

**100 Ideas for
Secondary Teachers:
Outstanding Computing
(Sample) Lessons**

Simon Johnson

BLOOMSBURY EDUCATION
Bloomsbury Publishing Plc
50 Bedford Square, London, WC1B 3DP, UK
29 Earlsfort Terrace, Dublin 2, Ireland

BLOOMSBURY, BLOOMSBURY EDUCATION and the Diana logo are
trademarks of Bloomsbury Publishing Plc

First published in Great Britain 2021

Text copyright © Simon Johnson, 2021

Simon Johnson has asserted his right under the Copyright, Designs and
Patents Act, 1988, to be identified as Author of this work

Bloomsbury Publishing Plc does not have any control over, or responsibility
for, any third-party websites referred to or in this book. All internet addresses
given in this book were correct at the time of going to press. The author and
publisher regret any inconvenience caused if addresses have changed or sites
have ceased to exist, but can accept no responsibility for any such changes

All rights reserved. No part of this publication may be reproduced or
transmitted in any form or by any means, electronic or mechanical, including
photocopying, recording, or any information storage or retrieval system,
without prior permission in writing from the publishers

A catalogue record for this book is available from the British Library

ISBN: PB: 978-1-4729-8440-1; ePDF: 978-1-4729-8442-5;
ePub: 978-1-4729-8441-8

2 4 6 8 10 9 7 5 3 1 (paperback)











Typeset by Newgen KnowledgeWorks Pvt. Ltd., Chennai, India
Printed and bound in the UK by CPI Group Ltd, Croydon CR0 4YY



All papers used by Bloomsbury Publishing Plc are natural, recyclable
products from wood grown in well managed forests and other sources. The
manufacturing processes conform to the environmental regulations of the
country of origin

To find out more about our authors and books visit www.bloomsbury.com
and sign up for our newsletters

Contents

Introduction	i
Testimonials	ii
How to use this book	iii
Part 1: Programming strategies	1
 Paired programming	2
 Rubber duck debugging	3
 Code golf	4
 Game design	6
 PRIMM	7
 Parsons problems	8
 Use-modify-create	9
 Hour of Code	10
Part 5: Computational thinking	11
 Crazy characters	12
 Puzzle me	13

Introduction

I firmly believe that computer science is for everyone. Not only does it foster problem-solving, creativity and critical thinking skills, but it also has the potential to empower young people and give them the tools to express themselves in a variety of cool ways.

As society becomes increasingly more reliant on the use of technology, the need for a formal computing education or qualification becomes ever more important. Not only are we preparing students for the digital world, but we are also preparing them for jobs that don't even exist yet. It is therefore imperative that we provide our students with the necessary tools to prepare them for every opportunity that might come their way.

Writing this book gave me the opportunity to reflect on my own teaching practice. It reminded me that, despite graduating with honours in computer science, the thought of teaching computing for the first time filled me with trepidation. However, I was also reminded of how excited I was at the prospect of being able to try out some new teaching strategies, in particular, the idea of teaching computing without computers (also known as 'teaching unplugged').

I was also able to reflect on the valuable lessons learned while teaching computing, probably the most notable being that context is key! By making computing relevant and providing a 'real-life' context, you can create meaningful learning experiences for your students. This also applies to relating computing content to other aspects of the curriculum, which is why I have dedicated a chapter in this book to STEAM (science, technology, engineering, arts and mathematics).

I also learned that you should never be afraid to let your students teach you! One of the biggest mistakes that I made early on in my teaching career was to assume that I must be the fount of all knowledge. The truth of the matter is that it's impossible to know everything, especially with a subject like computing. In fact, there will be times when your students know more than you – coming to terms with this inevitable truth is an important step in your professional development. Be open to learning alongside your students and don't be afraid to ask them for help!

Finally, despite what the title of the book may suggest, the intention of this book is not to make every lesson outstanding. While it is admirable to aspire to be outstanding all the time, to achieve this would be unsustainable and a detriment to your health and wellbeing! Instead, this book is a compilation of tried and tested practical ideas, designed to be adapted and modified, which have the potential to create outstanding learning experiences for you and your students.

I would love to hear how you use these ideas in your own classroom, so please do get in touch! Please share your reflections with me @clcsimon on Twitter using the #100ideas hashtag or via my Facebook page, www.facebook.com/teachwithitc.

