CHAPTER 0

ORGANIZATION OF CHAPTERS

0.1 What Is This Book About?

This book is about the fundamentals of data structures and algorithms – the basic elements from which large and complex software projects are built. To develop a good understanding of a data structure requires three things: first, you must learn how the information is arranged in the memory of the computer; second, you must become familiar with the algorithms for manipulating the information contained in the data structure; and third, you must understand the performance characteristics of the data structure so that when called upon to select a suitable data structure for a particular application, you are able to make an appropriate decision.

The algorithms and data structures in this book are presented in the Python programming language. A unique feature of this book, when compared to the available books on the subject, is that it offers a balance of theory, practical concepts, problem solving, and interview questions.

Concepts + Problems + Interview Questions

The book deals with some of the most important and challenging areas of programming and computer science in a highly readable manner. It covers both algorithmic theory and programming practice, demonstrating how theory is reflected in real Python programs. Well-known algorithms and data structures that are built into the Python language are explained, and the user is shown how to implement and evaluate others.

The book offers a large number of questions, with detailed answers, so you can practice and assess your knowledge before you take the exam or are interviewed.

Salient features of the book are:

- Basic principles of algorithm design
- How to represent well-known data structures in Python
- How to implement well-known algorithms in Python
- How to transform new problems into well-known algorithmic problems with efficient solutions
- How to analyze algorithms and Python programs using both mathematical tools and basic experiments and benchmarks
- How to understand several classical algorithms and data structures in depth, and be able to implement these efficiently in Python

Note that this book does not cover numerical or number-theoretical algorithms, parallel algorithms or multi-core programming.

0.2 Should I Buy This Book?

The book is intended for Python programmers who need to learn about algorithmic problem-solving or who need a refresher. However, others will also find it useful, including data and computational scientists employed to do big data analytic analysis; game programmers and financial analysts/engineers; and students of computer science or programming-related subjects such as bioinformatics.

Although this book is more precise and analytical than many other data structure and algorithm books, it rarely uses mathematical concepts that are more advanced than those taught in high school. I have made an effort to avoid using any advanced calculus, probability, or stochastic process concepts. The book is therefore appropriate for undergraduate students preparing for interviews.

0.3 Organization of Chapters

Data structures and algorithms are important aspects of computer science as they form the fundamental building blocks of developing logical solutions to problems, as well as creating efficient programs that perform tasks optimally. This book covers the topics required for a thorough understanding of the subjects such concepts as Linked Lists, Stacks, Queues, Trees, Priority Queues, Searching, Sorting, Hashing, Algorithm Design Techniques, Greedy, Divide and Conquer, Dynamic Programming and Symbol Tables.

The chapters are arranged as follows:

1. ***Introduction***: This chapter provides an overview of algorithms and their place in modern computing systems. It considers the general motivations for algorithmic analysis and the various approaches to studying the performance characteristics of algorithms.
2. ***Recursion and Backtracking***: *Recursion* is a programming technique that allows the programmer to express operations in terms of themselves. In other words, it is the process of defining a function or calculating a number by the repeated application of an algorithm.

   For many real-world problems, the solution process consists of working your way through a sequence of decision points in which each choice leads you further along some path (for example problems in the Trees and Graphs domain). If you make the correct set of choices, you end up at the solution. On the other hand, if you reach a dead end or otherwise discover that you have made an incorrect choice somewhere along the way, you have to backtrack to a previous decision point and try a different path. Algorithms that use this approach are called *backtracking* algorithms, and backtracking is a form of recursion. Also, some problems can be solved by combining recursion with backtracking.
3. ***Linked Lists***: A *linked list* is a dynamic data structure. The number of nodes in a list is not fixed and can grow and shrink on demand. Any application which has to deal with an unknown number of objects will need to use a linked list. It is a very common data structure that is used to create other data structures like trees, graphs, hashing, etc.
4. ***Stacks***: A *stack* abstract type is a container of objects that are inserted and removed according to the last-in-first-out (LIFO) principle. There are many applications of stacks, including:
   a. Space for function parameters and local variables is created internally using a stack.
   b. Compiler's syntax check for matching braces is implemented by using stack.
   c. Support for recursion.
   d. It can act as an auxiliary data structure for other abstract data types.
5. ***Queues***: *Queue* is also an abstract data structure or a linear data structure, in which the first element is inserted from one end called as *rear* (also called *tail*), and the deletion of the existing element takes place from the other end, called as *front* (also called *head*). This makes queue as FIFO data structure, which means that element inserted first will also be removed first. There are many applications of stacks, including:
   a. In operating systems, for controlling access to shared system resources such as printers, files, communication lines, disks and tapes.
   b. Computer systems must often provide a *holding area* for messages between two processes, two programs, or even two systems. This holding area is usually called a *buffer* and is often implemented as a queue.
   c. It can act as an auxiliary data structure for other abstract data types.
6. ***Trees***: A *tree* is an abstract data structure used to organize the data in a tree format so as to make the data insertion or deletion or search faster. Trees are one of the most useful data structures in computer science. Some of the common applications of trees are:
   a. The library database in a library, a student database in a school or college, an employee database in a company, a patient database in a hospital, or basically any database would be implemented using trees.
   b. The file system in your computer, i.e. folders and all files, would be stored as a tree.
   c. And a tree can act as an auxiliary data structure for other abstract data types.

A tree is an example of a non-linear data structure. There are many variants in trees, classified by the number of children and the way of interconnecting them. This chapter focuses on some of these variants, including Generic Trees, Binary Trees, Binary Search Trees, Balanced Binary Trees, etc.

7. ***Priority Queues***: The *priority queue* abstract data type is designed for systems that maintain a collection of prioritized elements, where elements are removed from the collection in order of their priority. Priority queues turn up in various applications, for example, processing jobs, where we process each job based on how urgent it is. For example, operating systems often use a priority queue for the ready queue of processes to run on the CPU.
8. ***Graph Algorithms***: Graphs are a fundamental data structure in the world of programming. A graph abstract data type is a collection of nodes called *vertices*, and the connections between them called *edges*. Graphs are an example of a non-linear data structure. This chapter focuses on representations of graphs (adjacency list and matrix representations), shortest path algorithms, etc. Graphs can be used to model many types of relations and processes in physical, biological, social and information systems, and many practical problems can be represented by graphs.
9. ***Disjoint Set ADT***: A disjoint set abstract data type represents a collection of sets that are disjoint: that is, no item is found in more than one set. The collection of disjoint sets is called a partition, because the items are partitioned among the sets. As an example, suppose the items in our universe are companies that still exist today or were acquired by other corporations. Our sets are companies that still exist under their own name. For instance, "*Motorola*," "*YouTube*," and "*Android*" are all members of the "*Google*" set.

   This chapter is limited to two operations. The first is called a *union* operation, in which we merge two sets into one. The second is called a *find* query, in which we ask a question like, "What corporation does Android belong to today?" More generally, a *find* query takes an item and tells us which set it is in. Data structures designed to support these operations are called *union/find* data structures. Applications of *union/find* data structures include maze generation and Kruskal's algorithm for computing the minimum spanning tree of a graph.
10. ***Sorting Algorithms***: *Sorting* is an algorithm that arranges the elements of a list in a certain order [either ascending or descending]. The output is a permutation or reordering of the input, and sorting is one of the important categories of algorithms in computer science. Sometimes sorting significantly reduces the complexity of the problem, and we can use sorting as a technique to reduce search complexity. Much research has gone into this category of algorithms because of its importance. These algorithms are used in many computer algorithms, for example, searching elements and database algorithms. In this chapter, we examine both comparison-based sorting algorithms and linear sorting algorithms.
11. ***Searching Algorithms***: In computer science, *searching* is the process of finding an item with specified properties from a collection of items. The items may be stored as records in a database, simple data elements in arrays, text in files, nodes in trees, vertices and edges in graphs, or elements of other search spaces.

    Searching is one of the core computer science algorithms. We know that today's computers store a lot of information, and to retrieve this information we need highly efficient searching algorithms. There are certain ways of organizing the data which improves the searching process. That means, if we keep the data in proper order, it is easy to search the required element. Sorting is one of the techniques for making the elements ordered. In this chapter we will see different searching algorithms.
12. ***Selection Algorithms***: A *selection algorithm* is an algorithm for finding the k^{th} smallest/largest number in a list (also called as k^{th} order statistic). This includes finding the minimum, maximum, and median elements. For finding k^{th} order statistic, there are multiple solutions which provide different complexities, and in this chapter we will enumerate those possibilities. We will also look at a linear algorithm for finding the k^{th} element in a given list.
13. ***Symbol Tables*** (*Dictionaries*): Since childhood, we all have used a dictionary, and many of us have a word processor (say, Microsoft Word), which comes with a spell checker. The spell checker is also a dictionary but limited in scope. There are many real time examples for dictionaries and a few of them are:
    a. Spelling checker
    b. The data dictionary found in database management applications
    c. Symbol tables generated by loaders, assemblers, and compilers
    d. Routing tables in networking components (DNS lookup)

In computer science, we generally use the term 'symbol' table rather than dictionary, when referring to the abstract data type (ADT).

14. ***Hashing***: *Hashing* is a technique used for storing and retrieving information as fast as possible. It is used to perform optimal search and is useful in implementing symbol tables. From the *Trees* chapter we understand that balanced binary search trees support operations such as insert, delete and search in O(*logn*) time. In applications, if we need these operations in O(1), then *hashing* provides a way. Remember that the worst case complexity of hashing is still O(*n*), but it gives O(1) on the average. In this chapter, we will take a detailed look at the hashing process and problems which can be solved with this technique.
15. ***String Algorithms***: To understand the importance of string algorithms, let us consider the case of entering the URL (Uniform Resource Locator) in any browser (say, Internet Explorer, Firefox, or Google Chrome). You will observe that after typing the prefix of the URL, a list of all possible URLs is displayed. That means, the browsers are doing some internal processing and giving us the list of matching URLs. This technique is sometimes called *auto-completion*. Similarly, consider the case of entering the directory name in a command line interface (in both Windows and UNIX). After typing the prefix of the directory name, if we press tab button, we then get a list of all matched directory names available. This is another example of auto completion.

