Data Structures with Python

Get familiar with the common Data Structures and Algorithms in Python

Dr. Harsh Bhasin

www.bpbonline.com

First published: 2023

Published by BPB Online
WeWork
119 Marylebone Road
London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-55513-304

www.bpbonline.com

Dedicated to

My mother

About the Author

Dr. Harsh Bhasin is a researcher and practitioner. Dr. Bhasin is currently associated with the Center of Health Innovations, Manav Rachna Institutions. Dr. Bhasin has completed his Ph. D. in Mild Cognitive Impairment from Jawaharlal Nehru University, New Delhi. He worked as a Deep Learning consultant for various firms and taught at various Universities including Jamia Hamdard and DTU.

He has authored 11 books including Programming in C#, Oxford University Press, 2014; Algorithms, Oxford University Press, 2015; Python for Beginners, New Age International, 2018; Python Basics, Mercury, 2019; Machine Learning, BPB, 2020, to name a few.

Dr. Bhasin has authored 40 papers published in renowned journals including Alzheimer's and Dementia, Soft Computing, BMC Medical Informatics & Decision Making, AI & Society, etc. He is the reviewer of a few renowned journals and is the editor of a few special issues. He is a recipient of a distinguished fellowship.

His areas of expertise include Deep learning, Algorithms, and Medical Imaging. Outside work, he is deeply interested in Hindi Poetry: the progressive era, and Hindustani Classical Music: percussion instruments.

About the Reviewer

Sumeet Lalla has done his masters of Data Science from Higher School Of Economics Moscow and Bachelors of Engineering in Computer Engineering from Thapar University. He also has 6 years of experience in Data Science and Software Engineering. His career graph includes working as a Data Scientist in Cognizant and as a Software Developer in Siemens Technology, along with working in Services and Technology Analyst in Deloitte Consulting and Pvt Ltd.

Acknowledgement

"Feeling gratitude and not expressing it is like wrapping a present and not giving it."

– William Arthur Ward

I am blessed to have met people who encouraged me to learn continuously. First of all, I would like to thank Professor Moin Uddin, former Pro-Vice-Chancellor, Delhi Technological University for his unconditional support. He has deposed his faith in me when no one else did. Had it not been for his encouragement I would not have been able to achieve whatever I did.

I would also like to thank Professor I. K. Bhat, Vice Chancellor, Manav Rachna University, India; and Professor Sameer Singh, Rail Vision, United Kingdom; for their continuous support and encouragement. I would also like to express my sincere gratitude to the late Professor A. K. Sharma, former Dean, and Chairperson, the Department of Computer Science, YMCA, Faridabad, for his constant encouragement. I have been able to write this book, author papers, and work on projects only because of the encouragement provided by him. I would also like to thank Professor Naresh Chauhan, former Head and Chairperson, Department of Computer Science, YMCA University of Science and Technology, and Dr. S. K. Pal, Scientist, Department of Defence and Research Organization, Govt. of India for their constant support.

I am thankful to Mr. Nishant Kumar, NCU, India, for his contribution to editing, formatting, and developing some programs for this book. I would also like to thank my students and colleagues, for their critical reviews. I am also very thankful to the editorial team of BPB Publications for providing valuable assistance.

I would like to express my sincere gratitude to my Mother, Sister, and the rest of the family including my pets: Zoe & Xena, and friends for their unconditional support to me.

I would be glad to receive your comments or suggestions which can be incorporated into future editions of the book.

Preface

This book introduces the reader to Data Structures and Algorithms, the foundation stone of programming. The concepts discussed in this book will help the reader to understand various data structures, analyze the time and space complexity, and use these data structures for solving graded problems.

The first chapter introduces the reader to the fascinating world of Algorithms and Data Structures. The idea of complexity has been introduced in the chapter. It contains ample examples of finding the complexity of a given algorithm.

The next chapter takes the reader through various approaches to developing algorithms. The chapter introduces the Greedy approach, divide and conquer, dynamic programming, and backtracking. An introduction to branch and bound has also been included in the chapter.

Recursion has been introduced in the third chapter of this book. The chapter contains the mechanism, examples, and the process of finding the complexity of a recursive algorithm. The problems related to arrays have been discussed in the fourth chapter of this book. It presents insertion, deletion, and searching in arrays along with the complexities.

Chapter 5 discusses Linked Lists. The algorithms, complexity, and problems of linked lists have been covered in this chapter. This chapter forms the basis of the following chapters. The next two chapters introduce stacks and queues. The applications of these data structures have also been included in the chapters. These chapters contain assorted problems related to stacks and queues.

Chapters 8 and 9 deal with trees and heaps. The insertion and deletion in binary trees and other algorithms have been included in these chapters. Chapter ten introduces priority queues. The next chapter discusses graphs. It contains traversal algorithms, spanning tree algorithms, and shortest path algorithms.

Chapter 11 contains eleven sorting techniques and discusses the related problems. selection has been dealt with in the next chapter. A very efficient searching technique called hashing has been explained in detail in the fourteen chapters. The last chapter deals with the String algorithms.

The book also contains four appendices containing Dijkstra's algorithm, all pairs' shortest path, and tree traversals using stacks and problems.

Coloured Images

Please follow the link to download the
Coloured Images of the book:

https://rebrand.ly/6rz6cpm

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Piracy

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

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

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

For more information about BPB, please visit **www.bpbonline.com**.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Table of Contents

Chapter 1
Introduction to Data Structures

Richard Buckland from the University of New South Wales often uses the acronym PAPP while teaching Data Structures and Algorithms. Here, the first P stands for the problem, the A stands for the algorithm, the second P for the program, and the third P for the process. An algorithm is the sequence of steps to accomplish the task at hand or solve the problem. The algorithm is then implemented in some programming language, thus, giving rise to a program. When you execute this program, it becomes a process.

You learn languages, say Python, to implement an algorithm;this takes us from the algorithm to the program. You execute this program, thus, creating a process. To understand the transition from a program to a process, you learn the basics of Compiler Design and Operating Systems. In this chapter, you will learn the transition from Algorithm to Program and to some extent, the problem to the algorithm. That is, you will be presented with algorithms related to some data structures, which you will implement. Furthermore, at times, you will be presented with some problems that you need to solve using the implemented data structures. This chapter defines the term data structure and presents a brief overview of the things to come.