Testimonials

9.1/10 – *"Another great book within this fantastic series of practical teaching books, offering great support and resource ideas to support computing at secondary level."* – UKEdChat

"A pocketful of inspiration for your next computing lesson." – Hello World Magazine

"An outstanding book that can simply be picked up off the shelf as a means to inspire any teacher when delivering computing. Simon has shared these super quick ideas that will have impact in every classroom, from rubber duck debugging to pedagogical approaches in the subject. This is an excellent read!" – Matt Warne, Head of Computing and Digital Learning at RGS The Grange, CAS Master Teacher and Computing Champion, @MattWarne

"Read this if you teach Computing! Packed with lesson ideas but more importantly loads of nuggets of wisdom about pedagogy: research-informed techniques that work, all packaged into bitesize chunks you can read in your break or put this on your summer reading pile to come back refreshed and raring to go in September. I will definitely be doing "code golf" (write a program to solve a problem in as few lines as possible), "crazy characters" (back-to-back drawing to teach the importance of clear instructions) and "intelligent pieces of paper" (introducing AI by playing noughts and crosses against a written algorithm). Simon has pulled together a goldmine of inspirational and powerful ideas which is essential reading for the Computing teacher." –

Alan Harrison, Head of Computing at William Hulme's Grammar School, CAS Master Teacher, @MrAHarrisonCS

"I highly recommend this book. Whether primary or secondary contains a bunch of amazing ideas to inspire you and your students in all things computing, coding, programming etc." –

Tim Wilson, CAS Community Outreach Manager, @casmidlands

"This book is amazing for trainee teachers, newly qualified teachers, and the more experienced teachers. I have been a teacher for 6 years, feel I have the depth in subject knowledge but wanted exercises across the topics that were fresh, engaging and looked at cognitive load. This book ticks all the boxes! Hopefully there will be a version 2 soon." –

Ms. C. Gryspeerdt, CAS Master Teacher, Community Leader and Digital School House Lead teacher, @CLG7179

How to use this book

This book includes quick, easy and practical ideas for you to dip in and out of to help you deliver effective and engaging computing lessons. Each idea includes:

- a catchy title, easy to refer to and share with your colleagues
- an interesting quote linked to the idea
- a summary of the idea in bold, making it easy to flick through the book and identify an idea you want to use at a glance
- a step-by-step guide to implementing the idea.

Each idea also includes one or more of the following:

Teaching tip

Practical tips and advice for how and how not to run the activity or put the idea into practice.

Taking it further

Ideas and advice for how to extend the idea or develop it further.

Bonus idea



Bonus ideas in this book that are extra-exciting, extra-original and extra-interesting.

Share how you use these ideas and find out what other practitioners have done using #100ideas.

Online resources to accompany this book can be found at www.teachwithict.com/100ideas. Here you will find editable versions of all the code needed to run the ideas. You can copy and paste the code or tweak it to suit your requirements. There are also a variety of downloadable resources such as pre-prepared cards and worksheets to help you put the ideas into practice.

Programming strategies



Part 1

Paired programming

'Sir, is it my turn to drive yet?'

For many, the idea of students working in pairs at a computer, especially if access to a computer is limited, is fairly common. However, even if you have the luxury of students being able to work on their own at a computer, you may still wish to consider students working in pairs, particularly when learning how to code.

Teaching tip

Switch roles regularly to ensure that all the students get a fair share of being both driver and navigator. I find that setting a timer/alarm for five-minute intervals helps to manage the students' time effectively and keeps them engaged!

Paired programming, as the name suggests, sees students working in pairs, with one taking the role of the driver (inputting the code into the computer) and the other assuming the role of navigator (reading out instructions and checking each line of code as it is typed in).

Pairing students up makes sense for a number of reasons, even more so when learning how to code! Paired programming has been shown to improve overall confidence, produce fewer errors and increase engagement compared with learning to code individually. In fact, it is a practice that is widely used in industry by professional programmers.

A word of caution! While at first it may seem beneficial to pair up higher-achieving students with those of lower ability, I find that this can have negative effects, as often one student may feel that they are being held back or another may take over completely. Experience shows that pairing students of similar ability seems to produce the best outcomes.

Rubber duck debugging

'I name my duck Ducky McDuck Face!'

Have you ever started asking someone to help you to solve a problem and, halfway through, you figure it out for yourself? Well, this is pretty much how rubber duck debugging works. The majority of 'code bugs' originate from not being clear and explicit with instructions. By describing your problem to the duck, you force yourself to express your ideas in a clear and logical way!

Rubber duck debugging is a programming methodology used by software engineers to help them to find bugs and problems in their code. The term 'rubber duck' refers to an inanimate object that understands little or nothing about the given problem!

