Computational Thinking

A Primer for Programmers and Data Scientists

G Venkatesh
Madhavan Mukund

Copyright © 2020 Mylspot Education Services Private Limited.

All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods

Draft version v0.1 Release date 13-Oct-2020.

This draft version in pdf form is being distributed free only to the students registered in the first batch (Oct 2020) of the IIT Madras Online BSc degree in Programming and Data Science. Distribution of any part of this pdf document in any form to anyone outside this group is strictly prohibited.

First print planned for December 2020

Contents

Preface

How do we introduce computing to a lay person? This was the question that confronted us when we volunteered to offer such a course in the foundation year of the online BSc degree in Programming and Data Science at IIT Madras. We searched for a suitable book or any material to use, but found nothing useful. Our conception of what needs to be taught in the course seemed to differ widely from what is currently on offer. We figured that "computational thinking" means different things to different people. For some it is just another name for programming in a suitable language. Those with some knowledge of computer science see it as the study of algorithms (as distinct from the programs used to implement them). Then there are those who equate it to solving logical or mathematical puzzles using computers. None of these descriptions seem satisfactory.

Luckily, one of us had experimented with teaching computational thinking to school students under the banner of his startup company Mylspot Education Services. These sessions used data sets with each data element written out on a separate card. Students try to find the answer to a computational question by working through the cards by hand. For instance, they could look at students' marks cards to award prizes, or look at shopping bills to identify the loyal customers, or look at cards containing words from a paragraph to identify which person a pronoun refers to.

The idea of using this hands-on method to explain the key concepts in computational thinking was appealing on many counts. The practical utility becomes immediately obvious when interesting questions about commonly occurring and familiar data sets can be answered. Many of the key abstractions underlying computational thinking seem to surface naturally during the process of moving cards and noting down intermediate values. Finally, the difference in complexity of two alternate solutions can be directly experienced when tried out by hand on a real data set.

The challenge for us was to do all this in an online course format with recorded videos, where there is no possibility of interacting with students during the session. Our response to this was to employ a dialogue format, where the two of us would work together, posing questions and finding answers by working through the cards. The NPTel team in IIT Madras rose to the challenge by setting up the studio with one camera facing down on the work area and another recording our faces. The feedback we got from students who watched the first few recordings was excellent, and this gave us encouragement that the method seemed to be working.

It was at this point that we felt that we should write up all the material created for the recorded sessions in the form of a book. This could form a companion or reference to the students who take the online degree course. But our intent was to make it available to a much wider audience beyond the students of the online degree. The question was - is it even possible to provide the same level of excitement of the hands-on experience through a static book? Our answer was yes, it could - provided we are able to create a interactive digital version of the book that can provide the hands-on experience. We thus decided to also simultaneously develop an interactive version of the book that would be launched on the Mylspot platform for practicing problem solutions.

Just as with the online degree course, the audience for this book is likely to be very diverse. This would mean that we have to reach out not only to those being introduced to the world of computing for the first time, but also to those with many years of practical programming experience looking to strengthen their foundations. It would also mean that we have to be able to enthuse those with no background (or interest) in theory, without putting off those looking to gain an understanding of theoretical computer science. Accordingly, we have organised the material in such a way that there is a good mix of easy and challenging problems to work on. At the end of each chapter, we have provided some selected nuggets of information on algorithms, language design or on programming, which can be skipped by the first time learners, but should hopefully be of interest to the more advanced learners.

We had a wonderful team at IIT Madras supporting us during the production of this course. We thoroughly enjoyed creating, recording, writing and coding all of the material. We sincerely hope that you, the readers, will find the result of this effort both enjoyable and practically useful.

G Venkatesh and Madhavan Mukund
Chennai, India, Dec 2020

Acknowledgements

Several people have contributed in the creation of the material on which this book is based. At the outset, we have to thank Andrew Thangaraj and Prathap Haridoss, both Professors at IIT Madras, who are responsible for the online degree programme, and hence bear the risk of our experimentation with material and methodology for the computational thinking course. Then there is Bharathi Balaji, the fearless leader of the NPTel and online degree teams, without whose 24x7 support (and constant nagging), we would never have completed the material on time. It was through the efforts of these three that we were able to quickly put together the camera arrangements in the studio for recording our lectures.

The online degree content team for the course was led ably by Dhannya S M, who kept the clock ticking. We have to thank Prathyush P, the overall content team leader, for putting together most of the datasets, and recording the first few tutorials. Omkar Joshi arranged the review sessions, provided editorial support for the videos, and suggested many exercises for the course. He was ably supported in this activity by Vijay R and Karthik T. Sribalaji R has to be thanked for accommodating our urgent requests to schedule the recordings and patiently and professionally managing these long sessions. He was supported in this activity by Karthik B, Ravichandran V, Mahesh Kumar K P, Vignesh B, Komathi P and Akshai Kumar R.

Part I

Getting Started With Computational Thinking

1. Introduction

1.1 What is Computational Thinking?

We see computations all around us. When we scan the shelves in a supermarket for a particular item, we unconsciously follow a previously learnt searching method based on our knowledge of where different items are stored. A very similar search may be witnessed when we look for an item in the kitchen, when we look for some book on our bookshelves, or search for a document in our folders. Likewise, when we make our to-do list in the morning and check it off one by one as we carry out the tasks in it, the tracking system in use resembles the one where we make a shopping list and score out the items which we have found and purchased, or the one where we make an invitation list for a party and tick off the friends we have already called.

Consider now the process of taking out of clothes from a bucket and hanging them on multiple clothes lines. This process is strikingly similar to the one that we will use when we pick out books from a box and place them on a multiple shelves, or when we are given some travel items and are asked to pack them into a set of suitcases. As a last example consider finding a road route to take when we have to travel to a different city. This looks similar to running a wire between one room and another in our house. More surprisingly, this may also resemble the process of using our friends network to contact someone who can provide the correct medical or career advice.

If we look carefully into each of these examples of similar activities, we will observe a shared pattern consisting of a sequence of atomic actions. In this book, we identify **computational thinking** with the search for such shared patterns among commonly occurring computational activities.

Why do we need to study computational thinking? For one, it helps make our thought process clearer when we are dealing with activities of this sort. Would not our productivity increase greatly if we were able to efficiently retrieve and re-apply a previously learnt procedure rather than re-invent it each time? For another, it helps us communicate these procedures to others, so that they can quickly learn from us how to do something. If we can communicate the procedure to another person, surely we could communicate it to a computer as well, which would explain the use of the adjective "computational".

In short, computational thinking gives us the basis to organise our thought process, and a language to communicate such organisation to others including a computer. The organising part has relation to the study of **algorithms**, the language part to the **design of programming languages and systems** and the last communication part to the activity of **coding in a given programming language**. Thus computational thinking provides the foundation for learning how to translate any real world problem into a computer program or system through the use of algorithms, languages and coding.

1.1.1 Stepwise approach to problem solving

The basic unit of the computational activity is some kind of step. Scanning a shelf in a supermarket requires the basic step where we visually examine one item (sometimes we may need to pick up the item and look at its label carefully - so the examine step may itself be composed of a number of smaller steps). The scanning process requires us to move from one item to the next (which may or may not be adjacent to the item scanned earlier). When we have scanned one shelf, we move to the next one on the same rack. We may then move from one rack to another, and then from one aisle (part of the supermarket) to another aisle (another part of the supermarket). In essence we carry out the examine step many times, moving from item to item in a systematic manner so that we can avoid the examination of the same item twice.

What can we call a step? Is it enough for the step to be stated at a high level? Or does it need to be broken down to the lowest atomic level? This depends on who we are communicating the step to. If the person on the other side can understand the high level step, it is sufficient to provide instructions at a high level. If not, we may have to break down the high level step into more detailed steps which the other person can understand.

Suppose that we need to arrange the furniture in a room for a conference or a party. If the person taking the instructions understands how such arrangements are to be done, we could simply tell the person - "I am having a party tomorrow evening at 6 pm for about 100 people. Please arrange the furniture in the hall accordingly." On the other hand, if the person has never seen or carried out such an organisation before, we may have to say - "One round table can accommodate 5 people and I have 100 people, so we need 20 round tables and 100 chairs. Distribute these 20 tables around the room, each with 5 chairs."

1.1.2 Finding re-usable patterns

Once we have defined the different kinds of steps that we can use, the computational activity is then described as a sequence of these steps. It is wasteful to write down all the steps one by one. For instance, if we say how to organise one table with 5 chairs, then that same method can be applied to all the tables - we do not need to describe it for each table separately. So, we need some method or language to describe such grouping of steps.

Lets go back to the problem of picking clothes out of the bucket and hanging them out to dry on more than one clothes lines. We could pick out the first cloth item and hang it on the first line. Then pick another cloth item and hang it on the first line somewhere where there is space. If the next cloth item we pick cannot fit into the any of the free spaces on the line, we move to the next clothes line. We can immediately see that there is a problem with this method - the first line has clothes of random sizes placed in a random manner on the line leaving a lot of gaps between the cloth items. The next cloth item picked may be a large one not fitting into any of these gaps, so we move to another line wasting a lot of space on the first line. A better procedure may be to pick the largest cloth item first, place it on the leftmost side of the first line. Then pick the next largest item, hang it next to the first one, etc. This will eliminate the wasted space. When we are near the end of the space on the first line, we pick the cloth item that best fits the remaining space and hang that one at the end of the line.

What we have described above is a pattern. It consists of doing for each clothes line the following sequence of steps starting from one end: fit one next to another cloth items in decreasing order of *size* till the space remaining becomes *small*, then we pick the cloth item that best fits the remaining space on the line. This "pattern" will be the same for arranging books on multiple shelves, which consists of doing for each book shelf the following sequence of steps starting from one end: fit one next to another books in decreasing order of *size* till the space remaining becomes *small*, then we pick the book that best fits the remaining space on the shelf. Note that the only difference is we have replaced clothes line with book shelf and cloth item with book, otherwise everything else is the same.

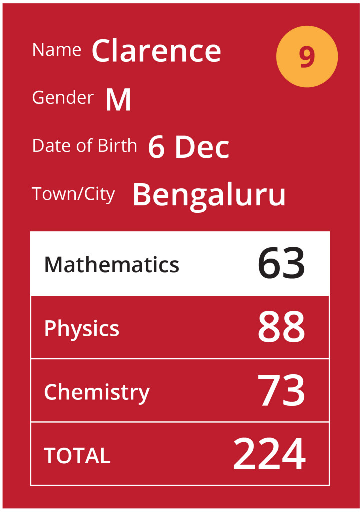
We can re-apply the pattern to the travel items and suitcases example. We do for each suitcase the following sequence of steps starting from one end: fit one next to another travel items in decreasing order of *size* till the space remaining becomes *small*, then we pick the travel item that best fits the remaining space in the suitcase. Can you see how the pattern is the same? Except for the items that are being moved, the places where they are being moved to, the definition of *size* and *small*, everything else looks exactly the same between the three.

1.2 Sample datasets

We have picked some commonly used datasets that are simple enough to represent, but yet have sufficient structure to illustrate all the concepts that we wish to introduce through this book. The data could be represented using physical cards (if the suggested procedures are to be carried out by hand as an activity in a classroom). But we have chosen to use a tabular representation of the dataset for the book, since this is more compact and saves precious book pages. There is a direct correspondence between a row in the table and a card, and we can work with table rows much as we do with physical cards. The accompanying digital companion will make this explicit. It will use the tabular form representation, with the key difference being that the individual table rows are separate objects like cards that can be moved around at will.

1.2.1 Classroom dataset

A data element in this dataset is the marks card of one student (shown below):



9

Name Clarence

Gender M

Date of Birth 6 Dec

Town/City Bengaluru

Mathematics	63
Physics	88
Chemistry	73
TOTAL	224

Figure 1.1: Grade card data element

The card carries the following fields:

- Unique id no: value 9 in the card above
- Name of student
- Gender: M for Male and F for Female
- Date of Birth: Only the day and month is recorded as it is sufficient for the questions that we will be asking related to this field
- Town/City
- Marks in Mathematics, Physics, Chemistry and the Total of these three marks

The dataset has 30 such cards with a mixture of Male and Female students from different cities.

Example questions are:

- Which students are doing well in each subject and overall ?
- Are the girls doing better than the boys?
- Can we group students based on their performance?
- Are there any two students born on the same day and month?
- Are those scoring high marks in Maths also scoring high in Physics?
- Can we make pairs (A,B) of students in such a way that A can help B in one subject, while B can help A in another subject?

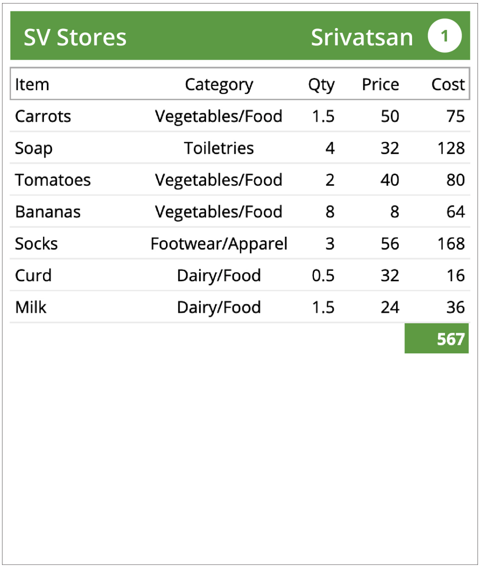
The entire data set is contained in the following table:

No	Name	Gen	DoB	Town/City	MA	PH	CH	Tot
0	Bhuvanesh	M	7 Nov	Erode	68	64	78	210
1	Harish	M	3 Jun	Salem	62	45	91	198
2	Shashank	M	4 Jan	Chennai	57	54	77	188
3	Rida	F	5 May	Chennai	42	53	78	173
4	Ritika	F	17 Nov	Madurai	87	64	89	240
5	Akshaya	F	8 Feb	Chennai	71	92	84	247
6	Sameer	M	23 Mar	Ambur	81	82	87	250
7	Aditya	M	15 Mar	Vellore	84	92	76	252
8	Surya	M	28 Feb	Bengaluru	74	64	51	189
9	Clarence	M	6 Dec	Bengaluru	63	88	73	224
10	Kavya	F	12 Jan	Chennai	64	72	68	204
11	Rahul	M	30 Apr	Bengaluru	97	92	92	281
12	Srinidhi	F	14 Jan	Chennai	52	64	71	187
13	Gopi	M	6 May	Madurai	65	73	89	227
14	Sophia	F	23 July	Trichy	89	62	93	244
15	Goutami	F	22 Sep	Theni	76	58	90	224
16	Tauseef	M	30 Dec	Trichy	87	86	43	216
17	Arshad	M	14 Dec	Chennai	62	81	67	210
18	Abirami	F	9 Oct	Erode	72	92	97	261
19	Vetrivel	M	30 Aug	Trichy	56	78	62	196
20	Kalyan	M	17 Sep	Vellore	93	68	91	252
21	Monika	F	15 Mar	Bengaluru	78	69	74	221
22	Priya	F	17 Jul	Nagercoil	62	62	57	181
23	Deepika	F	13 May	Bengaluru	97	91	88	276
24	Siddharth	M	26 Dec	Madurai	44	72	58	174
25	Geeta	F	16 May	Chennai	87	75	92	254
26	JK	M	22 Jul	Chennai	74	71	82	227
27	Jagan	M	4 Mar	Madurai	81	76	52	209
28	Nisha	F	10 Sep	Madurai	74	83	83	240
29	Naveen	M	13 Oct	Vellore	72	66	81	219

Figure 1.2: Classroom dataset

1.2.2 Shopping bill dataset

The basic data element in this dataset is the shopping bill generated when a customer visits a shop and buys something (as shown below):



SV Stores Srivatsan 1

Item	Category	Qty	Price	Cost
Carrots	Vegetables/Food	1.5	50	75
Soap	Toiletries	4	32	128
Tomatoes	Vegetables/Food	2	40	80
Bananas	Vegetables/Food	8	8	64
Socks	Footwear/Apparel	3	56	168
Curd	Dairy/Food	0.5	32	16
Milk	Dairy/Food	1.5	24	36
				567

Figure 1.3: Shopping bill data element

The card carries the following fields:

- Unique id no: value 1 in the card above
- Name of store: SV Stores
- Name of the customer: Srivatsan
- List of items purchased: Item name, category name/sub category name, Quantity purchased, Price per unit, Cost of item purchased
- Total bill value

Several other fields normally present in shopping bills is omitted. For instance, tax amounts or percentages (GST), store address, phone number etc. These are not required for the questions we are going to ask about these data elements. The dataset has 30 such cards (bills) with a mixture of different shops, customers and item categories.

Example questions are:

- Which customer is spending the most? Which shop is earning more revenues?
- Are the high spenders shopping at the higher revenue shops?
- Which category is selling more?
- Which shop attracts the most number of loyal customers (those who go only

to one shop)?

- Which customers are similar to each other in terms of their shopping behaviour ?

The entire data set is contained in the following table:

BNo	Shop	Customer
1	SV Stores	Srivatsan

Item	Category	Qty	Price	Cost	Total
Carrots	Vegetables/Food	1.5	50	75	567
Soap	Toiletries	4	32	128	
Tomatoes	Vegetables/Food	2	40	80	
Bananas	Vegetables/Food	8	8	64	
Socks	Footwear/Apparel	3	56	168	
Curd	Dairy/Food	0.5	32	16	
Milk	Dairy/Food	1.5	24	36	

BNo	Shop	Customer
2	Big Bazaar	Sudeep

Item	Category	Qty	Price	Cost	Total
Baked Beans	Canned/Food	1	125	125	1525
Chicken Wings	Meat/Food	0.5	600	300	
Cocoa powder	Canned/Food	1	160	160	
Capsicum	Vegetables/Food	0.8	180	144	
Tie	Apparel	2	390	780	
Clips	Household	0.5	32	16	

BNo	Shop	Customer
3	Sun General	Srivatsan

Item	Category	Qty	Price	Cost	Total
Batteries	Utilities	6	14	84	354
USB Cable	Electronics	1	85	85	
Ball Pens	Stationery	5	12	60	
Onions	Vegetables/Food	1.25	100	125	

BNo	Shop	Customer
4	SV Stores	Akshaya

Item	Category	Qty	Price	Cost	Total
Face Wash	Toiletries	1	89	89	1341
Shampoo	Toiletries	1	140	140	
Onions	Vegetables/Food	1	98	98	
Bananas	Fruits/Food	4	8	32	
Milk	Dairy/Food	1	24	24	
Biscuits	Packed/Food	2	22	44	
Maggi	Packed/Food	1	85	85	
Horlicks	Packed/Food	1	270	270	
Chips	Packed/Food	1	20	20	
Chocolates	Packed/Food	4	10	40	
Cereal	Packed/Food	1	220	220	
Handwash	Toiletries	1	139	139	

Air freshener	Toiletries	2	70	140	

BNo	**Shop**	**Customer**
5	Big Bazaar	Akshaya

Item	**Category**	**Qty**	**Price**	**Cost**	**Total**
Trousers	Women/Apparel	2	870	1740	4174
Shirts	Women/Apparel	1	1350	1350	
Detergent	Household	0.5	270	135	
Tee shirts	Women/Apparel	4	220	880	
Instant Noodles	Canned/Food	3	23	69	

BNo	**Shop**	**Customer**
6	Sun General	Rajesh

Item	**Category**	**Qty**	**Price**	**Cost**	**Total**
Notebooks	Stationery	3	20	60	378
Apples	Fruits/Food	6	24	144	
Pears	Fruits/Food	4	30	120	
Chart Paper	Stationery	2	22	44	
Ruler	Stationery	1	10	10	

BNo	**Shop**	**Customer**
7	SV Stores	Advaith

Item	**Category**	**Qty**	**Price**	**Cost**	**Total**
Milk	Dairy/Food	2	24	48	123
Bread	Packed/Food	1	30	30	
Eggs	Food	1	45	45	

BNo	**Shop**	**Customer**
8	Big Bazaar	Advaith

Item	**Category**	**Qty**	**Price**	**Cost**	**Total**
Trousers	Men/Apparel	2	950	1900	3132
Basa Fish	Meat/Food	1	350	350	
Boxers	Men/Apparel	4	160	640	
Face Wash	Toiletries	1	72	72	
Slippers	Footwear/Apparel	1	170	170	

BNo	**Shop**	**Customer**
9	Sun General	Aparna

Item	**Category**	**Qty**	**Price**	**Cost**	**Total**
Mosquito Coil	Household	2	24	48	186
Bananas	Fruits/Food	6	5	30	
Ball Pens	Stationery	4	12	48	
Paper Clips	Stationery	1	60	60	

BNo	**Shop**	**Customer**
10	SV Stores	Akhil

Item	**Category**	**Qty**	**Price**	**Cost**	**Total**
Bread	Packed/Food	1	30	30	96
Biscuits	Packed/Food	3	22	66	

BNo	**Shop**	**Customer**

11	Big Bazaar	Mohith

Item	Category	Qty	Price	Cost	Total
Lindt	Chocolate/Food	1	125	125	595
Socks	Footwear/Apparel	1	120	120	
Spring Onions	Vegetables/Food	0.5	220	110	
Lettuce	Vegetables/Food	0.6	150	90	
Cookies	Snacks/Food	2	75	150	

BNo	Shop	Customer
12	Sun General	Vignesh

Item	Category	Qty	Price	Cost	Total
Phone Charger	Utilities	1	230	230	656
Razor Blades	Grooming	1	12	12	
Razor	Grooming	1	45	45	
Shaving Lotion	Grooming	0.8	180	144	
Earphones	Electronics	1	210	210	
Pencils	Stationery	3	5	15	

BNo	Shop	Customer
13	SV Stores	Abhinav

Item	Category	Qty	Price	Cost	Total
Chocolates	Packed/Food	1	10	10	893
Cereal	Packed/Food	1	220	220	
Bananas	Fruits/Food	6	8	48	
Tomatoes	Vegetables/Food	1	40	40	
Curd	Dairy/Food	1	32	32	
Milk	Dairy/Food	2	24	48	
Horlicks	Packed/Food	1	270	270	
Plates	Household	4	45	180	
Eggs	Food	1	45	45	

BNo	Shop	Customer
14	Big Bazaar	Abhinav

Item	Category	Qty	Price	Cost	Total
Shoes	Footwear/Apparel	1	2700	2700	3060
Polish	Footwear/Apparel	1	120	120	
Socks	Footwear/Apparel	2	120	240	

BNo	Shop	Customer
15	Sun General	Abhinav

Item	Category	Qty	Price	Cost	Total
Keyboard	Electronics	1	780	780	1100
Mouse	Electronics	1	320	320	

BNo	Shop	Customer
16	SV Stores	George

Item	Category	Qty	Price	Cost	Total
Cereal	Packed/Food	1	220	220	279
Milk	Dairy/Food	1	24	24	
Cupcakes	Packed/Food	1	25	25	

Chocolates	Packed/Food	1	10	10	

BNo	**Shop**	**Customer**
17	Big Bazaar	Radha

Item	**Category**	**Qty**	**Price**	**Cost**	**Total**
Broccoli	Vegetables/Food	0.5	120	60	798
Chicken Legs	Meat/Food	0.5	320	160	
Basa Fish	Meat/Food	1	350	350	
Lettuce	Vegetables/Food	0.8	150	120	
Eggs	Meat/Food	12	9	108	

BNo	**Shop**	**Customer**
18	Sun General	Advaith

Item	**Category**	**Qty**	**Price**	**Cost**	**Total**
Pencils	Stationery	2	5	10	187
Notebooks	Stationery	4	20	80	
Geometry Box	Stationery	1	72	72	
Graph Book	Stationery	1	25	25	

BNo	**Shop**	**Customer**
19	SV Stores	Ahmed

Item	**Category**	**Qty**	**Price**	**Cost**	**Total**
Tomatoes	Vegetables/Food	1	40	40	603
Curd	Dairy/Food	1	32	32	
Cupcakes	Packed/Food	2	25	50	
Carrots	Vegetables/Food	1.5	50	75	
Beans	Vegetables/Food	1	45	45	
Onions	Vegetables/Food	0.5	98	49	
Turmeric	Packed/Food	1	82	82	
Ghee	Packed/Food	1	230	230	

BNo	**Shop**	**Customer**
20	Sun General	Suresh

Item	**Category**	**Qty**	**Price**	**Cost**	**Total**
Batteries	Utilities	2	14	28	315
Tomatoes	Vegetables/Food	1.5	80	120	
Spinach	Vegetables/Food	1	15	15	
Bananas	Fruits/Food	4	5	20	
Mosquito coils	Household	1	24	24	
Guava	Fruits/Food	0.4	120	48	
Potato	Vegetables/Food	1.5	40	60	

BNo	**Shop**	**Customer**
21	SV Stores	Ahmed

Item	**Category**	**Qty**	**Price**	**Cost**	**Total**
Tomatoes	Vegetables/Food	2	40	80	622
Curd	Dairy/Food	2	32	64	
Cupcakes	Packed/Food	3	25	75	
Carrots	Vegetables/Food	0.5	50	25	
Onions	Vegetables/Food	1	98	98	
Handwash	Toiletries	1	139	139	

	Bananas	Fruits/Food	12	8	96	
	Eggs	Food	1	45	45	
BNo	**Shop**	**Customer**				
22	SV Stores	Neeraja				
	Item	**Category**	**Qty**	**Price**	**Cost**	**Total**
	Carrots	Vegetables/Food	1	50	50	592
	Horlicks	Packed/Food	1	270	270	
	Chips	Packed/Food	1	20	20	
	Kajal	Cosmetics	1	180	180	
	Milk	Dairy/Food	3	24	72	
BNo	**Shop**	**Customer**				
23	SV Stores	Akshaya				
	Item	**Category**	**Qty**	**Price**	**Cost**	**Total**
	Curd	Dairy/Food	0.5	32	16	128
	Butter	Dairy/Food	0.2	320	64	
	Milk	Dairy/Food	2	24	48	
BNo	**Shop**	**Customer**				
24	SV Stores	Julia				
	Item	**Category**	**Qty**	**Price**	**Cost**	**Total**
	Carrots	Vegetables/Food	1.5	50	75	888
	Bananas	Fruits/Food	12	8	96	
	Curd	Dairy/Food	3	32	96	
	Milk	Dairy/Food	4	24	96	
	Cereal	Packed/Food	2	220	440	
	Maggi	Packed/Food	1	85	85	
BNo	**Shop**	**Customer**				
25	Sun General	Ahmed				
	Item	**Category**	**Qty**	**Price**	**Cost**	**Total**
	Earphones	Electronics	1	210	210	1364
	Phone cover	Accessories	1	140	140	
	Dongle	Electronics	1	790	790	
	A4 sheets	Stationery	200	1	200	
	Ball Pens	Stationery	2	12	24	
BNo	**Shop**	**Customer**				
26	SV Stores	Srivatsan				
	Item	**Category**	**Qty**	**Price**	**Cost**	**Total**
	Beans	Vegetables/Food	1	45	45	276
	Bread	Packed/Food	1	30	30	
	Onions	Vegetables/Food	0.5	98	49	
	Bananas	Fruits/Food	6	8	48	
	Curd	Dairy/Food	1	32	32	
	Milk	Dairy/Food	3	24	72	
BNo	**Shop**	**Customer**				
27	Sun General	Srivatsan				

Item	Category	Qty	Price	Cost	Total
Broom	Household	1	70	70	340
Dustpan	Household	1	45	45	
Floor Cleaner	Household	1	125	125	
Tissue Paper	Household	2	50	100	

BNo	Shop	Customer
28	SV Stores	Neeraja

Item	Category	Qty	Price	Cost	Total
Milk	Dairy/Food	3	24	72	92
Chips	Packed/Food	1	20	20	

BNo	Shop	Customer
29	SV Stores	Vignesh

Item	Category	Qty	Price	Cost	Total
Face Wash	Toiletries	1	89	89	514
Shampoo	Toiletries	1	140	140	
Maggi	Packed/Food	1	85	85	
Chips	Packed/Food	1	20	20	
Chocolates	Packed/Food	4	10	40	
Air Freshener	Toiletries	2	70	140	

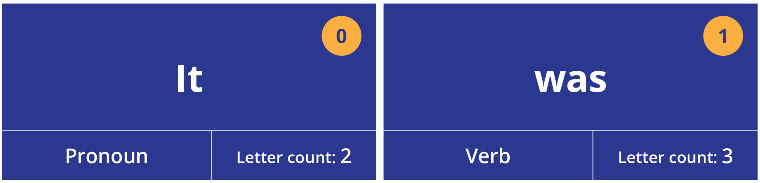
BNo	Shop	Customer
30	SV Stores	Ahmed

Item	Category	Qty	Price	Cost	Total
Chocolates	Packed/Food	1	10	10	106
Curd	Dairy/Food	2	32	64	
Bananas	Fruits/Food	4	8	32	

Figure 1.4: Shopping bill dataset

1.2.3 Words dataset

The basic data element in this dataset (shown below) is one word taken from a text paragraph from the book "Swami and Friends" by R K Narayanan. The first sentence consisting of 4 words is shown below:



0

It

Pronoun

Letter count: 2

1

was

Verb

Letter count: 3

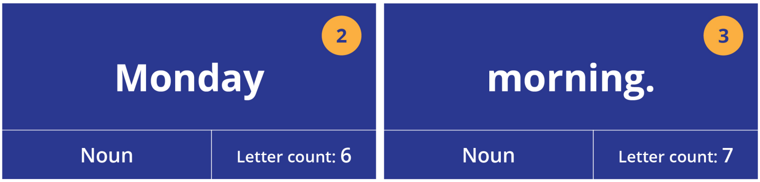


Figure 1.5: Words data elements

The card carries the following fields:

- Unique id no: values 0 to 3 in the cards above
- The word itself
- Part of speech or word category: Pronoun, Verb, Noun
- Letter count: 2, 3, 6 and 7 respectively for the cards above

The last word in the sentence also carries the full stop. Punctuation marks are stored along with corresponding words. Note that full stop (and other punctuation mark) are not counted for the letter count. The part of speech and letter count characteristics are similar to what you may find in a dictionary against that word. The dataset has 65 such cards (words) with a mixture of different parts of speech and number of letters.

