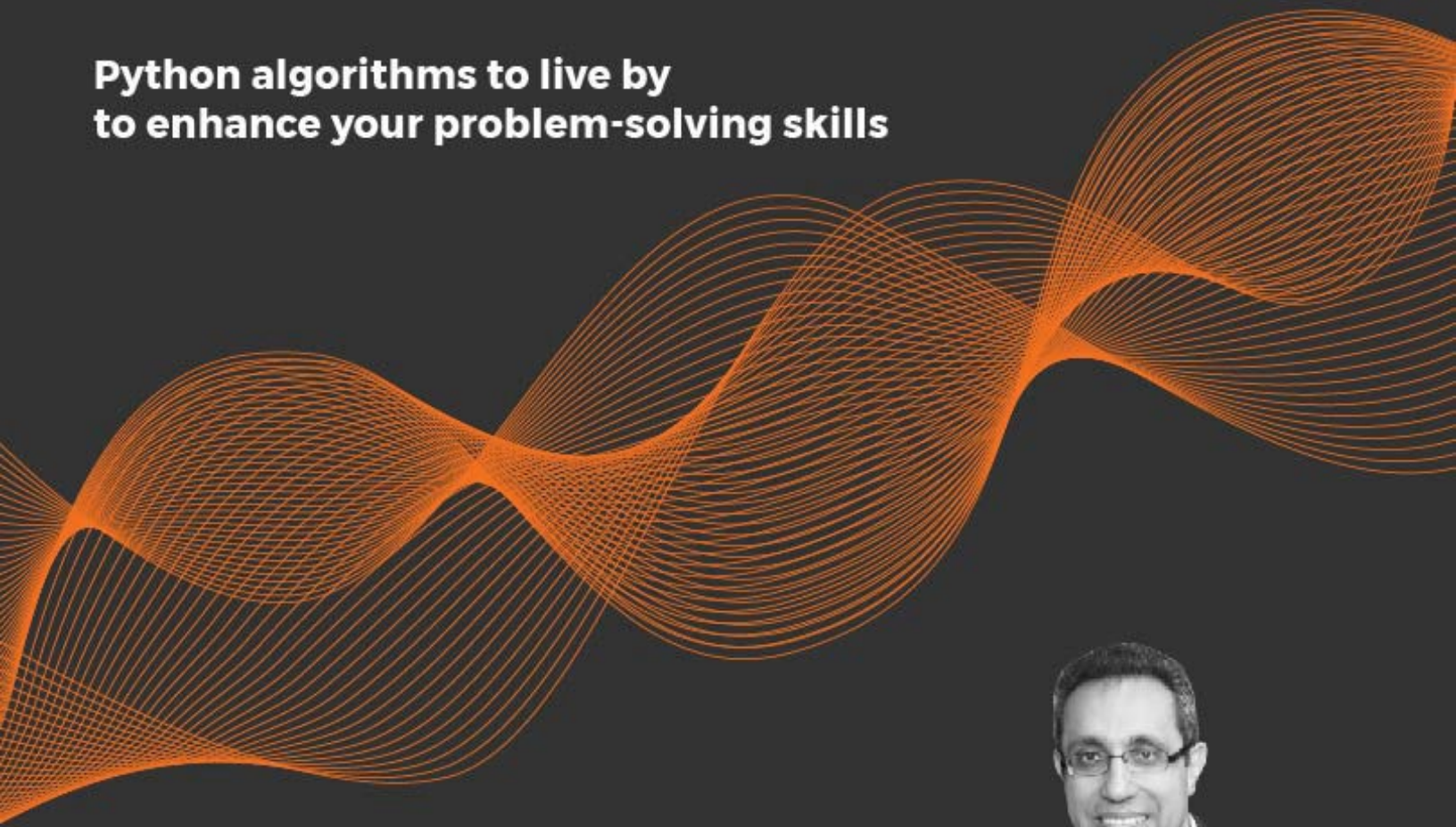
EXPERT INSIGHT

40 Algorithms Every Programmer Should Know

Python algorithms to live by to enhance your problem-solving skills



Second Edition

Imran Ahmad

Packt

40 Algorithms Every Programmer Should Know

Early Access Publication: 40 Algorithms Every Programmer Should Know

Early Access Production Reference: B18046

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK

ISBN: 978-1-80324-776-2

www.packt.com

Table of Contents

40 Algorithms Every Programmer Should Know, Second Edition: Python algorithms to live by to enhance your problem-solving skills

Welcome to Packt Early Access. We're giving you an exclusive preview of this book before it goes on sale. It can take many months to write a book, but our authors have cutting-edge information to share with you today. Early Access gives you an insight into the latest developments by making chapter drafts available. The chapters may be a little rough around the edges right now, but our authors will update them over time.

You can dip in and out of this book or follow along from start to finish; Early Access is designed to be flexible. We hope you enjoy getting to know more about the process of writing a Packt book.

15. Chapter 15: Practical Considerations

Section 1: Fundamentals and Core Algorithms

This section introduces the core aspects of algorithms. We will explore what an algorithm is and how to design it. We will also learn about the data structures used in algorithms. This section also introduces sorting and searching algorithms along with 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

“An Algorithm must be seen to be believed Donald Knuth”

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. This is the second edition of this book. In this edition, we specially focused 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. To provide a historical perspective, 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?
- The phases of an algorithm
- Development Environment
- 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 a 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 *Figure 1.1*:

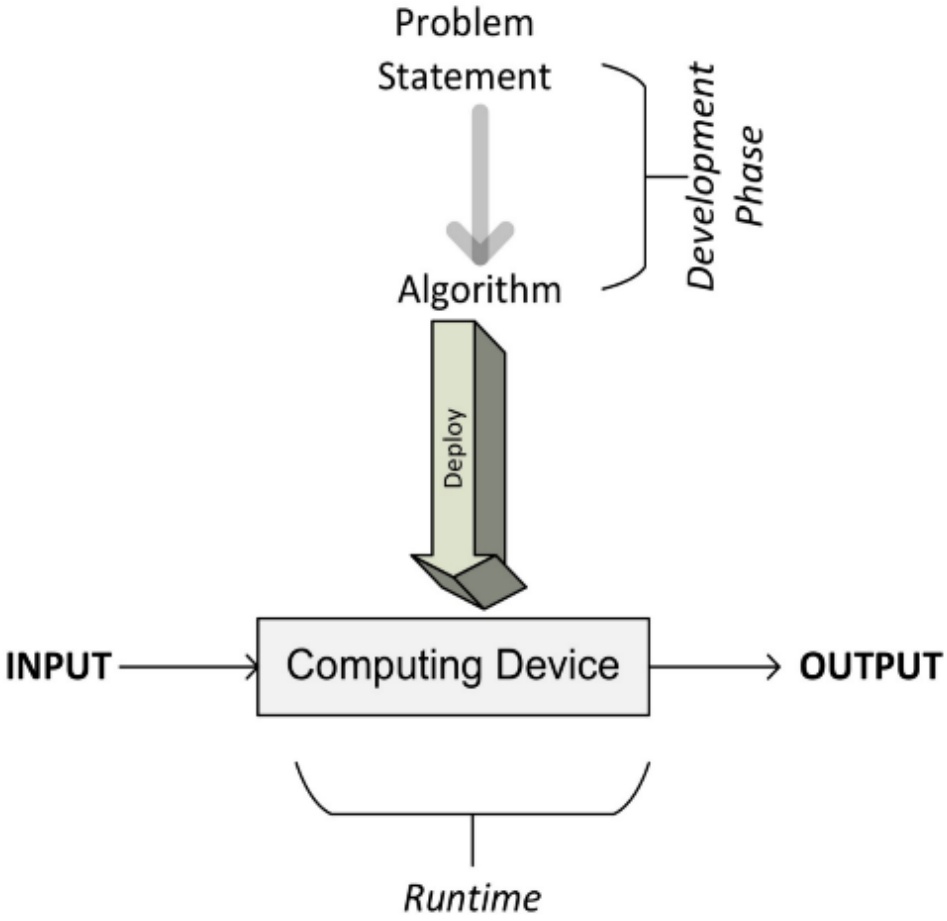


Figure 1.1: The different phases of developing, deploying, and using an algorithm

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:

1. **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 best solution to a given

problem, in many cases we will end up having more than one candidate algorithms. 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 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.

2. **The coding phase**: In the coding phase, the designed algorithm is converted into a computer program. It is important that the computer program implements all the logic and architecture suggested in the design phase.

The requirements of the business problem can be divided into functional and non-functional requirements. The requirements that directly specifies the expected features of the solutions are called the functional requirements. Functional requirements detail the expected behavior of the solution. On the other hand, the non-functional requirements are about the performance, scalability, usability and the accuracy of the algorithm. Non-functional requirements also establish the expectations about the security of the data. For example, let us consider that we are required to design an algorithm for a credit card company that can identify and flag the fraudulent transactions. Function requirements in this example will specify the expected behavior of a valid solution by providing the details of the expected output given a certain set of input data. In this case the input data may be the details of the transaction, and the output may be a binary flag that labels a transaction as fraudulent or non-fraudulent. In this example, the non-functional requirements may specify that the response time of each of the prediction. Non-functional requirements will also set the allowable thresholds for accuracy. As we are dealing with financial data in this example, the security requirements related to user authentication, authorization and data confidentiality are also expected to be part of non-functional requirements.

