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 course in the associated specialization at Coursera, Mathematical Thinking in Computer Science ⊡. Future editions will cover the additional four courses, Combinatorics and Probability, Graph Theory, Number Theory and Cryptography, and Delivery Problem.

There are 180 problems and 90 code snippets in the book. Most of the problems come with solutions and more than 50 are graded automatically (allowing you to check your solution immediately). 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

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:

<div align="center">

Try it online! ⎘

</div>

Note that such a link is *clickable*: clicking on it will open a browser page where you can submit your answer and get instant feedback. This requires an active subscription to our Coursera specialization ⎘. 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.
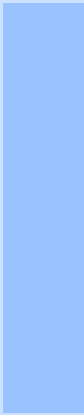
In the cloud. Alternatively, you may run all our code snippets from your Internet browser, without installing or configuring anything on your machine. For this, visit the page github.com/alexanderskulikov/discrete-math-python-scripts ⧉ and click the badge "Open in Colab". This will show you a list of notebooks that can be run in an interactive mode right in your browser (together with links to a tutorial on notebooks).

0.4 Acknowledgments

Mathematical Thinking in Computer Science

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 section, 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.[1] 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 8×8 square divided into 64 squares 1×1 (see Figure 1.1), a domino tile is a 1×2 (or 2×1) rectangle, and by saying "tiled" we mean that there are no overlaps or empty spaces. Try it online (level 0)! ⌐

---

[1] For some specific examples, Google for "Bugs in the Space Program", "Therac-25", and "Toyota unintended acceleration".

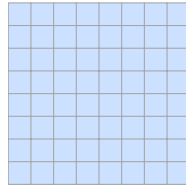Figure 1.1: An 8×8 chessboard. Can it be tiled with domino tiles?

Yes, there are many such tilings. Two of them are shown in Figure 1.2.
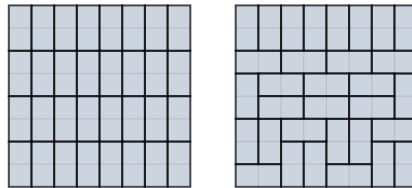
Figure 1.2: Two examples of tiling a chessboard.

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 online (level 1)! ⧉
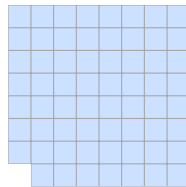
Figure 1.3: A chessboard without a corner. Can it be tiled with domino tiles?

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 we invent a systematic way to enumerate them, making sure that no possible tiling has been missed. However, here the space of possibilities is rather large.
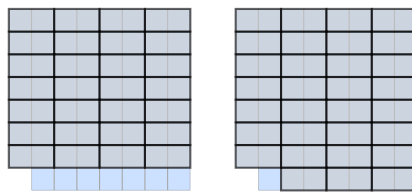
Figure 1.4: An unsuccessful attempt to tile a chessboard without a corner.
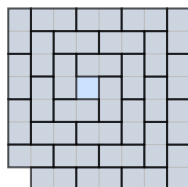
Figure 1.5: An unsuccessful attempt at a spiral tiling of a chessboard without a corner.

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 8×8 chessboard without a corner, by 1×2 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 8×8 chessboard without two adjacent corners, see Figure 1.6. Can it be tiled by domino tiles? Try it online (level 2)! ↗
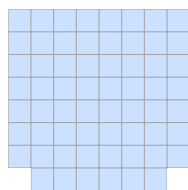
Figure 1.6: A chessboard without two adjacent corners. Can you tile it with domino tiles?

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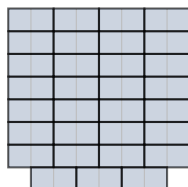
Figure 1.7: A tiling of a chessboard without two corners.

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

Problem 4 Consider an 8×8 chessboard without two opposite corners, see Figure 1.8. Can it be tiled by domino tiles? Try it online (level 3)! ⬀
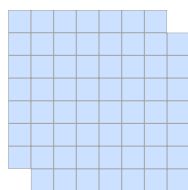
Figure 1.8: A chessboard without two opposite corners.

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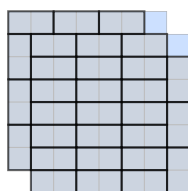
Figure 1.9: An unsuccessful attempt to tile a chessboard without two opposite corners.

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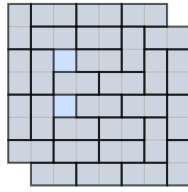
Figure 1.10: Another unsuccessful attempt to tile a board without two opposite corners.

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.
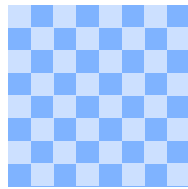
Figure 1.11: Chessboard coloring.

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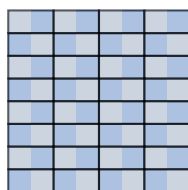
Figure 1.12: Pairing proves that the number of light cells is the same as the number of dark cells.

After this digression let us return to our board without opposite corners (Figure 1.13).
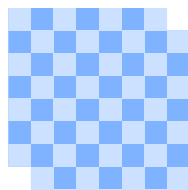
Figure 1.13: Looking again at the board without opposite corners.

> **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 8×8 chessboard without two opposite corners cannot be tiled by 1×2 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 online (levels 4 and 5)! ⧉

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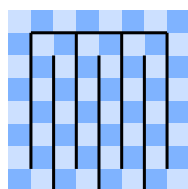
Figure 1.14: This "circular snake" helps us to prove that the board without two cells of opposite colors is tileable. Do you see how? For starters, tile the entire board by cutting the snake into domino tiles. What happens if you delete two cells from the snake?

Problem 6 We want to cover the figure shown in Figure 1.15 by 1×2 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 online (questions 1 and 2)! ⎘
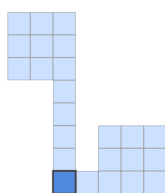
Figure 1.15: Board for Problem 6.

Now consider a 5×5 board divided into 25 cells 1×1 (Figure 1.16). It is not possible to tile it
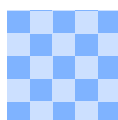
Figure 1.16: A 5×5 board (Problem 7).

by 1×2 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 (5×5 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 online (question 3)! ⬚

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

Problem 9 Can we tile a 10×10 board by 2×2 tiles? Can we tile this board by 1×4 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?
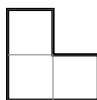
Figure 1.17: Can you cut this figure into two congruent pieces?

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 8 sides). Try it online (level 2)! ⧉
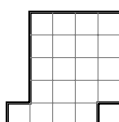
Figure 1.20: Can you cut this octagon into two congruent pieces?

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.)
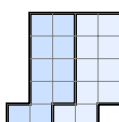
Figure 1.21: A solution to Problem 12.

How curious! There is another solution to this puzzle, see Figure 1.22.
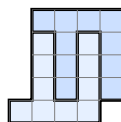
Figure 1.22: Another solution to Problem 12.

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.
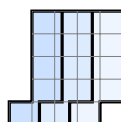
Figure 1.23: A solution to Problem 13.

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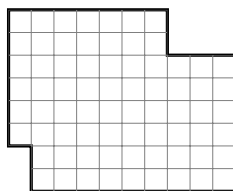
Figure 1.24: Can you cut this figure into two congruent pieces?

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*[2] (sequence) of N numbers $0, 1, 2, \ldots, N - 1$. 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`.[3]

> **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.

---

[2]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.

[3] 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 $\le$ b are integers, denotes a list that contains a, a+1, ..., 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 $7\,125$. Indeed, $7\,125 = 57 \cdot 125$ (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 $7\,125 = 57 \cdot 125$ *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.

$$x = \boxed{c \qquad\qquad z \qquad}$$
$$\overleftarrow{\qquad} k \overrightarrow{\qquad}$$

Then,

$$x = c\underbrace{00\ldots00}_{k \text{ zeros}} + z = c \cdot 10^k + z.$$

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[4] 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: $x = 7\,125$.

---

[4]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 $z = 1\,250$ and $x = 71\,250 = 57 \cdot 1\,250$. 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 $7\,125$ 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, i.e., to split items you take into two groups of the same weight. Try it online! ⬀

> **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.

```python
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 $[0, 2\pi]$. For this, one needs to know the value of π 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 $1, 11, 111, 1111, 11111, \ldots$ and divide each of these numbers by 57. We get a sequence of remainders: 1 mod 57 = 1, 11 mod 57 = 11, 111 mod 57 = 54, 1111 mod 57 = 28, 11111 mod 57 = 53 and so on (a mod 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.,

$$\underbrace{111\ldots111}_{m \text{ digits}} \bmod 57 = \underbrace{111\ldots111}_{n \text{ digits}} \bmod 57$$

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 $11\ldots1100\ldots00$, 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 $111\ldots111$ 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 $111\ldots111$ that is divisible by 57.

Problem 27 As of December 2020, there are $66\,758$ 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.

```python
from random import randint, seed

seed(10)

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

```
1546686 8195564 9096041 1248847 4457754 8760813
9242586 5656024 3688205 1577196 9735382 9222271
6499115 2276567 5194248 7059236 1747532 8053077
3326819 6957282 7402286 8067637 5758324 5401017
8664354 3930486 6085259 7083207 3231003 8665986
```

The function `randint(a,b)` returns a random integer in the range `a...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 a different 30 numbers, just change the argument of the `seed` function.)

We claim 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.

Stop and Think! Would you believe this claim?

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.

Our last example is *factoring*: one can prove using Fermat's little theorem[5] that the number

4120234369866659543855531365332575948179811699844327982845456264338764455652484261980988704231618418792614202471888694925609317763750334211309823974851509449091069102698610318627041148808669705649029036536588674337317208131041051908642547932826013912576240339463732693911

(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 \$70 000 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.

---

[5]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 *narrows* 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 4×4 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 barely visible. 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 $1 + 2 + \cdots + 16 = 136$. 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 4×4 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 1×1 table with number 1 written in its only cell. Of course, all sums are the same in this table.

> **Stop and Think! —** 2×2 **magic squares.** Do there exist 2×2 magic squares?

No, there are no 2×2 magic squares! Let us prove this. Consider a 2×2 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 29 —** 3×3 **magic squares.** Do there exist 3×3 magic squares? Try it online! ⧉

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

2.1.1 Narrowing the Search Space

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

> **Stop and Think! — Number of** 3×3 **tables?.** How many 3×3 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 $9! = 1 \times 2 \times \cdots \times 9 = 362\,880$. 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 $1 + 2 + \cdots + 9 = 45$. 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 $9 \times 8 \times 7 \times 6 = 3\,024$ choices. We do not need to consider all $362\,880$ tables anymore, because we reduced the size of the search space by a factor of 120: now it suffices to check only $3\,024$ tables. Let us not rush to check all those tables, but rather try to eliminate more cases.

Problem 30 — Value of the central entry. There is only one possible value of the central element a_5 in a 3×3 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 3×3 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 3×3 magic squares. Note that we reduced the number of cases we needed to consider from $362\,880$ to two!

We have proved that there exist 3×3 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 3×3 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 31 Does there exist a multiplicative 3×3 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 3×3 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

(a)

2^8	2^1	2^6
2^3	2^5	2^7
2^4	2^9	2^2

(b)

256	2	64
8	32	128
16	512	4

(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 32 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

128	1	32
4	16	64
8	256	2

Figure 2.9: Another multiplicative magic square.

powers (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).

1	2	0		2	4	1		0	2	1		1	9	3		2	36	3
0	1	2		1	2	4		2	1	0		9	3	1		9	6	4
2	0	1		4	1	2		1	0	2		3	1	9		12	1	18
(a)			(b)			(c)			(d)			(e)		

Figure 2.10: (a) "Additive" magic square with repetitions; (b) The corresponding multiplicative magic square with repetitions; (c) Another "additive" magic square with repetitions; (d) The corresponding multiplicative magic square with repetitions where we used powers of $c = 3$; (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 33 Does there exist a six-digit integer that starts with 100 and is a multiple of 9 127?