    In order to support these kinds of operations, we need a data structure which stores the string data efficiently. In this chapter, we will look at the data structures that are useful for implementing string algorithms. We start our discussion with the basic problem of strings: given a string, how do we search a substring (pattern)? This is called *string matching problem*. After discussing various string matching algorithms, we will see different data structures for storing strings.
16. ***Algorithms Design Techniques***: In the previous chapters, we have seen many algorithms for solving different kinds of problems. Before solving a new problem, the general tendency is to look for the similarity of the current problem to other problems for which we have solutions. This helps us to get the solution easily. In this chapter, we see different ways of classifying the algorithms, and in subsequent chapters we will focus on a few of them (e.g., Greedy, Divide and Conquer, and Dynamic Programming).
17. ***Greedy Algorithms***: A greedy algorithm is also called a *single-minded* algorithm. A greedy algorithm is a process that looks for simple, easy-to-implement solutions to complex, multi-step problems by deciding which next step will provide the most obvious benefit. The idea behind a greedy algorithm is to perform a single procedure in the recipe over and over again until it can't be done any more, and see what kind of results it will produce. It may not completely solve the problem, or, if it produces a solution, it may not be the very best one, but it is one way of approaching the problem and sometimes yields very good (or even the best possible) results. Examples of greedy algorithms include selection sort, Prim's algorithms, Kruskal's algorithms, Dijkstra algorithm, Huffman coding algorithm etc.
18. ***Divide And Conquer***: These algorithms work based on the principles described below.
    a. *Divide* - break the problem into several subproblems that are similar to the original problem but smaller in size
    b. *Conquer* - solve the subproblems recursively.
    c. *Base case*: If the subproblem size is small enough (i.e., the base case has been reached) then solve the subproblem directly without more recursion.
    d. *Combine* - the solutions to create a solution for the original problem

    Examples of divide and conquer algorithms include Binary Search, Merge Sort etc....
19. ***Dynamic Programming***: In this chapter we will try to solve the problems for which we failed to get the optimal solutions using other techniques (say, Divide & Conquer and Greedy methods). Dynamic Programming (DP) is a simple technique but it can be difficult to master. One easy way to identify and solve DP problems is by solving as many problems as possible. The term Programming is not related to coding; it is from literature, and it means filling tables (similar to Linear Programming).
20. ***Complexity Classes***: In previous chapters we solved problems of different complexities. Some algorithms have lower rates of growth while others have higher rates of growth. The problems with lower rates of growth are called easy problems (or easy solved problems) and the problems with higher rates of growth are called hard problems (or hard solved problems). This classification is done based on the running time (or memory) that an algorithm takes for solving the problem. There are lots of problems for which we do not know the solutions.

In computer science, in order to understand the problems for which solutions are not there, the problems are divided into classes, and we call them *complexity classes*. In complexity theory, a complexity class is a set of problems with related complexity. It is the branch of theory of computation that studies the resources required during computation to solve a given problem. The most common resources are time (how much time the algorithm takes to solve a problem) and space (how much memory it takes). This chapter classifies the problems into different types based on their complexity class.

21. ***Miscellaneous Concepts: Bit – wise Hacking***: The commonality or applicability depends on the problem in hand. Some real-life projects do benefit from bit-wise operations.

    Some examples:

    - You're setting individual pixels on the screen by directly manipulating the video memory, in which every pixel's color is represented by 1 or 4 bits. So, in every byte you can have packed 8 or 2 pixels and you need to separate them. Basically, your hardware dictates the use of bit-wise operations.
    - You're dealing with some kind of file format (e.g. GIF) or network protocol that uses individual bits or groups of bits to represent pieces of information.
    - Your data dictates the use of bit-wise operations. You need to compute some kind of checksum (possibly, parity or CRC) or hash value, and some of the most applicable algorithms do this by manipulating with bits.

    In this chapter, we discuss a few tips and tricks with a focus on bitwise operators. Also, it covers a few other uncovered and general problems.

At the end of each chapter, a set of problems/questions is provided for you to improve/check your understanding of the concepts. The examples in this book are kept simple for easy understanding. The objective is to enhance the explanation of each concept with examples for a better understanding.

0.4 Some Prerequisites

This book is intended for two groups of people: Python programmers who want to beef up their algorithmics, and students taking algorithm courses who want a supplement to their algorithms textbook. Even if you belong to the latter group, I'm assuming you have a familiarity with programming in general and with Python in particular. If you don't, the Python web site also has a lot of useful material. Python is a really easy language to learn. There is some math in the pages ahead, but you don't have to be a math prodigy to follow the text. We'll be dealing with some simple sums and nifty concepts such as polynomials, exponentials, and logarithms, but I'll explain it all as we go along.

CHAPTER

1

INTRODUCTION

The objective of this chapter is to explain the importance of the analysis of algorithms, their notations, relationships and solving as many problems as possible. Let us first focus on understanding the basic elements of algorithms, the importance of algorithm analysis, and then slowly move toward the other topics as mentioned above. After completing this chapter, you should be able to find the complexity of any given algorithm (especially recursive functions).

1.1 Variables

Before going to the definition of variables, let us relate them to old mathematical equations. All of us have solved many mathematical equations since childhood. As an example, consider the below equation:

$$x^2 + 2y - 2 = 1$$

We don't have to worry about the use of this equation. The important thing that we need to understand is that the equation has names (x and y), which hold values (data). That means the *names* (x and y) are placeholders for representing data. Similarly, in computer science programming we need something for holding data, and *variables* is the way to do that.

1.2 Data Types

In the above-mentioned equation, the variables x and y can take any values such as integral numbers (10, 20), real numbers (0.23, 5.5), or just 0 and 1. To solve the equation, we need to relate them to the kind of values they can take, and *data type* is the name used in computer science programming for this purpose. A *data type* in a programming language is a set of data with predefined values. Examples of data types are: integer, floating point, unit number, character, string, etc.

Computer memory is all filled with zeros and ones. If we have a problem and we want to code it, it's very difficult to provide the solution in terms of zeros and ones. To help users, programming languages and compilers provide us with data types. For example, *integer* takes 2 bytes (actual value depends on compiler), *float* takes 4 bytes, etc. This says that in memory we are combining 2 bytes (16 bits) and calling it an *integer*. Similarly, combining 4 bytes (32 bits) and calling it a *float*. A data type reduces the coding effort. At the top level, there are two types of data types:

- System-defined data types (also called *Primitive* data types)
- User-defined data types

System-defined data types (Primitive data types)

Data types that are defined by system are called *primitive* data types. The primitive data types provided by many programming languages are: int, float, char, double, bool, etc. The number of bits allocated for each primitive data type depends on the programming languages, the compiler and the operating system. For the same primitive data type, different languages may use different sizes. Depending on the size of the data types, the total available values (domain) will also change.

For example, "*int*" may take 2 bytes or 4 bytes. If it takes 2 bytes (16 bits), then the total possible values are minus 32,768 to plus 32,767 (-2^{15} *to* $2^{15}-1$). If it takes 4 bytes (32 bits), then the possible values are between −2,147,483,648 and +2,147,483,647 (-2^{31} *to* $2^{31}-1$). The same is the case with other data types.

User defined data types

If the system-defined data types are not enough, then most programming languages allow the users to define their own data types, called *user − defined data types*. Good examples of user defined data types are: structures in *C/C++* and classes in *Java*. For example, in the snippet below, we are combining many system-defined data types and calling the user defined data type by the name "*newType*". This gives more flexibility and comfort in dealing with computer memory.

```
struct newType {
    int data1;
    float data 2;
    ...
    char data;
};
```

1.3 Data Structures

Based on the discussion above, once we have data in variables, we need some mechanism for manipulating that data to solve problems. *Data structure* is a particular way of storing and organizing data in a computer so that it can be used efficiently. A *data structure* is a special format for organizing and storing data. General data structure types include arrays, files, linked lists, stacks, queues, trees, graphs and so on.

Depending on the organization of the elements, data structures are classified into two types:

1) *Linear data structures*: Elements are accessed in a sequential order but it is not compulsory to store all elements sequentially. *Examples*: Linked Lists, Stacks and Queues.
2) *Non − linear data structures*: Elements of this data structure are stored/accessed in a non-linear order. *Examples*: Trees and graphs.

1.4 Abstract Data Types (ADTs)

Before defining abstract data types, let us consider the different view of system-defined data types. We all know that, by default, all primitive data types (int, float, etc.) support basic operations such as addition and subtraction. The system provides the implementations for the primitive data types. For user-defined data types we also need to define operations. The implementation for these operations can be done when we want to actually use them. That means, in general, user defined data types are defined along with their operations.

To simplify the process of solving problems, we combine the data structures with their operations and we call this *Abstract Data Types* (ADTs). An ADT consists of *two parts*:

1. Declaration of data
2. Declaration of operations

Commonly used ADTs *include*: Linked Lists, Stacks, Queues, Priority Queues, Binary Trees, Dictionaries, Disjoint Sets (Union and Find), Hash Tables, Graphs, and many others. For example, stack uses LIFO (Last-In-First-Out) mechanism while storing the data in data structures. The last element inserted into the stack is the first element that gets deleted. Common operations of it are: creating the stack, pushing an element onto the stack, popping an element from stack, finding the current top of the stack, finding number of elements in the stack, etc.

While defining the ADTs do not worry about the implementation details. They come into the picture only when we want to use them. Different kinds of ADTs are suited to different kinds of applications, and some are highly specialized to specific tasks. By the end of this book, we will go through many of them and you will be in a position to relate the data structures to the kind of problems they solve.

1.5 What is an Algorithm?

Let us consider the problem of preparing an *omelette*. To prepare an omelette, we follow the steps given below:

1) Get the frying pan.
2) Get the oil.
    a. Do we have oil?
        i. If yes, put it in the pan.

ii. If no, do we want to buy oil?
    1. If yes, then go out and buy.
    2. If no, we can terminate.

3) Turn on the stove, etc...

What we are doing is, for a given problem (preparing an omelette), we are providing a step-by-step procedure for solving it. The formal definition of an algorithm can be stated as:

An algorithm is the step-by-step instructions to solve a given problem.

Note: We do not have to prove each step of the algorithm.