Note that functional and non-functional requirements aims to precisely define **what** needs to be done. Designing the solution is about figuring out **how** it will be done. And implementing the designing is actual solution in the programming language of your choice. Coming up with a design that fully meets both functional and non-functional requirements may take lots of time and effort. The choice of the right programming language and development/production environment may depend on the requirements of the problem. For example, as C/C++ is a lower-level language than Python, it may be a better choice for algorithms needing complied code and lower-level optimization.

Once the design phase is completed and the coding is complete, 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 14 Large Scale Algorithms*, and *Chapter 15 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.

Development Environment

Once designed, algorithms need to be implemented in a programming language as per the design. For this book, we have chosen the programming language Python. We chose it because Python is a flexible and is an open-source programming language. Python is also one of the languages that you can use in various 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.

A basic understanding of Python is required to better understand the concepts presented in this book.

For this book, we expect you to use the recent version of Python 3. At the time of writing, the most recent version is 3.10 which is what we will use to run the exercises in this book.

> We will be using Python throughout this book. We will also be using Jupyter Notebooks to run the code. The rest of the chapters in this book assume that Python is installed and Jupyter notebooks are properly configured and running.

Python packages

Python is a general-purpose language. It follows the the philosophy of "batteries included" which means that there is a standard library which is available, without making the user download separate packages. However, the standard library modules only provide the bare minimum functionality. Based on the specific use case you are working on additional packages may need to be installed. The official third party repository for Python packages is called PyPi which stands for Python Package Index. It hosts Python packages both as source distribution and pre-compiled code. Currently, there are more than 113,000 Python packages hosted at PyPi. The easiest way to install additional packages is through the pip package management system. The " `pip` " is a nerdy recursive acronym that are abundant in Python language. Pip stands for "Pip Installs Python". Good news is that from version 3.4 of Python, pip is installed by default. To check the version of pip you can type on the command line:

```
pip --version
```

This `pip` command can be used to install the additional packages:

```
pip install `PackageName`
```

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 `PackageName` –upgrade
```

And to install a specific version of a Python package:

```
pip install `PackageName==2.1`
```

> Adding the right libraries and versions have become part of setting up the Python programming environment. One feature that helps with maintaining these libraries is the ability of creating a requirements file that lists all the packages that are needed. The requirements file is a simple text file that contains the name of the libraries and their associated versions. A sample of requirements file look like as follows:
>
> ```
> scikit-learn==0.24.1
>
> tensorflow==2.5.0
>
> tensorboard==2.5.0
> ```
>
> By convention the `requirements.txt` is placed in the project's top-level directory.
>
> Once created, the requirement file can be used to setup the development environment by installing all the Python libraries and their associated versions by using the following command.

```
pip install -r requirements.txt
```

Now let us look into the main packages that we will be using in this book.

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. SciPy is performant as it acts as a thin wrapper around optimized code written in C/C++ or Fortran.

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 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/.

Using Python Notebooks

We will be using Jupyter Notebook and Google's colaboratory as the IDE. More details about the setup and the use of Jupyter Notebooks and Colab can be found in Appendix A and B.

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 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.

Figure 1.5 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.

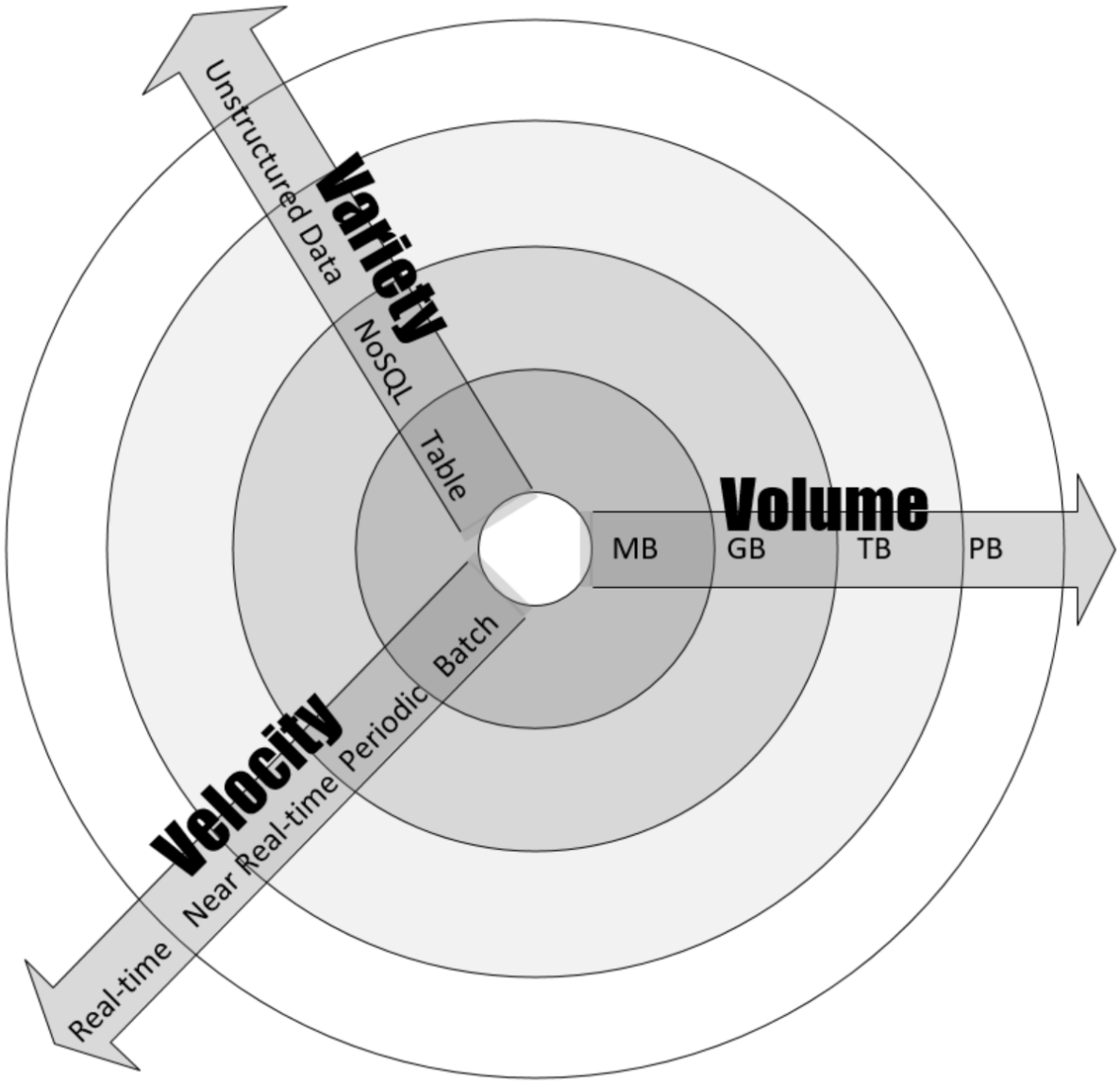


Figure 1.5: 3Vs of Data: Volume, Velocity and Variety

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

To characterize the compute dimension, we analyze the processing needs of the problem at hand. The processing needs of an algorithm determine what sort of design is most efficient for it. For example, complex algorithms, in general, require lots of processing power. For such algorithms, it may be important to have multi-node parallel architecture. Modern deep algorithms usually involve considerable numeric processing and may need the power of GPUs or TUPs as discussed in *Chapter 15*, *Practical Considerations*.

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:

- **Should be Correct**: A good algorithm should produce the correct result. To confirm that an algorithm is working correctly, it needs to be extensively tested, especially testing edge cases.
- **Should be Understandable**: A good algorithm should be understandable. The best algorithm in the world is not very useful if it's too complicated for us to implement on a computer.
- **Should be Efficient**: A good algorithm should be efficient. Even if an algorithm produces a correct result, it won't help us 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.

Let us study them one by one:

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 need to have efficient resource allocation mechanisms that are aware of the memory requirements at different execution phases of the algorithm. Complex algorithms tend to be iterative in nature. Instead of bringing all the information into the memory at once, such algorithms iteratively populate the data structures. To calculate the space complexity, it is important the first classify the type of iterative algorithm we plan to use. An iterative algorithm can use one of the following three types of iterations.

- **Converging Iterations**: As the algorithm proceed through iterations the amount of data it processes in each individual iteration decreases. In other words, space complexity decreases as the algorithm proceeds through its iterations. The main challenge is to tackle the space complexity of the initial iterations. Modern scalable cloud infrastructures such as AWS and Google Cloud are best suited to run such algorithms.
- **Diverging Iterations**: As the algorithm proceed through iterations the amount of data it processes in each individual iteration increases. As the space complexity increases with algorithm progress through iterations, it is important to set constraints to prevent the system becoming unstable. The constraints can be set by limiting the number of iterations and/or by setting a limit of the size of initial data.
- **Flat Iterations**: As the algorithm proceed through iterations the amount of data it processes in each individual iteration remains constant. As space complexity does not change, elasticity in infrastructure is not needed.

To calculate the space complexity, we need to focus one of the most complex iteration. For example, for an algorithm using converting iterations, we can choose one of the initial iterations. Once chosen, we estimate the total amount of memory used by the algorithm, including the memory used by its transient data structures, execution, and input values. This will give us a good estimate of the space complexity of an algorithm.

The following are guidelines to minimize the space complexity:

- Whenever possible, try to design an algorithm as iterative.
- While designing an iterative algorithm, whenever there is a choice, prefer larger number of iterations over smaller number of iterations. Fine-grained larger number of iterations are expected to have less space-complexity.
- Algorithms should bring only the information needed for current processing into memory. Whatever is not needed should be flushed out from the memory.

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 14, 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 two questions:

Will this algorithm scale? A well-designed algorithm should be fully capable of taking advantage of the modern elastic infrastructure available in cloud computing environments. An algorithm should be designed in a way such it can utilize the availability of more CPUs, processing cores, GPUs and memory. For example, an algorithm used for training a model in a machine learning problem should be able to use distributed training as more CPU are available. Such algorithm should also take advantage of GPUs and additional memory if made available during the execution of the algorithm.

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? Analyzing time complexity of an algorithm may answer 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 bunch of algorithms readily 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 simple algorithm. But as n increases, the choice of the right algorithm starts to make a difference. A poorly designed algorithm may 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 was first introduced by Bachmann in 1894 in a research paper to approximate the an algorithm's growth. He wrote:

"… with the symbol O(n) we express a magnitude whose order in respect to n does not exceed the order of n" (Bachmann 1894, p. 401)".

Thus Big-O notation measures an algorithm's asymptotic performance. To explain how we use Big-O notation consider two functions, f(n) and g(n) be functions. We can say that f = O(g) if and only as n increases to infinity, if

$\frac{f(n)}{g(n)}$ is bounded.

Let's look at a particular function:

f(n) = 1000n2 + 100n + 10

and

g(n) = n2.

Note that both functions will approach infinity as n goes approaches infinity. Let's find out if f = O(g) by applying the definition.

First let is calculate $\frac{f(n)}{g(n)}$

which will be equal to $\frac{f(n)}{g(n)} = \frac{1000n^2+100n+10}{n^2} = (1000 + \frac{100}{n} + \frac{10}{n^2})$

It is clear that $\frac{f(n)}{g(n)}$ is bounded and will not approach as n approaches infinity.

Thus f(n) = O(g) = O(n2).

(n2) represents that complexity of this function increases as the square of inputs n. If we double the number of input elements the complexity is expected to increase by 4.

Note the following 4 Rules when dealing with Big-O notation.

Rule 1:

Let us look into the complexity of loops in algorithms. If an algorithm performs a certain sequence of steps n times that it has O(n) performance.

Rule 2:

Let us look into the nested loops of the algorithms. If an algorithm performs a function that has a loop of n1 steps, and for loop it performs another n2 steps, the algorithm's total performance is O(n1 × n2).

For example if an algorithm as both outer and inner loops having n steps then the complexity of the algorithm will be represented by

O(n*n) = O(n2)

Rule 3:

If an algorithm performs a function f(n) that takes n1 steps and and then

performs another function g(n) takes n2 steps the algorithm's total performance is O(f(n)+g(n)).

Rule 4:

If an algorithm takes O(g(n) + h(n)) and the function g(n) is greater than

h(n) for large n, the algorithm's performance can be simplified to O(g(n)).

It means that O(1+n) = O(n)

And O(n2+ n3) = O(n2)

Rule 5:

When calulating complexity of an algorithm, ignore constant multiples. If k is a constant, O(kf(n)) is the same as

O(f(n)).

Also, O(f(k × n)) is the same as O(f(n)).

Thus O(5n2) = O(n2)

And O((3n2)) = O(n2)

Note that:

- The complexity quantified by Big O notation is only an estimate.
- For smaller size of data, we do not care about the time-complexity. n0 in the graph defines the threshold above which we are interested in finding the time-complexity. Shaded area describes this area of interest where we will analyze the time complexity.
- T(n) time complexity more than the original function. A good choice of T(n) will try to create a tight upper bound for F(n)

The following table summarizes the different kinds of Big O notation types discussed in this section:

Complexity Class	Name	Example Operations
O(1)	Constant	append, get item, set item.
O(logn)	Logarithmic	Finding an element in a sorted array.
O(n)	Linear	copy, insert, delete, iteration
nLogn	Linear-Logarithmic	Sort a list, merge - sort.
n2	Quadratic	Nested loops

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 nth 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):

