
The mean of [6, 9, 5, 4, 3] is 5.40
Remove 3
The mean of [6, 9, 5, 4] is 6.00
Remove 4
The mean of [6, 9, 5] is 6.67
Remove 5
The mean of [6, 9] is 7.50
Remove 6
```

The code shows that everybody got fired except for the person with maximum efficiency! This example also reveals that when a person (with minimal efficiency) is fired, the average efficiency increases (as HR management has planned). We encourage you to play around with this code before proceeding further. See what happens when all the numbers are equal and when there are several maximums.

The effect demonstrated in this example is used so frequently in discrete mathematics that we state it as a separate theorem.

**Theorem 12.1.1** Let  $a_1, \dots, a_n$  be a sequence of numbers and let

$$\mu = \frac{a_1 + \dots + a_n}{n},$$

$$m = \min(a_1, \dots, a_n),$$

$$M = \max(a_1, \dots, a_n).$$

be their mean, minimum, and maximum, respectively. Then,

1.  $m \leq \mu \leq M$ ;
2. if not all of the elements of the sequence are equal to each other, then  $m < \mu < M$ .

Applying this theorem to Problem 282, we see that unless everybody works equally well (which is extremely rare), there is always someone working below average! If we fire this person, the average performance will grow. And now there will be a new person who works below average and we have to fire him as well. This way, we will fire everyone except the persons with maximum efficiency. This is perhaps not what we would like to do.

*Proof of Theorem 12.1.1.* Clearly,

$$\begin{aligned} a_1 + \cdots + a_n &\geq m + \cdots + m = nm, \\ a_1 + \cdots + a_n &\leq M + \cdots + M = nM. \end{aligned}$$

Hence,

$$m \leq \frac{a_1 + \cdots + a_n}{n} \leq M.$$

In case not all of  $a_i$ 's are equal to each other, we know that there must be an element larger than  $m$  (otherwise all the elements would be equal to  $m$ ) and an element smaller than  $M$ . Then,

$$\begin{aligned} a_1 + \cdots + a_n &> m + \cdots + m = nm, \\ a_1 + \cdots + a_n &< M + \cdots + M = nM. \end{aligned}$$

■

**Problem 283** We throw a dice many times. What is the average outcome?

This is an ill-posed question: the answer depends on the outcomes of all throws. For example, it can happen that in all throws we see  $\square$ . Then the average is 1. If we see  $\boxdot$  each time, then the average is 6. At the same time, it is intuitively clear that getting the same number many times in a row is a rare event. And it turns out that we can give a good approximation to the average that we will observe.

Assume that we throw a dice  $n$  times, for some large number  $n$ . Then, we expect to see each of the six outcomes in about  $n/6$  throws. The sum of all outcomes is then approximately

$$\frac{n}{6} \cdot 1 + \frac{n}{6} \cdot 2 + \frac{n}{6} \cdot 3 + \frac{n}{6} \cdot 4 + \frac{n}{6} \cdot 5 + \frac{n}{6} \cdot 6 = \frac{n \cdot (1+2+3+4+5+6)}{6} = \frac{21n}{6} = 3.5n.$$

To get the average, we divide by  $n$  and obtain 3.5.

The following code throws a dice 100000 times and computes the average of the outcomes.

```
from random import randint, seed
from statistics import mean

seed(19)
print(mean([randint(1, 6) for _ in range(10 ** 5)]))

