40 Algorithms Every Programmer Should Know

Hone your problem-solving skills by learning different algorithms and their implementation in Python

Packt

www.packt.com

Imran Ahmad

40 Algorithms Every Programmer Should Know

Commissioning Editor: Kunal Chaudhari
Acquisition Editor: Karan Gupta
Content Development Editor: Pathikrit Roy
Senior Editor: Rohit Singh
Technical Editor: Pradeep Sahu
Copy Editor: Safis Editing
Project Coordinator: Francy Puthiry
Proofreader: Safis Editing
Indexer: Rekha Nair
Production Designer: Nilesh Mohite

First published: June 2020

Production reference: 1120620

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78980-121-7

www.packt.com

Table of Contents

Preface

Algorithms have always played an important role both in the science and practice of computing. This book focuses on utilizing these algorithms to solve real-world problems. To get the most out of these algorithms, a deeper understanding of their logic and mathematics is imperative. You'll start with an introduction to algorithms and explore various algorithm design techniques. Moving on, you'll learn about linear programming, page ranking, and graphs, and even work with machine learning algorithms, understanding the math and logic behind them. This book also contains case studies, such as weather prediction, tweet clustering, and movie recommendation engines, that will show you how to apply these algorithms optimally. As you complete this book, you will become confident in using algorithms for solving real-world computational problems.

Who this book is for

This book is for the serious programmer! Whether you are an experienced programmer looking to gain a deeper understanding of the math behind the algorithms or have limited programming or data science knowledge and want to learn more about how you can take advantage of these battle-tested algorithms to improve the way you design and write code, you'll find this book useful. Experience with Python programming is a must, although knowledge of data science is helpful but not necessary.

What this book covers

`Chapter 1`, *Overview of Algorithms*, summarizes the fundamentals of algorithms. It starts with a section on the basic concepts needed to understand the working of different algorithms. It summarizes how people started using algorithms to mathematically formulate certain classes of problems. It also mentions the limitations of different algorithms. The next section explains the various ways to specify the logic of an algorithm. As Python is used in this book to write the algorithms, how to set up the environment in order to run the examples is explained next. Then, the various ways in which an algorithm's performance can be quantified and compared against other algorithms are discussed. Finally, this chapter discusses various ways in which a particular implementation of an algorithm can be validated.

`Chapter 2`, *Data Structures Used in Algorithms*, focuses on algorithms' need for necessary in-memory data structures that can hold the temporary data. Algorithms can be data-intensive, compute-intensive, or both. But for all different types of algorithms, choosing the right data structures is essential for their optimal implementation. Many algorithms have recursive and iterative logic and require specialized data structures that are fundamentally iterative in nature. As we are using Python in this book, this chapter focuses on Python data structures that can be used to implement the algorithms discussed in this book.

`Chapter 3`, *Sorting and Searching Algorithms*, presents core algorithms that are used for sorting and searching. These algorithms can later become the basis for more complex algorithms. The chapter starts by presenting different types of sorting algorithms. It also compares the performance of various approaches. Then, various algorithms for searching are presented. They are compared and their performance and complexity are quantified. Finally, this chapter presents the actual applications of these algorithms.

`Chapter 4`, *Designing Algorithms*, presents the core design concepts of various algorithms. It also explains different types of algorithms and discusses their strengths and weaknesses. Understanding these concepts is important when it comes to designing optimal complex algorithms. The chapter starts by discussing different types of algorithmic designs. Then, it presents the solution for the famous traveling salesman problem. It then discusses linear programming and its limitations. Finally, it presents a practical example that shows how linear programming can be used for capacity planning.

`Chapter 5`, *Graph Algorithms*, focuses on the algorithms for graph problems that are common in computer science. There are many computational problems that can best be represented in terms of graphs. This chapter presents methods for representing a graph and for searching a graph. Searching a graph means systematically following the edges of the graph so as to visit the vertices of the graph. A graph-searching algorithm can discover a lot about the structure of a graph. Many algorithms begin by searching their input graph to obtain this structural information. Several other graph algorithms elaborate on basic graph searching. Techniques for searching a graph lie at the heart of the field of graph algorithms. The first section discusses the two most common computational representations of graphs: as adjacency lists and as adjacency matrices. Next, a simple graph-searching algorithm called *breadth-first search* is presented and shows how to create a breadth-first tree. The following section presents the depth-first search and provides some standard results about the order in which a depth-first search visits vertices.

`Chapter 6`, *Unsupervised Machine Learning Algorithms*, introduces unsupervised machine learning algorithms. These algorithms are classified as unsupervised because the model or algorithm tries to learn inherent structures, patterns, and relationships from given data without any supervision. First, clustering methods are discussed. These are machine learning methods that try to find patterns of similarity and relationships among data samples in our dataset and then cluster these samples into various groups, such that each group or cluster of data samples has some similarity, based on the inherent attributes or features. The following section discusses dimensionality reduction algorithms, which are used when we end up having a number of features. Next, some algorithms that deal with anomaly detection are presented. Finally, this chapter presents association rule-mining, which is a data mining method used to examine and analyze large transactional datasets to identify patterns and rules of interest. These patterns represent interesting relationships and associations, among various items across transactions.

`Chapter 7`, *Traditional Supervised Learning Algorithms*, describes traditional supervised machine learning algorithms in relation to a set of machine learning problems in which there is a labeled dataset with input attributes and corresponding output labels or classes. These inputs and corresponding outputs are then used to learn a generalized system, which can be used to predict results for previously unseen data points. First, the concept of classification is introduced in the context of machine learning. Then, the simplest of the machine learning algorithms, linear regression, is presented. This is followed by one of the most important algorithms, the decision tree. The limitations and strengths of decision tree algorithms are discussed, followed by two important algorithms, SVM and XGBoost.

`Chapter 8`, *Neural Network Algorithms*, first introduces the main concepts and components of a typical neural network, which is becoming the most important type of machine learning technique. Then, it presents the various types of neural networks and also explains the various kinds of activation functions that are used to realize these neural networks. The backpropagation algorithm is then discussed in detail. This is the most widely used algorithm to converge the neural network problem. Next, the transfer learning technique is explained, which can be used to greatly simplify and partially automate the training of models. Finally, how to use deep learning to detect objects in multimedia data is presented as a real-world example.

`Chapter 9`, *Algorithms for Natural Language Processing*, presents algorithms for **natural language processing** (**NLP**). This chapter proceeds from the theoretical to the practical in a progressive manner. First, it presents the fundamentals, followed by the underlying mathematics. Then, it discusses one of the most widely used neural networks to design and implement a couple of important use cases for textual data. The limitations of NLP are also discussed. Finally, a case study is presented where a model is trained to detect the author of a paper based on the writing style.

`Chapter 10`, *Recommendation Engines,* focuses on recommendation engines, which are a way of modeling information available in relation to user preferences and then using this information to provide informed recommendations on the basis of that information. The basis of the recommendation engine is always the recorded interaction between the users and products. This chapter begins by presenting the basic idea behind recommendation engines. Then, it discusses various types of recommendation engines. Finally, this chapter discusses how recommendation engines are used to suggest items and products to different users.

`Chapter 11`, *Data Algorithms,* focuses on the issues related to data-centric algorithms. The chapter starts with a brief overview of the issues related to data. Then, the criteria for classifying data are presented. Next, a description of how to apply algorithms to streaming data applications is provided and then the topic of cryptography is presented. Finally, a practical example of extracting patterns from Twitter data is presented.

`Chapter 12`, *Cryptography,* introduces the algorithms related to cryptography. The chapter starts by presenting the background. Then, symmetrical encryption algorithms are discussed. MD5 and SHA hashing algorithms are explained and the limitations and weaknesses associated with implementing symmetric algorithms are presented. Next, asymmetric encryption algorithms are discussed and how they are used to create digital certificates. Finally, a practical example that summarizes all these techniques is discussed.

`Chapter 13`, *Large-Scale Algorithms,* explains how large-scale algorithms handle data that cannot fit into the memory of a single node and involve processing that requires multiple CPUs. This chapter starts by discussing what types of algorithms are best suited to be run in parallel. Then, it discusses the issues related to parallelizing the algorithms. It also presents the CUDA architecture and discusses how a single GPU or an array of GPUs can be used to accelerate the algorithms and what changes need to be made to the algorithm in order to effectively utilize the power of the GPU. Finally, this chapter discusses cluster computing and discusses how Apache Spark creates **resilient distributed datasets** (**RDDs**) to create an extremely fast parallel implementation of standard algorithms.

`Chapter 14`, *Practical Considerations*, starts with the important topic of explainability, which is becoming more and more important now that the logic behind automated decision making has been explained. Then, this chapter presents the ethics of using an algorithm and the possibilities of creating biases when implementing them. Next, the techniques for handling NP-hard problems are discussed in detail. Finally, ways to implement algorithms, and the real-world challenges associated with this, are summarized.

To get the most out of this book

Chapter number	Software required (with version)	Free/Proprietary	Hardware specifications	OS required
1-14	Python version 3.7.2 or later	Free	Min 4GB of RAM, 8GB +Recommended.	Windows/Linux/Mac

If you are using the digital version of this book, we advise you to type the code yourself or access the code via the GitHub repository (link available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from your account at `www.packt.com`. If you purchased this book elsewhere, you can visit `www.packtpub.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packt.com`.
2. Select the **Support** tab.
3. Click on **Code Downloads**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/40-Algorithms-Every-Programmer-Should-Know`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://static.packt-cdn.com/downloads/9781789801217_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Let's see how to add a new element to a stack by using `push` or removing an element from a stack by using `pop`."

A block of code is set as follows:

```
define swap(x, y)
    buffer = x
    x = y
    y = buffer
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
define swap(x, y)
    buffer = x
    x = y
    y = buffer
```

Any command-line input or output is written as follows:

```
pip install a_package
```

Bold: Indicates a new term, an important word, or words that you see on screen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "One way to reduce the complexity of an algorithm is to compromise on its accuracy, producing a type of algorithm called an **approximate algorithm**."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at customercare@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

1

Section 1: Fundamentals and Core Algorithms

This section introduces us to the core aspects of algorithms. We will explore what an algorithm is and how to design it, and also learn about the data structures used in algorithms. This section also gives a deep idea on sorting and searching algorithms along with the algorithms to solve graphical problems. The chapters included in this section are:

1
Overview of Algorithms

This book covers the information needed to understand, classify, select, and implement important algorithms. In addition to explaining their logic, this book also discusses data structures, development environments, and production environments that are suitable for different classes of algorithms. We focus on modern machine learning algorithms that are becoming more and more important. Along with the logic, practical examples of the use of algorithms to solve actual everyday problems are also presented.

This chapter provides an insight into the fundamentals of algorithms. It starts with a section on the basic concepts needed to understand the workings of different algorithms. This section summarizes how people started using algorithms to mathematically formulate a certain class of problems. It also mentions the limitations of different algorithms. The next section explains the various ways to specify the logic of an algorithm. As Python is used in this book to write the algorithms, how to set up the environment to run the examples is explained. Then, the various ways that an algorithm's performance can be quantified and compared against other algorithms are discussed. Finally, this chapter discusses various ways a particular implementation of an algorithm can be validated.

To sum up, this chapter covers the following main points:

- What is an algorithm?
- Specifying the logic of an algorithm
- Introducing Python packages
- Algorithm design techniques
- Performance analysis
- Validating an algorithm

What is an algorithm?

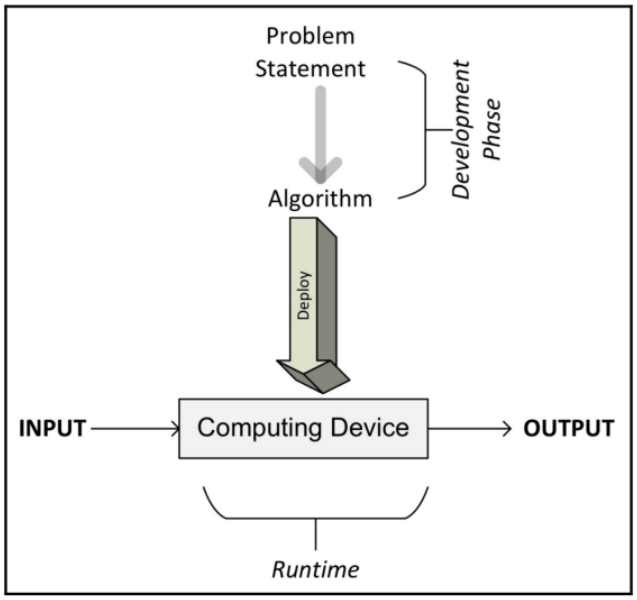
In the simplest terms, an algorithm is a set of rules for carrying out some calculations to solve a problem. It is designed to yield results for any valid input according to precisely defined instructions. If you look up the word algorithm in an English language dictionary (such as American Heritage), it defines the concept as follows:

> *"An algorithm is a finite set of unambiguous instructions that, given some set of initial conditions, can be performed in a prescribed sequence to achieve a certain goal and that has a recognizable set of end conditions."*

Designing an algorithm is an effort to create a mathematical recipe in the most efficient way that can effectively be used to solve a real-world problem. This recipe may be used as the basis for developing a more reusable and generic mathematical solution that can be applied to a wider set of similar problems.

