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

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

Imran Ahmad

Packt

BIRMINGHAM - MUMBAI

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`

To my father, Inayatullah Khan, who still keeps motivating me to keep learning and exploring new horizons.

Contributors

About the author

Imran Ahmad is a certified Google instructor and has been teaching for Google and Learning Tree for a number of years. The topics Imran teaches include Python, machine learning, algorithms, big data, and deep learning. In his PhD, he proposed a new linear programming-based algorithm called ATSRA, which can be used to optimally assign resources in a cloud computing environment. For the last 4 years, Imran has been working on a high-profile machine learning project at the advanced analytics lab of the Canadian Federal Government. The project is to develop machine learning algorithms that can automate the process of immigration. Imran is currently working on developing algorithms to use GPUs optimally to train complex machine learning models.

About the reviewer

Benjamin Baka is a full-stack software developer and is passionate about cutting-edge technologies and elegant programming techniques. He has 10 years of experience in different technologies, from C++, Java, and Ruby to Python and Qt. Some of the projects he's working on can be found on his GitHub page. He is currently working on exciting technologies for mPedigree.

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:

- `Chapter 1`, *Overview of Algorithms*
- `Chapter 2`, *Data Structures used in Algorithms*
- `Chapter 3`, *Sorting and Searching Algorithms*
- `Chapter 4`, *Designing Algorithms*
- `Chapter 5`, *Graph Algorithms*

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?

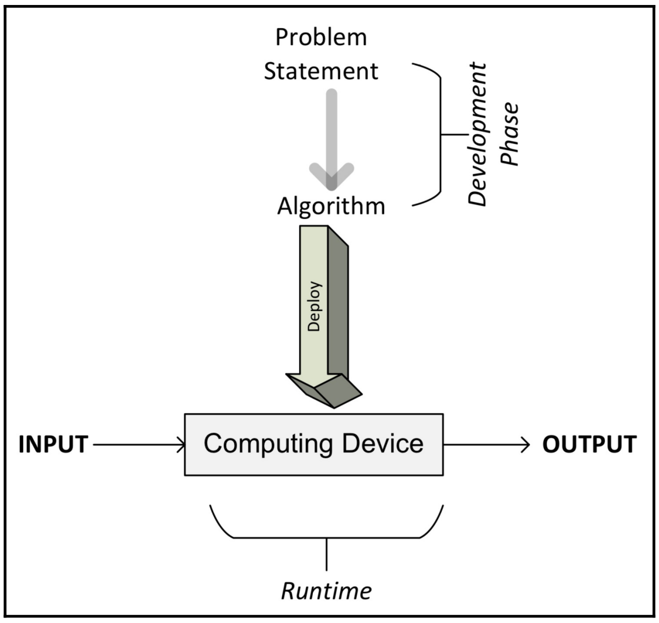
In the simplest terms, an algorithm is a set of rules for carrying out some calculations to solve a problem. It is designed to yield results for any valid input according to precisely defined instructions. If you look up the word algorithm in an English language dictionary (such as American Heritage),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

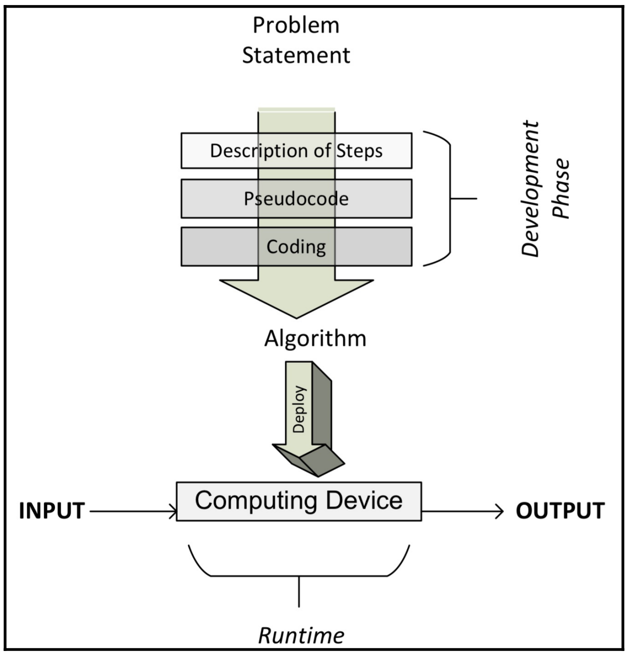
Once designed and implemented in a programming language of your choice, the code of the algorithm is ready to be deployed. Deploying an algorithm involves the design of the actual production environment where the code will run. The production environment needs to be designed according to the data and 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$_{T_i}$ [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`$_1$ 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:

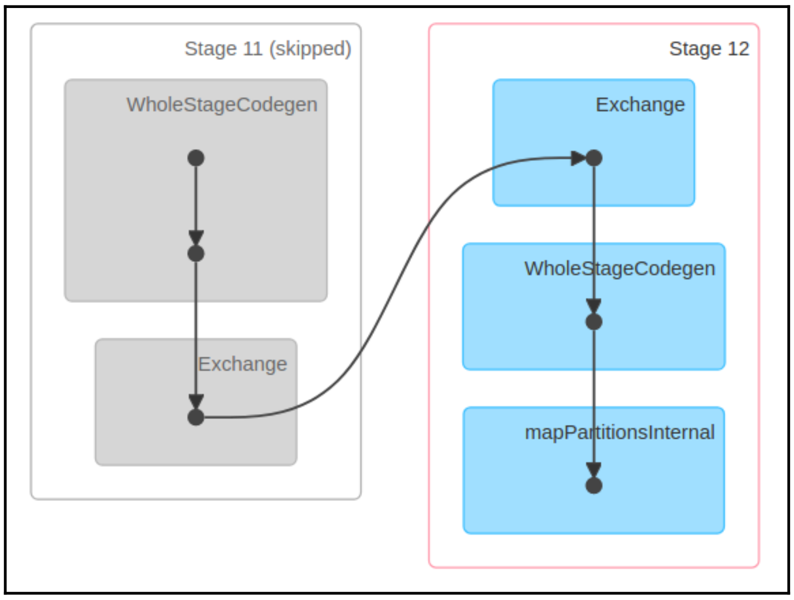
```
define swap(x, y)
    buffer = x
    x = y
    y = buffer
```

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.

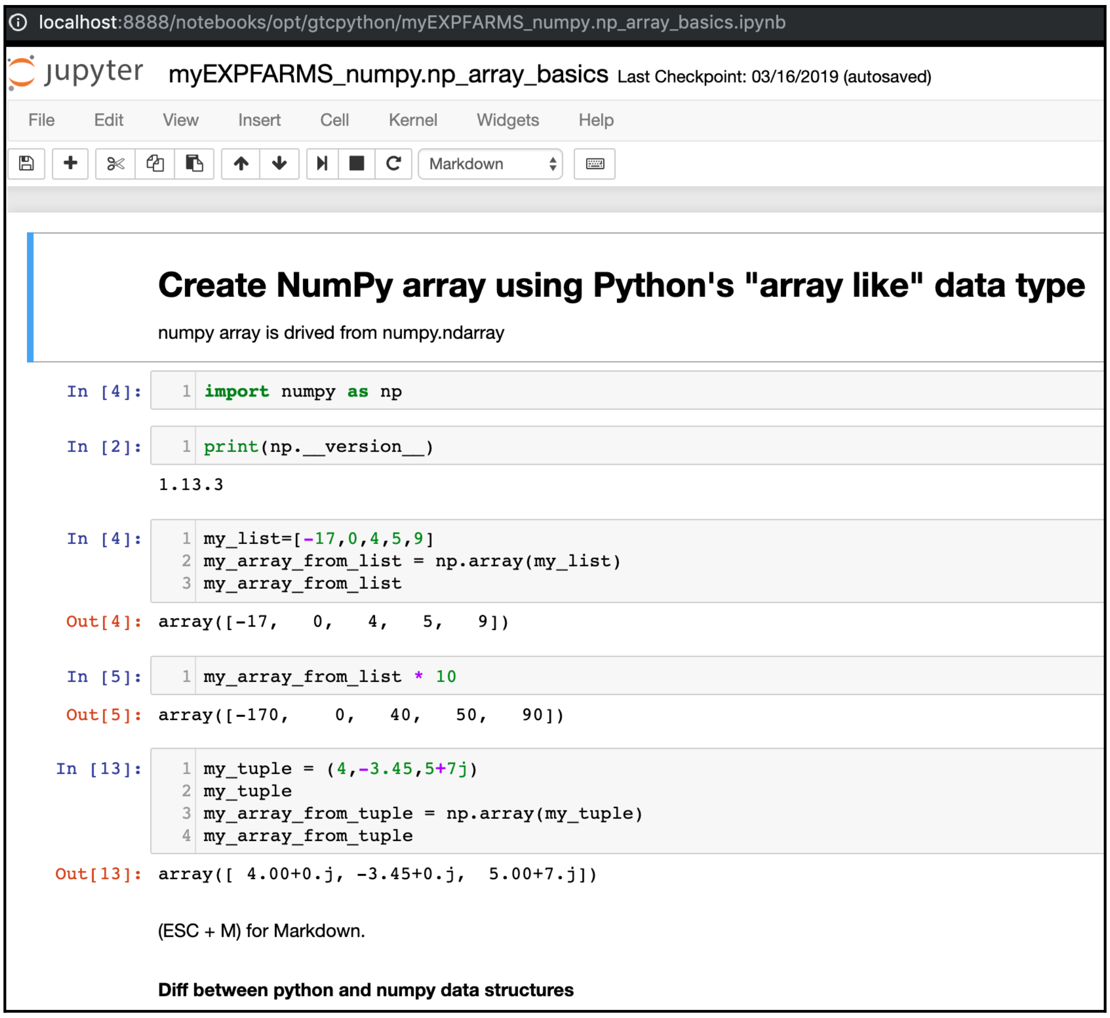
The following are the main packages that are part of this ecosystem:

- **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.

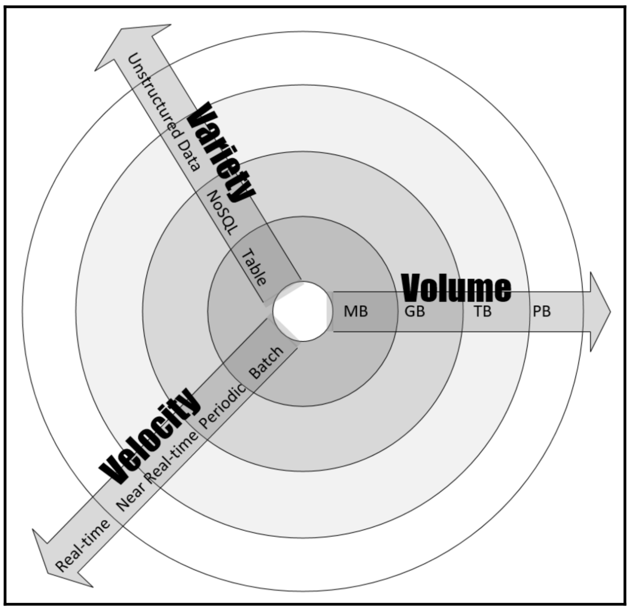
To characterize the problem in terms of its complexity and needs, it helps if we study its data and compute dimensions in more depth, which we will do in the following section.

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:



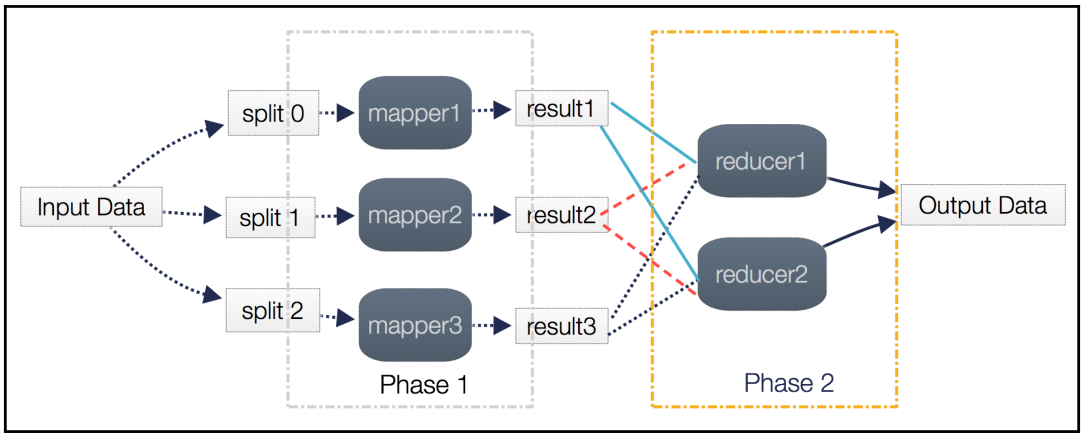
For example, if the input data is a simple `csv` file, then the volume, velocity, and variety of the data will be low. On the other hand, if the input data is the live stream of a security video camera, then the volume, velocity, and variety of the data will be quite high and this problem should be kept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


split 0
mapper1
result1
reducer1
Input Data
split 1
mapper2
result2
Output Data
reducer2
split 2
mapper3
result3
Phase 1
Phase 2