```
[10]  1 def getFirst(my_list):
      2     return my_list[0]
```

The output is shown as:

```
[11]  1 getFirst([1, 2, 3])

      1
```

```
[12]  1 getFirst([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

      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:

```
1 def get_sum(my_list):
2     sum = 0
3     for item in my_list:
4         sum = sum + item
5     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:

```
1 get_sum([1, 2, 3])
```

```
6
```

```
1 get_sum([1, 2, 3, 4])
```

```
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(n2)) 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 get_sum(my_list):
    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(n2):

```
[13]  get_sum([[1, 2], [3, 4]])

      10
```

```
[15]  get_sum([[1 ,2 , 3],[4 , 5, 6]])

      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 a logarithmic algorithm 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:

```
1  def search_binary(my_list, item):
2      first = 0
3      last = len(my_list)-1
4      found_flag = False
5      while(first <= last and not found_flag):
6          mid = (first + last)//2
7          if my_list[mid] == item:
8              found_flag = True
9          else:
10             if item < my_list[mid]:
11                 last = mid - 1
12             else:
13                 first = mid + 1
14     return found_flag
```

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:

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(n2) 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(n2) is not as bad as O(n3) 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**.

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:

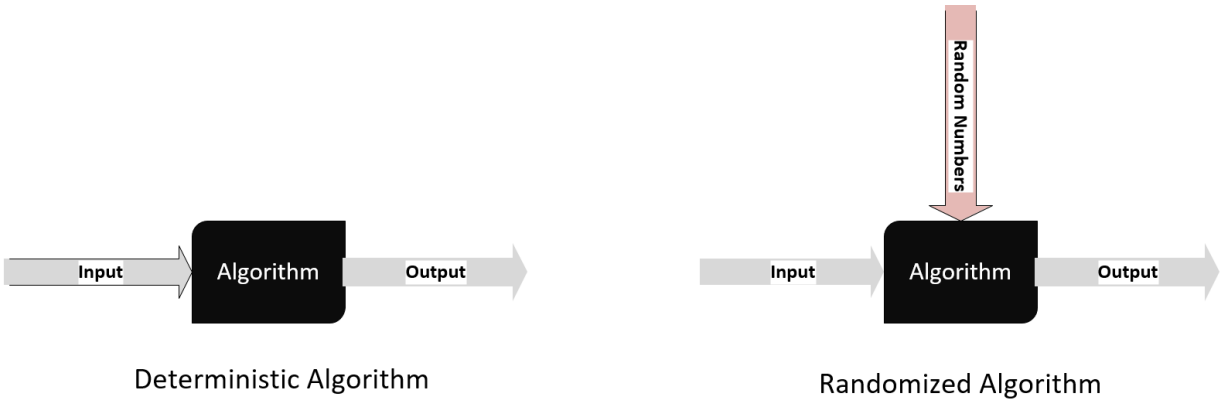


Figure 1.2 Deterministic and Randomized Algoirthms

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. 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.

Join our book's Discord space

Join the book's Discord workspace for a monthly *Ask me Anything* session with the author: https://packt.link/40Algos

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 Python build-in data types
- Using Series and Dataframes
- Exploring matrices and matrix operations
- Understanding Abstract Data Types (ADTs)

Exploring Python built-in data types

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, the important data structures that can be used to store collections are summarized in *Table 2.1*:

Data Structure	Brief Explanation	Example
Lists	An ordered, possible nested, mutable sequence of elements.	["John", 33,"Toronto", True]
Tuple	An ordered immutable sequence of elements	('Red','Green','Blue','Yellow')
Dictionary	An unordered collection of key-value pairs	{'food': 'spam', 'taste': 'yum'}
Set	An unordered collection of elements	{'a', 'b', 'c'}

Table 2.1: Python Data Structures

Let us look into them in more detail in the upcoming subsections.

Lists

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

A list can be defined by enclosing the elements 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:

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

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 *Figure 2.1*:

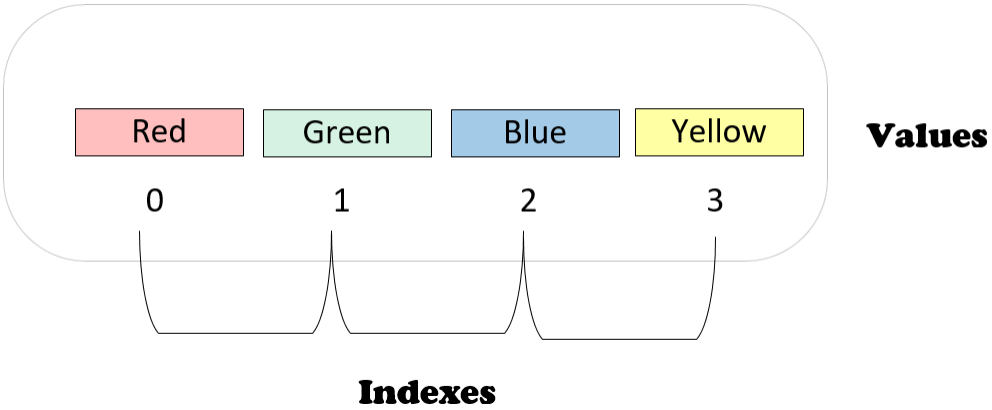


Figure 2.1: A four-element list in Python

Note that in Python is a zero-indexing language. Which means that initial index of any data structure, including lists, will be 0. And `"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, as demonstrated by the preceding code.

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 any 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 this_color in bin_colors:
        print(f"{this_color} Square")
Red Square
Green Square
Blue Square
Yellow Square
```

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

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:

```
>>> odd_num = range(3,29,2)
>>> odd_num
[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
```

> 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. When a data structure is passed to a function as immutable, its copy does not need to be created as the function cannot change it. So, the output can refer to the input data structure. This is called referential transparency and improves the performance. There is a price we pay for the ability to change data elements in lists 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, as Python is a zero-indexed based language, `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).