3.50433
```

**Problem 284** On a test, three students received the following grades: 78, 72, and 87. What is the average grade? Try it: [Coursera](#).

**Problem 285** Consider two numbers:  $1 \leq a \leq 5$  and  $2 \leq b \leq 7$ . Let  $\mu = (a+b)/2$  be their average. What is the smallest possible value of  $\mu$ ? What is the largest possible value of  $\mu$ ? Try it: [Coursera](#).

**Problem 286** We throw a fair coin many times and write down 0 each time we see heads and 1 each time we see tails. What is the approximate value of the average of all these bits? Try it: [Coursera](#).

**Problem 287** Consider four numbers:  $a, b, c, d$ . Find the value of  $d$ , if the average of  $a, b, c, d$  is 36, whereas the average of  $a, b, c$  is 27. Try it: [Coursera](#).

There is a natural generalization of average called *weighted average*. In this case, we are given  $n$  numbers  $a_1, \dots, a_n$  together with their weights  $w_1, \dots, w_n$ . Then, their weighted average is defined as

$$\frac{w_1 \cdot a_1 + \dots + w_n \cdot a_n}{w_1 + \dots + w_n}.$$

Thus, the average is a special case of the weighted average with all weights equal to 1.

**Problem 288** On a test, four students received a grade of 88, six students receive a grade of 84, and two students received a grade of 97. What is the weighted average of the grades?

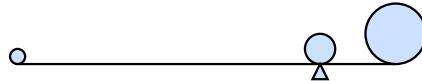
By definition, the answer is

$$\frac{4 \cdot 88 + 6 \cdot 84 + 2 \cdot 97}{4 + 6 + 2} = 87.5.$$

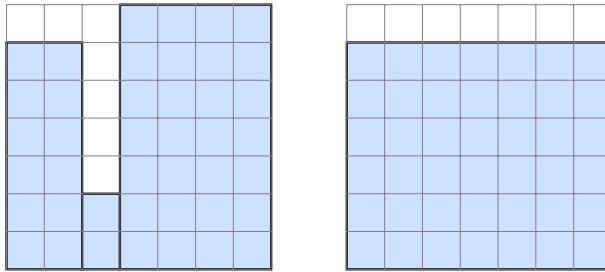
The physical interpretation works well for the weighted case. Consider a sequence of three integers  $a_1 = 6, a_2 = 2, a_3 = 7$  with weights  $w_1 = 2, w_2 = 1, w_3 = 4$ . Their weighted average is

$$\frac{2 \cdot 6 + 1 \cdot 2 + 4 \cdot 7}{2 + 2 + 4} = 6.$$

This is where you place a stand to balance three balls of weights  $w_1 = 2, w_2 = 1, w_3 = 4$  sitting at points  $a_1 = 6, a_2 = 2, a_3 = 7$ .



Finally, there is also a *geometric* interpretation of the weighted average. For the  $i$ -th element, consider a rectangle of size  $w_i \times a_i$ . Draw all these rectangles side by side. The width of the resulting picture is  $w_1 + \dots + w_n$ . The total area is  $w_1 a_1 + \dots + w_n a_n$  and it is equal to the area of a rectangle of size  $(w_1 + \dots + w_n) \times \mu$ , where  $\mu$  is the weighted average.



### 12.1.3 Expectation

Much as the average is the most natural number characteristics of a sequence of numbers, the *expectation* is the most natural number characteristics of a random variable. Informally, it is just a weighted average of all the values of the random variable where one uses the probabilities of the values as weights.

Formally, the *expectation* or *expected value* of a random variable  $\alpha: U \rightarrow \mathbb{R}$  is defined as follows:

$$\mathbb{E}[\alpha] = \sum_{u \in U} \alpha(u) \cdot \Pr[u]. \quad (12.1)$$

That is, it is the sum over all outcomes of the product of the value of this outcome and the probability of this outcome.

To give a familiar example of an expectation, consider a random variable  $\alpha$  that is equal to one of the six digits on a dice. Then,

$$\mathbb{E}[\alpha] = 1 \cdot \frac{1}{6} + 2 \cdot \frac{1}{6} + 3 \cdot \frac{1}{6} + 4 \cdot \frac{1}{6} + 5 \cdot \frac{1}{6} + 6 \cdot \frac{1}{6} = 3.5.$$

In this case, all outcomes are equiprobable and the expectation is equal to the average of the values of  $\alpha$ . Now, assume that a dice is biased:

|          |     |     |     |      |       |        |
|----------|-----|-----|-----|------|-------|--------|
| $u$      | □   | □□  | □□□ | □□□□ | □□□□□ | □□□□□□ |
| $\Pr[u]$ | 0.1 | 0.1 | 0.1 | 0.2  | 0.2   | 0.3    |

The dice is biased towards larger values, hence it is not surprising that the expectation of  $\alpha$  is larger in this case:

$$\mathbb{E}[\alpha] = 1 \cdot 0.1 + 2 \cdot 0.1 + 3 \cdot 0.1 + 4 \cdot 0.2 + 5 \cdot 0.2 + 6 \cdot 0.3 = 4.2.$$

Returning to the unbiased case, the value 3.5 is a reasonable approximation of  $\alpha$  in the following sense: half of the outcomes lead to a number smaller than 3.5, the other half lead to a number larger than 3.5. Generalizing this toy example further, one can prove the following counterpart of Theorem 12.1.1.

**Theorem 12.1.2** Let  $\alpha: U \rightarrow \mathbb{R}$  be random variable. Let  $m$  and  $M$  be the minimum and the maximum values of  $\alpha$ :

$$m = \min_{u \in U} \alpha(u) \quad \text{and} \quad M = \max_{u \in U} \alpha(u).$$