The reader is expected to know Python;for that matter, he/she must be versed in at least one programming language. The following discussion will extensively use loops, nested loops, lists, tuples, dictionaries, arrays (both 1-dimensional and 2-dimensional), and the difference between reference types and value types.

Structure

In this chapter, we will cover the following topics:

- Define data structures
- Define data type
- Classify data structures
- Learn a way to sort numbers

Objectives

This chapter aims to introduce the fascinating subject of data structures to the readers. The chapter will deal with the basic data types, types of data structures, and a problem related to sorting.

After reading this chapter, the reader will be able to classify data structures and understand why this study is important. The reader will also learn when to use which data structure and what operations can be performed on them. This discussion will act as a foundation stone of the building called data structures.

Introduction

The single value stored in a variable is called a datum. The set of values of a variable is called data. Note that data may contain many values, each of which is referred to as a data item. Each data item can further be divided into sub-items. For example, a variable called **name**, which stores the name of a student, may have the value "Brandon Walsh". This data item can be divided into two sub-items, "Brandon" and "Walsh", which are the first name and the last name, respectively, though some data items like PAN CARD NUMBER cannot be divided into sub-items.

Some of you might have studied Object Oriented Programming and have some basic idea of a "Class", which is a real or a conceptual entity having importance to the problem at hand. An entity has attributes, each of which can be assigned some values and these values may belong to a particular data type. For example, in the following illustration, an entity called Movie has attributes name, year, genre, and so on. The data type of Name is a string, that of Year is an integer, and that of Genre, Director, and Music are strings. The values assigned to these variables are "Sairat", 2016, "Don't talk about it", "Nagraj Manjule", and "Ajay-Atul" respectively.

Movie

Name	:	Sairat
Year	:	2016

Genre	:	Don't talk about it
Director	:	Nagraj Manjule
Music	:	Ajay-Atul

The set of similar entities constitutes an entity set, which will help us to solve the preceding problem. This takes us toward meaningful data, which can be processed. Here comes information that can be considered as processed organized data. The organization is important, and this subject will teach you how to organize data.

Let us consider a file containing records of movies. Each record contains five fields: Name, Year, Genre, Director, and Music. So, we have a file containing records, and each record contains fields. This is an example of fixed-length records. There are files containing variable-length records as well. In such files, the maximum and minimum length is generally mentioned. The data needs to be organized to facilitate efficient and effective handling of this data. This subject teaches you data structures, which will help you to organize data efficiently and effectively.

Data structures: Data structures include the organization of records into complex structures, the implementation of such structures and the analysis of the amount of memory and time taken by such structures.

Having seen the definition of data structures, let us now move to the definition of data types.

Data types

"*The data types constraints values that a variable can take and define operations that can be performed on it.*" The basic data types such as int, char, and float are also called primitive data types. In older versions of C, for example, an integer is used to take two bytes of memory (16 bits). Out of these, one bit was reserved for the sign of the integer, and the remaining 15 bits were for storing the values. Note that the maximum value can therefore be 2^{15} - 1 = 32767 and the minimum value could be -2^{15} = -32768. Likewise, in the versions of C, which allot 4 bytes to integers, the maximum value can be 2^{31} - 1 and the minimum can be -2^{31} .

The **int** data type, therefore, constrains a) the types of values that can be stored in an integer type variable b) the range of values that can be stored in such a variable, and c) the organization of memory (as in one bit will store the sign information and the rest binary equivalent of a given number).

The primitive data types are provided by the programming language, such as integer, character, float, Boolean and so on. The amount of memory allocated to each depends on the language and some other factors. For example, an **int** in C occupies two bytes in Turbo (DOS) and four in Visual Studio.

Most programming languages also allow user-defined data types as well. In Object Oriented Languages, classes provide a way to create user-defined data types. In C, structures can be used to create these.

Let us now have a brief overview of the types of Data Structures. The following section will give you a glimpse of things to come.

Types of data structures

This subject primarily focuses on the organization of data. This organization can be modeled in different ways, each of which is referred to as a data structure. For example, a set of ordered numbers can be placed in a linear array or in a tree-like structure. The first method is easy but may require more time for some operations like insertion and deletion, whereas the second method though slightly complex, will help in easy insertion and deletion.

Any model that you choose must be representative of the relationship between the data members, and the structure should be simple, effective, and efficient. Some of the data structures that will be explored in the following chapters are as follows:

- **Arrays**: An array is one of the simplest data structures. It is homogeneous, and the elements are stored at consecutive memory locations. The elements of the array will be represented by the name of the array followed by square brackets ([ ]) containing the index of the element. Note that the first element is generally placed at index zero, the second at index one, and so on.
- **Two-dimensional arrays**: Two-dimensional arrays are table-like structures containing rows and columns. The elements in these matrix-like structures can be accessed by the name of the array, followed by the two indices depicting the row index and the column index. Generally, in memory, the two-dimensional arrays are stored in the row-major or column-major format, as discussed in *Chapter 4, Arrays*.
- **Linked list**: The linked list is a data structure in which the units called nodes are linked together. Each of these nodes contains at least two parts normally: (a) the information and (b) the address of the next node. The address of the last node is null/none. We may also have more than one container for the address in each node, like in the case of a doubly linked list. In a doubly linked list, each node contains the address of the previous node, information and that of the next node.

- **Stacks**: Stacks is a linear data structure that follows the principle of **Last in first out** (**LIFO**) or **First in last out** (**FILO**). So, if you insert 51, 78, 90, and 49 in a stack, the order in which these elements would be removed is 49, 90, 78, and 51 (which is LIFO). Stacks is extremely useful while implementing recursion and in evaluating various types of expressions such as postfix, prefix, and so on.
- **Queue**: Queue is a linear data structure that follows the principle of **First in first out** (**FIFO**). So, if you insert 51, 78, 90, and 49 in a queue, the order in which these elements would be removed is 51, 78, 90 and 49 (which is First In First Out). Queues are used in the implementation of scheduling algorithms such as FIFO and so on.
- **Graph**: Graph is a set containing {V, E}, where V is a finite non-empty set of vertices, and E is a finite non-empty set of edges. They are used in almost all facets of Computer Science, such as circuits and systems, networking, and so on.
- **Trees**: Trees generally contain nodes depicting hierarchical relationships. Technically, it is a graph that does not contain a cycle, isolated vertex, or isolated edge(s). Such data structure greatly helps us in searching and even sorting.