1.6 Why the Analysis of Algorithms?

To go from city *"A"* to city *"B"*, there can be many ways of accomplishing this: by flight, by bus, by train and also by bicycle. Depending on the availability and convenience, we choose the one that suits us. Similarly, in computer science, multiple algorithms are available for solving the same problem (for example, a sorting problem has many algorithms, like insertion sort, selection sort, quick sort and many more). Algorithm analysis helps us to determine which algorithm is most efficient in terms of time and space consumed.

1.7 Goal of the Analysis of Algorithms

The goal of the *analysis of algorithms* is to compare algorithms (or solutions) mainly in terms of running time but also in terms of other factors (e.g., memory, developer effort, etc.)

1.8 What is Running Time Analysis?

It is the process of determining how processing time increases as the size of the problem (input size) increases. Input size is the number of elements in the input, and depending on the problem type, the input may be of different types. The following are the common types of inputs.

- Size of an array
- Polynomial degree
- Number of elements in a matrix
- Number of bits in the binary representation of the input
- Vertices and edges in a graph.

1.9 How to Compare Algorithms

To compare algorithms, let us define a few *objective measures*:

Execution times? *Not a good measure* as execution times are specific to a particular computer.

Number of statements executed? *Not a good measure*, since the number of statements varies with the programming language as well as the style of the individual programmer.

Ideal solution? Let us assume that we express the running time of a given algorithm as a function of the input size n (i.e., $f(n)$) and compare these different functions corresponding to running times. This kind of comparison is independent of machine time, programming style, etc.

1.10 What is Rate of Growth?

The rate at which the running time increases as a function of input is called *rate of growth*. Let us assume that you go to a shop to buy a car and a bicycle. If your friend sees you there and asks what you are buying, then in general you say *buying a car*. This is because the cost of the car is high compared to the cost of the bicycle (approximating the cost of the bicycle to the cost of the car).

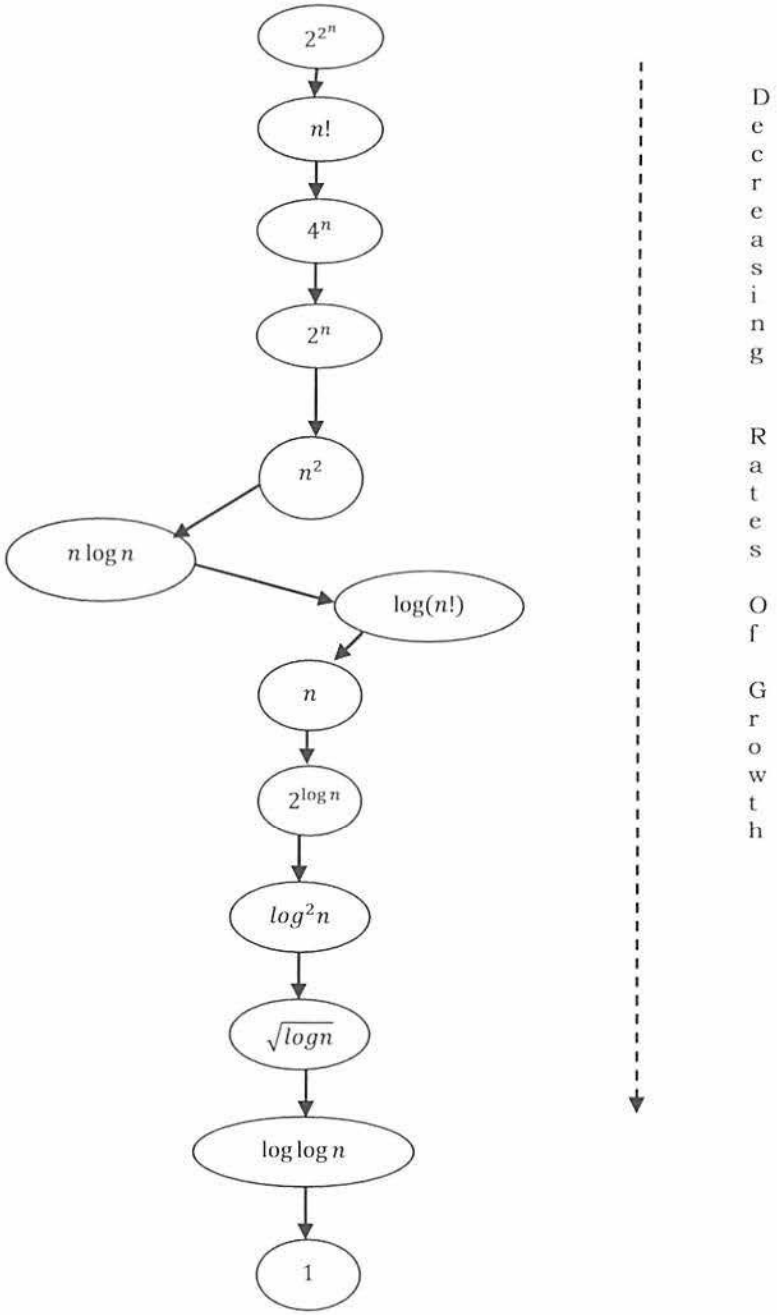
$$Total\ Cost = cost\_of\_car + cost\_of\_bicycle$$
$$Total\ Cost \approx cost\_of\_car\ (approximation)$$

For the above-mentioned example, we can represent the cost of the car and the cost of the bicycle in terms of function, and for a given function ignore the low order terms that are relatively insignificant (for large value of input size, n). As an example, in the case below, n^4, $2n^2$, $100n$ and 500 are the individual costs of some function and approximate to n^4 since n^4 is the highest rate of growth.

$$n^4 + 2n^2 + 100n + 500 \approx n^4$$

1.11 Commonly Used Rates of Growth

The diagram below shows the relationship between different rates of growth.



Below is the list of growth rates you will come across in the following chapters.

Time Complexity	Name	Example
1	Constant	Adding an element to the front of a linked list
$logn$	Logarithmic	Finding an element in a sorted array
n	Linear	Finding an element in an unsorted array
$nlogn$	Linear Logarithmic	Sorting n items by 'divide-and-conquer' - Mergesort
n^2	Quadratic	Shortest path between two nodes in a graph
n^3	Cubic	Matrix Multiplication
2^n	Exponential	The Towers of Hanoi problem

1.12 Types of Analysis

To analyze the given algorithm, we need to know with which inputs the algorithm takes less time (performing well) and with which inputs the algorithm takes a long time. We have already seen that an algorithm can be represented in the form of an expression. That means we represent the algorithm with multiple expressions: one for the case where it takes less time and another for the case where it takes more time.

In general, the first case is called the *best case* and the second case is called the *worst case* for the algorithm. To analyze an algorithm we need some kind of syntax, and that forms the base for asymptotic analysis/notation. There are three types of analysis:

- **Worst case**
  - Defines the input for which the algorithm takes a long time.
  - Input is the one for which the algorithm runs the slowest.
- **Best case**
  - Defines the input for which the algorithm takes the least time.
  - Input is the one for which the algorithm runs the fastest.
- **Average case**
  - Provides a prediction about the running time of the algorithm.
  - Assumes that the input is random.

$$Lower\ Bound <= Average\ Time <= Upper\ Bound$$

For a given algorithm, we can represent the best, worst and average cases in the form of expressions. As an example, let $f(n)$ be the function which represents the given algorithm.

$$f(n) = n^2 + 500, \text{ for worst case}$$
$$f(n) = n + 100n + 500, \text{ for best case}$$

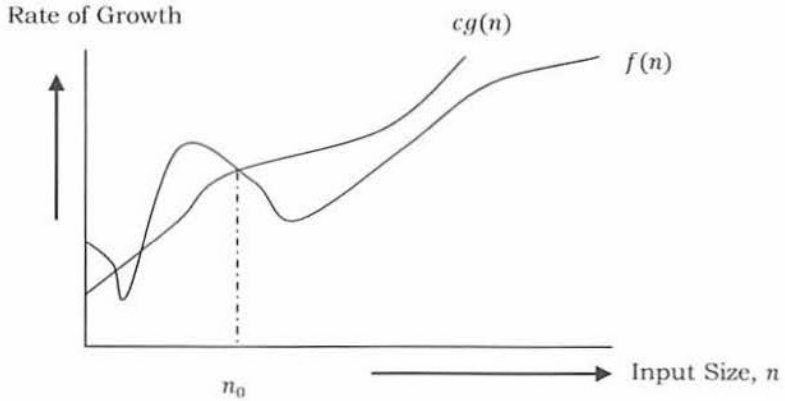
Similarly for the average case. The expression defines the inputs with which the algorithm takes the average running time (or memory).

1.13 Asymptotic Notation

Having the expressions for the best, average and worst cases, for all three cases we need to identify the upper and lower bounds. To represent these upper and lower bounds, we need some kind of syntax, and that is the subject of the following discussion. Let us assume that the given algorithm is represented in the form of function $f(n)$.

1.14 Big-O Notation

This notation gives the *tight* upper bound of the given function. Generally, it is represented as $f(n) = O(g(n))$. That means, at larger values of n, the upper bound of $f(n)$ is $g(n)$. For example, if $f(n) = n^4 + 100n^2 + 10n + 50$ is the given algorithm, then n^4 is $g(n)$. That means $g(n)$ gives the maximum rate of growth for $f(n)$ at larger values of n.



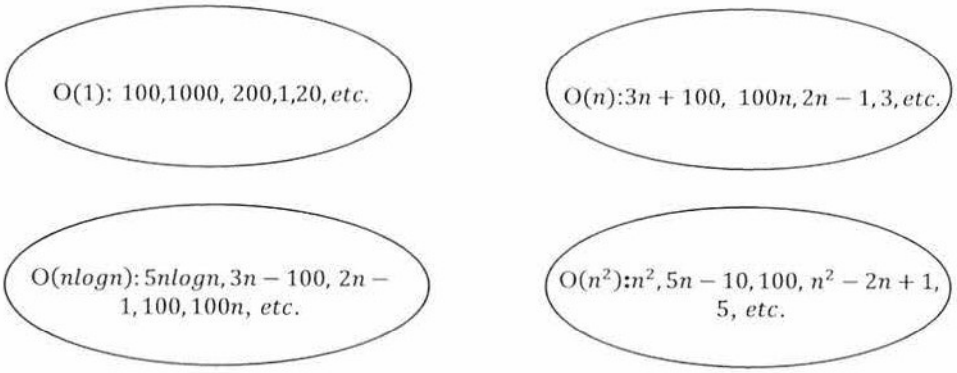
Let us see the O–notation with a little more detail. O–notation defined as $O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$. $g(n)$ is an asymptotic tight upper bound for $f(n)$. Our objective is to give the smallest rate of growth $g(n)$ which is greater than or equal to the given algorithms' rate of growth $f(n)$.