Example questions are:

- How many sentences does the paragraph have?
- Which words are occurring with high frequency?
- Are the higher frequency words shorter?
- How does one resolve the pronoun to a personal noun?
- Is the paragraph coherent (i.e. talking about a connected set of nouns)?

The entire data set is contained in the following table:

SNo	**Word**	**Part of Speech**	**Letter count**
0	It	Pronoun	2
1	was	Verb	3
2	Monday	Noun	6
3	morning.	Noun	7
4	Swaminathan	Noun	11
5	was	Verb	3
6	reluctant	Adjective	9
7	to	Preposition	2
8	open	Verb	4

9	his	Pronoun	3
10	eyes.	Noun	4
11	He	Pronoun	2
12	considered	Verb	10
13	Monday	Noun	6
14	specially	Adverb	9
15	unpleasant	Adjective	10
16	in	Preposition	2
17	the	Article	3
18	calendar.	Noun	8
19	After	Preposition	5
20	the	Article	3
21	delicious	Adjective	9
22	freedom	Noun	7
23	of	Preposition	2
24	Saturday	Noun	8
25	And	Conjunction	3
26	Sunday,	Noun	6
27	it	Pronoun	2
28	was	Verb	3
29	difficult	Adjective	9
30	to	Preposition	2
31	get	Verb	3
32	into	Preposition	4
33	the	Article	3
34	Monday	Noun	6
35	mood	Noun	4
36	of	Preposition	2
37	work	Noun	4
38	and	Conjunction	3
39	discipline.	Noun	10
40	He	Pronoun	2
41	shuddered	Verb	9
42	at	Preposition	2
43	the	Article	3
44	very	Adverb	4
45	thought	Noun	7
46	of	Preposition	2
47	school:	Noun	6
48	the	Article	3
49	dismal	Adjective	6
50	yellow	Adjective	6
51	building;	Noun	8
52	the	Article	3
53	fire-eyed	Adjective	8
54	Vedanayagam,	Noun	11
55	his	Pronoun	3
56	class	Noun	5
57	teacher,	Noun	7
58	and	Conjunction	3
59	headmaster	Noun	10

60	with	Preposition	4
61	his	Pronoun	3
62	thin	Adjective	4
63	long	Adjective	4
64	cane . . .	Noun	4

Figure 1.6: Words dataset

1.2.4 Train timetable dataset

1.2.5 Numerical expressions dataset

1.2.6 Rectangles dataset

1.3 Organisation of the book

The book is organised into three parts - the first part introduces the basic concepts underlying computational thinking, the second part gets into the use of computational thinking in data science, and the last part takes the reader through a tour of some of the advanced concepts in computational thinking.

1.3.1 Basics: getting started with computational thinking

If we are asked to find the one thing that differentiates a computer from a human being, the answer that will most likely come up is - repetition. A computer can do repetitive tasks easily, while a human will get bored when asked to repeat something, and will surely make mistakes.

It is thus quite natural to start the exploration into the world of computing with repetition. The key concept that we will use for this is the **iterator** introduced in Chapter 2. The need to keep track of what is going on while repeating something leads us naturally to the concept of a **variable** in Chapter 3. To be able to process the data selectively, we bring in the concept of **filtering** in Chapter 4. To maintain and keep track of different types of things with a reasonable level of sanity, we introduce the concept of **datatype** in Chapter 5.

By allowing the filtering condition to keep changing during the iteration (repetition), we are able to carry out quite sophisticated computations, for instance to search for something, or to find the best candidate meeting some criterion. These investigations form the content of Chapter 6.

To be able to visualise these activities, we use a **flowchart**, that is first introduced in Chapter 2 and developed with increasing complexity through the following chapters. When we find the flowchart too cumbersome to draw, we introduce the concept of **pseudocode** in Chaoter 7 to communicate textually what is contained in the flowchart. Even the pseudocode can become needlessly long and messy, so we organise it using **procedures** in Chapter 8, and find a way to re-use the same pseudocode for different purposes using **parameters**.

1.3.2 Computational thinking for Data Science

Data science deals with sets of data drawn from the physical world. We could use statistical tools to draw general inferences about these data sets. Such tools may work on the entire data set or some random samples drawn from it. Before we can apply these statistical tools, the data sets may need to be first re-organised and formatted into tables.

We could also directly draw many useful (non-statistical) inferences by examining the structure of the dataset captured through the tables above. Drawing such inferences would require us to systematically process the entire dataset to look for interesting relationships between the elements. This processing usually falls into some well defined patterns of computational thinking, which we shall explore in great detail through the chapters in the second part of this book.

We first take up in Chapter 9 the relationship between one data element and some aggregate measure of the entire dataset. For instance, we could find students who are doing well by comparing each student with the average.

We then take up the problem of determining the pairwise relationship between data elements. This requires us to introduce **nested iterations** in Chapter 10. Nested iterations are costly in terms of the time they take to complete execution, so it is useful to find simple heuristics that could cut down the number of pairwise comparisons that are to be carried out.

In Chapter 11, we look at scanning the dataset to collect information of relevance to us and storing this information in a collection called a **list**. By examining multiple lists together we are able to draw some useful inferences about the relationships present in the data. The list could also be ordered for ease of processing (called **sorted lists**), as shown in Chapter 13.

In Chapter 12, we discuss the problems that arise with procedures that have **side effects**, and the care that needs to be taken while using such procedures. Chapter 13 on sorted lists will take advantage of this.

We then turn to clustering or binning the data elements into multiple bins through what is called a **dictionary** described in Chapter 14. Combining dictionaries with lists gives us yet more power to organise the information in a variety of interesting ways. This organisation method described in Chapter 15 resembles the use of a table, but improves on it.

Finally, we try to create a visual representation of the relationship using something called a **graph** in Chapter 16. Questions about the dataset turn out to have answers in terms of properties of these graphs. This gives us a powerful tool to cut the clutter in the data and focus only on what is relevant to answer the question at hand. Chapter 17 explains the different methods of representing a graph using tables or lists and dictionaries, so that the properties we need to study can be determined by suitable procedures on these representations.

1.3.3 Higher level computational thinking

To bring computational thinking closer to what we do in real life, we will need what are called **higher order** computational concepts.

We first discuss the use of **recursion**, in which a procedure P can call itself (self recursion), or can call other procedures, one of which calls P (mutual recursion). It turns out that recursion allows many natural search procedures to be written much more neatly. For instance to find a way to go from one station A to another B using a sequence of trains, we could explore all the trains passing through A, and for each of these trains, we could explore all the stations C on its route, which in turn will explore trains passing through C, and so on, till we reach station B. Recursion is discussed in Chapter 18. Chapter 19 looks at the **tree** structure that is generated by the recursive procedure calls. The reverse also works - namely if we need to use a tree structure to store intermediate information, then this could be accomplished by the use of an appropriate recursion.

The tree structure that emerges when we use recursion to search a graph is called a **spanning tree**, which is like the skeleton of the graph and can be very useful in determining some of the graph properties. Spanning trees are discussed in Chapter 20.

We then move on to the observation that in real life, procedures are typically carried out by someone or something - they don't hang around on their own. The idea of packaging procedures along with the data they need into **objects** is called **encapsulation**. It lies at the core of **object oriented thinking**, a specific form of computational thinking. In object oriented thinking, we ask an object to do something using its data - it does so as best as it can, and returns some useful result. The object may also consult other objects to get this task done. If all this looks very familiar, it is because this is how humans do things. All the forms the content of Chapter 21.

In Chapter 22, we take a diversion into **functional computational thinking**, in which we describe the solution to a problem through a set of rules, each of which is an expression to be computed. The right rule to select depends on the pattern of the input data, and so the selection involves **pattern matching**.

We have missed yet another significant facet of real life computing. In real life, activities do not necessarily go on sequentially, on the other hand, most of the times activities can take place simultaneously or in parallel with one another. This facet is called **concurrency**. How do we think about concurrent computations? This is discussed in Chapter 23. Since input and output from a computer system is an inherently concurrent activity, it provides many use cases for us to study in Chapter 24. We wrap up our discussion of concurrency in Chapter 25 by looking at an example **real time system** consisting of a client connected to a remote web server.

The procedures introduced up to now are all top down (i.e. start from a problem definition and work out a solution). In Chapter 26, we take a bottom up approach to problem solving - a technique made popular through **machine learning**. In this approach, we try to generalise from a few example data elements and check if the generalised model works on new data elements. If it does not work that well, then we adapt the generalised model to fit the new data. In this way, the program is generated through a process of learning from examples.

Chapter 27 concludes by summarising the contents of the book while providing some tips for further reading in each of the areas.

1.4 A guide to the digital companion

Summary of chapter

Difference between logical thinking and computational thinking

Computational thinking is often mistaken for logical thinking. While we need to think logically if we have to do any computational thinking, and logical thinking does involve a sequence of reasoning steps, it would be wrong to conclude that the two are the same.

The first major difference between the two is that logical thinking is *deductive* while computational thinking is *inductive*. Deductive thinking goes from general observations (laws) to the specific (application). For instance in Maths, we could state a theorem describing some general properties of numbers (e.g. squares of integers are positive), and then proceed to verify that it is true for a certain number (e.g. apply it to -2). In contrast, inductive thinking goes from the specific observations to general rules. If we observe that one cow produces milk, we could generalise that all cows produce milk.

The essence of repeating something in computational thinking is clearly inductive. In the example of arranging the tables and chairs for the party, we were able to create the general step of arranging any table by doing it first for one table. This allows us to apply repetition to expand the scope from a single table to all the tables. Similarly, in the hanging clothes on the clothes line example, we could go from one observation of hanging a cloth item to the general step of picking the largest item and hanging it on the clothes line next to the previous one. This allowed us to repeat the general step many times.

The second difference is that logical thinking has to be exact: a logical statement is either true or it is false, and we cannot be ambivalent about its truth. For instance 2 + 2 = 4 is unambiguously true, while 2 - 3 = 0 is unambiguously false. Computational thinking is much more pragmatic - we do not demand exactness, we just want to get the job done in an acceptable way. Consider for instance the problem of finding the best student in the class. Logical thinking may come back with the answer that there is no single best student, since we can always find some parameter (subject) in which each student is weaker than someone else in the class. But computational thinking may come up with a sequence of steps that produces a single student which can be justifiably accepted as a solution to the problem.

The third and most significant difference is that in logical thinking we can go about showing something through refutation. To show that a number $\sqrt{2}$ is irrational, we can assume it is rational and argue that it leads to a contradiction. Since we don't like contradictions, we have to accept that the opposite must be true, i.e. that it cannot be rational - so it must be irrational. This method is simply meaningless in computational thinking, which needs to construct a sequence of steps for producing the answer to a problem. We cannot say we have produced a solution by showing that the assumption that there is no solution leads to an absurdity or contradiction. The sub-discipline of **Constructive Mathematics** bans the use of proofs by contradiction, and would hence bring Maths much closer to computational thinking.

Exercises

2. Iterator

2.1 Going through a dataset

The most common form of computation is a **repetition**. To find the total of the physics marks of all the students, we can start with a total value of zero, then repeatedly select one student and add the student's physics marks to the total. Similarly to determine the number of occurrences of a certain word W in the paragraph, we can repeatedly examine the words one by one, and increment the count if the examined word is W. Consider now that we have to arrange students in a class by their heights. We can do this by making them stand in a line and repeating the following step: pick any two students who are out of order (taller person is before a shorter person), and simply ask them to exchange their places.

The pattern of doing something repetitively is called an **iterator**. Any such repetition will need a way to go through the items systematically, making sure that every item is visited, and that no item is visited twice. This in turn may require the items to be arranged or kept in a certain way. For instance, to arrange students by height, we may first have to make them stand in one line, which makes it easier for us to spot students who are out of order with respect to height. We have to know what to do at any step, and also how to proceed from one item to the next one. Finally, we will need to know when to stop the iterator.

2.1.1 Starting

The steps involved to setup the required context for the iterator to function is collectively called **initialisation**. When we work with cards, the initialisation typically consists of arranging all the cards in a single pile (may or may not be in some order). For instance the marks cards set can be in any order at the start, while for the paragraph words cards set, the words would need to be in the same sequence as they appear in the paragraph.

We may also need to maintain other cards where we write down what we are seeing as we go through the repetition. But this forms part of the discussion in the next chapter. Once we have setup the context for the iterator, we can proceed to the steps that need to be repeated.

2.1.2 Picking and examining one data element

Within the repeated step, we will first have to pick one element to work on. For an arbitrarily ordered pile of cards, any one card can be picked from the pile - the top card, the bottom card, or just any card drawn from the pile at random. If the card pile is ordered (as in the words data set), then it has to be the top card that has to be picked from the pile.

The card that is picked is kept aside for examination. Specific fields in the card may be read out and their values processed. These values may be combined with other values that are written down on other cards (which we will come to in the next chapter). The values on the card could also be altered, though we do not consider examples of this kind in this book. since altering the original data items from the data set can lead to many inadvertent errors.

When we use datasets in tabular form (rather than physical cards), picking a card corresponds to extracting one row of the table. Extracting would mean that the table is reduced by one row (the one extracted). In the case of the ordered datasets, we will always extract the top row of the table.

2.1.3 Marking the data element as seen

Once we are done examining the chosen element, we need to move on to the next data element to process. But that would be exactly the same as what we did in the previous step. For instance with cards, we just have to pick another card from the pile and examine it. So can we just repeat the previous step many times?

In principle we could, but we have not really said what we would do with the card (or extracted table row) that we have kept aside for examination. Would we just return that card to the pile of cards after doing whatever we need to do with its field values? If we did that, then we will eventually come back to the same card that we have picked earlier and examine it once again. Since we don't want to repeat the examination of any card, this means that we cannot return the examined card to the original pile.

What we can do is to create a second pile into which we can move the card that we have examined. Think about this second pile as the pile of "seen" cards, while the original pile would be the pile of "unseen" cards. This ensures that we will never see the same card twice, and also ensures that all the cards in the dataset will definitely be seen. Thus the simple process of maintaining two piles of cards - the original "unseen" pile and the second "seen" pile ensures that we systematically visit all the cards in the pile, without visiting any card twice.

Note that the second pile will also need to be initialised during the initialisation step of the iterator. If we are working with cards, this just consists of making some free space available to keep the seen cards. If we are working with the tabular form, it means that there is a second empty "seen" table kept next to the original "unseen" table of rows, and at each step we extract (remove) one row from the "unseen" table and add the row to the "seen" table.

2.1.4 Stopping

Finally, we have to know when we should stop the iteration. With cards, this naturally happens when there are no further cards to examine. Since we are moving cards from one pile (the "unseen" pile) to another (the "seen" pile), the original "unseen" pile keeps getting smaller and smaller and eventually there will be no more cards in it to examine. We have to stop at this stage as the step of picking a card to examine cannot be performed anymore.

In the tabular form, extracting a row makes the table smaller by one row. Eventually, the table will run out of rows and we have to stop because we cannot perform the extraction step.

2.1.5 Description of the iterator

To summarise, the iterator will execute the following steps one after the other:

- Step 1: **Initialisation step:** arrange all the cards in an "unseen" pile
- Step 2: **Continue or exit?** if there are no more cards in the "unseen" pile, we exit otherwise we continue.
- Step 3: **Repeat step:** pick an element from the "unseen" pile, do whatever we want to with this element, and then move it to another "seen" pile.

- Step 4: Go back to Step 2

Note that the step that checks whether we should continue is written before the step in which we are picking a card for examination, unlike how we described it earlier. This accommodates the strange situation in which the pile given to us is empty to start with. Something like this is never likely to happen when we deal with natural data (which will need to have at least one element), but could happen with other intermediate data sets that we nay generate and iterate over. It is just a convenient way of representing all the possible situations that can occur.

Note also that the step that causes repetition is Step 4, which directs us to go back to the previous step (Step 2) and do it all over again. We could also have written here "Repeat Steps 2 and 3 until we reach exit in step 2" - which is in essence what going back to Step 2 means.

2.2 Flowcharts

The step-wise description of the iterator that we saw in the previous section can be visualised nicely using a diagram called a **flowchart**. The flowchart was developed to describe manual decision making processes - for instance to diagnose what went wrong with a piece of electrical equipment or in an instruction manual to describe the steps for installing a piece of equipment. They are also useful to explain a protocol or procedure to another person, who could be a customer or a business partner.

Our eye is trained to look for visually similar patterns (this is how we identify something to be a tree or a cat for example). The flowchart thus gives us a visual aid that may be useful to spot similar looking patterns much more easily. This in turn may allow us to re-use a solution for one problem to another similar problem situation from a different domain.

2.2.1 Main symbols used in flowcharts

We will mainly be using only 4 symbols from those that are typically found in flowcharts. These symbols are shown in Figure 2.1 along with the explanation of where they need to be used. The box with rounded edges is a terminal symbol used to denote the start and end of the flowchart. The rectangular box is used to write any activity consisting of one or more steps. The diamond is used to represent a decision, where a condition is checked, and if the condition is true, then one of the branches is taken, otherwise the other branch is taken. The arrow is used to connect these symbols.

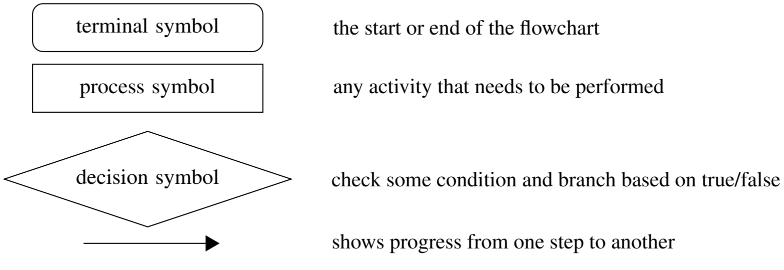

terminal symbol
the start or end of the flowchart
process symbol
any activity that needs to be performed
decision symbol
check some condition and branch based on true/false
shows progress from one step to another

Figure 2.1: Four key symbols of flowcharts

2.2.2 The generic Iterator flowchart

Using the flowchart symbols in Figure 2.1 we can now draw a generic flowchart for the iterator using the description in Section 2.1.5. The result is shown in Figure 2.2. After start comes the initialisation step, followed by a decision if the iteration is to continue or stop. If it is to stop it goes to the end terminal. Otherwise, it goes to the repeat step after which it returns to the decision box.

We can visualise easily the iteration loop in the diagram which goes from the decision box to the repeated steps and back. Any diagram which has such a loop can be modeled using an iterator pattern.

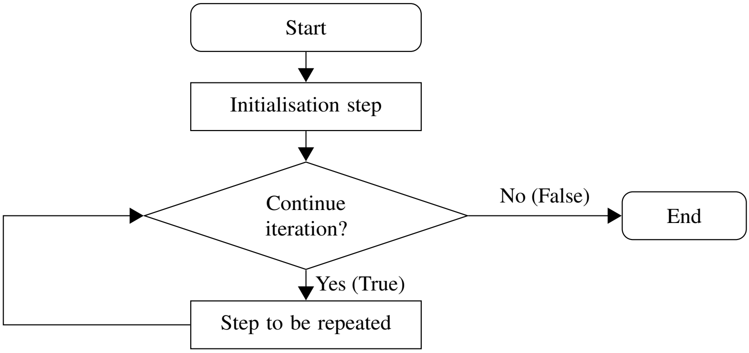


Figure 2.2: Generic iterator flowchart

2.3 Iterator flowchart for cards

The flowchart for a generic iterator can be modified to create one specifically for cards. We need to keep all the cards in an "unseen" pile, and have some free space for the "seen" pile. The decision step involves checking if there are any more cards in the "unseen" pile. If so, we pick one card from the "unseen" pile, examine it, and then move it to the "seen" pile. The result is shown in Figure 2.3.

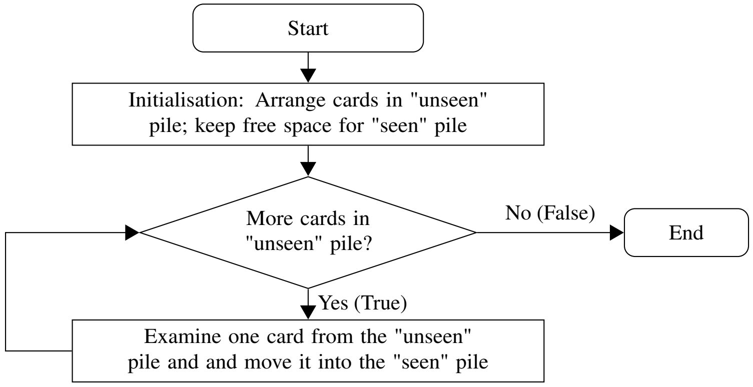

Start
Initialisation: Arrange cards in "unseen" pile; keep free space for "seen" pile
More cards in "unseen" pile?
No (False)
End
Yes (True)
Examine one card from the "unseen" pile and and move it into the "seen" pile

Figure 2.3: Iterator flowchart for cards

Summary of chapter

We can iterate over any collection. Specifically the collection could be a set, or a relation (set of ordered pairs), or a more complex multi-dimensional relation (set of ordered tuples). For instance, we could iterate over the collection of all pairs of cards drawn from the marks card data set. A more interesting example may be iteration over all possible subsets of the marks card data set.

Iterators and complexity: examining the structure of the iterator immediately gives us some clue about the order of complexity of the procedure we are implementing using the iterator. If we are examining all elements of the card set, it will be $O(N)$. If over all pairs, it will be $O(N^2)$, over all k-tuples, it will be $O(N^k)$, and over all subsets, it will be $O(2^N)$.

Iterators and programming language design: Some notes on iterating over different kinds of collections: a number N can be viewed as the template for a higher order function that can apply its first argument N times to its second argument. This can be very neatly encoded in the *lambda calculus*, a formal language that is used to model all possible computational functions. Which means that the number N has a computational meaning, which is "iterating N times". Likewise a generic list without any elements is an iterator that goes through the list doing things with each element. In this, we can construct much more interesting "higher order" analogues to data objects, which make the objects come alive and be active.

Iterators found in programming languages: How programming languages model iteration: Fortran, Python, C, C++, Java, Javascript, Haskell

Exercises

3. Variables

In the previous chapter, we saw how the iterator can be used to repeat some activity many times. Specifically, it can be used to go through the cards systematically, ensuring that we visit each card exactly once. We now look at doing something with the iterator. In order to get any result through iteration, we will need to maintain some extra information. The main vehicle to do this is the **variable**.

When we work with cards, we can use an extra card to note down this intermediate information, so the extra card could be independently numbered or named and would represent a variable. Alternatively, to allow for the card to be better used, we can note down multiple pieces of intermediate information by giving each variable a name, and noting down the intermediate information against each of the named variables on the card. Unlike a field of the card, whose value does not change, and so is called a **constant**, the value of the variable can keep changing during the iteration, which would explain why we call it a *variable*.

If we are using tables to represent the information, then we need some extra space somewhere to record the variable names and their intermediate values.

3.1 Generic flowchart for iterator with variables

We now modify the iterator flowchart for cards shown in Figure 2.3 to include variables to store intermediate information. The basic structure of the iterator remains the same, since we have to examine each card exactly once. But we will need to initialise the variables before the iteration loop starts, and update the variables using the values in the chosen card at each repeat step inside the iteration loop. The result is shown in Figure 3.1 below.

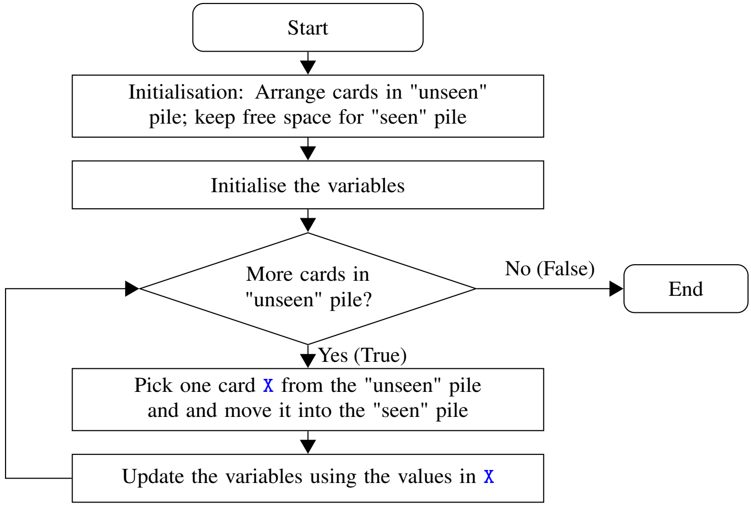

Start
Initialisation: Arrange cards in "unseen" pile; keep free space for "seen" pile
Initialise the variables
More cards in "unseen" pile?
No (False)
End
Yes (True)
Pick one card X from the "unseen" pile and and move it into the "seen" pile
Update the variables using the values in X

Figure 3.1: Generic flowchart for iterator with variables

3.2 Counting

Lets start with the simplest of problems - counting the number of cards in the pile. To find the number of cards, we can go through the cards one at a time (which the iterator will do for us). As we go through each card, we keep track of the count of the cards we have seen so far. How do we maintain this information? We can use a variable for this (written on an extra card). Let us name this variable `count`.

The iterator has an initialisation step, a repeat step and a way to determine when to stop. What would we initialise the variable `count` to at the start of the iteration? Since the variable `count` stores the number of cards seen so far, it needs to be initialised to 0, since at the start we have not seen any cards. At the repeat step, we need to **update** the value of the variable `count` by incrementing it (i.e. by increasing its value by 1). There is nothing extra to be done to determine when to stop, since we will stop exactly when there are no more cards to be seen.

The resulting flowchart is shown in Figure 3.2.

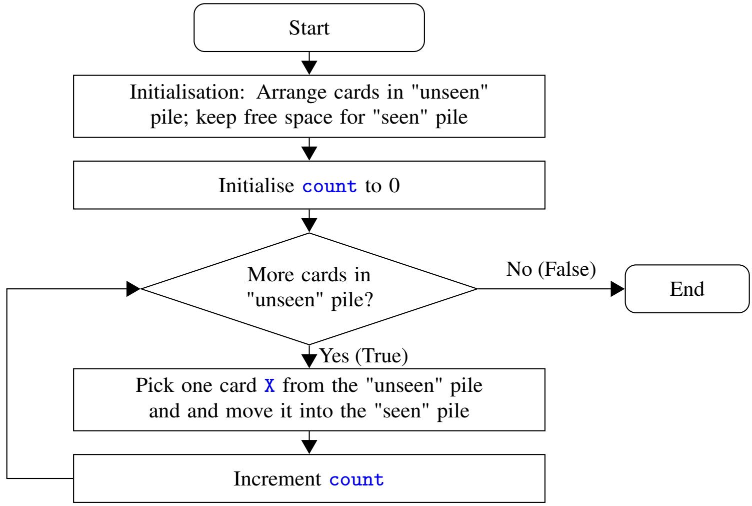

Start
Initialisation: Arrange cards in "unseen" pile; keep free space for "seen" pile
Initialise count to 0
More cards in "unseen" pile?
No (False)
End
Yes (True)
Pick one card X from the "unseen" pile and and move it into the "seen" pile
Increment count

Figure 3.2: Flowchart for counting cards

3.3 Sum

The counting example in the previous section did not require us to examine the contents of the cards at all. We merely moved the cards from one pile to another while keeping track of how many we have seen in a variable called `count`. We now look at something a bit more interesting - that of finding the sum of the values of some field on the card. In the case of the classroom dataset, we could find the sum of the total marks scored by all students put together, or we could find the total marks of all students in one subject - say Maths. In the shopping bill dataset, we could find the total spend of all the customers put together. In the words dataset, we could find the total number of letters in all the words.

3.3.1 Generic flowchart for finding the sum

How do we modify the flowchart in Figure 3.2 to find the sum of some field value? First, we observe that the overall structure of the flowchart is exactly the same. We need to systematically go through all the cards, visiting each card exactly once. This is achieved by putting all the cards in the "unseen" pile, and moving it to the "seen" pile after examining it. However, instead of just keeping count of the cards, we need to do something else.

Instead of the variable count, we could maintain a variable sum to keep track of the total of field values of the cards seen so far. When we have seen no cards, sum is 0, so we can initialise sum to 0 at the initialisation step of the iterator. So far, everything is pretty straightforward and not very different from the flowchart for counting.

The key step that needs to be different is the update step inside the iteration loop. During this step, we have to examine the card we have picked (lets call it card X) and pick out the value of the relevant field. For a field named F, let X.F denote the value of the field named F in the card named X. Then the update step will require us to add X.F to sum.

The resulting flowchart is shown in Figure 3.3. The flowchart for sum is exactly the same as that for counting, except that we have named the element picked in the update step as X, and in the update step, we add the value of the relevant field X.F to the variable sum.

To find the total marks of all students, we can replace X.F in the flowchart above with X.totalMarks. Similarly, to find the total spend of all customers, we can replace X.F by X.billAmount, and for finding the total number of all letters, we can use X.letterCount.