The phases of an algorithm

The different phases of developing, deploying, and finally using an algorithm are illustrated in the following diagram:



As we can see, the process starts with understanding the requirements from the problem statement that detail what needs to be done. Once the problem is clearly stated, it leads us to the development phase.

The development phase consists of two phases:

- **The design phase**: In the design phase, the architecture, logic, and implementation details of the algorithm are envisioned and documented. While designing an algorithm, we keep both accuracy and performance in mind. While searching for the solution to a given problem, in many cases we will end up having more than one alternative algorithm. The design phase of an algorithm is an iterative process that involves comparing different candidate algorithms. Some algorithms may provide simple and fast solutions but may compromise on accuracy. Other algorithms may be very accurate but may take considerable time to run due to their complexity. Some of these complex algorithms may be more efficient than others. Before making a choice, all the inherent tradeoffs of the candidate algorithms should be carefully studied. Particularly for a complex problem, designing an efficient algorithm is really important. A correctly designed algorithm will result in an efficient solution that will be capable of providing both satisfactory performance and reasonable accuracy at the same time.
- **The coding phase**: In the coding phase, the designed algorithm is converted into a computer program. It is important that the actual program implements all the logic and architecture suggested in the design phase.

The designing and coding phases of an algorithm are iterative in nature. Coming up with a design that meets both functional and non-functional requirements may take lots of time and effort. Functional requirements are those requirements that dictate what the right output for a given set of input data is. Non-functional requirements of an algorithm are mostly about the performance for a given size of data. Validation and performance analysis of an algorithm are discussed later in this chapter. Validating an algorithm is about verifying that an algorithm meets its functional requirements. Performance analysis of an algorithm is about verifying that it meets its main non-functional requirement: performance.

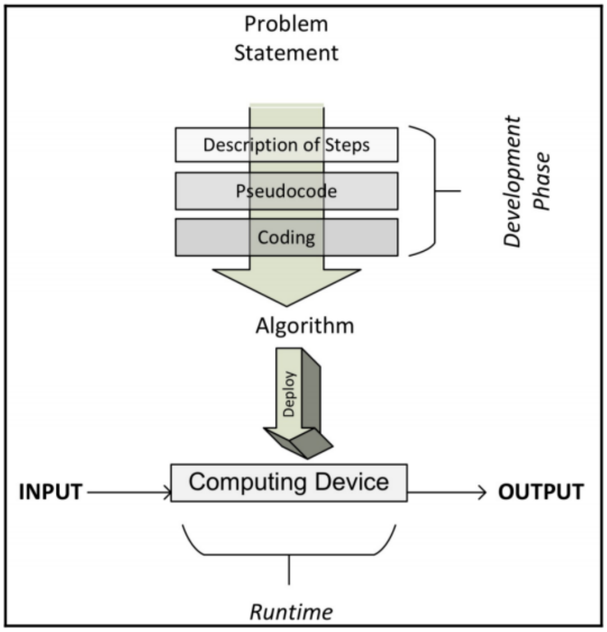
Once designed and implemented in a programming language of your choice, the code of the algorithm is ready to be deployed. Deploying an algorithm involves the design of the actual production environment where the code will run. The production environment needs to be designed according to the data and processing needs of the algorithm. For example, for parallelizable algorithms, a cluster with an appropriate number of computer nodes will be needed for the efficient execution of the algorithm. For data-intensive algorithms, a data ingress pipeline and the strategy to cache and store data may need to be designed. Designing a production environment is discussed in more detail in `Chapter 13`, *Large Scale Algorithms*, and `Chapter 14`, *Practical Considerations*. Once the production environment is designed and implemented, the algorithm is deployed, which takes the input data, processes it, and generates the output as per the requirements.

Specifying the logic of an algorithm

When designing an algorithm, it is important to find different ways to specify its details. The ability to capture both its logic and architecture is required. Generally, just like building a home, it is important to specify the structure of an algorithm before actually implementing it. For more complex distributed algorithms, pre-planning the way their logic will be distributed across the cluster at running time is important for the iterative efficient design process. Through pseudocode and execution plans, both these needs are fulfilled and are discussed in the next section.

Understanding pseudocode

The simplest way to specify the logic for an algorithm is to write the higher-level description of an algorithm in a semi-structured way, called **pseudocode**. Before writing the logic in pseudocode, it is helpful to first describe its main flow by writing the main steps in plain English. Then, this English description is converted into pseudocode, which is a structured way of writing this English description that closely represents the logic and flow for the algorithm. Well-written algorithm pseudocode should describe the high-level steps of the algorithm in reasonable detail, even if the detailed code is not relevant to the main flow and structure of the algorithm. The following figure shows the flow of steps:



Note that once the pseudocode is written (as we will see in the next section), we are ready to code the algorithm using the programming language of our choice.

A practical example of pseudocode

Figure 1.3 shows the pseudocode of a resource allocation algorithm called **SRPMP**. In cluster computing, there are many situations where there are parallel tasks that need to be run on a set of available resources, collectively called a **resource pool**. This algorithm assigns tasks to a resource and creates a mapping set, called Ω. Note that the presented pseudocode captures the logic and flow of the algorithm, which is further explained in the following section:

```
BEGIN Mapping_Phase
$\Omega$ = { }
k = 1
FOREACH $T_i \in T$
    $\omega_i$ = RA($\Delta_k$, $T_i$)
    add {$\omega_i$, $T_i$} to $\Omega$
    $state\_change_{Ti}$ [STATE 0: Idle/Unmapped] $\rightarrow$ [STATE 1: Idle/Mapped]
    k=k+1
    IF (k>q)
       k=1
    ENDIF
```

```
12: END FOREACH
13: END Mapping_Phase
```

Let's parse this algorithm line by line:

1. We start the mapping by executing the algorithm. The Ω mapping set is empty.
2. The first partition is selected as the resource pool for the T_i task (see line 3 of the preceding code). **Television Rating Point** (**TRPS**) iteratively calls the **Rheumatoid Arthritis** (**RA**) algorithm for each T_i task with one of the partitions chosen as the resource pool.
3. The RA algorithm returns the set of resources chosen for the T_i task, represented by ω_i (see line 5 of the preceding code).
4. T_i and ω_i are added to the mapping set (see line 6 of the preceding code).
5. The state of T_i is changed from `STATE 0:Idle/Mapping` to `STATE 1:Idle/Mapped` (see line 7 of the preceding code).
6. Note that for the first iteration, `k=1` and the first partition is selected. For each subsequent iteration, the value of `k` is increased until `k>q`.
7. If `k` becomes greater than `q`, it is reset to `1` again (see lines 9 and 10 of the preceding code).
8. This process is repeated until a mapping between all tasks and the set of resources they will use is determined and stored in a mapping set called Ω.
9. Once each of the tasks is mapped to a set of the resources in the mapping phase, it is executed.

Using snippets

With the popularity of simple but powerful coding language such as Python, an alternative approach is becoming popular, which is to represent the logic of the algorithm directly in the programming language in a somewhat simplified version. Like pseudocode, this selected code captures the important logic and structure of the proposed algorithm, avoiding detailed code. This selected code is sometimes called a **snippet**. In this book, snippets are used instead of pseudocode wherever possible as they save one additional step. For example, let's look at a simple snippet that is about a Python function that can be used to swap two variables:

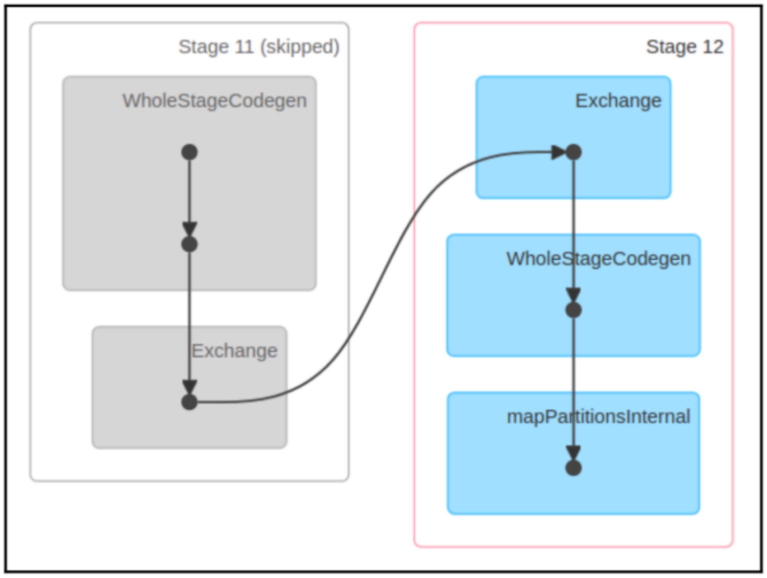
```
define swap(x, y)
    buffer = x
    x = y
    y = buffer
```

Note that snippets cannot always replace pseudocode. In pseudocode, sometimes we abstract many lines of code as one line of pseudocode, expressing the logic of the algorithm without becoming distracted by unnecessary coding details.

Creating an execution plan

Pseudocode and snippets are not always enough to specify all the logic related to more complex distributed algorithms. For example, distributed algorithms usually need to be divided into different coding phases at runtime that have a precedence order. The right strategy to divide the larger problem into an optimal number of phases with the right precedence constraints is crucial for the efficient execution of an algorithm.

We need to find a way to represent this strategy as well to completely represent the logic and structure of an algorithm. An execution plan is one of the ways of detailing how the algorithm will be subdivided into a bunch of tasks. A task can be mappers or reducers that can be grouped together in blocks called **stages**. The following diagram shows an execution plan that is generated by an Apache Spark runtime before executing an algorithm. It details the runtime tasks that the job created for executing our algorithm will be divided into:



Note that the preceding diagram has five tasks that have been divided into two different stages: **Stage 11** and **Stage 12**.

Introducing Python packages

Once designed, algorithms need to be implemented in a programming language as per the design. For this book, I chose the programming language Python. I chose it because Python is a flexible and open source programming language. Python is also the language of choice for increasingly important cloud computing infrastructures, such as **Amazon Web Services** (**AWS**), Microsoft Azure, and **Google Cloud Platform** (**GCP**).

The official Python home page is available at `https://www.python.org/`, which also has instructions for installation and a useful beginner's guide.

If you have not used Python before, it is a good idea to browse through this beginner's guide to self-study. A basic understanding of Python will help you to better understand the concepts presented in this book.

For this book, I expect you to use the recent version of Python 3. At the time of writing, the most recent version is 3.7.3, which is what we will use to run the exercises in this book.

Python packages

Python is a general-purpose language. It is designed in a way that comes with bare minimum functionality. Based on the use case that you intend to use Python for, additional packages need to be installed. The easiest way to install additional packages is through the pip installer program. This `pip` command can be used to install the additional packages:

```
pip install a_package
```

The packages that have already been installed need to be periodically updated to get the latest functionality. This is achieved by using the `upgrade` flag:

```
pip install a_package --upgrade
```

Another Python distribution for scientific computing is Anaconda, which can be downloaded from `http://continuum.io/downloads`.

In addition to using the `pip` command to install new packages, for Anaconda distribution, we also have the option of using the following command to install new packages:

```
conda install a_package
```

To update the existing packages, the Anaconda distribution gives us the option to use the following command:

```
conda update a_package
```

There are all sorts of Python packages that are available. Some of the important packages that are relevant for algorithms are described in the following section.

The SciPy ecosystem

Scientific Python (SciPy)—pronounced *sigh pie*—is a group of Python packages created for the scientific community. It contains many functions, including a wide range of random number generators, linear algebra routines, and optimizers. SciPy is a comprehensive package and, over time, people have developed many extensions to customize and extend the package according to their needs.