Generally we discard lower values of n. That means the rate of growth at lower values of n is not important. In the figure, n_0 is the point from which we need to consider the rate of growth for a given algorithm. Below n_0, the rate of growth could be different. n_0 is called threshold for the given function.

Big-O Visualization

$O(g(n))$ is the set of functions with smaller or the same order of growth as $g(n)$. For example; $O(n^2)$ includes $O(1), O(n), O(nlogn)$, etc.

Note: Analyze the algorithms at larger values of n only. What this means is, below n_0 we do not care about the rate of growth.



Big-O Examples

Example-1 Find upper bound for $f(n) = 3n + 8$

Solution: $3n + 8 \leq 4n$, for all $n \geq 8$
$\therefore 3n + 8 = O(n)$ with c = 4 and $n_0 = 8$

Example-2 Find upper bound for $f(n) = n^2 + 1$

Solution: $n^2 + 1 \leq 2n^2$, for all $n \geq 1$
$\therefore n^2 + 1 = O(n^2)$ with $c = 2$ and $n_0 = 1$

Example-3 Find upper bound for $f(n) = n^4 + 100n^2 + 50$

Solution: $n^4 + 100n^2 + 50 \leq 2n^4$, for all $n \geq 11$
$\therefore n^4 + 100n^2 + 50 = O(n^4)$ with $c = 2$ and $n_0 = 11$

Example-4 Find upper bound for $f(n) = 2n^3 - 2n^2$

Solution: $2n^3 - 2n^2 \leq 2n^3$, for all $n \geq 1$
$\therefore 2n^3 - 2n^2 = O(2n^3)$ with $c = 2$ and $n_0 = 1$

Example-5 Find upper bound for $f(n) = n$

Solution: $n \leq n$, for all $n \geq 1$
$\therefore n = O(n)$ with $c = 1$ and $n_0 = 1$

Example-6 Find upper bound for $f(n) = 410$

Solution: $410 \leq 410$, for all $n \geq 1$
$\therefore 410 = O(1)$ with $c = 1$ and $n_0 = 1$

No Uniqueness?

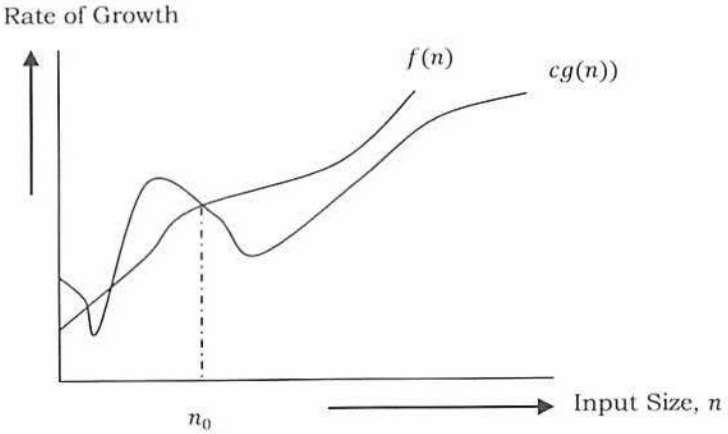
There is no unique set of values for n_0 and c in proving the asymptotic bounds. Let us consider, $100n + 5 = O(n)$. For this function there are multiple n_0 and c values possible.

Solution1: $100n + 5 \leq 100n + n = 101n \leq 101n$, for all $n \geq 5$, $n_0 = 5$ and $c = 101$ is a solution.

Solution2: $100n + 5 \leq 100n + 5n = 105n \leq 105n$, for all $n \geq 1$, $n_0 = 1$ and $c = 105$ is also a solution.

1.15 Omega-Ω Notation

Similar to the O discussion, this notation gives the tighter lower bound of the given algorithm and we represent it as $f(n) = \Omega(g(n))$. That means, at larger values of n, the tighter lower bound of $f(n)$ is $g(n)$. For example, if $f(n) = 100n^2 + 10n + 50$, $g(n)$ is $\Omega(n^2)$.



The Ω notation can be defined as Ω(g(n)) = {f(n): there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$}. $g(n)$ is an asymptotic tight lower bound for $f(n)$. Our objective is to give the largest rate of growth $g(n)$ which is less than or equal to the given algorithm's rate of growth $f(n)$.

Ω Examples

Example-1 Find lower bound for $f(n) = 5n^2$.

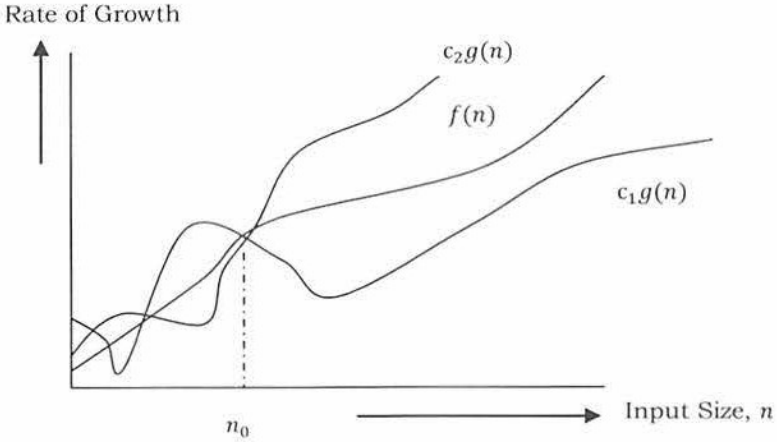
Solution: $\exists\, c,\ n_0$ Such that: $0 \leq cn^2 \leq 5n^2 \Rightarrow cn^2 \leq 5n^2 \Rightarrow c = 1$ and $n_0 = 1$
$\therefore 5n^2 = \Omega(n^2)$ with $c = 1$ and $n_0 = 1$

Example-2 Prove $f(n) = 100n + 5 \neq \Omega(n^2)$.

Solution: $\exists\, c,\ n_0$ Such that: $0 \leq cn^2 \leq 100n + 5$
$100n + 5 \leq 100n + 5n\ (\forall n \geq 1) = 105n$
$cn^2 \leq 105n \Rightarrow n(cn - 105) \leq 0$
Since n is positive $\Rightarrow cn - 105 \leq 0 \Rightarrow n \leq 105/c$
$\Rightarrow$ Contradiction: n cannot be smaller than a constant

Example-3 $2n = \Omega(n)$, $n^3 = \Omega(n^3)$, $logn = \Omega(logn)$.

1.16 Theta-Θ Notation



This notation decides whether the upper and lower bounds of a given function (algorithm) are the same. The average running time of an algorithm is always between the lower bound and the upper bound. If the upper

bound (O) and lower bound (Ω) give the same result, then the Θ notation will also have the same rate of growth. As an example, let us assume that $f(n) = 10n + n$ is the expression. Then, its tight upper bound $g(n)$ is $O(n)$. The rate of growth in the best case is $g(n) = O(n)$.

In this case, the rates of growth in the best case and worst case are the same. As a result, the average case will also be the same. For a given function (algorithm), if the rates of growth (bounds) for O and Ω are not the same, then the rate of growth for the Θ case may not be the same. In this case, we need to consider all possible time complexities and take the average of those (for example, for a quick sort average case, refer to the *Sorting* chapter).

Now consider the definition *of* Θ notation. It is defined as $\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2 and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0\}$. $g(n)$ is an asymptotic tight bound for $f(n)$. $\Theta(g(n))$ is the set of functions with the same order of growth as $g(n)$.

Θ Examples

Example 1 Find Θ bound for $f(n) = \frac{n^2}{2} - \frac{n}{2}$

Solution: $\frac{n^2}{5} \leq \frac{n^2}{2} - \frac{n}{2} \leq n^2$, for all, $n \geq 1$

$\therefore \frac{n^2}{2} - \frac{n}{2} = \Theta(n^2)$ with $c_1 = 1/5, c_2 = 1$ and $n_0 = 1$

Example 2 Prove $n \neq \Theta(n^2)$

Solution: $c_1 n^2 \leq n \leq c_2 n^2 \Rightarrow$ only holds for: $n \leq 1/c_1$

$\therefore n \neq \Theta(n^2)$

Example 3 Prove $6n^3 \neq \Theta(n^2)$

Solution: $c_1 n^2 \leq 6n^3 \leq c_2 n^2 \Rightarrow$ only holds for: $n \leq c_2/6$

$\therefore 6n^3 \neq \Theta(n^2)$

Example 4 Prove $n \neq \Theta(logn)$

Solution: $c_1 logn \leq n \leq c_2 log\, n \Rightarrow c_2 \geq \frac{n}{\log n}, \forall n \geq n_0$ – Impossible

Important Notes

For analysis (best case, worst case and average), we try to give the upper bound (O) and lower bound (Ω) and average running time (Θ). From the above examples, it should also be clear that, for a given function (algorithm), getting the upper bound (O) and lower bound (Ω) and average running time (Θ) may not always be possible. For example, if we are discussing the best case of an algorithm, we try to give the upper bound (O) and lower bound (Ω) and average running time (Θ).

In the remaining chapters, we generally focus on the upper bound (O) because knowing the lower bound (Ω) of an algorithm is of no practical importance, and we use the Θ notation if the upper bound (O) and lower bound (Ω) are the same.

1.17 Why is it called Asymptotic Analysis?

From the discussion above (for all three notations: worst case, best case, and average case), we can easily understand that, in every case for a given function $f(n)$ we are trying to find another function $g(n)$ which approximates $f(n)$ at higher values of n. That means $g(n)$ is also a curve which approximates $f(n)$ at higher values of n.