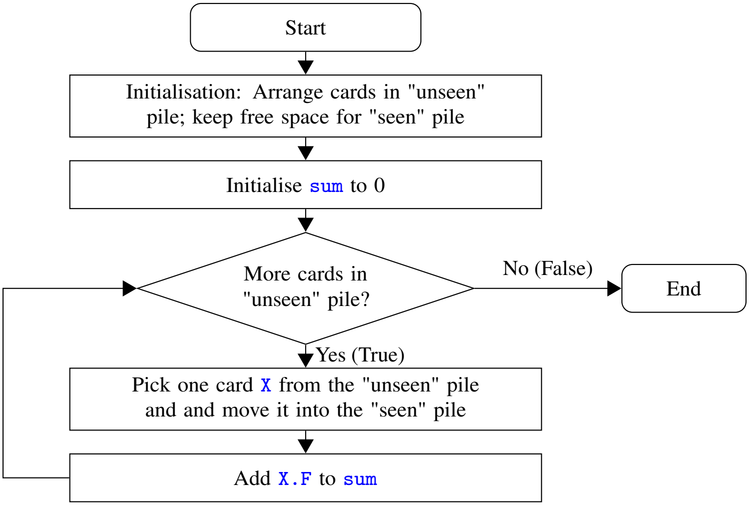

Start
Initialisation: Arrange cards in "unseen" pile; keep free space for "seen" pile
Initialise sum to 0
More cards in "unseen" pile?
No (False)
End
Yes (True)
Pick one card X from the "unseen" pile and and move it into the "seen" pile
Add X.F to sum

Figure 3.3: Generic flowchart for sum

3.3.2 Find the total distance travelled and total travel time of all the trains

3.3.3 Find the total area of all the rectangles

3.4 Average

Suppose now that we wish to find the average value of some field from all the cards in the dataset. For example we could find the average physics marks of all students, or the average bill amount, or the average letter count.

How do we determine the average? We need to find the total value of the field from all the cards and divide this total by the number of cards. We used the flowchart in Figure 3.3 to find the value of the total represented through the variable `sum`. Likewise, the number of cards was represented by the variable `count` whose value was found using the flowchart in Figure 3.2. We can get the average now by simply dividing `sum` by `count`.

The problem with the procedure describe above is that it requires us to make two passes through the cards - once to find `sum` and once again to find `count`. Is it possible to do this with just one pass through the cards? If we maintain both variables `sum` and `count` at the same time while iterating through the cards, we should be able to determine their values together. The flowchart shown in Figure 3.4 below does this.

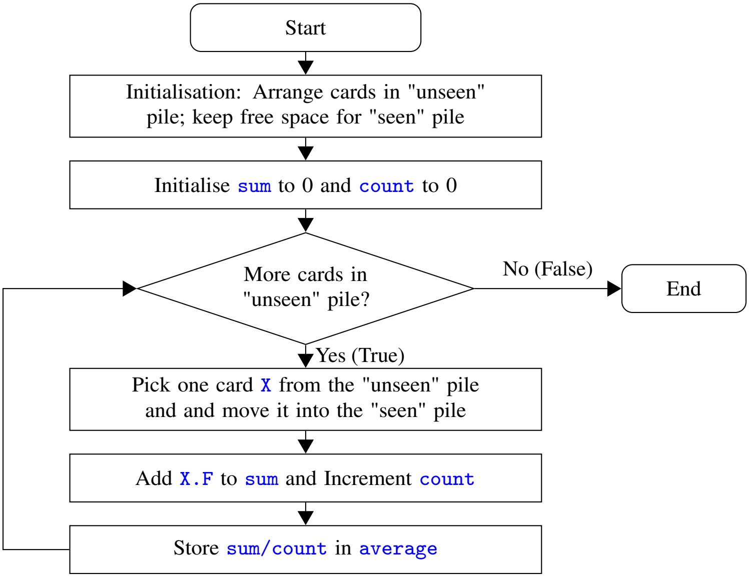

Start
Initialisation: Arrange cards in "unseen" pile; keep free space for "seen" pile
Initialise sum to 0 and count to 0
More cards in "unseen" pile?
No (False)
End
Yes (True)
Pick one card X from the "unseen" pile and and move it into the "seen" pile
Add X.F to sum and Increment count
Store sum/count in average

Figure 3.4: Flowchart for finding average

3.4.1 Generic flowchart for average

The flowchart above works, but seems a bit unsatisfactory. The computation of average is inside the iteration loop and so keeps track of the average of all the cards seen so far. This is unnecessary. We can just find the average after all the cards are seen. The generic flowchart for finding average shown in Figure 3.5 does exactly this.

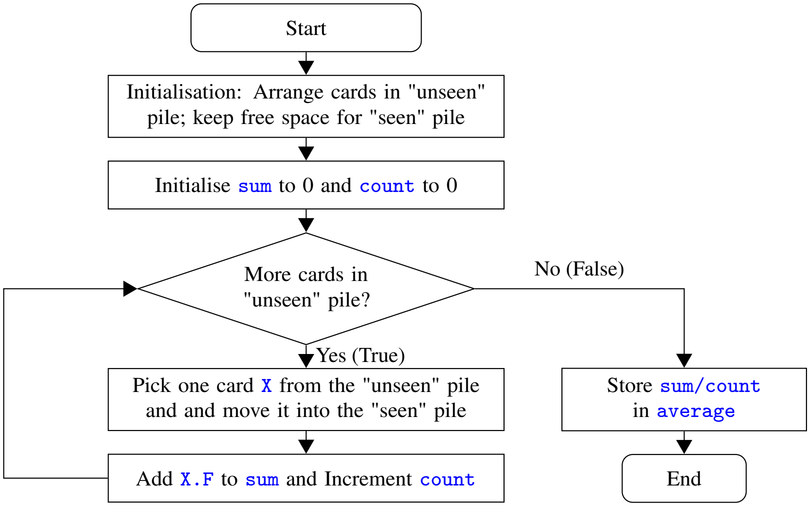

Start
Initialisation: Arrange cards in "unseen" pile; keep free space for "seen" pile
Initialise sum to 0 and count to 0
More cards in "unseen" pile?
No (False)
Yes (True)
Pick one card X from the "unseen" pile and and move it into the "seen" pile
Add X.F to sum and Increment count
Store sum/count in average
End

Figure 3.5: Generic flowchart for finding average

We can use this generic flowchart for finding the average physics marks of all students - just replace `X.F` in the flowchart above by `X.physicsMarks`. Similarly to find the average letter count, we can replace it by `X.letterCount`.

3.4.2 Harder example: Centre of gravity of rectangles

3.4.3 Collecting values in a list

What are the intermediate values that we can store in a variable? Is it only numbers, or can we also store other kinds of data in the variables? We will discuss this in more detail in Chapter 5. But for now, let us consider just the example of collecting field values into a variable using what is called a **list**. An example of a list consisting of marks is [46, 88, 92, 55, 88, 75]. Note that the same element can occur more than once in the list.

The flowchart shown below in Figure 3.6 collects all the Physics marks from all the cards in the list variable `marksList`. It is initialised to [] which represents the empty list. The operation append M to `marksList` adds a mark M to the end of the list - i.e. if `marksList` is $[a_1, a_2, a_3, ..., a_k]$ before we append M, then it will be $[a_1, a_2, a_3, ..., a_k, M]$ after the append operation.

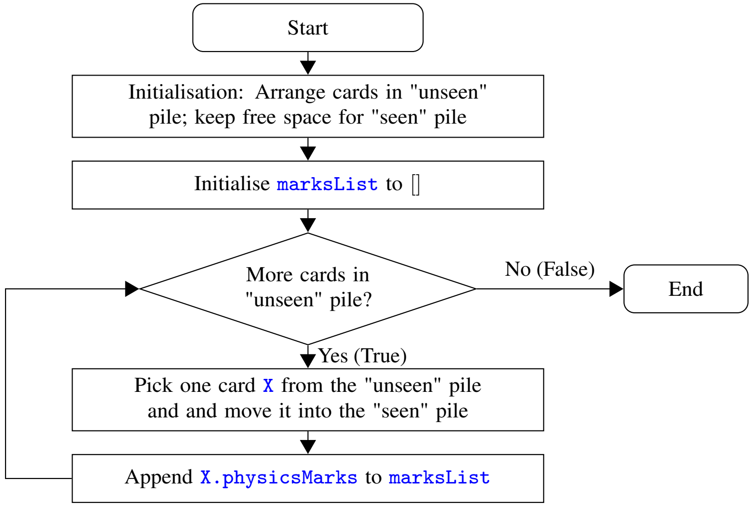

Start
Initialisation: Arrange cards in "unseen" pile; keep free space for "seen" pile
Initialise marksList to []
More cards in "unseen" pile?
No (False)
End
Yes (True)
Pick one card X from the "unseen" pile and and move it into the "seen" pile
Append X.physicsMarks to marksList

Figure 3.6: Flowchart for collecting all the Physics marks

3.5 Accumulator

Note the similarity between the flowchart for sum and that for collecting the list of physics marks. In both cases we have a variable that accumulates something - `sum` accumulates (adds) the total marks, while `marksList` accumulates (appends) the physics marks. The variables were initialised to the value that represents empty - for `sum` this was simply 0, while for `marksList` this was the empty list [].

In general, a pattern in which something is accumulated during the iteration is simply called an **accumulator** and the variable used in such a pattern is called an **accumulator variable**.

We have seen two simple examples of accumulation - addition and collecting items into a list. As another example, consider the problem of finding the product of a set of values. This could be easily done through an accumulator in which the accumulation operation will be multiplication. But in all of these cases, we have not really done any processing of the values that were picked up from the data elements, we just accumulated them into the accumulator variable using the appropriate operation (addition, appending or multiplication).

The general accumulator will also allow us to first process each element through an operation (sometimes called the **map** operation) which is then followed by the accumulation operation (sometimes also called the **reduce** operation).

For instance consider the problem of finding the total number of items purchased by all the customers in the shopping bill dataset. Each shopping bill has a list of items bought, of which some may have fractional quantities (for example 1.5 kg of Tomatoes or 0.5 litres of milk), and the remaining are whole numbers (e.g. 2 shirts). Since it is meaningless to add 1.5 kg of tomatoes to 0.5 litres of milk or to 2 shirts, we could simply take one item row in the shopping bill to be a single packed item - so the number of item rows will be exactly the number of such packed items purchased. This is not very different from what happens at a supermarket when we buy a number of items which are individually packed and sealed. The watchman at the exit gate of the supermarket may check the bags to see if the number of packages in the bill match the number of packages in the shopping cart.

In this example, the map operation will take one bill and return the number of packages (item rows) in the bill. The reduce operation will add the number of packages to the accumulator variable, so that the final value of the accumulator variable will be the total number of packed items purchased.

Summary of chapter

Exercises

4. Filtering

When we counted the cards or found the sum of some field, we did not have to discriminate between the cards. We will now consider situations where we need to do some operation only for selected cards from the dataset. The process of selecting only some cards for processing is called **filtering**.

For instance consider the problem of finding the sum of only the girls' marks. This will require us to examine the card to check if the gender on the card is M (i.e. a boy) or F (i.e. a girl). If it is M, then we can ignore the card. Similarly, if we wish to find the total spend of one customer C, then we check the card to see if the customer name field is C. If not, we just ignore the card.

4.1 Selecting cards: Iterator with filtering

How do we modify the generic iterator flowchart with variables in Figure 3.1 to include filtering? The update step has to be done only for those satisfying the filter condition, so the check needs to be done for each card that is picked from the "unseen" pile just before the variable update step. This ensures that if the check condition turns out false then we can ignore the card and go back to the start of the iteration loop. The resulting flowchart is shown in the Figure 4.1. Note that the No (False) branch from the "Select card?" condition check takes us back to the beginning of the loop.

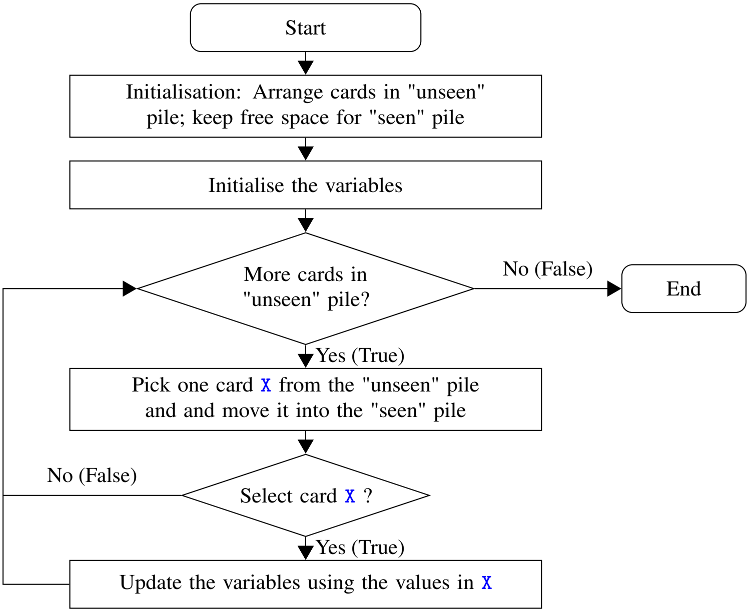

Start
Initialisation: Arrange cards in "unseen" pile; keep free space for "seen" pile
Initialise the variables
More cards in "unseen" pile?
No (False)
End
Yes (True)
Pick one card X from the "unseen" pile and and move it into the "seen" pile
No (False)
Select card X ?
Yes (True)
Update the variables using the values in X

Figure 4.1: Generic flowchart for iterator with filtering

4.2 Examples

Let us now consider examples of applying filtering to find something useful from the datasets.

4.2.1 Total spend of one person

Suppose we want to find the total spend of one of the customers (lets say Ahmed). We can do this by examining all the shopping bills, filtering it by the customer name field (checking if the customer name is Ahmed) and then accumulating the bill amounts in an accumulator variable (lets call this variable `ahmedSpend`). The flowchart for this is shown in Figure 4.2.

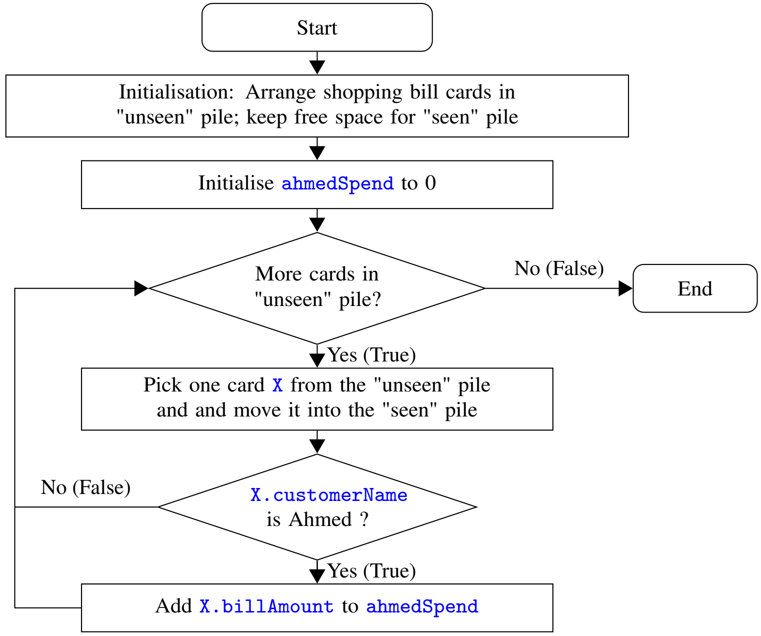

Start
Initialisation: Arrange shopping bill cards in "unseen" pile; keep free space for "seen" pile
Initialise ahmedSpend to 0
More cards in "unseen" pile?
No (False)
End
Yes (True)
Pick one card X from the "unseen" pile and and move it into the "seen" pile
No (False)
X.customerName is Ahmed ?
Yes (True)
Add X.billAmount to ahmedSpend

Figure 4.2: Flowchart for finding the total spend of Ahmed

4.2.2 Number of verbs

Suppose we want to find the number of verbs in the words dataset. This is similar to the customer spend problem, and we should be able to just modify the flowchart 4.2 to create one for this problem. The variable we need to maintain can be called `numVerbs`. It can be initialised to 0 at the initialisation step.

We need to make two more changes - change the filtering condition (replacement for Is `X.customerName` Ahmed?), and change the accumulation step (replacement for Add `X.billAmount` to `ahmedSpend`).

We are looking only for verbs and can ignore all other cards. So the filtering condition should be: X.partOfSpeech is Verb ? The update operation is pretty straightforward, we just increment verbCount (or equivalently, we add 1 to verbCount). The resulting flowchart is shown in 4.3.

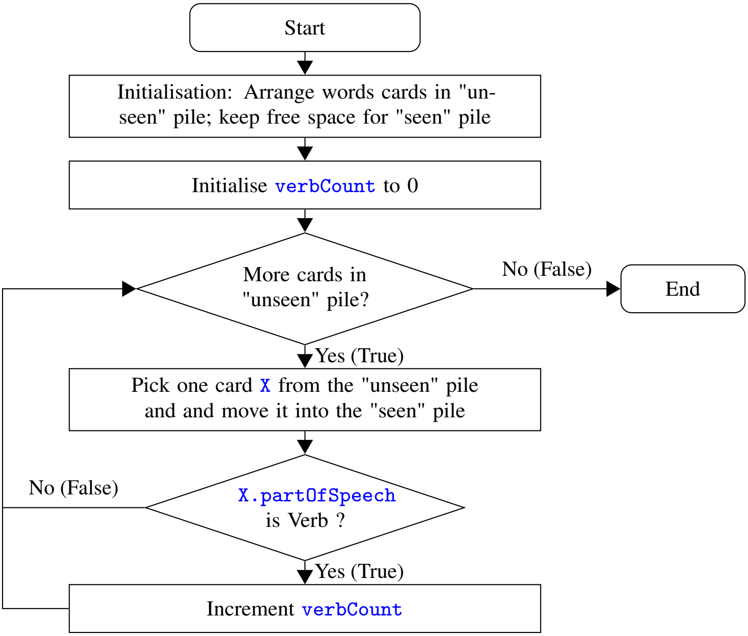

Start
Initialisation: Arrange words cards in "unseen" pile; keep free space for "seen" pile
Initialise verbCount to 0
More cards in "unseen" pile?
No (False)
End
Yes (True)
Pick one card X from the "unseen" pile and and move it into the "seen" pile
No (False)
X.partOfSpeech is Verb ?
Yes (True)
Increment verbCount

Figure 4.3: Flowchart for finding the number of verbs

4.2.3 Sum of both girls' marks and boys' marks

Suppose now that we want to find the total marks of the boys and girls separately. One approach would be to do a filtered iteration checking for boys (flowchart is similar to Figure 4.3), filter condition being X.gender is M. We store the sum in a variable boysTotalMarks. We then proceed to find the girlsTotalMarks in exactly the same way, except that we check if X.gender is F.

The issue with this method is that it needs two passes through the cards, once for the boys and once again for the girls. Can we find both in a single pass? For this, we first note that when we check for the filtering condition (X.gender is M), we are using only one branch - the branch for Yes (True). The other branch for No

(False) is not used, it simply takes us back to the beginning of the loop. Now if the result of the check X.gender is M returns No (False), then it must be the case that the gender is F (since gender can only have one of two values M or F). So we could put some activity on that branch. This is illustrated in Figure 4.4.

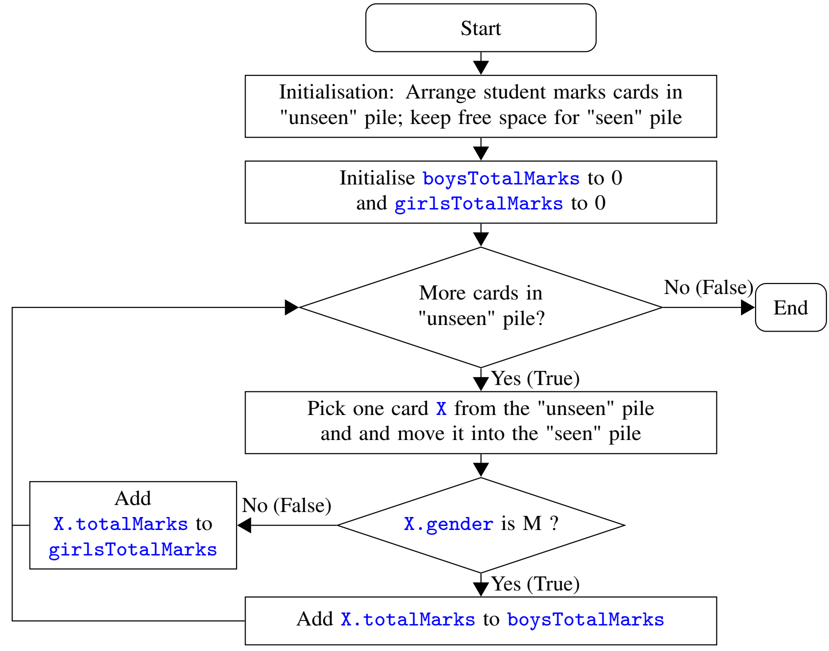

Start
Initialisation: Arrange student marks cards in "unseen" pile; keep free space for "seen" pile
Initialise boysTotalMarks to 0 and girlsTotalMarks to 0
More cards in "unseen" pile?
No (False)
End
Yes (True)
Pick one card X from the "unseen" pile and and move it into the "seen" pile
X.gender is M ?
No (False)
Add X.totalMarks to girlsTotalMarks
Yes (True)
Add X.totalMarks to boysTotalMarks

Figure 4.4: Flowchart for finding total marks for both boys and girls

4.2.4 Total revenues of each shop

In the shopping bills dataset, we could try to find the total revenues earned by each shop. Note that there are three shops - SV Stores, Big Bazaar and Sun General, so we have to find the total value of all the bills generated from each of these shops. Let svr, bbr and sgr be variables that represent the total revenues earned by SV Stores, Big Bazaar and Sun General respectively. We can run one pass with multiple filtering conditions, one following another, first check for SV Stores, then Big Bazaar, and finally for Sun General. This is shown in Figure 4.5.

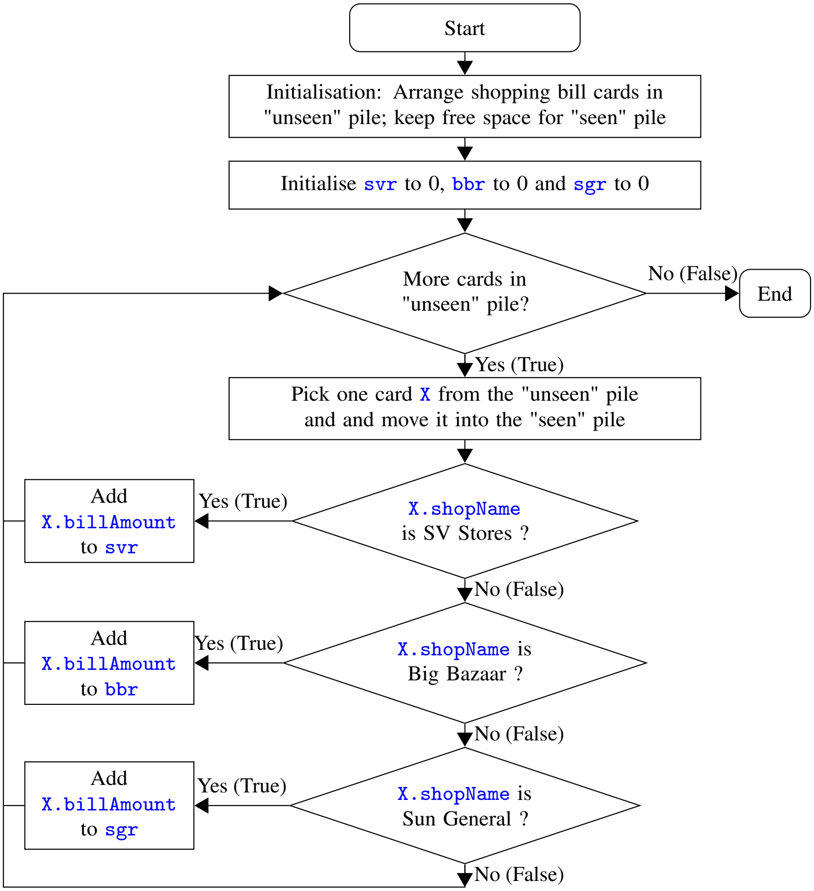

Start
Initialisation: Arrange shopping bill cards in "unseen" pile; keep free space for "seen" pile
Initialise svr to 0, bbr to 0 and sgr to 0
More cards in "unseen" pile?
No (False)
End
Yes (True)
Pick one card X from the "unseen" pile and and move it into the "seen" pile
X.shopName is SV Stores ?
Yes (True)
Add X.billAmount to svr
No (False)
X.shopName is Big Bazaar ?
Yes (True)
Add X.billAmount to bbr
No (False)
X.shopName is Sun General ?
Yes (True)
Add X.billAmount to sgr
No (False)

Figure 4.5: Flowchart for finding the revenues of all the shops

Note that to make the diagram easier to read, the Yes (True) and No (False) branches are switched as compared to the single customer spend flowchart we saw in Figure 4.2.

Observe the flowchart carefully. The straight path through all the filtering conditions in which the shop name is not SV Stores, Big Bazaar or Sun General is not actually possible. However, for completeness it is always good to express all the conditions in the flowchart. If for instance the shop name is wrongly written, or a new shop name is introduced, the flowchart will still work correctly.

4.2.5 Check if a student is at the median

Suppose we want to check if a specific student S is near the median of total marks - i.e. if the students are arranged in increasing order of total marks, the student is somewhere in the middle. Is it possible to do this without first ordering the cards?

Note that the student will be at the median (or in the middle) if the number of students scoring less than the student is more or less equal to those scoring equal to or greater than the student. This can be easily determined by keeping a variable balanced for the difference between these two numbers. If we see a student below, we decrement balanced, and if we see a student equal to or above, we increment balanced. The value of balanced can be used to determine whether the student is in the middle (for instance any value which is between -5 and 5 may be an acceptable definition of middle).

The flowchart in Figure 4.6 implements this, in which ST is the constant that holds the student S's total marks. The flowchart just returns the variable balanced and leaves it to the user to interpret whether the student is in the middle or not.

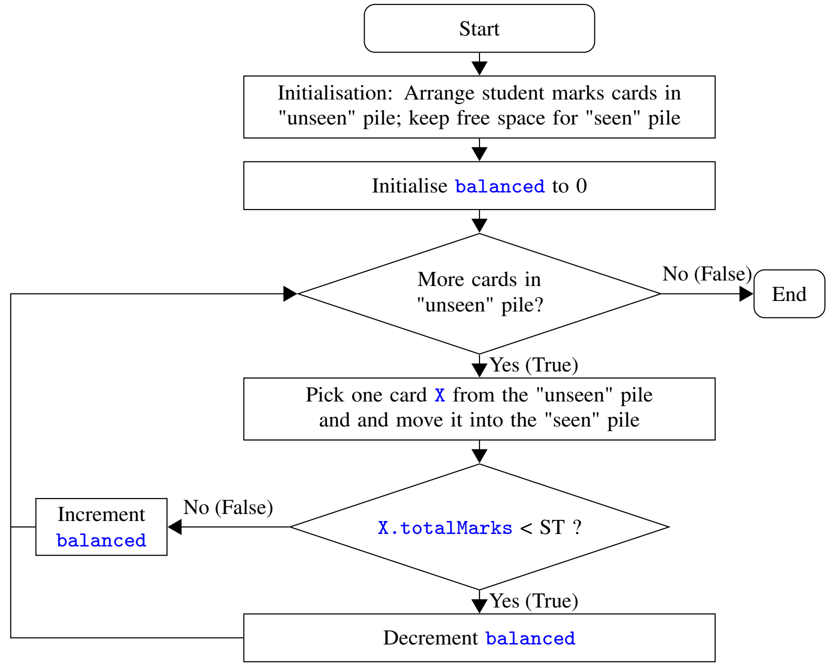

Start
Initialisation: Arrange student marks cards in "unseen" pile; keep free space for "seen" pile
Initialise balanced to 0
More cards in "unseen" pile?
No (False)
End
Yes (True)
Pick one card X from the "unseen" pile and and move it into the "seen" pile
X.totalMarks < ST ?
No (False)
Increment balanced
Yes (True)
Decrement balanced

Figure 4.6: Flowchart for checking if a student is at the median

4.2.6 Number of trains on a specific week day

4.2.7 Number of operators in an expression

4.2.8 Harder Example: Number of words in the sentences

Lets consider a slightly harder example now if finding the number of words in all the sentences of the words dataset.

There are many ways in which this example differs from the ones we have seen so far. Firstly, when we pick a card from the pile, we will always need to pick the topmost (i.e. the first) card. This is to ensure that we are examining the cards in the same sequence as the words in the paragraph. Without this, the collection words will not reveal anything and we cannot even determine what is a sentence. Secondly, how do we detect the end of a sentence? We need to look for a word that ends with a full stop symbol. Any such word will be the last word in a sentence. Finally, there are many sentences, so the result we are expecting is not a single number. As we have seen before we can use a list variable to hold multiple numbers. Lets try and put all this together now.

We use a `count` variable to keep track of the words we have seen so far *within one sentence*. How do we do this? We initialise `count` to 0 at the start of *each sentence*, and increment it every time we see a word. At the end of the sentence (check if the word ends with a full stop), we append the value of `count` into a variable `swcl` which is the sentence word count list. Obviously, we have to initialise the list `swcl` to [] during the iterator initialisation step.