We are looking for a number that starts with 100 and has three more digits (from 000 to 999) and is divisible by 9 127. It gives us 1 000 candidate numbers, and we should just check whether one of them is a multiple of 9 127. A computer can easily do this.

```python
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 9 127. 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 9 127. We want to find an integer k such that

$$100\,000 \leq 9127k \leq 100\,999.$$

For this we just need to divide both 100 000 and 100 999 by 9 127:

$$10.95\ldots \leq k \leq 11.06\ldots.$$

Thus, we take $k = 11$, and the answer to the problem is $9\,127k = 100\,397$. Our reasoning also shows that 100 397 is the only number satisfying the given requirements.

Problem 34 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 35 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 36. Assume that n^2 starts with 31415 followed by $2k$ digits:

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

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:

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

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

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

This already gives us the first digits of n. Indeed, by taking the square root of both sides of the last equation, from $\sqrt{31\,415} = 177.242771\ldots$ and $\sqrt{31\,416} = 177.245592\ldots$, we have that

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

In particular, we can take $n/10^k = 177.243$, e.g., $n = 177\,243$ and $k = 3$. This gives us an example of such a number, because $n^2 = 31\,415\,081\,049$.

Problem 36 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 37 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 38** 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 online! ⊏

Consider another version of this problem.

> **Problem 39** 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 40** 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 online (level 1)! ⊏

These numbers are too large to check all the possibilities. Let us try to identify a smaller sub-problem. Solving it will lead us to a solution for 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 41** 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 online (level 2)! ☐

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 finding 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 online! ☐).

> **Problem 42** Does there exist a 5-digit integer N such that N^2 starts with 27182?

> **Problem 43** 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 44** 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 45** Some country uses only coins of values 12, 20, and 30 florins. What is the smallest (positive) amount that can be paid if both sides 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 46 — 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, i.e., 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, i.e., 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

$$M \quad \leq \quad 500, \tag{2.1}$$
$$D \quad \leq \quad 200, \tag{2.2}$$
$$M + D \quad \leq \quad 600. \tag{2.3}$$

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:

$$20D \quad \leq \quad 4000,$$
$$10M + 10D \quad \leq \quad 6000.$$

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.
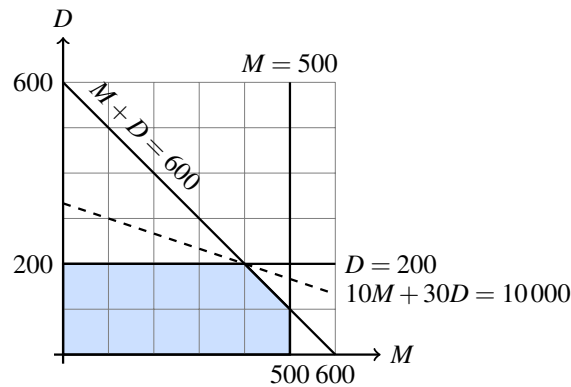
Figure 2.11: We plot all the restrictions on the pair (M, D); the region of possible production plans is highlighted. The dashed line indicates the production plans with a profit of $10\,000$ per day. We see that there is exactly one production plan that achieves this profit. The same would be true even 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 $c_1(M+D) + c_2 D = 30D + 10M$, we use $c_1 = 10$ and $c_2 = 20$.

Summary

A proof that α is optimal consists of two parts:
- there exists a solution achieving the value α;
- no solutions achieve a value better than α.

2.2.2 Maximum Number of Two-digit Integers

Problem 47 — Maximum number of two-digit integers. There are 90 cards with all two-digit numbers on them:

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

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 online!

In mathematical language, the problem can be restated as follows: what is the maximum number of elements in a subset of $\{10, 12, ..., 99\}$ that does not contain two numbers x and y with $x + y = 100$? Even more formal version: find

$$\max\{|S| : S \subseteq \{10, 11, \ldots, 99\} \text{ and } \forall \text{ distinct } x, y \in S, (x + y \neq 100)\}.$$

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., $10, 11, 12, \ldots, 18, 19$), ten numbers starting with 2, $\ldots$, 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 $m \ldots n$ (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:

$$(10,90), (11,89), \ldots, (49,51). \tag{2.4}$$

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

The integers $91, 92, \ldots, 99$ 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:

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

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

$$50, 51, \ldots, 99.$$

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: $50, 51, \ldots, 99$. 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 not contain 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.
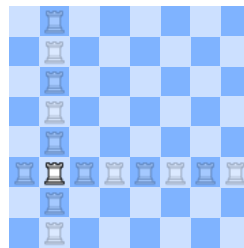

Figure 2.12: A chess rook moves vertically and horizontally.

> **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 48** What is the maximum number of rooks on a chessboard such that no two rooks attack each other? Try it online! ⬀

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.
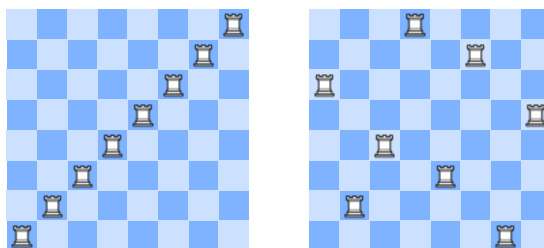

Figure 2.13: Two ways of placing eight rooks that do not attack each other. (Do we really need both of them? Of course not, one would be enough.)

> 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 49** 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 50** List all the possible numbers of cells that a knight may attack.

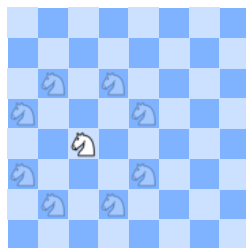You probably guessed what is our next question.

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

> **Problem 51** What is the maximum number of knights on a chessboard such that no two of them attack each other? Try it online! ⬀

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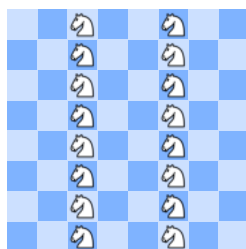
Figure 2.16: In this configuration, we have 16 knights and can't add more since all the cells are under attack. But this does not mean that 16 is the maximum number of knights on the board.

> **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 47, 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).
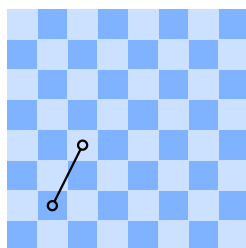

Figure 2.17: A pair of cells that can't be used together.

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 2×4 (smaller that this boards do not work).

> **Stop and Think!** Do you see how to group the 8 cells of a 2×4 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 8×8 board?

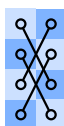The required pairing for a 2×4 board is shown in Figure 2.18.

Figure 2.18: A 2×4 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
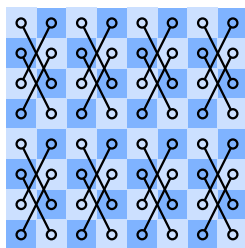
Figure 2.19: Combining eight 2×4 boards to get a pairing for 8×8 board.

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

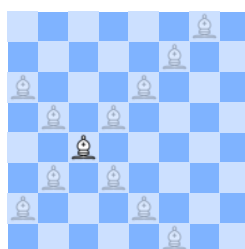A *bishop* attacks in every direction diagonally (Figure 2.20).

Figure 2.20: A bishop moves diagonally.

Problem 52 How many bishops not attacking each other can be placed on a chessboard?

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).

(a)   (b)

Figure 2.21: A solution to Problem 52.
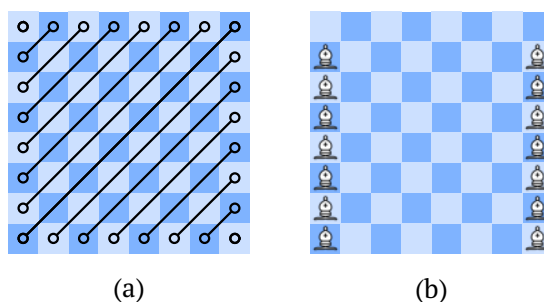
There are two remaining chess figures, the royal ones. If you know how they move, you can try to find out the maximum number of *queens* that can be placed on a chessboard. Since the rook moves are available to queens, we cannot place more than eight queens; it turns out (see Section 2.3) that it is 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 53 Prove that one can place 16 kings not attacking each other, but no more.

Hint: divide the board into 2×2 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 54 Prove that the minimum number of kings attacking all cells is $n = 9$.

Hint: cut the board into 3×3 pieces (this can be done for a 9×9 board, but for an 8×8 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 47, 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 55 What is the maximum number of integers among $1, 2, \ldots, 50$ that one can select, if one is not allowed to select simultaneously x and $2x$? Try it online! ⬀

▌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 $x/2$ and $2x$ (if present among $1, 2, \ldots, 50$).

▌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 $27, 29, \ldots, 49$ 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 $1, 2, \ldots, 50$ 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?

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
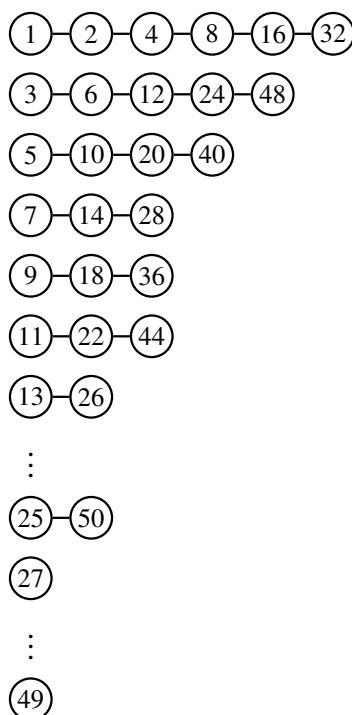
(1)—(2)—(4)—(8)—(16)—(32)

(3)—(6)—(12)—(24)—(48)

(5)—(10)—(20)—(40)

(7)—(14)—(28)

(9)—(18)—(36)

(11)—(22)—(44)

(13)—(26)

⋮

(25)—(50)

(27)

⋮

(49)

Figure 2.22: Chains of mutually incompatible numbers.

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 $7,9,11$, and all the remaining chains give only one node. In total, we get the sum $3+3+2+2+2+2+1+1+\cdots+1$.

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: $1,3,5,\ldots,49$. 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 $3+3+2+2+2+2+19 = 33$. 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 56   How many different maximum solutions are there (subsets of $\{1,\ldots,50\}$ 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 512!

Problem 57   What is the maximum number of distinct integers from $1,2,\ldots,n$ 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:

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

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 $\left\lfloor \frac{n+1}{2} \right\rfloor$ numbers, the third column has $\left\lfloor \frac{n/4+1}{2} \right\rfloor$ 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 58 — 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 online! ⬀
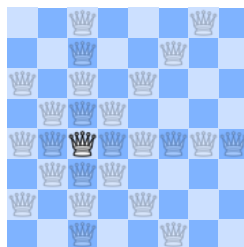

Figure 2.23: Moves of a chess queen.

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 59** 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, in particular, 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.
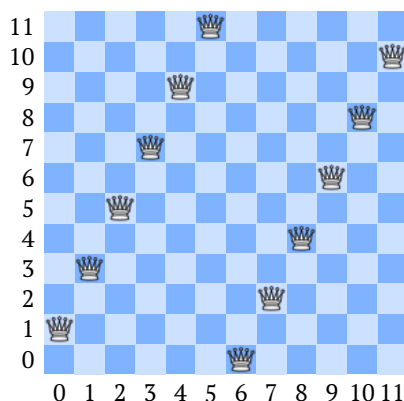
Figure 2.24: A correct placement of 12 queens according to the formula from Problem 59.

Implementing such 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 60 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 $r_0, r_1, \ldots, r_{n-1}$ of n integers: r_i is the index of the row containing a queen in the i-th column. That is, a sequence $r_0, r_1, \ldots, r_{n-1}$ defines the following n cells on the board:

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

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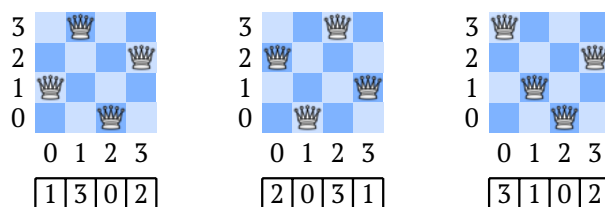

Figure 2.25: A placement of n queens in different rows and columns can be represented as a permutation.

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!

```python
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?

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

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

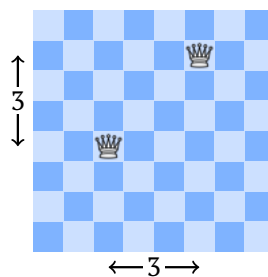i.e., if the horizontal shift is the same as the vertical shift (see Figure 2.26).

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.

```python
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.

```python
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)
```
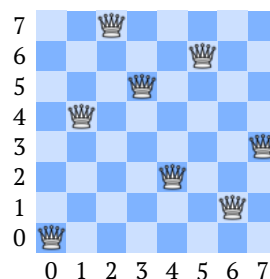

Figure 2.27: A solution to the 8 queens problem found by the program that we've implemented.

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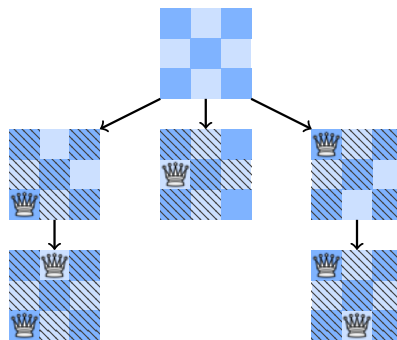
Figure 2.28: A backtracking approach to the 3 queens problem.

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.

```python
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.

```python
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. For 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).

```python
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.

```python
def can_be_extended(perm):
    i = len(perm) - 1
```

```python
    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 61 In a 5×5 grid, draw 16 diagonals that do not touch each other. (E.g., in 3×3 and 4×4 grids one can place 6 and 10 diagonals, respectively. See Figure 2.29.) Try it online! ⬀
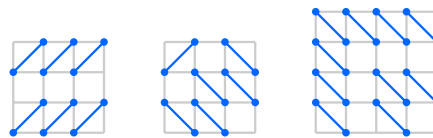

Figure 2.29: Placing many non-intersecting diagonals.

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 62 Prove that one cannot place more than 16 non-intersecting diagonals on a 5×5 grid.

This is another challenging problem. To see that 16 is optimal, note that a 5×5 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.
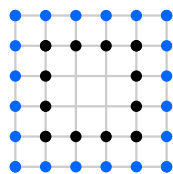
Figure 2.30: Two layers of nodes in the 5×5 grid.

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 5×5 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 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 1×2 rectangles. For example, one can place 20 such diagonals on a 6×6 grid, see Figure 2.31.
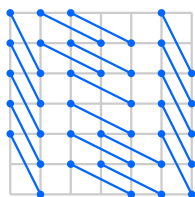
Figure 2.31: 20 non-intersecting 1×2 diagonals on a 6×6 grid.

Problem 63 Prove that it is possible to place 21 non-intersecting diagonals of length $\sqrt{5}$ in a 6×6 grid.

In Section 7.3, we use ILP-solvers to solve the diagonals problems 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. Being able to see such decompositions 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.

```python
def length(lst):
    if not lst:
        return 0
    else:
```
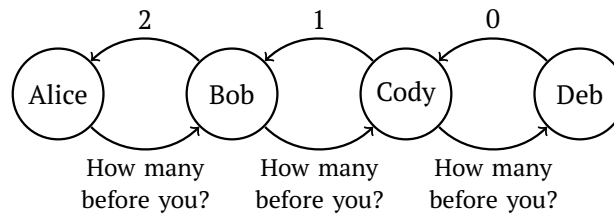
Figure 3.1: In order to find the number of people in the line before her, Alice asks Bob, "How many people are ahead of you?". In turn, Bob asks Cody the same question, and Cody asks Deb. There are no more people before Deb and Deb tells this to Cody. Cody realizes that there is exactly one person before 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.

```
        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!

```python
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.

```python
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 $1, 2, \ldots, n$. 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 64** 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:

```python
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 $2, 3, \ldots, n$. It is done as follows: `range(1, n + 1)` is (according to Python conventions) the list $[1, 2, \ldots, n]$. The line `result *= i` (equivalent to: `result = result * i`) multiplies the result by i. After we do this for i=1,2,...,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 $1 \cdot 2 \ldots n$ 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 letter.

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:

$$n! = (n-1)! \cdot n. \tag{3.1}$$

Hence, if you want to compute $n!$, you may ask somebody to compute $(n-1)!$, and then multiply the result by n. You may even ask yourself what is $(n-1)!$ and then use the same trick (but it is important to remember that later the value of $(n-1)!$ should be multiplied by n). This is what the Python interpreter does when you run it on the following program:

```python
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 $2, 3, \ldots, n$ 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 `factorial(0)=1`?