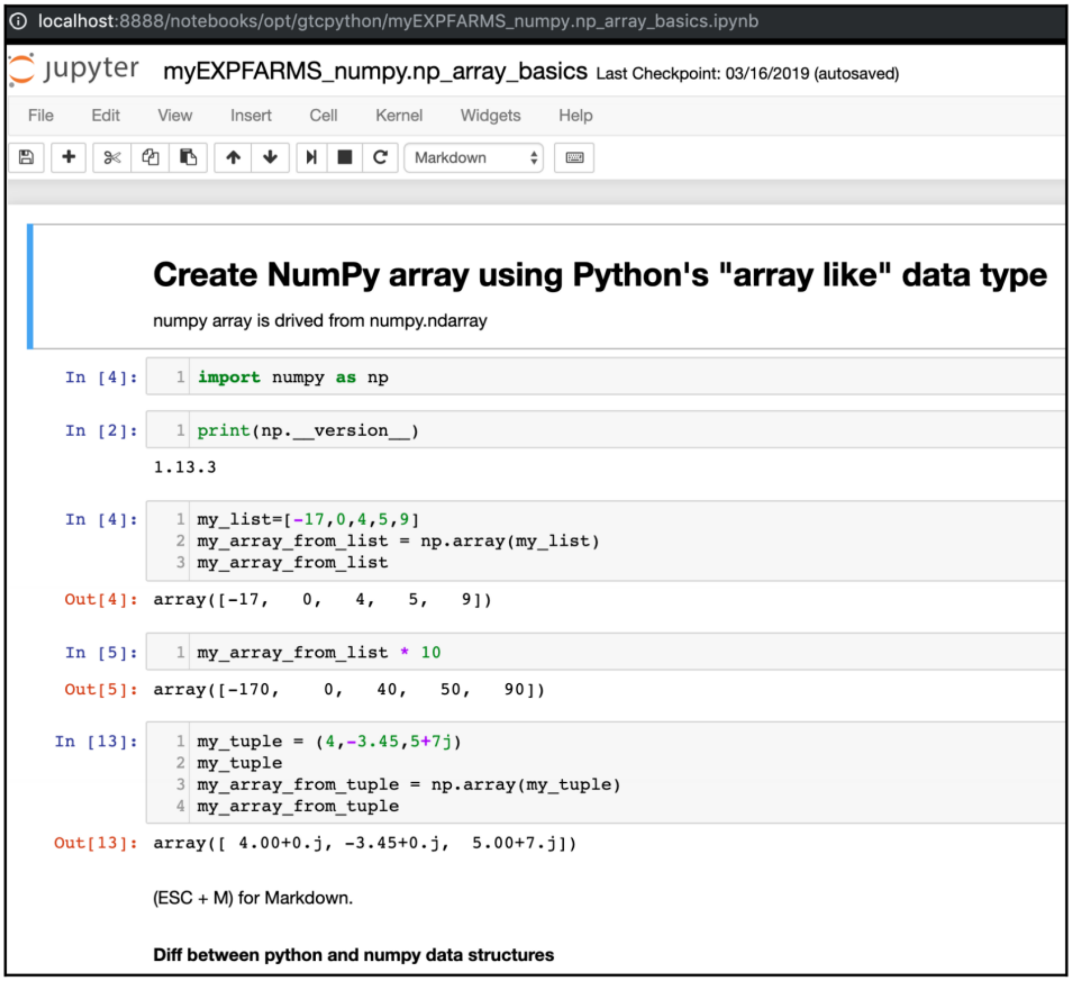
The following are the main packages that are part of this ecosystem:

- **NumPy**: For algorithms, the ability to create multi-dimensional data structures, such as arrays and matrices, is really important. NumPy offers a set of array and matrix data types that are important for statistics and data analysis. Details about NumPy can be found at `http://www.numpy.org/`.
- **scikit-learn**: This machine learning extension is one of the most popular extensions of SciPy. Scikit-learn provides a wide range of important machine learning algorithms, including classification, regression, clustering, and model validation. You can find more details about scikit-learn at `http://scikit-learn.org/`.
- **pandas**: pandas is an open source software library. It contains the tabular complex data structure that is used widely to input, output, and process tabular data in various algorithms. The pandas library contains many useful functions and it also offers highly optimized performance. More details about pandas can be found at `http://pandas.pydata.org/`.
- **Matplotlib**: Matplotlib provides tools to create powerful visualizations. Data can be presented as line plots, scatter plots, bar charts, histograms, pie charts, and so on. More information can be found at `https://matplotlib.org/`.
- **Seaborn**: Seaborn can be thought of as similar to the popular ggplot2 library in R. It is based on Matplotlib and offers an advanced interface for drawing brilliant statistical graphics. Further details can be found at `https://seaborn.pydata.org/`.
- **iPython**: iPython is an enhanced interactive console that is designed to facilitate the writing, testing, and debugging of Python code.
- **Running Python programs**: An interactive mode of programming is useful for learning and experimenting with code. Python programs can be saved in a text file with the `.py` extension and that file can be run from the console.

Implementing Python via the Jupyter Notebook

Another way to run Python programs is through the Jupyter Notebook. The Jupyter Notebook provides a browser-based user interface to develop code. The Jupyter Notebook is used to present the code examples in this book. The ability to annotate and describe the code with texts and graphics makes it the perfect tool for presenting and explaining an algorithm and a great tool for learning.

To start the notebook, you need to start the `Juypter-notebook` process and then open your favorite browser and navigate to `http://localhost:8888`:


localhost:8888/notebooks/opt/gtcpython/myEXPFARMS_numpy.np_array_basics.ipynb
jupyter
myEXPFARMS_numpy.np_array_basics Last Checkpoint: 03/16/2019 (autosaved)
File Edit View Insert Cell Kernel Widgets Help
Markdown

Create NumPy array using Python's "array like" data type

numpy array is drived from numpy.ndarray

```
In [4]:  import numpy as np

In [2]:  print(np.__version__)

         1.13.3

In [4]:  my_list=[-17,0,4,5,9]
         my_array_from_list = np.array(my_list)
         my_array_from_list

Out[4]:  array([-17,   0,   4,   5,   9])

In [5]:  my_array_from_list * 10

Out[5]:  array([-170,    0,   40,   50,   90])

In [13]: my_tuple = (4,-3.45,5+7j)
         my_tuple
         my_array_from_tuple = np.array(my_tuple)
         my_array_from_tuple

Out[13]: array([ 4.00+0.j, -3.45+0.j,  5.00+7.j])
```

(ESC + M) for Markdown.

Diff between python and numpy data structures

Note that a Jupyter Notebook consists of different blocks called **cells**.

Algorithm design techniques

An algorithm is a mathematical solution to a real-world problem. When designing an algorithm, we keep the following three design concerns in mind as we work on designing and fine-tuning the algorithms:

- **Concern 1**: Is this algorithm producing the result we expected?
- **Concern 2**: Is this the most optimal way to get these results?
- **Concern 3**: How is the algorithm going to perform on larger datasets?

It is important to better understand the complexity of the problem itself before designing a solution for it. For example, it helps us to design an appropriate solution if we characterize the problem in terms of its needs and complexity. Generally, the algorithms can be divided into the following types based on the characteristics of the problem:

- **Data-intensive algorithms:** Data-intensive algorithms are designed to deal with a large amount of data. They are expected to have relatively simplistic processing requirements. A compression algorithm applied to a huge file is a good example of data-intensive algorithms. For such algorithms, the size of the data is expected to be much larger than the memory of the processing engine (a single node or cluster) and an iterative processing design may need to be developed to efficiently process the data according to the requirements.
- **Compute-intensive algorithms**: Compute-intensive algorithms have considerable processing requirements but do not involve large amounts of data. A simple example is the algorithm to find a very large prime number. Finding a strategy to divide the algorithm into different phases so that at least some of the phases are parallelized is key to maximizing the performance of the algorithm.
- **Both data and compute-intensive algorithms**: There are certain algorithms that deal with a large amount of data and also have considerable computing requirements. Algorithms used to perform sentiment analysis on live video feeds are a good example of where both the data and the processing requirements are huge in accomplishing the task. Such algorithms are the most resource-intensive algorithms and require careful design of the algorithm and intelligent allocation of available resources.

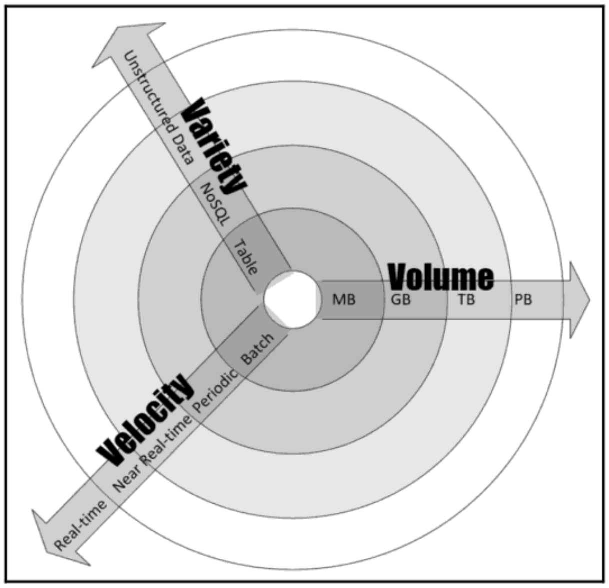
To characterize the problem in terms of its complexity and needs, it helps if we study its data and compute dimensions in more depth, which we will do in the following section.

The data dimension

To categorize the data dimension of the problem, we look at its **volume**, **velocity**, and **variety** (the **3Vs**), which are defined as follows:

- **Volume**: The volume is the expected size of the data that the algorithm will process.
- **Velocity**: The velocity is the expected rate of new data generation when the algorithm is used. It can be zero.
- **Variety**: The variety quantifies how many different types of data the designed algorithm is expected to deal with.

The following figure shows the 3Vs of the data in more detail. The center of this diagram shows the simplest possible data, with a small volume and low variety and velocity. As we move away from the center, the complexity of the data increases. It can increase in one or more of the three dimensions. For example, in the dimension of velocity, we have the **Batch** process as the simplest, followed by the **Periodic** process, and then the **Near Real-Time** process. Finally, we have the **Real-Time** process, which is the most complex to handle in the context of data velocity. For example, a collection of live video feeds gathered by a group of monitoring cameras will have a high volume, high velocity, and high variety and may need an appropriate design to have the ability to store and process data effectively. On the other hand, a simple `.csv` file created in Excel will have a low volume, low velocity, and low variety:



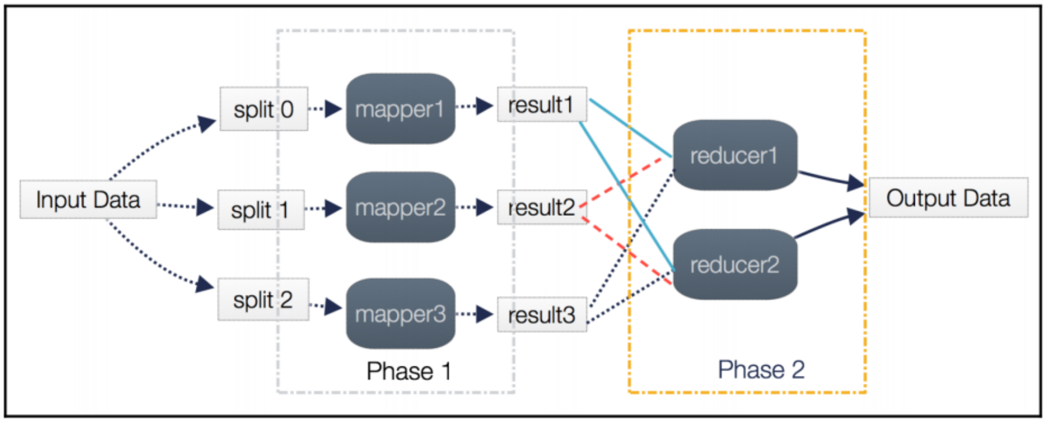
For example, if the input data is a simple `csv` file, then the volume, velocity, and variety of the data will be low. On the other hand, if the input data is the live stream of a security video camera, then the volume, velocity, and variety of the data will be quite high and this problem should be kept in mind while designing an algorithm for it.

Compute dimension

The compute dimension is about the processing and computing needs of the problem at hand. The processing requirements of an algorithm will determine what sort of design is most efficient for it. For example, deep learning algorithms, in general, require lots of processing power. It means that for deep learning algorithms, it is important to have multi-node parallel architecture wherever possible.

A practical example

Let's assume that we want to conduct sentiment analysis on a video. Sentiment analysis is where we try to flag different portions of a video with human emotions of sadness, happiness, fear, joy, frustration, and ecstasy. It is a compute-intensive job where lots of computing power is needed. As you will see in the following figure, to design the compute dimension, we have divided the processing into five tasks, consisting of two stages. All the data transformation and preparation is implemented in three mappers. For that, we divide the video into three different partitions, called **splits**. After the mappers are executed, the resulting processed video is inputted to the two aggregators, called **reducers**. To conduct the required sentiment analysis, the reducers group the video according to the emotions. Finally, the results are combined in the output:



Note that the number of mappers directly translates to the runtime parallelism of the algorithm. The optimal number of mappers and reducers is dependent on the characteristics of the data, the type of algorithm that is needed to be used, and the number of resources available.

Performance analysis

Analyzing the performance of an algorithm is an important part of its design. One of the ways to estimate the performance of an algorithm is to analyze its complexity.

Complexity theory is the study of how complicated algorithms are. To be useful, any algorithm should have three key features:

- It should be correct. An algorithm won't do you much good if it doesn't give you the right answers.
- A good algorithm should be understandable. The best algorithm in the world won't do you any good if it's too complicated for you to implement on a computer.
- A good algorithm should be efficient. Even if an algorithm produces a correct result, it won't help you much if it takes a thousand years or if it requires 1 billion terabytes of memory.

There are two possible types of analysis to quantify the complexity of an algorithm:

- Space complexity analysis: Estimates the runtime memory requirements needed to execute the algorithm.
- Time complexity analysis: Estimates the time the algorithm will take to run.