The operations that can be performed on each of the data structures are as follows:

- **Traversal**: A traversal defines a way to visit each element of a given data structure. A graph, for example, can be traversed using Depth First Search, Breadth First Search, Level First Search, and so on. A binary tree can be traversed using In-order, Post-order, or Pre-order traversal, and so on.
- **Insertion**: The process of inserting a node in a given data structure is important both in terms of time complexity and memory management. For example, inserting an element at the end of an array takes O(n) time, whereas inserting an element in a stack takes O(1) time.
- **Deletion**: Like in the case of insertion, the deletion of an element is important both in terms of time and space complexity. For example, deleting an element from the end of an array takes O(1) time, whereas deleting an element from the beginning of an array takes O(n) time.
- **Searching**: This is one of the most important operations in data structures. In fact, many data structures are designed so that an item can be efficiently searched from them.

Let us now have a look at one of the most important problems in data structures: Sorting.

Game of clones

Consider the following situation. You have five people (all the same: clones?) with placards having a number written on them. All the cards have different numbers. The five people are assigned numbers indicating their positions, which are from 1 to 5 (*figure 1.1*). They start playing a game as per the following rules.

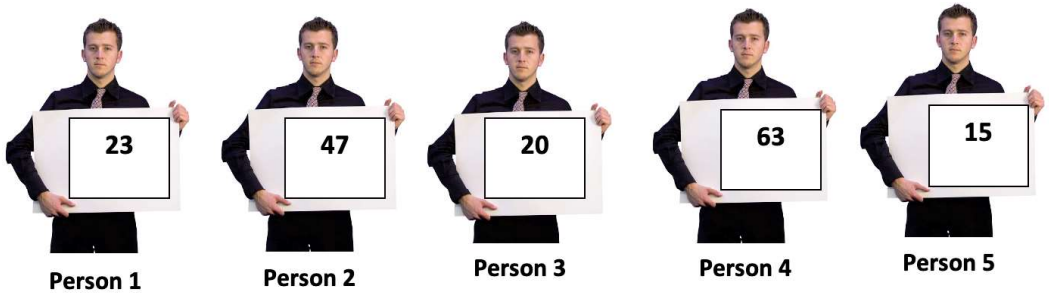

***Figure 1.1**: Five persons holding different placards*

Starting from i=1 (till i=4), person i and (i+1) exchange their cards if the number on i's card is greater than (i+1)'s card. That is, person 1 will exchange the card with person 2 if person 1's card has a number greater than person 2's card; then person 2 will exchange the card with person 3 if person 2's card has a number greater than person 3's card, person 3 will exchange the card with person 4 if person 3's card has a number greater than person 4's card, and finally, person 4 will exchange the card with person 5 if person 4's card has a number greater than person 5's card. This way, person 5 will have the card having the greatest number after this step (*figure 1.2*):

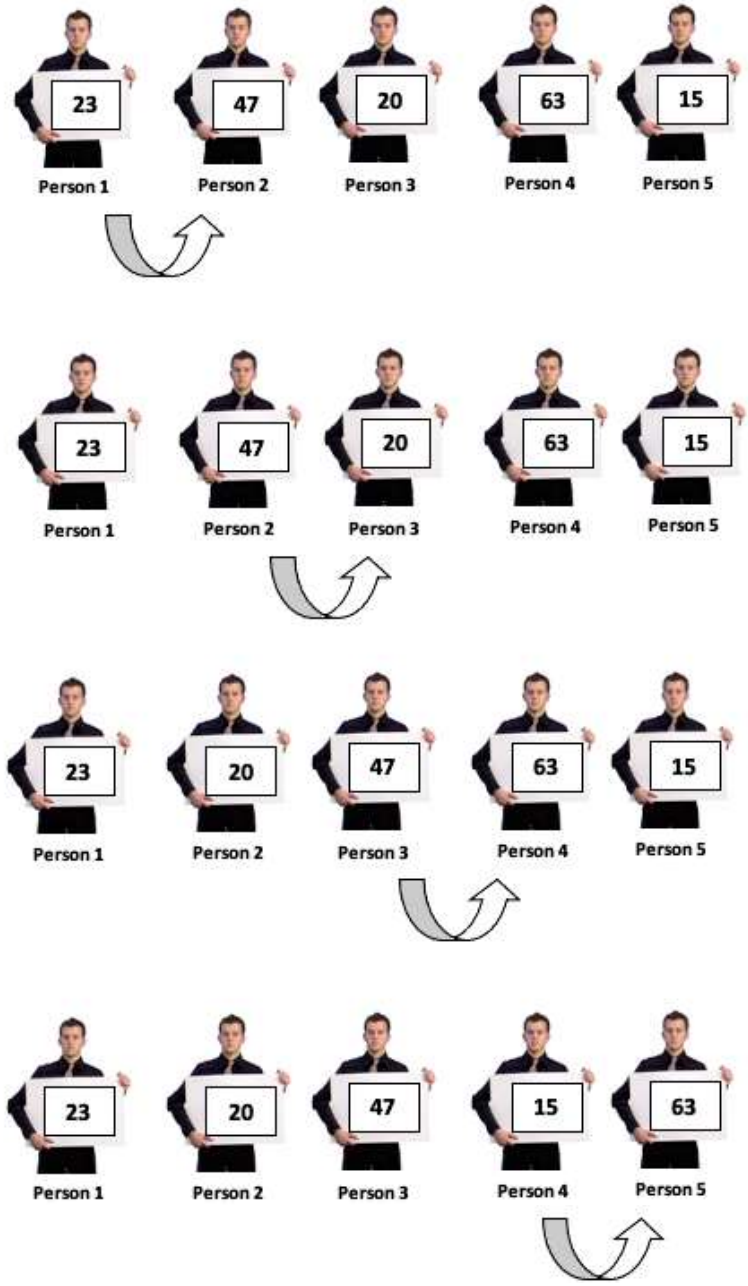

***Figure 1.2**: At the end of Step 1, person 5 will have the card having the largest number*

In the next step, the preceding process is repeated, except for the exchange between persons 4 and 5, as person 5 already has the greatest number. After this, Step 4 will have the second largest number (*figure 1.3*):