First, we should replace `n > 0` by `n >= 0` in the assertion. For the non-recursive program, this is enough: for $n = 0$, the list `range(1, n + 1)=range(1, 1)` is empty, no multiplication is performed, and `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.

```python
from numpy import prod


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

```python
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

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

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 `factorial(10)`, we need to compute `factorial(11)`, for that we need to compute `factorial(12)`, and we never come to anything known. Trying to run this program, we get an error message.

```python
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.

Problem 65 For what integers n does the following program terminate?

```python
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 `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 `strange(11)` will call `strange(12)` that will call `strange(13)`, and so on. Thus, for $n = 11$ the program never terminates (as well as for $12, 13, \ldots$). You may try to run the program to see what kind of error it will produce.

Problem 66 What `strange(1)` and `strange(0)` do return?

Sometimes it is difficult to guarantee that the 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.

```python
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.

```python
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.)

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

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

Stop and Think! For what values of n this function terminates (and returns 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 67 What is the value returned by the following program for an integer $n \geq 0$?

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

Problem 68 The same question for the program

```python
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 the decompressor program is distributed in a compressed form, to get it running we first need to get it running.[1]

If you have a movable webcam, you can point it on the screen and see the chain of images: of a screen, of an image of a screen on the screen, etc. Something similar happens when sharing

---

[1]This is a real story: the documentation even acknowledged this problem by saying "This is a bit of chicken and egg problem".

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.
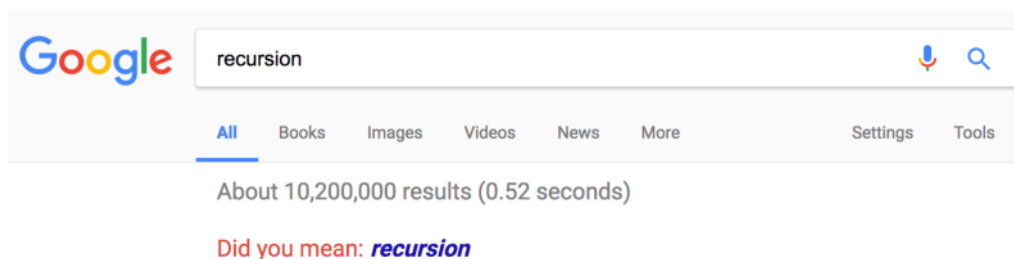

Figure 3.2: Google and recursion.

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 69 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 integers with sum n where only 3 and 5 could appear.

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

$$8 = 3 + 5,$$
$$9 = 3 + 3 + 3,$$
$$10 = 5 + 5,$$
$$11 = 5 + 3 + 3.$$

These examples do not show us an obvious pattern, so it is hard to derive to a rule of thumb with what we have found. Can we move on from $8, 9, 10$, and 11 to 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, you 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 70** How can we convert this proof into a program that gets $n \geq 8$ and prints the corresponding sum of 3 and 5 summands?

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

```python
def change(amount):
    assert amount >= 8
    if amount == 8:
        return [3, 5]
    if amount == 9:
        return [3, 3, 3]
    if amount == 10:
        return [5, 5]

    coins = change(amount - 3)
    coins.append(3)
    return coins


amount = 200
coins = change(amount)

print(f"{amount}={'+'.join(map(str, coins))}")
```

```
200=3+5+3+3+3+3+3+3+3+3+3+3+3+3+3+3+3+3+
3+3+3+3+3+3+3+3+3+3+3+3+3+3+3+3+3+3+3+3+
3+3+3+3+3+3+3+3+3+3+3+3+3+3+3+3+3+3+3+3+
3+3+3+3+3
```

Following programming best practices, we change the parameter name to a more mnemonic `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 is 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 (integer 3 is not the same as 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 show also a more compact solution implementing the same recursive idea.

```python
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?

```python
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!**   We did not use the last line of our initial table, $11 = 5 + 3 + 3$. Is it OK? Or did we forget something?

The last line was redundant, since $11 = 8 + 3$ and can be paid in a general way (by a recursive call).

> **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[2] 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 it!) to `change(9)=[3,3,3]`. So in fact we will not use 5-coins at all.

> **Problem 71**   What is the smallest amount that can be paid in two different ways?

As we have seen in our example, our program for large inputs uses mostly 3-coins

> **Problem 72**   Say we need to use more 5-coins and fewer 3-coins when there is a choice. What changes should be implemented in the program?

---

[2] 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.

Problem 73 Is it true that our program always gives an *optimal* solution in the sense that it uses as few 5-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.

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. There are no other combinations of zero, one or two coins that are in this range ($3 + 5$ is already 8), so this is the full list of payable amounts.

We considered coins of value 3 and 5, but the same question could be asked for different set 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 74 What is the maximum integer amount that *cannot* be paid with 5- and 7-coins? Try it online! ☐

Problem 75 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, 7]` or `[5, 5, 5, 5, 5, 5, 5, 7, 7]` or `[7, 5, 5, 5, 5, 5, 5, 5, 7]`. Try it online! ☐

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 disc at a time (this disc should be the upper one on some rod), and a larger disc should never be placed on top of a smaller one.
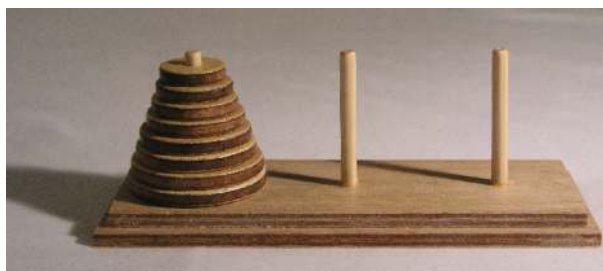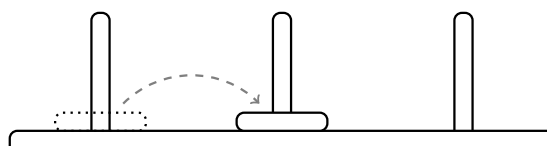

Figure 3.3: The Hanoi Towers puzzle. (Source: Wikipedia ☐.)

Stop and Think! Can you solve this puzzle for $n = 1, 2, 3$? Try it online! ☐

For $n = 1$, this is easy: we move the (only) disc to another rod. (Any disc 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 disc to one of the empty rods; by that we "free" the large disc and can move it to another rod.
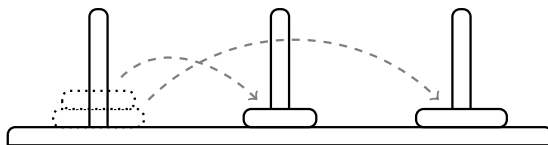

It remains to move the small disc on top of the large disc to get the solution for $n = 2$.

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

Stop and Think! When is it possible? How should we prepare this move?

First, the largest disc should be free to move, that is, it should be the only disc 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 small disks should be placed on the third rod (in a correct order). Hence, we need to achieve the following configuration.
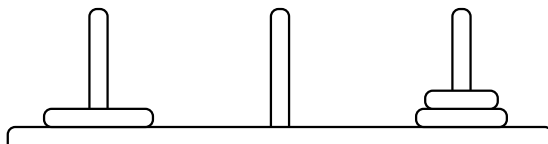
For that, we need to move two smaller disks from the first rod to the third one. And — aha! — this is exactly what we did for $n = 2$: when we move smaller disc, the largest one is not an obstacle (any disc can be placed on top of it).

After these preparations, we move the largest disc to the middle rod.
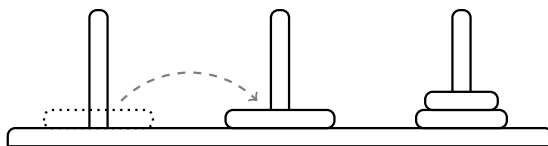
Stop and Think! Do you see how to complete the argument?

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 76 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.

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

To move the largest disk, we need to free it (leave 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:
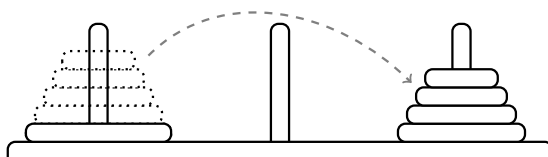
In this picture the dotted arrow indicates the effect of the preparation moves: all the disks except for the largest one should be moved to the third rod. Then, the largest disk is moved to the second rod, and it remains to bring the other disks there:
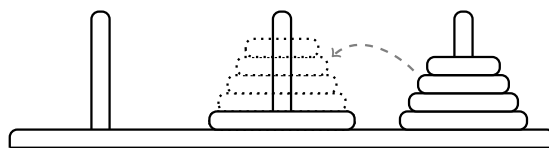
The dotted arrow indicates what should be done before and after the move of the largest disk. Fortunately, we already know (recursion!) that this is possible and recursively call the solution for $n-1$ disks. Problem 76 is solved. We can now implement it in Python.

```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 discs to the unused rod, then move the largest disk, and finally move $n-1$ discs 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 $(\texttt{from\_rod}+\texttt{to\_rod})$ from 6.)

> **Stop and Think!** We have seen the solution for $n = 1$ (one move), 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 online! ↗

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 $T_n + 1$ (that should be a power of two):

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

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 $n = 1, 2, 3, 4, 5, 6, 7, 8$ (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 $T_n = 2T_{n-1} + 1$. There are two parts:
- $T_n \leq 2T_{n-1} + 1$: 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 put accents 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.[3] 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 77 — Guess a number.** You are playing a 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 "$x < y$", or "$x > y$". 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 to do this? Try it online (level 1)! ⍰

---

[3] 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.

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

Let's see what happens if our first question is whether $x = 2$. Again, if the opponent replies that $x = 2$, then we are done. If she replies that $x < 2$, then we already know that $x = 1$. Hence, we just ask "Is $x = 1$?" as our second question and get the desired "yes" response. The case $x > 2$ is treated similarly, of course.

Problem 78 Now, let $n = 7$ and $k = 3$: guess an integer $1 \le x \le 7$ by asking at most three questions.

Probably, you have guessed already that we 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*:
- if $x < 4$, then x is equal to either 1, 2, or 3;
- if $x > 4$, then x is equal to either 4, 5, or 6.

This, in turn, means that in both cases one can invoke the solution to Problem 77. Recursion! The resulting protocol of questions is shown in Figure 3.4.
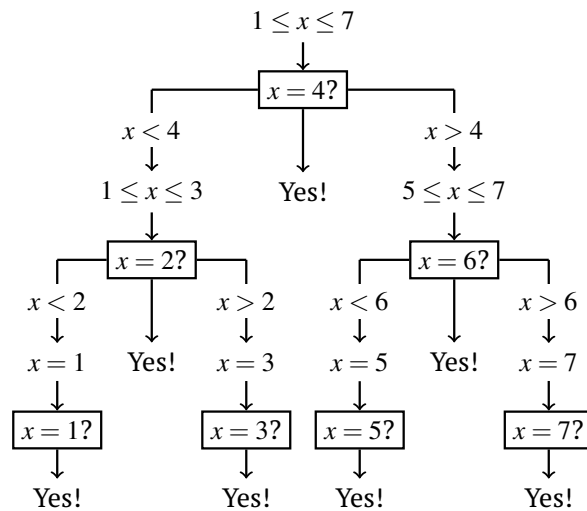

Figure 3.4: Guessing an integer $1 \le x \le 7$ by asking at most three questions.

Interestingly enough, this strategy allows us to guess an integer $1 \le x \le 2\,097\,151$ (about two millions!) in just 21 questions.

Problem 79 Guess an integer $1 \le x \le 2\,097\,151$ by asking at most 21 questions. Try it online (level 4)! ↗

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

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

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

```python
def query(y):
    x = 1618235
    if x == y:
        return 'equal'
```

```python
    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:
        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 $1 \le x \le n$ will make 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 $n \le 10^9$, then $\log_2 n < 30$.

```python
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.

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

$$a_0 \leq a_1 \leq \cdots \leq a_{n-1}.$$

It turns out that in this case about $\log_2 n$ comparisons are enough! This is a massive improvement: remember that for large values of n, $\log_2 n$ is much smaller than n. Say, for $n = 10^9$, the linear scan may make a billion comparisons, whereas the binary search never makes more than 30 comparisons.

The idea is again to try to half the search space. For 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,

$$x < a_{n/2} \leq a_{n/2+1} \leq \cdots \leq a_{n-1}.$$

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

$$a_0 \leq a_1 \leq \cdots \leq a_{n/2} < x.$$

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

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

    if len(a) == 0:
        return False

    if a[len(a) // 2] == x:
        print('Found!')
        return True
    elif a[len(a) // 2] < x:
        return binary_search(a[len(a) // 2 + 1:], x)
    else:
        return binary_search(a[:len(a) // 2], x)


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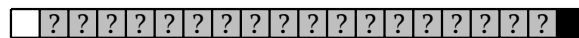
```
Searching 8 in [1, 2, 3, 3, 5, 6, 6, 8, 9, 9, 9]
Searching 8 in [6, 8, 9, 9, 9]
Searching 8 in [6, 8]
Found!
```

You might have noticed that in the guess-a-number problem we also used implicitly that the input data is sorted. At the same time, the binary search method could help even when the data is not sorted. We illustrate this by the following example.

> **Problem 80 — Two cells of opposite colors.** In the line made of twenty cells, the first (leftmost) one is white, and the last one is black. All others are either black or white.
>
>
>
> You may reveal the color of any cell. Find two adjacent cells of opposite colors by revealing the color of at most five cells. Try it online! ⤢

First, let us focus on the following question: *How do we know that there is a pair of adjacent cells of opposite colors?* Informally, this is true because when moving from left to right, the color of a cell should switch from white to black at least once. Formally, consider the leftmost cell. It is white. If its neighbor cell is black, then we are done. If it is white, then consider the next cell (to the right). If it is black, then, again, we are done. Otherwise proceed one step to the right. We know that this cannot continue forever as the rightmost cell is black. Note three 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 doesn't prove that the sequence always contains a *unique* pair of adjacent cells of opposite colors. There may be many such pairs. What we proved above is that there is *at least* one such pair.