In mathematics we call such a curve an *asymptotic curve*. In other terms, $g(n)$ is the asymptotic curve for $f(n)$. For this reason, we call algorithm analysis *asymptotic analysis*.

1.18 Guidelines for Asymptotic Analysis

There are some general rules to help us determine the running time of an algorithm.

1) **Loops**: The running time of a loop is, at most, the running time of the statements inside the loop (including tests) multiplied by the number of iterations.

```
# executes n times
for i in range(0,n):
    print 'Current Number :', i    #constant time
```

Total time = a constant $c \times n = cn = O(n)$.

2) **Nested loops:** Analyze from the inside out. Total running time is the product of the sizes of all the loops.

```
# outer loop executed n times
for i in range(0,n):
    # inner loop executes n times
    for j in range(0,n):
        print 'i value %d and j value %d' % (i,j) #constant time
```

Total time = $c \times n \times n = cn^2 = O(n^2)$.

3) **Consecutive statements:** Add the time complexities of each statement.

```
n = 100
# executes n times
for i in range(0,n):
    print 'Current Number :', i                    #constant time
# outer loop executed n times
for i in range(0,n):
    # inner loop executes n times
    for j in range(0,n):
        print 'i value %d and j value %d' % (i,j)  #constant time
```

Total time = $c_0 + c_1 n + c_2 n^2 = O(n^2)$.

4) **If-then-else statements:** Worst-case running time: the test, plus *either* the *then* part *or* the *else* part (whichever is the larger).

```
if n == 1:                          #constant time
  print "Wrong Value"
  print n
else:
  for i in range(0,n):              #n times
    print 'Current Number :', i     #constant time
```

Total time = $c_0 + c_1 * n = O(n)$.

5) **Logarithmic complexity:** An algorithm is O(*logn*) if it takes a constant time to cut the problem size by a fraction (usually by ½). As an example let us consider the following program:

```
def Logarithms(n):
  i = 1
  while i <= n:
            i= i * 2
            print i
Logarithms(100)
```

If we observe carefully, the value of i is doubling every time. Initially $i = 1$, in next step $i = 2$, and in subsequent steps $i = 4, 8$ and so on. Let us assume that the loop is executing some k times. At k^{th} step $2^k = n$ and we come out of loop. Taking logarithm on both sides, gives

$log(2^k) = logn$
$klog2 = logn$
$k = logn$ //if we assume base-2

Total time = O(*logn*).

Note: Similarly, for the case below, the worst case rate of growth is O(*logn*). The same discussion holds good for the decreasing sequence as well.

```
def Logarithms(n):
  i = n
  while i >= 1:
            i= i // 2
            print i
Logarithms(100)
```

Another example: binary search (finding a word in a dictionary of n pages)

- Look at the center point in the dictionary
- Is the word towards the left or right of center?
- Repeat the process with the left or right part of the dictionary until the word is found.

1.19 Properties of Notations

- Transitivity: $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$. Valid for O and Ω as well.
- Reflexivity: $f(n) = \Theta(f(n))$. Valid for O and Ω.
- Symmetry: $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$.
- Transpose symmetry: $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.

1.20 Commonly used Logarithms and Summations

Logarithms

$$\log x^y = y \log x \qquad \log n = \log_{10}^{n}$$

$$\log xy = \log x + \log y \qquad \log^k n = (\log n)^k$$

$$\log \log n = \log(\log n) \qquad \log \frac{x}{y} = \log x - \log y$$

$$a^{\log_b^x} = x^{\log_b^a} \qquad \log_b^x = \frac{\log_a^x}{\log_a^b}$$

Arithmetic series

$$\sum_{K=1}^{n} k = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

Geometric series

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 \ldots + x^n = \frac{x^{n+1} - 1}{x - 1} (x \neq 1)$$

Harmonic series

$$\sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \ldots + \frac{1}{n} \approx \log n$$

Other important formulae

$$\sum_{k=1}^{n} \log k \approx n \log n$$

$$\sum_{k=1}^{n} k^p = 1^p + 2^p + \cdots + n^p \approx \frac{1}{p+1} n^{p+1}$$

1.21 Master Theorem for Divide and Conquer

All divide and conquer algorithms (Also discussed in detail in the *Divide and Conquer* chapter) divide the problem into sub-problems, each of which is part of the original problem, and then perform some additional work to compute the final answer. As an example, a merge sort algorithm [for details, refer to *Sorting* chapter] operates on two sub-problems, each of which is half the size of the original, and then performs $O(n)$ additional work for merging. This gives the running time equation:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

The following theorem can be used to determine the running time of divide and conquer algorithms. For a given program (algorithm), first we try to find the recurrence relation for the problem. If the recurrence is of the below form then we can directly give the answer without fully solving it.

If the recurrence is of the form $T(n) = aT(\frac{n}{b}) + \Theta(n^k \log^p n)$, where $a \geq 1$, $b > 1$, $k \geq 0$ and p is a real number, then:

1) If $a > b^k$, then $T(n) = \Theta\left(n^{\log_b^a}\right)$
2) If $a = b^k$
   a. If $p > -1$, then $T(n) = \Theta\left(n^{\log_b^a} \log^{p+1} n\right)$
   b. If $p = -1$, then $T(n) = \Theta\left(n^{\log_b^a} \log \log n\right)$
   c. If $p < -1$, then $T(n) = \Theta\left(n^{\log_b^a}\right)$
3) If $a < b^k$
   a. If $p \geq 0$, then $T(n) = \Theta(n^k \log^p n)$
   b. If $p < 0$, then $T(n) = O(n^k)$

1.22 Divide and Conquer Master Theorem: Problems & Solutions

For each of the following recurrences, give an expression for the runtime $T(n)$ if the recurrence can be solved with the Master Theorem. Otherwise, indicate that the Master Theorem does not apply.

Problem-1 $T(n) = 3T(n/2) + n^2$

Solution: $T(n) = 3T(n/2) + n^2 \Rightarrow T(n) = \Theta(n^2)$ (Master Theorem Case 3.a)

Problem-2 $T(n) = 4T(n/2) + n^2$

Solution: $T(n) = 4T(n/2) + n^2 \Rightarrow T(n) = \Theta(n^2 logn)$ (Master Theorem Case 2.a)

Problem-3 $T(n) = T(n/2) + n^2$

Solution: $T(n) = T(n/2) + n^2 \Rightarrow \Theta(n^2)$ (Master Theorem Case 3.a)

Problem-4 $T(n) = 2^n T(n/2) + n^n$

Solution: $T(n) = 2^n T(n/2) + n^n \Rightarrow$ Does not apply (a is not constant)

Problem-5 $T(n) = 16T(n/4) + n$

Solution: $T(n) = 16T(n/4) + n \Rightarrow T(n) = \Theta(n^2)$ (Master Theorem Case 1)

Problem-6 $T(n) = 2T(n/2) + nlogn$

Solution: $T(n) = 2T(n/2) + nlogn \Rightarrow T(n) = \Theta(nlog^2 n)$ (Master Theorem Case 2.a)

Problem-7 $T(n) = 2T(n/2) + n/logn$

Solution: $T(n) = 2T(n/2) + n/logn \Rightarrow T(n) = \Theta(nloglogn)$ (Master Theorem Case 2.b)

Problem-8 $T(n) = 2T(n/4) + n^{0.51}$

Solution: $T(n) = 2T(n/4) + n^{0.51} \Rightarrow T(n) = \Theta(n^{0.51})$ (Master Theorem Case 3.b)

Problem-9 $T(n) = 0.5T(n/2) + 1/n$

Solution: $T(n) = 0.5T(n/2) + 1/n \Rightarrow$ Does not apply ($a < 1$)

Problem-10 $T(n) = 6T(n/3) + n^2 logn$

Solution: $T(n) = 6T(n/3) + n^2 logn \Rightarrow T(n) = \Theta(n^2 logn)$ (Master Theorem Case 3.a)

Problem-11 $T(n) = 64T(n/8) - n^2 logn$

Solution: $T(n) = 64T(n/8) - n^2 logn \Rightarrow$ Does not apply (function is not positive)

Problem-12 $T(n) = 7T(n/3) + n^2$

Solution: $T(n) = 7T(n/3) + n^2 \Rightarrow T(n) = \Theta(n^2)$ (Master Theorem Case 3.as)

Problem-13 $T(n) = 4T(n/2) + logn$

Solution: $T(n) = 4T(n/2) + logn \Rightarrow T(n) = \Theta(n^2)$ (Master Theorem Case 1)

Problem-14 $T(n) = 16T(n/4) + n!$

Solution: $T(n) = 16T(n/4) + n! \Rightarrow T(n) = \Theta(n!)$ (Master Theorem Case 3.a)

Problem-15 $T(n) = \sqrt{2}T(n/2) + logn$

Solution: $T(n) = \sqrt{2}T(n/2) + logn \Rightarrow T(n) = \Theta(\sqrt{n})$ (Master Theorem Case 1)

Problem-16 $T(n) = 3T(n/2) + n$

Solution: $T(n) = 3T(n/2) + n \Rightarrow T(n) = \Theta(n^{log3})$ (Master Theorem Case 1)

Problem-17 $T(n) = 3T(n/3) + \sqrt{n}$

Solution: $T(n) = 3T(n/3) + \sqrt{n} \Rightarrow T(n) = \Theta(n)$ (Master Theorem Case 1)

Problem-18 $T(n) = 4T(n/2) + cn$

Solution: $T(n) = 4T(n/2) + cn \Rightarrow T(n) = \Theta(n^2)$ (Master Theorem Case 1)

Problem-19 $T(n) = 3T(n/4) + nlogn$

Solution: $T(n) = 3T(n/4) + nlogn \Rightarrow T(n) = \Theta(nlogn)$ (Master Theorem Case 3.a)

Problem-20 $T(n) = 3T(n/3) + n/2$

Solution: $T(n) = 3T(n/3) + n/2 \Rightarrow T(n) = \Theta(nlogn)$ (Master Theorem Case 2.a)

1.23 Master Theorem for Subtract and Conquer Recurrences

Let $T(n)$ be a function defined on positive n, and having the property

$$T(n) = \begin{cases} c, & \text{if } n \leq 1 \\ aT(n-b) + f(n), & \text{if } n > 1 \end{cases}$$

for some constants $c, a > 0, b > 0, k \geq 0$, and function $f(n)$. If $f(n)$ is in $O(n^k)$, then

$$T(n) = \begin{cases} O(n^k), & \text{if } a < 1 \\ O(n^{k+1}), & \text{if } a = 1 \\ O\left(n^k a^{\frac{n}{b}}\right), & \text{if } a > 1 \end{cases}$$

1.24 Variant of Subtraction and Conquer Master Theorem

The solution to the equation $T(n) = T(\alpha n) + T((1-\alpha)n) + \beta n$, where $0 < \alpha < 1$ and $\beta > 0$ are constants, is $O(nlogn)$.