Then,

$$m \leq \mathbb{E}[\alpha] \leq M.$$

*Proof.* Let us prove the first inequality (the second one is proved similarly).

$$\begin{aligned} \mathbb{E}[\alpha] &= \\ &= \sum_{u \in U} \alpha(u) \Pr[u] && (\text{definition (12.1) of expectation}) \\ &\geq \sum_{u \in U} m \Pr[u] && (\alpha(u) \geq m \text{ for all } u) \\ &= m \sum_{u \in U} \Pr[u] && (\text{take } m \text{ out}) \\ &= m. && (\text{sum of probabilities is 1}) \end{aligned}$$

■

Why expectation is actually called expectation? Consider some random experiment with four outcomes  $u_1, u_2, u_3, u_4$  whose probabilities are  $p_1, p_2, p_3, p_4$  respectively. Consider a random variable  $f$  with values  $a_1, a_2, a_3, a_4$  respectively. Suppose we repeat the random experiment  $n$  times for large  $n$ . What is the average value of  $f$  on the outcomes approximately?

**Stop and Think!** Can you guess what is the answer to this question?

We can repeat the same argument as with the dice in the previous section. In  $n$  experiments the value  $a_i$  of the random variable will occur approximately  $p_i n$  times. To compute the average we need to sum up the values in all outcomes and divide by the number of outcomes  $n$ . The value  $a_1$  will appear in the sum  $p_1 n$  times, the value  $a_2$  will appear  $p_2 n$  times and so on. Overall, the average is approximately equal to

$$\frac{a_1 p_1 n + a_2 p_2 n + a_3 p_3 n + a_4 p_4 n}{n} = a_1 p_1 + a_2 p_2 + a_3 p_3 + a_4 p_4.$$

Observe, that we obtain exactly the expectation! The intuition behind the expectation is that it is the approximation to what we would expect as an average outcome of an experiment repeated many times.

In this book, we consider random variables over *finite* sets of outcomes  $U$ . The expectation of such a random variable is always a number. However if  $U$  is infinite, the expected value might also turn to be infinite. To give an example, consider the following random variable:  $\Pr[\alpha = 2^k] = 2^{-k}$ , for every positive integer  $k$ . Then,

$$\mathbb{E}[\alpha] = \sum_{k=1}^{\infty} 2^k \cdot 2^{-k} = \sum_{k=1}^{\infty} 1 = \infty.$$

**Problem 289** Consider a random variable with outcomes 0 and 1 having probabilities  $1/3$  and  $2/3$ , respectively. What is the expected value of this random variable? Try it: [Coursera ↗](#).

**Problem 290** Consider a random variable with outcomes 1, 3, and 4 having probabilities  $1/4$ ,  $1/2$ , and  $1/4$  respectively. What is the expected value of this random variable? Try it: [Coursera ↗](#).

In many cases, the following form of the expectation allows to compute it more conveniently. Here, instead of summing up over all outcomes, we sum over all values of  $\alpha$ .

**Theorem 12.1.3** For any random variable  $\alpha: U \rightarrow \mathbb{R}$ ,

$$\mathbb{E}[\alpha] = \sum_{a \in \mathbb{R}} a \Pr[\alpha = a]. \quad (12.2)$$

*Proof.*

$$\begin{aligned} \mathbb{E}[\alpha] &= \\ &= \sum_{u \in U} \alpha(u) \Pr[u] && \text{(definition (12.1) of expectation)} \\ &= \sum_{a \in \mathbb{R}} \left( \sum_{u: \alpha(u)=a} \alpha(u) \Pr[u] \right) && \text{(group by common values)} \\ &= \sum_{a \in \mathbb{R}} \left( \sum_{u: \alpha(u)=a} a \Pr[u] \right) && \text{(simplify)} \\ &= \sum_{a \in \mathbb{R}} \left( a \cdot \sum_{u: \alpha(u)=a} \Pr[u] \right) \\ &= \sum_{a \in \mathbb{R}} a \Pr[\alpha = a]. \end{aligned}$$

■

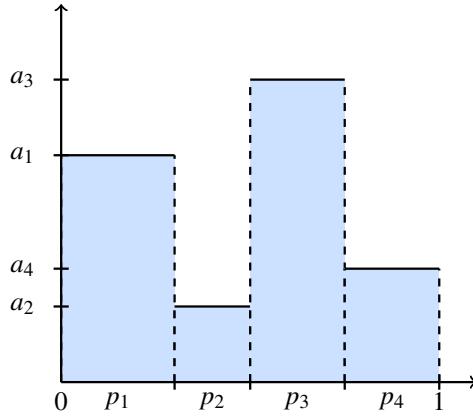
Since expectation is the weighted average, one can adapt the geometric interpretation considered in the previous section. To give an example, consider the following random variable  $\alpha$ :

| $u$         | $u_1$ | $u_2$ | $u_3$ | $u_4$ |
|-------------|-------|-------|-------|-------|
| $\Pr[u]$    | $p_1$ | $p_2$ | $p_3$ | $p_4$ |
| $\alpha(u)$ | $a_1$ | $a_2$ | $a_3$ | $a_4$ |

By definition,

$$\mathbb{E}[\alpha] = a_1 p_1 + a_2 p_2 + a_3 p_3 + a_4 p_4.$$

Now, we know that  $p_1 + p_2 + p_3 + p_4 = 1$ . Thus,  $\mathbb{E}[\alpha]$  is just the total area of the rectangles of size  $p_1 \times a_1$ ,  $p_2 \times a_2$ ,  $p_3 \times a_3$ , and  $p_4 \times a_4$ . See Figure 12.1.3.

Figure 12.1:  $E[\alpha]$  is the area of colored region.

One particularly useful property of the expectation is: one a random variable is multiplied by a constant, its expectation is multiplied by the same constant. Going back to our working example, consider a random variable  $\alpha$  that is equal to the number on a fair dice and let  $\beta = 2\alpha$ . That is,  $\beta$  gets the following values:

| $u$         | 1   | 2   | 3   | 4   | 5   | 6   |
|-------------|-----|-----|-----|-----|-----|-----|
| $\Pr[u]$    | 1/6 | 1/6 | 1/6 | 1/6 | 1/6 | 1/6 |
| $\alpha(u)$ | 1   | 2   | 3   | 4   | 5   | 6   |
| $\beta(u)$  | 2   | 4   | 6   | 8   | 10  | 12  |

Then,  $E[\beta] = 2E[\alpha] = 7$ . You may want to check this by plugging all the values of  $\beta$  into the definition of the expectation. In general, this is proved as follows.

**Theorem 12.1.4** Let  $c$  be a constant and  $\alpha$  be a random variable. Then,

$$E[c\alpha] = cE[\alpha].$$

*Proof.*

$$\begin{aligned}
 E[c\alpha] &= && \text{(definition (12.1) of expectation)} \\
 &= \sum_{u \in U} \Pr[u](c\alpha)(u) && \text{(definition of } c\alpha\text{)} \\
 &= \sum_{u \in U} \Pr[u]c\alpha(u) && \text{(take } c\text{ out)} \\
 &= c \sum_{u \in U} \Pr[u]\alpha(u) && \text{(definition (12.1) of expectation)} \\
 &= cE[\alpha].
 \end{aligned}$$

■

**Problem 291** Let  $A \subseteq U$  be an event of probability  $p$  (that is,  $\Pr[A] = p$ ). Find  $E[\chi_A]$  (where  $\chi_A$  is an indicator random variable:  $\chi_A(u) = [u \in A]$ ).

It turns out that

$$E[\chi_A] = \Pr[A]. \tag{12.3}$$

Let us prove it.

$$\begin{aligned}
 E[\chi_A] &= && \text{(definition of expectation)} \\
 &= \sum_{u \in U} \alpha(u) \Pr[u] && \text{(split sum into two parts)} \\
 &= \sum_{u \in A} 1 \cdot \Pr[u] + \sum_{u \notin A} 0 \cdot \Pr[u] && \text{(simplify)} \\
 &= \sum_{u \in A} \Pr[u] && \text{(definition of } \Pr[A]\text{)} \\
 &= \Pr[A] = p.
 \end{aligned}$$

## 12.2 Linearity of Expectation

**Problem 292** Toss a coin three times. Let a random variable  $\alpha$  be equal to the number of  $\oplus$ . Find  $E[\alpha]$ .

Let us do this step by step. The problem statement does not explicitly define the set of outcomes  $U$ . Since we are tossing a coin three times, we implicitly assume the following probability space.

| $u$         | $\oplus\oplus\oplus$ | $\oplus\oplus\ominus$ | $\oplus\ominus\oplus$ | $\oplus\ominus\ominus$ | $\ominus\oplus\oplus$ | $\ominus\oplus\ominus$ | $\ominus\ominus\oplus$ | $\ominus\ominus\ominus$ |
|-------------|----------------------|-----------------------|-----------------------|------------------------|-----------------------|------------------------|------------------------|-------------------------|
| $\Pr[u]$    | 1/8                  | 1/8                   | 1/8                   | 1/8                    | 1/8                   | 1/8                    | 1/8                    | 1/8                     |
| $\alpha(u)$ | 3                    | 2                     | 2                     | 1                      | 2                     | 1                      | 1                      | 0                       |

With this table at hand, it is straightforward to compute  $E[\alpha]$ :

$$E[\alpha] = 3 \cdot \frac{1}{8} + 2 \cdot \frac{1}{8} + 2 \cdot \frac{1}{8} + 1 \cdot \frac{1}{8} + 2 \cdot \frac{1}{8} + 1 \cdot \frac{1}{8} + 1 \cdot \frac{1}{8} + 0 \cdot \frac{1}{8} = \frac{12}{8} = 1.5.$$

**Problem 293** What is the expectation of the number of  $\oplus$  if we toss a coin ten times?

**Stop and Think!** Computing the expectation by hand would be not so easy as the set of outcomes contains  $2^{10} = 1024$  elements. Still, could you guess the answer?

Below, we prove a property of the expectation, called *linearity*, that allows to solve problems like this without any calculations.

**Theorem 12.2.1 – linearity of expectation.** For any two random variables  $\alpha, \beta : U \rightarrow \mathbb{R}$ ,

$$E[\alpha + \beta] = E[\alpha] + E[\beta].$$

*Proof.*

$$\begin{aligned}
 E[\alpha + \beta] &= \\
 &= \sum_{u \in U} (\alpha + \beta)(u) \Pr[u] = \\
 &= \sum_{u \in U} (\alpha(u) + \beta(u)) \Pr[u] = \\
 &= \sum_{u \in U} \alpha(u) \Pr[u] + \sum_{u \in U} \beta(u) \Pr[u] = \\
 &= E[\alpha] + E[\beta].
 \end{aligned}$$

■

Let us return to our toy example to show how the linearity of expectation helps in calculations. Let  $\alpha$  be equal to the number of  $\oplus$  among three coin tosses. Here is a useful and frequently used trick.

Represent  $\alpha$  as a sum of indicators:  $\alpha = \alpha_1 + \alpha_2 + \alpha_3$ , where  $\alpha_i$  is a 0/1-random variable representing the number of  $(\text{H})$  in the  $i$ -th toss (for  $\alpha = 1, 2, 3$ ).