3. It proves that there exists a pair of adjacent cells, 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 required pair of adjacent cells, but in the worst case it will require us to reveal the color of all cells.

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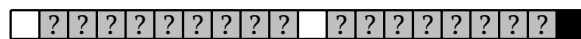

Stop and Think! Do you see how to proceed?

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).
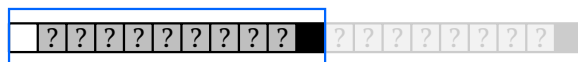

Hence, in any case, we decrease the size by a factor of (almost) two: from sequence of length n, we get a subsequence of length $\lceil \frac{n+1}{2} \rceil$. This way, we will need to reveal the color of at most five cells before we find what we are looking for. Indeed, the size of the sequence will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

$$20 \to 11 \to 6 \to 4 \to 3 \to 2\,.$$

When the size of the current sequence is equal to two we already know the required pair since the leftmost cells 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.
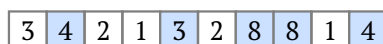
Problem 81 — Local maximum. Consider an integer sequence of length 16 whose elements we don't know.
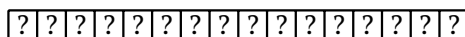
Find (any) local maximum by revealing at most seven of them. Try it online! ↗

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

Mathematical induction is as simple as falling dominoes, but it allows to make convincing arguments for arbitrarily large and complex problems by decomposing them into smaller problems, and then solving them step by step.

Say, we have statements $A(1), A(2), A(3) \ldots$, and we want to prove that they all are true: for every $n \geq 1, A(n)$ is true. A proof by induction 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 $n \geq 1$. Indeed, $A(1)$ is true, we proved this in the base case. Now we take $n = 1$, and the induction step ensures that $A(2)$ is true as well. We take $n = 2$, and now the induction step proves the statement $A(3)$. This way, we prove all the statements $A(n)$ (see Figure 3.5).
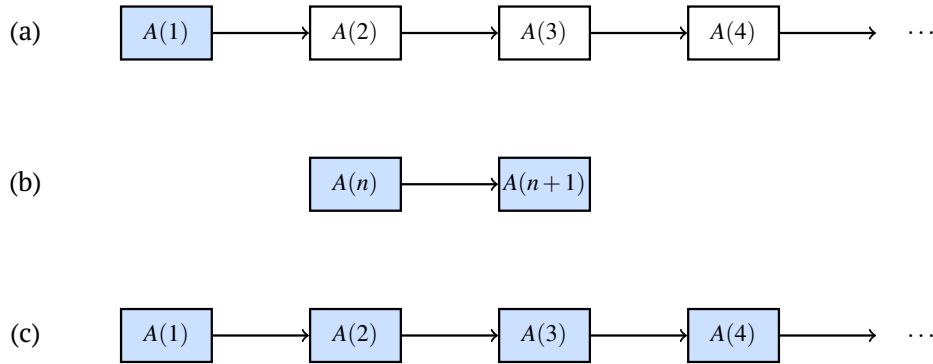

Figure 3.5: (a) The base case ensures that the first statement is true. (b) The induction step lets us move from any statement to the following one. (c) From this, we conclude that all the statements are true.

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 any other number $n = c$ (say, $c = 0$, or $c = 15$, or $c = -3$). In this case, we will prove all statements $A(c), A(c+1), A(c+2), \ldots$

The second part of an inductive 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 *inductive hypothesis*. Assuming the inductive hypothesis, we prove the statement $A(n+1)$, and this finishes the proof. Sometimes it comes in handy to use a stronger inductive hypothesis: assume that all the previous statements $A(1), \ldots, A(n)$ are true, and use them to prove correctness of $A(n+1)$. This variation of inductive proofs is often called *strong induction*.

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 by the previous one.

> **Problem 82 — arithmetic series.** Prove that for every $n \geq 1$, the sum of all integers from 1 to n is $n(n+1)/2$:
>
> $$\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}. \tag{3.3}$$

Below, we will prove this formula for all n, but in the meantime let us check that it holds at least for some values of n.

```python
for n in (17, 251, 1356):
    assert sum(range(1, n + 1)) == n * (n + 1) // 2
```

Note that this code only convinces us that the formula holds sometimes, but to be sure that the formula holds for *all values of n* we need a rigorous mathematical proof. This is reminiscent

Figure 3.6: The domino effect: pushing the first domino knocks down all other dominoes. (Source: Wikipedia ⬈.)

of software testing. While we 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).

We can use induction to prove the arithmetic series formula (3.3). 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:

$$\sum_{i=1}^{n+1} i = \left(\sum_{i=1}^{n} i \right) + (n+1)$$
$$= \frac{n(n+1)}{2} + (n+1) \qquad \text{(by ind. hypothesis)}$$
$$= \frac{n(n+1)}{2} + \frac{2(n+1)}{2}$$
$$= \frac{(n+1)(n+2)}{2}.$$

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 ⬈).

1	+	2	+	$\cdots$	+	$(n-1)$	+	n
n	+	$(n-1)$	+	$\cdots$	+	2	+	1
$(n+1)$	+	$(n+1)$	+	$\cdots$	+	$(n+1)$	+	$(n+1)$

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).
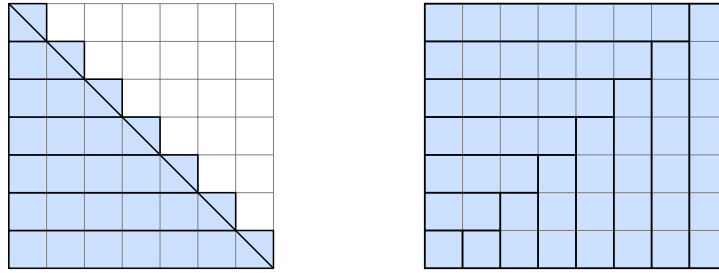
Figure 3.7: Geometric proofs of the arithmetic series formula.

Figure 3.8: Johann Carl Friedrich Gauss (1777–1855). (Source: Wikipedia ⬀.)

Problem 83 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.
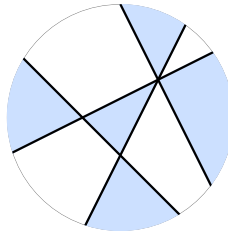
Note that it is crucial that all lines in this problem are *straight* and *infinitely long.* Indeed, the figure below shows that if the lines are not straight (a), or if the lines are not infinitely long in both directions (b), then more than two colors may be needed.
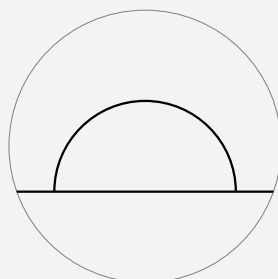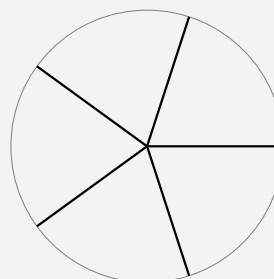
(a)   (b)

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 83. 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,

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.
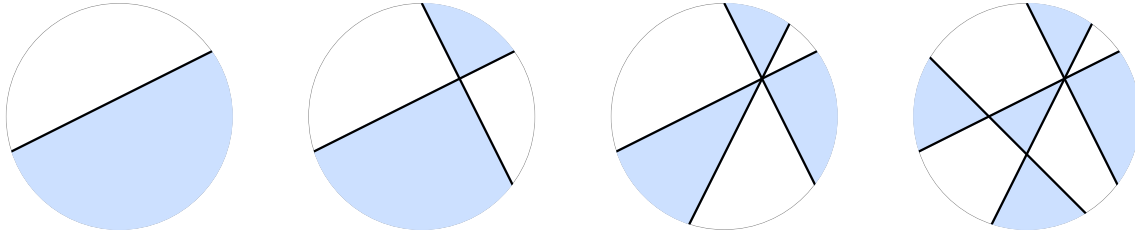
all regions on one side of L keep their colors, while all regions on the other side of L switch their colors. Consider two neighboring regions R_1 and R_2. If their common border is not a part of L, then they had different colors in the old coloring, and they still have different colors in the new coloring. On the other hand, if their common border belongs to L, then they use to be one region and had the same color, but one of the regions switched their color.

3.2.1 Compound Percents

> **Stop and Think!** Simple interest deposit is a kind of deposit where you earn $x\%$ of the *initial* deposit each period (day, month or year). 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. And this holds in general: compound interest is always no worse that 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
> $$(1+x)^n \geq 1 + xn. \tag{3.4}$$

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 $n \geq 0$:

$$
\begin{aligned}
(1+x)^{n+1} &= (1+x)^n \cdot (1+x) \\
&\geq (1+xn) \cdot (1+x) && \text{(by ind. hypothesis)} \\
&= 1 + x(n+1) + x^2 n \\
&\geq 1 + x(n+1),
\end{aligned}
$$

where we used both the induction hypothesis for n, and the fact that an inequality can be multiplied by a non-negative value $(1+x)$.

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 $(1 + n \cdot 0.02)$. Bernoulli's inequality assures that the former is always no worse than the latter. The following code demonstrates the (huge) difference between these two cases for large values of n.

Figure 3.10: Jacob Bernoulli (1655–1705). (Source: Wikipedia ⧉.)

```python
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')
```
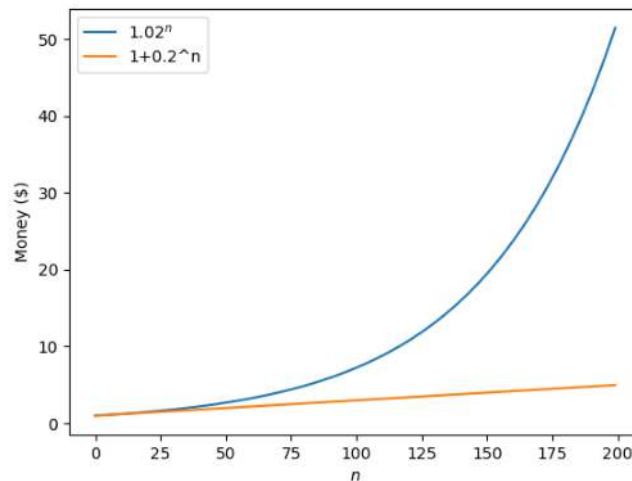

It is instructive to see how exponential functions (such as 1.02^n) grow, and to see how fast they reach large values. For example, how many days of 2% compound interest does it take to get from $1 000 to $1 000 000?

```python
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 online! ⬀).

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 $x_1, \ldots, x_n > 0$

$$\frac{x_1 + x_2 + \cdots + x_n}{n} \geq \sqrt[n]{x_1 x_2 \cdots x_n}. \tag{3.5}$$

For simplicity, first we scale all x_i so that $x_1 x_2 \cdots x_n = 1$. Namely, assume that the geometric mean of these numbers is $c = \sqrt[n]{x_1 x_2 \cdots x_n}$. 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 $x_1 x_2 \cdots x_n = 1$ without loss of generality.

It remains to show that $x_1 x_2 \cdots x_n = 1$ implies

$$x_1 + x_2 + \cdots + x_n \geq n.$$

We will prove this inequality by induction on n. The base case of $n = 1$ it trivial: $x_1 \geq 1$. For the induction step from $n \geq 1$ to $(n+1)$ we assume that we have $x_1, \ldots, x_{n+1} > 0$ whose product equals 1. In particular, this means that at least one of the numbers is ≤ 1 (indeed, if all numbers are > 1, then their product is > 1). Similarly, at least one of the numbers is ≥ 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

$$x_1 + x_2 + \cdots + x_n + x_{n+1} \geq 1 + (x_1 x_2) + x_3 + \ldots + x_{n+1}.$$

Now, consider n numbers

$$(x_1 x_2), x_3, \ldots, x_{n+1}.$$

Since their product is 1, we can apply the induction hypothesis to them: $x_1 x_2 + x_3 + \ldots + x_{n+1} \geq n$. We conclude that

$$x_1 + x_2 + \cdots + x_{n+1} \geq 1 + (x_1 x_2) + x_3 + \cdots + x_n \geq 1 + n.$$

Problem 84 Prove that for every $n \geq 1$:

$$1 \cdot (n-1) + 2 \cdot (n-2) + \cdots + (n-1) \cdot 1 = \frac{(n-1)n(n+1)}{6}.$$

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 $n+1$:

$$1 \cdot n + 2 \cdot (n-1) + \cdots + n \cdot 1 = \frac{n(n+1)(n+2)}{6}.$$

Indeed,

$$
\begin{aligned}
& 1 \cdot n + 2 \cdot (n-1) + \cdots + n \cdot 1 \\
=& 1 \cdot (n-1) + 2 \cdot (n-2) + \cdots + (n-1) \cdot 1 \\
& + 1 + 2 + \ldots + (n-1) + n \\
=& \frac{(n-1)n(n+1)}{6} && \text{(by ind. hypothesis)} \\
& + 1 + 2 + \ldots + (n-1) + n \\
=& \frac{(n-1)n(n+1)}{6} + \frac{n(n+1)}{2} && \text{(by Problem 82)} \\
=& \frac{n(n+1)(n+2)}{6}.
\end{aligned}
$$

Practice proofs by induction on the following problems.

Problem 85 Prove that the following equalities hold for every $n > 0$.
1. $1 + 3 + 5 + \cdots + (2n - 1) = n^2$ (for a geometric proof of this formula see Figure 3.11);
2. $1 \cdot 2^1 + 2 \cdot 2^2 + 3 \cdot 2^3 + \cdots + n \cdot 2^n = (n-1) \cdot 2^{n+1} + 2$;
3. $1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{2^n} \geq \frac{n}{2} + 1$.
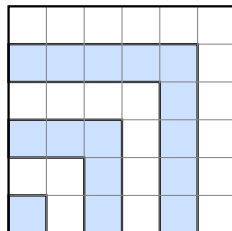
Figure 3.11: As it often happens with induction proofs, the induction proof convinces us that the formula is correct, but gives no clue where the formula is actually coming from. For this particular case, there is again a neat geometric visualization of the formula.

3.2.2 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. For this, we can just start our induction base at $n = c$ rather than at $n = 1$.

Problem 86 Prove that $2^n \geq n^3$ for all $n \geq 10$.

The following plots show that this statement is not even true for $n < 10$.