The resulting flowchart is shown in the Figure 4.7 below.

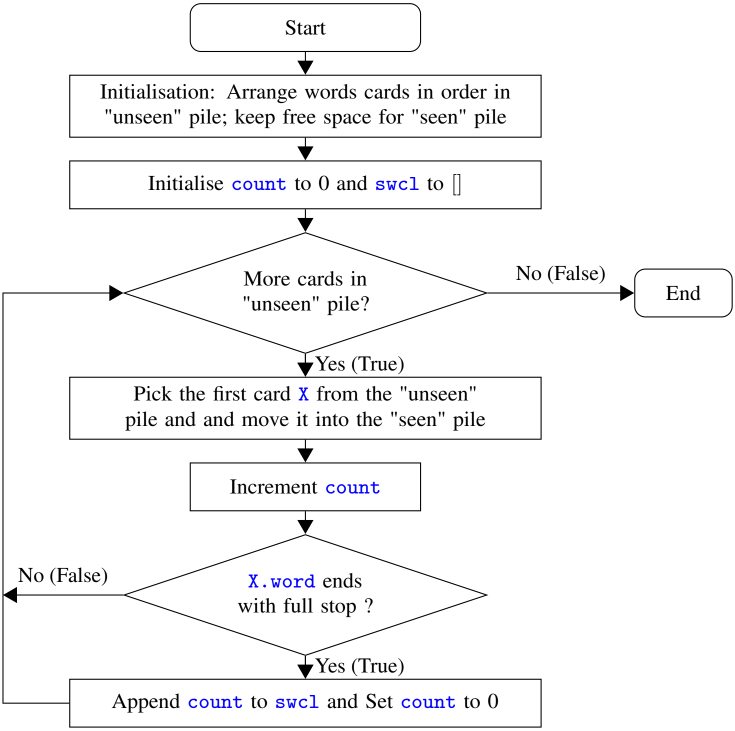

Start
Initialisation: Arrange words cards in order in "unseen" pile; keep free space for "seen" pile
Initialise count to 0 and swcl to []
More cards in "unseen" pile?
No (False)
End
Yes (True)
Pick the first card X from the "unseen" pile and and move it into the "seen" pile
Increment count
X.word ends with full stop ?
No (False)
Yes (True)
Append count to swcl and Set count to 0

Figure 4.7: Flowchart for finding the word count of all sentences

Note that we do increment of `count` before the filtering condition check since the word count in the sentence does not depend on whether we are at the end of the sentence or not. The filtering condition decides whether the `count` needs to be reset to 0 (to start counting words of the next sentence), but before we do the reset, we first append the existing `count` value to the list of word counts `swcl`.

4.3 Compound conditions

In the previous examples, the filtering was done on a single field of the card. In many situations, the filtering condition may involve multiple fields of the card. We now look at examples of such **compound** conditions.

4.3.1 Sum of Chennai girls' marks

As the first example, lets say we want to find the total marks of all the girls from Chennai in the classroom dataset. This requires us to check for two conditions - gender is F and town/city is Chennai. We can build the flowchart for this in the same way that we did the shop revenues flowchart in 4.5, where the conditions are checked one after another. The resulting flowchart is shown in Figure 4.8, where the accumulator variable cgSum holds the required sum of the Chennai girls total marks. Filtering conditions of this kind can be called **nested** since the loop for one condition contains the loop for the next.

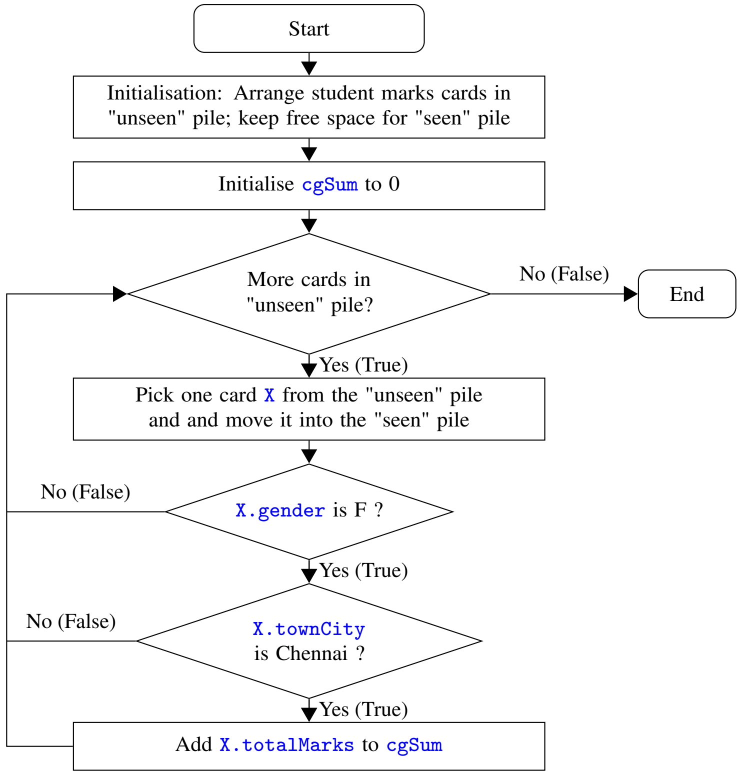

Start
Initialisation: Arrange student marks cards in "unseen" pile; keep free space for "seen" pile
Initialise cgSum to 0
More cards in "unseen" pile?
No (False)
End
Yes (True)
Pick one card X from the "unseen" pile and and move it into the "seen" pile
No (False)
X.gender is F ?
Yes (True)
No (False)
X.townCity is Chennai ?
Yes (True)
Add X.totalMarks to cgSum

Figure 4.8: Flowchart using nested conditions

4.3.2 Compound conditions instead of nested conditions

As we saw in the Chennai girls sum example in Figure 4.8, when we need to check two different fields of the card, we can use nested conditions where one condition is checked after another. This however seems unnecessarily complicated for something that looks a lot simpler. Can we not simply check for both conditions together within the filter decision box?

The way to check for more complex decisions (maybe using multiple data fields) is to use **compound** conditions. Compound conditions are created from the basic conditions by using the operators AND, OR or NOT (called **Boolean operators**).

The redrawn flowchart for finding the sum of all Chennai girls' total marks using a compound condition in the filtering decision box is shown in Figure 4.9 below.

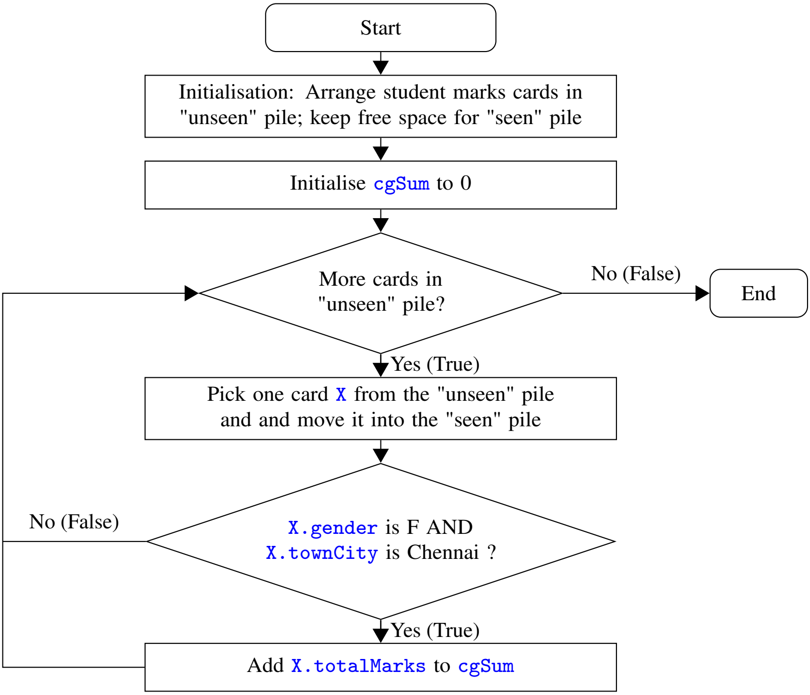

Start
Initialisation: Arrange student marks cards in "unseen" pile; keep free space for "seen" pile
Initialise cgSum to 0
More cards in "unseen" pile?
No (False)
End
Yes (True)
Pick one card X from the "unseen" pile and and move it into the "seen" pile
No (False)
X.gender is F AND X.townCity is Chennai ?
Yes (True)
Add X.totalMarks to cgSum

Figure 4.9: Flowchart using compound condition

4.3.3 Boys born in the first half of the year

Compound decisions may be required even when we have to look at only one field of the card, for instance when we have to check if the value of that field is within a certain range. Consider the problem of finding the number of students who have marks between 60 and 80 in Maths. We have to check if the Maths marks is above 60 (first condition) and that it is below 80 (second condition).

Say we have to find the number of students who were born in the first half of the year, i.e. any day from Jan 1 to Jun 30. However, we have a problem here - while we could compare a students marks with 60 or 80, can we really compare a student's date of birth with two given dates? This will require date to be a number or something else that can be compared. We will discuss this issue in more detail in Chapter 5. For now, assume that $d_1 \leq d_2$ works for two dates d_1 and d_2. The flowchart in Figure 4.10 uses the accumulator countBFH with an appropriate compound condition.

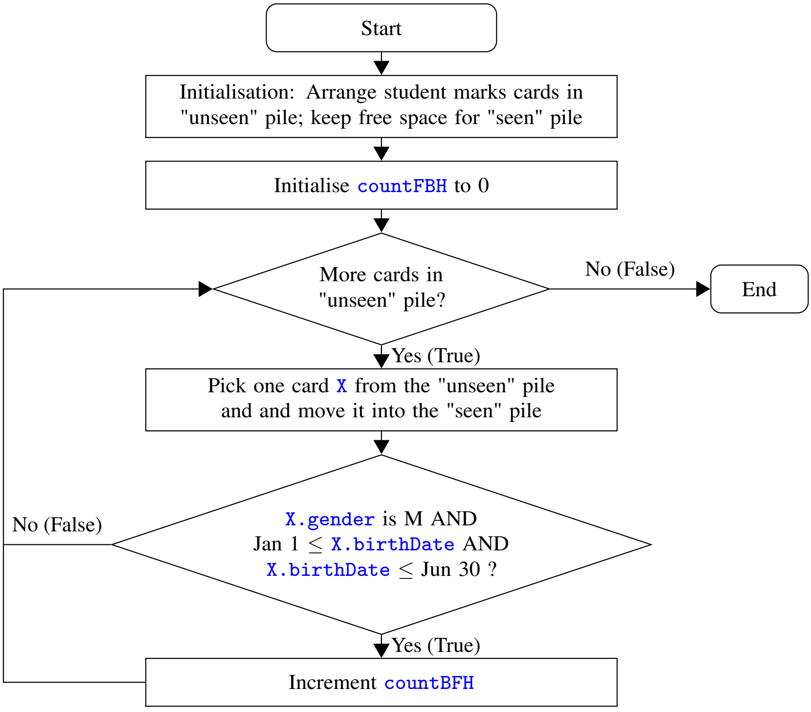

Start
Initialisation: Arrange student marks cards in "unseen" pile; keep free space for "seen" pile
Initialise countFBH to 0
More cards in "unseen" pile?
No (False)
End
Yes (True)
Pick one card X from the "unseen" pile and and move it into the "seen" pile
X.gender is M AND Jan 1 ≤ X.birthDate AND X.birthDate ≤ Jun 30 ?
No (False)
Yes (True)
Increment countBFH

Figure 4.10: Flowchart for number of boys born in the first half of the year

4.3.4 Number of high scorers

Let us say we want to find the number of students who are able to score high marks in at least one subject, where high marks means higher than 90. How would we go about finding this? We have to check the Maths marks, the Physics marks and the Chemistry marks to see if any one of them is above 90 - so we cannot use compounding using the AND Boolean operator as we did in the last example. Luckily, there is another Boolean operator OR which does exactly this - checks if one or more of the component conditions are true. The required flowchart using OR for finding the count of highscorers `countHS` is shown in Figure 4.11.

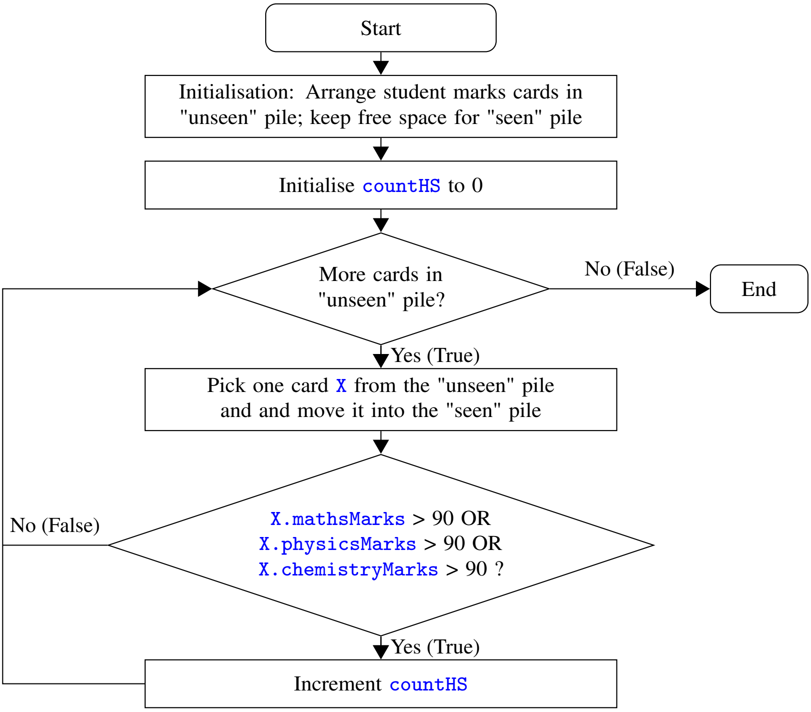

Start
Initialisation: Arrange student marks cards in "unseen" pile; keep free space for "seen" pile
Initialise countHS to 0
More cards in "unseen" pile?
No (False)
End
Yes (True)
Pick one card X from the "unseen" pile and and move it into the "seen" pile
No (False)
X.mathsMarks > 90 OR X.physicsMarks > 90 OR X.chemistryMarks > 90 ?
Yes (True)
Increment countHS

Figure 4.11: Flowchart for counting the high scorers

4.3.5 Sequence of conditions is not the same as AND or OR

We saw that nested conditions can be replaced by AND. At first glance, we may think that a sequence of conditions making the same update to a variable but with different conditions behaves like an OR. Consider for example the flowchart stub in Figure 4.12. This is clearly the same as taking the OR of the three conditions as shown in 4.13.

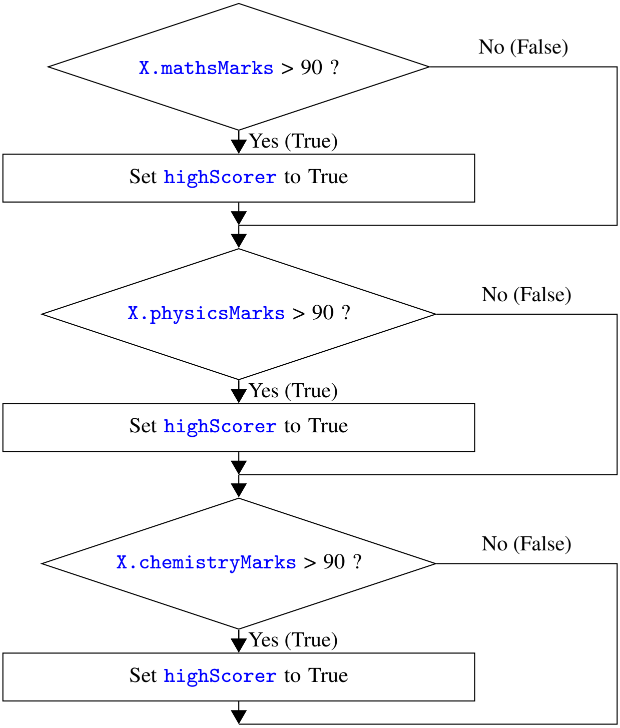


Figure 4.12: Part of flowchart showing sequence of conditions

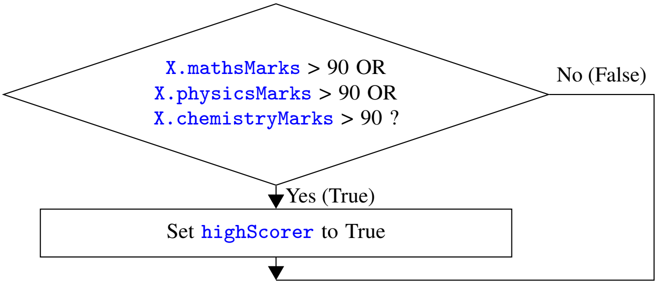


Figure 4.13: Part of flowchart showing OR of conditions

We can quickly check that two flowchart stubs in Figures 4.12 and 4.13 are exactly the same. In both cases, the Boolean variable `highScorer` is set to True if at least one of the three conditions holds.

Now consider the sequence of conditions shown in the stub 4.14. Is this the same as the compound condition shown in the stub 4.15?

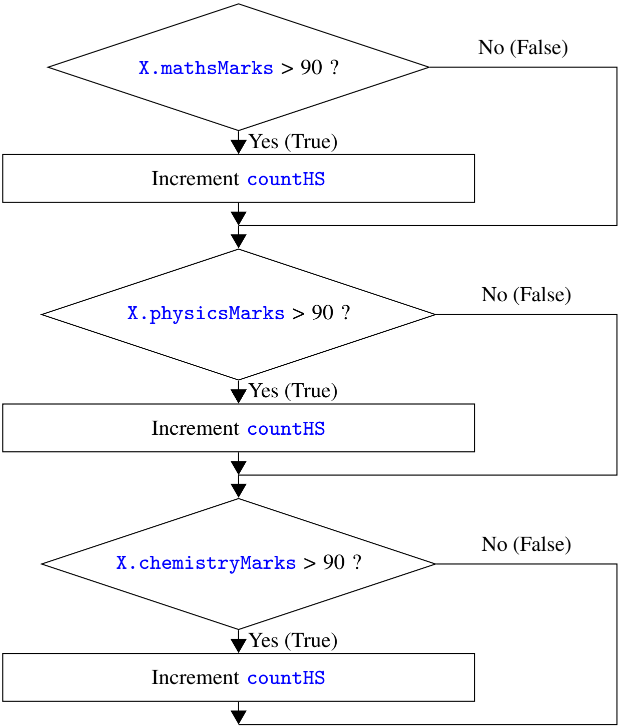


Figure 4.14: Part of flowchart showing sequence of conditions

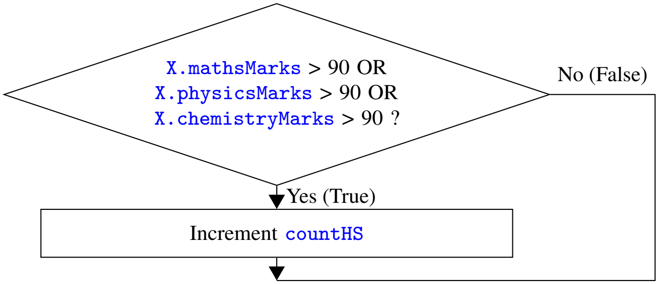


Figure 4.15: Part of flowchart showing OR of conditions

The answer is No ! The flowchart stubs are clearly not equivalent. To see this, consider the case where a student has scored more than 90 in two subjects - say Maths and Physics. Then the stub 4.14 will increment `countHS` *twice* once for the condition `X.mathsMarks` > 90 and again for the condition `X.physicsMarks` > 90. What will the stub in 4.15 do? It will increment `countHS` only *once* (since there is only one compound condition that is checked). So the resulting `countHS` of 4.14 will be 1 more than that of 4.15.

Why is it that the update to a Boolean variable `highScorer` worked, while incrementing the variable `countHS` did not? The difference between the two is that setting the Boolean variable `highScorer` to True twice is just the same as setting it once, whereas incrementing the variable `countHS` twice is not the same as incrementing it once.

This argument also tells us that if the conditions are mutually disjoint (i.e. it is never possible for any two of them to be true at the same time), then the sequence of conditions with exactly the same update operation can be replaced by an OR of the conditions.

In general, a sequence of conditions may or may not resemble OR - we have to analysed the updates more carefully before determining if they are the same or not.

4.3.6 Number of rectangles containing a point

4.4 Looking for a data element

Suppose we want to search for any one card that satisfies some property specified through a condition.

We could in principle just iterate through all the cards looking for a card which satisfies the desired condition. When such a card is found, it is set aside, but we continue through all the remaining cards till the iterator finishes its job. While this works, it seems wasteful to continue with the iteration when we have found what we want. Can't we just stop the iterator when we have finished what we want to do, which is finding some card? The problem is that we have not said how we can stop an iterator - we can start it, we can do something in each step and we can wait till it is over. So far, we are not allowed to stop it mid way.

Let us look at this through an example.

4.4.1 Search for a high scoring student

Let us say that we want to search for a high scoring student, defined as someone who has scored more than 90 in all the subjects. The filtering condition we are looking at can be written as X.mathsMarks > 90 AND X.physicsMarks > 90 AND X.chemistryMarks > 90, which does not change across iterations (except of course that we are applying the same condition to a different X each time). We can write a simple filtered iterator that checks for this condition and accumulates all the student names that satisfies this condition in a accumulator variable that is a list called `highscorers`. But this is wasteful as we are not asking to find *all* students that satisfies the condition, we are only asking for *any one* student who does.

If we want to stop the flowchart when the condition is satisfied, all we will need to do is to exit the loop in the iteration and go from there straight to the End terminal in the flowchart. This is shown in the Figure 4.16 below.

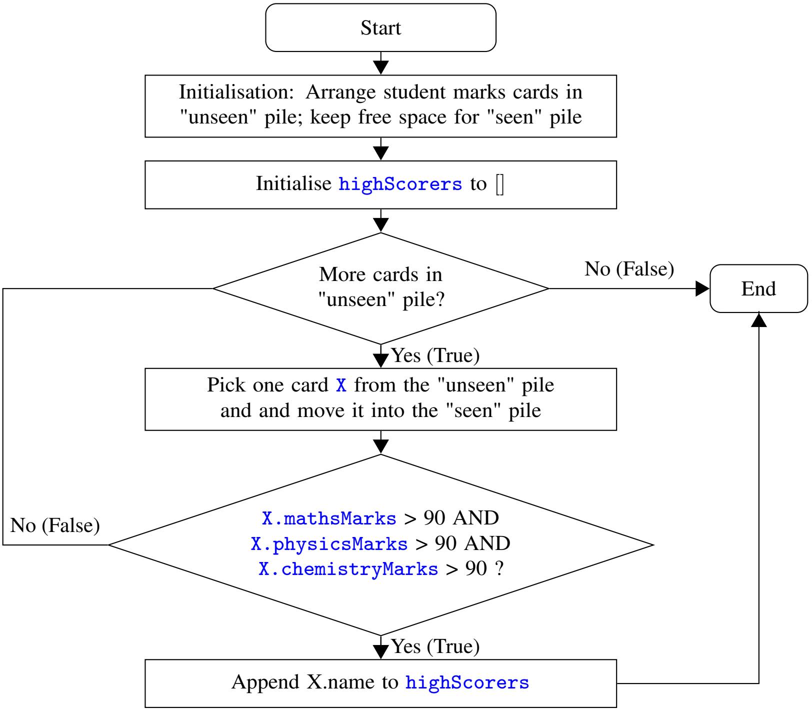

Start
Initialisation: Arrange student marks cards in "unseen" pile; keep free space for "seen" pile
Initialise highScorers to []
More cards in "unseen" pile?
No (False)
End
Yes (True)
Pick one card X from the "unseen" pile and and move it into the "seen" pile
X.mathsMarks > 90 AND
X.physicsMarks > 90 AND
X.chemistryMarks > 90 ?
No (False)
Yes (True)
Append X.name to highScorers

Figure 4.16: Flowchart for searching for a high scorer

Note that rather than go back to the start of the iteration after the append step in the last activity box, we have exited the iteration loop by going directly to the End terminal.

4.4.2 Simplifying the flowchart using compound condition

The flowchart in Figure 4.16 is not very satisfactory as there are two places from which the iterator can exit - either when the iterator is done with its job (visited all the cards) or when the item we are searching for is found. In this example, the step immediately following the completion of the iterator is the End of the procedure, so it did not turn out that bad. However, when we have to do something with the values we generate from the iteration (as we saw for example in the example to find the average in Figure 3.5), we have to know exactly where to go when we exit from the loop.

Can we make the exit happen at the same decision box where the iterator checks for end of iteration? For this we need to extend the exit condition to include something that checks if the required card has been found. We can do this by using a new Boolean variable called found, which is initialised to False and turns True when the required card is found. We then add found to the exit condition box. This is illustrated in the flowchart in 4.17.

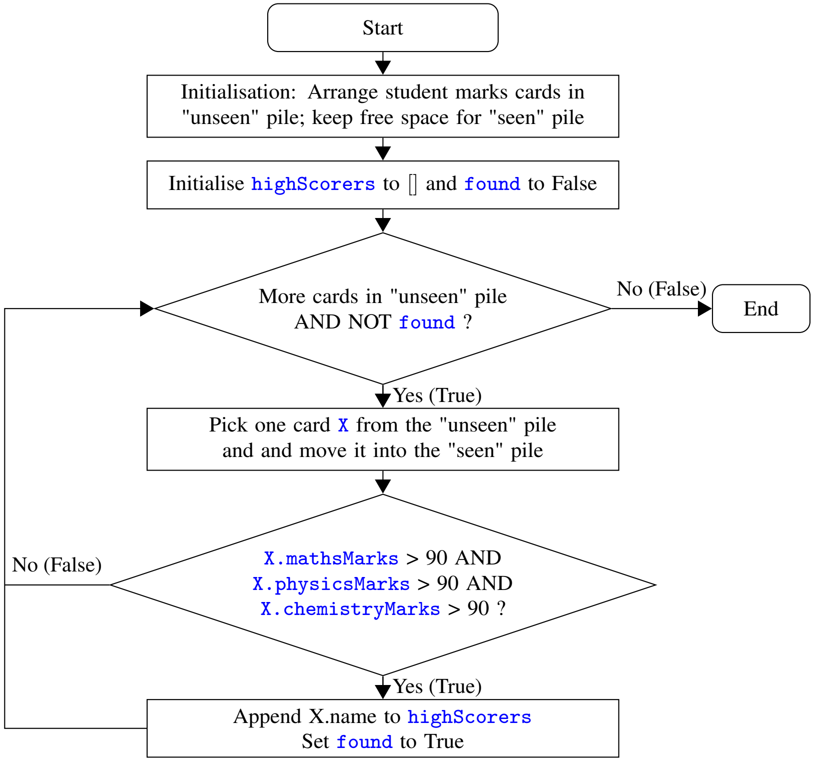

Start
Initialisation: Arrange student marks cards in "unseen" pile; keep free space for "seen" pile
Initialise highScorers to [] and found to False
More cards in "unseen" pile AND NOT found ?
No (False)
End
Yes (True)
Pick one card X from the "unseen" pile and and move it into the "seen" pile
X.mathsMarks > 90 AND X.physicsMarks > 90 AND X.chemistryMarks > 90 ?
No (False)
Yes (True)
Append X.name to highScorers Set found to True

Figure 4.17: High scorer search with single exit point

4.4.3 Generic flowchart for find/search

We can now write the generic flowchart for searching for a card, which can be adapted for other similar situations. The generic flowchart is shown in Figure 4.18.

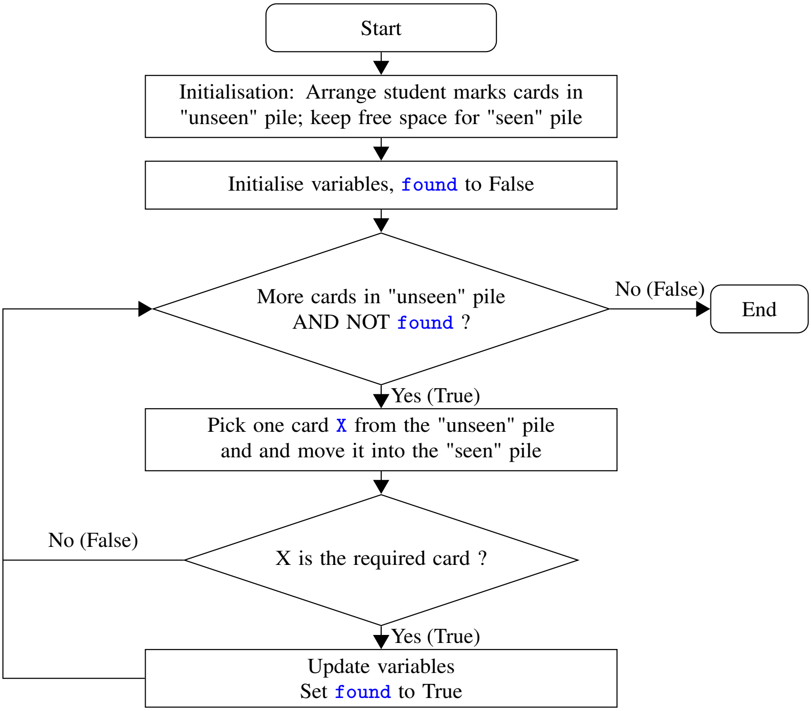

Start
Initialisation: Arrange student marks cards in "unseen" pile; keep free space for "seen" pile
Initialise variables, found to False
More cards in "unseen" pile AND NOT found ?
No (False)
End
Yes (True)
Pick one card X from the "unseen" pile and and move it into the "seen" pile
No (False)
X is the required card ?
Yes (True)
Update variables
Set found to True

Figure 4.18: Generic flowchart for searching for a card

4.4.4 Harder Example: Find a rectangle that is cut in half by a given line

Summary of chapter

Exercises

5. Datatypes

In the datasets that we have considered so far, the field names and values where all correctly written. This is usually not the case in real life, where the actual data is entered by humans who are prone to making mistakes. Field names may be misspelt, data values may have errors in them, and the field names and values may not be properly matched.