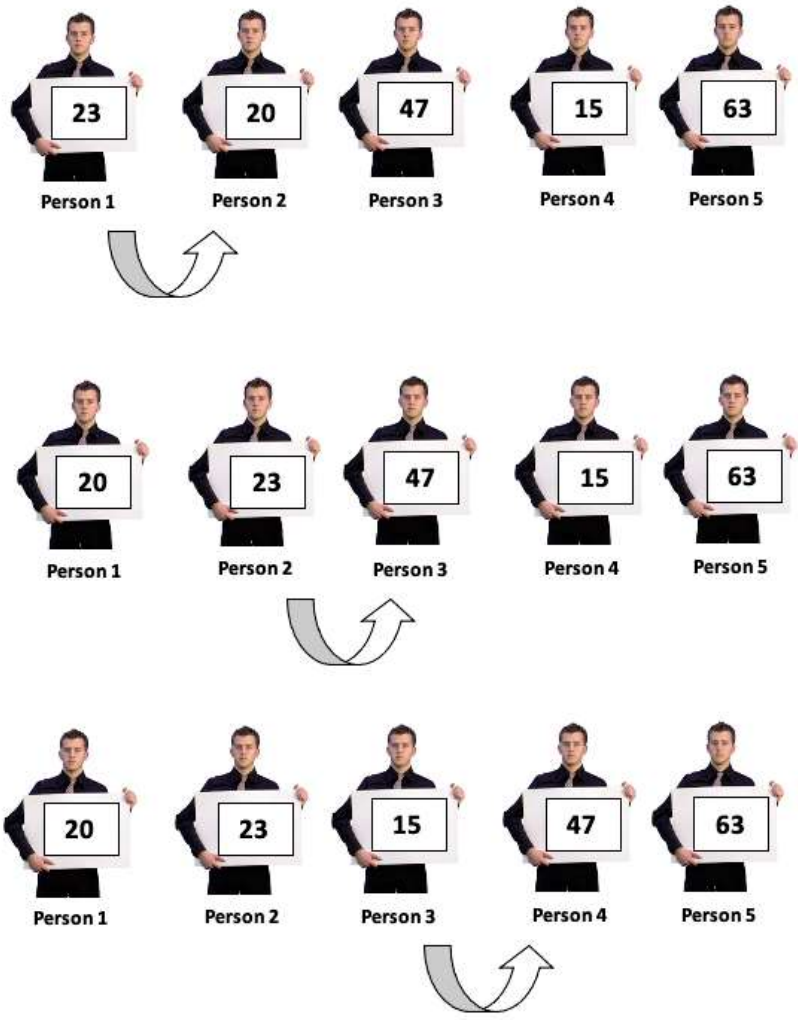

***Figure 1.3**: At the end of Step 2, person 4 will have the card having the second largest number*

In the next step, the preceding process is repeated till all the numbers are sorted (*figure 1.4*):

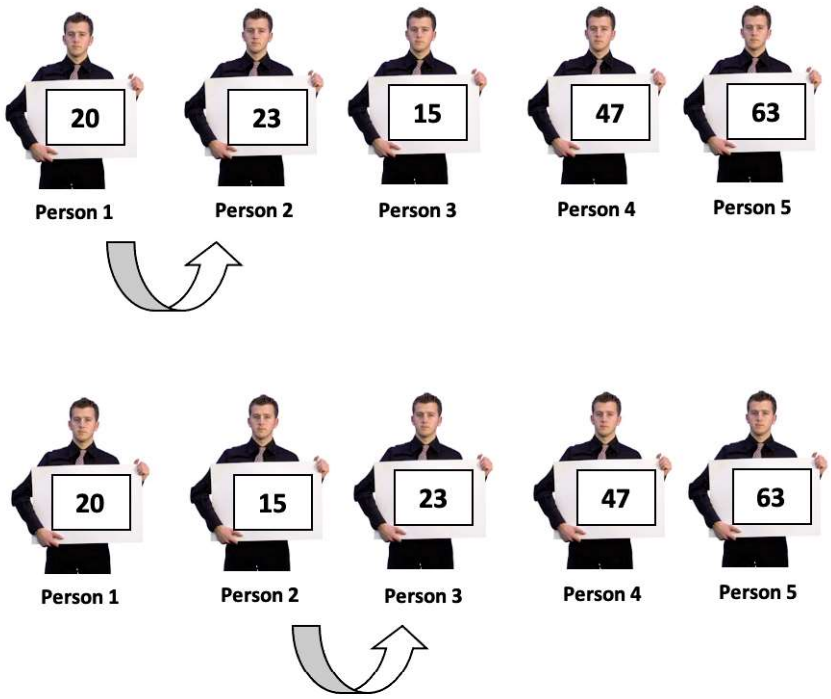

***Figure 1.4**: At the end of Step 3, person 3 will have the card having the third largest number*

Note that in the last step, only a single comparison is required (*figure 1.5*):

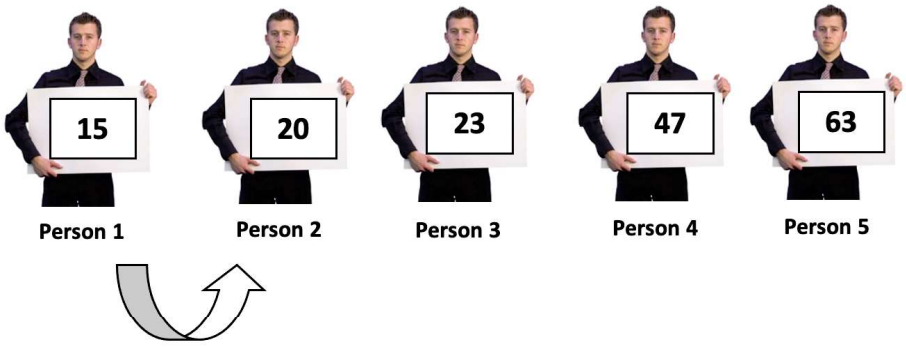

***Figure 1.5**: At the end of Step 4, person 2 will have the card having the fourth largest number*

Note that after each step, the largest element of the remaining array is placed at the $(i+1)^{th}$ last position. Here, i varies from 0 to (n–1), n being the number of terms. The complete process results in the sorted array. Note that in the preceding process, the total number of comparisons is 4 + 3 + 2 + 1 = 10. Had there been *n* numbers, the total of comparisons would have been (n-1) + (n-2)+ … + 1 = (n × (n-1))/2. That is of order n^2. The process scales very poorly. When the number of elements doubles, the complexity changes by almost four times, provided n is a large number, and all the overheads take a negligible amount of time.