First, you'll need to acquire some rubber ducks! I find that the local pound store is usually a good place to start. Give each student a duck and ask them to name it. Once the students have named their duck and have stopped throwing them around the classroom (!), ask them to place their duck next to their computer. Tell the students to address their duck by its name and tell the duck something about themselves – while feeling strange at first, this should help the students to get over the fear of talking to the duck in front of their peers!

Share with students some broken code (I tend to start with some simple syntax errors) and a brief explanation of what the code is meant to do. Tell the students to inform their duck that they're going to go over some code with it! Instruct the students to explain to their duck what the code is supposed to do and then explain the code in detail (line by line).

Eventually, the students should spot the seemingly obvious error, at which point they can thank their duck for being such a help!

Teaching tip

Despite what the name suggests, you don't have to use a rubber duck. Any inanimate object will suffice: a can of pop, a hat, a scarf or even a favourite teddy!

Code golf

'Anyone for a game of code golf? Fore!'

One strategy that exemplifies the concept of gamification in the teaching of coding is a game called 'code golf'.

Teaching tip

To make the challenge fair and consistent, I recommend that you set some competition rules. For example:

- Blank lines do not count as lines of code.
- Comments do not count as lines of code.
- Importing libraries do not count as lines of code.
- Everything else counts.

Code golf owes its name to its resemblance to the scoring system used in conventional golf, where participants aim to achieve the lowest score possible. The idea is simple: participants are given a problem (or working solution) and are challenged to solve it using the fewest lines of code.

The aim of code golf is to encourage efficient use of code. Efficient code uses less RAM, compiles quicker and uses up less storage space. Students can use a combination of features such as loops (for, while, repeat) and functions (or sub-routines) to achieve their optimised code. However, readability and usability must not be sacrificed at the expense of code optimisation. Therefore, white space and comments do not count as lines; we still want to encourage students to break up and comment their code so that it is comprehensible to others, easier to debug and easy for others to reuse.

There are two main ways to play code golf. The first way requires students to solve a given problem using the fewest lines of code. The second method, which requires a little more preparation from the teacher, requires the students to optimise a given working solution. In both methods, the challenge is for the students to create a solution using the least amount of code. To add a little extra challenge, the teacher can add a 'par' value (or target number), with the par being the optimal number of lines of code. This par value can be altered for different levels of ability (similar to

the 'handicap' system in conventional golf), thus allowing the teacher to differentiate the activity.

The following is an example of a simple 'par challenge' using the turtle library in Python. In this example, students are challenged to create a square using six lines of code (par 6). Share the following example code and ask the students to identify the repeating pattern.

```
import turtle
window = turtle.Screen()
timmy = turtle.Turtle()
timmy.forward(100)
timmy.right(90)
timmy.forward(100)
timmy.right(90)
timmy.forward(100)
timmy.right(90)
timmy.forward(100)
timmy.right(90)
```

Ask students to suggest ways in which the code can be made more efficient – draw out answers such as 'use a loop' or 'use a repeat', etc. Share the optimised solution for the above example:

```
import turtle
window = turtle.Screen()
timmy = turtle.Turtle()
for loopCounter in range(4):
    timmy.forward(100)
    timmy.right(90)
```

Inform the students that, in the modified example, the code has been made more efficient by using a counted loop. Explain that efficient code uses less RAM, compiles quicker and uses up less storage space.

Put students into pairs (see Idea 1: Paired programming) and challenge them to create a series of regular polygon shapes (e.g. equilateral triangle, pentagon, hexagon, etc.) using the fewest lines of code possible.

Bonus idea ★

Create a score card for students to record the number of lines used for each shape.

Game design

‘Sir, can we make Flappy Bird?’

Often, the first time that you mention the word ‘coding’ to children, their initial reaction is ‘are we going to make games?’. While coding is not about making games, game design can be used as a hook to encourage students to learn to code.

Taking it further

Many game-creation tools allow you to add your own sprites and graphics. I find that this provides a perfect opportunity to explore Creative Commons and gets students designing their own game sprites.

Game design, as the name suggests, is the process of planning the content and rules of a game. It also includes the design of gameplay, environment, storyline and even characters. Now, thanks to a plethora of free online tools, coding games has never been easier!

Scratch (scratch.mit.edu) is perfect for creating games and, despite its simplicity and appeal to younger audiences, should be on everyone’s list of ‘go-to’ game-creation tools.

Stencyl (www.stencyl.com) is a free game-creation platform, which, utilising a Scratch-like interface, allows students to create games for iOS, Android, Windows and Mac.