Dictionaries and sets

In this section we will discuss sets and dictionary which are used to store data which there is no explicit or implicit ordering. Both dictionary and sets are quite similar. The difference is that dictionary has a key and a value pair. A set can be thought of a collection of unique keys.

Let us look into them one by one

Dictionaries

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 limitation on the value of keys is that they must be hashable types. An hashable are the type of objects on which we can run the hash function generating a hash code that never changes during its lifetime. This ensures that the keys are unique and searching for the key is fast. Numeric types and flat immutable types str are all hashable and are good choice for the dictionary keys. 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:

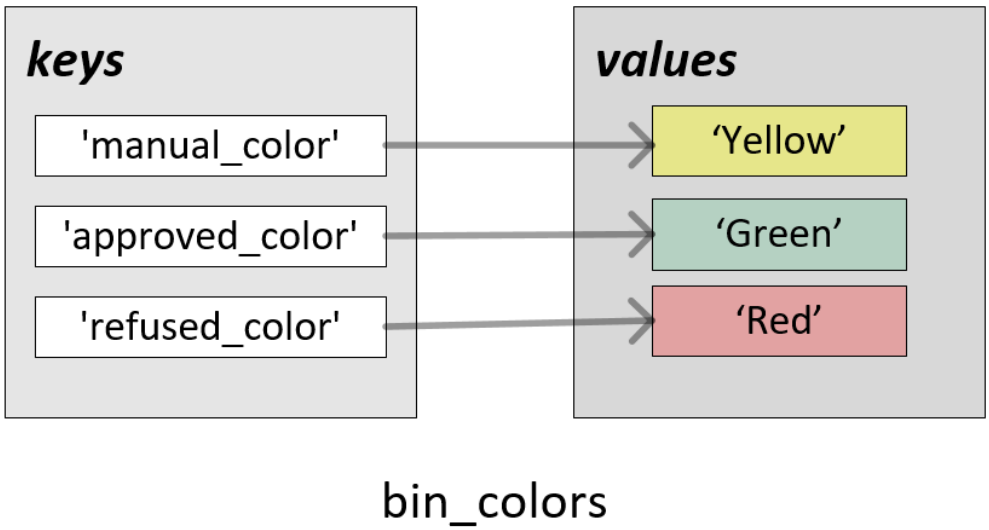


Figure 2.2: Key-value pairs in a simple dictionary

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

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'
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.

Sets

Closely related to dictionary is a set which is defined as an unordered collection of distinct elements that can be of different types. One of the ways to define a set is to enclose the values in { }. 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:

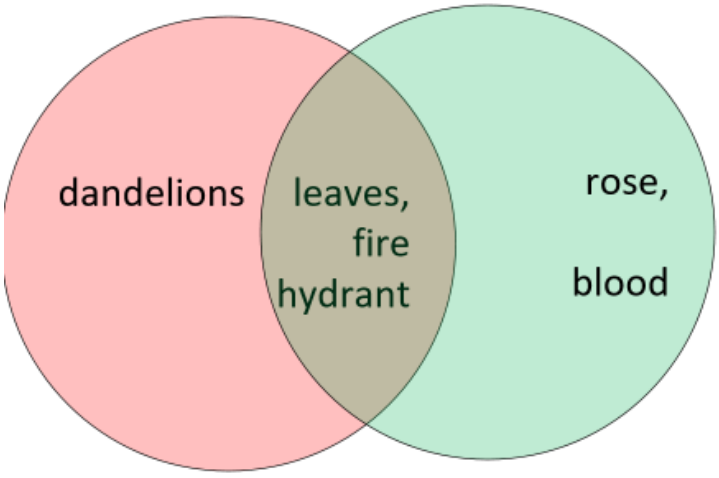


Figure 2.3: Venn diagram showing how elements are stored in sets

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','leaves'}
```

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 dictionary and sets

Following is the time complexity analysis for sets:

Sets	Complexity
Add an element	O(1)
Retrieve an element	O(1)
Copy	O(n)

An important thing to note from the complexity analysis of the dictionary or sets is that they use hash tables to achieve O(1) performance for both elements additions and lookup. So, the time taken to get or set a key-value is totally independent of the size of the dictionary or sets. This is because in a dictionary, the hash function maps the keys to an integer which is used to store the key-value pairs. The pandas library cleverly uses the hash function to turn an arbitrary key into the index for a sequence. This means that the time taken to get a key-value pair to a dictionary of a size of three is the same as the time taken to get a key-value pair to a dictionary of a size of one million. In sets only the keys are hashed and stored resulting in the same time complexity.

When to use a dictionary and when to use a set?

Let us assume that we are looking for a data structure for our phone book. We want to store the phone number of the employees of a company. For this purpose, a dictionary is the right data structure. Name of each employee will be the key and the value would be the phone number

```
phonebook = {
    "Ikrema Hamza": "555-555-5555",
    "Joyce Doston" : "212-555-5555",
}
```

But if we want to store only the unique value of the employees, then that should be done using sets.

```
employees = {
    "Ikrema Hamza",
    "Joyce Doston"
}
```

Using series and DataFrames

Processing data is one of the core things that need to be done while implementing most of the algorithms. In Python the data processing is usually done by using various functions and data structures of pandas library. In this section, we will look into the following two

important data structures of pandas library which will be using in implementing various algorithms later in this book.

Series: one dimensional array of values

DataFrame: two-dimensional data structure used to store tabular data

Let us look into Series data structure first.

Series

In pandas library, `Series` is a one-dimensional array of values for homogenous data. We can think of Series as a single column in a spreadsheet. You can think series is holding various values of a particular variable.

A series can be defined as follows:

```
[1]: import pandas as pd
     pd.Series(['red', 'green', 'blue', 'white', 'black'])
```

Note that in pandas Series based data structures, there is a term “axis” used to represent a sequence of values in a particular dimension. Series has only “axis 0” because it has only one dimension. We will see that how this axis concept is applied to DataFrame in the next section which is a 2-dimensional data structure.

DataFrame

A DataFrame is built upon the Series data structure and is store 2-dimensional tabular data . 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. In `DataFrame`, a single column or row of is called an **axis**.

> 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.

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 all 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 that 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:

```
>>> s
  id name age decision
1 2 Elena 23 False
2 3 Steven 40 True
```

Note that the preceding code will return the second and third row plus all columns. It uses the `iloc` method that allows us to access the elements by their numerical index.

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.

Matrices

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

Abstract Data Types (**ADT**) are high-level abstractions whose behavior is defined by a set of variables and set of related operations. ADTs define the implementation guidance of ‘what’ needs to be expected but give the programmer freedom that ‘how’ it will be exactly implemented. Examples are vectors, queues and stacks. It means that two different programmers can take two different approaches to implement an ADT like stacks 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

Figure 2.4 shows how push and pop operations can be used to add and remove data from a stack:

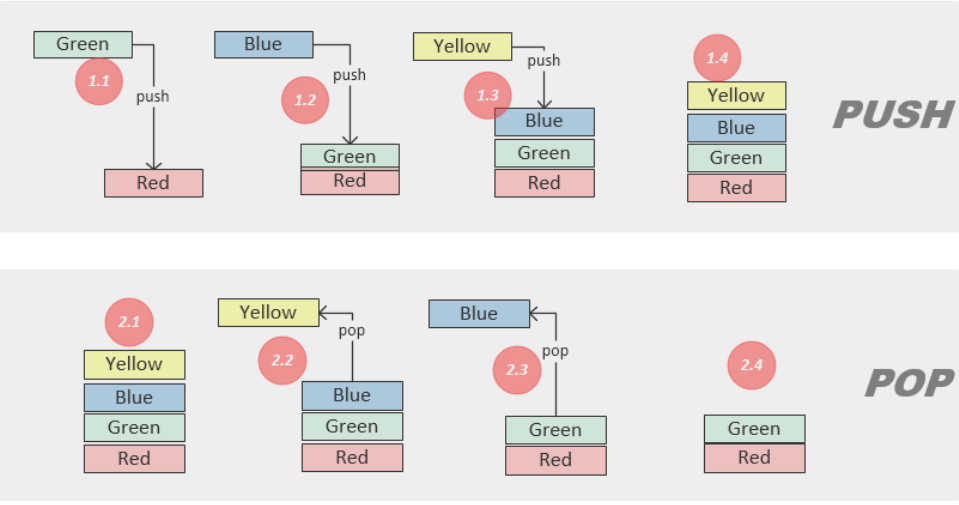


Figure 2.4: Push and pop operations

The top portion of *Figure 2.4* 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

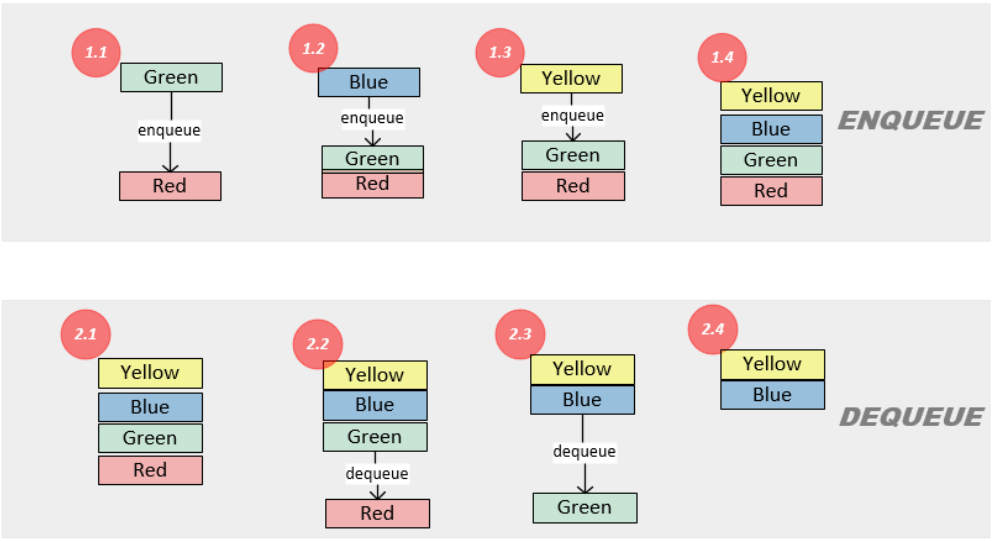
A stack is used as the data structure in many use cases. For example, when a user wants to browse the history in a web browser, it is a LIFO data access pattern, and a stack can be used to store the history. Another example is when a user wants to perform an `Undo` operation in word processing software.

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:



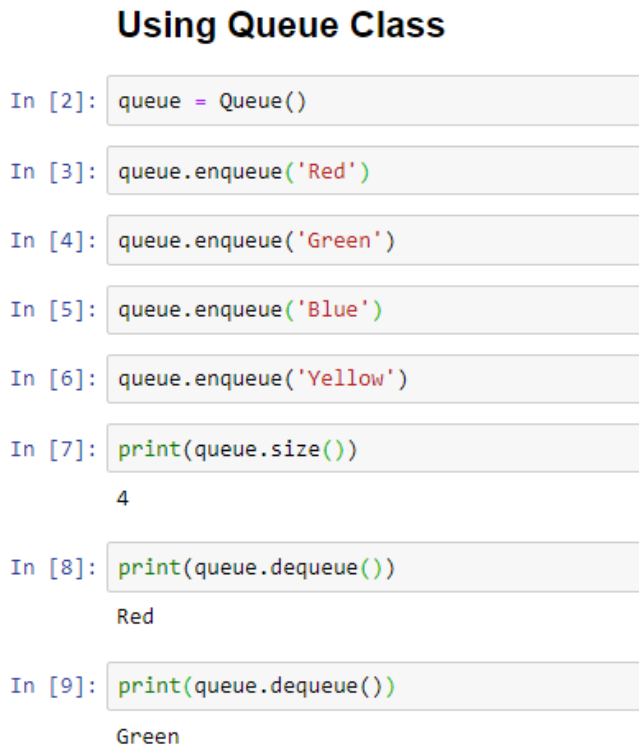
Queue

Figure 2.5: Enqueue and dequeue operations

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:



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	The degree of a node is the number of children it has. For example, in the

of a node	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:

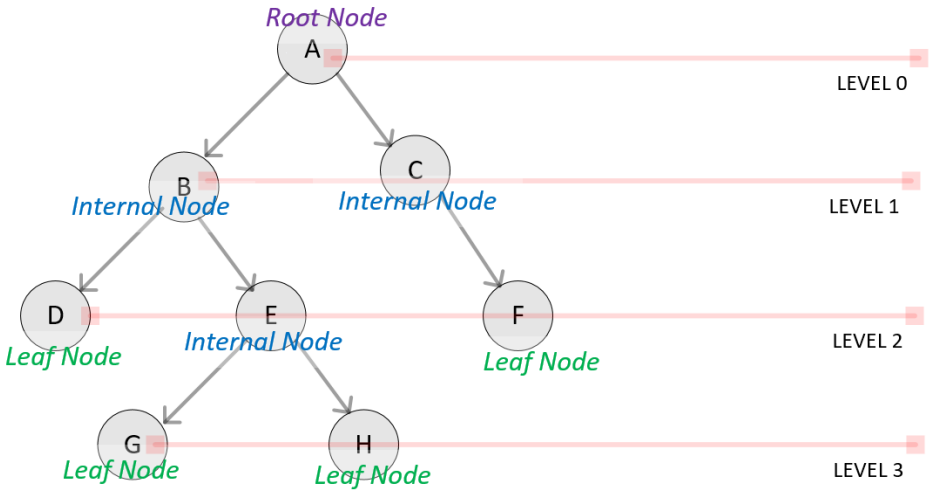


Figure 2.6: A binary tree

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:

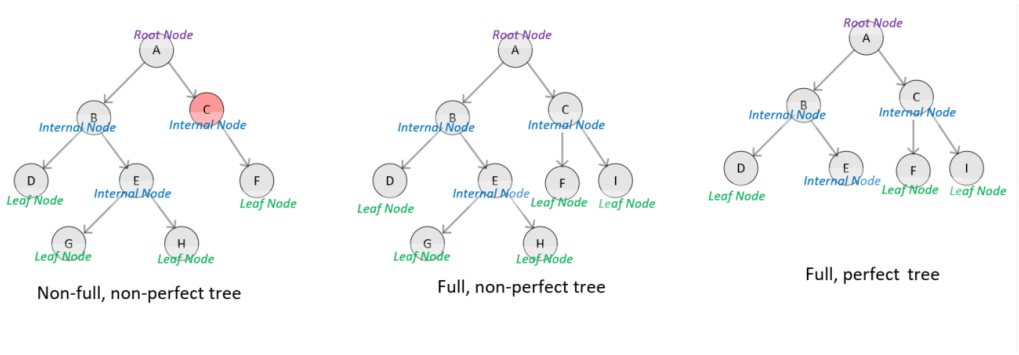


Figure 2.7: A full tree

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, 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.

Join our book's Discord space

Join the book's Discord workspace for a monthly *Ask me Anything* session with the author: https://packt.link/40Algos

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. These include Natural Language Processing (NLP) and patterns extracting algorithms. 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 studied.

By the end of this chapter, we should be able to understand the various algorithms that are used for sorting and searching, and we will be able to apprehend their strengths and weaknesses. As searching and sorting algorithms are the building blocks for many complex algorithms, understanding them in detail will help us better comprehend understand modern complex algorithms as well presented in the later chapters.

The following are the main concepts discussed in this chapter:

- Introducing sorting algorithms
- Introducing searching algorithms
- Performance analysis of sorting and searching algorithms
- Practical applications of sorting and searching

Let's first look at some sorting algorithms.

Introducing sorting algorithms

The ability to efficiently sort and search items in a complex data structure is 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. Thus, carefully analyzing the performance of these algorithms is important.

Sorting algorithms are used extensively in distributed data storage systems such as modern NoSQL databases that enable the cluster and cloud computing architectures. In such data storage systems, data elements need to be regularly sorted and stored so that they can be retrieved efficiently.

The following sorting algorithms are presented in this chapter:

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

But before we present these algorithms, let us first discuss the variable swapping technique in Python that we will be using in the code presented in this chapter.

Swapping variables in Python

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

```
var_1 = 1
var_2 = 2
```

```
var_1,var_2 = var_2,var_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 one of the simplest and slowest algorithms used for sorting. It is designed in a way that the highest value in a list of data *bubbles* its way to the top as the algorithm loops through iterations. Bubble sort requires little runtime memory to run because all the ordering occurs within the original data structure. No new data structures are needed as temporary buffers. But its worst-case performance is $O(N^2)$, which is quadratic time complexity (where N is the number of elements being sorted). As discussed in the following section, it recommended to be used only for smaller datasets. Actual recommended limits for the size of the data for the use of bubble sort for sorting will depend on the memory and the processing resources available but keeping the number of elements(N) below 1000 can be considered as a general recommendation.

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. To understand its working, let's focus on the first iteration: pass one.

The goal of pass one is to push the highest value to the highest index (top of the list). In other words, we will see the highest value of the list *bubbling* its way to the top as pass one progresses.