The game of clones revisited

The preceding game was being watched by Phineas Fletcher. He approached the group of people playing the game and suggested a simple change. He suggested selecting the highest number in each step and replacing it with the i^{th} element, i starting from 0. The process is shown in the following figure.

In the figure that follows, the largest number (63) is selected in the first step and swapped with the element at the 0^{th} index. In the next step, the second largest number is selected and swapped with the element at the 1^{st} index. Likewise, in the successive steps, the largest numbers from the remaining array are chosen and swapped with the element at the i^{th} index (*figure 1.6*). Note that if selecting the largest element from the remaining array takes time, O(log n) the whole process should take time of order O(n log n), which is much better than the previous method of sorting numbers.

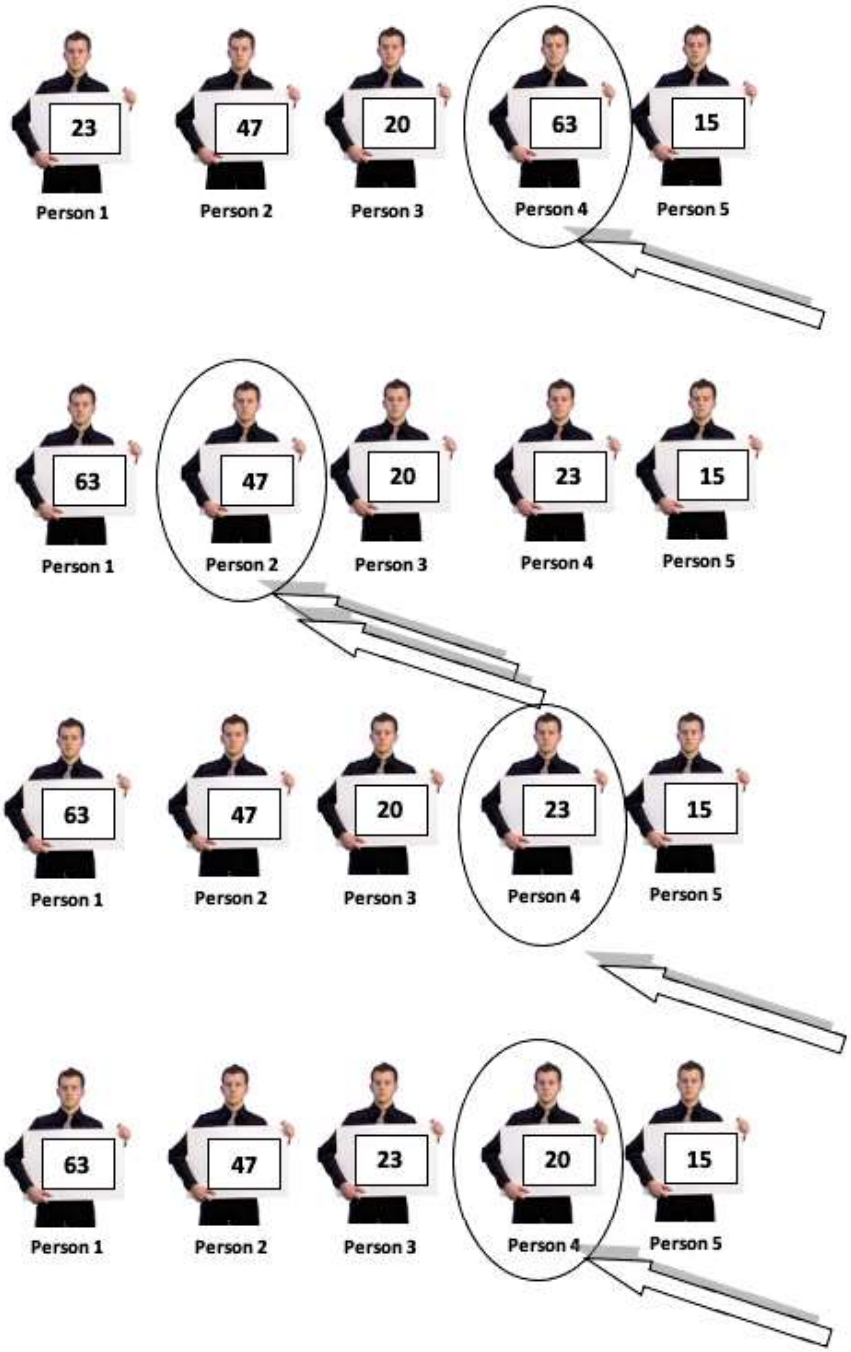

***Figure 1.6**: Another way to sort numbers*

The following graph (*figure 1.7*) shows the variation of n^2 and (n log n) with n and makes a strong case for the second method of sorting numbers:

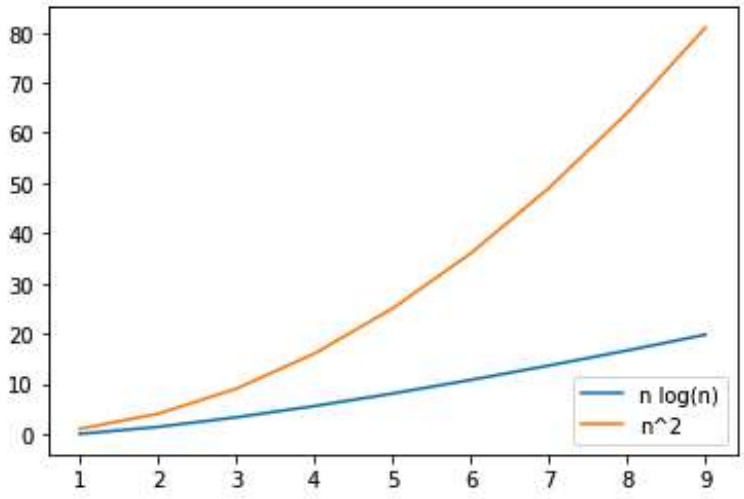

***Figure 1.7**: Variation of nlogn and n^2 with n*

Conclusion

This chapter introduced the reader, the fascinating world of data structures. The chapter defined the term data structure, explained its types, stated the difference between data structures and data types, and presented a problem to explain the importance of time complexity. The reader should get hold of the idea of data structure, its types, and their applications after reading this chapter. This chapter is your first step towards becoming an accomplished programmer.