Alice (www.alice.org) is a free and open-source 3D-programming environment designed to teach students object-oriented and event-driven programming. In Alice, students drag and drop graphic tiles in order to create animations or program simple games in 3D.

App Inventor (appinventor.mit.edu) is a great tool to teach programming. Like Scratch, App Inventor uses a drag-and-drop interface that allows you to assemble code from blocks.

Microsoft MakeCode Arcade (arcade.makecode.com) is a web-based, beginner-friendly code editor that allows you to create retro arcade games. With MakeCode Arcade, students can create games using either blocks, JavaScript or a combination of both.

PRIMM

'My students seem so much more confident when learning to code using PRIMM.'

PRIMM incorporates discussion and investigation of sample code through scaffolded tasks to help students to understand code before they start writing their own code.

I was first introduced to the idea of PRIMM at a Computing at School workshop led by Dr Sue Sentence (King's College London).

PRIMM is a research-based approach to teaching programming. It is made up of five stages: Predict, Run, Investigate, Modify, Make. Each stage is used in planning lessons and activities to support the learning of programming.

The five stages of PRIMM are:

- **Predict:** Students are given a working program and challenged to predict what the code will do. At this level, the focus is on what the code actually does.
- **Run:** Students run the program so that they can test their predictions and discuss their findings with their partner/rest of the class.
- **Investigate:** The teacher provides a range of scaffolded activities aimed to help the students to explore what each line of code does. Strategies can include tracing, commenting code, annotating, debugging, etc.
- **Modify:** Students are challenged to modify the working program in order to change its functionality in some way.
- **Make:** Students design a new program that is based on the given solution but which solves a new problem.

Teaching tip

It is not expected that you cover all five stages in one lesson. In fact, you may wish to focus on one stage over several lessons.

Parsons problems

'Help students to learn how to code by removing some of the barriers!'

When students move from block-based languages to text-based languages, they often get frustrated with syntax errors. One method that helps to reduce this frustration is 'Parsons problems'.

Teaching tip

You don't need a computer to introduce Parsons problems. I find that eliminating computers altogether can further reduce confusion and frustration! I suggest starting by printing and laminating some code blocks or lines of code for students to sort.

Parsons problems are programming puzzles where a working solution to a problem has been broken up into blocks of code and jumbled up. Students are given the mixed-up code and challenged to reassemble the code in the correct order.

Some Parsons problems, often referred to as two-dimensional Parsons problems, also require the code blocks to be indented correctly. Parsons problems can also contain extra lines of code, called distractors, which are not needed for the code to work.

Although primarily used with text-based languages, Parsons problems can be used with block-based languages too! The idea is to allow students to focus on the core concepts, such as flow of control, conditionals and loops, without the frustration of syntax errors.

Example of a Parsons problem with distractors:

Parsons problem

```
timmy = turtle.Turtle()
window = turtle.Screen()
timmy.forward(100)
timmy.right(90)
timmy.turn(90)
For loopCounter in range(4)
for loopCounter in range(4):
import turtle
```

Solution

```
import turtle
window = turtle.Screen()
timmy = turtle.Turtle()
for loopCounter in range(4):
    timmy.forward(100)
    timmy.right(90)
```

Use-modify-create

'Reduce anxiety while supporting growth with this simple three-stage approach to learning to code!'

When we learn to read and write as children, we often tend to learn how to read first. So, why is it that when we teach children to code, we often get them to write code before they can read it? This is the main rationale behind a three-stage approach to learning to code, known as 'use-modify-create'.

The idea behind this approach is to encourage children to 'use' an existing snippet of code and explore what it does, 'modify' (or tinker with) the code to change its behaviour and, once they understand how the program works, 'create' a new program of their own. This strategy compliments the Papert-style approach of learning by exploring and tinkering.

Teaching tip

When moving to the 'create' phase, students should be encouraged to test, analyse and improve their programs regularly, just as they would do with any new project.

How it works

Use: Provide students with a snippet of working code. Give students time to run the program and figure out what the code is meant to do and how it works. Where appropriate, provide students with questions or prompts to aid with their investigations.

Modify: As the students become more comfortable with the program, encourage them to start changing it. For example, if the program is written in Scratch, the students could start by simply changing the sprite or by changing some of the variables. As the students' confidence begins to grow, they can start to make more complex changes to the code.

Create: Once the students are confident with how the program works, have them create their own program using what they have learned.

IDEA 8

Hour of Code

'Give your students a 'byte'-sized introduction to computer science with an hour of code.'