Bubble sort's logic is based on comparing adjacent neighbor values. If the value at a higher index 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 *Figure 3.1*:

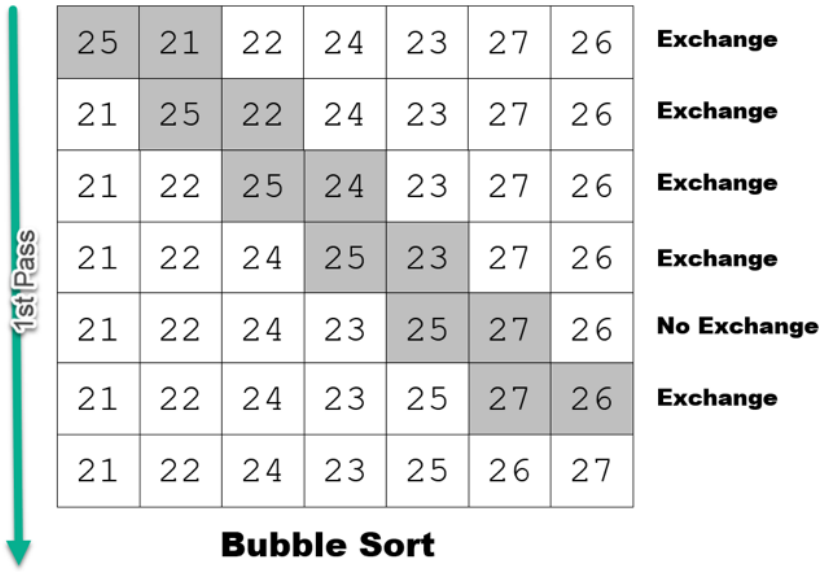


Figure 3.1: Bubble sort algorithm

Let's now see how bubble sort can be implemented using Python If we implement pass one of bubble sort in Python, it will look as follows:

```
last_element_index = len(list)-1
print(0,list)
for idx in range(last_element_index):
                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]
```

Note that after *first pass*:

- the highest value is at the top of the list stored at `idx+1`.
- while executing the *first pass*, the algorithm has to compare each of the elements of the list individually to "bubble" the maximum value to that top.

After completing the *first pass*, the algorithm moves on to the *second pass*. The goal of the *second pass* is to move the second highest value to the second highest index of 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 value at the top index, which was put in the right place by the *first pass*. So, it will have one less data element to tackle.

After completing *second pass*, the algorithm keeps on performing the third pass and subsequent ones 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 implementation of bubble sort in Python is as follows:

We mentioned that performance is one of the limitations of the `bubble sort` algorithm. Let's quantify the performance of `bubble sort` through the performance analysis of the `bubble sort` algorithm:

```
def bubble_sort(list):
# Excahnge the elements to arrange in order
    last_element_index = len(list)-1
    for pass_no in range(last_element_index,0,-1):
        for idx in range(pass_no):
            if list[idx]>list[idx+1]:
                list[idx],list[idx+1]=list[idx+1],list[idx]
    return list
```

Optimizing bubble sort

The above implementation of bubble sort cannot take advantage of fully sorted or partially sorted list. It will need to run though complete inner and outer loops to sort the given list and will have a time complexity of $O(N^2)$ even if a fully sorted list is given as the input. To optimize the bubble sort algorithm, we can check if there is any swapping operation while executing a particular pass (inner loop. If no swapping happens through any pass it means that the given list is now fully sorted, and we can exit. This way we can minimize the number of passes. The following is the implementation of the optimized bubble sort algorithm:

A performance analysis of bubble sort

It is easy 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.

The time complexity of the bubble sort of algorithm is as follows:

- **Best Case**: If the list is already sorted)or almost all elements are sorted) then the runtime complexity is O(1)
- **Worst Case**: If none or very few elements are sorted, then the worst-case runtime complexity is $O(n^2)$ as the algorithm will have to completely run through both inner and the outer loops.

Now let us look into the insertion sort algorithm.

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:

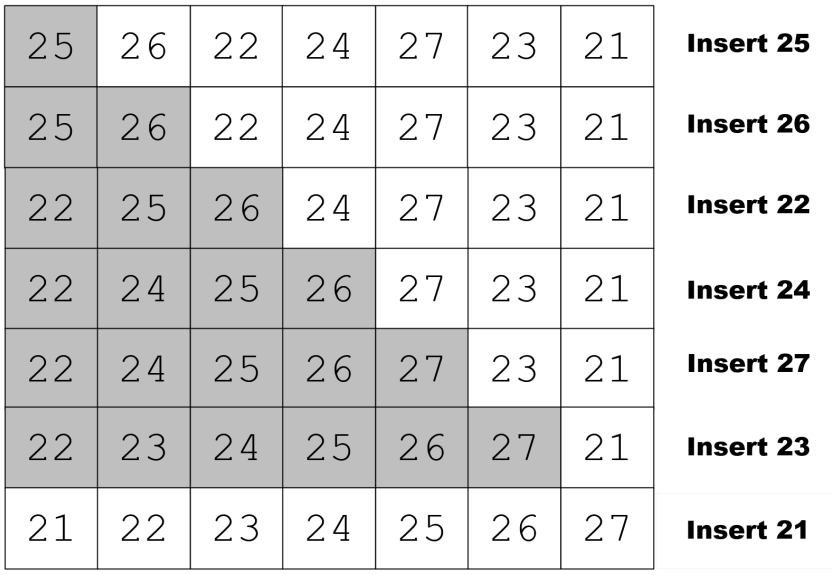


Figure 3.2 Insertion sort algorithm

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

```
def insertion_sort (elements):
    for i in range(1, len(elements)):
        j = i-1
        next = elements[i]
# Compare the current element with next one

        while (elements[j] > next) and (j >= 0):
            elements[j+1] = elements[j]
            j=j-1
```

```
        elements[j+1] = next
    return elements
```

Note that in the main loop, we iterate throughout all of the list. In each iteration, the two adjacent elements are `list[j]` (the current element) and `list[i]` (the next element).

In `list[j] > element_next` and `j >= 0`, we compare the current element with the next element.

Let's look at the performance of the insertion sort algorithm.

A performance analysis of insertion sort

It's obvious from the description of the algorithm that if the data structure is already sorted, insertion sort will perform very fast. In fact, if the data structure is sorted, then the insertion sort will have a linear running time; that is, O(n). The worst case is when each of the inner loops has to move all the elements in the list. If the inner loop is defined by *i*, the worst-case performance of the insertion sort algorithm is given by the following:

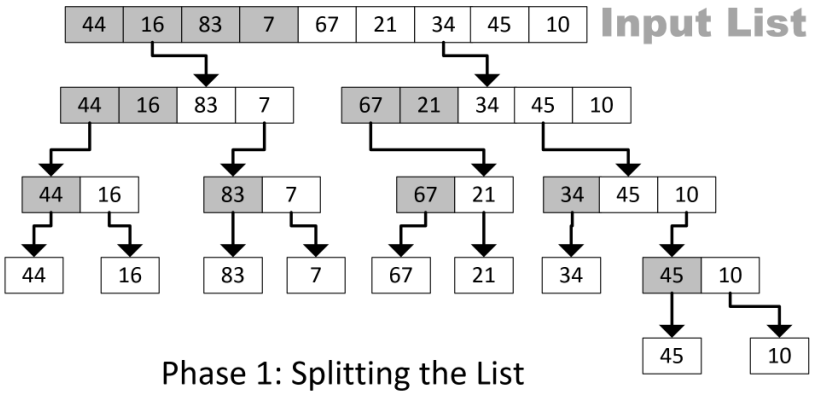
$$w(N) = \sum_{i=1}^{N-1} i = \frac{(N-1)N}{2} = \frac{N^2 - N}{2}$$

$$w(N) \approx \frac{1}{2}N^2 = O(N^2)$$

In general, insertion can be used on small data structures. For larger data structures, insertion sort is not recommended due to quadratic average performance.