Space complexity analysis

Space complexity analysis estimates the amount of memory required by the algorithm to process input data. While processing the input data, the algorithm needs to store the transient temporary data structures in memory. The way the algorithm is designed affects the number, type, and size of these data structures. In an age of distributed computing and with increasingly large amounts of data that needs to be processed, space complexity analysis is becoming more and more important. The size, type, and number of these data structures will dictate the memory requirements for the underlying hardware. Modern in-memory data structures used in distributed computing—such as **Resilient Distributed Datasets** (**RDDs**)—need to have efficient resource allocation mechanisms that are aware of the memory requirements at different execution phases of the algorithm.

Space complexity analysis is a must for the efficient design of algorithms. If proper space complexity analysis is not conducted while designing a particular algorithm, insufficient memory availability for the transient temporary data structures may trigger unnecessary disk spillovers, which could potentially considerably affect the performance and efficiency of the algorithm.

In this chapter, we will look deeper into time complexity. Space complexity will be discussed in `Chapter 13`, *Large-Scale Algorithms*, in more detail, where we will deal with large-scale distributed algorithms with complex runtime memory requirements.

Time complexity analysis

Time complexity analysis estimates how long it will take for an algorithm to complete its assigned job based on its structure. In contrast to space complexity, time complexity is not dependent on any hardware that the algorithm will run on. Time complexity analysis solely depends on the structure of the algorithm itself. The overall goal of time complexity analysis is to try to answer these important questions—will this algorithm scale? How well will this algorithm handle larger datasets?

To answer these questions, we need to determine the effect on the performance of an algorithm as the size of the data is increased and make sure that the algorithm is designed in a way that not only makes it accurate but also scales well. The performance of an algorithm is becoming more and more important for larger datasets in today's world of "big data."

In many cases, we may have more than one approach available to design the algorithm. The goal of conducting time complexity analysis, in this case, will be as follows:

"Given a certain problem and more than one algorithm, which one is the most efficient to use in terms of time efficiency?"

There can be two basic approaches to calculating the time complexity of an algorithm:

- **A post-implementation profiling approach**: In this approach, different candidate algorithms are implemented and their performance is compared.
- **A pre-implementation theoretical approach**: In this approach, the performance of each algorithm is approximated mathematically before running an algorithm.

The advantage of the theoretical approach is that it only depends on the structure of the algorithm itself. It does not depend on the actual hardware that will be used to run the algorithm, the choice of the software stack chosen at runtime, or the programming language used to implement the algorithm.

Estimating the performance

The performance of a typical algorithm will depend on the type of the data given to it as an input. For example, if the data is already sorted according to the context of the problem we are trying to solve, the algorithm may perform blazingly fast. If the sorted input is used to benchmark this particular algorithm, then it will give an unrealistically good performance number, which will not be a true reflection of its real performance in most scenarios. To handle this dependency of algorithms on the input data, we have different types of cases to consider when conducting a performance analysis.

The best case

In the best case, the data given as input is organized in a way that the algorithm will give its best performance. Best-case analysis gives the upper bound of the performance.

The worst case

The second way to estimate the performance of an algorithm is to try to find the maximum possible time it will take to get the job done under a given set of conditions. This worst-case analysis of an algorithm is quite useful as we are guaranteeing that regardless of the conditions, the performance of the algorithm will always be better than the numbers that come out of our analysis. Worst-case analysis is especially useful for estimating the performance when dealing with complex problems with larger datasets. Worst-case analysis gives the lower bound of the performance of the algorithm.

The average case

This starts by dividing the various possible inputs into various groups. Then, it conducts the performance analysis from one of the representative inputs from each group. Finally, it calculates the average of the performance of each of the groups.

Average-case analysis is not always accurate as it needs to consider all the different combinations and possibilities of input to the algorithm, which is not always easy to do.

Selecting an algorithm

How do you know which one is a better solution? How do you know which algorithm runs faster? Time complexity and Big O notation (discussed later in this chapter) are really good tools for answering these types of questions.

To see where it can be useful, let's take a simple example where the objective is to sort a list of numbers. There are a couple of algorithms available that can do the job. The issue is how to choose the right one.

First, an observation that can be made is that if there are not too many numbers in the list, then it does not matter which algorithm do we choose to sort the list of numbers. So, if there are only 10 numbers in the list (n=10), then it does not matter which algorithm we choose as it would probably not take more than a few microseconds, even with a very badly designed algorithm. But as soon as the size of the list becomes 1 million, now the choice of the right algorithm will make a difference. A very badly written algorithm might even take a couple of hours to run, while a well-designed algorithm may finish sorting the list in a couple of seconds. So, for larger input datasets, it makes a lot of sense to invest time and effort, perform a performance analysis, and choose the correctly designed algorithm that will do the job required in an efficient manner.

Big O notation

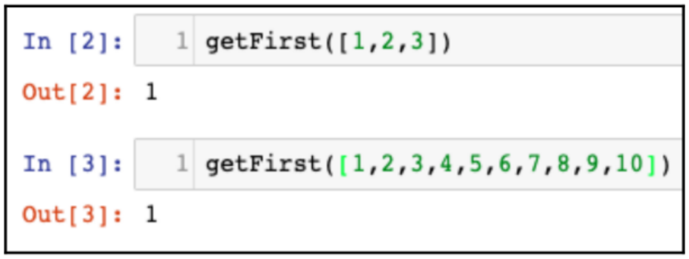
Big O notation is used to quantify the performance of various algorithms as the input size grows. Big O notation is one of the most popular methodologies used to conduct worst-case analysis. The different kinds of Big O notation types are discussed in this section.

Constant time (O(1)) complexity

If an algorithm takes the same amount of time to run, independent of the size of the input data, it is said to run in constant time. It is represented by O(1). Let's take the example of accessing the n^{th} element of an array. Regardless of the size of the array, it will take constant time to get the results. For example, the following function will return the first element of the array and has a complexity of O(1):

```
def getFirst(myList):
    return myList[0]
```

The output is shown as:



```
In [2]:  getFirst([1,2,3])
Out[2]: 1

In [3]:  getFirst([1,2,3,4,5,6,7,8,9,10])
Out[3]: 1
```

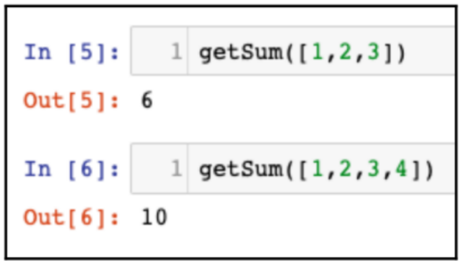
- Addition of a new element to a stack by using `push` or removing an element from a stack by using `pop`. Regardless of the size of the stack, it will take the same time to add or remove an element.
- Accessing the element of the hashtable (as discussed in `Chapter 2`, *Data Structures Used in Algorithms*).
- Bucket sort (as discussed in `Chapter 2`, *Data Structures Used in Algorithms*).

Linear time (O(n)) complexity

An algorithm is said to have a complexity of linear time, represented by O(n), if the execution time is directly proportional to the size of the input. A simple example is to add the elements in a single-dimensional data structure:

```
def getSum(myList):
    sum = 0
    for item in myList:
        sum = sum + item
    return sum
```

Note the main loop of the algorithm. The number of iterations in the main loop increases linearly with an increasing value of *n*, producing an O(n) complexity in the following figure:



```
In [5]:  getSum([1,2,3])
Out[5]:  6

In [6]:  getSum([1,2,3,4])
Out[6]:  10
```

Some other examples of array operations are as follows:

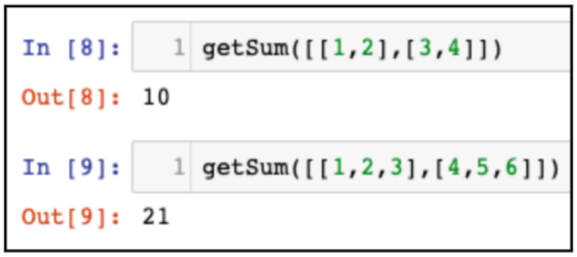
- Searching an element
- Finding the minimum value among all the elements of an array

Quadratic time (O(n^2)) complexity

An algorithm is said to run in quadratic time if the execution time of an algorithm is proportional to the square of the input size; for example, a simple function that sums up a two-dimensional array, as follows:

```
def getSum(myList):
    sum = 0
    for row in myList:
        for item in row:
            sum += item
    return sum
```

Note the nested inner loop within the other main loop. This nested loop gives the preceding code the complexity of O(n^2):



```
In [8]:  getSum([[1,2],[3,4]])
Out[8]: 10

In [9]:  getSum([[1,2,3],[4,5,6]])
Out[9]: 21
```

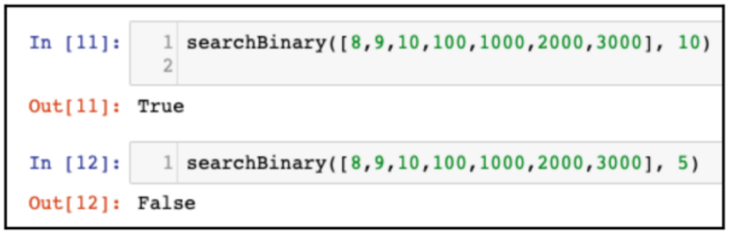
Another example is the **bubble sort algorithm** (as discussed in `Chapter 2`, *Data Structures Used in Algorithms*).

Logarithmic time (O(logn)) complexity

An algorithm is said to run in logarithmic time if the execution time of the algorithm is proportional to the logarithm of the input size. With each iteration, the input size decreases by a constant multiple factor. An example of logarithmic is binary search. The binary search algorithm is used to find a particular element in a one-dimensional data structure, such as a Python list. The elements within the data structure need to be sorted in descending order. The binary search algorithm is implemented in a function named `searchBinary`, as follows:

```
def searchBinary(myList,item):
    first = 0
    last = len(myList)-1
    foundFlag = False
    while( first<=last and not foundFlag):
        mid = (first + last)//2
        if myList[mid] == item :
            foundFlag = True
        else:
            if item < myList[mid]:
                last = mid - 1
            else:
                first = mid + 1
    return foundFlag
```

The main loop takes advantage of the fact that the list is ordered. It divides the list in half with each iteration until it gets to the result:



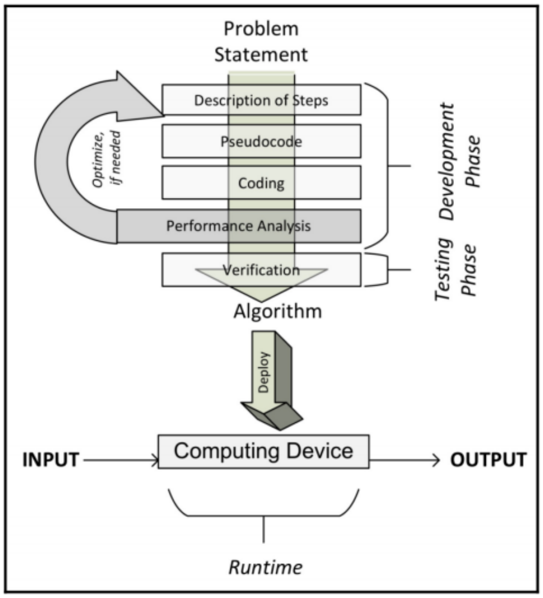
```
In [11]:  searchBinary([8,9,10,100,1000,2000,3000], 10)

Out[11]: True

In [12]:  searchBinary([8,9,10,100,1000,2000,3000], 5)
Out[12]: False
```

After defining the function, it is tested to search a particular element in lines 11 and 12. The binary search algorithm is further discussed in `Chapter 3`, *Sorting and Searching Algorithms*.

Note that among the four types of Big O notation types presented, O(n^2) has the worst performance and O(logn) has the best performance. In fact, O(logn)'s performance can be thought of as the gold standard for the performance of any algorithm (which is not always achieved, though). On the other hand, O(n^2) is not as bad as O(n^3) but still, algorithms that fall in this class cannot be used on big data as the time complexity puts limitations on how much data they can realistically process.

One way to reduce the complexity of an algorithm is to compromise on its accuracy, producing a type of algorithm called an **approximate algorithm**.

The whole process of the performance evaluation of algorithms is iterative in nature, as shown in the following figure:



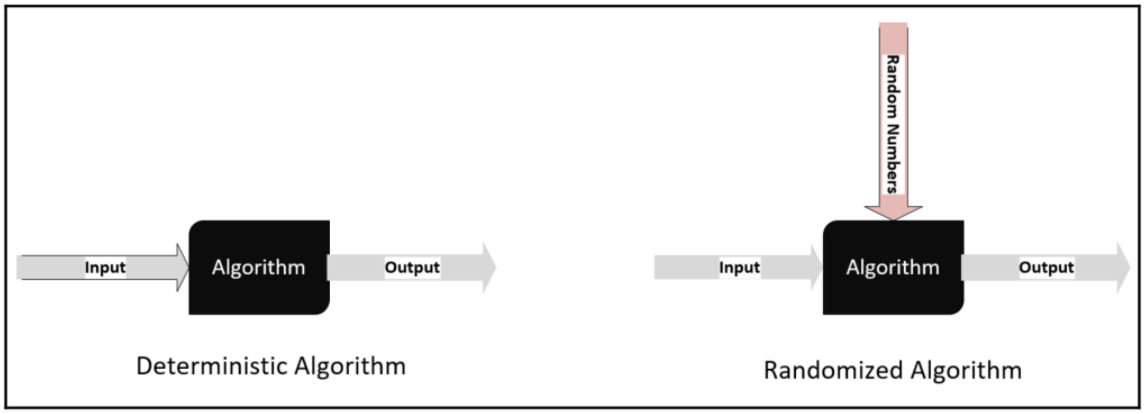
Validating an algorithm

Validating an algorithm confirms that it is actually providing a mathematical solution to the problem we are trying to solve. A validation process should check the results for as many possible values and types of input values as possible.