The problem with erroneous data is that it creates a significant overload on the computational process. Before we process any data element, we would need to first check the sanity of the data in the card or table row. Every such possible error will need a decision box in the flow chart to identify the error. Then we will need to know what to do if an error is found - do we just ignore the card, or do we try to rectify the error and then go ahead. What are the rules of rectification, i.e. which values would replace the wrong ones?

The flowcharts or pseudocode will be so full of all these kinds of decisions that the main logic/pattern of the problem solution will be lost in all the noise. Worse, when so much checking has to be done, it is quite possible that some will be left out, or the wrong checking logic is implemented in the flowchart or pseudocode. This would let an erroneous data element to pass through the flowchart or pseudocode and give rise to all kinds of unexpected behaviour.

Even if the data elements are correctly written, we could have made errors in how the values in the data element are used within our procedures. For instance, we could attempt to add two name fields, or multiply one date field value with another, or subtract one word from another - all of which are meaningless operations that are very likely to produce some nonsense values. These values may be used later in the procedure, resulting in totally unpredictable outcomes.

Would it not be nice if we could simply specify what it means for the data element to be sane, and what operations are allowed to be performed on it? The development environment of programming system that processes our procedures should be able to take these specifications and automatically check that they are complied with. If not, they can flag errors at processing time, so that the errors can be rectified. This would greatly relieve the writing of the flowcharts or pseudo code, which would no longer need to check for any of these kinds of errors.

In this chapter, we will first look at the different conditions that we can place on the field values in our datasets and the operations allowed to be performed on them. This will lead us to the concept of a **datatype**, which lets us define clear specifications to be written about the data items and the operations permitted on them.

5.1 Sanity of data

Let us first look at some of types of fields we can find in the datasets. Some of the fields hold numbers - like marks, prices, amounts, distances etc. Many of them, for example names, are sequences of alphabet characters, perhaps separated by a space or some special character (like a full stop). We can also see items like the date field which seem to be like numbers, but are written with non-numeric characters - for example Jun 1. We now go through all of these data elements systematically, identifying valid values and operations for each of them.

5.1.1 Classroom dataset

There are four types of fields in the classroom data element:

- Marks: which is a natural number taking values in the range 0 to 100 (in the case of subjects), and 0 to 300 in the case of total marks. he marks field should allow addition and subtraction, but multiplication or division of one mark by another does not make any sense. We should also be allowed to compare two numbers - check if they are equal, less than, greater than etc. In terms of the field names itself, we expect the subject field names to be valid subjects.
- Name, City: Is a sequence of characters which need to be from the alphabet (A-Z or a-z) along with some spacing characters, like blank, full stop or hyphen. We can compare one name or city with another to check for equality, but cannot do any other operation with it. We also expect that the city/town name is from the list of recognised names for cities/towns.
- Gender: is a single character with two possible values - M or F. Again, the only operation allowed should be equality checking.
- Date of birth: is a date, which we have abbreviated to hold only the month and the date in the month. The month can take one of 12 values, one for each month (Jan, Feb, ..., Dec) and the date can take numerical values from 1 to 31 only. We should be able to compare two dates for equality, and to check if one comes before another (so less than, greater than etc should be possible).

There is yet another implicit constraint that the classroom data element must satisfy, which is that the total marks must be equal to the sum of the three subject marks. If this is not the case, then it implies either that one or more of the subject marks are wrongly entered, or that a mistake was made in computing the total. Note that we do not really need to provide the total - it can always be computed from the subject marks. However, we often do provide such redundant fields in data elements precisely for the purpose of sanity checking. If we did not have the total marks written, then some errors in the entry of marks could go undetected. For example, while we will immediately detect a wrong entry with value -5 or 105 (since both these are outside the range of valid values), we will not be able to detect an entry error where the Physics marks (say 45) is wrongly entered also in the Maths field (which should be say 65). But if total is provided, we can catch this error since the total will correctly add 65 for Maths and not 45. Note that this does not ensure that we catch all such errors though - for instance the Maths and Physics marks may get erroneously interchanged. This will not be caught by providing the total, since the total remains the same.

5.1.2 Shopping bill dataset

The shopping bills dataset has these types of fields:

- Quantity, price, amount: are fractional numbers, with possibly 2 decimal places (but not more). We can set some upper limits for each of these. Addition and subtraction is allowed, but not multiplication or division.
- Names of customer, shop and item categories: Is exactly like the name field in the classroom dataset.

In this example as well, we have some redundant values. The amount on each line item should be equal to price multiplied by quantity. Then the total bill amount must equal the sum of all the line item amounts.

5.1.3 Words dataset

The words dataset has these types of fields:

- Letter count: is a number. We can set an upper limit for this based on the longest word in the dictionary. Addition and subtraction is allowed, but not multiplication or division.
- The word itself and the category: Is exactly like the name field, but additional special characters (like comma, semicolon etc) may be allowed for the word. The word should be of the right category, which can also be checked using a dictionary.

Here too we have a redundant value. The letter count needs to be exactly equal to the alphabet letters in the word.

5.1.4 Trains dataset

5.1.5 Expressions dataset

5.1.6 Rectangles dataset

5.2 Basic datatypes

The most natural way of preventing wrong data entries or operations on the data fields is to associate what is called a **datatype** which each of them.

A datatype (or simply **type**) is a way of telling the computer (or another person) how we intend to use a data element:

- What are the values (or range of values) that the element can take ?
- What are the operations that can be performed on the data element ?

Thus when we specify that a variable is of a specific type, we are describing the constraints placed on that variable in terms of the values it can store, and the operations that are permitted on it.

There are three **basic datatypes** that we have used in our datasets. These are:

- Boolean: for holding for instance the True or False values of the filtering conditions
- Integer: for holding numerical values like count, marks, price, amount etc
- Character: for holding single character fields like gender which has M and F as values

We now look at each of these in turn.

5.2.1 Boolean

An element of the **Boolean datatype** takes one of only two values - True of False. So any attempt to store any value other than True or False in a variable of Boolean datatype should result in an error.

What are the operations that should be allowed on elements of this datatype? We need to be able to combine Boolean datatype values and also check if a given Boolean value is True or False. The table given below lists some of the possible operations possible on the Boolean datatype.

Operation	Number of args	Resulting type
AND	2	Boolean
OR	2	Boolean
NOT	1	Boolean
=	2	Boolean

5.2.2 Character

An element of the **Character datatype** take the following kinds of values:

- Alphabet: Capital letters A,B,...,Z or small letters a,b,...,z
- Numeral: Digits 0,1,...,9
- Special characters: Separators like blank, comma, fullstop, semicolon, colon, hyphen, underscore, slash, backslash, exclamation mark, question mark; other special characters like $,#, %, @, ...

The only operation that makes sense for characters is check for equality, which returns Boolean.

Operation	Number of args	Resulting type
=	2	Boolean

5.2.3 Integer

An **Integer datatype** element take the values ...,-3,-2,-1,0,1,2,3,... (i.e. the integer can be a negative number, zero or a positive number.

We can add, subtract, multiply integers. We can also compare two integers. It may or may not be possible to divide one integer by another, unless we define a special integer operation that takes only the quotient after division. These operations are shown in the table below:

Operation	Number of args	Resulting type
$+$	2	Integer
$-$	2	Integer
$\times$	2	Integer
$\div$	2	Integer
$=$	2	Boolean
$>$	2	Boolean
$<$	2	Boolean

5.3 Compound datatypes

We can construct more involved datatypes from the basic datatypes using datatype **constructors**. The resulting datatypes are called **compound** datatypes. The three main kinds of compound datatypes that we have used in our datasets are:

- **strings**: are just sequences of characters of arbitrary length. Our datasets have names of people, shops and cities, shopping item names, categories, and words from a paragraph - all of which are strings.
- **lists**: are a sequence of elements, typically all of the same datatype. Our datasets have for example lists of items in the shopping bill.
- **records**: are a set of named fields along with their associated value which is of some specific datatype. Each card in our dataset is an example of a record - the student marks card, the shopping bill, the word from the paragraph.

We examine each of them in more detail below.

5.3.1 Strings

A string datatype is made by "stringing" together a sequence of characters. There can be any number of characters in the string. There is also no restriction on the types of characters that may be present in the string - we could have letters from the alphabet, numerals or special characters. It is not necessary that the string that we make should make any sense when seen as a word. We will later introduce the notion of **subtype** to place appropriate restrictions on the string datatype to make it more suitable to be used for example as a name.

What are the operations that are allowed on strings? We should be able to append or concatenate one string to another to make a longer string. We should also be able to check if one string is equal to another, or is present at the beginning or at the end or somewhere within another string. The table below shows these operations:

Operation	Number of args	Resulting type
append	2	String
=	2	Boolean
startsWith	2	Boolean
endsWith	2	Boolean
contains	2	Boolean

5.3.2 Lists

Can we collect together several elements, each of which is of some datatype? There are two mechanisms to do this: lists and records.

In **lists**, the elements in the collection are not named, and have to be identified by their position in the list. For instance, in the list $L = [a_1, a_2, a_3, ..., a_k]$, we can pick out a_3, by going down the list L from the beginning till we get to the third element.

Typically, all elements in a list are of the same datatype, though this is not strictly necessary. There is also no restriction on the length of the list.

What are the operations that should be allowed on lists? Firstly, we should be able to check if the list is the empty list []. We should be able to make a longer list by putting two lists together (append one list to another), or by inserting a new element at the beginning of the list. We should also be able to take out the first element from the list.

Operation	No of args	Resulting type	What it does
append	2	List	Appends one list to another
add	2	List	Add an element at start of list
=	2	Boolean	Check if two lists are equal
isEmpty	1	Boolean	Check if the list is empty
head	1	Element's type	First element of the list
tail	1	List	List with first element removed

Note that the last two operations head and tail are possible only if the list is not empty.

5.3.3 Records

Unlike lists, a **record** is a collection of *named* fields, with each field having a name and a value. The value can be of any other datatype. Typically there is no restriction of any kind in terms of the number of fields or on the datatypes of the fields.

What operations would we want to perform on the record? The one that we will need to use for our datasets is that of picking out any field using its name. This operation is simply denoted by ".", where X.F returns the value of the field F from the record X. The result type is the same as the type of the field F.

5.4 Subtypes

A **subtype** of a datatype is defined by placing restrictions on the range of values that can be taken, limiting the kinds of operations that can be performed, and maybe adding more constraints on the element to ensure that there is sufficient sanity in the data values.

We now look at each of the subtypes that we have used in our datasets.

5.4.1 Subtypes of Character

The gender field in the classroom dataset is an example of a subtype of the Character basic datatype. This field is allowed only one of two character values, M or F. The equality operation of the Character datatype can also be applied to the Gender subtype.

5.4.2 Subtypes of Integer

There are a number of places in our datasets where we have used integers with restrictions. These are listed below:

- Sequence no: should have only positive values (negative values and 0 is disallowed). Let us name this datatype **SeqNo**. Since we are not likely to have a very large number of data items in the dataset, we could also set a MAX value for the upper limit of the range of data values. None of the operations that are allowed on integers $(+,-,\times,\div)$ make sense for SeqNo. The only meaningful operation is the = operation that checks if two SeqNo elements have the same value.
- Marks: should have values starting from 0 (i.e. negative numbers are not allowed). For the subject marks, we call the datatype **Marks** and let its value range from 0 to 100 (since in our dataset, subject marks cannot exceed 100). For the total marks field, we can call the datatype **TMarks** and let its values range from 0 to 300. While we can add and subtract marks, multiplication and division of one mark with another mark makes no sense at all. All the comparison operations (=, > and <) can now be applied to marks.
- Count: allows 0 or any positive integer value (negative numbers are disallowed) upto a sufficiently high limit MAX. The words dataset has a field letter count. The variable count was also used to hold the number of cards seen so far in iterations. We can simply write **Count** to represent this datatype. We can add or subtract counts, but multiplication and division does not make sense. Comparison opertaions are all possible.

The table below summarises all of these restrictions.

Subtype	Values allowed	Operations allowed
SeqNo	0..MAX	=
Marks	0..100	$+, -, =, <, >$
TMarks	0..300	$+, -, =, <, >$
Count	0..MAX	$+, -, =, <, >$

5.4.3 Subtypes of String

We have used strings in our datasets with many restrictions. These are listed below:

- Names: should have only alphabet characters and could have blank or fullstop characters additionally to separate the parts of the name. Let us call this datatype *Names*. All the string operations can be allowed for Names. We could also add some new operations - for instance to extract the first and last name.
- City: is exactly the same as Names. We could call this subtype *City*.
- Words: All strings can be considered as words, including those with numerals and special characters. Let us call this subtype *Words*. All string operations are valid for Words
- Word category: The values need to be specific strings from the set *WordCategoryNames* = {"Noun", "Pronoun", "Verb", "Adjective", ...}. Let us call this datatype *WordCategory*. Except =, all the other operations should be disallowed for this subtype.
- Items: values need to be specific strings which represent legitimate item names, *ItemNames* = {"Carrots", "Soap", "Socks", ...}. As for *WordCategory*, only the = operation is admissible. Let us call this datatype *Items*.
- Item category: Values need to be specific strings form the set *ItemCategoryNames* = {"Vegetables/Food", "Toiletries", "Footwear/Apparel", ...}. Again, except =, all other operations should be disallowed. Let us call this subtype *ItemCategory*.

The table below summarises all of these restrictions.

Subtype	Values allowed	Ops allowed
Names	Only alphabet chars with blank, fullstop	All
City	Only alphabet chars with blank, fullstop	All
Words	All chars allowed	All
WordCategory	*WordCategoryNames*	Only =
Items	*ItemNames*	Only =
ItemCategory	*ItemCategoryNames*	Only =

5.5 Transforming the data element

We have more or less taken care of most of the fields present in our datasets. But there are a few tricky ones that we have not yet dealt with - the date of birth field in the classroom dataset, the fractional quantity and prices in the shopping bills. Even for the marks field which consists of whole numbers, the average marks could be fractional.

In each of these cases, we have many options for storing the data. Some of these options may not really suitable as they may not permit the kinds of operations that we want to perform on them. We may have to first transform the data element to ensure that we can easily store it, while allowing the operations we want to perform on it.

5.5.1 Date

Let us first consider the date of birth field. We have chosen only to represent the month and date (within the month) values, and have ignored the year of birth. This is sufficient for the questions we want to ask (mostly to determine birthday within the year).

Now, a typical value of this field, for instance "6 Dec", looks like a string. So it is probably natural for us to store this field as a string, with suitable restrictions on values that can be taken. However, using a string is not very satisfactory since it will not allow us to compare one date D1 with another D2 to check if D1 < D2 (which would mean that D1 comes earlier in the year than D2). We may also want to subtract D2 from D1 (to find how many days are there between birth date D1 and D2). All of this means that we need something other than string to represent date.

We can apply the following simple transformation to the date field to make it suitable for the comparison and subtraction operations. Since we are only looking at the month and date within the month, we can represent the date using a single whole number representing the number of days starting from Jan 1, which is taken as day 0. So Jan 6 would be represented by the number 5, Jan 31 would be represented by 30 and Feb 1 would be represented by 31. What should be the number representing Mar 1? This depends on whether the year is a leap year (in which Feb will have 29 days) or not. This additional bit of information is required to correctly store the date of birth as a whole number.

Let us say it is not a leap year, so that Feb has 28 days. Then Mar 1 will be represented by the number 59 (31 days in Jan and 28 days in Feb adds to 59). What is the number that should be associated with Dec 6? Note that the number used to represent a date is just the number of days in the calendar year preceding that date. We can thus determine the associated date for all the dates in the calendar. Dec 31 will have the number 364 associated with it (since that is the number of days preceding Dec 31 in the calendar). So Dec 6 should be represented by 339.

Since date is stored as a whole number, we can perform comparison operations on it easily. To check if one date D1 precedes another D2, we just have to check that the number corresponding to D1 is less than that corresponding to D2. Similarly to find the number of days between two calendar dates, we can just subtract their associated whole numbers.

We will also need two convenience operations to go between strings and numbers. The first store(D) will convert a date string D (say "6 Dec") into its number equivalent - this will require us to add up all the days in the months Jan to Nov and add 5 to it to get the number 339 representing "6 Dec". In the other direction, print(N) will take a number in the range 0 to 364, and print the date string corresponding to it. This is important to do, otherwise any external user will have to go through the tedious and error prone task of finding out which date we are referring to using a number. Thus, store("6 Dec") = 339, and print(339) = "6 Dec".

Note now that after this transformation, date just becomes an integer, so we can perform any integer like operations on it. Of course, multiplying two dates does not make any sense, so we should place restrictions on the operations that can be performed. The values also have to be constrained to lie within the range 0 to 364. To take care of these constraints, we could define date as a subtype of integer. All the operations needed to work on the date field can now be shown in the table below:

Operation	No of args	Result type	What it does
=	2	Boolean	Checks if two dates are equal
<	2	Boolean	Checks if one date precedes another
−	2	Integer	Gap between two dates
store	1	Integer	Stores string in a date subtype
print	1	String	Converts date subtype into a string

5.5.2 Fractional values: quantity, price and marks

In the shopping bill dataset, the price, quantity, cost and final bill amount could all be fractional values. We could try storing these in an integer after rounding off the fractional value to the nearest integer, but this means that some information is lost.

There is yet another basic datatype that could be used for storing fractional values, which is called **float**. However, this is a very elaborate datatype that can allow a very wide range of values from the very small or the very large (both of which would need exponents to write). We have some simple values with at most two fractional decimal places, so the use of float may be excessive for us. Is there something simple we could do?

The simplest way of dealing with this would be to simply multiply the data values by 100. This will make the value into an integer (for instance 62.75 will be converted into 6275, which is an integer). Like in the date example, we will need two convenience functions store(62.75) which will return 6275 to store it as an integer subtype, and print(6275) which will convert the integer subtype into a string "62.75" which can be presented when needed.

We can now define Quantity, Price, Cost and Amount as subtypes of Integer. What operations should be allowed on these subtypes? Clearly addition should be allowed, since we can accumulate quantities and costs. Subtraction could also be allowed to determine for instance price difference between two items.

But multiplication and division do not make any sense for these quantities. Luckily, this is rather convenient for us, because if we needed to multiply two fractional numbers, then our convenient mechanism of multiplying them by 100 would fail ! Note that print (store(a) × store (b)) is not the fractional value a × b, since it multiplies both a and b by 100 and so when we take their product, it is multiplied by 10000 in the subtype, and print only reduces this product by 100, so the result is 100 times too large. Division does exactly the reverse, it makes the number 100 times too small after division. Since we don't need to multiply or divide, we do not have to worry about these anomalies introduced due to our transformation.

We can also compare any two of these subtypes using =, < or >. The operations allowed on the subtypes Quantity, Price, Cost and Amount are shown in the table below:

Operation	No of args	Result type	What it does
$+$	2	Integer	Add two elements
$-$	2	Integer	Subtract two elements
=	2	Boolean	Compares for equality
<	2	Boolean	Compared for less than
>	2	Boolean	Compared for greater than
store	1	Integer	Stores string in the subtype
print	1	String	Converts subtype into a string

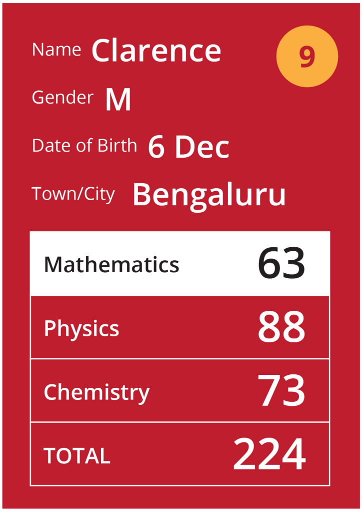
5.6 Datatypes for the elements in the dataset

We now have suitable datatypes for all the fields in our datasets. Let us put these together to make the right datatypes for each of the data elements in our dataset.

The main instrument for us to construct the datatype for the entire data element is the record, since each data element has some named fields which takes values from some datatype. Let us see what kind of records are produced for each of the datasets.

5.6.1 Student datatype

Let us take a relook at the classroom dataset element, which is the marks card of one student.



Name Clarence

9

Gender M

Date of Birth 6 Dec

Town/City Bengaluru

Mathematics	63
Physics	88
Chemistry	73
TOTAL	224

Figure 5.1: Grade card data element

We note that the card carries the following fields, so we can create a record datatype StudentMarks whose field names along with their respective datatypes is shown below:

- uniqueId: SeqNo
- studentName: Names

- gender: Gender
- dateOfBirth: Date
- townCity: City
- mathsMarks: Marks
- physicsMarks: Marks
- chemistryMarks: Marks
- totalMarks: TMarks

What should be the datatype for the entire classroom dataset ? Clearly it is simply the compound datatype: List of StudentMarks.

5.6.2 Word datatype

Let us now turn to the Words dataset, an element of which is pictured below.

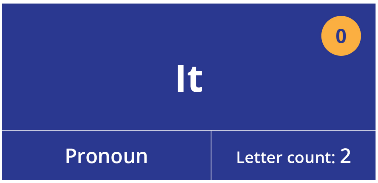


Figure 5.2: Words data element

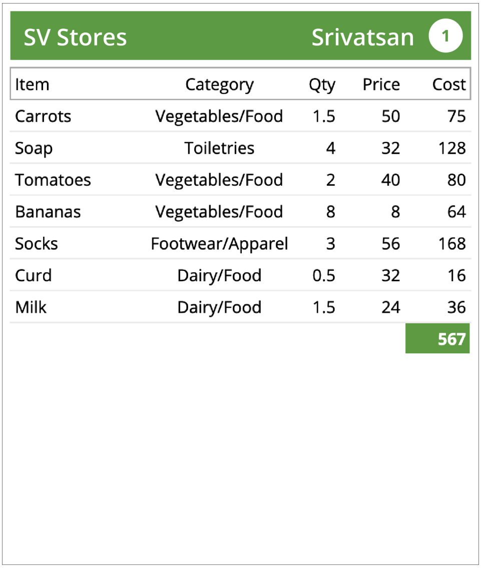
Clearly, we can make a record datatype ParagraphWord for this, whose field names and respective types are as shown below:

- uniqueId: SeqNo
- word: Words
- partOfSpeech: WordCategory
- letterCount: Count

What should be the datatype for the entire Words dataset ? Clearly it is simply the compound datatype: List of ParagraphWord.

5.6.3 Shopping bill datatype

Consider now the shopping bill dataset element pictured below.



SV Stores Srivatsan 1

Item	Category	Qty	Price	Cost
Carrots	Vegetables/Food	1.5	50	75
Soap	Toiletries	4	32	128
Tomatoes	Vegetables/Food	2	40	80
Bananas	Vegetables/Food	8	8	64
Socks	Footwear/Apparel	3	56	168
Curd	Dairy/Food	0.5	32	16
Milk	Dairy/Food	1.5	24	36
				567

Figure 5.3: Shopping bill data element

We can attempt to make a record for this. The fields which are obvious are shown below along with their respective datatypes.

- uniqueId: SeqNo
- storeName: Names
- customerName: Names
- billAmount: Amount

But what do we do with the items purchased? Note that there are a number of purchased items, and that the number may vary from bill to bill. The obvious datatype to use for this is a *List*. But a list of what type?

Each purchased item has the item name, category name/sub category name, quantity purchased, price per unit and cost of item purchased. So the item itself looks like a record. Let us try to make a record datatype called BillItem with fields and datatypes as shown below:

- itemName: Items
- category: ItemCategory
- quantity: Quantity
- price: Price
- cost: Cost

Now that we have the datatype for a line in the shopping bill, we can make a List of BillItem and use that as a field in the shopping bill. The final record ShoppingBill will have the following fields:

- uniqueId: SeqNo
- storeName: Names
- customerName: Names
- items: List of BillItem
- billAmount: Amount

What should be the datatype for the entire shopping bill dataset ? Clearly it is simply the compound datatype: List of ShoppingBill. Note that the shopping bill dataset is a list, each of its elements has within it another list of type BillItem. So it is a list of lists.

5.6.4 Train datatype

5.6.5 Station datatype

5.6.6 Expression datatype

5.6.7 Rectangle datatype

Summary of chapter

Exercises

6. Filtering: dynamic conditions

The filtering conditions in the examples considered in Chapter 4 were all of the type where we compared a element's field with a constant value. In other words, the filtering conditions were constant and not varying across successive repeat steps of the iterator. In this chapter, we look at situations where we have to compare the element's field values with the value of a variable (rather than a constant value). This would make the filtering condition itself change as the variable changes its value, i.e. the filtering condition is dynamic.

6.1 Maximum

We first consider the problem of finding the maximum value of some (numerical) field in the dataset. The method we are going to use for finding the maximum is this: we keep a variable called `max` which will carry the maximum value seen so far (for the desired field). At the end of the iteration, all the cards will have been seen, and `max` will carry the maximum value of the field. When we pick up a card `X`, we check if its field `X.F` has a value higher than the current value of `max`. If so, `max` will need to be updated with this new high value `X.F`. Note that this means that the value of `max` can only keep increasing as we proceed through the cards.

What should the initial value of `max` be? We could start with something that is lower than all the field values. In our dataset, we don't have any fields with negative values, so we can safely initialise `max` to 0 at the start.

The generic flowchart for finding the maximum of field F is shown in Figure 6.1.

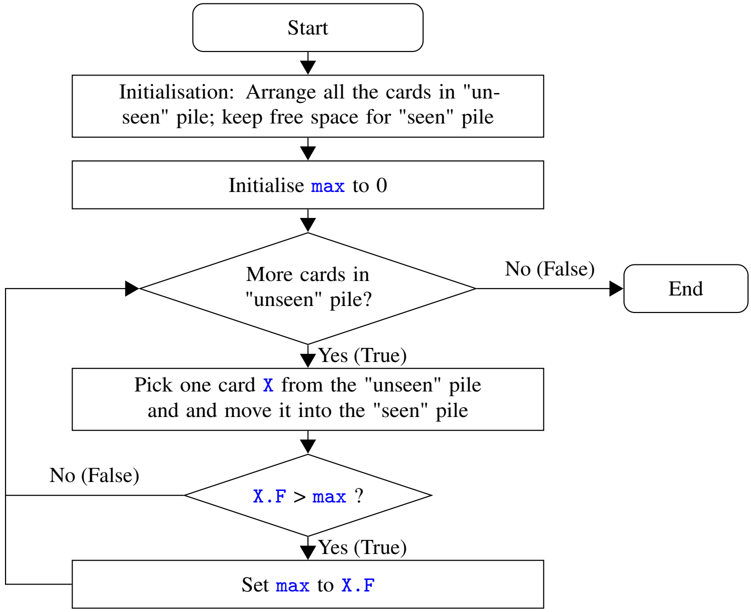

Start
Initialisation: Arrange all the cards in "unseen" pile; keep free space for "seen" pile
Initialise max to 0
More cards in "unseen" pile?
No (False)
End
Yes (True)
Pick one card X from the "unseen" pile and and move it into the "seen" pile
No (False)
X.F > max ?
Yes (True)
Set max to X.F

Figure 6.1: Flowchart for finding max value of field F

Note that the filtering condition is not static anymore, since the card's field is compared with a *variable* (`max` in this case), and not with a constant. The way to view this is: the filtering condition in the beginning admits all cards (i.e. filters out nothing). As the iteration progresses, it filters out cards with field values lower than `max`, and picks only those that contribute to increasing the `max`.

We can use the generic flowchart above to find the maximum value of different (numerical) fields by substituting that field in place of F in the generic flowchart above - for instance to find the maximum Maths marks, we use `X.mathsMarks` in place of `X.F` in the flowchart. Similarly, to find the bill with the highest spend, we use `X.billAmount` and to find the longest word, we can use `X.letterCount`.

6.1.1 Collect all the max elements

The flowchart in Figure 6.1 produces the maximum value of field F, but it does not tell us which cards contribute to this max value. For example, it will give us the maximum total marks, but will not say which students scored these marks.

To remedy this, we can keep a second variable called maxCardId, which keeps track of the unique sequence number of the card that holds the maximum field value out of cards seen so far. What should maxCardId be initialised to? Any value that will definitely *not* occur in the dataset should be fine. The simplest such integer is -1, which can never be any card's sequence number, which can only have 0 or positive values. The required flowchart is shown below in Figure 6.2.