More formally, we consider the following random variables (over the same probability space):

| $u$           | $\text{HHH}$ | $\text{HHT}$ | $\text{HTH}$ | $\text{HTT}$ | $\text{TTH}$ | $\text{THT}$ | $\text{TTH}$ | $\text{TTT}$ |
|---------------|--------------|--------------|--------------|--------------|--------------|--------------|--------------|--------------|
| $\Pr[u]$      | 1/8          | 1/8          | 1/8          | 1/8          | 1/8          | 1/8          | 1/8          | 1/8          |
| $\alpha(u)$   | 3            | 2            | 2            | 1            | 2            | 1            | 1            | 0            |
| $\alpha_1(u)$ | 1            | 1            | 1            | 1            | 0            | 0            | 0            | 0            |
| $\alpha_2(u)$ | 1            | 1            | 0            | 0            | 1            | 1            | 0            | 0            |
| $\alpha_3(u)$ | 1            | 0            | 1            | 0            | 1            | 0            | 1            | 0            |

What is good about  $\alpha_1, \alpha_2, \alpha_3$  is that it is easy to compute their expectation: due to (12.3),

$$\mathbb{E}[\alpha_1] = \Pr[\text{the first toss is } (\text{H})] = 1/2.$$

For the same reason,  $\mathbb{E}[\alpha_2] = \mathbb{E}[\alpha_3] = 1/2$ . Thus, by the linearity of expectation (Theorem 12.2.1),

$$\mathbb{E}[\alpha] = \mathbb{E}[\alpha_1 + \alpha_2 + \alpha_3] = \mathbb{E}[\alpha_1] + \mathbb{E}[\alpha_2] + \mathbb{E}[\alpha_3] = 1.5.$$

Moreover, this argument generalizes easily to any number of coins tosses. For example, the answer for Problem 293 is 5.

**Problem 294** Throw a dice two times. What is the expected value of the sum of two numbers of the throws? Try it: [Coursera](#).

In this case, our random variable  $\alpha$  is equal to the sum  $\alpha_1 + \alpha_2$  of two random variables  $\alpha_1, \alpha_2$  that are not indicators. Still, we know that  $\mathbb{E}[\alpha_1] = \mathbb{E}[\alpha_2] = 3.5$  and hence  $\mathbb{E}[\alpha] = 7$ .

**Problem 295** Toss a coin four times. What is the expected number of  $(\text{T})$ ? Try it: [Coursera](#).

**Problem 296** Throw a dice four times. What is the expected value of  $\square$ ? Try it: [Coursera](#).

**Problem 297** Bob invited thirty friends to his party. From the previous experience, Bob knows that each of his friends will show up to the party with probability  $2/5$ . What is the expected number of guests at Bob's party? Try it: [Coursera](#).

### 12.2.1 Birthday Problem

In this section, we discuss the following famous Birthday Problem.

**Problem 298** There is a group of 35 people. Assuming that each of them has a birthday on a random day, what is the probability that there are two of them having the same birthday? Try it: [Coursera](#).

**Problem 299** In the previous setting, what is the expected value of the number of pairs of people having the same birthday? Try it: [Coursera](#).

**Problem 300** How large should  $n$  be so that for  $n$  randomly selected persons the probability that there are two people with the same birthday is at least  $1/2$ ? Try it: [Coursera](#).

**Problem 301** How large should  $n$  be so that for  $n$  randomly selected persons the expected number of pairs with the same birthday is at least one? Try it: [Coursera](#).

Problem 300 is also known as the Birthday Paradox, since (as for most paradoxes) the answer is counterintuitive: a wild guess would be that  $n$  should be as large as  $365/2 = 182.5$  to achieve a probability at least  $1/2$ . Below, we will see that the true answer is much smaller!

Let us start with a simulation. Denote by  $p(n)$  the probability that among  $n$  randomly selected persons there are two with the same birthday. Instead of computing this probability directly,

we compute the probability  $q(n)$  that among  $n$  randomly selected people no two share the same birthday. Clearly,  $p(n) = 1 - q(n)$ . One can compute  $q(n)$  as follows:

$$q(n) = \frac{365}{365} \cdot \frac{364}{365} \cdot \frac{363}{365} \cdots \frac{(365-n+1)}{365}.$$

Indeed, the first person may have a birthday on any of 365 days, the second one may have a birthday on all days except for the birthday of the first person and so on. Computing  $q(n)$  by hand is not that easy. Instead, let us compute  $q(n)$  using Python.

The code below computes the value of  $p(n)$  for all  $n = 1, 2, \dots, 365$ . It then plots the function  $p(n)$ .