Exact, approximate, and randomized algorithms

Validating an algorithm also depends on the type of the algorithm as the testing techniques are different. Let's first differentiate between deterministic and randomized algorithms.

For deterministic algorithms, a particular input always generates exactly the same output. But for certain classes of algorithms, a sequence of random numbers is also taken as input, which makes the output different each time the algorithm is run. The k-means clustering algorithm, which is detailed in `Chapter 6`, *Unsupervised Machine Learning Algorithms*, is an example of such an algorithm:



Algorithms can also be divided into the following two types based on assumptions or approximation used to simplify the logic to make them run faster:

- **An exact algorithm:** Exact algorithms are expected to produce a precise solution without introducing any assumptions or approximations.
- **An approximate algorithm:** When the problem complexity is too much to handle for the given resources, we simplify our problem by making some assumptions. The algorithms based on these simplifications or assumptions are called approximate algorithms, which doesn't quite give us the precise solution.

Let's look at an example to understand the difference between the exact and approximate algorithms—the famous traveling salesman problem, which was presented in 1930. A traveling salesman challenges you to find the shortest route for a particular salesman that visits each city (from a list of cities) and then returns to the origin, which is why he is named the traveling salesman. The first attempt to provide the solution will include generating all the permutations of cities and choosing the combination of cities that is cheapest. The complexity of this approach to provide the solution is O(n!), where *n* is the number of cities. It is obvious that time complexity starts to become unmanageable beyond 30 cities.

If the number of cities is more than 30, one way of reducing the complexity is to introduce some approximations and assumptions.

For approximate algorithms, it is important to set the expectations for accuracy when gathering the requirements. Validating an approximation algorithm is about verifying that the error of the results is within an acceptable range.

Explainability

When algorithms are used for critical cases, it becomes important to have the ability to explain the reason behind each and every result whenever needed. This is necessary to make sure that decisions based on the results of the algorithms do not introduce bias.

The ability to exactly identify the features that are used directly or indirectly to come up with a particular decision is called the **explainability** of an algorithm. Algorithms, when used for critical use cases, need to be evaluated for bias and prejudice. The ethical analysis of algorithms has become a standard part of the validation process for those algorithms that can affect decision-making that relates to the life of people.

For algorithms that deal with deep learning, explainability is difficult to achieve. For example, if an algorithm is used to refuse the mortgage application of a person, it is important to have the transparency and ability to explain the reason.

Algorithmic explainability is an active area of research. One of the effective techniques that has been recently developed is **Local Interpretable Model-Agnostic Explanations** (**LIME**), as proposed in the proceedings of the 22nd **Association for Computing Machinery** (**ACM**) at the **Special Interest Group on Knowledge Discovery** (**SIGKDD**) international conference on knowledge discovery and data mining in 2016. LIME is based on a concept where small changes are induced to the input for each instance and then an effort to map the local decision boundary for that instance is made. It can then quantify the influence of each variable for that instance.

Summary

This chapter was about learning the basics of algorithms. First, we learned about the different phases of developing an algorithm. We discussed the different ways of specifying the logic of an algorithm that are necessary for designing it. Then, we looked at how to design an algorithm. We learned two different ways of analyzing the performance of an algorithm. Finally, we studied different aspects of validating an algorithm.

After going through this chapter, we should be able to understand the pseudocode of an algorithm. We should understand the different phases in developing and deploying an algorithm. We also learned how to use Big O notation to evaluate the performance of an algorithm.

The next chapter is about the data structures used in algorithms. We will start by looking at the data structures available in Python. We will then look at how we can use these data structures to create more sophisticated data structures, such as stacks, queues, and trees, which are needed to develop complex algorithms.

2
Data Structures Used in Algorithms

Algorithms need necessary in-memory data structures that can hold temporary data while executing. Choosing the right data structures is essential for their efficient implementation. Certain classes of algorithms are recursive or iterative in logic and need data structures that are specially designed for them. For example, a recursive algorithm may be more easily implemented, exhibiting better performance, if nested data structures are used. In this chapter, data structures are discussed in the context of algorithms. As we are using Python in this book, this chapter focuses on Python data structures, but the concepts presented in this chapter can be used in other languages such as Java and C++.

By the end of this chapter, you should be able to understand how Python handles complex data structures and which one should be used for a certain type of data.

Hence, here are the main points discussed in this chapter:

- Exploring data structures in Python
- Exploring abstract data type
- Stacks and queues
- Trees

Exploring data structures in Python

In any language, data structures are used to store and manipulate complex data. In Python, data structures are storage containers to manage, organize, and search data in an efficient way. They are used to store a group of data elements called *collections* that need to be stored and processed together. In Python, there are five various data structures that can be used to store collections:

- **Lists**: Ordered mutable sequences of elements
- **Tuples**: Ordered immutable sequences of elements
- **Sets**: Unordered bags of elements
- **Dictionary**: Unordered bags of key-value pairs
- **Data frames**: Two-dimensional structures to store two-dimensional data

Let's look into them in more detail in the upcoming subsections.

List

In Python, a list is the main data structure used to store a mutable sequence of elements. The sequence of data elements stored in the list need not be of the same type.

To create a list, the data elements need to be enclosed in [ ] and they need to be separated by a comma. For example, the following code creates four data elements together that are of different types:

```
>>> aList = ["John", 33,"Toronto", True]
>>> print(aList)
['John', 33, 'Toronto', True]Ex
```

In Python, a list is a handy way of creating one-dimensional writable data structures that are needed especially at different internal stages of algorithms.

Using lists

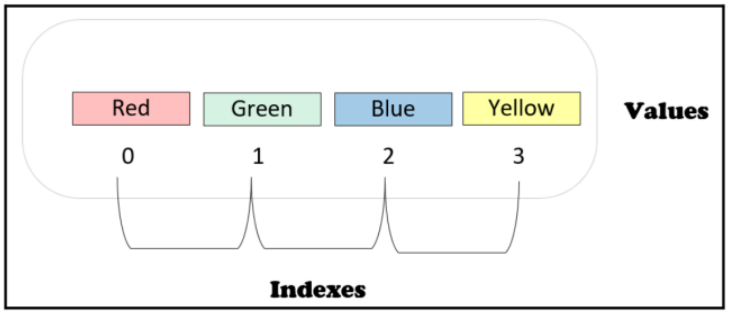
Utility functions in data structures make them very useful as they can be used to manage data in lists.

Let's look into how we can use them:

- **List indexing**: As the position of an element is deterministic in a list, the index can be used to get an element at a particular position. The following code demonstrates the concept:

```
>>> bin_colors=['Red','Green','Blue','Yellow']
>>> bin_colors[1]
'Green'
```

  The four-element list created by this code is shown in the following screenshot:



  Note that the index starts from 0 and therefore **Green**, which is the second element, is retrieved by index **1**, that is, `bin_color[1]`.

- **List slicing**: Retrieving a subset of the elements of a list by specifying a range of indexes is called **slicing**. The following code can be used to create a slice of the list:

```
>>> bin_colors=['Red','Green','Blue','Yellow']
>>> bin_colors[0:2]
['Red', 'Green']
```

Note that lists are one of the most popular single-dimensional data structures in Python.

While slicing a list, the range is indicated as follows: the first number (inclusive) and the second number (exclusive). For example, `bin_colors[0:2]` will include `bin_color[0]` and `bin_color[1]` but not `bin_color[2]`. While using lists, this should be kept in mind as some users of the Python language complain that this is not very intuitive.

Let's have a look at the following code snippet:

```
>>> bin_colors=['Red','Green','Blue','Yellow']
>>> bin_colors[2:]
['Blue', 'Yellow']
>>> bin_colors[:2]
['Red', 'Green']
```

If the starting index is not specified, it means the beginning of the list, and if the ending index is not specified, it means the end of the list. The preceding code actually demonstrates this concept.

- **Negative indexing**: In Python, we also have negative indices, which count from the end of the list. This is demonstrated in the following code:

```
>>> bin_colors=['Red','Green','Blue','Yellow']
>>> bin_colors[:-1]
['Red', 'Green', 'Blue']
>>> bin_colors[:-2]
['Red', 'Green']
>>> bin_colors[-2:-1]
['Blue']
```

Note that negative indices are especially useful when we want to use the last element as a reference point instead of the first one.

- **Nesting**: An element of a list can be of a simple data type or a complex data type. This allows nesting in lists. For iterative and recursive algorithms, this provides important capabilities.

Let's have a look at the following code, which is an example of a list within a list (nesting):

```
>>> a = [1,2,[100,200,300],6]
>>> max(a[2])
300
>>> a[2][1]
200
```

- **Iteration**: Python allows iterating over each element on a list by using a `for` loop. This is demonstrated in the following example:

```
>>> bin_colors=['Red','Green','Blue','Yellow']
>>> for aColor in bin_colors:
        print(aColor + " Square")
Red Square
Green Square
```

```
Blue Square
Yellow Square
```

Note that the preceding code iterates through the list and prints each element.

Lambda functions

There are a bunch of lambda functions that can be used on lists. They are specifically important in the context of algorithms and provide the ability to create a function on the fly. Sometimes, in the literature, they are also called *anonymous functions*. This section demonstrates their uses:

- **Filtering data**: To filter the data, first, we define a predicate, which is a function that inputs a single argument and returns a Boolean value. The following code demonstrates its use:

```
>>> list(filter(lambda x: x > 100, [-5, 200, 300, -10, 10, 1000]))
[200, 300, 1000]
```

Note that, in this code, we filter a list using the `lambda` function, which specifies the filtering criteria. The filter function is designed to filter elements out of a sequence based on a defined criterion. The filter function in Python is usually used with `lambda`. In addition to lists, it can be used to filter elements from tuples or sets. For the preceding code, the defined criterion is `x > 100`. The code will iterate through all the elements of the list and will filter out the elements that do not pass this criterion.

- **Data transformation**: The `map()` function can be used for data transformation using a lambda function. An example is as follows:

```
>>> list(map(lambda x: x ** 2, [11, 22, 33, 44,55]))
[121, 484, 1089, 1936, 3025]
```

Using the `map` function with a `lambda` function provides quite powerful functionality. When used with the `map` function, the `lambda` function can be used to specify a transformer that transforms each element of the given sequence. In the preceding code, the transformer is multiplication by two. So, we are using the `map` function to multiply each element in the list by two.

- **Data aggregation**: For data aggregation, the `reduce()` function can be used, which recursively runs a function to pairs of values on each element of the list:

  ```
  from functools import reduce
  def doSum(x1,x2):
      return x1+x2
  x = reduce(doSum, [100, 122, 33, 4, 5, 6])
  ```

  Note that the `reduce` function needs a data aggregation function to be defined. That data aggregation function in the preceding code is `functools`. It defines how it will aggregate the items of the given list. The aggregation will start from the first two elements and the result will replace the first two elements. This process of reduction is repeated until we reach the end, resulting in one aggregated number. `x1` and `x2` in the `doSum` function represent two numbers in each of these iterations and `doSum` represents the aggregation criterion for them.

  The preceding code block results in a single value (which is `270`).

The range function

The `range` function can be used to easily generate a large list of numbers. It is used to auto-populate sequences of numbers in a list.

The `range` function is simple to use. We can use it by just specifying the number of elements we want in the list. By default, it starts from zero and increments by one:

```
>>> x = range(6)
>>> x
[0,1,2,3,4,5]
```

We can also specify the end number and the step:

```
>>> oddNum = range(3,29,2)
>>> oddNum
[3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27]
```

The preceding range function will give us odd numbers starting from `3` to `29`.

The time complexity of lists

The time complexity of various functions of a list can be summarized as follows using the Big O notation:

Different methods	**Time complexity**
Insert an element	O(1)
Delete an element	O(n) (as in the worst case may have to iterate the whole list)
Slicing a list	O(n)
Element retrieval	O(n)
Copy	O(n)

Please note that the time taken to add an individual element is independent of the size of the list. Other operations mentioned in the table are dependent on the size of the list. As the size of the list gets bigger, the impact on performance becomes more pronounced.

Tuples

The second data structure that can be used to store a collection is a tuple. In contrast to lists, tuples are immutable (read-only) data structures. Tuples consist of several elements surrounded by ( ).

Like lists, elements within a tuple can be of different types. They also allow complex data types for their elements. So, there can be a tuple within a tuple providing a way to create a nested data structure. The capability to create nested data structures is especially useful in iterative and recursive algorithms.