The next chapter takes the discussion further and introduces algorithms and complexity. The features of an algorithm, the types of algorithms, and asymptotic notations for complexity are discussed in the upcoming chapter.

Multiple choice questions

1. **Which of the following follows the principle of Last In First Out (LIFO)?**
    a) Stack
    b) Queue
    c) Linked List
    d) None of above

2. **Which of the following follows the principle of First In First Out (FIFO)?**
   a) Stack
   b) Queue
   c) Linked List
   d) None of the above

3. **Which of the following is a linear data structure?**
   a) Stacks
   b) Queue
   c) Array
   d) All the above

4. **Which of the following is a non-linear data structure?**
   a) Tree
   b) Graph
   c) Plex
   d) All the above

5. **Which of the following data structure is used to evaluate a postfix expression?**
   a) Queue
   b) Stacks
   c) Array
   d) None of the above

6. **Which of the following data structure is used to evaluate a prefix expression?**
   a) Stacks
   b) Queue
   c) Array
   d) None of the above

7. **Which of the following data structure is used to find the shortest path?**
   a) Trees
   b) Graph
   c) Plex
   d) None of the above

8. **Which of the following data structure is used for an efficiently searching an element?**
   a) Binary Search Trees
   b) Queue
   c) Stack
   d) None of the above

9. **An integer in Turbo C takes two bytes;what is the maximum value that can be stored in it?**
   a. 32767
   b. 32768
   c. 65536
   d. 65535

10. **In the preceding question, if it is an unsigned integer, what is the maximum value that can be stored in it?**
    a. 32767
    b. 32768
    c. 65536
    d. 65535

Theory-based questions

1. What is a data structure?
2. Define data type and differentiate between primary and secondary data types?
3. What are linear and non-linear data structures?
4. Define Stack and give any two applications of Stacks?
5. Define Queue and give any two applications of Queues?

6. Define Trees and give any two applications of Trees?
7. Define Graph and give any two applications of Graphs?
8. How will you sort a list of numbers without spending time?
9. Explain PAPP.
10. Write a short note on why you should study data structures.

Application-based questions

1. **Harsh keeps track of music videos by saving the following data on his PC:**
    - Name
    - Singers
    - Music Director
    - Album
    - Year
    - Duration
    - Lyricist
        a. What steps would you take if you needed to store the preceding information of all the tracks in a file?
        b. State the data types of all the attributes?
        c. Give reasons for choosing a particular data type for an attribute? For example, explain why you choose a list/an array for the variable singers.
        d. The records need to be stored in a file and accessed by a Python program. Can you suggest a better way of storing the data?
2. **Listening to songs continuously motivated Harsh to keep track of albums as well.**
    a. What attributes must he store according to you?
    b. State the data type of each of the attribute?
    c. Can you suggest which attribute will always be unique for a particular record. What is such attribute called?
    d. Can you link the previous question and this question with such unique attribute?

3. In the preceding two questions, segregate the attributes as variable length attributes and fixed length ones?
4. Give a brief description of how will you search a record in this file?
5. Harsh decides to sort the records alphabetically. Based on the techniques introduced in the chapter, can you suggest a method to him?
6. Consider the following expression: *(a + b) - c*

   Create a tree for the same.
7. In your organization or University, there must be some hierarchy. For example, the Dean reports to the Vice Chancellor, the HOD (Head of Department) reports to the Dean and the faculty of a department reports to the HOD. Furthermore, each faculty may have a Teaching Assistance (TA). Create a tree representing this hierarchy.
8. You have given a list of numbers. Find out the subset of the list that sums to a particular number. Suggest a method for the same?

   For example, If the given list is [1, 2, 3, 4, 5, 6] and the sum is 6, then the subsets that sum to 6 are: {1, 5}, {2, 4}, {6}

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 2
Design Methodologies

The previous chapter discussed the definition, types, and importance of data structures. Some approaches for sorting a list of numbers were also discussed. This chapter takes the discussion forward and introduces design techniques. The techniques, their applicability, and when to use them have been discussed in this chapter. Also, the chapter gives an ephemeral introduction to asymptotic complexity, which has been discussed in detail in the Appendix of this book.

Let us begin with an example. Suppose you need to find a number in a given list. What will you do? If the list is not sorted, you may match the element at each index with the given number. If they match, you will print the index. In the other case, you will move ahead. If you are not able to find the number, then you will print the message "Not Found". This is Linear Search and takes time proportional to n (the number of elements in the list). If the list is sorted, we may apply Binary Search, which divides the list into two halves and reduce the size of the problem in half at each step. The algorithm takes time proportional to $O(\log n)$. Both these techniques are discussed in detail in the following chapters.

Now, hold and think about a situation where the sorted list is shifted by k positions (a finite number). How will you find a given element in this list? You may apply Linear Search, a variant of Binary Search, or a heap (*Chapters 3, Arrays* and *9, Heaps*). The chapters that follow will address all these questions. However, any solution that you think should be the following:

a) Correct
b) Efficient

The first and foremost thing in any solution that you propose is that it must be correct and must cater to all possible test cases. Efficiency comes next. The program should optimally use resources: both memory and time. Some of the approaches that you follow to solve a problem can be the following:

- Divide and conquer
- Greedy approach
- Dynamic programming
- Backtracking and branch and bound

Divide and Conquer is applied when a problem can be divided into similar sub-problems, and each sub-problem can be solved using that approach. Also, if required, there should be a way to club together the solutions to the sub-problems to return the solution to the original problem. Exploring each possibility and backtracking from the leaves to get the solution can be another approach, though expensive. If the sub-solutions can be memorized using a table, it leads to dynamic programming. The following sections will touch upon each of these approaches and will help you develop an understanding of these approaches and their applicability.

Structure

In this chapter, we will cover the following topics:

- Divide and conquer
- Greedy approach
- Dynamic programming and backtracking
- Longest common subsequence

Objectives

This chapter presents an overview of various methodologies, including the Greedy approach.

Divide and conquer, dynamic programming, and backtracking. This chapter also explains the difference between divide and conquer and dynamic programming and between dynamic programming and backtracking. Each methodology is explained using examples and numerical. Finally, the chapter will help you to move towards applying the learnt methods to solve problems.