```
from itertools import accumulate
import matplotlib.pyplot as plt
from operator import mul

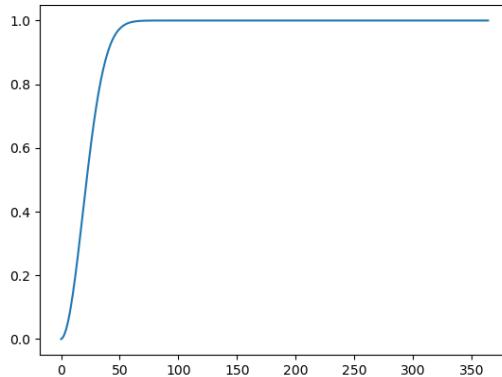
num_days = 365

factors = [1 - i / num_days for i in range(num_days)]
prob_no_collision = list(accumulate(factors, mul))
prob_collision = [1 - p for p in prob_no_collision]

plt.plot(prob_collision)
plt.savefig('birthdays.png')

for n in (22, 23, 35, 100):
    print(f'The probability for {n} people '
          f'is {prob_collision[n - 1]}')
```

```
The probability for 22 people is 0.4756953076625502
The probability for 23 people is 0.5072972343239854
The probability for 35 people is 0.8143832388747152
The probability for 100 people is 0.9999996927510721
```



As we see, already for  $n = 23$ ,  $p(n) > 1/2$  (hence, 23 is the answer for Problem 300). Also,  $p(35) \approx 0.814$  (and this is the answer for Problem 298). Moreover, for  $n > 50$ ,  $p(n)$  is close to 1. Intuitively, the effect is explained as follows: among  $n$  people, there are  $\binom{n}{2}$  pairs of people. For example, there are  $\binom{23}{2} = 253$  pairs among 23 people. We use this observation below, for solving Problems 299 and 301.

Let  $\chi_n$  be a random variable equal to the number of pairs with the same birthday. We use a standard trick for estimating  $E[\chi_n]$ : break  $\chi_n$  into a sum of indicators. Namely,

$$\chi_n = \sum_{1 \leq i < j \leq n} \chi_{ij},$$

where  $\chi_{ij}$  is the indicator of the following event: “persons  $i$  and  $j$  have the same birthday”. Since  $\chi_{ij}$  is an indicator,

$$E[\chi_{ij}] = \Pr[\text{persons } i \text{ and } j \text{ have the same birthday}]$$

(recall (12.3)). Now, this probability is equal to  $1/365^2$ : among  $365^2$  possibilities of two birthdays, there are 365 cases when the two birthdays coincide. Then,

$$\begin{aligned} E[\chi] &= E\left[\sum_{1 \leq i < j \leq n} \chi_{ij}\right] && \text{(linearity of expectation)} \\ &= \sum_{1 \leq i < j \leq n} E[\chi_{ij}] \\ &= \sum_{1 \leq i < j \leq n} \frac{1}{365} \\ &= \binom{n}{2} \cdot \frac{1}{365} \\ &= \frac{n(n-1)}{730}. \end{aligned}$$

Thus, among 35 people we expect  $\frac{35 \cdot 34}{730} \approx 1.63$  people with the same birthday (this is the answer for Problem 299). The smallest value of  $n$  when  $n(n-1)/730 \geq 1$  is 27 (this answers Problem 301).

**Problem 302** Suppose we toss a coin 20 times in a row. We are considering how many times it happens that we have an outcome ‘heads’ and the next outcome is ‘tails’. This is a random variable since it depends on outcome of our tosses. What is the expectation of this random variable? Try it: [Coursera ↗](#).

## 12.3 Expectation is Not All

Recall a standard dice random variable  $\alpha$ :

|             |     |     |     |     |     |     |
|-------------|-----|-----|-----|-----|-----|-----|
| $u$         | ⚀   | ⚁   | ⚂   | ⚃   | ⚄   | ⚅   |
| $\Pr[u]$    | 1/6 | 1/6 | 1/6 | 1/6 | 1/6 | 1/6 |
| $\alpha(u)$ | 1   | 2   | 3   | 4   | 5   | 6   |

Now, consider another random variable  $\beta$ :

| $u$        | $a$   | $b$ |
|------------|-------|-----|
| $\Pr[u]$   | 1/3   | 2/3 |
| $\beta(u)$ | -45.5 | 28  |

Compute the expectation of  $\beta$ :

$$E[\beta] = -45.5 \cdot 1/3 + 28 \cdot 2/3 = 3.5.$$

Thus,  $\alpha$  and  $\beta$  have the same expectation. This toy example shows that the expectation, despite of being the most important number characteristic of a random variable, still loses a lot of information about it: it tells us nothing about the number of outcomes, their probabilities, and the maximum value of the random variable.

Now, consider a more surprising example. Alice and Bob play the following game. Each of them has fair, but unconventional dice. The Numbers on the sides of Alice's dice are

$$2, 2, 2, 2, 3, 3,$$

whereas the numbers on the sides of Bob's dice are

$$1, 1, 1, 1, 6, 6.$$

They throw their dices. The one with the larger number on the dice wins.

**Problem 303** If Alice and Bob play the game many times, who is likely to win more often?

Let us compute the expectations of the corresponding two random variables. Alice gets 2 with probability  $2/3$  and gets 3 with probability  $1/3$ . Her expectation is

$$2 \cdot \frac{2}{3} + 3 \cdot \frac{1}{3} = \frac{7}{3}.$$

Bob gets 1 with probability  $2/3$  and gets 6 with probability  $1/3$ . His expected value is

$$1 \cdot \frac{2}{3} + 6 \cdot \frac{1}{3} = \frac{8}{3}.$$

Therefore, Bob has better expected value.

**Stop and Think!** Does it mean that he wins more often?

Let us compute the probabilities for each player to win directly. The result of the game depends only on the outcome of Bob: indeed, if he throws 1, he loses, if he throws 6, he wins. Bob throws 1 with probability  $2/3$ . Thus, Bob loses with probability  $2/3$ . Bob loses more often despite having larger expected value.

The following code snippet simulates the game 10000 times and shows that, indeed, Alice wins in about  $2/3$  of all games.