1.25 Method of Guessing and Confirming

Now, let us discuss a method which can be used to solve any recurrence. The basic idea behind this method is:

guess the answer; and then *prove* it correct by induction.

In other words, it addresses the question: What if the given recurrence doesn't seem to match with any of these (master theorem) methods? If we guess a solution and then try to verify our guess inductively, usually either the proof will succeed (in which case we are done), or the proof will fail (in which case the failure will help us refine our guess).

As an example, consider the recurrence $T(n) = \sqrt{n}\, T(\sqrt{n}) + n$. This doesn't fit into the form required by the Master Theorems. Carefully observing the recurrence gives us the impression that it is similar to the divide and conquer method (dividing the problem into $\sqrt{n}$ subproblems each with size $\sqrt{n}$). As we can see, the size of the subproblems at the first level of recursion is n. So, let us guess that $T(n) = O(nlogn)$, and then try to prove that our guess is correct.

Let's start by trying to prove an *upper* bound $T(n) \leq cnlogn$:

$$\begin{aligned} T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\ &\leq \sqrt{n}.\, c\sqrt{n}\, log\sqrt{n} + n \\ &= n.\, c\, log\sqrt{n} + n \\ &= n.c.\frac{1}{2}.logn + n \\ &\leq cnlogn \end{aligned}$$

The last inequality assumes only that $1 \leq c.\frac{1}{2}.logn$. This is correct if n is sufficiently large and for any constant c, no matter how small. From the above proof, we can see that our guess is correct for the upper bound. Now, let us prove the *lower* bound for this recurrence.

$$\begin{aligned} T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\ &\geq \sqrt{n}.\, k\sqrt{n}\, log\sqrt{n} + n \\ &= n.\, k\, log\sqrt{n} + n \\ &= n.k.\frac{1}{2}.logn + n \\ &\geq knlogn \end{aligned}$$

The last inequality assumes only that $1 \geq k.\frac{1}{2}.logn$. This is incorrect if n is sufficiently large and for any constant k. From the above proof, we can see that our guess is incorrect for the lower bound.

From the above discussion, we understood that $\Theta(nlogn)$ is too big. How about $\Theta(n)$? The lower bound is easy to prove directly:

$$T(n) = \sqrt{n}\, T(\sqrt{n}) + n \quad \geq n$$

Now, let us prove the upper bound for this $\Theta(n)$.

$$\begin{aligned} T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\ &\leq \sqrt{n}.c.\sqrt{n} + n \\ &= n.\, c + n \\ &= n\,(c+1) \\ &\not\leq cn \end{aligned}$$

From the above induction, we understood that $\Theta(n)$ is too small and $\Theta(nlogn)$ is too big. So, we need something bigger than n and smaller than $nlogn$. How about $n\sqrt{logn}$?

Proving the upper bound for $n\sqrt{logn}$:

$$\begin{aligned} T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\ &\leq \sqrt{n}.c.\sqrt{n}\sqrt{log\sqrt{n}} + n \\ &= n.\ c.\frac{1}{\sqrt{2}}\ log\sqrt{n} + n \\ &\leq cnlog\sqrt{n} \end{aligned}$$

Proving the lower bound for $n\sqrt{logn}$:

$$\begin{aligned} T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\ &\geq \sqrt{n}.k.\sqrt{n}\sqrt{log\sqrt{n}} + n \\ &= n.\ k.\frac{1}{\sqrt{2}}\ log\sqrt{n} + n \\ &\not\geq knlog\sqrt{n} \end{aligned}$$

The last step doesn't work. So, $\Theta(n\sqrt{logn})$ doesn't work. What else is between n and $nlogn$? How about $nloglogn$?

Proving upper bound for $nloglogn$:

$$\begin{aligned} T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\ &\leq \sqrt{n}.c.\sqrt{n}loglog\sqrt{n} + n \\ &= n.\ c.loglogn - c.n + n \\ &\leq cnloglogn, \text{ if } c \geq 1 \end{aligned}$$

Proving lower bound for $nloglogn$:

$$\begin{aligned} T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\ &\geq \sqrt{n}.k.\sqrt{n}loglog\sqrt{n} + n \\ &= n.\ k.loglogn - k.n + n \\ &\geq knloglogn, \text{ if } k \leq 1 \end{aligned}$$

From the above proofs, we can see that $T(n) \leq cnloglogn$, if $c \geq 1$ and $T(n) \geq knloglogn$, if $k \leq 1$. Technically, we're still missing the base cases in both proofs, but we can be fairly confident at this point that $T(n) = \Theta(nloglogn)$.

1.26 Amortized Analysis

Amortized analysis refers to determining the time-averaged running time for a sequence of operations. It is different from average case analysis, because amortized analysis does not make any assumption about the distribution of the data values, whereas average case analysis assumes the data are not "bad" (e.g., some sorting algorithms do well *on average* over all input orderings but very badly on certain input orderings). That is, amortized analysis is a worst-case analysis, but for a sequence of operations rather than for individual operations.

The motivation for amortized analysis is to better understand the running time of certain techniques, where standard worst case analysis provides an overly pessimistic bound. Amortized analysis generally applies to a method that consists of a sequence of operations, where the vast majority of the operations are cheap, but some of the operations are expensive. If we can show that the expensive operations are particularly rare we can *change them* to the cheap operations, and only bound the cheap operations.

The general approach is to assign an artificial cost to each operation in the sequence, such that the total of the artificial costs for the sequence of operations bounds the total of the real costs for the sequence. This artificial cost is called the amortized cost of an operation. To analyze the running time, the amortized cost thus is a correct way of understanding the overall running time — but note that particular operations can still take longer so it is not a way of bounding the running time of any individual operation in the sequence.

When one event in a sequence affects the cost of later events:

- One particular task may be expensive.
- But it may leave data structure in a state that the next few operations become easier.

Example: Let us consider an array of elements from which we want to find the k^{th} smallest element. We can solve this problem using sorting. After sorting the given array, we just need to return the k^{th} element from it. The cost of performing the sort (assuming comparison based sorting algorithm) is $O(nlogn)$. If we perform n such selections then the average cost of each selection is $O(nlogn/n) = O(logn)$. This clearly indicates that sorting once is reducing the complexity of subsequent operations.

1.27 Algorithms Analysis: Problems & Solutions

Note: From the following problems, try to understand the cases which have different complexities ($O(n), O(logn), O(loglogn)$ etc.).

Problem-21 Find the complexity of the below recurrence:

$$T(n) = \begin{cases} 3T(n-1), if\ n > 0, \\ 1, \quad otherwise \end{cases}$$

Solution: Let us try solving this function with substitution.

$T(n) = 3T(n-1)$

$T(n) = 3(3T(n-2)) = 3^2T(n-2)$

$T(n) = 3^2(3T(n-3))$

$T(n) = 3^nT(n-n) = 3^nT(0) = 3^n$

This clearly shows that the complexity of this function is $O(3^n)$.

Note: We can use the *Subtraction and Conquer* master theorem for this problem.

Problem-22 Find the complexity of the below recurrence:

$$T(n) = \begin{cases} 2T(n-1) - 1, if\ n > 0, \\ 1, \quad otherwise \end{cases}$$

Solution: Let us try solving this function with substitution.

$T(n) = 2T(n-1) - 1$

$T(n) = 2(2T(n-2) - 1) - 1 = 2^2T(n-2) - 2 - 1$

$T(n) = 2^2(2T(n-3) - 2 - 1) - 1 = 2^3T(n-4) - 2^2 - 2^1 - 2^0$

$T(n) = 2^nT(n-n) - 2^{n-1} - 2^{n-2} - 2^{n-3} \ldots . 2^2 - 2^1 - 2^0$

$T(n) = 2^n - 2^{n-1} - 2^{n-2} - 2^{n-3} \ldots . 2^2 - 2^1 - 2^0$

$T(n) = 2^n - (2^n - 1)$ $[note: 2^{n-1} + 2^{n-2} + \cdots + 2^0 = 2^n]$

$T(n) = 1$

∴ Time Complexity is O(1). Note that while the recurrence relation looks exponential, the solution to the recurrence relation here gives a different result.

Problem-23 What is the running time of the following function?

```
def Function(n):
        i = s = 1
        while  s < n:
                i = i+1
                s = s+i
                print("*")
Function(20)
```

Solution: Consider the comments in the below function:

```
def Function(n):
    i = s = 1
    while  s < n:                    # s is increasing not at rate 1 but i
            i = i+1
            s = s+i
            print("*")
Function(20)
```

We can define the 's' terms according to the relation $s_i = s_{i-1} + i$. The value of 'i' increases by 1 for each iteration. The value contained in 's' at the i^{th} iteration is the sum of the first 'i' positive integers. If k is the total number of iterations taken by the program, then the *while* loop terminates if:

$$1 + 2 + \ldots + k = \frac{k(k+1)}{2} > n \Rightarrow k = O(\sqrt{n}).$$

Problem-24 Find the complexity of the function given below.

```
def Function(n):
    i = 1
    count = 0
    while i*i <n:
        count = count +1
        i = i + 1
    print(count)
Function(20)
```

Solution: In the above-mentioned function the loop will end, if $i^2 \leq n \Rightarrow T(n) = O(\sqrt{n})$. This is similar to Problem-23.