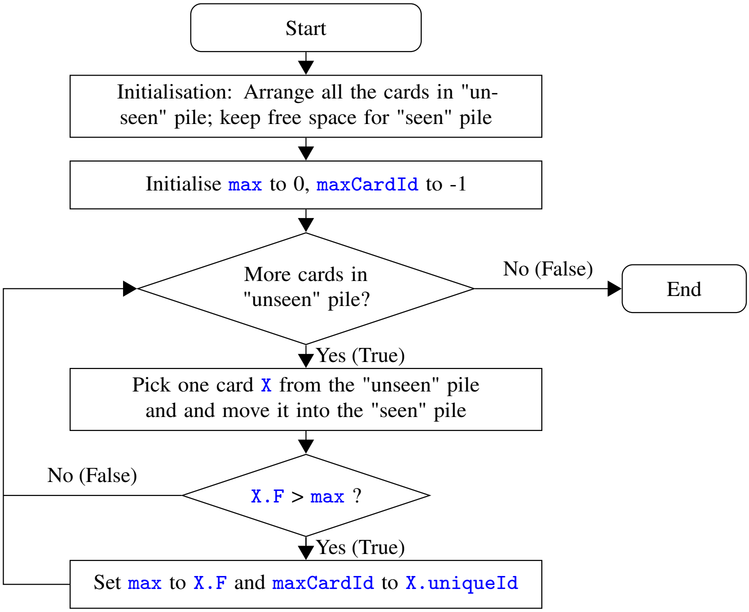

Start
Initialisation: Arrange all the cards in "unseen" pile; keep free space for "seen" pile
Initialise max to 0, maxCardId to -1
More cards in "unseen" pile?
No (False)
End
Yes (True)
Pick one card X from the "unseen" pile and and move it into the "seen" pile
No (False)
X.F > max ?
Yes (True)
Set max to X.F and maxCardId to X.uniqueId

Figure 6.2: Finding the card that has the maximum value of a field

Is this flowchart correct? Suppose that the first 4 cards have fields shown below:

UniqueId	9	2	7	11
ChemistryMarks	65	82	77	82

Then max takes value 0 after initialisation and 65, 82, 82, 82 after the first, second, third and fourth cards are read and processed. The first card sets max to 65. At the second card, X.chemistryMarks > max, so max gets set to 82. In the third and fourth cards, the Chemistry marks is not greater than 82, and so they are skipped.

What will the value of maxCardId be? It will be -1 after initialisation. After the first card, it is set to 9. The second card sets it to 2, and since the third and fourth card are skipped (filtered out), they don't change the value of maxCardId. So, the procedure is correct in the limited sense that it holds *one* student whose Chemistry marks is the highest. But this is not so satisfactory since the fourth card has exactly the same marks as the second card, and it should receive equal treatment. This would mean that we have to keep both cards, and not just one. We would need a list variable to hold multiple cards - let us call this list maxCardIds.

The modified flowchart is shown in Figure 6.3.

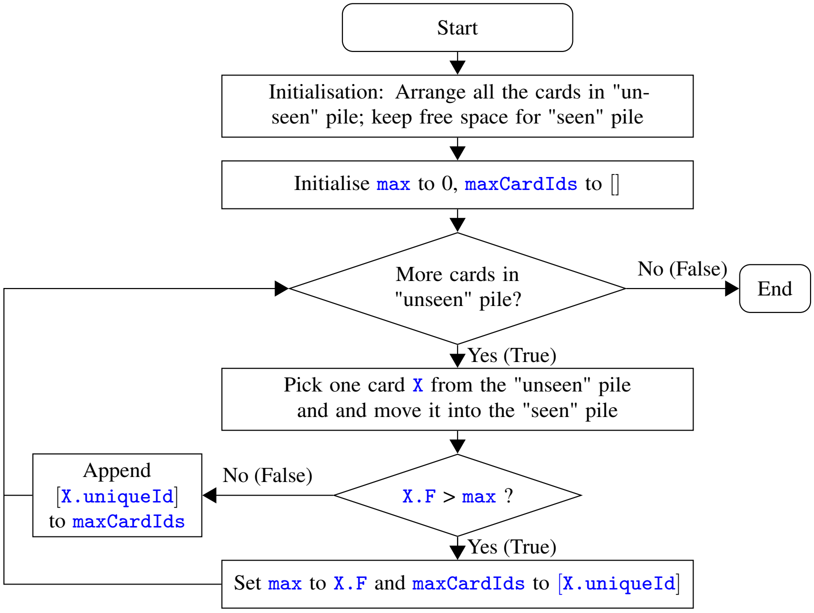

Start
Initialisation: Arrange all the cards in "unseen" pile; keep free space for "seen" pile
Initialise max to 0, maxCardIds to []
More cards in "unseen" pile?
No (False)
End
Yes (True)
Pick one card X from the "unseen" pile and and move it into the "seen" pile
X.F > max ?
No (False)
Append [X.uniqueId] to maxCardIds
Yes (True)
Set max to X.F and maxCardIds to [X.uniqueId]

Figure 6.3: Flowchart for finding all the maximum cards

Note that the list variable maxCardIds needs to be initialised to the empty list [] at the start. If we find a card N with a bigger field value than the ones we have seen so far, the old list needs to be discarded and replaced by $[N]$. But as we have seen above, if we see another card M with field value equal to max (82 in the example above), then we would need to append $[M]$ to the list maxCardIds.

We can apply this flowchart to find all the words with the greatest number of letters in the Words dataset. For this, replace X.F with X.letterCount.

6.2 Minimum

In the last section, we saw how to find the maximum value of some field, and the cards that hold that maximum value. Suppose now that we want to find the minimum value instead of the maximum value. Does the same method work?

In principle it should. Instead of keeping a variable called `max`, we can keep a variable called `min` which holds the minimum value of the required field. In the update step, instead of checking whether the field F has a value larger than max, we now check if the field value is lower than min. So far it looks pretty straightforward.

The key question is: what do we initialise `min` with? In the case of `max`, we could initialise with 0, since in our dataset there are no negative values in the numeric fields. Since we know that `min` will keep decreasing as we go through the cards, we could in principle start with the maximum value of the field across all the cards - but this requires us to find the maximum first ! Rather than do that, we just set `min` to some high enough value `MAX` that is guaranteed to be higher than the maximum value of the field. Usually, the field's subtype would have some constraint on the range of values allowed - so we can simply take the upper end of the range as the value of `MAX`.

In the classroom dataset, we saw that the subject marks had a maximum value of 100, so `MAX` would be 100 if are trying to find the minimum marks in a subject, while the total marks had a maximum value of 300, so `MAX` can be set to 300 if are trying to find the minimum total marks.

The flowchart in Figure 6.4 produces the minimum value of field F and the list of cards carrying this minimum value.

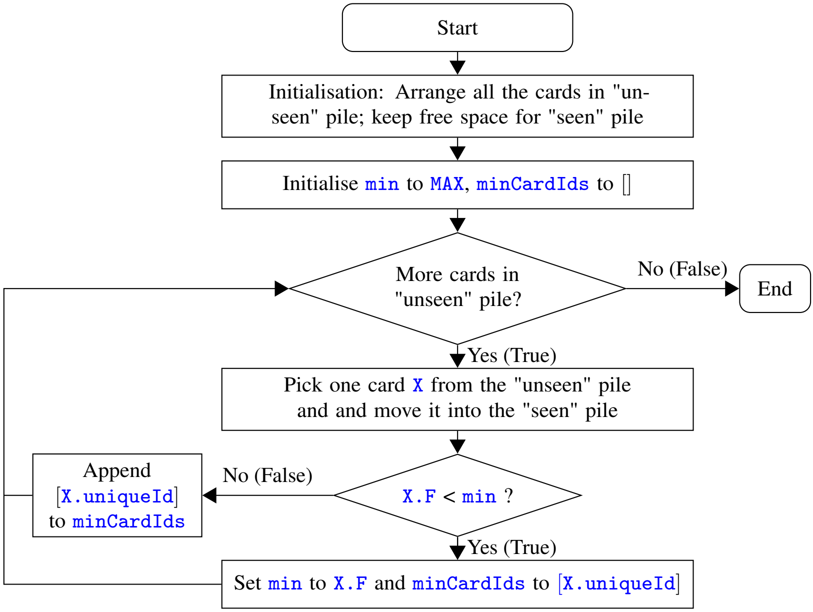

Start
Initialisation: Arrange all the cards in "unseen" pile; keep free space for "seen" pile
Initialise min to MAX, minCardIds to []
More cards in "unseen" pile?
No (False)
End
Yes (True)
Pick one card X from the "unseen" pile and and move it into the "seen" pile
Append [X.uniqueId] to minCardIds
No (False)
X.F < min ?
Yes (True)
Set min to X.F and minCardIds to [X.uniqueId]

Figure 6.4: Flowchart for finding all the minimum cards

6.3 Examples using maximum or minimum

We now look at examples of situations where the maximum or minimum is combined with other conditions to make the filtering really dynamic.

6.3.1 Find the length of the longest sentence

Let us now consider a slightly more difficult problem - that of finding the length of the longest sentence (i.e. the sentence with the largest number of words). For this, we have to count the words in each sentence, and as we do this we have to keep track of the maximum value of this count. The flowchart in Figure 4.7 gives us the word count of all the sentences. Can we modify it to give us just the maximum count rather than the list of all the sentence word counts? The required flowchart is shown in Figure 6.5.

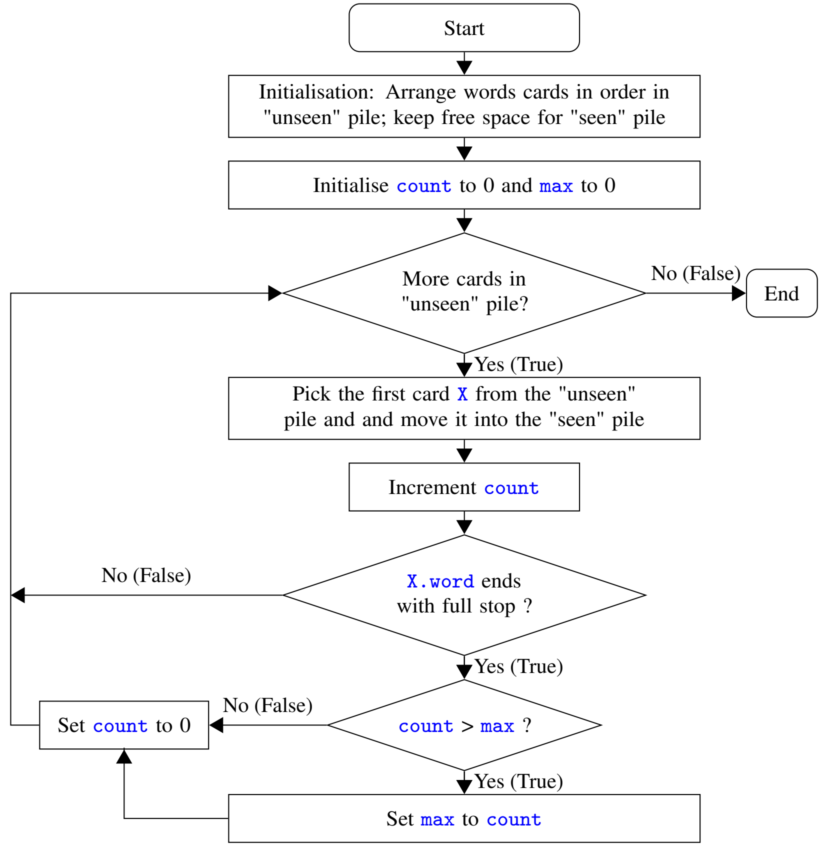

Start
Initialisation: Arrange words cards in order in "unseen" pile; keep free space for "seen" pile
Initialise count to 0 and max to 0
More cards in "unseen" pile?
No (False)
End
Yes (True)
Pick the first card X from the "unseen" pile and and move it into the "seen" pile
Increment count
No (False)
X.word ends with full stop ?
Yes (True)
Set count to 0
No (False)
count > max ?
Yes (True)
Set max to count

Figure 6.5: Flowchart for finding length of largest sentence

6.3.2 Harder Example: Find the boundary rectangles

6.4 Combining static, dynamic and state conditions

We now look at finding something (for example the maximum or minimum) only within a range of cards and not for the whole set of cards.

The obvious one is to pick cards which satisfy some (static) filtering condition - for example we could find the largest verb in the paragraph by iterating over the cards, filtering to check if the word's part of speech is Verb, and within the verbs, we find the word with the highest letter count. All we will need to do is to replace the filtering condition `X.F` > `max` in Figure 6.1 with the condition `X.partOfSpeech` is Verb AND `X.letterCount` > `max`. This is thus an example of filtering using a combination of a static condition (`X.partOfSpeech` is Verb) along with a dynamic condition (`X.letterCount` > `max`). As another example consider the problem of finding the bill with the highest spend for a given customer named N. We could check for the customer name (static condition `X.customerName` is N) along with the dynamic condition `X.billAmount` > `max`.

Sometimes, we have to go beyond using a static condition to select the range of cards to process. Suppose that as we go over the cards, the iteration passes through distinct phases (each of which we can call a **state**). The state of the iteration can come about because of certain cards that we have seen in the *past*. Without state, we will have to use filtering only to check the values of the current card (independent of what we have seen in cards in the past). In the examples that follow, we will look at a few examples of the use of state - for instance to keep track of the sentence number we are currently in, or to keep track of the last bill seen of a certain customer.

A good example of the use of state is through the Boolean variable `found` for exiting during the middle of an iteration that we say in Figure 4.18. The iteration goes through two states - the state before the item was found (when `found` is False) and the state after it is found (when `found` is True). However, in that example, we did not do anything in the latter state, except exit the iterator. So the use of state is not of much use in that example.

We could argue that keeping track of `max` is also like maintaining a state (the value of these variables). But this is not very useful as a way to understand what is going on - there are too many states (values of `max`), and the transition from one state to another is not very logical (`max` can change to anything depending on the card we are seeing). Compare this with the case of the sentence number - there are only a few states (number of sentences in the paragraph), and the transition is quite meaningful (sentence number increments by one during each transition).

So maintaining the state of iteration in a variable (or variables) is a way of simplifying complex dynamic conditions (wherever possible) by separating the part that depends on history (the state) from those that depend only on the current card.

6.4.1 Largest noun in the third sentence

Let us see an example where we do a combination of filtering conditions - static, dynamic and state. Let us say we want to find the longest noun in the third sentence. This requires us to select all the nouns - an example of static filtering (since we will be comparing the field partOfSpeech with the constant "Noun"). But it also requires us to keep track of which sentence we are currently in (the state), for which we will need to compare a variable snum for sentence number, which is incremented each time we see a full stop. It also requires us to find the maximum length word - which is a dynamic condition.

Since we are not interested in looking beyond the third sentence, we should exit the iteration loop once we have finished with the third sentence. The exit condition needs to take care of this (as we did in Figure 4.18 for exit after finding something).

The flowchart for doing all this is shown in Figure 6.6.

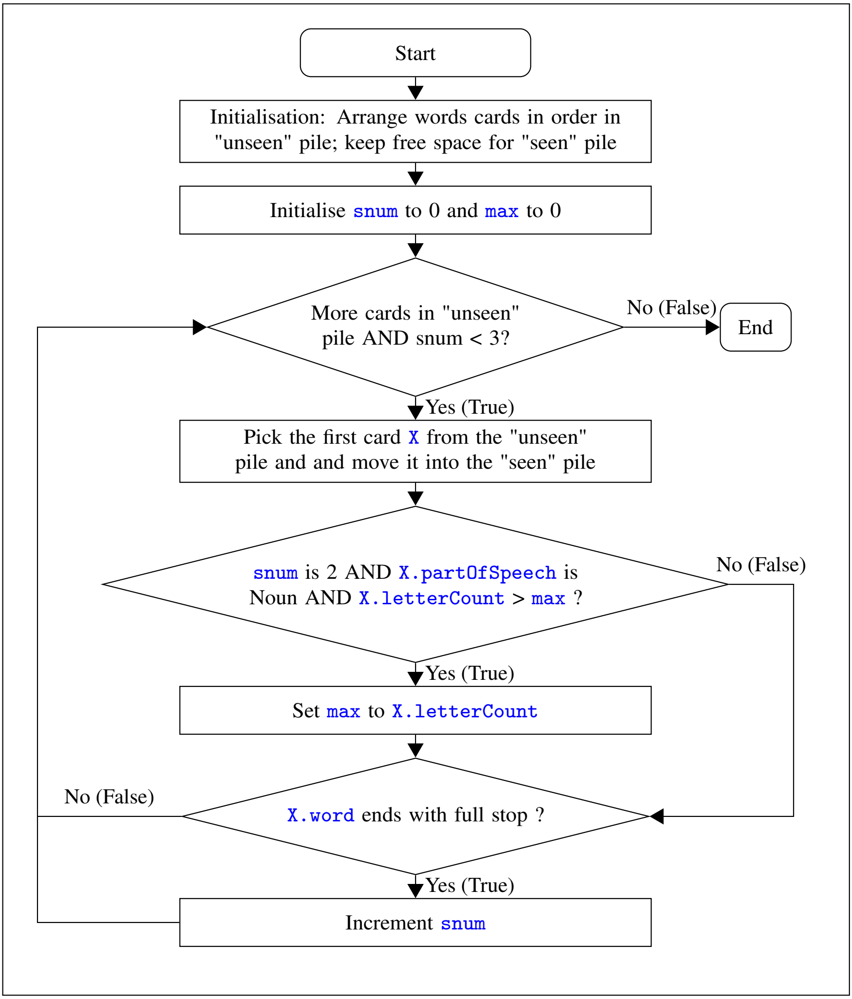

Start
Initialisation: Arrange words cards in order in "unseen" pile; keep free space for "seen" pile
Initialise snum to 0 and max to 0
More cards in "unseen" pile AND snum < 3?
No (False)
End
Yes (True)
Pick the first card X from the "unseen" pile and and move it into the "seen" pile
snum is 2 AND X.partOfSpeech is Noun AND X.letterCount > max ?
No (False)
Yes (True)
Set max to X.letterCount
X.word ends with full stop ?
No (False)
Yes (True)
Increment snum

Figure 6.6: Length of the largest noun in the third sentence

Observe the flowchart closely. There are a number of tricks that have been used. Firstly, to exit the flowchart after the third sentence, we could have used a Boolean variable found as we did in the generic find flowchart in Figure 4.18. However, this is not really necessary, as there is a good equivalent in the form of checking a condition of the state (snum < 3). Note that the state represents if we are in the first sentence (snum=0), second (snum=1) or third sentence (snum=2). Secondly, note that all the conditions are being checked in a single decision box - check for a state (snum=2 meaning that we are in the third sentence), the word is a noun (static condition), and that the word is longer than max (the dynamic condition). Only if all these are satisfied will max be updated. Thirdly, both branches of this decision box converge to the check for end of sentence, which if true causes snum to be incremented.

6.4.2 Checking if a customer is loyal

In the case of the words dataset, the words were assumed to be in an order according to their occurrence in the paragraph. So using the count of the number of sentences as a state variable made some sense. What kind of state information can we carry when the cards are not in any order?

Let us now consider an example of cards that may not be any order - the shopping bill dataset. Say we want to find out if a certain customer named N is loyal - defined as follows: N is loyal if all the bills that have N as the customer name all share the same shop name. This would mean that customer N shops only in one shop (i.e. is loyal to that shop). How do we check if the given customer N is loyal? As we iterate through the cards, we keep looking for the customer name N. If we find such a card, we change the state (variable shop initially empty) to the name of the shop on the card (say S). As we go through the remaining cards, we keep looking for a card with customer name N (static condition) and also for state not being empty (i.e. we have seen a previous card of the same customer), and check if the card's shop name will change the state again. If it will, then the customer is not loyal and we can exit the iteration.

The flowchart is shown below in 6.7. The Boolean variable loyal starts with the value True, and becomes false when a second shop is encountered for the same customer name.

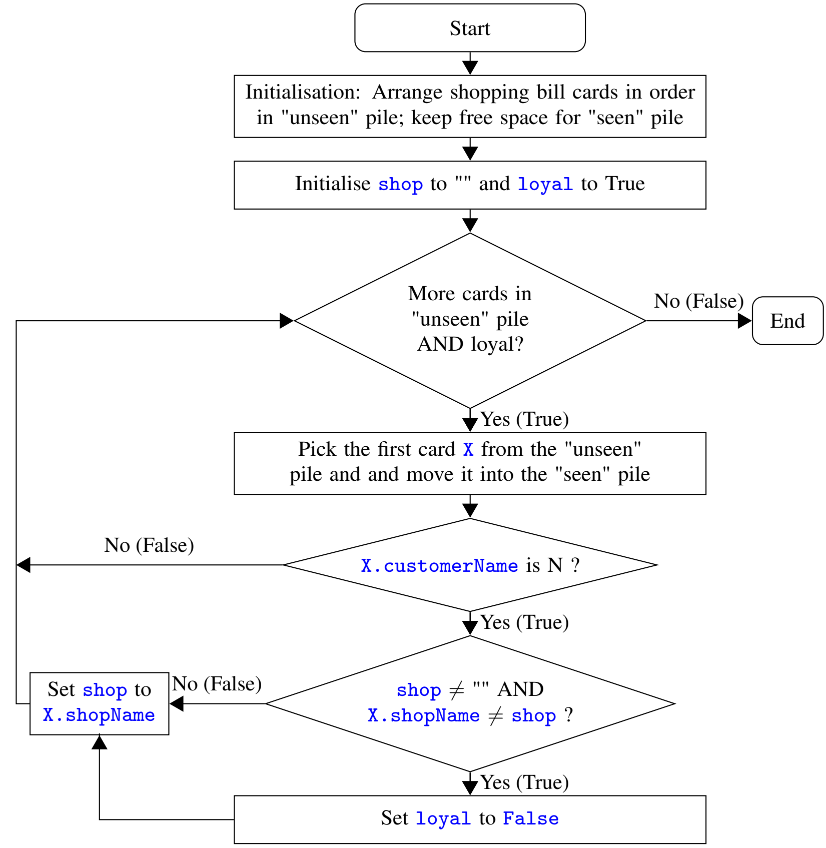

Start
Initialisation: Arrange shopping bill cards in order in "unseen" pile; keep free space for "seen" pile
Initialise shop to "" and loyal to True
More cards in "unseen" pile AND loyal?
No (False)
End
Yes (True)
Pick the first card X from the "unseen" pile and and move it into the "seen" pile
No (False)
X.customerName is N ?
Yes (True)
Set shop to X.shopName
No (False)
shop ≠ "" AND X.shopName ≠ shop ?
Yes (True)
Set loyal to False

Figure 6.7: Flowchart for checking if a customer is loyal

A few points to note about the flowchart. The exit condition is strengthened with a check for loyal. If we have found that the customer is not loyal, then the iteration exits (nothing further to do). Within the iterator, we first check for the static condition - whether the customer name is N. If not, we skip the card. If it is N, then we check if the state `shop` is previously set and if so if it is different from the current shop name - which cause `loyal` to be set to False. In any event, we set the state shop to the shop name on the card. This is merely for convenience (so we don't have to write another decision box). If shop was "" earlier, then it will take the shop name from the card. If the shop name remains the same, assigning it to `shop` will not change the state. If the shop name is different, then `shop` is set to the latest shop name (i.e. the state changes), but this is never used again because we will exit the loop.

6.4.3 Example:

Summary of chapter

Exercises

7. Pseudocode

Flowcharts provide an appealing pictorial representation of a computational process. Their layout makes it easy to visualize the overall structure of the computation.

However, there are also some disadvantages with using flowcharts. The first is size: complex processes generate large flowcharts. As the visual representation grows, it is no longer possible to fit the entire picture on a single page, and digesting the overall structure becomes harder. The second drawback is that pictures are not convenient for collaboration. Computational processes are typically developed in teams, and it is important to be able to share descriptions of these processes in a format that can be edited and updated easily. A natural consequence of collaboration is that the description will change over time, with updates made by more than one person. It is important to be able to compare changes between versions, which is difficult with flowcharts.

Let us revisit our simple flowchart for counting the number of cards in a pile.

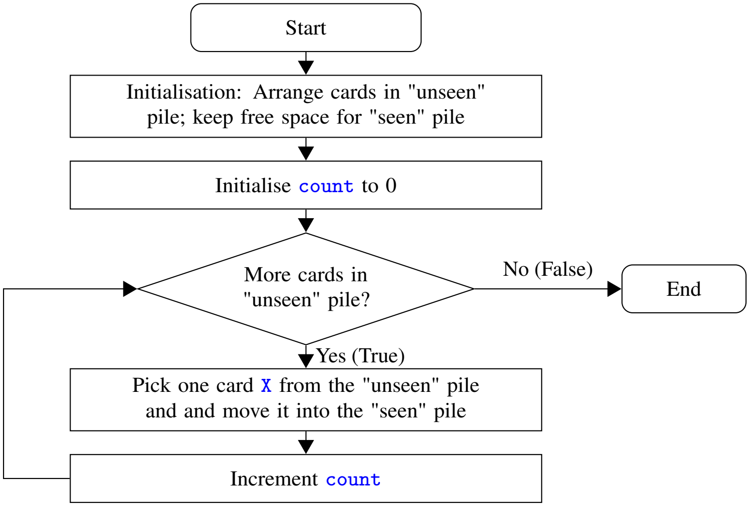

Start
Initialisation: Arrange cards in "unseen" pile; keep free space for "seen" pile
Initialise count to 0
More cards in "unseen" pile?
No (False)
End
Yes (True)
Pick one card X from the "unseen" pile and and move it into the "seen" pile
Increment count

Figure 7.1: Flowchart for counting cards

We can, instead, describe the process in words, as a sequence of steps.

Counting cards

Step 0 Start
Step 1 Initialize `count` to 0
Step 2 Check cards in Pile 1
Step 3 If no more cards, go Step 8
Step 4 Pick a card `X` from Pile 1
Step 5 Move `X` to Pile 2
Step 6 Increment `count`
Step 7 Go back to Step 2
Step 8 End

However, this text notation does not emphasize the logical structure of the computation. For instance, Steps 2, 3 and 7 together describe how we iterate over the cards till all of them are processed. Steps 4, 5 and 6 are the steps that we perform repeatedly, in each iteration.

Instead of using plain English to describe the steps, is is better to develop specialized notation that captures the basic building blocks of computational processes.

This specialized notation is called a *programming language*. We would expect it to have more direct ways of describing conditional and repeated execution.

We will gradually build up such a programming language to describe the processes that we encounter in our exploration of computational thinking. A computational process written using a programming language is referred to as *code* and is typically translated into low level basic instructions that can be executed by a computer. The language we develop will be less formal than a real programming language, so we refer to our notation as *pseudocode*. to emphasize that it is not “real” code.

7.1 Basic iteration

Let us begin by representing our process to count cards using pseudocode.

```
Start
count = 0
while (Pile 1 has more cards) {
   Pick a card X from Pile 1
   Move X to Pile 2
   Increment count
}
End
```

Pseudocode 7.1: Counting cards

In this example, we have a few basic elements of pseudocode.

- The statement “count = 0” assigns a value 0 to the variable count. Not surprisingly, this is called an *assignment statement*.
- The special word **while** signals an iteration where we repeatedly perform a block of actions till some condition becomes false.
  The condition to be checked is enclosed in brackets after the word **while**—in this case, the condition is “Pile 1 has more cards”. If this condition evaluates to true, the next round of iteration begins. If it is false, the iteration terminates.
  The scope of the iteration is demarcated by a pair of matching curly brackets, or braces, “{“ and “}”. All statements between this pair of matching braces will be executed in each iteration. Such a set of statements, enclosed within a pair of matching braces, is often referred to as a *block* of code.

With minor changes, we can modify this pseudocode to add up the values in a particular field F across all the cards. Instead of incrementing a variable count with each iteration, we update a variable sum by adding the value X.F to sum for each card X that we read. Here is the pseudocode for the flowchart from Figure 3.3.

```
Start
sum = 0
while (Pile 1 has more cards) {
    Pick a card X from Pile 1
    Move X to Pile 2
    Add X.F to sum
}
End
```

Pseudocode 7.2: Generic sum

We can merge and extend these computations to compute the average, as we had earlier seen in the flowchart of Figure 3.5.

```
Start
count = 0
sum = 0
while (Pile 1 has more cards) {
    Pick a card X from Pile 1
    Move X to Pile 2
    count = count + 1
    sum = sum + X.F
}
average = sum/count
End
```

Pseudocode 7.3: Average

This pseudocode represents a variant of the computation described in Figure 3.4. In that flowchart, average is updated with each card that we read, inside the body of the iteration. In the pseudocode above, we only update count and sum when we iterate over the cards. When we exit the iteration, we move to the first statement after the closing brace that defines the scope of the iteration and use a single assignment statement to store the final value of sum/count in average.

We have made one more change with respect to Pseudocode 7.1 and 7.2. The increment to count and the update to sum are also now described using assignment statements. The statement "count = count + 1" is not to be read as an equation—such an equation would never be true for any value of count! Rather, it represents an update to count. The expression on the right hand side reads the current value of count and adds 1 to it. This new value is then stored back in count. In the same way, the assignment "sum = sum + X.F" takes the current value of sum, adds X.F to it, and stores the resulting value back in sum.