The Hour of Code started as a one-hour introduction to computer science, designed to demystify 'code' to show that anybody can learn the basics, and to broaden participation in the field of computer science.

Teaching tip

Don't worry if you're new to coding yourself. Hour of Code is geared towards beginners. It can be a great opportunity to learn along with your students – and in fact, that's part of the fun!

Hour of Code takes place each year during Computer Science Education Week (usually during the second or third week of December), to mark and celebrate the birth of computing pioneer Admiral Grace Murray Hopper (9 December 1906).

Why Hour of Code?

Every student should have the opportunity to learn computer science. It helps to nurture problem-solving skills, logic and creativity. By starting early, it may also encourage some students to take up computer science as a GCSE, when it comes to their options.

How to participate in the Hour of Code

You can organise an Hour of Code event in your school at any time, not just during Computer Science Education Week. There is a handy guide to getting started on the official Hour of Code website (hourofcode.com/uk). Hour of Code can be delivered during normal lessons or as part of an extra-curricular club.

Where to start

Thankfully, you will find a plethora of resources available online dedicated to Hour of Code. Below are just a few of my favourites:

- Hour of Code: hourofcode.com/uk
- Code.org: code.org/hourofcode/overview
- Tynker: tynker.com/hour-of-code

#HourOfCode

Computational thinking

Part 5

Crazy characters

'You told me to draw a circle, but you never said how big!'

One way to reinforce the importance of clear and precise instructions when writing code is to challenge students to write a simple set of instructions for drawing a 'crazy character'!

Teaching tip

The key to the success of this activity is to be pedantic when following the students' instructions! This will encourage students to refine and improve their instructions.

Taking it further

Time permitting, challenge the students in their pairs to design their own 'crazy character' and write a set of instructions for drawing it.

Put students into pairs and ask each pair to sit with their backs to one another. Give one pair an activity card (containing a simple character design and set of instructions) and the other a pen and paper/mini-whiteboard. Example instructions:

- Create a circle for the body.
- Add two eyes.
- Add four legs.
- Add a mouth.
- Add a tooth.

Inform the students with the activity cards that they must read out their instructions to their partner, who must try to recreate their character using instructions alone. After five minutes, ask the students to compare their drawings with the original and to note down any similarities/differences between the two images.

Ask: 'Why are all the drawings so different to the originals?' Draw out answers such as, 'Not enough detail in the instructions' and 'The instructions need to be more precise'. Ask the students to suggest ways to improve the instructions and make a note of these.

Set a timer for 15 minutes and challenge the students, in their designated pairs, to improve the instructions on the activity cards. Once the time is up, select pairs at random to read their instructions out aloud for you to follow, ensuring that you follow their instructions exactly while drawing their character on a whiteboard or flipchart for the class to see.

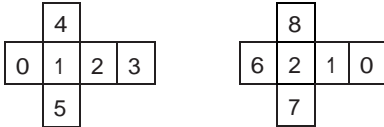
Puzzle me

‘Sir, I’m puzzled!’

One way to provide students with the opportunity to practise computational thinking skills (decomposition, pattern-matching, abstraction and algorithm design) is to have them solve puzzles.

Cube challenge

A two-cube calendar is a desk calendar consisting of two cubes with faces marked with digits 0 to 9. Students are challenged to fill in the gaps on two cube nets so that it is possible to arrange the cubes to represent any chosen day of the month (from 01 through to 31). Both cubes must be used at all times – for example, to represent day seven, both 0 and 7 must be visible. Example solution:



Note: While at first it appears that there are not enough sides for all the numbers, students will eventually realise that the 6 doubles as a 9 when turned upside down.

Word ladders

Word ladders were invented by Lewis Carroll, author of *Alice in Wonderland*. Students are given a start word and end word and, by progressively altering a single letter at a time, they must change the start word into the end word. Each word in the ladder must be a valid English word and must have the same length. Often the start word and end word are related. For example, to turn ‘COLD’ into ‘WARM’, one possible ladder could be: COLD → CORD → WORD → WARD → WARM.

Taking it further

Have students test their puzzle-solving skills by entering them in the Bebras UK challenge (www.bebas.uk), a competition aimed at raising awareness of computer science by challenging players to solve computational thinking puzzles.

Bonus idea

★ Have students attempt to solve other popular puzzles such as Sudoku, ‘camel crossing’, ‘river-crossing conundrum’, Tower of Hanoi, etc.

**100 Ideas for
Secondary Teachers:
Outstanding Computing
Lessons**

Simon Johnson

Bloomsbury
bit.ly/100IdeasSecCS

Amazon
bit.ly/100ideasCS