The following code demonstrates how to create tuples:

```
>>> bin_colors=('Red','Green','Blue','Yellow')
>>> bin_colors[1]
'Green'
>>> bin_colors[2:]
('Blue', 'Yellow')
>>> bin_colors[:-1]
('Red', 'Green', 'Blue')
# Nested Tuple Data structure
>>> a = (1,2,(100,200,300),6)
>>> max(a[2])
300
>>> a[2][1]
200
```

TIP

Wherever possible, immutable data structures (such as tuples) should be preferred over mutable data structures (such as lists) due to performance. Especially when dealing with big data, immutable data structures are considerably faster than mutable ones. There is a price we pay for the ability to change data elements in lists, for example, and we should carefully analyze that it is really needed so we can implement the code as read-only tuples, which will be much faster.

Note that, in the preceding code, `a[2]` refers to the third element, which is a tuple, `(100,200,300)`. `a[2][1]` refers to the second element within this tuple, which is `200`.

The time complexity of tuples

The time complexity of various functions of tuples can be summarized as follows (using Big O notation):

Function	**Time Complexity**
`Append`	O(1)

Note that `Append` is a function that adds an element toward the end of the already existing tuple. Its complexity is O(1).

Dictionary

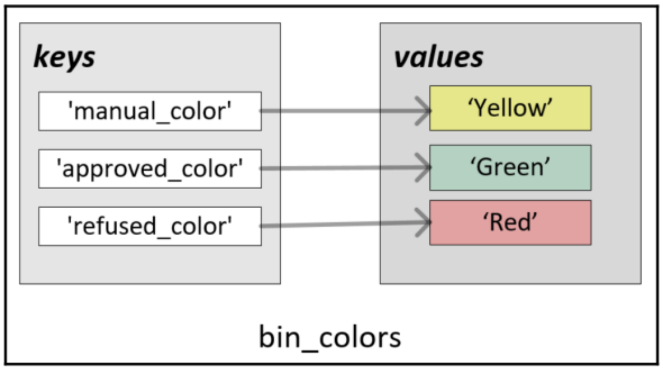
Holding data as key-value pairs is important especially in distributed algorithms. In Python, a collection of these key-value pairs is stored as a data structure called a *dictionary*. To create a dictionary, a key should be chosen as an attribute that is best suited to identify data throughout data processing. The value can be an element of any type, for example, a number or string. Python also always uses complex data types such as lists as values. Nested dictionaries can be created by using a dictionary as the data type of a value.

To create a simple dictionary that assigns colors to various variables, the key-value pairs need to be enclosed in { }. For example, the following code creates a simple dictionary consisting of three key-value pairs:

```
>>> bin_colors ={
      "manual_color": "Yellow",
      "approved_color": "Green",
      "refused_color": "Red"
    }
```

```
>>> print(bin_colors)
{'manual_color': 'Yellow', 'approved_color': 'Green', 'refused_color':
'Red'}
```

The three key-value pairs created by the preceding piece of code are also illustrated in the following screenshot:



Now, let's see how to retrieve and update a value associated with a key:

1. To retrieve a value associated with a key, either the `get` function can be used or the key can be used as the index:

   ```
   >>> bin_colors.get('approved_color')
   'Green'
   >>> bin_colors['approved_color']
   'Green'
   ```

2. To update a value associated with a key, use the following code:

   ```
   >>> bin_colors['approved_color']="Purple"
   >>> print(bin_colors)
   {'manual_color': 'Yellow', 'approved_color': 'Purple',
   'refused_color': 'Red'}
   ```

   Note that the preceding code shows how we can update a value related to a particular key in a dictionary.

The time complexity of a dictionary

The following table gives the time complexity of a dictionary using Big O notation:

Dictionary	**Time complexity**
Get a value or a key	O(1)
Set a value or a key	O(1)
Copy a dictionary	O(n)

An important thing to note from the complexity analysis of the dictionary is that the time taken to get or set a key-value is totally independent of the size of the dictionary. This means that the time taken to add a key-value pair to a dictionary of a size of three is the same as the time taken to add a key-value pair to a dictionary of a size of one million.

Sets

A set is defined as a collection of elements that can be of different types. The elements are enclosed within { }. For example, have a look at the following code block:

```
>>> green = {'grass', 'leaves'}
>>> print(green)
{'grass', 'leaves'}
```

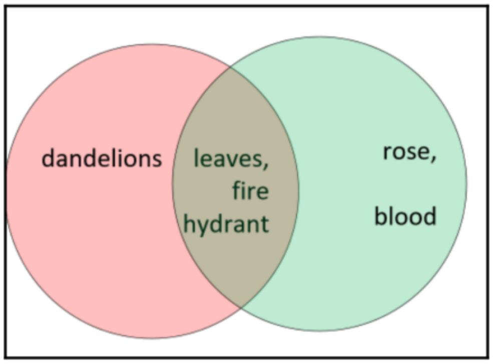
The defining characteristic of a set is that it only stores the distinct value of each element. If we try to add another redundant element, it will ignore that, as illustrated in the following:

```
>>> green = {'grass', 'leaves','leaves'}
>>> print(green)
{'grass', 'leaves'}
```

To demonstrate what sort of operations can be done on sets, let's define two sets:

- A set named yellow, which has things that are yellow
- Another set named red, which has things that are red

Note that some things are common between these two sets. The two sets and their relationship can be represented with the help of the following Venn diagram:



If we want to implement these two sets in Python, the code will look like this:

```
>>> yellow = {'dandelions', 'fire hydrant', 'leaves'}
>>> red = {'fire hydrant', 'blood', 'rose', 'leaves'}
```

Now, let's consider the following code, which demonstrates set operations using Python:

```
>>> yellow|red
{'dandelions', 'fire hydrant', 'blood', 'rose', 'leaves'}
>>> yellow&red
{'fire hydrant'}
```

As shown in the preceding code snippet, sets in Python can have operations such as unions and intersections. As we know, a union operation combines all of the elements of both sets, and the intersection operation will give a set of common elements between the two sets. Note the following:

- `yellow|red` is used to get the union of the preceding two defined sets.
- `yellow&red` is used to get the overlap between yellow and red.

Time complexity analysis for sets

Following is the time complexity analysis for sets:

Sets	**Complexity**
Add an element	O(1)
Remove an element	O(1)
Copy	O(n)

An important thing to note from the complexity analysis of the sets is that the time taken to add an element is totally independent of the size of a particular set.

DataFrames

A DataFrame is a data structure used to store tabular data available in Python's `pandas` package. It is one of the most important data structures for algorithms and is used to process traditional structured data. Let's consider the following table:

id	name	age	decision
1	Fares	32	True
2	Elena	23	False
3	Steven	40	True

Now, let's represent this using a DataFrame.

A simple DataFrame can be created by using the following code:

```
>>> import pandas as pd
>>> df = pd.DataFrame([
...             ['1', 'Fares', 32, True],
...             ['2', 'Elena', 23, False],
...             ['3', 'Steven', 40, True]])
>>> df.columns = ['id', 'name', 'age', 'decision']
>>> df
  id    name  age  decision
0  1   Fares   32      True
1  2   Elena   23     False
2  3  Steven   40      True
```

Note that, in the preceding code, `df.column` is a list that specifies the names of the columns.

The DataFrame is also used in other popular languages and frameworks to implement a tabular data structure. Examples are R and the Apache Spark framework.

Terminologies of DataFrames

Let's look into some of the terminologies that are used in the context of a DataFrame:

- **Axis**: In the pandas documentation, a single column or row of a DataFrame is called an axis.

- **Axes**: If there is more than one axis, they are called axes as a group.
- **Label**: A DataFrame allows the naming of both columns and rows with what's called a label.

Creating a subset of a DataFrame

Fundamentally, there are two main ways of creating the subset of a DataFrame (say the name of the subset is `myDF`):

- Column selection
- Row selection

Let's see them one by one.

Column selection

In machine learning algorithms, selecting the right set of features is an important task. Out of all of the features that we may have, not all of them may be needed at a particular stage of the algorithm. In Python, feature selection is achieved by column selection, which is explained in this section.

A column may be retrieved by *name*, as in the following:

```
>>> df[['name','age']]
     name  age
0   Fares   32
1   Elena   23
2  Steven   40
```

The positioning of a column is deterministic in a DataFrame. A column can be retrieved by its position as follows:

```
>>> df.iloc[:,3]
0 True
1 False
2 True
```

Note that, in this code, we are retrieving the first three rows of the DataFrame.

Row selection

Each row in a DataFrame corresponds to a data point in our problem space. We need to perform row selection if we want to create a subset of the data elements we have in our problem space. This subset can be created by using one of the two following methods:

- By specifying their position
- By specifying a filter

A subset of rows can be retrieved by its position as follows:

```
>>> df.iloc[1:3,:]
  id name age decision
1 2 Elena 23 False
2 3 Steven 40 True
```

Note that the preceding code will return the first two rows and all columns.

To create a subset by specifying the filter, we need to use one or more columns to define the selection criterion. For example, a subset of data elements can be selected by this method, as follows:

```
>>> df[df.age>30]
  id    name  age  decision
0  1   Fares   32      True
2  3  Steven   40      True

>>> df[(df.age<35)&(df.decision==True)]
  id   name  age  decision
0  1  Fares   32      True
```

Note that this code creates a subset of rows that satisfies the condition stipulated in the filter.

Matrix

A matrix is a two-dimensional data structure with a fixed number of columns and rows. Each element of a matrix can be referred to by its column and the row.

In Python, a matrix can be created by using the `numpy` array, as shown in the following code:

```
>>> myMatrix = np.array([[11, 12, 13], [21, 22, 23], [31, 32, 33]])
>>> print(myMatrix)
[[11 12 13]
[21 22 23]
[31 32 33]]
>>> print(type(myMatrix))
<class 'numpy.ndarray'>
```

Note that the preceding code will create a matrix that has three rows and three columns.

Matrix operations

There are many operations available for matrix data manipulation. For example, let's try to transpose the preceding matrix. We will use the `transpose()` function, which will convert columns into rows and rows into columns:

```
>>> myMatrix.transpose()
array([[11, 21, 31],
       [12, 22, 32],
       [13, 23, 33]])
```

Note that matrix operations are used a lot in multimedia data manipulation.

Now that we have learned about data structures in Python, let's move onto the abstract data types in the next section.

Exploring abstract data types

Abstraction, in general, is a concept used to define complex systems in terms of their common core functions. The use of this concept to create generic data structures gives birth to **Abstract Data Types** (**ADT**). By hiding the implementation level details and giving the user a generic, implementation-independent data structure, the use of ADTs creates algorithms that result in simpler and cleaner code. ADTs can be implemented in any programming language such as C++, Java, and Scala. In this section, we shall implement ADTs using Python. Let's start with vectors first.

Vector

A vector is a single dimension structure to store data. They are one of the most popular data structures in Python. There are two ways of creating vectors in Python as follows:

- Using a Python list: The simplest way of creating a vector is by using a Python list, as follows:

  ```
  >>> myVector = [22,33,44,55]
  >>> print(myVector)
  [22 33 44 55]
  >>> print(type(myVector))
  <class 'list'>
  ```

  Note that this code will create a list with four elements.

- Using a `numpy` array: Another popular way of creating a vector is by using NumPy arrays, as follows:

  ```
  >>> myVector = np.array([22,33,44,55])
  >>> print(myVector)
  [22 33 44 55]
  >>> print(type(myVector))
  <class 'numpy.ndarray'>
  ```

Note that we created `myVector` using `np.array` in this code.

In Python, we can represent integers using underscores to separate parts. It makes them more readable and less error-prone. This is especially useful when dealing with large numbers. So, one billion can be represented as a=1

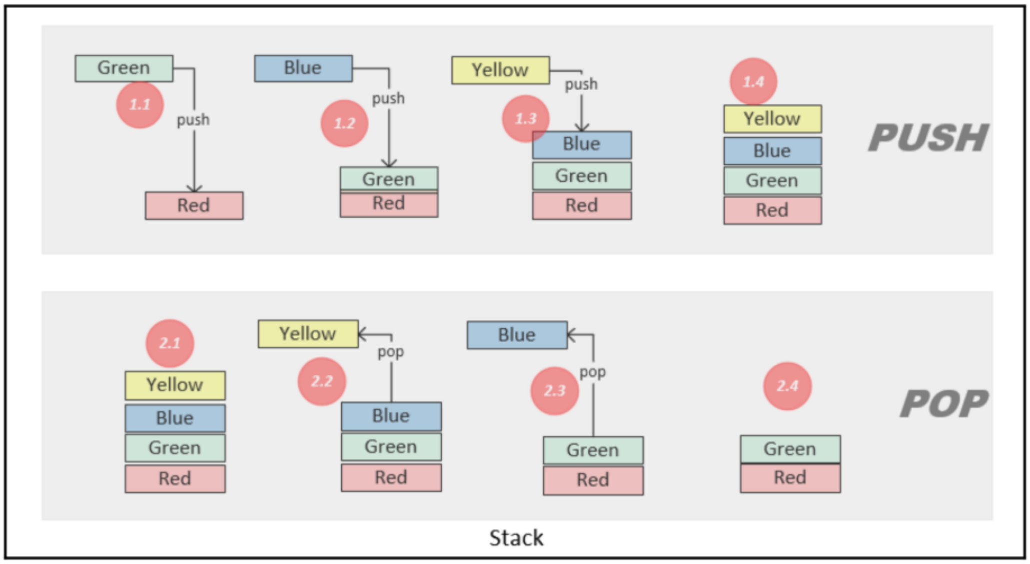
Stacks

A stack is a linear data structure to store a one-dimensional list. It can store items either in **Last-In, First-Out** (**LIFO**) or **First-In, Last-Out** (**FILO**) manner. The defining characteristic of a stack is the way elements are added and removed from it. A new element is added at one end and an element is removed from that end only.