7.2 Filtering

The next step is to add a conditional statement that will allow us to do filtering. Consider the flowchart for finding total marks for both boys and girls, shown in Figure 4.4. Here is the corresponding pseudocode.

```
Start
boysTotalMarks = 0
girlsTotalMarks = 0
while (Pile 1 has more cards) {
  Pick a card X from Pile 1
  Move X to Pile 2
  if (X.gender == M) {
    boysTotalMarks = boysTotalMarks + X.totalMarks
  }
  else {
    girlsTotalMarks = girlsTotalMarks + X.totalMarks
  }
}
End
```

Pseudocode 7.4: Total marks of both boys and girls

We have introduced a new word **if** to describe conditional execution to be performed once, unlike **while** which captures steps to be performed repeatedly till the given condition becomes false. As before, the condition to be checked appears in brackets after the word **if**. In this case, we are checking whether the field gender on the card, X.gender, is equal to M. Notice that we use an unusual symbol "==" to denote equality, to avoid confusion with the operator "=" that assigns a value to a variable. If the condition is true, the statements within the first set of braces are executed. The block labelled **else** is executed if the statement is false. There may be no **else** block. For instance, here is an example where we only sum the total marks of girls. If `X.gender` is not equal to `F`, we don't perform any action within the body of the iteration.

```
Start
girlsTotalMarks = 0
while (Pile 1 has more cards) {
  Pick a card X from Pile 1
  Move X to Pile 2
  if (X.gender == F) {
    girlsTotalMarks = girlsTotalMarks + X.totalMarks
  }
}
End
```

Pseudocode 7.5: Total marks of girls

7.3 Compound conditions

We can combine conditions using AND, as we have done in flowcharts. Here is the pseudocode for the flowchart in Figure 4.9 that computes the total marks of girls from Chennai.

```
Start
cgSum = 0
while (Pile 1 has more cards) {
  Pick a card X from Pile 1
  Move X to Pile 2
  if (X.gender == F AND X.townCity == Chennai)) {
    cgSum = cgSum + X.totalMarks
  }
}
End
```

Pseudocode 7.6: Total marks of girls from Chennai

7.4 Filtering with dynamic conditions

As a final example, let us look at filtering with dynamic conditions. Figure 6.2 describes a flowchart to find the card that has the maximum value of a field. Here is the corresponding pseudocode.

```
Start
max = 0
maxCardID = -1
while (Pile 1 has more cards) {
  Pick a card X from Pile 1
  Move X to Pile 2
  if (X.F > max) {
    max = X.F
    maxCardID = X.uniqueId
  }
}
End
```

Pseudocode 7.7: Card with maximum value of a field

Summary of chapter

Exercises

8. Procedures and Parameters

We have seen that some patterns are repeated across different computations. Understanding the structure of these patterns helps us build up abstractions such as filtered iterations, which can be applied in many contexts.

At a more concrete level, variants of the same computation may be required in for the same context. For example, the computation described in Figure 6.1 allow us to find the maximum value of any field F in a pile of cards. We can instantiate F to get a concrete computation—for instance, for the Scores dataset, if we choose F to be chemistryMarks, we would get the maximum marks in Chemistry across all the students.

We can think of such a generic pattern as a unit of computation that can be parcelled out to an external agent. Imagine a scenario where we have undertaken a major project, such as building a bridge. For this, we need to execute some subtasks, for which we may not have the manpower or the expertise. We outsource such tasks to a subcontractor, by clearly specifying what is to be performed, which includes a description of what information and material we will provide the subcontractor and what we expect in return.

In the same we, we can package a unit of computation as a “contract” to be performed on our behalf and invoke this contract whenever we need it. This packaged code that we contract out to be performed independently is called a *procedure*. We pass the information required to execute the computation as *parameters*—for instance, to tell a procedure that computes a generic maximum which field F we are interested in. The procedure then returns the value that corresponds to our expectation of the task to be performed, in this case, the maximum value of field F across all the cards.

8.1 Pseudocode for procedures

Let us write the flowchart in Figure 6.1 to illustrate our pseudocode for procedures.

```
Procedure findMax(field)
  max = 0
  while (Pile 1 has more cards) {
    Pick a card X from Pile 1
    Move X to Pile 2
    if (X.field > max) {
      max = X.field
    }
  }
  return(max)
End findMax
```

Pseudocode 8.1: Procedure for generic maximum

We use the word **Procedure** to announce the start of the procedure and signal the end of the code with the word **End**. The words **Procedure** and **End** are tagged by the name of the procedure—in this case, `findMax`. We indicate the parameters to be passed in brackets, after the name of the procedure in the first line. At the end of its execution, the procedure has to send back its answer. This is achieved through the **return** statement which, in this case, sends back the value of the variable `max`.

Here is a typical statement invoking this procedure.

```
maxChemistryMarks = findMax(chemistryMarks)
```

The right hand side calls the procedure with argument `chemistryMarks`. This starts off the procedure with the parameter `field` set to `chemistryMarks`. At the end of the procedure, the value `max` is returned as the outcome of the procedure call, and is assigned in the calling statement to the variable `maxChemistryMarks`. Thus, the call to the procedure is typically part of an expression that is used to compute a value to be stored in a variable.

8.2 Parameters

In the previous example, the parameter passed was a field name. We can also pass a value as a parameter. Suppose we want compute the maximum value in a particular field for students of a particular gender. This would require two parameters, one specifying the field to examine and the other describing the gender, as shown below.

```
Procedure findMax(field,genderValue)
  max = 0
  while (Pile 1 has more cards) {
    Pick a card X from Pile 1
    Move X to Pile 2
    if (X.field > max AND X.gender == genderValue) {
      max = X.field
    }
  }
  return(max)
End findMax
```

Pseudocode 8.2: Procedure for maximum of a specific gender

To compute the maximum Physics marks among the boys in the class, we would write a statement like the following.

maxPhysicsMarksBoys = findMax(physicsMarks,M)

The arguments are substituted by position for the parameters in the procedure, so field is assigned the name physicsMarks and genderValue takes the value M. As before, the value returned by the procedure is stored in the variable maxPhysicsMarksBoys.

8.3 Checking for a single outstanding student

To illustrate the use of procedures, let us examine the problem of determining whether there is a single student who is the best performer across all the subjects. If the same student has achieved the top mark in all subjects, then this student's total should be the same as the sum of the highest marks in all the subjects. We can call the procedure in Pseudocode 8.1 multiple times to determine this.

```
Start
maxMaths = findMax(mathsMarks)
maxPhysics = findMax(physicsMarks)
maxChemistry = findMax(chemistryMarks)
maxTotal = findMax(totalMarks)
subjTotal = maxMaths + maxPhysics + maxChemistry
```

```
if (maxTotal == subjTotal) {
  singleTopper = True
}
else {
  singleTopper = False
}
End
```

Pseudocode 8.3: Check if there is a single outstanding student

8.4 Side-effects

What happens when a variable we pass as an argument is modified within a procedure? Consider the following procedure that computes the sum $1+2+\cdots+n$ for any upper bound n. The procedure decrements the parameter value from n down to 1 and accumulates the sum in the variable sum.

```
Procedure prefixSum(upperLimit)
  sum = 0
  while (upperLimit > 0) {
    sum = sum + upperLimit
    upperLimit = upperLimit - 1
  }
  return(sum)
End prefixSum
```

Pseudocode 8.4: Prefix sum

Suppose we call this procedure as follows.

```
Start
n = 10
sumToN = prefixSum(n)
End
```

Pseudocode 8.5: Calling prefix sum with a variable as argument

What is the value of `n` after `prefixSum` executes? Is it still 10, the value we assigned to the variable before calling the procedure? Or has it become 0, because `prefixSum` decrements its input parameter to compute the return value?

Clearly, we would not like the value of `n` to be affected by what happens within the procedure. If we use the metaphor that the procedure is executed by a subcontractor, the data that we hold should not be affected by what happens within a computation whose details are not our concern. Imagine if the procedures we wrote to compute the average of some field in a pile of cards overwrote the values on the cards!

When a procedure modifies a value and this has an impact outside the computation of the procedure, we say that the procedure has a *side effect*. In general, side effects are not desirable. Getting back to the idea of drawing up a contract with the procedure, one of the clauses we impose is that none of the values passed as parameters are modified globally by the procedure. Implicitly, each argument is copied into a local variable when the procedure starts, so all updates made within the procedure are only to the local copy.

However, in the examples we have seen, some data is shared with the procedure without explicitly passing a parameter. For instance, we never pass the pile of cards to work on as a parameter; the procedure implicitly has direct access to the pile of cards. When the procedure moves cards from one pile to another, this will typically have the side effect of rearranging the pile.

In some situations, the outcome we want from a procedure is the side effect, not a return value. For instance, we will see later how to rearrange a pile of cards in ascending order with respect to some field value on the card. A procedure to execute this task of sorting cards would necessarily have to modify the order of the pile of cards.

We will return to the topic of side effects and examine it in more detail in Chapter 12.

Summary of chapter

Exercises

Part II

Computational Thinking for Data Science

9. Element ⟷ Dataset

There are many ways to make sense of the data that is given to us. In descriptive statistics, we try to create some summary information about the dataset using aggregate values such as the mean (same as average) of the data. This is usually accompanied by information about how dispersed the data is around the mean, and whether the data is symmetric around the mean or is skewed to one side.

As we have seen in Part I, computational methods allow us to collect such aggregate values (and perhaps many more) from the entire dataset, by making one or more passes through it using iterators and (accumulator) variables to store the results.

In this section, we will see how we can make further sense of the dataset by identifying where each element stands with respect to these aggregate values collected from the entire dataset.

As an example, consider the procedure to find the maximum Chemistry marks and the element(s) that contribute to this value. The procedure not only finds the maximum Chemistry marks but also creates a relationship between the maximum elements and the rest of the dataset - all the elements will have Chemistry marks below these maximum elements. Likewise, when we find the minimum, all the elements will be positioned above the elements with minimum marks.

As another example, say we have collected the average total marks of all the students, then we can separate the students into two groups - those who are above average and those who are below average. We could also form three groups - those around the average marks (say within average plus or minus 5 marks), those significantly above average (with total marks that is at least 5 above average), and those significantly below average (with total marks that is at least 5 below average).

If we know the maximum, minimum and average, we could make many more groups - lets say groups A, B, C, D and E (which are grades that can be awarded to the students). Those in the average ±5 can be given C grade. We can divide the range between the maximum and average + 5 into two equal parts, with the higher half getting A and the lower half getting B. Similarly, the range between minimum and average $-$ 5 can be divided into two equal halves, with the higher being awarded D and the lower E.

We can also use filtering to find averages for subsets of the data. For example, say we find the average of the boys and the average of the girls total marks. Then by comparing these averages, we can say if the girls are doing better than the boys in this class. This may not be entirely satisfactory because there could be one boy with high marks who skews the boys' average to the higher side. A better method may be to find the number of boys who are above the overall average and compare it with the number of girls who are above average. But this is not really accurate, since the class may have many more boys than girls - so what we have to do is to find the percentage of girls who are above average and compare it with the percentage of boys who are above average.

Similarly, in the shopping bill dataset, we can find the average bill value and then proceed to categorise a customer as low spenders, average spenders or high spenders depending on whether the average bill value for *that customer* (use filtering) is much below average, around the average or much above average respectively.

In the words dataset, we could find the average frequency and average letter count of the words. Then we could determine which words are occurring frequently (i.e. above the average frequency) or are long (i.e. above the average lettercount).

9.1 Sequenced iterations

As we saw in the examples above, to find the relation between a data element and the entire dataset, we will need at least two iterations - executed in sequence. The first iteration is used to find the aggregate values such as the average, and the second (or later) iterations are used to position an element (or subset) in relation to the entire dataset.

9.1.1 Flowchart for sequenced iterations

The generic flowchart for doing this is shown in Figure 9.1. The variables used in the first iteration are called *aggregate variables* to distinguish them from the variables we will use in the second iteration.

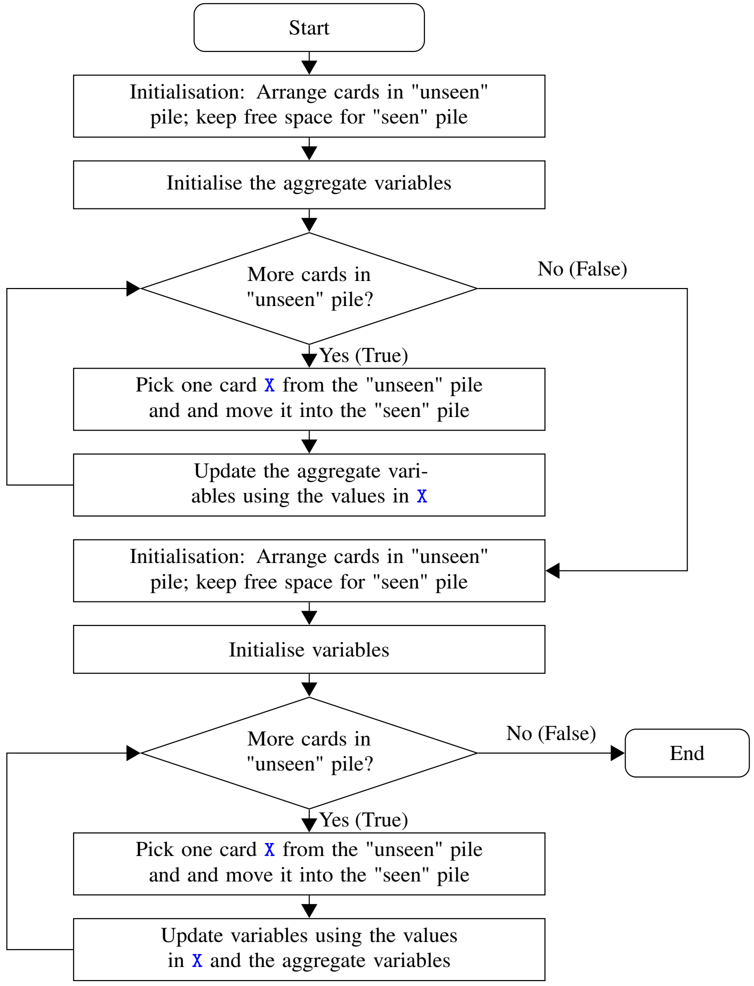

Start
Initialisation: Arrange cards in "unseen" pile; keep free space for "seen" pile
Initialise the aggregate variables
More cards in "unseen" pile?
No (False)
Yes (True)
Pick one card X from the "unseen" pile and and move it into the "seen" pile
Update the aggregate variables using the values in X
Initialisation: Arrange cards in "unseen" pile; keep free space for "seen" pile
Initialise variables
More cards in "unseen" pile?
No (False)
End
Yes (True)
Pick one card X from the "unseen" pile and and move it into the "seen" pile
Update variables using the values in X and the aggregate variables

Figure 9.1: Generic flowchart for two iterations in sequence

If you observe the flowchart carefully and compare it with the flowcharts that we saw in Part I, you will notice that there are two iteration stubs embedded in the flowchart (identified through the initialisation, decision to end the iteration, and update steps). Whereas in most of the iterators seen earlier the iterator exited and went straight to the end terminal, in this flowchart, the first iteration exits and goes to the initialisation of the second iteration. It is the second iteration that exits by going to the end terminal. That is precisely why we say that the two iterations are in sequence. The second iteration will start only after the first iteration is complete.

9.1.2 Pseudocode for sequenced iterations

The pseudocode corresponding to this flowchart is shown below (where Pile 1 is the "Unseen" Pile, Pile 2 is the "Seen" pile).

```
Start
Arrange all cards in Pile 1
Initialise the aggregate variables
while (Pile 1 has more cards) {
  Pick a card X from Pile 1
  Move X to Pile 2
  Update aggregate variables using the field values of X
}
Arrange all cards in Pile 1
Initialise the variables
while (Pile 1 has more cards) {
  Pick a card X from Pile 1
  Move X to Pile 2
  Update variables using the aggregate variables and the field values of X
}
End
```

Pseudocode 9.1: Sequenced iterations

9.2 Compare with average

To start with, we first look at comparing each element with the average to see which side of the average it lies. By using appropriate filtering conditions, we could determine the average only for a subset of the data elements, and compare these with the average of the whole, which may reveal something about that subset.

9.2.1 Students performing well

Let us say we want to find out which students are doing well. How do we define "doing well"? We could start with a simple definition - say doing well is the same as above average. The pseudocode pattern for sequential iteration can be modified to do this as shown below:

```
Start
Arrange all cards in Pile 1
sum = 0, count = 0
while (Pile 1 has more cards) {
  Pick a card X from Pile 1
  Move X to Pile 2
  sum = sum + X.totalMarks
  count = count + 1
}
average = sum/count
Arrange all cards in Pile 1
aboveList = []
while (Pile 1 has more cards) {
  Pick a card X from Pile 1
  Move X to Pile 2
  if (X.totalMarks > average) {
    Append [X] to aboveList
  }
}
End
```

Pseudocode 9.2: Above average students

9.2.2 Are the girls doing better than the boys?

As we discussed at the beginning of the chapter, to find out if the girls are doing better than the boys, we could find the percentage of girls who are above the overall average and compare it with the percentage of boys who are above average. This is better than just checking if the girls average is higher than the boys average, since the girls average could be skewed higher due to a single high scoring girl student (or alternatively, the boys average may be skewed lower due to a single low scoring boy student).

The pseudocode for finding the above average students can be modified using filters to find the count of the above average boys and above average girls as shown below:

```
Start
Arrange all cards in Pile 1
sum = 0, count = 0, numBoys = 0, numGirls = 0
while (Pile 1 has more cards) {
  Pick a card X from Pile 1
  Move X to Pile 2
  sum = sum + X.totalMarks
  count = count + 1
  if ( X.gender == Girl ) {
    numGirls = numGirls + 1
  } else {
    numBoys = numBoys + 1
  }
}
average = sum/count
Arrange all cards in Pile 1
boysAbove = 0, girlsAbove = 0
while (Pile 1 has more cards) {
  Pick a card X from Pile 1
  Move X to Pile 2
  if (X.totalMarks > average) {
    if ( X.gender == Girl ) {
      girlsAbove = girlsAbove + 1
    } else {
      boysAbove = boysAbove + 1
    }
  }
}
girlsAreDoingBetter = False
```

```
if ( girlsAbove/numGirls > boysAbove/numBoys ) {
  girlsAreDoingBetter = True
}
End
```

Pseudocode 9.3: Are the girls doing better than the boys?

9.2.3 Harder Example: Find the rectangles near the "centre" of the figure

9.3 Classifying/Ranking the elements

In this section, we look at using sequenced iterations in situations where we need to classify (or grade) the elements in the dataset. As before, we first find the aggregate values (such as average, max and min) from the entire dataset in the first pass through the dataset. In the second pass, we filter the elements using boundary conditions derived from the aggregate values and place them in different buckets.

9.3.1 Assigning grades to students

Suppose we want to assign grades to the students based on the total marks scored in the classroom dataset. Grades can be A (for the excellent students), B (for slightly above average students), C (for slightly below average students) and D (for students performing poorly).

A logical division (that is often used by teachers) is to divide the range between the minimum and maximum marks into four equal intervals, each of which is assigned a different grade. Say min and max are the minimum and maximum total marks respectively. Let delta = (max − Min)/4. Then we assign D grade to students lying between min and min + delta, C to students lying between min + delta and min + 2 × delta, B to min + 2 × delta and min + 3 × delta and finally A to min + 3 × delta and max (= min + 4 × delta). The pseudocode to do this is shown below:

```
Start
Arrange all cards in Pile 1
max = 0, min = 300
```

```
while (Pile 1 has more cards) {
    Pick a card X from Pile 1
    Move X to Pile 2
    if ( X.totalMarks > max ) {
        max = X.totalMarks
    }
    if ( X.totalMarks < min ) {
        min = X.totalMarks
    }
}
delta = (max - min)/4
Arrange all cards in Pile 1
listA = [], listB = [], listC = [], listD = []
while (Pile 1 has more cards) {
    Pick a card X from Pile 1
    Move X to Pile 2
    if (X.totalMarks < min + delta) {
        Append [X] to listD
    }
    else if (X.totalMarks < min + 2 × delta) {
        Append [X] to listC
    }
    else if (X.totalMarks < min + 3 × delta) {
        Append [X] to listB
    }
    else {
        Append [X] to listA
    }
}
End
```

Pseudocode 9.4: Assigning grades to students

9.3.2 Awarding 3 prizes to the top students

Suppose we want to award prizes to the three best students in the class. The most obvious criterion is to choose the students with the three highest total marks. However, we may have a situation where students excel in one subject but not in others, so a person who is merely above average in all three subjects ends up having a higher total than students who excel in individual subjects. To take this into account, we add another constraint: to get a prize, a student must not only be within the top three total marks, but must also be in the top three in at least one subject.

To solve this problem, we have to first find the top three marks in each individual subject. Once we have this data, we can iterate over all the students and keep track of the top three total marks, checking each time that the student is also within the top three in one of the subjects.

How do we find the top three values in a set of cards? We have seen how to compute the maximum value: initialize a variable max to 0, scan all the cards, and update max each time we see a large value.

To find the two highest values, we maintain two variables, max and secondmax. When we scan a card, we have three possibilities. If the new value is smaller than secondmax, we discard it. If it is bigger than secondmax but smaller than max, we update secondmax to the new value. Finally, if it is larger than max, we save the current value of max as secondmax and then update max to the new value.

We can extend this to keep track of the three largest values. We maintain three variables, max, secondmax and thirdmax. For each card we scan, there are four possibilities: the value on the card is below thirdmax, or it is between secondmax and thirdmax, or it is between secondmax and max, or it is greater than max. Depending on which case holds, we update the values max, secondmax and thirdmax in an apropriate manner.

Here is a procedure to compute the top three values of a field in a pile of cards.

```
Procedure topThreeMarks(field)
    max = 0
    secondmax = 0
    thirdmax = 0
    while (Pile 1 has more cards) {
      Pick a card X from Pile 1
      Move X to Pile 2
      if (X.field > max) {
        thirdmax = secondmax
        secondmax = max
        max = X.field
      }
      if (max > X.field AND X.field > secondmax) {
        thirdmax = secondmax
```

```
            secondmax = X.field
          }
          if (secondmax > X.field AND X.field > thirdmax) {
            thirdmax = X.field
          }
          return(thirdmax)
end topThreeMarks
```

Pseudocode 9.5: Procedure for top three values

We start by initializing max, secondmax and thirdmax to 0.

If the value on the card is bigger than max, we shift the values of secondmax and max to thirdmax and secondmax, respectively, and then update max to the value on the current card, Note the sequence in which we shift the values—if we do this in the wrong order, we will overwrite a variable before we can copy its value.

If the value on the card is between max and secondmax, we don't touch the value of max. We just copy secondmax to thirdmax and update secondmax to the value on the current card.

Finally, if the value on the card is between secondmax and thirdmax, we don't need to touch the values of max and secondmax. We directltly update thirdmax to the value on the current card.

Our procedure could return max, secondmax and thirdmax as a list of three values. However, we observe that we only need to know the third highest mark to know whether a student is within the top three marks in a subject, so we just return the value of thirdmax.

We can now use this procedure to compute the top three students overall. We begin with another procedure that checks whether the student in a particular card is within the top three in at least one subject. The procedure takes as parameters the card to be checked along with the third highest marks in each subject and returns True if the card has a mark above the third highest value in at least one subject.

```
Procedure subjectTopper(card, mathsCutoff,
                        physicsCutoff, chemistryCutoff)
     if (card.mathsMarks >= mathsCutoff OR
       card.physicsMarks >= physicsCutoff OR
       card.chemistryMarks >= chemistryCutoff) {
            return(True)
         }
         else {
            return(False)
         }
   end subjectTopper
```

Pseudocode 9.6: Check if student is a subject topper

We can now write the code for the main iteration to compute the three top students overall.

We have to maintain not only the three highest marks, but also the identities of the students with these marks, so we have variables max, secondmax and thirdmax to record the marks and variables maxid, secondmaxid and thirdmaxid to record the identities of the corresponding students.

We record the third highest mark in each subject by calling the procedure topThreeMarks three times, once per subject. We then iterate over all the cards. We first check if the card represents a subject topper; if not we don't need to process it. If the card is a subject topper, we compare its total marks with the three highest values and update them in the same way that we had done in the procedure topThreeMarks, except that we also need to update the corresponding identities.

```
Start
max = 0
secondmax = 0
thirdmax = 0

maxid = -1
secondmaxid = -1
thirdmaxid = -1

mathsThird = topThreeMarks(mathsMarks)
physicsThird = topThreeMarks(physicsMarks)
chemistryThird = topThreeMarks(chemistryMarks)

while (Pile 1 has more cards) {
   Pick a card X from Pile 1
   Move X to Pile 2
   if (subjectTopper( X, mathsThird,
       physicsThird, chemistryThird) {
      if (X.totalMarks > max) {
            thirdmax = secondmax
            thirdmaxid = secondmaxid
            secondmax = max
            secondmaxid = maxid
            max = X.totalMarks
            maxid = X.id
          }
          if (max > X.totalMarks AND X.totalMarks > secondmax) {
            thirdmax = secondmax
            thirdmaxid = secondmaxid
            secondmax = X.totalMarks
            secondmaxid = X.id
          }
          if (secondmax > X.totalMarks AND
             X.totalMarks > thirdmax) {
            thirdmax = X.field
            thirdmaxid = X.id
          }
        }
      }
    End
```

Pseudocode 9.7: Procedure for top three values

We could add more criteria for awarding the top three prizes. For instance, we could insist that there must be at least one boy and one girl in the top three—it should not be the case that all three are boys or all three are girls. To handle this, we could examine the top three students we have found using the process we have described and check their gender. If there are both boys and girls in the list, we are done. Otherwise, we discard the third highest card and repeat the computation for the reduced pile.

If we repeat this process enough times, are we guaranteed to find a suitable set of three students to award prizes? What if all the subject toppers are of the same gender? What if there are ties within the top three totals?

As you can see, framing the requirement for a computation can also be a complicated process. We need to anticipate extreme situations—sometimes called *corner cases*—and decide how to handle them. If we do not do a thorough analysis and our code encounters an unanticipated corner case, it will either get stuck or produce an unexpected outcome.

9.3.3 Harder Example: Assign a precedence number to an operator in an expression

Summary of chapter

Exercises

10. Element ⟷ Element

In the last chapter, we used two iterations one after another to find relations between elements and the entire dataset. In the examples that we considered, we saw that establishing such a relationship positions each element relative to the entire dataset. For example, we could say whether a student was above average, or we could assign a grade to the student based upon his relative position using the minimum and maximum mark values.

But what if the field of the dataset does not allow numerical operations such as addition to be performed? Then we cannot determine the average, nor the minimum or maximum. Even if the field admits numerical operations, we may not be happy with the coarse positioning of the element relative to the average, or some intermediate points based on the minimum, maximum and average. We may want to position the element much more precisely relative to some other elements.

In this chapter, we look at directly establishing pair-wise relationships between elements. As a simple example, consider the problem of determining if two students share the same birthday (i.e. are born in the same month and on the same day of the month). This will require us to compare every pair of students to check for this. As another example, we may want to determine if a customer has made two purchases of the same product from two different shops. We have to compare each bill of that customer with other bills of the same customer to check for this. A somewhat more complicated problem is that of resolving the personal pronouns (for example the word "he") in a paragraph - i.e. to identify the name of the (male) person that the personal pronoun ("he") refers to.

To compare every pair of elements, we have to generate all the pairs first. How do we do that? The basic iterator pattern allows us to go through all the elements (cards) systematically, without repeating any element twice. To compare an element picked in the first iteration with all other elements, We will need a second iteration that sits inside the first iteration, and so this pattern is called a **nested iteration** (one iteration within another).

10.1 Nested iterations

As we saw above in a nested iteration, there is an outer iteration and an inner iteration that sits within the outer iteration. At a step in the outer iteration, we are looking at a single element, say A. With this element A fixed, we now will need to do *another* iteration, where we go through all the elements (other than A) systematically. At each repeated step of the inner iteration, we will be looking at an element Y. The elements A (item from the outer iteration) is now paired with Y (element from the inner iteration). As we sweep through all the elements through the inner iteration, all pairs made with A will be found. We can then move to the next element in the outer iteration (which picks a different element B) and find all the pairs made with B. This can continue till all the pairs are found.

The problem with this procedure is that we have to keep track of "seen" elements at two places using only one set of cards - the cards that have been visited during the outer iteration, and again the cards that have been visited during the inner iteration. To do the inner iteration, we have to arrange all the cards again in a single pile (except the selected card that is set aside). But then we have lost all the information about which cards were seen in the outer iteration. To do this correctly, we will have to make a list of all the unique ids of cards in the "unseen" pile, lets call this list L. Then arrange all the cards into the "unseen" pile (except the selected card A), and do the inner iteration. After the inner iteration completes (all the cards will now be in the "unseen" pile), we have to go through the list L and move all the cards with unique ids in L back into the "unseen" pile. This looks like a very cumbersome procedure to do !

Fortunately, we do not have to go through such a long winded procedure. Let us look carefully at what is happening when we do the nested iterations. Note that for any two cards A and B, the pair is picked up twice - once when A is selected in the outer iteration and B in the inner iteration, and again when B is selected in the outer iteration, and A in the inner iteration. In other words, we will produce two pairs (A,B) and (B,A) through the nested iteration. For most practical applications (for example those in which we will need to only compare the fields of A and B), it is enough if only one of these two pairs, either (A,B) or (B,A) is produced - we don't need both.

What is the consequence of this? Go back to the nested iterations and look closely. Every element moves from the "unseen" pile to the "seen" pile in the outer iteration. So if we simply compare the element that is being moved to the "seen" pile with all the elements in the "unseen" pile, we will have produced exactly one out of (A,B) or (B,A). Can you see why this is so? If it was A that was picked in the outer iteration before B, then B is still in the "unseen" pile when A is picked, and so the pair (A,B) is produced. When B is picked, A is already moved into the "seen" pile, so we will never compare B with A - and so (B,A) will never be produced.

10.1.1 Flowchart for nested iterations

The generic flowchart for nested iterations discussed above is depicted in Figure 10.1 below.