Merge sort

We have presented, so far, two sorting algorithms: bubble sort and insertion sort. The performance of both of them will be better if the data is partially sorted. The third algorithm presented in this chapter is the merge sort algorithm, which was developed in 1940 by John von Neumann. The defining feature of this algorithm is that its performance is not dependent on whether the input data is sorted. It works well for the large datasets. Like other big data algorithms, it is based on a divide and conquer strategy. In the first phase, called splitting, the algorithm keeps on dividing the data into two parts recursively, until the size of the data is less than a defined threshold. In the second phase, called **merging**, the algorithm keeps on merging and processing until we get the final result. The logic of this algorithm is explained in the following diagram:



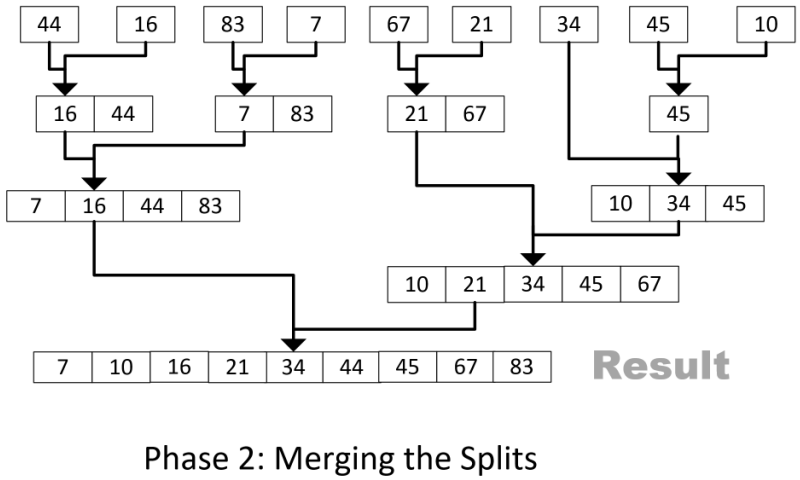


Figure 3.3 Merge Sort Algorithm

Let's first look into the pseudocode of the merge sort algorithm:

```
merge_sort (elements, start, end)
    if(start < end)
        midPoint = (end - start) / 2 + start
        merge_sort (elements, start, midPoint)
        merge_sort (elements, midPoint + 1, start)
        merge(elements, start, midPoint, end)
```

As we can see the algorithm has the following three steps:

1. It divides the input list into two equal parts divided around `midPoint`
2. It uses recursion to split until the length of each list is 1
3. Then, it merges the sorted parts into a sorted list and returns it

The code for implementing `merge_sort` is shown here:

```
 def merge_sort(elements):
    if len(elements)>1:
        mid = len(elements)//2 #splits list in half
        left = elements[:mid]
        right = elements[mid:]
        merge_sort(left)#repeats until length of each list is 1
```

```
        merge_sort(right)
        a = 0
        b = 0
        c = 0
        while a < len(left) and b < len(right):
            if left[a] < right[b]:
                elements[c]=left[a]
                a = a + 1
            else:
                elements[c]=right[b]
                b = b + 1
            c = c + 1
        while a < len(left):
            elements[c]=left[a]
            a = a + 1
            c = c + 1
        while b < len(right):
            elements[c]=right[b]
            b = b + 1
            c = c + 1
    return elements
```

Shell sort

The bubble sort algorithm compares immediate neighbors and exchanges them if they are out of order. On the other hand, insertion sort creates the sorted list by transferring one element at a time. If we have a partially sorted list, insertion sort should give reasonable performance.

But for a totally unsorted list, sized *N*, you can argue that bubble sort will have to fully iterate through *N-1* passes in order to get it fully sorted.

Donald Shell proposed Shell sort (named after him), which questions the importance of selecting immediate neighbors for comparison and swapping.

Now, let's understand this concept.

In pass one, instead of selecting immediate neighbors, we use elements that are at a fixed gap, eventually sorting a sublist consisting of a pair of data points. This is shown in the following diagram. In pass two, it sorts sublists containing four data points (see the following diagram). In subsequent passes, the number of data points per sublist keeps on increasing and the number of sublists keeps on decreasing until we reach a situation where there is just one sublist that consists of all the data points. At this point, we can assume that the list is sorted:

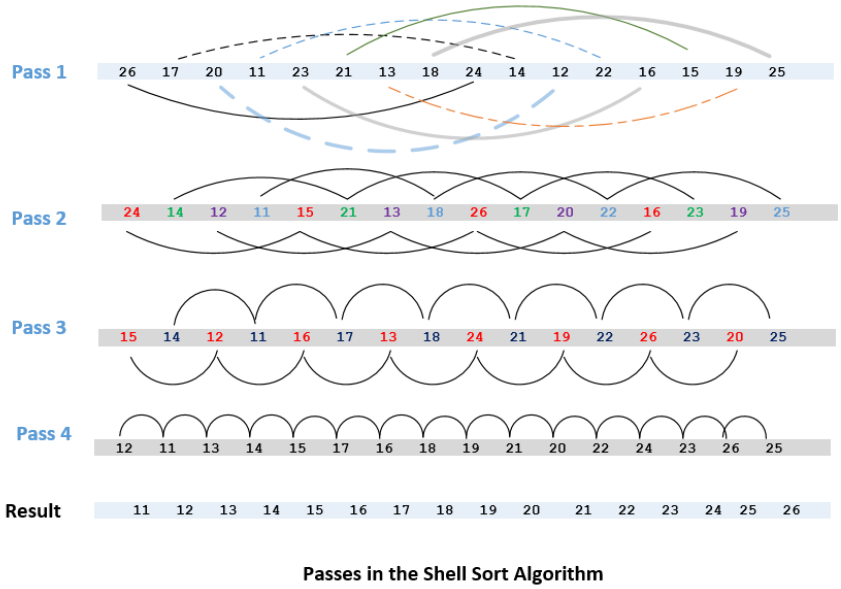


Figure 3.4 Passes in the Shell Sort Algorithm

In Python, the code for implementing the Shell sort algorithm is as follows:

```
def shell_sort(elements):
    distance = len(elements) // 2
    while distance > 0:
        for i in range(distance, len(elements)):
            temp = elements[i]
            j = i
# Sort the sub list for this distance
            while j >= distance and elements[j - distance] > temp:
                list[j] = elements[j - distance]
                j = j-distance
            list[j] = temp
# Reduce the distance for the next element
        distance = distance//2
    return elements
```

Note that calling the `ShellSort` function has resulted in sorting the input array.

A performance analysis of Shell sort

It can be observed that in the worst case, the Shell sort algorithm will have to run through both loops giving it a complexity of $O(n^2)$. Shell sort is not for big data. It is used for medium-sized datasets. Roughly speaking, it has a reasonably good performance on a list with up to 6,000 elements. If the data is partially in the correct order, the performance will be better. In a best-case scenario, if a list is already sorted, it will only need one pass through *N* elements to validate the order, producing a best-case performance of *O(N)*.

Selection sort

As we saw earlier in this chapter, bubble sort is one of the simplest sorting algorithms. Selection sort is an improvement on bubble sort, where we try to minimize the total number of swaps required with the algorithm. It is designed to make one swap for each pass, compared to *N-1* passes with the bubble sort algorithm. Instead of bubbling the largest value toward the top in baby steps (as done in bubble sort, resulting in *N-1* swaps), we look for the largest value in each pass and move it toward the top. So, after the first pass, the largest value will

be at the top. After the second pass, the second largest value will be next to the top value. As the algorithm progresses, the subsequent values will move to their correct place based on their values. The last value will be moved after the *(N-1)*th pass. So, selection sort takes *N-1* passes to sort *N* items:

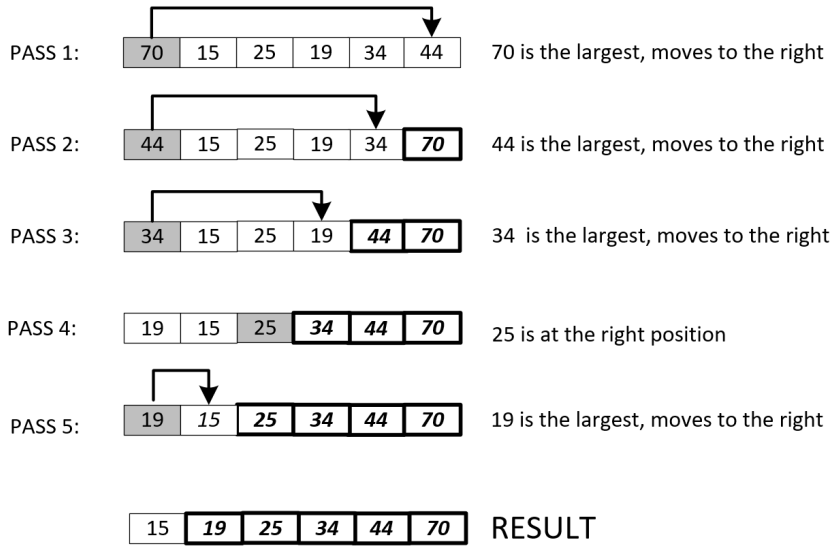


Figure 3.5: Selection Sort Algorithm

The implementation of selection sort in Python is shown here:

```
def selection_sort(list):
    for fill_slot in range(len(list) - 1, 0, -1):
        max_index = 0
        for location in range(1, fill_slot + 1):
            if list[location] > list[max_index]:
                max_index = location
        list[fill_slot],list[max_index] = list[max_index],list[fill_slot]
    return list
```

The performance of the selection sort algorithm

Selection sort's worst-case performance is *O(N2)*. Note that its worst performance is similar to bubble sort, and it should not be used for sorting larger datasets. Still, selection sort is a better designed algorithm than bubble sort and its average performance is better than bubble sort due to the reduction in the number of exchanges.

Choosing a sorting algorithm

The choice of the right sorting algorithm depends both on the size and the state of the current input data. For small input lists that are sorted, using an advanced algorithm will introduce unnecessary complexities to the code, with a negligible improvement in performance. For example, we do not need to use merge sort for smaller datasets. Bubble sort will be way easier to understand and implement. If the data is partially sorted, we can take advantage of that by using insertion sort. For larger datasets, the merge sort algorithm is the best one to use.

Introduction to searching algorithms

Efficiently searching data in complex data structures is one of the most important functionalities. The simplest approach, which will not be that efficient, is to search for the required data in each data point. But, as the data

becomes bigger in size, we need more sophisticated algorithms designed for searching data.