Problem-25 What is the complexity of the program given below:

```
def Function(n):
    count = 0
    for i in range(n/2, n):
        j = 1
        while j + n/2 <= n:
            k = 1
            while k <= n:
                count = count + 1
                k = k * 2
            j = j + 1
    print (count)
Function(20)
```

Solution: Observe the comments in the following function.

```
def Function(n):
    count = 0
    for i in range(n/2, n):                 #Outer loop execute n/2 times
        j = 1
        while j + n/2 <= n:                 #Middle loop executes n/2 times
            k = 1
            while k <= n:                   #Inner loop execute logn times
                count = count + 1
                k = k * 2
            j = j + 1
    print (count)

Function(20)
```

The complexity of the above function is $O(n^2 logn)$.

Problem-26 What is the complexity of the program given below:

```
def Function(n):
    count = 0
    for i in range(n/2, n):
        j = 1
        while j + n/2 <= n:
            k = 1
            while k <= n:
                count = count + 1
                k = k * 2
            j = j * 2
    print (count)
Function(20)
```

Solution: Consider the comments in the following function.

```
def Function(n):
    count = 0
    for i in range(n/2, n):                 #Outer loop execute n/2 times
        j = 1
        while j + n/2 <= n:                 #Middle loop executes logn times
```

```
                k = 1
                while k <= n:              #Inner loop execute logn times
                        count = count + 1
                        k = k * 2
                j = j * 2
    print (count)
Function(20)
```

The complexity of the above function is $O(nlog^2n)$.

Problem-27 Find the complexity of the program below.

```
def Function(n):
        count = 0
        for i in range(n/2, n):
                j = 1
                while j + n/2 <= n:
                        break
                        j = j * 2
        print (count)
Function(20)
```

Solution: Consider the comments in the function below.

```
def Function(n):
    count = 0
    for i in range(n/2, n):                #Outer loop execute n/2 times
            j = 1
            while j + n/2 <= n:            #Middle loop has break statement
                    break
                    j = j * 2
    print (count)
Function(20)
```

The complexity of the above function is $O(n)$. Even though the inner loop is bounded by n, but due to the break statement it is executing only once.

Problem-28 Write a recursive function for the running time $T(n)$ of the function given below. Prove using the iterative method that $T(n) = \Theta(n^3)$.

```
def Function(n):
        count = 0
        if n <= 0:
                return
        for i in range(0, n):
                for j in range(0, n):
                        count = count + 1
        Function(n-3)
        print (count)
Function(20)
```

Solution: Consider the comments in the function below:

```
def Function(n):
    count = 0
    if n <= 0:
            return
    for i in range(0, n):                  #Outer loop executes n times
            for j in range(0, n):          #Outer loop executes n times
                    count = count + 1
    Function(n-3)                          #Recursive call
    print (count)
Function(20)
```

The recurrence for this code is clearly $T(n) = T(n - 3) + cn^2$ for some constant $c > 0$ since each call prints out n^2 asterisks and calls itself recursively on n - 3. Using the iterative method we get: $T(n) = T(n - 3) + cn^2$. Using the *Subtraction and Conquer* master theorem, we get $T(n) = \Theta(n^3)$.

Problem-29 Determine Θ bounds for the recurrence relation: $T(n) = 2T\left(\frac{n}{2}\right) + nlogn$.

Solution: Using Divide and Conquer master theorem, we get: $O(nlog^2n)$.

Problem-30 Determine Θ bounds for the recurrence: $T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{4}\right) + T\left(\frac{n}{8}\right) + n$.

Solution: Substituting in the recurrence equation, we get: $T(n) \leq c1 * \frac{n}{2} + c2 * \frac{n}{4} + c3 * \frac{n}{8} + cn \leq k * n$, where k is a constant. This clearly says $\Theta(n)$.

Problem-31 Determine Θ bounds for the recurrence relation: $T(n) = T(\lceil n/2 \rceil) + 7$.

Solution: Using Master Theorem we get: $\Theta(logn)$.

Problem-32 Prove that the running time of the code below is $\Omega(logn)$.

```
def Read(n):
        k = 1
        while  k < n:
                k = 3*k
```

Solution: The *while* loop will terminate once the value of 'k' is greater than or equal to the value of 'n'. In each iteration the value of 'k' is multiplied by 3. If i is the number of iterations, then 'k' has the value of $3i$ after i iterations. The loop is terminated upon reaching i iterations when $3^i \geq n \leftrightarrow i \geq \log_3 n$, which shows that $i = \Omega(logn)$.

Problem-33 Solve the following recurrence.

$$T(n) = \begin{cases} 1, & if\ n = 1 \\ T(n-1) + n(n-1), & if\ n \geq 2 \end{cases}$$

Solution: By iteration:

$$T(n) = T(n-2) + (n-1)(n-2) + n(n-1)$$

...

$$T(n) = T(1) + \sum_{i=1}^{n} i(i-1)$$

$$T(n) = T(1) + \sum_{i=1}^{n} i^2 - \sum_{i=1}^{n} i$$

$$T(n) = 1 + \frac{n((n+1)(2n+1)}{6} - \frac{n(n+1)}{2}$$

$$T(n) = \Theta(n^3)$$

Note: We can use the *Subtraction and Conquer* master theorem for this problem.

Problem-34 Consider the following program:

```
def Fib(n):
        if n == 0: return 0
        elif n == 1: return 1
        else: return Fib(n-1)+ Fib(n-2)
print(Fib(3))
```

Solution: The recurrence relation for the running time of this program is: $T(n) = T(n-1) + T(n-2) + c$. Note T($n$) has two recurrence calls indicating a binary tree. Each step recursively calls the program for n reduced by 1 and 2, so the depth of the recurrence tree is O(n). The number of leaves at depth n is 2^n since this is a full binary tree, and each leaf takes at least O(1) computations for the constant factor. Running time is clearly exponential in n and it is O(2^n).

Problem-35 Running time of following program?

```
def Function(n):
        count = 0
        if n <= 0:
                return
        for i in range(0, n):
                j = 1
                while j <n:
                        j = j + i
                        count = count + 1
        print (count)
Function(20)
```

Solution: Consider the comments in the function below:

```
def Function(n):
    count = 0
    if n <= 0:
        return
    for i in range(0, n):          #Outer loop executes n times
        j = 1                      #Inner loop executes j increase by the rate of i
        while j <n:
            j = j + i
            count = count + 1
    print (count)
Function(20)
```

In the above code, inner loop executes n/i times for each value of i. Its running time is $n \times (\sum_{i=1}^{n} n/i) = O(nlogn)$.

Problem-36 What is the complexity of $\sum_{i=1}^{n} log\, i$?

Solution: Using the logarithmic property, $logxy = logx + logy$, we can see that this problem is equivalent to

$$\sum_{i=1}^{n} logi = log\,1 + log\,2 + \cdots + log\,n = log(1 \times 2 \times \ldots \times n) = log(n!) \leq log(n^n) \leq nlogn$$

This shows that the time complexity = $O(nlogn)$.

Problem-37 What is the running time of the following recursive function (specified as a function of the input value n)? First write the recurrence formula and then find its complexity.

```
def Function(n):
    if n <= 0:
        return
    for i in range(0, 3):
        Function(n/3)
Function(20)
```

Solution: Consider the comments in the below function:

```
def Function(n):
    if n <= 0:
        return
    for i in range(0, 3):          #This loop executes 3 times with recursive value of n/3 value
        Function(n/3)
Function(20)
```

We can assume that for asymptotical analysis $k = \lceil k \rceil$ for every integer $k \geq 1$. The recurrence for this code is $T(n) = 3T(\frac{n}{3}) + \Theta(1)$. Using master theorem, we get $T(n) = \Theta(n)$.

Problem-38 What is the running time of the following recursive function (specified as a function of the input value n)? First write a recurrence formula, and show its solution using induction.

```
def Function(n):
    if n <= 0:
        return
    for i in range(0, 3):          #This loop executes 3 times with recursive value of n/3 value
        Function(n-1)
Function(20)
```

Solution: Consider the comments in the function below:

```
def Function(n):
    if n <= 0:
        return
    for i in range(0, 3):          #This loop executes 3 times with recursive value of n – 1 value
        Function(n-1)
Function(20)
```

The *if* statement requires constant time [O(1)]. With the *for* loop, we neglect the loop overhead and only count three times that the function is called recursively. This implies a time complexity recurrence:

$$T(n) = c, if\ n \leq 1;$$
$$= c + 3T(n-1), if\ n > 1.$$

Using the *Subtraction and Conquer* master theorem, we get $T(n) = \Theta(3^n)$.

Problem-39 Write a recursion formula for the running time $T(n)$ of the function whose code is below.

```
def Function3(n):
    if n <= 0:
        return
    for i in range(0, 3):        #This loop executes 3 times with recursive value of n/3 value
        Function3(0.8 * n)
Function3(20)
```

Solution: Consider the comments in the function below:

```
def Function3(n):
    if n <= 0:
        return
    for i in range(0, 3):        #This loop executes 3 times with recursive value of 0.8n value
        Function3(0.8 * n)
Function3(20)
```

The recurrence for this piece of code is $T(n) = T(.8n) + O(n) = T(4/5n) + O(n) = 4/5\ T(n) + O(n)$. Applying master theorem, we get $T(n) = O(n)$.

Problem-40 Find the complexity of the recurrence: $T(n) = 2T(\sqrt{n}) + logn$

Solution: The given recurrence is not in the master theorem format. Let us try to convert this to the master theorem format by assuming $n = 2^m$. Applying the logarithm on both sides gives, $logn = mlog2 \Rightarrow m = logn$. Now, the given function becomes:

$$T(n) = T(2^m) = 2T\left(\sqrt{2^m}\right) + m = 2T\left(2^{\frac{m}{2}}\right) + m.$$

To make it simple we assume $S(m) = T(2^m) \Rightarrow S(\frac{m}{2}) = T(2^{\frac{m}{2}}) \Rightarrow S(m) = 2S\left(\frac{m}{2}\right) + m$.
Applying the master theorem format would result in $S(m) = O(mlogm)$.
If we substitute $m = logn$ back, $T(n) = S(logn) = O((logn)\ loglogn)$.

Problem-41 Find the complexity of the recurrence: $T(n) = T(\sqrt{n}) + 1$

Solution: Applying the logic of Problem-40 gives $S(m) = S\left(\frac{m}{2}\right) + 1$. Applying the master theorem would result in $S(m) = O(logm)$. Substituting $m = logn$, gives $T(n) = S(logn) = O(loglogn)$.