Greedy approach

Axl is a teenager, who lives in Orson. He decides to open his burger joint named "The HecksBurger"and employs his younger brother Brick to handle the cash counter. The cash counter has bills having denominations 500, 100, 50, 20, 10, 5, and 1. Assuming that the prices of the burgers are in natural numbers, how do you think he would return the change with a minimum number of bills?

Let us understand this with the help of an example: If he gets 1,000 and needs to return 657. He will start with 500 (the highest denomination) and find the number of bills of this denomination required by dividing the amount to be returned by the denomination (and then taking the floor), which is 657/500, which is 1. Now, the remaining balance can be found by subtracting from 657, which gives 157. This is followed by repeating the process with the next highest denomination. That is, for finding the number of denominations of 100, divide the remaining, i.e., 157 by 100 (and then take the floor), which gives 1. The remaining amount in the next step will be . Likewise, the number of notes of 50, 20, 10, 5, and 1 will be 1, 0, 0, 0, and 2. That is, the number of bills required for change are [1, 1, 1, 0, 0, 0, 2].

The following Python code takes the amount to be returned as the input along with a list containing denominations in the sorted order and returns the list containing the number of denominations.

Code:

```
def coin_changing(L, amount):
  denomination = []
  i=0
  while(i<len(L)):
    num = int(amount/L[i])
    amount = amount - num*L[i]
    denomination.append(num)
    i+=1
  return denomination
L=[500, 100, 50, 20, 10, 5, 2, 1]
print(len(L))
den=coin_changing(L, 657)
print(den)
```

Output:

```
1. 8
2. [1, 1, 1, 0, 0, 1, 1, 0]
```

The preceding is an example of a Greedy approach, since at each step, the "best option at that point" is selected, and we proceed further. The approach is good but might not lead to an optimal solution always. We will explore this approach further in the chapters that follow. Particularly, this approach would be used to solve the minimum spanning tree problem.

Divide and conquer

In order to understand divide and conquer, let us consider the following illustration. Once upon a time, there was a king named Harsh. He was overtly fascinated with himself, and hence, stopped the coins used earlier in the kingdom. The new coins had his photo engraved. Moreover, all the subjects of the kingdom were asked to do the transactions in 2^k coins, where $k \epsilon Z$, for reasons which only future generations would understand.

The group of corrupt people called corrupts followed a protocol. They used to give one fake coin while doing a transaction. The weight of the fake coin was not the same as that of the original coin, and we do not know the weight of the valid coin. In order to identify the fake coin from the given coins in a transaction, the Algorithm department (he had established this department after selling the government IT company to his businessman friend) came up with an interesting solution, which is as follows:

Divide the set of n coins into two sets having $n/2$ coins each. If the weights of the two sets are equal, then there is no fake coin in the given set of n coins. In the other case, divide each set of $n/2$ coins into two sets of $n/4$ coins. The part whose two subsets of $n/4$ coins have equal weights do not have any fake coin, and the other set has a fake coin. The remaining coins can be divided further to reach a set of two coins, one of which is fake. Finally, the fake coin can be identified by comparing the remaining two coins with any coin in the set containing valid coins.

Figure 2.1 exemplifies the solution. In case there are 16 coins, then the coins can be divided into two sets having eight coins each. One of the sets has a fake coin. When the sets are further divided into two sets (we will have four sets of four coins). The two parts having the same weight (shown in gray) will not have any fake coin because there is only one fake coin, and the set having the fake coin will show different weights of its two subsets. The process is repeated till the last level (at which we have just one coin).

Note that the depth of the preceding tree is , as it the second level, there will be $n/2$ elements in each subset; at the third, there will be $n/2^2$, in the next step, there will be $n/2^3$ values, the process stops when only one value remains. That is, when $n/2^i = 1$, or $\log_2 n$.

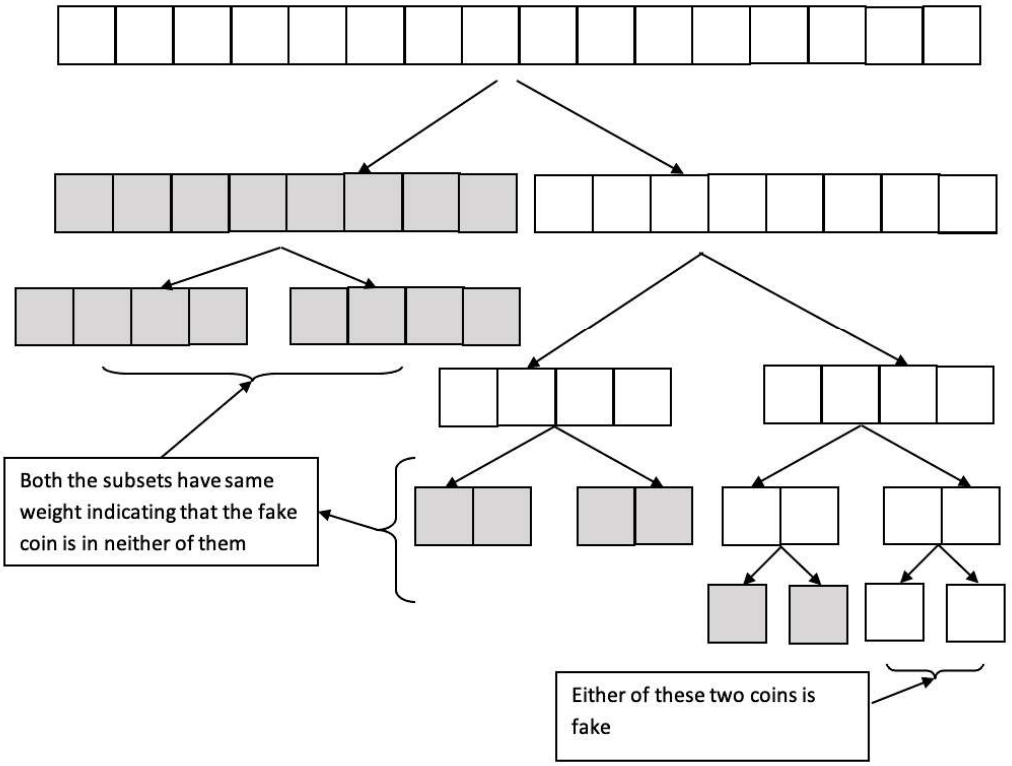

***Figure 2.1**: Finding a fake coin*

This was an example of divide and conquer. In divide and conquer, we divide the given problem into sub-problems of the same type. If needed, the solutions to all these problems are then clubbed, and the final solution is returned.