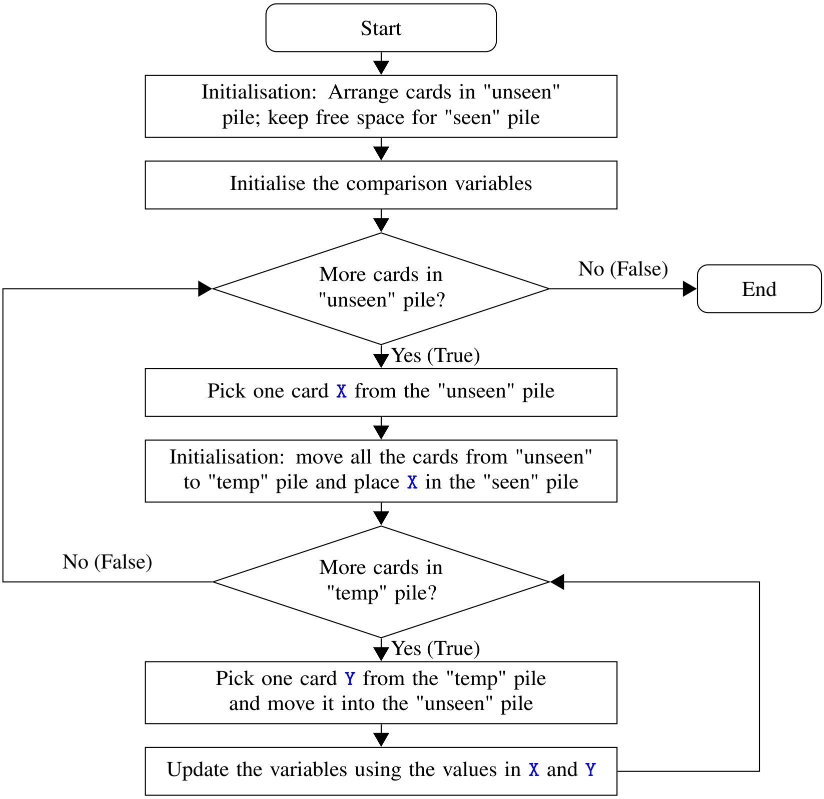

Start
Initialisation: Arrange cards in "unseen" pile; keep free space for "seen" pile
Initialise the comparison variables
More cards in "unseen" pile?
No (False)
End
Yes (True)
Pick one card X from the "unseen" pile
Initialisation: move all the cards from "unseen" to "temp" pile and place X in the "seen" pile
No (False)
More cards in "temp" pile?
Yes (True)
Pick one card Y from the "temp" pile and move it into the "unseen" pile
Update the variables using the values in X and Y

Figure 10.1: Generic flowchart for nested iterations

Observer the flowchart carefully. Unlike in the case of the sequenced iterations, where the exit from the first iteration took us to the initialisation step of the second iteration, here it is in reverse. The exit of the second (inner) iteration takes us to the exit condition box of the first (outer) iteration.

Note also that we are properly keeping track of the cards that have been visited. The "unseen" pile is supposed to keep track of cards that remain to be processed in the outer iteration. We now also need to keep track of cards visited during the inner iteration. To do that, we move all the cards from the "unseen" pile to a "temp" pile. So for the inner iteration, the "temp" pile holds all the unvisited cards, and the "unseen" pile holds the cards visted in the inner iteration, but not yet visited in the outer iteration. At the end of the inner iteration, all the cards are transferred back to the "unseen" pile, and the outer iteration can then proceed as if the "unseen" pile were never disturbed.

10.1.2 Pseudocode for nested iterations

The pseudocode for the generic nested iteration is shown below, where Pile 1 is the "unseen" pile, Pile 2 is the "seen" pile and Pile 3 is the "temp" pile.

```
Start
Arrange all cards in Pile 1, Initialise the variables
while (Pile 1 has more cards) {
  Pick a card X from Pile 1
  Move all the remaining cards from Pile 1 to Pile 3
  Move X to Pile 2
  while (Pile 3 has more cards) {
    Pick a card Y from Pile 3
    Move Y to Pile 1
    Update variables using the field values of X and Y
  }
}
End
```

Pseudocode 10.1: Nested iterations

10.1.3 Alternate flowchart for nested iterations

The trick we used to manage to keep track of cards in both the outer and the inner iteration was to move the "unseen" cards to a third "temp" pile, and move cards one by one in the inner iteration from the "temp" pile back to the "unseen" pile.

There is another way that we can produce all the pairs. Instead of comparing all the cards that are picked from the "unseen" pile with only the remaining cards in the "unseen" pile, we can compare all the cards that have been picked with only the cards in the "seen" pile. This will do the job as well !

Let us see why this is so. Consider a pair (A,B), with A coming before B in the outer iteration. When we pick the a card A, "seen" pile does not contain B, so (A,B) is not produced at this stage. But after this, A is moved to the "seen" pile so that when we pick the card B, B is now compared all the elements in the "seen" pile - so in particular we will compare B with A. Note that this way of doing things will produce the pair (B,A), whereas the earlier one produced the pair (A,B).

The generic flowchart for nested iterations discussed above is depicted in Figure 10.2 below.

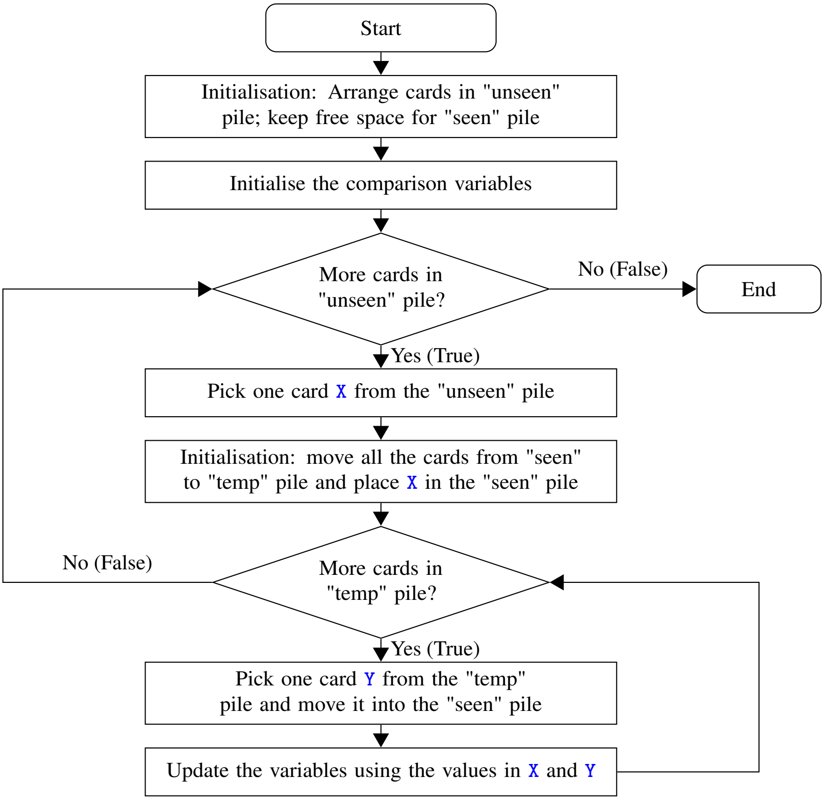

Start
Initialisation: Arrange cards in "unseen" pile; keep free space for "seen" pile
Initialise the comparison variables
More cards in "unseen" pile?
No (False)
End
Yes (True)
Pick one card X from the "unseen" pile
Initialisation: move all the cards from "seen" to "temp" pile and place X in the "seen" pile
No (False)
More cards in "temp" pile?
Yes (True)
Pick one card Y from the "temp" pile and move it into the "seen" pile
Update the variables using the values in X and Y

Figure 10.2: Alternative flowchart for nested iterations

Observer the difference between this flowchart and the earlier one. Here, we move all the cards (except the current card X) from the "seen" pile to a "temp" pile. So for the inner iteration, the "temp" pile holds all the unvisited cards. At the end of the inner iteration, all the cards are transferred back from the "temp" pile to the "seen" pile. The outer iteration can then proceed as if the "seen" pile were never disturbed.

10.1.4 Alternate pseudocode for nested iterations

The pseudocode for this alternate nested iteration is shown below, where Pile 1 is the "unseen" pile, Pile 2 is the "seen" pile and Pile 3 is the "temp" pile.

```
Start
Arrange all cards in Pile 1, Initialise the variables
while (Pile 1 has more cards) {
  Pick a card X from Pile 1
  Move all cards from Pile 2 to Pile 3
  Move X to Pile 2
  while (Pile 3 has more cards) {
    Pick a card Y from Pile 3
    Move Y to Pile 2
    Update variables using the field values of X and Y
  }
}
End
```

Pseudocode 10.2: Another way to do nested iterations

10.1.5 List of all pairs of elements

As the first example, let us just return all the pairs of elements as a list pairsList. The pseudocode to do this 9using the alternate method above) is shown below.

```
Start
Arrange all cards in Pile 1, Initialise pairsList to []
while (Pile 1 has more cards) {
  Pick a card X from Pile 1
  Move all cards from Pile 2 to Pile 3
  Move X to Pile 2
  while (Pile 3 has more cards) {
    Pick a card Y from Pile 3
    Move Y to Pile 2
    Append [ (X,Y) ] to pairsList
  }
}
```

```
End
```

Pseudocode 10.3: List of all the pairs

10.2 Reducing the number of comparisons

Let us now consider the concrete example of checking if the classroom dataset has a pair of students who share the same birthday (i.e. the same month of birth and the same date within the month). The simplest way to do this may be to first generate all the pairs of students (pairsList)from the dataset as above, and then iterate through this list checking each pair (A,B) to see if A and B share the same birthday. But this requires two passes over all pairs of students. Can't we do the comparison within the nested iteration itself? This way, we can also exit the iteration when we find a pair that meets our desired condition. To exit from the loop early, we will use a Boolean variable found, as we saw in Figure 4.18.

The modified pseudocode for checking if two students have the same birthday is shown below (using the generic pseudocode for nested iteration):

```
Start
Arrange all cards from the classroom dataset in Pile 1
found = False
while (Pile 1 has more cards AND NOT found) {
  Pick a card X from Pile 1
  Move all remaining cards from Pile 1 to Pile 3
  Move X to Pile 2
  while (Pile 3 has more cards AND NOT found) {
    Pick a card Y from Pile 3
    Move Y to Pile 1
    if (X.dateOfBirth == Y.dateOfBirth) {
      found = True
    }
  }
}
End
```

Pseudocode 10.4: Same birthday

Note that we exit *both* the outer and the inner iterations if a pair is found with the same birthday (by adding an AND NOT `found` in both the exit conditions). When `found` becomes True, it will cause an exit of the inner iteration, which takes us directly to the exit condition of the outer iteration, which will also exit.

10.2.1 Estimating the number of comparisons required

How many comparisons are required to be done before we can conclusively say whether a pair of students share a birthday? If there is no such pair, then we have to examine all the pairs before we can conclude this to be the case. The entire nested iteration will exit with `found` remaining False. How many comparisons were carried out?

If there are N elements in the classroom dataset, then the number of all pairs is $N \times N = N^2$. However, we will never compare an element with itself, so the number of meaningful comparisons is reduced by N, and do becomes $N \times N - N$ which can be rewritten as $N \times (N-1)$. We saw that our pseudocode will avoid duplicate comparisons (if (A,B) is produced, then the reverse (B,A) will not be produced due to the trick we employed in maintaining the piles). So the number of comparisons is reduced to half of the number of meaningful comparisons, i.e. to $\frac{N \times (N-1)}{2}$.

But is the same as the number of comparisons produced by the generic nested iteration in Figure 10.1? Let us check. The first element picked in the outer iteration will be compared with all the "unseen" elements (there are $N-1$ of these after the first element is picked). The second element will be compared with only $N-2$ elements since two elements would have been moved to the "seen" pile from the "unseen" pile. So, we can see that the number of comparisons will be $(N-1)+(N-2)+...+2+1$ which can be written in reverse as $1+2+...+(N-2)+(N-1)$ that we know from our school algebra to be equal to $\frac{N \times (N-1)}{2}$.

What about the alternate nested iteration method in Figure 10.2? The first element from the outer loop will have nothing to be compared with (since "seen" will be empty). The second element will be compared with 1 element (the first element). At the end, the last element will be compared with the entire "seen" pile which will have all the remaining elements, i.e. $N-1$ elements. So the number of comparisons is just $1+2+...+(N-2)+(N-1)$ which is equal to $\frac{N \times (N-1)}{2}$. So it checks out !

The problem with nested iterations is that the number of comparisons can grow really large when the dataset size N is large, since it is a quadratic in N, as shown in Figure 10.3.

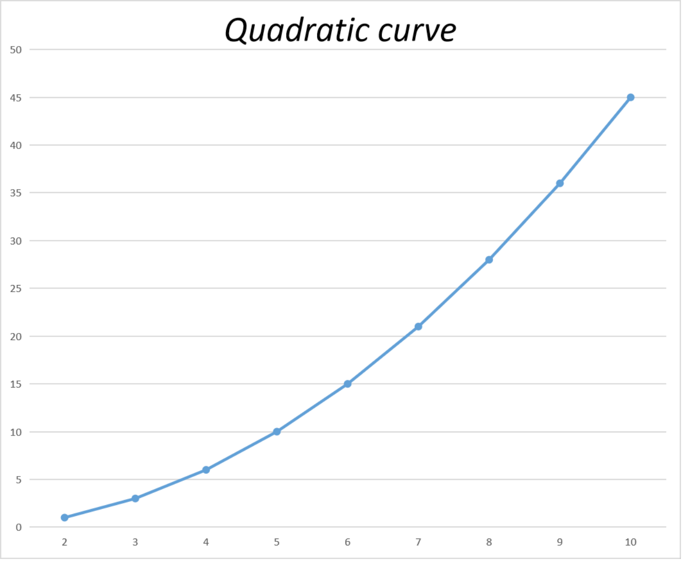


Figure 10.3

The following table shows how the number of comparisons grows with N. As we can see, when N is one million, the number of comparisons becomes really really large.

N	$\frac{N \times (N-1)}{2}$
2	1
3	3
4	6
5	10
6	15
7	21
8	28
9	36
10	45
100	49500
1000	499500
10000	49995000
100000	4999950000
1000000	499999500000

10.2.2 Using binning to reduce the number of comparisons

As we saw above, the number of comparison pairs in nested iterations grows quadratically with the number of elements. This becomes a major issue with large datasets. Can we somehow reduce the number of comparisons to make it more manageable?

In the example above where we are trying to find two students with the same birthday, the problem has a deeper structure that we can exploit. Note that two students will share the same birthday only if firstly they share the same month of birth. Which means that there is simply no sense in comparing two students if they are born in different months. This should result in a substantial reduction in comparisons.

In order to take advantage of this, we have to first separate the students into bins, one for each month. Let us name the bins Bin[1], Bin[2], ..., Bin[12]. So for example, Bin[3] will be the list of all students who are born in March. We should first process all the cards in the classroom dataset and put each of them in the correct bin. Once this is done, we can go through the bins one by one and compare each pair of cards within the bin.

The pseudocode to do this is shown below. It uses a procedure **processBin**(bin) to do all the comparisons within each bin, whose body will be exactly the same as the Same birthday pseudocode that we saw earlier, except that instead of processing the entire classroom dataset, it will process only the cards in Bin[i] and return the value of `found`. The procedure **month**(`X`) returns the month part of `X.dateOfBirth` as a numeral between 1 and 12.

```
Start
Arrange all cards from the classroom dataset in Pile 1
Initialise Bin[i] to [] for all i from 1 to 12
while (Pile 1 has more cards) {
  Pick a card X from Pile 1
  Move X to Pile 2
  Append [X] to Bin[month(X)]
}

found = False, i = 1
while (i < 13 AND NOT found) {
  found = processBin(Bin[i])
  i = i + 1
}
```

```
    End
```

Pseudocode 10.5: Same birthday using binning

10.2.3 Improvement achieved due to binning

can we make an estimate of the reduction in comparisons achieved through binning? Let us look at the boundary conditions first. If all the students in the classroom dataset happen to have the same month of birth, then there is simply no reduction in the number of comparisons ! So, in the worst case, binning achieves nothing.

At the other extreme, consider the situation where the students are equally distributed into the 12 bins Bin[1],...,Bin[12] (of course this also means that the total number of students N is a multiple of 12. Clearly, this will yield the least number of comparisons. The size of each bin will be $N/12$ and the number of comparisons within each bin will be $\frac{N/12 \times (N/12-1)}{2} = \frac{N \times (N-12)}{288}$. Then the total number of comparisons is just 12 times this number, i.e. it is $\frac{N \times (N-12)}{24}$.

Say $N = 36$, then the number of comparisons without binning would be $\frac{N \times (N-1)}{2} = \frac{36 \times 35}{2} = 630$. With binning, it would be $\frac{N \times (N-12)}{24} = \frac{36 \times 24}{24} = 36$. So, we have reduced 630 comparisons to just 36 comparisons ! The reduction factor R tells us by how many times the number of comparisons has reduced. In this example, the number of comparison is reduced by $R = 630/36 = 17.5$ times.

In the general case, if the number of bins is K, and the N elements are equally divided into the bins, then we can easily calculate the reduction factor in the same manner as above to be $R = \frac{N-1}{N/K-1}$.

In most practical situations, we are not likely to see such an extreme case where all students have the same month of birth. Nor is it likely that the students will be equally distributed into the different bins. So, the reduction in number of comparisons will be somewhere between 0 and R times.

10.3 Examples of nested iterations

We now look at a few more examples of finding pairs of elements using nested iterations. In each case, we will try to reduce the number of comparisons by using binning wherever possible. If binning does not work, we will try to use some other structural information in the problem (such as *state*) to reduce the number of comparisons.

10.3.1 Study group pairing

In any class, students typically pair up (or form small groups), so that within each pair, one student can help the other in at least one subject. For a study pair (A,B) to be meaningful, A has to be able to help B in some subject. This will clearly benefit B (at least for that subject). For this to be of benefit to A, it should also be the case that B can help A in some other subject. When can we say that A can help B in a subject? We can define this as A's marks in some subject is at least 10 marks more than that of B in that subject. So (A,B) would form a meaningful study pair if there are subjects S_1 and S_2 such that A's marks exceeds B's marks in S_1 by at least 10, and B's marks exceeds A's marks in S_2 by at least 10.

The pseudocode for finding the study pairs taking into account only two subjects Maths and Physics is shown below. The result is a list studyPairs of student pairs who can help each other.

```
Start
Arrange all cards from the classroom dataset in Pile 1
studyPairs = []
while (Pile 1 has more cards) {
  Pick a card X from Pile 1
  Move all remaining cards from Pile 1 to Pile 3
  Move X to Pile 2
  while (Pile 3 has more cards) {
    Pick a card Y from Pile 3
    Move Y to Pile 1
    if (X.mathsMarks - Y.mathsMarks > 9 AND
      Y.physicsMarks - X.physicsMarks > 9) {
      Append [(X, Y)] to studyPairs
    }
    if (Y.mathsMarks - X.mathsMarks > 9 AND
      X.physicsMarks - Y.physicsMarks > 9) {
      Append [(X, Y)] to studyPairs
    }
  }
}
End
```

Pseudocode 10.6: Study group pairing

The number of comparisons in this pseudocode will be large because we are comparing every student with every other student. Can we reduce the number of comparisons using some form of binning?

There is no obvious field that we can use (like the month of birth in the case of the same birthday problem). But, we can make a simplification which could help us. Observe that if a pair (A,B) is produced by the above pseudocode, then A should be better than B in either Maths (or Physics), and B should be better in A in Physics (or Maths). This would mean that it is quite likely that the *sum* of the Maths and Physics marks will be roughly the same ! Of course, this need not be always true, but if it works in most of the cases, that is good enough for us. Such simplifying assumptions that works in most cases is called a *heuristic*.

Now that we have this simplifying heuristic, we can bin the students into groups based on their sum of the Maths and Physics marks. This sum can be any number between 0 and 200, but since most of the students have marks above 50, we are more likely to find the range to be between 100 and 200. So let us bin the students according to whether the sum of their Maths and Physics marks is in the ranges 0-125, 125-150, 150-175 or 175-200, which gives us 4 bins to work with. Let us call these bins binD, binC, binB and binA (as if we are awarding grades to the students based on the sum of their Maths and Physics marks). Then the simplifying heuristic tells us that we should look for pairs only within a bin and not across bins.

Let us first convert the pseudocode in 10.6 into a procedure that works within one bin.

```
Procedure processBin(bin)
Arrange all cards from bin in Pile 1
binPairs = []
while (Pile 1 has more cards) {
  Pick a card X from Pile 1
  Move all remaining cards from Pile 1 to Pile 3
  Move X to Pile 2
  while (Pile 3 has more cards) {
    Pick a card Y from Pile 3
    Move Y to Pile 1
    if (X.mathsMarks - Y.mathsMarks > 9 AND
      Y.physicsMarks - X.physicsMarks > 9) {
      Append [(X, Y)] to binPairs
    }
    if (Y.mathsMarks - X.mathsMarks > 9 AND
      X.physicsMarks - Y.physicsMarks > 9) {
      Append [(X, Y)] to binPairs
    }
  }
}
return (binPairs)
End processBin
```

Pseudocode 10.7: Study group pairing procedure

The pseudocode for finding study pairs using binning is now shown below.

```
Start
Arrange all cards from the classroom dataset in Pile 1
Initialise binA to [], binB to [], binC to [], binD to []
while (Pile 1 has more cards) {
  Pick a card X from Pile 1
  Move X to Pile
  if (X.mathsMarks + X.physicsMarks < 125) {
    Append [X] to binD
  }
  else if (X.mathsMarks + X.physicsMarks < 150) {
    Append [X] to binC
  }
  else if (X.mathsMarks + X.physicsMarks < 175) {
    Append [X] to binB
  }
```

```
        else {
            Append [X] to binA
        }
    }

    studyPairs = []
    Append processBin (binA) to studyPairs
    Append processBin (binB) to studyPairs
    Append processBin (binC) to studyPairs
    Append processBin (binD) to studyPairs
    End
```

Pseudocode 10.8: Study group pairing using binning

Note that through the use of a simplifying heuristic (that we are more likely to find pairs within the bins rather than across the bins), we can reduce the comparisons substantially.

10.3.2 Matching pronouns to nouns

Let us now turn our attention to the words dataset, which are words drawn from a paragraph. Typically, a paragraph is broken down into sentences, each of which could be a simple sentence (i.e. has a single clause) or a compound sentence (made of multiple clauses connected by a comma, semicolon or a conjunction). The sentence itself will have a verb, some nouns and perhaps some adjectives or adverbs to qualify the nouns and verbs. In most sentences, the noun will be replaced by a short form of the noun, called a pronoun. For instance the first sentence may use the name of a person as the noun (such as Swaminathan), while subsequent references to this person may be depicted using an appropriate personal pronoun (he, his, him etc).

Now, the question we are asking here is this: given a particular personal pronoun (such as the word "he"), can we determine who this "he" refers to in the paragraph? Clearly, "he" has to refer to a male person - so we can look at an appropriate word that represents a name (and hopefully a male name). If we look back from the pronoun word "he" one word at a time, we should be able to find the closest personal noun that occurs before this pronoun. Maybe that is the person which is referred to by "he"? This is a good first approximation to the solution. So let us see how to write it in pseudocode form.

Note that unlike in the other datasets, in the words dataset, the order of the words matters (since the meaning of a sentence changes entirely or becomes meaningless if the words are jumbled). Note also that for a given pronoun, we have to search all the cards before that pronoun to look for a personal noun, and that we will search in a specific order - we start from the word just before the pronoun and start moving *back* through the words one by one till we find the desired noun.

The pseudocode given below attempts to do find all the possible matches for each pronoun in a list `possibleMatches`. It uses the Boolean procedures `isProperName` that checks if a noun is a proper name, and `personalPronoun` that checks if the pronoun is a personal pronoun (code for both not written here !). Recall that the "seen" cards will be the words before the current word, while the "unseen" words will be the ones after the current word. So, the alternate pseudocode in 10.2 is the one we need to use.

```
Start
Arrange all words cards in Pile 1 in order
possibleMatches = []
while (Pile 1 has more cards) {
  Pick a card X from Pile 1
  if (X.partOfSpeech == Pronoun AND
      personalPronoun(X.word)) {
    Move all cards from Pile 2 to Pile 3 (retaining the order)
    while (Pile 3 has more cards) {
      Pick the top card Y from Pile 3
      Move Y to the bottom of Pile 2
      if (Y.partOfSpeech == Noun AND
          isProperName(Y.word)) {
        Append [(X,Y)] to possibleMatches
      }
    }
  }
  Move X to the top of Pile 2
}
End
```

Pseudocode 10.9: Possible matches for each pronoun

There are a number of interesting things going on in this pseudocode. Firstly note that we are not carrying out the inner iteration for all the words, we start an inner iteration only when the current word is a personal pronoun. So, this already reduces the number of comparisons to be done.

Secondly, when we are at such a personal pronoun, we start looking in the "seen" pile (Pile 2 moved to Pile 3). To make sure that we are looking through the "seen" pile in such a way that we are reading *backwards* in the paragraph, the Pile 2 has to be maintained in *reverse* order of the words in Pile 1. This is ensured by placing the card picked up on the *top* of Pile 2. While transferring cards from Pile 2 to Pile 3, we need to preserve this reverse order. The inner iteration should not disturb the order of cards, so when we are done examining a card from Pile 3, we place it at the bottom of Pile 2, so that after the inner iteration is done, Pile 2 will be exactly as before. The code to move card `X` to Pile 2 is moved after the inner iteration, so that it does not affect the "seen" pile (Pile 2) over which we need to search. It also needs to be done whether we are executing the inner loop or not.

Finally, note that we do not stop when we find the proper name noun `Y` matching the pronoun `X`, since this match may in some cases not be the correct one (but we keep the pair as the first possible match in our list). If we are interested only in finding the first possible match, we can use a Boolean variable `found` which is set to True when the match is found, and the inner iteration can exit on this condition.

10.3.3 Finding connecting trains

10.3.4 Finding matching brackets in an expression

10.3.5 Finding overlapping rectangles

Summary of chapter

Exercises

11. Lists

11.1 List as a set or collection

11.1.1 Example: Collect all students born in May

11.1.2 Example: Collect all students from Chennai

11.1.3 Example: Collect all the bills of one customer

11.1.4 Example: Collect all the items sold of one category

11.1.5 Example: Collect the first verb from each sentence

11.1.6 Example: Collect all trains between two stations

11.1.7 Example: Find all rectangles at the leftmost boundary

11.1.8 How do we store the same element in two different lists ?

11.1.9 Using indexing: Lists of indices rather than lists of elements

11.2 Using the list collections

11.2.1 Example: Students scoring over 80 in all subjects

11.2.2 Example: Are the Chennai students doing better than the Bangalore students?

11.2.3 Example: Which shop has the most loyal customers?

11.2.4 Example: Is there a word that is both a noun and a verb?

11.2.5 Example: Finding the rectangles at the boundary

Summary of chapter

Exercises

12. Procedures with Side Effects

Summary of chapter

Exercises

13. Sorted Lists

13.1 Keeping lists in sorted form may speedup subsequent processing

13.1.1 Example: Assign grades to students

13.1.2 Example: Match pronoun and noun

13.1.3 Example: Find transit station for going from one city to another

13.1.4 Example: Find the subject noun for a verb in a sentence

13.2 How do we sort?

13.2.1 Insertion sort

13.2.2 Pseudocode for insertion sort

13.3 Using the insertion sort procedure

13.3.1 Pseudocode: Assigning grades to students

13.3.2 Pseudocode: Matching pronoun to noun

Summary of chapter

Exercises

14. Dictionaries

14.0.1 Finding the frequency of all the words

14.0.2 Are the words with high frequency short?

If we take a look at the whole paragraph in the words dataset, we can observe that there are quite a number of short words, and some of these seem to occur more than once. So this leads us to ask the hypothetical question: "Are the high frequency words mostly short?". How do we determine this? What does "mostly" mean?

The usual way to settle questions of the kind: "Do elements with property A satisfy property B?" is to put all the elements into 4 buckets - A and B, A and not B, not A and B, not A and not B. This is shown in the table below, where the top row is B and the bottom row is not B, the left column is not A and the right column is A:

not A and B	A and B
not A and not B	A and not B

We then count the number of elements for each box of the table, and represent them using percentages of the total number of elements. If the majority of elements fall along the diagonal (the sum of the percentages of the boxes (not A and not B) and (A and B) is much higher than the sum of percentages in the boxes (A and not B) and (not A and B)), then we know that A and B go mostly together, and so the hypothesis must be true.

We are now ready to carry out this procedure as shown in the pseudocode below. The variables HFHL, HFLL, LFHL and LFLL represent high frequency and high length, high frequency and low length, low frequency and high length and low frequency and low length, where high and low represent above average and below average values respectively.

14.0.3 Finding the total spend of each customer

14.0.4 Do the high spenders shop at the biggest shop?

14.0.5 Total distance and total time of all the trains

14.0.6 Are the longer distance trains the fastest?

Summary of chapter

Exercises

15. Dictionaries with Lists

Summary of chapter

Exercises

16. Graphs

Summary of chapter

Exercises

17. Graphs, Lists and Dictionaries

Summary of chapter

Exercises

Part III

Higher Level Computational Thinking

18. Recursion

Summary of chapter

Exercises

19. Recursion Trees

Summary of chapter

Exercises

20. Spanning Trees

Summary of chapter

Exercises

21. Object Oriented Computing

Summary of chapter

Exercises

22. Functional Computing

Summary of chapter

Exercises

23. Concurrency

Summary of chapter

Exercises

24. Input and Output

Summary of chapter

Exercises

25. Real Time Systems

Summary of chapter

Exercises

26. Bottom-up Computing

Summary of chapter

Exercises

27. Summary & Further Reading