Following are the operations related to stacks:

- **isEmpty:** Returns true if the stack is empty
- **push:** Adds a new element
- **pop**: Returns the element added most recently and removes it

The following diagram shows how push and pop operations can be used to add and remove data from a stack:



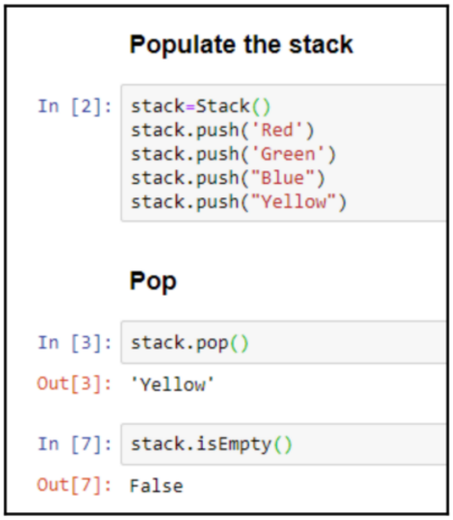
The top portion of the preceding diagram shows the use of push operations to add items to the stack. In steps **1.1**, **1.2**, and **1.3**, push operations are used three times to add three elements to the stack. The bottom portion of the preceding diagram is used to retrieve the stored values from the stack. In steps **2.2** and **2.3**, pop operations are used to retrieve two elements from the stack in LIFO format.

Let's create a class named `Stack` in Python, where we will define all of the operations related to the stack class. The code of this class will be as follows:

```
class Stack:
     def __init__(self):
         self.items = []
     def isEmpty(self):
         return self.items == []
     def push(self, item):
         self.items.append(item)
```

```
        def pop(self):
            return self.items.pop()
        def peek(self):
            return self.items[len(self.items)-1]
        def size(self):
            return len(self.items)
```

To push four elements to the stack, the following code can be used:



Populate the stack

```
In [2]: stack=Stack()
        stack.push('Red')
        stack.push('Green')
        stack.push("Blue")
        stack.push("Yellow")
```

Pop

```
In [3]: stack.pop()
Out[3]: 'Yellow'

In [7]: stack.isEmpty()
Out[7]: False
```

Note that the preceding code creates a stack with four data elements.

The time complexity of stacks

Let's look into the time complexity of stacks (using Big O notation):

Operations	**Time Complexity**
`push`	O(1)
`pop`	O(1)
`size`	O(1)
`peek`	O(1)

An important thing to note is that the performance of none of the four operations mentioned in the preceding table depends on the size of the stack.

Practical example

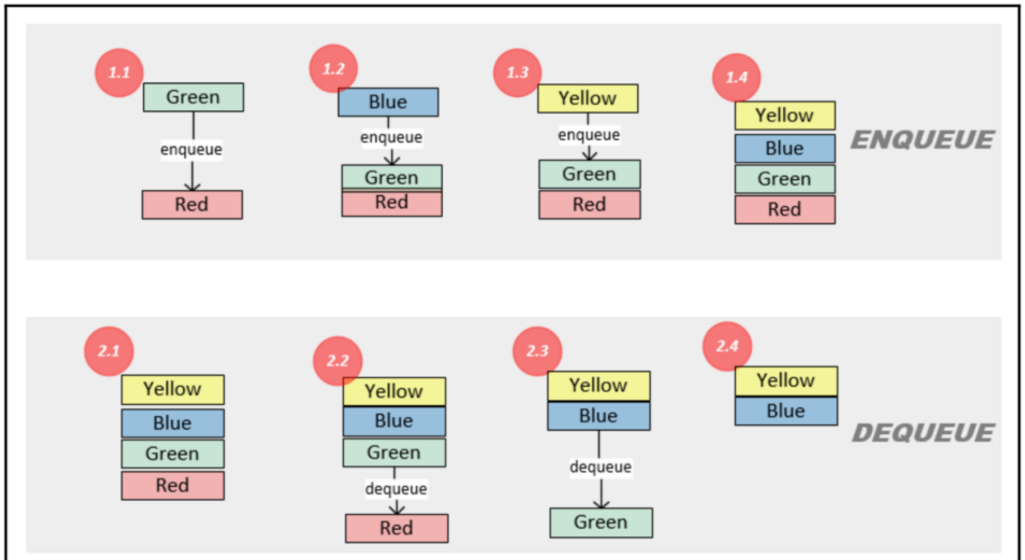
A stack is used as the data structure in many use cases. For example, when a user wants to browse the history in a web browser, it is a LIFO data access pattern and a stack can be used to store the history. Another example is when a user wants to perform an `Undo` operation in word processing software.

Queues

Like stacks, a queue stores *n* elements in a single-dimensional structure. The elements are added and removed in **FIFO** format. One end of the queue is called the *rear* and the other is called the *front*. When elements are removed from the front, the operation is called *dequeue*. When elements are added at the rear, the operation is called *enqueue*.

In the following diagram, the top portion shows the enqueue operation. Steps **1.1**, **1.2**, and **1.3** add three elements to the queue and the resultant queue is shown in **1.4**. Note that **Yellow** is the *rear* and **Red** is the *front*.

The bottom portion of the following diagram shows a dequeue operation. Steps **2.2**, **2.3**, and **2.4** remove elements from the queue one by one from the front of the queue:

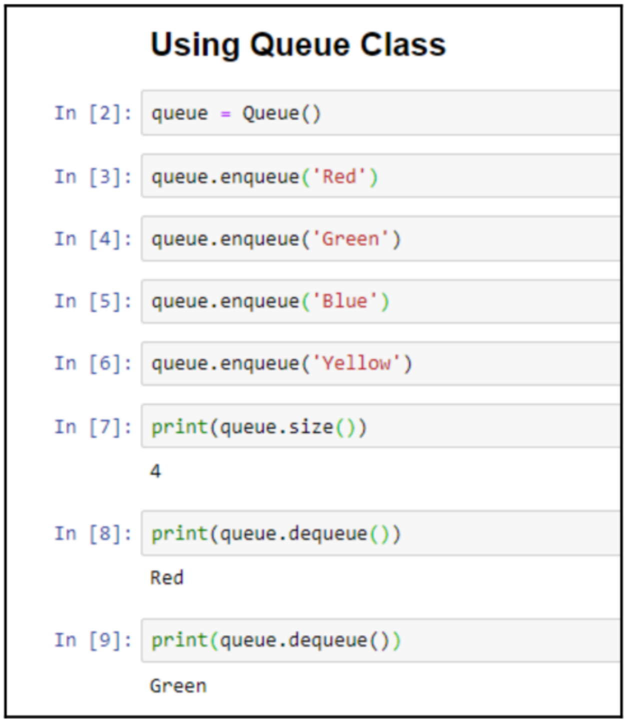


Queue

The queue shown in the preceding diagram can be implemented by using the following code:

```
class Queue(object):
   def __init__(self):
      self.items = []
   def isEmpty(self):
      return self.items == []
   def enqueue(self, item):
       self.items.insert(0,item)
   def dequeue(self):
      return self.items.pop()
   def size(self):
      return len(self.items)
```

Let's enqueue and dequeue elements as shown in the preceding diagram with the help of the following screenshot:



Using Queue Class

```
In [2]: queue = Queue()

In [3]: queue.enqueue('Red')

In [4]: queue.enqueue('Green')

In [5]: queue.enqueue('Blue')

In [6]: queue.enqueue('Yellow')

In [7]: print(queue.size())

        4

In [8]: print(queue.dequeue())

        Red

In [9]: print(queue.dequeue())

        Green
```

Note that the preceding code creates a queue first and then enqueues four items into it.

The basic idea behind the use of stacks and queues

Let's look into the basic idea behind the use of stacks and queues using an analogy. Let's assume that we have a table where we put our incoming mail from our postal service, for example, Canada Mail. We stack it until we get some time to open and look at the mail, one by one. There are two possible ways of doing this:

- We put the letter in a stack and whenever we get a new letter, we put it on the top of the stack. When we want to read a letter, we start with the one that is on top. This is what we call a *stack*. Note that the latest letter to arrive will be on the top and will be processed first. Picking up a letter from the top of the list is called a *pop* operation. Whenever a new letter arrives, putting it on the top is called *push* operation. If we end up having a sizable stack and lots of letters are continuously arriving, there is a chance that we never get a chance to reach a very important letter waiting for us at the lower end of the stack.
- We put the letter in pile, but we want to handle the oldest letter first: each time we want to look at one or more letters, we take care to handle the oldest one first. This is what we call a *queue*. Adding a letter to the pile is called an *enqueue* operation. Removing the letter from the pile is called *dequeue* operation.

Tree

In the context of algorithms, a tree is one of the most useful data structures due to its hierarchical data storage capabilities. While designing algorithms, we use trees wherever we need to represent hierarchical relationships among the data elements that we need to store or process.

Let's look deeper into this interesting and quite important data structure.

Each tree has a finite set of nodes so that it has a starting data element called a *root* and a set of nodes joined together by links called *branches*.

Terminology

Let's look into some of the terminology related to the tree data structure:

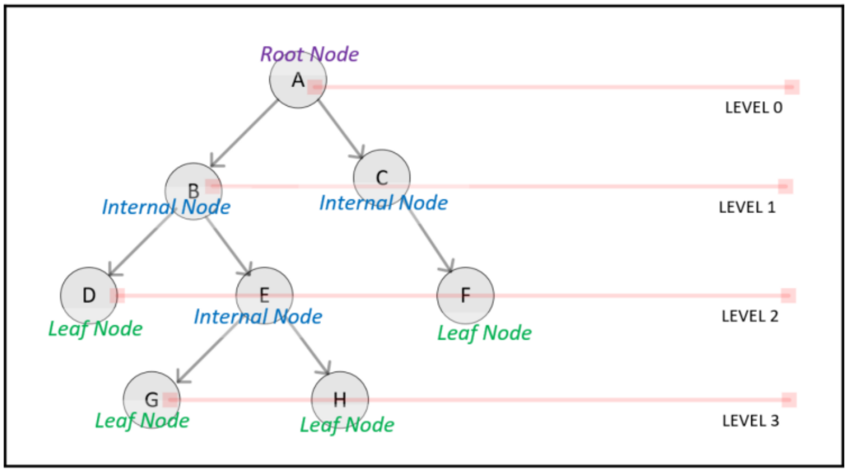
Root node	A node with no parent is called the *root* node. For example, in the following diagram, the root node is **A**. In algorithms, usually, the root node holds the most important value in the tree structure.
Level of a node	The distance from the root node is the level of a node. For example, in the following diagram, the level of nodes **D**, **E,** and **F** is two.
Siblings nodes	Two nodes in a tree are called *siblings* if they are at the same level. For example, if we check the following diagram, nodes **B** and **C** are siblings.
Child and parent node	A node, **F**, is a child of node **C**, if both are directly connected and the level of node **C** is less than node **F**. Conversely, node **C** is a parent of node **F**. Nodes **C** and **F** in the following diagram show this parent-child relationship.
Degree of a node	The degree of a node is the number of children it has. For example, in the following diagram, node **B** has a degree of two.
Degree of a tree	The degree of a tree is equal to the maximum degree that can be found among the constituent nodes of a tree. For example, the tree presented in the following diagram has a degree of two.
Subtree	A subtree of a tree is a portion of the tree with the chosen node as the root node of the subtree and all of the children as the nodes of the tree. For example, a subtree at node **E** of the tree presented in the following diagram consists of node **E** as the root node and node **G** and **H** as the two children.
Leaf node	A node in a tree with no children is called a *leaf* node. For example, in the following figure, **D**, **G**, **H,** and **F** are the four leaf nodes.
Internal node	Any node that is neither a root nor a leaf node is an internal node. An internal node will have at least one parent and at least one child node.

Note that trees are a kind of network or graph that we will study in `Chapter 6`, *Unsupervised Machine Learning Algorithms*. For graphs and network analysis, we use the terms link or edge instead of branches. Most of the other terminology remains unchanged.

