

DISCRETE MATHEMATICS

for Computer Science

Marie Brodsky
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 three courses in the associated specialization at Coursera: [Mathematical Thinking in Computer Science](#), [Combinatorics and Probability](#), and [Number Theory and Cryptography](#). Future editions will cover the remaining two courses: [Graph Theory](#) and [Delivery Problem](#).

This book was last updated on April 12, 2023. There are 374 problems and 191 code snippets in the book. Most of the problems come with solutions and 165 of them are graded automatically (allowing you to get instant feedback). We're constantly working on extending and improving the book. Please ask questions, report typos, and suggest improvements through this [form](#). Get the latest version of the book at <https://leanpub.com/discrete-math>.

Contents

0	About the Book	7
0.1	Active Learning	7
0.2	Problem-based Learning	7
0.3	Python Programming Language	8
0.4	Acknowledgments	9
I	Mathematical Thinking in Computer Science	
1	Proofs: Convincing Arguments	13
1.1	Warm Up	13
1.2	Existence Proofs	20
2	Finding an Example	31
2.1	How to Find an Example	31
2.2	Optimality	39
2.3	Computer Search	49
3	Recursion and Induction	57
3.1	Recursion	57
3.2	Induction	75
4	Logic	93
4.1	Examples and Counterexamples	93
4.2	Logic	97
4.3	Reductio ad Absurdum	103

5	Invariants	109
5.1	Double Counting	109
5.2	Searching for Invariants	111
5.3	Termination	112
5.4	Even and Odd Numbers	115
6	Project: 15-Puzzle	121
6.1	The Puzzle	121
6.2	Permutations and Transpositions	122
6.3	Why 15-puzzle Has No Solution	133
6.4	When 15-puzzle Has a Solution	135
6.5	Implementation	142
7	Appendix: SAT and ILP Solvers	149
7.1	Cutting a Figure	149
7.2	Using SAT Solvers	149
7.3	Using ILP Solvers	157
7.4	Visualizing Football Fans	160

II

Combinatorics and Probability

8	Basic Counting	165
8.1	Starting to Count	165
8.2	Recursive Counting	171
8.3	Tuples and Permutations	177
9	Binomial Coefficients	185
9.1	Number of Games in a Tournament	185
9.2	Combinations	188
9.3	Binomial Theorem	194
9.4	Practice Counting	197
10	Advanced Counting	201
10.1	Review	201
10.2	Combinations with Repetitions	202
10.3	Practice Counting	206
11	Probability	215
11.1	What is Probability?	215
11.2	Probability: Do's and Don'ts	231
11.3	Conditional Probability	245
11.4	Monty Hall Paradox	257

12	Random Variables	263
12.1	Random Variables and Their Expectations	263
12.2	Linearity of Expectation	272
12.3	Expectation is Not All	275
12.4	Markov's Inequality	277
13	Dice Games	279
13.1	Dice Game Problem	279
13.2	Optimal Strategy	282

III

Number Theory and Cryptography

14	Modular Arithmetic	287
14.1	Divisors and Multiples	287
14.2	Modular Arithmetic	293
15	Euclid's Algorithm	301
15.1	Greatest Common Divisor	301
15.2	Applications	308
16	Building Blocks for Cryptography	317
16.1	Integer Factorization	317
16.2	Chinese Remainder Theorem	327
16.3	Modular Exponentiation	333
17	Cryptography	347
17.1	Secure Communication	347
17.2	Substitution Ciphers	348
17.3	One-time Pad	351
17.4	RSA Cryposystem	357
17.5	Attacks and Vulnerabilities	360

0. About the Book

0.1 Active Learning

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

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

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

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

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

0.2 Problem-based Learning

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

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

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

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

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

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

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

0.3 Python Programming Language

0.3.1 Why Programming?

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

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

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


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


0.3.2 Why Python?

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

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

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

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

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

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

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

0.3.3 How to Catch Up with Python?

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

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

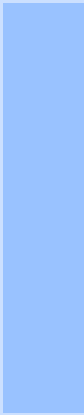
In the cloud. Alternatively, you may run all our code snippets from your Internet browser, without installing or configuring anything on your machine. To do this, visit the [repository page](#) and click the badge “Open in Colab”. This will show you a list of notebooks that can be run in an interactive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

0.4 Acknowledgments

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

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

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



Mathematical Thinking in Computer Science

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

1. Proofs: Convincing Arguments

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

1.1 Warm Up

1.1.1 Why Proofs?

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

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

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

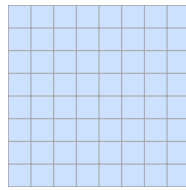


Figure 1.1: An 8×8 chessboard. Can it be tiled with domino tiles?