```python
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')
```
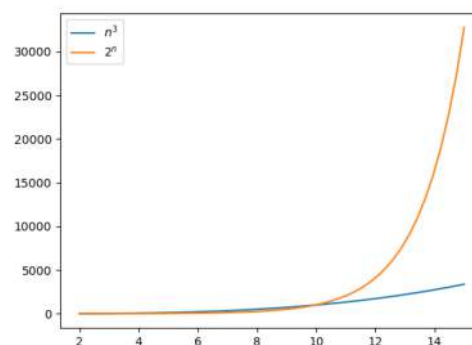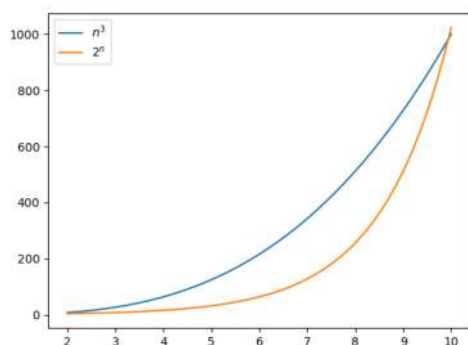

We can prove the statement from Problem 86 for all $n \geq 10$ using mathematical induction. The base case of $n = 10$ is easy to check: $2^n = 1024 > 1000 = n^3$. For the step from n to $n + 1$, the left-hand side is multiplied by 2, but the right-hand side is multiplied by

$$\frac{(n+1)^3}{n^3} = \left(1 + \frac{1}{n}\right)^3.$$

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 87 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.
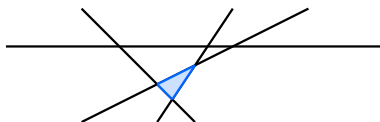
We will prove this statement by induction on the number of lines. The base case is when there are only three lines. Since the lines intersect at three different points, they must form a triangle.
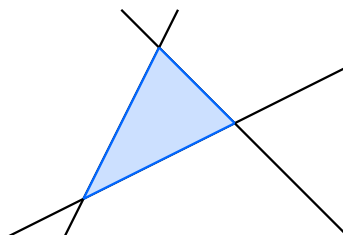
Indeed, the only case when three lines do not form a triangle — when they intersect at one point — is forbidden.
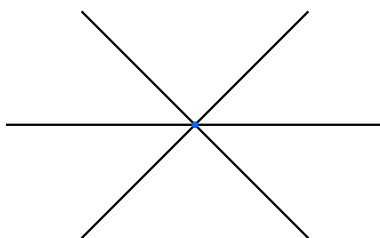
In order to prove the induction step, we start with $n \geq 3$ lines (which, by the induction hypothesis, already form a triangle T) and add one more line L. Consider two cases:

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.
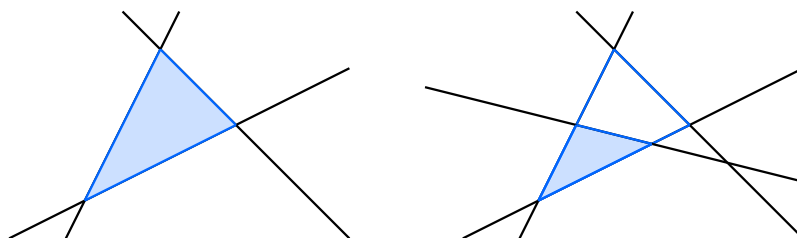
Case 2: L does not intersect T. In this case, T remains intact, so the new picture still has the triangle T.
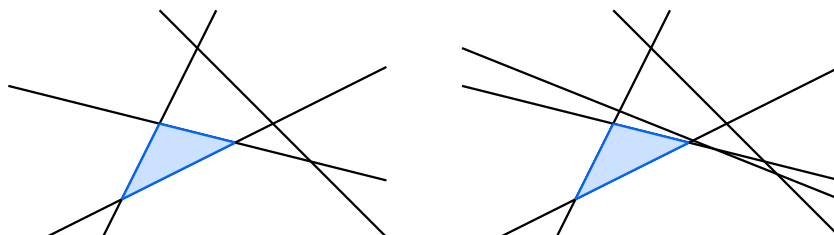
3.2.3 Proving Stronger Statements May Be Easier!

Recall from Section 3.1.5 that the number of moves in the Hanoi Towers puzzle with n discs is given be the recurrent formula

$$T_n = \begin{cases} 2T_{n-1} + 1 & \text{for } n > 1; \\ 1 & \text{for } n = 1. \end{cases}$$

A simple inductive arguments shows that $T_n = 2^n - 1$ for every $n \geq 1$. The base case holds as $T_1 = 1 = 2^1 - 1$. The inducttion step follows from

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

We have an inductive 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 inductive proof one needs to strengthen the statement. The trick here is that this also allows one to use a *stronger inductive hypothesis*. And, having a stronger hypothesis, one has more tools to prove stronger statements.

> **Problem 88** Prove that
>
> $$1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \cdots + \frac{1}{99} - \frac{1}{100}$$
> $$= \frac{1}{51} + \frac{1}{52} + \cdots + \frac{1}{100}$$

We will solve a more general problem:

> **Problem 89** Prove that for every $k \geq 1$:
>
> $$1 - \frac{1}{2} + \cdots + \frac{1}{2k-1} - \frac{1}{2k}$$
> $$= \frac{1}{k+1} + \frac{1}{k+2} + \cdots + \frac{1}{2k}$$

This will imply Problem 88 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

$$\frac{1}{k+2} + \frac{1}{k+3} + \cdots + \frac{1}{2k} + \frac{1}{2k+1} + \frac{1}{2(k+1)}$$
$$- \left(\frac{1}{k+1} + \frac{1}{k+2} + \frac{1}{k+3} + \cdots + \frac{1}{2k} \right)$$
$$= \frac{1}{2k+1} + \frac{1}{2(k+1)} - \frac{1}{k+1}$$
$$= \frac{1}{2k+1} - \frac{1}{2(k+1)}.$$

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 90 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.

```python
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')
```
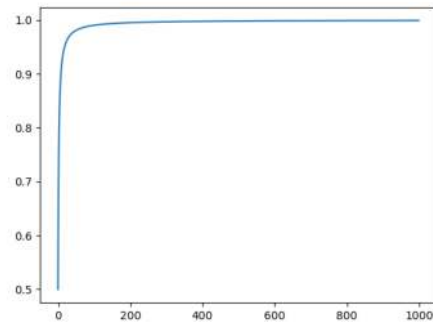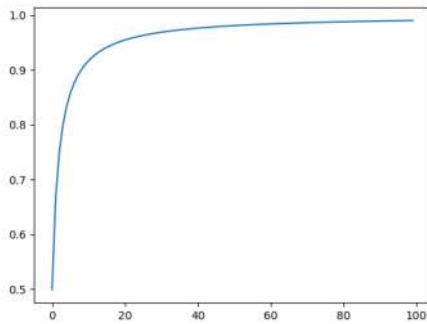

Now, let us prove an even stronger statement: not only is this sum less than one, but it actually equals $1 - \frac{1}{n+1}$. We will use mathematical induction to prove this stronger statement. When $n = 1$,

$$\sum_{i=1}^{n} \frac{1}{i(i+1)} = \frac{1}{2} = 1 - \frac{1}{n+1}.$$

Now if we assume that this holds for some $n \geq 1$, then for $n + 1$ we have

$$\sum_{i=1}^{n+1} \frac{1}{i(i+1)}$$

$$= \sum_{i=1}^{n} \frac{1}{i(i+1)} + \frac{1}{(n+1)(n+2)} \qquad \text{(by ind. hypothesis)}$$

$$= 1 - \frac{1}{n+1} + \frac{1}{(n+1)(n+2)}$$

$$= 1 - \frac{1}{n+2}$$

3.2.4 What Can Go Wrong with Induction?

Problem 91 The Fibonacci sequence

$$0, 1, 1, 2, 3, 5, 8, \ldots$$

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
>
> $$\begin{aligned} F_n &= F_{n-2} + F_{n-1} \\ &\geq 2^{\frac{n-2}{2}} + 2^{\frac{n-1}{2}} \qquad\qquad\qquad \text{(by ind. hypothesis)} \\ &= 2^{\frac{n-2}{2}} \cdot (1 + \sqrt{2}) \\ &\geq 2^{\frac{n-2}{2}} \cdot 2 \\ &= 2^{\frac{n}{2}}. \end{aligned}$$

The issue with this solution is quite subtle: we prove the base case only for one value of n, but we use it as 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 inductive 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 92 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 $p_1, p_2, \ldots, p_{n+1}$. By the induction hypothesis, there is a line L_1 containing the points $p_1, p_2, \ldots, p_n$. Similarly, there is a line L_2 containing the points $p_2, \ldots, p_{n+1}$. Since L_1 and L_2 both contain all the points $p_2, p_3, \ldots, p_n$, 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, we can indeed find such a line by the induction hypothesis. Similarly, we can indeed find L_2. If these two lines intersected at least two (distinct) points, then they would indeed coincide. They intersect at $n - 1$ points: $p_2, \ldots, p_n$. But for $n = 2$, this is just one point in the intersection. 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 inductive 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 93 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 first take 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 too. 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.

This problem is similar to the previous one. Try to carefully examine what assumptions the induction step proof makes, and find a mistake in this proof.

> **Problem 94** 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 $\leq n$, 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 $n + 1 = 1$.

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 online! ⬀).

> **Problem 95** What 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 96** Find a formula for the sum of all odd numbers from 1 to $2n - 1$: $1 + 3 + 5 + \cdots + (2n - 3) + (2n - 1)$:
> - n^2
> - $n(n + 1)$
> - $n(2n - 1)$
> - $3n - 2$

> **Problem 97** What is the value of the sum $\frac{1}{1 \cdot 2} + \frac{1}{2 \cdot 3} + \frac{1}{3 \cdot 4} + \cdot + \frac{1}{99 \cdot 100}$?

> **Problem 98** 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 99 Function T 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 100 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 $\leq n$ to $n+1$, 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 101 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 102 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, but 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 103** For which of these statements one *example* is enough to prove them? For which of them one *counterexample* is 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. In order to prove the second statement, we could just demonstrate one *example*: a white lion. 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 104 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*: $2^{16} = 65\,536$.

```python
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 105 Does there exist a power of 2 that starts with 17?

Interestingly, a much stronger statement is true: 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 106 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 online! ⬀

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.

```python
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
```

```python
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 $2 \le d < n$ (recall that $n = 1$ is not prime by convention). We then go through all $1 \le n < 100$ 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.

```python
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.

```python
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 $1 \le a \le b \le c < 20$ (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 107 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 prove that there exist such numbers. 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 108 — Sum of powers conjecture. In 1769, Euler conjectured that for any integer $n > 2$, one needs at least n summands to represent the nth power of a positive integer as a sum of the nth 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 109 — 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 nth powers are required to sum to an nth 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 his counterexample in 1972 — 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.

```python
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 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 if 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`-statemenets and `while`-loops, and, thus, "give directions" to

our computer programs.

```python
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 110** 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 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:

```python
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 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 indeed pays Bob $100. In this case Alice keeps 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, therefore S is true!
- There is an elephant in the refrigerator, but Alice does not pay Bob. In this case Alice does not keep her promise, so S is certainly false.
- There is no elephant, but Alice still decided 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 T is true or U 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, because $n = 6$ indeed *implies* that n is even. In other words, the only way to have this If-Then statement false is to have $n = 6$ and n not even, which is impossible.
- "If $n = 5$, then n is even" is false, because for $n = 5$, n is *not* even.
- "If $1 = 2$, then $2 = 3$" 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 in the case when the If-part is not satisfied.)
- "If $1 = 2$, then I am an elephant" is true as well for the same reason!

Problem 111 Is the following statement true: "If pigs can fly, then $2 \times 2 = 4$"?

Yes, when the hypothesis ("if pigs can fly") is false, then 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 satetement is true.

Problem 113 Is the following statement true: "If $1 = 2$, then I'm the Pope"?

Problem 114 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. Say, 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, therefore, 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 115 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".

Figure 4.5: Augustus De Morgan (1806–1871). (Source: Wikipedia ⌐.)

Similarly, the negation of an OR is an AND of negations:

$$\neg(S \vee T) = \neg S \wedge \neg T \, .$$

Problem 116 What is the negation of the statement "Either he will do it on time, or he won't get paid"?