```
from random import randint, seed

seed(27)

dice1 = [2, 2, 2, 2, 3, 3]
dice2 = [1, 1, 1, 1, 6, 6]

num_rounds = 10 ** 5
num_dice1_wins = 0

for _ in range(num_rounds):
    dice1_result = dice1[randint(0, 5)]
    dice2_result = dice2[randint(0, 5)]

    if dice1_result > dice2_result:
        num_dice1_wins += 1

print(f'Out of {num_rounds} throws, dice1 won {num_dice1_wins} times')

Out of 100000 throws, dice1 won 66868 times
```

Why the larger expected value does not help Bob to win? Note that when Bob wins, he wins by *far*: he gets 6 against 2 or 3. But when Bob loses, he loses just slightly: he gets 1 against 2 or 3. This difference affects expected value, but it does not affect the game: it does not matter if your outcome is substantially larger.

## 12.4 Markov's Inequality

### 12.4.1 From Expectation to Probability

**Problem 304** A lottery ticket costs \$10. A 40% of the lottery budget goes to prizes. Show that the chances to win \$500 or more are less than 1%.

Assume the contrary: the probability to win at least \$500 is at least 0.01. Let us denote the number of tickets sold by  $n$ . Then the budget of the lottery is  $10n$  dollars. The amount spent on the prizes is  $10n \cdot 0.4 = 4n$  dollars. By our assumption, at least  $\frac{n}{100}$  tickets win at least \$500. In total, these tickets win  $\frac{n}{100} \cdot 500 = 5n$  dollars. But this amount exceeds the budget  $4n$  dollars spent on the prizes! We arrive at the contradiction, hence our assumption was wrong.

**Problem 305** An article on the internet claims that 10% of citizens of a country earn at least 15 times more money than the average income in this country. Can it be the case? Try it: [Coursera](#).

### 12.4.2 Markov's Inequality

The argument we just used to solve the problem is a special case of Markov's Inequality. The inequality upper bounds the probability that a *non-negative* random variable gets a large value.



Figure 12.2: Andrey Andreyevich Markov (1856–1922).

**Theorem 12.4.1 – Markov's Inequality.** For any *non-negative* random variable  $\alpha: U \rightarrow \mathbb{R}$  (that is,  $\alpha(u) \geq 0$  for all  $u \in U$ ) and any constant  $c > 0$ ,

$$\Pr[\alpha \geq c] \leq \frac{\mathbb{E}[\alpha]}{c}.$$

*Proof.*

$$\begin{aligned} \mathbb{E}[\alpha] &= && \text{(by (12.2))} \\ &= \sum_a a \Pr[\alpha = a] = && \\ &= \sum_{a < c} a \Pr[\alpha = a] + \sum_{a \geq c} a \Pr[\alpha = a] \geq && (\alpha \geq 0) \\ &\geq \sum_{a \geq c} a \Pr[\alpha = a] \geq && \\ &\geq \sum_{a \geq c} c \Pr[\alpha = a] = && \\ &= c \sum_{a \geq c} \Pr[\alpha = a] = && \\ &= c \Pr[\alpha \geq c]. && \blacksquare \end{aligned}$$

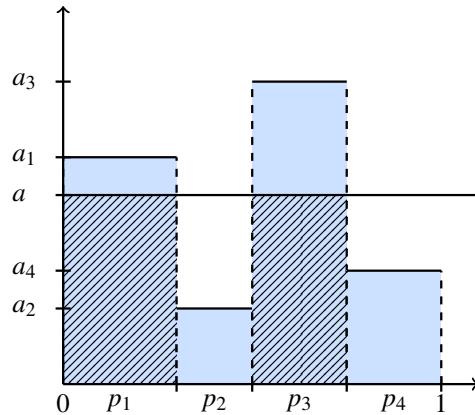


Figure 12.3:  $E[\alpha]$  is the area of colored region and  $c \Pr[\alpha \geq c]$  is the area of dashed region.

Below, we provide a geometric interpretation of Markov's inequality on a toy example. Consider a probability space with four outcomes having probabilities  $p_1, p_2, p_3, p_4$  and a random variable  $\alpha$  that gets *non-negative* values  $a_1, a_2, a_3, a_4$  on these outcomes. If we draw four rectangles of sizes  $p_1 \times a_1, p_2 \times a_2, p_3 \times a_3$ , and  $p_4 \times a_4$ , then their total area is  $E[\alpha]$ . Now, take any non-negative value  $c$  and consider an event  $\alpha \geq c$ . The probability of this event is the sum of all  $p_i$ 's such that  $a_i \geq c$ . Thus,  $c \Pr[\alpha \geq c]$  is equal to the area of the shaded region in Figure 12.4.2. Markov's inequality says that it cannot be larger than  $E[\alpha]$ : indeed, the shaded region is contained in the colored region of area  $E[\alpha]$ .

**Problem 306** Bob invited thirty friends to his party. From the previous experience, Bob knows that each of his friends will show up to the party with probability  $2/5$ . From Problem 297, we know that the expected number of guests at the party is 12. Bob is deciding how much snacks to purchase for the party. He would like to upper bound the probability that there will be at least 18 people. What upper bound on this probability can he get from Markov's inequality? Try it: [Coursera](#).

**Problem 307** Alice makes three mistakes on average on a random test in the course she is taking. What is the best upper bound we can get from Markov's inequality on the probability that she will make at least 15 mistakes? Try it: [Coursera](#).

### Summary

- Random variables correspond to measurements of random processes.
- The expectation is the most important number characteristics of a random variables: it is the weighted average of all values.
- Markov's inequality allows to show that the probability that a non-negative random variable attains a large value is small.

## 13. Dice Games

In this chapter, we will apply accumulated knowledge to analyze a simple dice game. In the game, each of two players selects a dice from a given pool of dice. Then, they throw their dice, and the one with a larger number wins. What can be simpler, right? Somewhat counterintuitively, the game turns out to be not as simple as it looks. We will see some surprising properties of this game and will implement an optimal strategy for playing this game.

### 13.1 Dice Game Problem

Someone offers you to play the following game. There are several fair dice on the table with various numbers on their sides. You and your opponent pick one dice each. Both of you throw the dice and whoever has the larger number wins. You are going to play this game many times and you are, of course, interested in maximizing the probability of winning.

To give you an advantage, your opponent lets you pick your dice first: you pick a dice and then your opponent picks one of the remaining dice.

**Stop and Think!** How would you select a dice?

Of course, everything depends on the specific dice that are on the table. Say, if there are several standard dice (with numbers 1, 2, 3, 4, 5, 6 on their sides), then it does not matter which one to choose. But what if you see the dice shown in Figure 13.1?

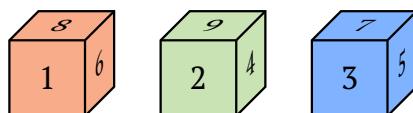


Figure 13.1: Three dice: [1, 1, 6, 6, 8, 8], [2, 2, 4, 4, 9, 9], and [3, 3, 5, 5, 7, 7].

**Stop and Think!** Which one would you choose?

The expected value of each dice is equal to 5:

$$\frac{1+8+6}{3} = 5, \quad \frac{2+4+9}{3} = 5, \quad \frac{3+7+5}{3} = 5.$$

**Stop and Think!** Does this mean that it does not matter which one to choose?

As we have discussed in Section 12.3, comparing dices by their expected values is not a good idea. Instead, let us compute, for each pair of dice, which one of them beats the other one more often.

Let us start with the first two dice. Figure 13.2 shows all 36 outcomes. The highlighted part shows that

$$\Pr[\text{dice 2 beats dice 1}] = \frac{20}{36} \approx 0.555.$$

Thus, the second dice beats the first dice more often.

|     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|
| 1,2 | 1,2 | 1,4 | 1,4 | 1,9 | 1,9 |
| 1,2 | 1,2 | 1,4 | 1,4 | 1,9 | 1,9 |
| 6,2 | 6,2 | 6,4 | 6,4 | 6,9 | 6,9 |
| 6,2 | 6,2 | 6,4 | 6,4 | 6,9 | 6,9 |
| 8,2 | 8,2 | 8,4 | 8,4 | 8,9 | 8,9 |
| 8,2 | 8,2 | 8,4 | 8,4 | 8,9 | 8,9 |

Figure 13.2: Comparison of the first two dice: the second one beats the first one with probability  $20/36 \approx 0.555$ .

Since the second dice is better than the first one, let us compare it to the third dice. Figure 13.3 shows that

$$\Pr[\text{dice 3 beats dice 2}] = \frac{20}{36} \approx 0.555.$$

|     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|
| 2,3 | 2,3 | 2,5 | 2,5 | 2,7 | 2,7 |
| 2,3 | 2,3 | 2,5 | 2,5 | 2,7 | 2,7 |
| 4,3 | 4,3 | 4,5 | 4,5 | 4,7 | 4,7 |
| 4,3 | 4,3 | 4,5 | 4,5 | 4,7 | 4,7 |
| 9,3 | 9,3 | 9,5 | 9,5 | 9,7 | 9,7 |
| 9,3 | 9,3 | 9,5 | 9,5 | 9,7 | 9,7 |

Figure 13.3: Comparing dice 2 and 3: the third one beats the second one with probability  $20/36 \approx 0.555$ .

Thus, the third dice is better than the second one, whereas the second one is better than the first one. Definitely, you would like to choose the third dice! Assuming that you have selected the third dice and your opponent has selected the first dice, let us simulate the game (as we did in Section 12.3).