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

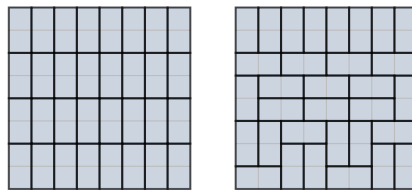


Figure 1.2: Two examples of tiling a chessboard.

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

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

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

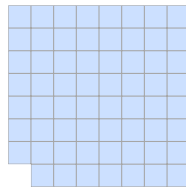


Figure 1.3: A chessboard without a corner. Can it be tiled with domino tiles?

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

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

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

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

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

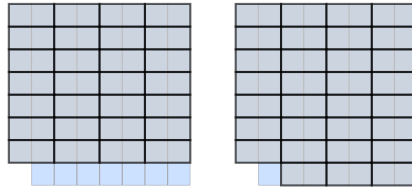


Figure 1.4: An unsuccessful attempt to tile a chessboard without a corner.

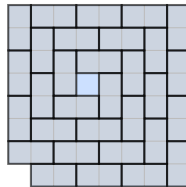


Figure 1.5: An unsuccessful attempt at a spiral tiling of a chessboard without a corner.

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

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

More specifically:

Stop and Think! Imagine there is a tiling of an 8×8 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 (question 3): [Coursera](#) [↗](#).

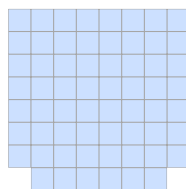


Figure 1.6: A chessboard without two adjacent corners. Can you tile it with domino tiles?

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

Stop and Think! Do you agree with this reasoning?

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

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

Here it is, see Figure 1.7.

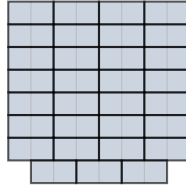


Figure 1.7: A tiling of a chessboard without two corners.

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

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

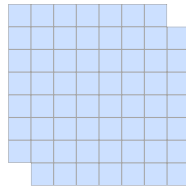


Figure 1.8: A chessboard without two opposite corners.

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

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

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

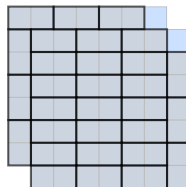


Figure 1.9: An unsuccessful attempt to tile a chessboard without two opposite corners.

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

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

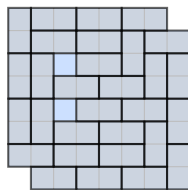


Figure 1.10: Another unsuccessful attempt to tile a board without two opposite corners.

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

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

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

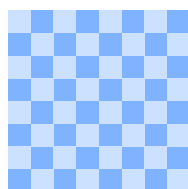


Figure 1.11: Chessboard coloring.

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

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

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

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

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

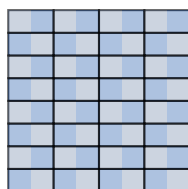


Figure 1.12: Pairing proves that the number of light cells is the same as the number of dark cells.

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

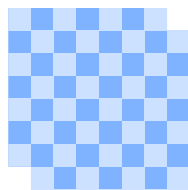


Figure 1.13: Looking again at the board without opposite corners.

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

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

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

Imagine that a tiling existed. It would use $62/2 = 31$ tiles. Each tile covers one dark cell and one light cell. So in total all tiles would cover 31 dark cells and 31 light cells. And — *aha!* — our board has 30 dark and 32 light cells. This mismatch shows that tiling it 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 (questions 5 and 6): [Coursera](#) ↗.

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

The following problem provides an answer to this question.

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

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

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

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

It remains to prove that the board without one dark and one light cell is always tileable. To keep the suspense, we will not give the solution of this problem. Here is a diagram to think about, Figure 1.14. If you are old enough (some of the authors are), you may remember the computer game where a growing snake moves along itself eating food items placed in certain cells and increasing its 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.

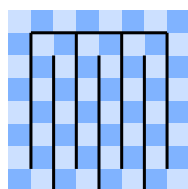


Figure 1.14: This “circular snake” helps us to prove that the board without two cells of opposite colors is tileable. Do you see how? For starters, tile the entire board by cutting the snake into domino tiles. What happens if you delete two cells from the snake?

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 (questions 1 and 2): [Coursera](#) [↗](#).

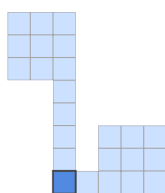


Figure 1.15: Board for Problem 6.


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 (question 3): [Coursera](#) .

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 the 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.)