We negate each of the parts in 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 117 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 118 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 statements holds at least for some instance:
- *There exist* black swans.
- *One can* 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 119 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 looks as follows: "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...") 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 $\forall x\, \neg P(x)$.

```python
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 120 Alice says that all white lions weigh more than 100kg. Bob wants to say that she is wrong (no less, no more). What should he say?

In order to negate this universal quantifier, we use the existential quantifier: "There are some white lions that weigh at most 100kg".

Problem 121 Alice says that all elephants are tall and heavy. Bob wants to say that she is wrong (no less, no more). What should he say?

First, we change the universal quantifier to the universal 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:"Some elephants are short or lightweight".

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 online! ⧉).

Problem 122 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. What does he need to do? (Note that in order to answer this question you do not need to know what an ordinal is, or when it is 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 123 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 124 Alice claims that every student in the 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 125 Alice claims that every student in the group knows either French or 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 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?

Problem 128 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.

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. Its version was used in *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 believes are inconsistent.

Problem 129 There are boys and girls in a class. Some of them study French, others study German. Prove that there are a boy and a girl studying different languages. Try it online! ⬀

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; mission would also be impossible, if nobody studied, say, French.

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.

For 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). From our assumption, every boy in the class must study French, too. Since there is at least one boy, every girl (but not just Alice) must study French, contradicting the assumption that somebody studies German.

4.3.1 Balls in Boxes

Problem 130 Is it possible to place 30 balls in ten boxes so that all boxes contain different number of balls? Try it online! ⬈

If you have tried to place 30 balls in ten boxes, perhaps you see an obstacle: in a sense, we have too few balls. To make this formal, assume that this mission is possible. Let us place the boxes in the order of increasing number of balls and let us denote these numbers by $b_0, \ldots, b_9$ (this 0-based indexing, used in programming constantly, will prove useful in this problem):

$$b_0 < b_1 < \cdots < b_9\,.$$

What do we know about b_0? Almost nothing: b_0 is non-negative (that is, $b_0 \geq 0$), but no more than that. 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 $i = 0, 1, \ldots, 9$. This means that

$$b_0 + b_1 + \cdots + b_9 \geq 0 + 1 + \cdots + 9 = 45\,,$$

contradicting the fact that we have 30 balls.

4.3.2 Numbers in Tables

Problem 131 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 other cells with integers in such a way that consecutive numbers differ by at most 3? Try it online! ⬈

By trying to do this, you'll notice that, in a sense, the numbers grow too slow to get 30 in 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	≤ 4	≤ 7	≤ 10	≤ 13	≤ 16	≤ 19	≤ 22	≤ 25	30

It reveals that the mission is impossible: the ninth integer is at most 25, hence, it differs from the last integer by more than 3.

Problem 132 There is a sequence of 20 cells with 1 and 50 at the ends.

1																			50
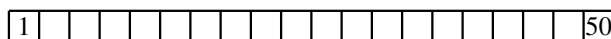
We would like to place integers into the cells so that the neighboring cells differ by at most k. What is the minimum k that such a construction exists? Try it online! ⌲

Consider a more challenging problem of a similar flavor.

Problem 133 Is it possible to place integers $1, 2, \ldots, 64$ on the chessboard in such a way that neighbors (sharing a side) differ by at most 4? Try it online! ⌲

In this case, the sequence of cells is not linear, but two-dimensional. Still, one can apply the above idea in this case, too! For this, consider the two cells containing integers 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 same column and at most seven vertical moves are enough to reach the same row. See Figure 4.6 for an example.
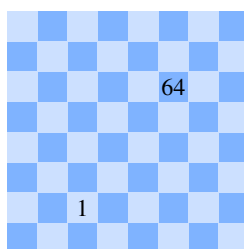

Figure 4.6: In this case, one can reach 64 from 1 in just seven moves: three moves to the right and four moves up. In general, at most 14 moves are enough.

And this shows that our mission 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.
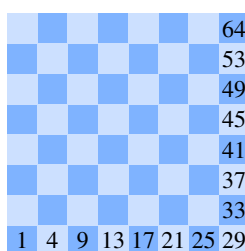
Figure 4.7: An attempt to place integers on a shortest path from 1 to 64. All the neighboring integers differ by four except for the last pair: $64 - 53 = 11 > 4$.

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 134 — Antimagic square. Is it possible to fill a 3×3 grid by integers $-1, 0, 1$ so that the sums of each row, each column, and both diagonals are different? Figure 4.8 gives an example. Try it online! ⬀

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 each of these integers: $-3, -2, -1, 0, 1, 2, 3$.

Problem 135 There are n persons at a party, some of them had shaken hands. Prove that there are two persons who had shaken equal number of hands. Figure 4.9 gives an example.
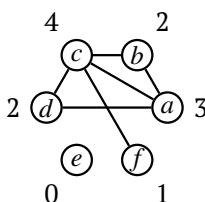

Figure 4.9: There are $n = 6$ people at a party: a, b, c, d, e, f. We join two persons if they shook hands. Near each person, her number of handshakes is shown. The persons b and d shook equal number of hands.

We have n integers (the number of handshakes for each of n persons) and each of these integers may take values $0, 1, \ldots, n-1$. The pigeonhole principle does not seem to be applicable here: it is perfectly fine if each of the integers $0, 1, \ldots, n-1$ 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 a person who shook no hands, whereas the integer $n-1$ corresponds to a person who shook all other hands. But these two cases cannot happen simultaneously! Indeed, if a person 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 its opposite is true and derives a contradiction.
- This is one of the most popular proof arguments.
- It is usually combined with other proof ideas.

Problem 136 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 online! ↗

Problem 137 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 online! ↗

Problem 138 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 online! ↗

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 an 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 139** Fill a 3×5 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 online! ↗

Let us see whether we are lucky enough to do this at the first try. For 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	$\rightarrow$ 20
5	5	1	6	3	$\rightarrow$ 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	$\rightarrow$ 20
5	5	1	6	3	$\rightarrow$ 20
3	1	2	1	3	

$\downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow$
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 mission 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. This trick will be useful in the next problem.

> **Problem 140** 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 online! ⬀

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 141** 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 online! ⬀

> **Problem 142** 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 online! ⬀

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 a 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 143** Four chess players played 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 that finishes the 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 online! ⬀

> **Problem 144** 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 145 — 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   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. To illustrate the main idea, we start with the following toy problem.

> **Problem 146** 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 online! ⌕

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! Figure 5.1 helps to realize this. 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.
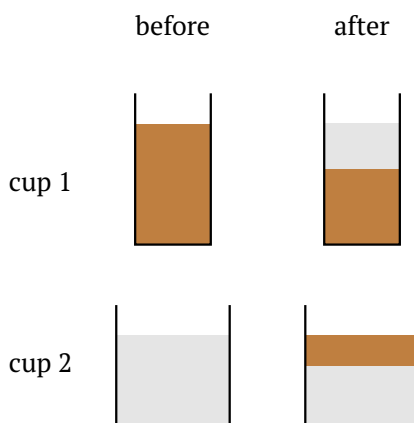

Figure 5.1: Two cups in Problem 146 before and after.

> **Problem 147** There are two equally sized cups: cup 1 contains coffee and cup 2 contains milk. Both cups are half full. Your favorite drink is $1/3$ coffee and $2/3$ milk. Can you get such a drink in cup 1 by pouring (any amounts of) drinks from one cup to another? Any amount of your favorite drink would work, the right proportion is what matters. Try it online! ⌕

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 amounts of coffee and milk are equal (and they will be 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 148** 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 online! ⧉

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:

$$\#\text{pending} = 1 + 2 \cdot \#\text{fixed}.$$

Hence, the answer to this problem is $1 + 2 \cdot 15 = 31$.

It is also straightforward to model this process in Python.

```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 149**
> 1. You have 10\$ in each of ten different banks. You would like to merge all accounts into one. For this, each day you pick two banks where you still have money left and move all 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 online! ⧉

5.3 Termination

5.3.1 Football Fans

> **Problem 150** 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 forever? (For the sake of this problem, we assume that the friendship is always mutual and that both the population and the friendship connection do not change.) Try it online! ⧉

Figure 5.2 shows a toy example with seven citizens. Two persons are friends if they are connected

Figure 5.2: A toy example of switching teams: on the first day, person 6 switches; on the second day, person 1 switches.
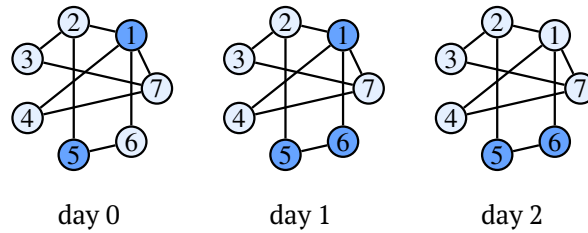
by a segment in the picture (for example, persons 3 and 7 are friends whereas persons 2 and 6 are not). Initially (day 0), persons 1 and 5 support the first team (call it dark) and all others support the second team (call it light). On the first day, person 6 realizes that all of her friends (persons 1 and 5) support the dark team and starts to support it, too. On the second day, person 1 switches to support the light team as he has three friends supporting the light team (persons 2, 4, and 7) whereas only one of his friends (person 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. For this, we find a quantity that decreases each time a person switches. Since this quantity is a non-negative integer, 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 5.3 repeats Figure 5.2 with shaky connections shown dotted. As you see, the number of shaky connec-
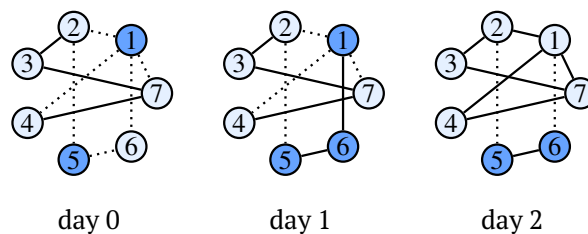

Figure 5.3: Population from Figure 5.2 with shaky connections shown dotted.

tions drops down: 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 connections, when a person switches, the number of shaky connections decreases.

To show this, consider a person 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. But this means, that after the switch, the number of shaky connections decreases. See Figure 5.4.

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 code that visulizes the switching process for any initial population. We encourage you to fool around with its parameters (size of population, number of friends) and see what happens.
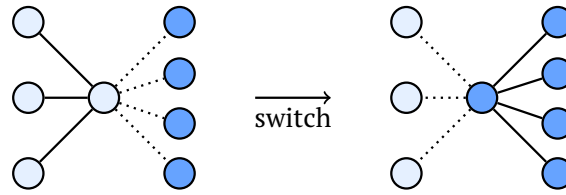
Figure 5.4: When a person switches, the number of shaky connections decreases.

5.3.2 Arthur's Books

In the following problem, we will use invariants twice — to show that something can be done efficiently and to show that this cannot be done too efficiently.

> **Problem 151** King Arthur has a shelf of his works consisting of 10 volumes, numbered $1, 2, \ldots, 10$. 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 online! ⧉
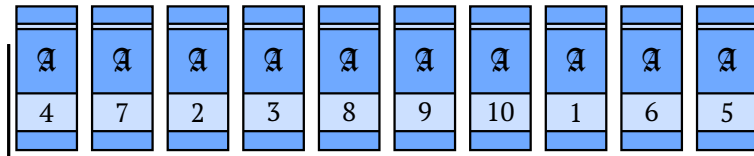
Let us show that nine days are enough for any initial permutation of the volumes. For this, on the first day, Merlin finds the position j of volume 1 (in the example above, $j = 8$). If $j \neq 1$, he exchanges the volumes at the positions 1 and j.
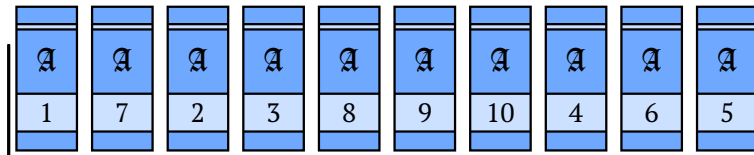
On the second day, he finds the position j of volume 2. Note that j cannot be equal to 1, since the first position is occupied by volume 1. If $j \neq 2$, Merlin exchanges the books at positions 2 and j.
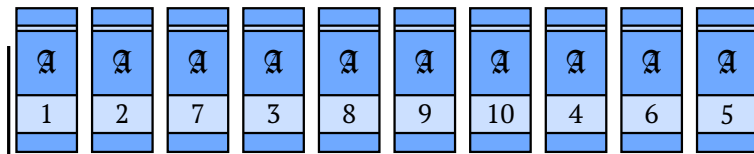
Proceeding in the same fashion, on day i, Merlin ensures that volume i stays at position i. This way, he maintains the following *invariant*: after i days, the first i volumes stay at their intended positions. It remains to note that by the end of the ninth day, volume 10 must be at position 10: indeed, volumes $1, 2, \ldots, 9$ are at positions $1, 2, \ldots, 9$, 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.

```python
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 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.
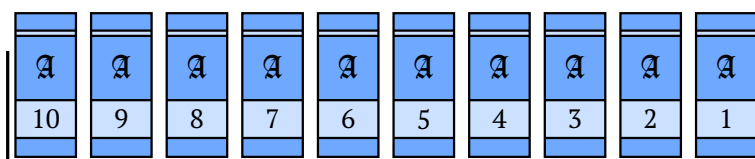
However, it can be sorted in just five days:

$$10 \leftrightarrow 1, \quad 9 \leftrightarrow 2, \quad 8 \leftrightarrow 3, \quad 7 \leftrightarrow 4, \quad 6 \leftrightarrow 5.$$

Consider the following permutation.
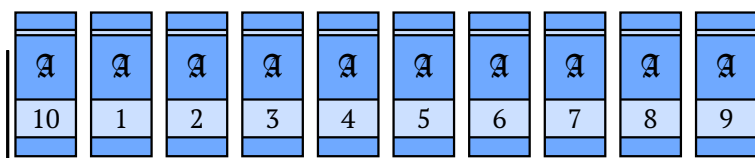
Try to sort it manually or to run the code above on it (don't forget to subtract one from the volume numbers, as the code assumes 0-based indexing). Note that if you (and the function sort_books) are not able to sort it in fewer than nine days, this *does not* mean that it is impossible: it is conceivable that some other tricky strategy does the job in, say, eight days. Below, we exploit invariants to prove formally that no such strategy exists.

Consider the following integer parameter of a permutation π: $\text{inv}(\pi)$ is the number of elements staying to the right of their intended positions. The following example shows such elements in the presented permutations.

$$\text{inv}([1,2,3,4,5,6,7,8,9,10]) = 0,$$
$$\text{inv}([10,9,8,7,6,5,4,3,2,1]) = 5,$$
$$\text{inv}([3,1,4,8,6,10,5,9,2,7]) = 4,$$
$$\text{inv}([10,1,2,3,4,5,6,7,8,9]) = 9.$$

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 should be at least the number of highlighted elements. We make this formal below.

We claim that a permutation π cannot be sorted in fewer than $\mathrm{inv}(\pi)$ days. Indeed, when π is sorted, $\mathrm{inv}(\pi) = 0$ (when all volumes stay at their intended places, no volume stands to the right of its intended position). Each day, the value of $\mathrm{inv}(\pi)$ drops by at most one. Indeed, each day, one book moves to the right and one book moves to the left. Only the latter book may contribute to reducing the value of $\mathrm{inv}(\pi)$.

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. Indeed, in this example, initially $\mathrm{inv}(\pi) = 9$, it must equal zero in the end, and it reduces by no more than one each day.

5.4 Even and Odd Numbers

Integers divisible by 2 are called *even* (examples: 6, -8, 0), all others are called *odd* (examples: -3, 7, 19). This basic property of integers, called *parity*, finds applications in many problems.

Problem 152 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 online! ⬀
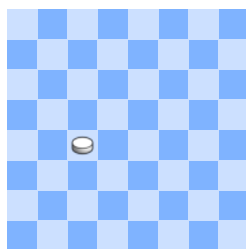
The first thing to note is that the piece can return to its original position in principle: just move one step (say, to the right) and then move back. This means that returning back in 18 steps is also possible: repeat the previous strategy nine times.

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.

By moving the cell, you might have noticed that with each move, the color of the current cell changes. If the piece starts at a light cell, in one move 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 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 153   • 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 online! ⬀

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 is even, see Figure 5.5.

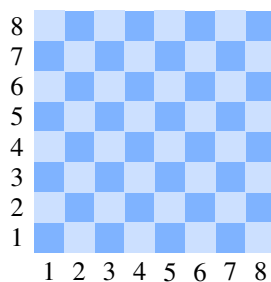Now, let's consider a more challenging problem.

Figure 5.5: The cell $(2,3)$ is light, since $2+3 = 5$ is odd. The cell $(3,5)$ is dark, since $3+5$ is even.

Problem 154 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 online! ↗

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 155 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 156 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 $1 \pm 2 \pm \cdots \pm n$.