```

from random import randint, seed

seed(27)

your_dice = [3, 3, 5, 5, 7, 7]
opponent_dice = [1, 1, 6, 6, 8, 8]

num_rounds = 10 ** 5
num_wins = 0

for _ in range(num_rounds):
    if your_dice[randint(0, 5)] > opponent_dice[randint(0, 5)]:
        num_wins += 1

print(f'Out of {num_rounds} games, you won {num_wins} times')

```

Out of 100000 throws, you won 44625 times

**Stop and Think!** This looks strange: the third dice beats the first one less often. How is this possible? Is there a flaw in our previous argument?

In fact, all our computations above are correct. The place where we were too fast is when we concluded that the third dice is better than the first one. To verify this, let us compare these two dice directly.

|     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|
| 3,1 | 3,1 | 3,6 | 3,6 | 3,8 | 3,8 |
| 3,1 | 3,1 | 3,6 | 3,6 | 3,8 | 3,8 |
| 5,1 | 5,1 | 5,6 | 5,6 | 5,8 | 5,8 |
| 5,1 | 5,1 | 5,6 | 5,6 | 5,8 | 5,8 |
| 7,1 | 7,1 | 7,6 | 7,6 | 7,8 | 7,8 |
| 7,1 | 7,1 | 7,6 | 7,6 | 7,8 | 7,8 |

Figure 13.4: Comparing of dice 1 and 3: the first one beats the third one with probability  $20/36 \approx 0.555$ .

Thus, the second dice is better than the first one, the third one is better than the second one, and the first one is better than the first one! How is this possible? We are used to comparing numbers and we are used to certain natural properties of numbers. One of such properties is known as *transitivity*: if  $a < b$  and  $b < c$ , then  $a < c$ . This experience translates to real life where we usually compare numbers: price, size, weight, etc.

However random variables are more complicated objects than numbers. After all, to define a random variable, you write down a table with outcomes, their probabilities, and the values of the random variable for each outcome. What is the right way to compare such tables?

Now, what does this mean for our game? We got that for each dice, there is another one that is better! This means that when your opponent offers you to pick the dice first, he does *not* give you an advantage! On the contrary, your opponent gets an advantage, since once he knows your dice, he can pick the better one!

## 13.2 Optimal Strategy

After going through the example of the previous section, let us try to come up with an optimal strategy for the game by answering the following questions.

**Problem 308** You see a pool of dice on a table. Would you like to choose a dice first? If you would, which dice would you like to choose? Try it: [Coursera ↗](#).

To summarize our setting, the game is specified by a pool of dice. The first player picks a dice, next the second player picks one of the remaining dice. They throw their dice and whoever gets the larger outcome wins. Each players wants to win in the long run, or in other words, wants to have a better winning probability in each round. Our goal is to determine who of the players has a winning strategy and to find this strategy.

To find the player with a winning strategy, we first need to describe in reasonably simple terms how to determine the winner. It turns out that the following simple criteria holds.

**Proposition** The first player has a winning strategy if and only if there is a dice that is better than all others.

*Proof.*

1. If there is a dice that is better than all others, then *the first player wins*. Indeed, if there is the best dice, then the first player can just pick this dice. Whatever the second player takes is worse, so the first player wins.
2. If there is *no* dice that is better than all others, then *the first player does not win*. Indeed, this means that for any dice, there is another dice that is better. Then, the second player wins: whatever the first player picks, the second player can just pick a better dice.

■

We are ready to implement a solution! Try it: [Coursera ↗](#). Before we start, we need to fix the format of the input. We pass a set of dice as a list of lists: each element of the outer list is a dice, each inner list specifies a dice by six integers on its sides. For example, the input corresponding to the set of dices from the previous section looks like this.

```
dataset1 = [
    [1, 1, 6, 6, 8, 8],
    [2, 2, 4, 4, 9, 9],
    [3, 3, 5, 5, 7, 7],
]
```

We implement a solution step by step. To solve the game, we definitely need to compare pairs of dice. Let us implement this as a separate function. The function takes two dice as an input and counts the number of winning outcomes for each of two dice by enumerating all 36 outcomes.

```
def count_wins(dice1, dice2):
    assert len(dice1) == 6 and len(dice2) == 6
    dice1_wins, dice2_wins = 0, 0

    for (i1, i2) in product(range(6), repeat=2):
        if dice1[i1] > dice2[i2]:
            dice1_wins += 1
        elif dice2[i2] > dice1[i1]:
            dice2_wins += 1

    return dice1_wins, dice2_wins
```

To check whether there exists the best dice, we iterate through all dice and for each dice, we check whether it is better than all others. If no dice happens to be the best, we return  $-1$ .

```
def find_the_best_dice(dices):
    assert all(len(dice) == 6 for dice in dices)

    for i in range(len(dices)):
        is_the_best = True

        for j in range(len(dices)):
            if i == j:
                continue

            (a, b) = count_wins(dices[i], dices[j])
            if a <= b:
                is_the_best = False

        if is_the_best:
            return i

    return -1
```

Finally, we need to determine the winner and find a winning strategy. To do this, we start by checking whether there is the best dice. If it exists, we set the value of the key `choose_first` to `True` in the `strategy` dictionary and return the strategy. Otherwise, we set this flag to `False`. Then, for each dice, we find another dice that is better.

```
def compute_strategy(dices):
    assert all(len(dice) == 6 for dice in dices)

    strategy = dict()
    strategy['choose_first'] = True
    strategy['first_dice'] = 0
    for i in range(len(dices)):
        strategy[i] = (i + 1) % len(dices)

    best_dice = find_the_best_dice(dices)
    if best_dice != -1:
        strategy['choose_first'] = True
        strategy['first_dice'] = best_dice
        return strategy

    strategy['choose_first'] = False
    for i in range(len(dices)):
        for j in range(len(dices)):
            if i == j:
                continue

            this_dice = count_wins(dices[i], dices[j])
            current_dice = count_wins(dices[i], dices[strategy[i]])

            if this_dice[0] - this_dice[1] < \
               current_dice[0] - current_dice[1]:
                strategy[i] = j

    return strategy
```

**Summary**

- The probability is tricky! Something might seem intuitively true, but turns out to be false.
- One should be careful when translating known properties to new domains. We cannot always deal with random variables as with numbers.