Types of trees

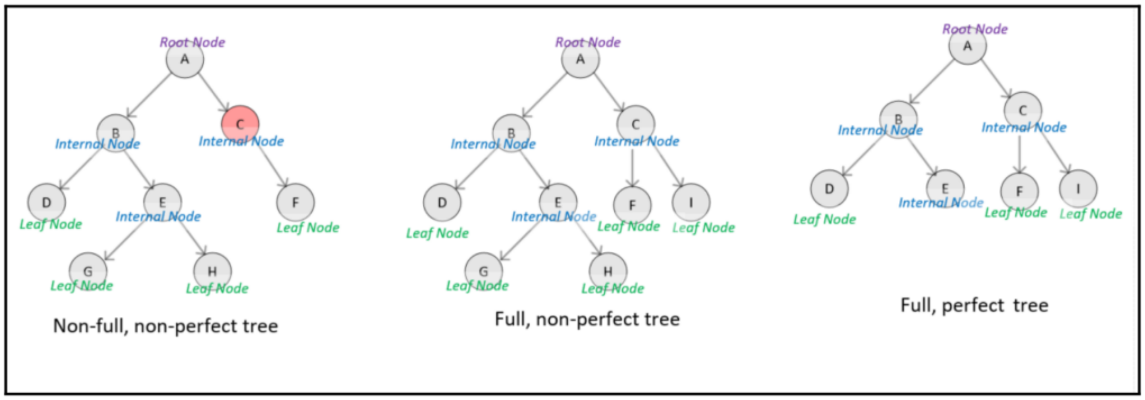
There are different types of trees, which are explained as follows:

- **Binary tree:** If the degree of a tree is two, that tree is called a *binary tree*. For example, the tree shown in the following diagram is a binary tree as it has a degree of two:



Note that the preceding diagram shows a tree that has four levels with eight nodes.

- **Full tree:** A full tree is the one in which all of the nodes are of the same degree, which will be equal to the degree of the tree. The following diagram shows the kinds of trees discussed earlier:



Note that the binary tree on the left is not a full tree, as node **C** has a degree of one and all other nodes have a degree of two. The tree in the middle and the one on the left are both full trees.

- **Perfect tree:** A perfect tree is a special type of full tree in which all the leaf nodes are at the same level. For example, the binary tree on the right as shown in the preceding diagram is a perfect, full tree as all the leaf nodes are at the same level, that is, **level 2**.
- **Ordered tree**: If the children of a node are organized in some order according to particular criteria, the tree is called an *ordered tree*. A tree, for example, can be ordered left to right in an ascending order in which the nodes at the same level will increase in value while traversing from left to right.

Practical examples

An abstract data type tree is one of the main data structures that are used in developing decision trees as will be discussed in `Chapter 7`, *Traditional Supervised Learning Algorithms*. Due to its hierarchical structure, it is also popular in algorithms related to network analysis as will be discussed in detail in `Chapter 6`, *Unsupervised Machine Learning Algorithms*. Trees are also used in various search and sort algorithms where divide and conquer strategies need to be implemented.

Summary

In this chapter, we discussed data structures that can be used to implement various types of algorithms. After going through this chapter, I expect that you should be able to select the right data structure to be used to store and process data by an algorithm. You should also be able to understand the implications of our choice on the performance of the algorithm.

The next chapter is about sorting and searching algorithms, where we will be using some of the data structures presented in this chapter in the implementation of the algorithms.

3
Sorting and Searching Algorithms

In this chapter, we will look at the algorithms that are used for sorting and searching. This is an important class of algorithms that can be used on their own or can become the foundation for more complex algorithms (presented in the later chapters of this book). This chapter starts by presenting different types of sorting algorithms. It compares the performance of various approaches to designing a sorting algorithm. Then, some searching algorithms are presented in detail. Finally, a practical example of the sorting and searching algorithms presented in this chapter is explored.

By the end of this chapter, you will be able to understand the various algorithms that are used for sorting and searching, and you will be able to apprehend their strengths and weaknesses. As searching and sorting algorithms are the building blocks for most of the more complex algorithms, understanding them in detail will help you understand modern complex algorithms as well.

The following are the main concepts discussed in this chapter:

- Introducing sorting algorithms
- Introducing searching algorithms
- A practical example

Let's first look at some sorting algorithms.

Introducing Sorting Algorithms

In the era of big data, the ability to efficiently sort and search items in a complex data structure is quite important as it is needed by many modern algorithms. The right strategy to sort and search data will depend on the size and type of the data, as discussed in this chapter. While the end result is exactly the same, the right sorting and searching algorithm will be needed for an efficient solution to a real-world problem.

The following sorting algorithms are presented in this chapter:

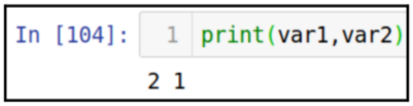
- Bubble sort
- Merge sort
- Insertion sort
- Shell sort
- Selection sort

Swapping Variables in Python

When implementing sorting and searching algorithms, we need to swap the values of two variables. In Python, there is a simple way to swap two variables, which is as follows:

```
var1 = 1
var2 = 2
var1,var2 = var2,var1
>>> print (var1,var2)
>>> 2 1
```

Let's see how it works:



```
In [104]:  print(var1,var2)
2 1
```

This simple way of swapping values is used throughout the sorting and searching algorithms in this chapter.

Let's start by looking at the bubble sort algorithm in the next section.

Bubble Sort

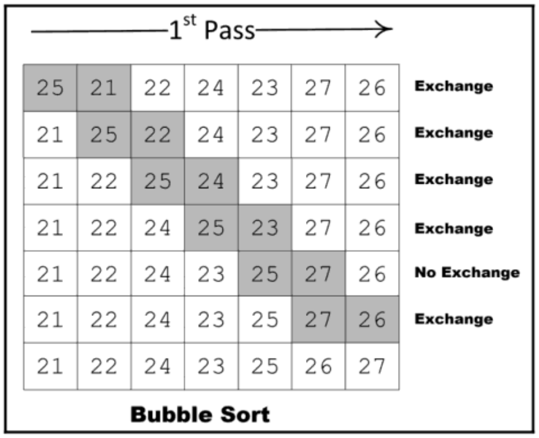
Bubble sort is the simplest and slowest algorithm used for sorting. It is designed in a way that the highest value in its list bubbles its way to the top as the algorithm loops through iterations. As its worst-case performance is O(N^2), as discussed previously, it should be used for smaller datasets.

Understanding the Logic Behind Bubble Sort

Bubble sort is based on various iterations, called **passes**. For a list of size *N*, bubble sort will have *N*-1 passes. Let's focus on the first iteration: pass one.

The goal of pass one is pushing the highest value to the top of the list. We will see the highest value of the list bubbling its way to the top as pass one progresses.

Bubble sort compares adjacent neighbor values. If the value at a higher position is higher in value than the value at a lower index, we exchange the values. This iteration continues until we reach the end of the list. This is shown in the following diagram:

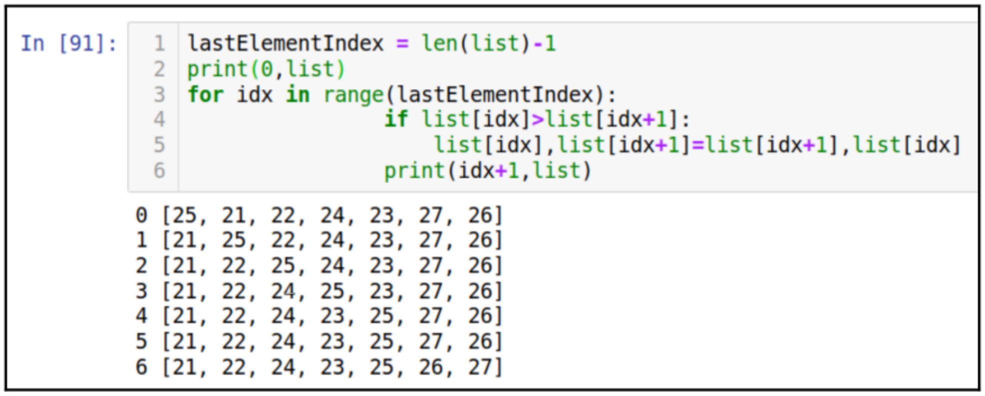


25	21	22	24	23	27	26	Exchange
21	25	22	24	23	27	26	Exchange
21	22	25	24	23	27	26	Exchange
21	22	24	25	23	27	26	Exchange
21	22	24	23	25	27	26	No Exchange
21	22	24	23	25	27	26	Exchange
21	22	24	23	25	26	27	

Let's now see how bubble sort can be implemented using Python:

```
#Pass 1 of Bubble Sort
lastElementIndex = len(list)-1
print(0,list)
for idx in range(lastElementIndex):
                    if list[idx]>list[idx+1]:
list[idx],list[idx+1]=list[idx+1],list[idx]
print(idx+1,list)
```

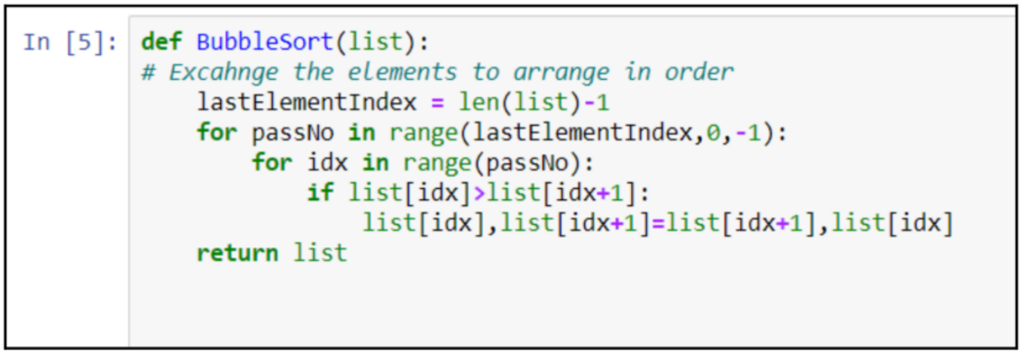
If we implement pass one of bubble sort in Python, it will look as follows:



```
In [91]: lastElementIndex = len(list)-1
         print(0,list)
         for idx in range(lastElementIndex):
                         if list[idx]>list[idx+1]:
                             list[idx],list[idx+1]=list[idx+1],list[idx]
                         print(idx+1,list)

0 [25, 21, 22, 24, 23, 27, 26]
1 [21, 25, 22, 24, 23, 27, 26]
2 [21, 22, 25, 24, 23, 27, 26]
3 [21, 22, 24, 25, 23, 27, 26]
4 [21, 22, 24, 23, 25, 27, 26]
5 [21, 22, 24, 23, 25, 27, 26]
6 [21, 22, 24, 23, 25, 26, 27]
```

Once the first pass is complete, the highest value is at the top of the list. The algorithm next moves on to the second pass. The goal of the second pass is to move the second highest value to the second highest position in the list. To do that, the algorithm will again compare adjacent neighbor values, exchanging them if they are not in order. The second pass will exclude the top element, which was put in the right place by pass one and need not be touched again.

After completing pass two, the algorithm keeps on performing pass three and so on until all the data points of the list are in ascending order. The algorithm will need *N*-1 passes for a list of size *N* to completely sort it. The complete implementation of bubble sort in Python is as follows:



```
In [5]: def BubbleSort(list):
        # Excahnge the elements to arrange in order
            lastElementIndex = len(list)-1
            for passNo in range(lastElementIndex,0,-1):
                for idx in range(passNo):
                    if list[idx]>list[idx+1]:
                        list[idx],list[idx+1]=list[idx+1],list[idx]
            return list
```

Now let's look into the performance of the `BubbleSort` algorithm.

A Performance Analysis of Bubble Sort

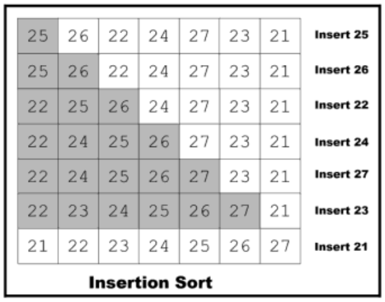
It is easier to see that bubble sort involves two levels of loops:

- **An outer loop**: This is also called **passes**. For example, pass one is the first iteration of the outer loop.
- **An inner loop**: This is when the remaining unsorted elements in the list are sorted, until the highest value is bubbled to the right. The first pass will have *N*-1 comparisons, the second pass will have *N*-2 comparisons, and each subsequent pass will reduce the number of comparisons by one.

Due to two levels of looping, the worst-case runtime complexity would be $O(n^2)$.

Insertion Sort

The basic idea of insertion sort is that in each iteration, we remove a data point from the data structure we have and then insert it into its right position. That is why we call this **the insertion sort algorithm**. In the first iteration, we select the two data points and sort them. Then, we expand our selection and select the third data point and find its correct position, based on its value. The algorithm progresses until all the data points are moved to their correct positions. This process is shown in the following diagram:



25	26	22	24	27	23	21	Insert 25
25	26	22	24	27	23	21	Insert 26
22	25	26	24	27	23	21	Insert 22
22	24	25	26	27	23	21	Insert 24
22	24	25	26	27	23	21	Insert 27
22	23	24	25	26	27	21	Insert 23
21	22	23	24	25	26	27	Insert 21

Insertion Sort

The insertion sort algorithm can be coded in Python as follows:

```
def InsertionSort(list):
    for i in range(1, len(list)):
        j = i-1
        element_next = list[i]
        while (list[j] > element_next) and (j >= 0):
            list[j+1] = list[j]
            j=j-1
        list[j+1] = element_next
    return list
```