```python
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):
    print(v, represent(9, v))
```

```
7 [1, -2, -3, 4, 5, -6, 7, -8, 9]
15 [-1, -2, 3, 4, -5, 6, -7, 8, 9]
22 False
33 [1, -2, 3, -4, 5, 6, 7, 8, 9]
```

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 3.2.3).

Problem 157 All cells of a 4×4 table contain plus signs except for the top left cell that contains a minus sign.

In one step, one is allowed to switch signs in any row or column. Is it possible to switch all signs to '+'? Try it online! ⤢

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. Hence, the parity of the number of minuses does not change. The following picture shows two examples. In the first one, the number of minuses does not change, in the second one, it changes by 2.

After this warm up, consider a more challenging version of the problem.

Problem 158 Is it possible to switch all signs to '+' in the following table?

Try it online! ⤢

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 2×2 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 4×4 table, we also switch a row in this 2×2 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 2×2 subtable, and this is why the mission is impossible.

After discussing two versions of this puzzle, it is natural to ask the following general question.

Problem 159 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 2×2 subtable of a cofiguration contains an odd number of minuses, then it is unsolvable. It turns out that it is the only obstacle!

Problem 160 Prove that if every 2×2 subtable of a configuration contains an even number of minuses, then one can switch all signs to pluses.

A hint: show that by placing pluses in the top row and left column one gets the desired configuration (that is, a table full of pluses).

+	+	+	+
+			
+			
+			

Finally, to hone your skills of discovering parity invariants, attempt the following problem.

Problem 161 Consider the standard coloring of an 8×8 chessboard. In one step, one is allowed to pick a 2×2 subboard and tswitch the colors of all its cells. Is it possible to get a board containing a single cell of a particular color? Try it online! ⤢

Summary

- Invariants are important tools for proving impossibility, termination, and various bounds.
- Invariants may take many forms: numbers, "being even or odd", equations, inequalities.
- 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 in order 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 4×4 box and put 15 square pieces numbered $1, 2, \ldots, 15$ (Figure 6.1) in it. Each piece is of size 1×1, 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.
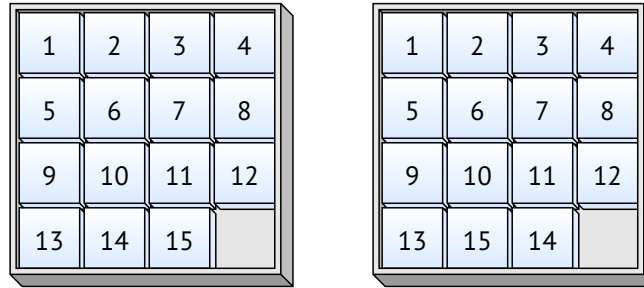
Figure 6.2: Can you transform the left configuration into the right one by several moves (not taking the pieces out)? The wikipedia ⬀ article says that once it was a $1 000 question.

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 online! ⬀.

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 proof is the final goal of 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.

$$\boxed{S\,|\,T\,|\,O\,|\,P} \rightarrow \boxed{S\,|\,P\,|\,O\,|\,T}$$

Now, we can exchange some other pair, for example, S and P, and get PSOT.

$$\boxed{S\,|\,P\,|\,O\,|\,T} \rightarrow \boxed{P\,|\,S\,|\,O\,|\,T}$$

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 O.

Stop and Think!   Can you convert STOP to TOPS by a sequence of transpositions (pair exchanges)?

Here is one such sequence.

$$\boxed{S\,|\,T\,|\,O\,|\,P} \rightarrow \boxed{T\,|\,S\,|\,O\,|\,P} \rightarrow \boxed{T\,|\,O\,|\,S\,|\,P} \rightarrow \boxed{T\,|\,O\,|\,P\,|\,S}$$
$$0\ 1\ 2\ 3 \qquad 0\ 1\ 2\ 3 \qquad 0\ 1\ 2\ 3 \qquad 0\ 1\ 2\ 3$$

Let us 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.

```python
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 stoed 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 pairs of letters).

> **Problem 162** 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 online! ⧉

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.

$$\boxed{M\,A\,R\,I\,N\,E} \;\rightarrow\; \boxed{A\,M\,R\,I\,N\,E}$$
$$0\ 1\ 2\ 3\ 4\ 5 \qquad\quad 0\ 1\ 2\ 3\ 4\ 5$$
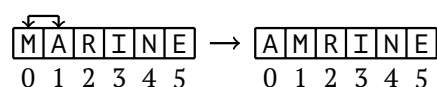
Then we fill position 1 with letter I by applying the transposition (1,3):

$$\boxed{A\,M\,R\,I\,N\,E} \;\rightarrow\; \boxed{A\,I\,R\,M\,N\,E}$$
$$0\ 1\ 2\ 3\ 4\ 5 \qquad\quad 0\ 1\ 2\ 3\ 4\ 5$$
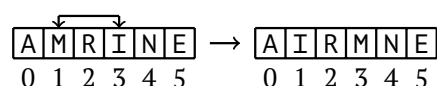
Now we are lucky: positions 2 and 3 are already filled correctly by R and M. But position 4 contains N instead of E, so we need to apply transposition (4,5). After that, position 4 contains E as required. Moreover, the last position 5 is also filled correctly: N was moved there from position 4, so nothing 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:

$$\boxed{\texttt{M}\,\texttt{A}\,\texttt{R}\,\texttt{I}\,\texttt{N}\,\texttt{E}} \longrightarrow \boxed{\texttt{M}\,\texttt{A}\,\texttt{R}\,\texttt{I}\,\texttt{E}\,\texttt{N}}$$
$$0\ 1\ 2\ 3\ 4\ 5 \qquad\qquad 0\ 1\ 2\ 3\ 4\ 5$$
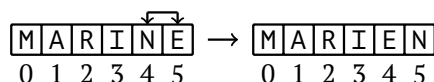
Then we make position 3 correct:

$$\boxed{\texttt{M}\,\texttt{A}\,\texttt{R}\,\texttt{I}\,\texttt{E}\,\texttt{N}} \longrightarrow \boxed{\texttt{I}\,\texttt{A}\,\texttt{R}\,\texttt{M}\,\texttt{E}\,\texttt{N}}$$
$$0\ 1\ 2\ 3\ 4\ 5 \qquad\qquad 0\ 1\ 2\ 3\ 4\ 5$$

Position 2 is already correct, so we make position 1 correct:

$$\boxed{\texttt{I}\,\texttt{A}\,\texttt{R}\,\texttt{M}\,\texttt{E}\,\texttt{N}} \longrightarrow \boxed{\texttt{A}\,\texttt{I}\,\texttt{R}\,\texttt{M}\,\texttt{E}\,\texttt{N}}$$
$$0\ 1\ 2\ 3\ 4\ 5 \qquad\qquad 0\ 1\ 2\ 3\ 4\ 5$$

> **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 will cancel out.

> **Problem 163** Prove that *any* permutation can be obtained by a sequence of transpositions.

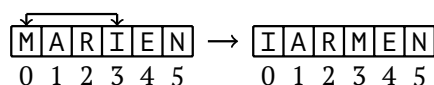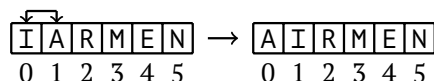In fact, our approach to the AIRMEN example was general enough and it will work for every permutation. We fill all the positions one by one in some order (in our example, from left to right), until all the positions are filled correctly.

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 $0, 1, \ldots, k-1$ (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 yes, 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 $0, 1, \ldots, k-1$ 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.[1]

---

[1] 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 $0, 1, \ldots, k-1$ 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 $0, 1, \ldots, k-1$ while doing this? No, because they are filled by the letters whose target positions are $0, 1, \ldots, k-1$, and we consider a letter whose target position is k, and its current position is also greater than $k-1$.

Problem 164 Prove that any permutation of n objects can be obtained by at most $n-1$ transpositions.

Problem 165 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 166 Implement a Python function

        transform(first, second)

that takes two permutations of $\{0, 1, \ldots, n-1\}$ (i.e., two lists containing each integer from $\{0, 1, \ldots, n-1\}$ 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.

```python
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 $0, 1, \ldots, n-1$ 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 $i = 0, 1, \ldots, n-1$, we fill the i-th place 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`.

Problem 167 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 may 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 168 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.

```python
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:

$$AB \to BA \to AB \to BA \to \ldots$$

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 6 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):

$$\boxed{ABC} \quad \boxed{ACB} \quad \boxed{BAC} \quad \boxed{BCA} \quad \boxed{CAB} \quad \boxed{CBA}$$

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):
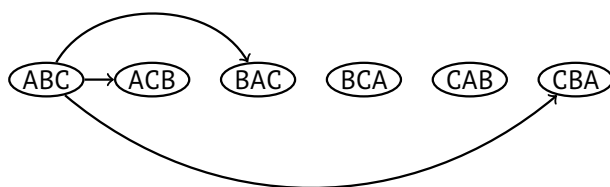
Stop and Think! Since we need to study the effect of multiple transpositions, we should add arrows starting from other orderings too. Can you do this?

This requires some patience, but not much: for each vertex (three-letter string), we have three outgoing edges that correspond to three possible transpositions:
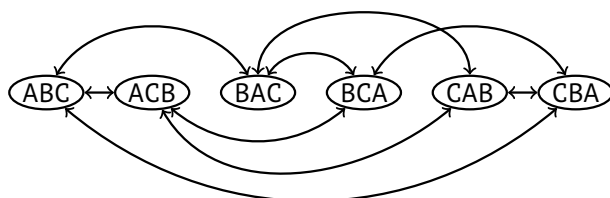
Stop and Think! In the picture above, all edges are bidirectional — can you explain why this happens? How many edges does each vertex have? Why?

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 $X \to Y$ is accompanied by an arrow $Y \to X$. 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?

For 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:

$$1, 3, 5, 7, \ldots \text{ steps:}\qquad \text{ACB, BAC, CBA}$$
$$2, 4, 6, 8, \ldots \text{ steps:}\qquad \text{ABC, BCA, CAB.}$$

This behavior becomes even more clear if we reposition the vertices of our graph keeping the same edge connections:
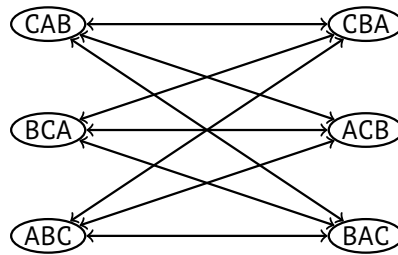
(Check that we do not cheat and connections are the same!)

> **Stop and Think!** Do you see why we come back to the original configuration only after an even number of moves?

Now, it is easy: 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 169** 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,

$$\boxed{S\,|\,T\,|\,O\,|\,P} \to \boxed{S\,|\,P\,|\,O\,|\,T}$$
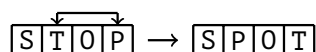
is not a neighbor transposition: we exchange T and P that are not neighbors (there is O between them). On the other hand, our second example

$$\boxed{S\,|\,P\,|\,O\,|\,T} \to \boxed{P\,|\,S\,|\,O\,|\,T}$$

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?

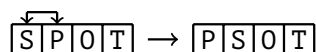This is an easy question — just to check that we are on the same page: i and j should differ by 1; one can write this condition as $|i - j| = 1$. Note that we require $i \neq j$ for every transposition (we cannot exchange a letter with itself).

> **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 be a sequence of *neighbor* transpositions?

The answer is yes, and we will see why soon. But let us first look at our example.

> **Problem 170** Represent the permutation STOP $\to$ 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:

$$\boxed{S\,|\,T\,|\,O\,|\,P} \to \boxed{S\,|\,O\,|\,T\,|\,P} \to \boxed{S\,|\,O\,|\,P\,|\,T} \to \boxed{S\,|\,P\,|\,O\,|\,T}$$
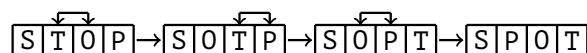
In this example, we had one letter between the letters T and P that we neede to exchange. The same trick works for cases with several letters in between, see Figure 6.3.
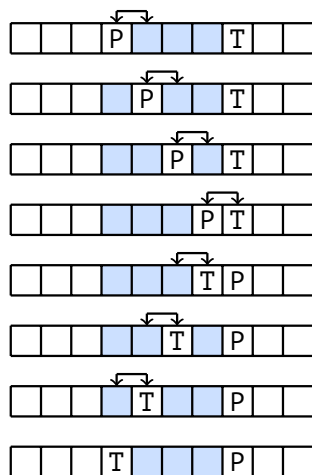
Figure 6.3: Representing an arbitrary transposition as a sequence of neighbor transpositions. The intermediate elements are shaded: P and T jump over them.