The following searching algorithms are presented in this section:

- Linear search
- Binary search
- Interpolation search

Let's look at each of them in more detail.

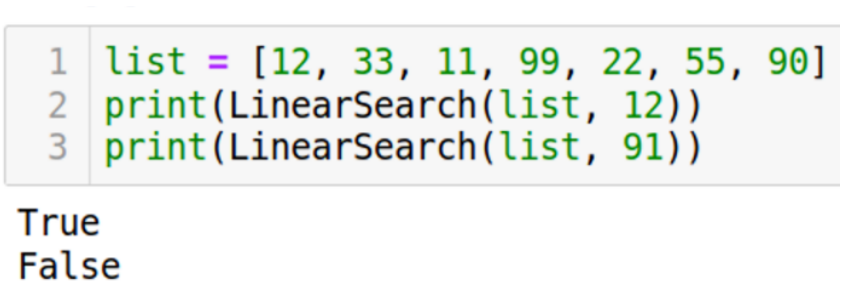
Linear search

One of the simplest strategies for searching data is to simply loop through each element looking for the target. Each data point is searched for a match and when a match is found, the results are returned, and the algorithm exits the loop. Otherwise, the algorithm keeps on searching until it reaches the end of the data. The obvious disadvantage of linear search is that it is very slow due to the inherent exhaustive search. The advantage is that the data does not need to be sorted, as required by the other algorithms presented in this chapter.

Let's look at the code for linear search:

```
def linear_search(elements, item):
    index = 0
    found = False
# Match the value with each data element
    while index < len(elements) and found is False:
        if elements[index] == item:
            found = True
        else:
            index = index + 1
    return found
```

Let's now look at the output of the preceding code:



Note that running the `LinearSearch` function returns a `True` value if it can successfully find the data.

The performance of linear search

As discussed, linear search is a simple algorithm that performs an exhaustive search. Its worst-case behavior is *O(N)*. More info can be found at maybe you could also point the reader to this page: https://wiki.python.org/moin/TimeComplexity.

Binary Search

The pre-requisite of the binary search algorithm is sorted data. The algorithm iteratively divides a list into two parts and keeps a track of the lowest and highest indices until it finds the value it is looking for:

```
def binary_search(elements, item):
    first = 0
    last = len(elements)-1
    while first<=last and not found:
        midpoint = (first + last)//2
        if list[midpoint] == item:
            return True
        else:
            if item < elements[midpoint]:
                last = midpoint-1
            else:
                first = midpoint+1
    return False
```

The output is as follows:

```
[48]: list = [12, 33, 11, 99, 22, 55, 90]
      sorted_list = BubbleSort(list)
      print(BinarySearch(list, 12))
      print(BinarySearch(list, 91))

      True
      False
```

Note that calling the `BinarySearch` function will return `True` if the value is found in the input list.

The performance of binary search

Binary search is so named because at each iteration, the algorithm divides the data into two parts. If the data has *N* items, it will take a maximum of *O(logN)* steps to iterate. This means that the algorithm has an *O(logN)* runtime.

Interpolation search

Binary search is based on the logic that it focuses on the middle section of the data. Interpolation search is more sophisticated. It uses the target value to estimate the position of the element in the sorted array. Let's try to understand it by using an example. Let's assume we want to search for a word in an English dictionary, such as the word river. We will use this information to interpolate and start searching for words starting with r. A more generalized interpolation search can be programmed as follows:

```
def int_polsearch(list,x ):
    idx0 = 0
    idxn = (len(list) - 1)
    while idx0 <= idxn and x >= list[idx0] and x <= list[idxn]:
# Find the mid point
        mid = idx0 +int(((float(idxn - idx0)/( list[idxn] - list[idx0])) * ( x - list[idx0])))
# Compare the value at mid point with search value
        if list[mid] == x:
            return True
        if list[mid] < x:
            idx0 = mid + 1
    return False
```

The output is as follows:

```
In [16]:
1 list = [12, 33, 11, 99, 22, 55, 90]
2 sorted_list = BubbleSort(list)
3 print(IntPolsearch(list, 12))
4 print(IntPolsearch(list,91))

True
False
```

Note that before using `IntPolsearch`, the array first needs to be sorted using a sorting algorithm.

The performance of interpolation search

If the data is unevenly distributed, the performance of the interpolation search algorithm will be poor. The worst-case performance of this algorithm is *O(N)* and if the data is somewhat reasonably uniform, the best performance is *O(log(log N))*.

Practical applications

The ability to efficiently and accurately search data in a given data repository is critical to many real-life applications. Depending on your choice of searching algorithm, you may need to sort the data first as well. The choice of the right sorting and searching algorithms will depend on the type and the size of the data, as well as the nature of the problem you are trying to solve.

Let's try to use the algorithms presented in this chapter to solve the problem of matching a new applicant at the immigration department of a certain country with historical records. When someone applies for a visa to enter the country, the system tries to match the applicant with the existing historical records. If at least one match is found, then the system further calculates the number of times that the individual has been approved or refused in the past. On the other hand, if no match is found, the system classes the applicant as a new applicant and issues them a new identifier. The ability to search, locate, and identify a person in the historical data is critical for the system. This information is important because if someone has applied in the past and the application is known to have been refused, then this may affect that individual's current application in a negative way. Similarly, if someone's application is known to have been approved in the past, this approval may increase the chances of that individual getting approval for their current application. Typically, the historical database will have millions of rows, and we will need a well-designed solution to match new applicants in the historical database.

Let's assume that the historical table in the database looks like the following:

Personal ID	Application ID	First name	Surname	DOB	Decision	Decision date
45583	677862	John	Doe	2000-09-19	Approved	2018-08-07
54543	877653	Xman	Xsir	1970-03-10	Rejected	2018-06-07
34332	344565	Agro	Waka	1973-02-15	Rejected	2018-05-05
45583	677864	John	Doe	2000-09-19	Approved	2018-03-02
22331	344553	Kal	Sorts	1975-01-02	Approved	2018-04-15

In this table, the first column, `Personal ID`, is associated with each of the unique applicants in the historical database. If there are 30 million unique applicants in the historical database, then there will be 30 million unique personal IDs. Each personal ID identifies an applicant in the historical database system.

The second column we have is `Application ID`. Each application ID identifies a unique application in the system. A person may have applied more than once in the past. So, this means that in the historical database, we will have more unique application IDs than personal IDs. John Doe will only have one personal ID but has two application IDs, as shown in the preceding table.

The preceding table only shows a sample of the historical dataset. Let's assume that we have close to 1 million rows in our historical dataset, which contains the records of the last 10 years of applicants. New applicants are continuously arriving at the average rate of around 2 applicants per minute. For each applicant, we need to do the following:

- Issue a new application ID for the applicant.
- See if there is a match with an applicant in the historical database.
- If a match is found, use the personal ID for that applicant, as found in the historical database. We also need to determine that how many times the application has been approved or refused in the historical database.
- If no match is found, then we need to issue a new personal ID for that individual.

Suppose a new person arrives with the following credentials:

- `First Name: John`
- `Surname: Doe`
- `DOB: 2000-09-19`

Now, how can we design an application that can perform an efficient and cost-effective search?

One strategy for searching the new application in the database can be devised as follows:

- Sort the historical database by `DOB`.
- Each time a new person arrives, issue a new application ID to the applicant.
- Fetch all the records that match that date of birth. This will be the primary search.
- Out of the records that have come up as matches, perform a secondary search using the first and last name.
- If a match is found, use `Personal ID` to refer to the applicants. Calculate the number of approvals and refusals.
- If no match is found, issue a new personal ID to the applicant.

Let's try choosing the right algorithm to sort the historical database. We can safely rule out bubble sort as the size of the data is huge. Shell sort will perform better, but only if we have partially sorted lists. So, merge sort may be the best option for sorting the historical database.

When a new person arrives, we need to locate and search that person in the historical database. As the data is already sorted, either interpolation search or binary search can be used. Because applicants are likely to be equally spread out, as per `DOB`, we can safely use binary search.

Initially, we search based on `DOB`, which returns a set of applicants sharing the same date of birth. Now, we need to find the required person within the small subset of people who share the same date of birth. As we have successfully reduced the data to a small subset, any of the search algorithms, including bubble sort, can be used to search for the applicant. Note that we have simplified the secondary search problem here a bit. We also need to calculate the total number of approvals and refusals by aggregating the search results, if more than one match is found.

In a real-world scenario, each individual needs to be identified in the secondary search using some fuzzy search algorithm, as the first and last names may be spelled slightly differently. The search may need to use some kind of distance algorithm to implement the fuzzy search, where the data points whose similarity is above a defined threshold are considered the same.

Summary

In this chapter, we presented a set of sorting and searching algorithms. We also discussed the strengths and weaknesses of different sorting and searching algorithms. We quantified the performance of these algorithms and learned when to use each algorithm.

In the next chapter, we will study dynamic algorithms. We will also look at a practical example of designing an algorithm and the details of the page ranking algorithm. Finally, we will study the linear programming algorithm.

Join our book's Discord space

Join the book's Discord workspace for a monthly *Ask me Anything* session with the author:
https://packt.link/40Algos