This technique can also be applied to problems like searching if the given array is sorted. The following chapters discuss some of the important applications of divide and conquer, such as merge sort, quick sort, matrix multiplication, exponentiation, and so on.

Backtracking and dynamic programming

Mathematical researchers faced some problems in the 1950s. *Charles Erwin Wilson* was the Secretary of Defense under President Eisenhower. Earlier, he worked as the CEO of *General Motors*. According to Bellman, *"he had pathological fear and hatred of the word research"*. This gives an idea of his relationship with mathematics. Bellman was working with the RAND Corporation that was employed by the Air Force. To hide that he was doing mathematics inside RAND, he devised the word Dynamic Programming so as to save the project that RAND got from being scrapped [1].

In **Dynamic Programming** (**DP**), we divide the problem into smaller subproblems, and the solution to these smaller problems helps to construct that of the bigger one. Here also we divide the problem into subproblems like in the case of divide and conquer, but there is a difference. In divide and conquer, all the problems are solved individually, and then if needed, the solution is combined. In dynamic programming, on the other hand, the smaller solutions contribute to the larger ones, which contribute to still larger ones. This technique is effective, much more effective than techniques like backtracking.

To understand this, consider a simple program to find the factorial of a number. The factorial of a number is found by taking the product of all the numbers starting from 1 to that number, that is,

$$n! = 1 \times 2 \times 3 \times \ldots \times n$$

Note that,

$$n! = 1 \times 2 \times 3 \times \ldots \times (n - 1) \times n = (n - 1)! \times n$$

That is,

$$fac(n) = n \times fac\ (n - 1)$$

factorial can be expressed in terms of itself and the base case, which is

$$fac(1) = 1$$

The following code assumes that the user enters a number ≥ 1. For negative numbers, the code can be modified by including a branch in the if-elif-else ladder. The implementation of the preceding using recursion is shown as follows:

Code:

```
def fac(n):
  if n==1:
    return 1
  else:
    return fac(n-1)*n
```

Output:

```
>>fac(1)
1
>>fac(8)
40320
>>fac(40)
815915283247897734345611269596115894272000000000
```

The preceding program is not the most efficient approach to solve this problem. The reason is that in calculating the factorial of higher numbers, that of the lower numbers is calculated again and again. To understand this, have a look at *figure 2.2*. Note that in calculating a factorial of 7, that of 6 is required, for which that of 5, 4, 3, 2, and 1 are required. A factorial of 1 is the base case of this recursive algorithm.

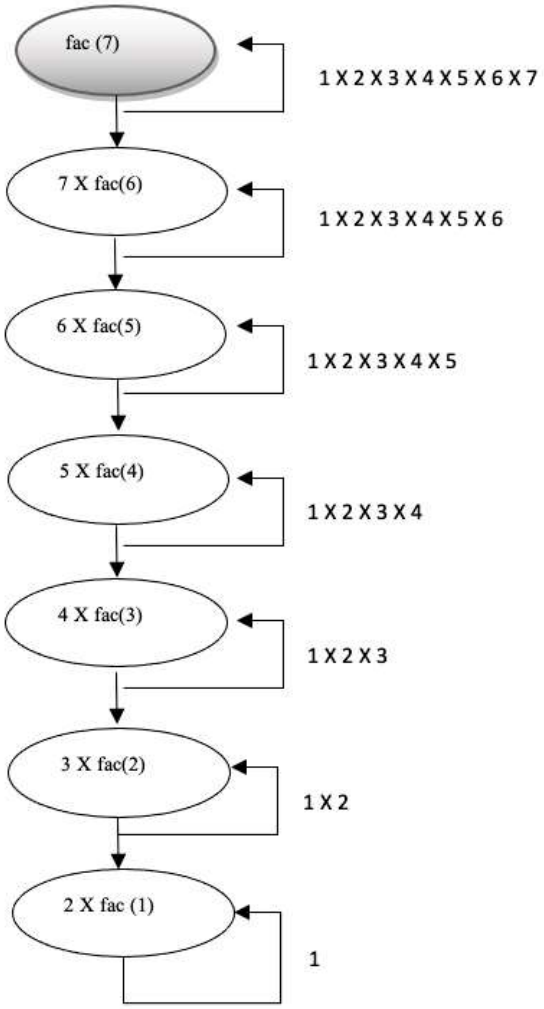

***Figure 2.2**: Factorial using recursion*

Now, consider the following implementation of factorial. Here, a list stores the factorial of a number and can be used to find the factorial of the later numbers.

Code:

```
def fac1(n):
  fac_num=[]
```

```
  if n==1:
    fac_num.append(1)
  else:
    fac_num.append(1)
    for i in range(1, n):
      fac_num.append(fac_num[i-1]*(i+1))
  return fac_num[-1]
```

Output:

```
fac1(40)
815915283247897734345611269596115894272000000000
```

This solution requires O(n) time. Note that this solution calculates the factorial of a number, and the result is used to find the factorial of a larger number. That is, memoization helps in reducing the complexity of the algorithm. Let us consider another problem to understand the concept.

Longest common sub-sequence

This section aims to find the longest common sub-sequence, given two strings. For example, if the first string is as follows:

`str1 = “ghijklmno”`

and the second string is:

`str2 = “gijo”`

Then the longest common sub-sequence is “gijo”, as the alphabet “g” is at the 0th index in the first string, “i” is at the 2nd index, “j” at the 3rd, and “o” is at the 8th index. Note that the matching indices in the first string are {0, 2, 3, 8}, which is an increasing sequence. The sequence of matching indices should be ordered.

As such, the problem seems easy. Find all the common sub-sequences of the two strings and return the longest. Wait! Try and implement this solution, and you will realize that this solution has an exponential time complexity.

Another option can be to use backtracking. In backtracking, we start with the 0th index and check if the alphabets at this index match. If they match, we increment the value of indices in both the strings and proceed further, else we explore both options: that of incrementing the index of the first string, keeping that of the second same,and the other of incrementing the index of the second keeping that of the first same.

In *figure 2.3*, the root node compares the character at the 0^{th} index of the two strings. Since they are equal, the values of both i and j are incremented. Note that str1[1] is