> **Problem 171** 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 172 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 ordering 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 heavy 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 173 Find a sequence of *neighbor* transpositions that converts MARINE to AIRMEN. Try it online! ⌧

Problem 174 Implement a Python function

```
transform(first, second)
```

that takes two permutations of $\{0, 1, \ldots, n-1\}$ 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.

```python
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 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 remain 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 175**   Consider the following sequence of transpositions:
>
> $$\underline{\text{TA}}\text{STE} \to \text{AT}\underline{\text{STE}} \to \text{A}\underline{\text{TSET}} \to \underline{\text{A}}\text{ES}\underline{\text{TT}} \to \underline{\text{T}}\text{ES}\underline{\text{TA}} \to \text{TASTE}$$
>
> 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?

The trick is simple (even if not immediately obvious). 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.

Let us finish this section by a philosophical (but 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). So the argument we used to prove theorem 6.2.1 is valid only for this case. However, after this theorem is proved for objects that form a line, we may conclude that it is true for any objects: just create an imaginary line (curve) that goes through all of them.

6.2.4 Even and Odd Permutations

Recall the classification of permutations of n objects (different letters) for $n = 3$:
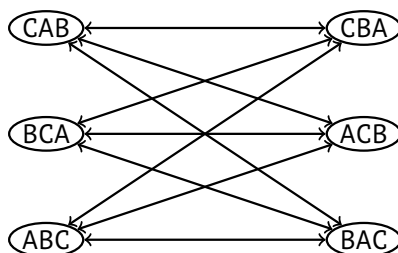
On the left, we have the original string ABC and all other strings that appear after an even number of transpositions; on the right, we have strings that are obtained by an odd number of transpositions. The permutations that correspond to the left part (that are obtained by an even number of transpositions) 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/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. How can we prove it?

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:
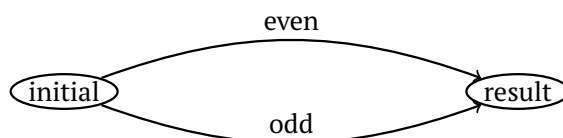
Each arrow denotes a sequence of transpositions; one contains an even number of transposition, and the other contains an odd number of transpositions.

▎**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.

even

initial   result

odd reversed
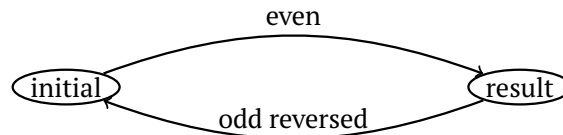
Since both sequences (even and odd) end at the same position, we get a consistent picture (no sudden changes in the middle of the combined video). In other words, we now have a sequence of transpositions that starts and ends in the same position, and the total number of transpositions is even + odd, which is odd. This is impossible (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., $0, 1, 2, \ldots, n-1$.

> **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 176** Implement a program that gets a list of length n consisting of the integers $0, 1, \ldots, n-1$ in some order, and returns `True` if and only if the corresponding permutation is even. Try it online! ⌐

This should be an easy task since we have already seen a 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. Can you invent such a method? (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

Now 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 that contains 0, see Figure 6.5 (left). 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. 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 (left). A move in the original game (middle) is now an exchange with the 0-piece (right).

> **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).

But there is another reasoning that 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). So the total number (left + right + up + down) is even (being equal to $2 \cdot \text{left} + 2 \cdot \text{up}$).

> **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 right lower 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/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 position into some other position 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 $1, 2, \ldots, 15$.

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. Imagine that the only allowed move during the meeting is when someone moves to an empty chair near her (so that others are not disturbed).

> **Stop and Think!**   Do you see how we can 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. 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!**   Do you see how this remark may help us? Recall that we need to bring a given piece in a 15-puzzle to the left upper corner.

When we speak about a cycle 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 (left).
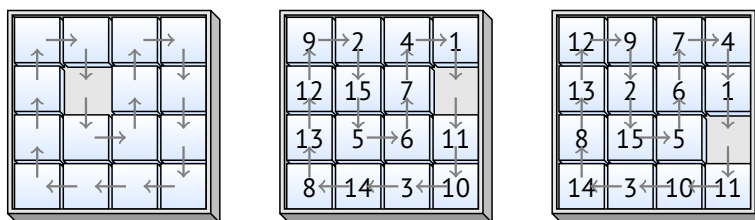

Figure 6.7: The left picture shows a cycle through all the cells of the board. The middle picture shows a particular configuration, whereas the right picture shows the result of a cyclic move applied to this configuration. To perform the cyclic move, one first moves 1 down (and the hole moves up), then 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 once making one step along the cycle, we need 15 moves.

> **Stop and Think!**   How many cycles are still needed (after this first one, Figure 6.7, right) to bring 3 into the top left corner?

Looking at the arrows, we see that after the first cycle (that moved 3 left) we need 4 more cycles to put 3 in the left upper 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 at the right of the hole after 4 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 then the following configuration (Figure 6.8; check the cycle ordering). Thus, on the right of the hole we have 1.
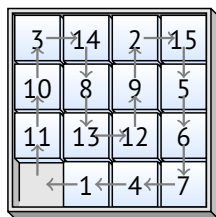

Figure 6.8: After five cycles 3 is in the top left corner and the cycle ordering remains the same.

6.4.3 Moving One More Piece

Now 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 much more modest: we know only how to fill the top left corner with a given piece. Still, it can be considered as 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. So 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.[2] 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.
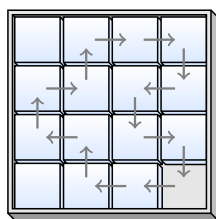

Figure 6.9: Using this cycle, we can move any piece (with one exception: left bottom corner) to the cell below the top left corner. Note the the cycle must include a hole to make the cyclic movement possible. In our picture the hole is shown in the right bottom corner, but any other place is possible (except the bottom left corner that is not in the cycle).

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.

---

[2]There is a simple reason why this is unavoidable, the same as for the chessboard tiling: going along the cycle, we alternate between white and black cells (in a chessboard coloring). To close the cycle, we should return to the initial cell, and this cannot happen after 15 moves (or any other odd number of moves).

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?
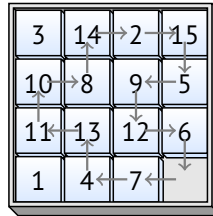

Figure 6.10: The top left corner is occupied by 3 forever; we want to bring piece 9 in the cell below the top left corner.

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 Fig. 6.11.
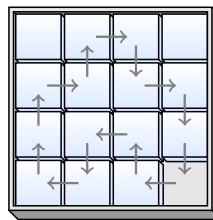
Figure 6.11: Using this cycle, we can move the piece from the bottom left corner (as well as others, except for the top right corner) to the place below the top left corner. Again, we assume that the cell outside the cycle (top right corner) is not a hole; if it is, we fill it by its neighbor.

6.4.4   Moving the Third Piece

Where are we now? A bit closer to our goal: we know how to fill *two* cells 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.

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).

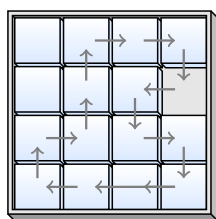**Problem 182** Show how one can fill the cell in the second row and second column, using the same technique.

However, this 20% or even 26% ($4/15 = 0.2666\ldots$) 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:
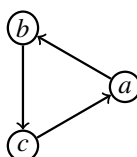
It is easy to see that a 3-cycle can be obtained by two transpositions and therefore is an even permutation.

Problem 183 Find two transpositions that (being performed sequentially) give the 3-cycle $a \to b \to c \to a$.

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.[3]

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

---

[3] 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.

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 permutations of $\{0, 1, \ldots, n-1\}$ and returns a list of 3-cycles needed to get the second list from the first one. 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).

```python
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


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 $0, 1, \ldots, n-1$, 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.

Now we developed a little 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 Sect. 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, left).
Also we know from Sect. 6.4.1 how to move the empty cell ("hole") in any direction, so we can put it near them (Figure 6.13, right)

> **Stop and Think!** Why don't we destroy the a-b-c corner while moving the hole?

This is indeed possible 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, for example, the shortest path).
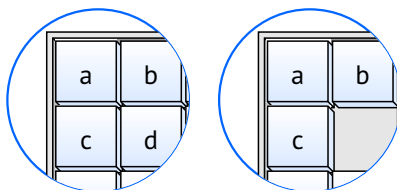
Figure 6.13: Three pieces a, b, c moved to the top left corner (left) with a hole near them (right).

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 2×2 square (Figure 6.14).
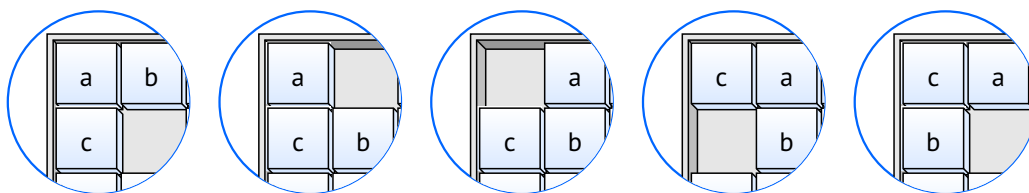
Figure 6.14: A 3-cycle performed inside a 2×2 square in the top left corner.

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.

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 in 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, and 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 position 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 position is now represented by a permutation of $0, 1, \ldots, 15$, and we assume (as the statement of Theorem 6.4.1 does) that in the initial position 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 a 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 cells, 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
>
> ```
> solve(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 online! ⤤

  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 b to the right of a;
      – move c below a;
      – move the empty cell (piece 0) below b;
      – make four moves to mimic the 3-cycle (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.)

To help you to solve this challenge, we show you a part of the reference solution below. Feel free either to extend it or to implement your own solution from scratch.

```python
def fancy_print(configuration):
    assert len(configuration) == 16
    for i in range(len(configuration)):
        if configuration[i] < 10:
            print(' ', end='')
        print(configuration[i], end=' ')
        if i % 4 == 3:
            print()
    print()


def to_2d_index(index):
    assert index in range(16)
    row = index // 4
    column = index % 4
    return row, column


def apply_transposition(configuration, transposition):
    i, j = transposition
    assert i != j
    assert configuration[i] == 0 or \
            configuration[j] == 0

    i2d, j2d = to_2d_index(i), to_2d_index(j)
    assert abs(i2d[0] - j2d[0]) + \
            abs(i2d[1] - j2d[1]) == 1

    configuration[i], configuration[j] = \
        configuration[j], configuration[i]


def cyclic_shift_to_start(configuration,
                          cycle_transpositions):
    for i in range(16):
        if not configuration[cycle_transpositions[i][0]]:
            return cycle_transpositions[i:] + \
                    cycle_transpositions[:i - 1]

    assert False


configuration = [
    9,  2,  4,  1,
    12, 15, 7,  0,
    13, 5,  6,  11,
    8,  14, 3,  10
]
fancy_print(configuration)

full_cycle = [
    (0, 4), (4, 8), (8, 12), (12, 13),
```

```
    (13, 14), (14, 15), (15, 11), (11, 7),
    (7, 3), (3, 2), (2, 6), (6, 10),
    (10, 9), (9, 5), (5, 1), (1, 0)
]
for transposition in cyclic_shift_to_start(full_cycle):
    apply_transposition(configuration, transposition)

fancy_print(configuration)
```

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

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

Here, the function `fancy_print` just outputs the given configuration in a 4×4 grid (you will use it a lot for debugging purposes!). The function `to_2d_index` converts an index of a cell into two indices — its row and column, see Figure 6.15. The next function, `apply_transposition`,

Figure 6.15: Indexing of pieces as well as row and columns of the board.

takes a configuration and a transposition (i.e., a pair of indices to be swapped) and applies the transposition. According to the puzzle rules, a transposition is a valid move if the two exchanged positions are neighbors on the board and one of them is an empty cell. We check that both of these conditions are satisfied and apply the transposition.

The function `cyclic_shift_to_start` takes as input a configuration and a list of transpositions of a cycle on the board and shifts this cycle so that it starts with the empty cell. E.g., to apply the cycle from the left part of Figure 6.7 to the configuration shown in the middle, one first moves 1, then 4, and so on.

Finally, the bottom part of the code shows an example of using these functions. We first set up a configuration (the same as in Figure 6.7). We then set up a snake cycle from Figure 6.7 as a sequence of transpositions. Finally, we apply this cycle and print the result.

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 complicated 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 4×4 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.
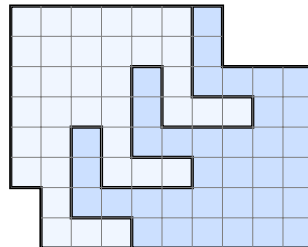

Figure 7.1: Solution to Problem 14.

7.2 Using SAT-solvers

In this section, we revisit Problem 60: 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! For 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:

- ($x_1 = 0$ or $x_2 = 1$);
- ($x_1 = 1$ or $x_2 = 1$);
- ($x_1 = 1$ 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. For 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 $x = 1, x_2 = 1$. 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).

```python
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.)

   For this, we will use n^2 0/1-variables: for $0 \le i, j < n$, $x_{ij} = 1$ 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 $0 \le i < n$, the i-th row contains at least one queen:

$$(x_{i1} = 1 \text{ or } x_{i2} = 1 \text{ or } \ldots \text{ or } x_{i(n-1)} = 1).$$

This ensures that there *at least n* queens on the board.

- For each $0 \le i < n$, the i-th row contains at most one queen. To state this, we use many clauses: for all $0 \le j_1 \ne j_2 < n$, we add a clause

$$(x_{ij_1} = 0 \text{ or } x_{ij_2} = 0).$$

  This ensures, that the are *at most n* queens. Together with the previous item, this gives that there are *exactly n* queens.
- For each $0 \le j < n$, the j-th column contains at most one queen: for all $0 \le i_1 \ne i_2 < n$, we add a clause

$$(x_{i_1 j} = 0 \text{ or } x_{i_2 j} = 0).$$

- 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

$$(x_{i_1 j_1} = 0 \text{ or } x_{i_2 j_2} = 0).$$

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 $0 \le i, j < n$ it gives a unique integer in the range $\{1, 2, \ldots, n^2\}$.

```python
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.

```python
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 = []
```

```python
        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 46 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)):

$$M \le 500, \quad D \le 200, \quad M + D \le 600.$$

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.

```python
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 61, the goal is to place 16 non-intersecting diagonals in a 5×5 grid. This can be done as follows. The grid has 36 nodes: let's index them with pairs of integers $(0,0), (0,1), \ldots, (5,5)$. 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. For 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.

```python
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 63 where the goal is to draw 21 non-intersecting diagonals of length $\sqrt{5}$ in a 6×6 grid.

7.4 Visualizing Football Fans

The code below produces a visulization for Problem 150. 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)
```

```python
    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')
```