Problem-42 Find the complexity of the recurrence: $T(n) = 2T(\sqrt{n}) + 1$

Solution: Applying the logic of Problem-40 gives: $S(m) = 2S\left(\frac{m}{2}\right) + 1$. Using the master theorem results $S(m) = O\left(m^{log\frac{2}{2}}\right) = O(m)$. Substituting $m = logn$ gives $T(n) = O(logn)$.

Problem-43 Find the complexity of the below function.

```
import math
count = 0
def Function(n):
    global count
    if n <= 2:
        return 1
    else:
        Function(round(math.sqrt(n)))
        count = count + 1
        return count
print(Function(200))
```

Solution: Consider the comments in the function below:

```
import math
count = 0
def Function(n):
    global count
    if n <= 2:
        return 1
    else:
        Function(round(math.sqrt(n)))    #Recursive call with √n value
        count = count + 1
        return count
print(Function(200))
```

For the above code, the recurrence function can be given as: $T(n) = T(\sqrt{n}) + 1$. This is same as that of Problem-41.

Problem-44 Analyze the running time of the following recursive pseudo-code as a function of n.

```
def function(n):
        if (n < 2):
                return
        else:
                counter = 0
        for i in range(0,8):
                function (n/2)
        for i in range(0,n**3):
                counter = counter + 1
```

Solution: Consider the comments in below pseudo-code and call running time of function(n) as $T(n)$.

```
def function(n):
    if (n < 2):                                  # Constant time
       return
    else:
            counter = 0                  # Constant time
    for i in range(0,8):                 # This loop executes 8 times with n value half in every call
            function (n/2)
    for i in range(0,n**3):              # This loop executes n^3 times with constant time loop
                    counter = counter + 1
```

$T(n)$ can be defined as follows:

$T(n) = 1$ *if* $n < 2$,

$= 8T(\frac{n}{2}) + n^3 + 1$ *otherwise.*

Using the master theorem gives: $T(n) = \Theta(n^{log_2^8} logn) = \Theta(n^3 logn)$.

Problem-45 Find the complexity of the below pseudocode.

```
count = 0
def Function(n):
        global count
        count  = 1
        if n <= 0:
                return
        for i in range(0, n):
                count =  count + 1
        n  = n//2;
        Function(n)
        print count

Function(200)
```

Solution: Consider the comments in the pseudocode below:

```
count = 0
def Function(n):
    global count
    count  = 1
    if n <= 0:
            return
    for i in range(1, n):        # This loops executes n times
            count =  count + 1
    n  = n//2;          #Integer Divison
    Function(n)         #Recursive call with n/2 value
    print count
Function(200)
```

The recurrence for this function is $T(n) = T(n/2) + n$. Using master theorem we get $T(n) = O(n)$.

Problem-46 Running time of the following program?

```
def Function(n):
        for i in range(1, n):
                j = 1
```

```
            while j <= n:
                j = j * 2
                print("*")
    Function(20)
```

Solution: Consider the comments in the below function:

```
def Function(n):
    for i in range(1, n):       # This loops executes n times
        j = 1
        while j <= n:           # This loops executes logn times from our logarithms guideline
            j = j * 2
            print("*")
Function(20)
```

Complexity of above program is: $O(nlogn)$.

Problem-47 Running time of the following program?

```
def Function(n):
    for i in range(0, n/3):
        j = 1
        while j <= n:
            j = j + 4
            print("*")

Function(20)
```

Solution: Consider the comments in the below function:

```
def Function(n):
    for i in range(0, n/3):     #This loops executes n/3 times
        j = 1
        while j <= n:           #This loops executes n/4 times
            j = j + 4
            print("*")
Function(20)
```

The time complexity of this program is: $O(n^2)$.

Problem-48 Find the complexity of the below function:

```
def Function(n):
    if n <= 0:
        return
    print ("*")
    Function(n/2)
    Function(n/2)
    print ("*")
Function(20)
```

Solution: Consider the comments in the below function:

```
def Function(n):
    if n <= 0:          #Constant time
        return
    print ("*")         #Constant time
    Function(n/2)       #Recursion with n/2 value
    Function(n/2)       #Recursion with n/2 value
    print ("*")
Function(20)
```

The recurrence for this function is: $T(n) = 2T\left(\frac{n}{2}\right) + 1$. Using master theorem, we get $T(n) = O(n)$.

Problem-49 Find the complexity of the below function:

```
count = 0
def Logarithms(n):
    i = 1
    global count
    while i <= n:
```

```
                    j = n
                    while j > 0:
                            j = j//2
                            count = count + 1
                    i= i * 2
            return count
print(Logarithms(10))
```

Solution:

```
count = 0
def Logarithms(n):
    i = 1
    global count
    while i <= n:
            j = n
            while j > 0:
                    j = j//2 # This loops executes logn times from our logarithms guideline
                    count = count + 1
            i= i * 2         # This loops executes logn times from our logarithms guideline
    return count

print(Logarithms(10))
```

Time Complexity: $O(logn * logn) = O(log^2 n)$.

Problem-50 $\sum_{1 \le k \le n} O(n)$, where $O(n)$ stands for order n is:
(a) $O(n)$ (b) $O(n^2)$ (c) $O(n^3)$ (d) $O(3n^2)$ (e) $O(1.5n^2)$

Solution: (b). $\sum_{1 \le k \le n} O(n) = O(n) \sum_{1 \le k \le n} 1 = O(n^2)$.

Problem-51 Which of the following three claims are correct?
I $(n + k)^m = \Theta(n^m)$, where k and m are constants II $2^{n+1} = O(2^n)$ III $2^{2n+1} = O(2^n)$
(a) I and II (b) I and III (c) II and III (d) I, II and III

Solution: (a). (I) $(n + k)^m = n^k + c1*n^{k-1} + \ldots k^m = \Theta(n^k)$ and (II) $2^{n+1} = 2*2^n = O(2^n)$

Problem-52 Consider the following functions:
$f(n) = 2^n$ $g(n) = n!$ $h(n) = n^{logn}$
Which of the following statements about the asymptotic behavior of f(n), g(n), and h(n) is true?
(A) f(n) = O(g(n)); g(n) = O(h(n)) (B) f(n) = Ω (g(n)); g(n) = O(h(n))
(C) g(n) = O(f(n)); h(n) = O(f(n)) (D) h(n) = O(f(n)); g(n) = Ω (f(n))

Solution: **(D)**. According to the rate of growth: h(n) < f(n) < g(n) (g(n) is asymptotically greater than f(n), and f(n) is asymptotically greater than h(n)). We can easily see the above order by taking logarithms of the given 3 functions: $lognlogn < n < log(n!)$. Note that, $log(n!) = O(nlogn)$.

Problem-53 Consider the following segment of C-code:

```
j = 1
while j <=n:
        j = j*2
```

The number of comparisons made in the execution of the loop for any $n > 0$ is:
(A) ceil(log_2^n)+ 1 (B) n (C) ceil(log_2^n) (D) floor(log_2^n) + 1

Solution: (a). Let us assume that the loop executes k times. After k^{th} step the value of j is 2^k. Taking logarithms on both sides gives $k = log_2^n$. Since we are doing one more comparison for exiting from the loop, the answer is ceil(log_2^n)+ 1.

Problem-54 Consider the following C code segment. Let T(n) denote the number of times the for loop is executed by the program on input n. Which of the following is true?

```
import math
def IsPrime(n):
    for i in range(2, math.sqrt(n)):
        if n%i == 0:
                print("Not Prime")
                return 0
    return 1
```

(A) $T(n) = O(\sqrt{n})$ and $T(n) = \Omega(\sqrt{n})$ (B) $T(n) = O(\sqrt{n})$ and $T(n) = \Omega(1)$
(C) $T(n) = O(n)$ and $T(n) = \Omega(\sqrt{n})$ (D) None of the above

Solution: (**B**). Big O notation describes the tight upper bound and Big Omega notation describes the tight lower bound for an algorithm. The *for* loop in the question is run maximum $\sqrt{n}$ times and minimum 1 time. Therefore, $T(n) = O(\sqrt{n})$ and $T(n) = \Omega(1)$.

Problem-55 In the following C function, let $n \geq m$. How many recursive calls are made by this function?

```
def gcd(n,m):
    if n%m ==0:
        return m
    n = n%m
    return gcd(m,n)
```

(A) $\Theta(log_2^n)$ (B) $\Omega(n)$ (C) $\Theta(log_2 log_2^n)$ (D) $\Theta(n)$

Solution: No option is correct. Big O notation describes the tight upper bound and Big Omega notation describes the tight lower bound for an algorithm. For $m = 2$ and for all $n = 2^i$, the running time is O(1) which contradicts every option.

Problem-56 Suppose $T(n) = 2T(n/2) + n$, T(0)=T(1)=1. Which one of the following is false?

(A) $T(n) = O(n^2)$ (B) $T(n) = \Theta(nlogn)$ (C) $T(n) = \Omega(n^2)$ (D) $T(n) = O(nlogn)$

Solution: (**C**). Big O notation describes the tight upper bound and Big Omega notation describes the tight lower bound for an algorithm. Based on master theorem, we get $T(n) = \Theta(nlogn)$. This indicates that tight lower bound and tight upper bound are the same. That means, $O(nlogn)$ and $\Omega(nlogn)$ are correct for given recurrence. So option (C) is wrong.

Problem-57 Find the complexity of the below function:

```
def Function(n):
    for i in range(1, n):
        j = i
        while j <i*i:
            j = j + 1
            if j %i == 0:
                for k in range(0, j):
                    print(" * ")
Function(10)
```

Solution:

```
def Function(n):
    for i in range(1, n):                  # Executes n times
        j = i
        while j <i*i:                      # Executes n*n times
            j = j + 1
            if j %i == 0:
                for k in range(0, j):      #Executes j times = (n*n) times
                    print(" * ")
Function(10)
```

Time Complexity: $O(n^5)$.

Problem-58 To calculate 9^n, give an algorithm and discuss its complexity.

Solution: Start with 1 and multiply by 9 until reaching 9^n.

Time Complexity: There are $n - 1$ multiplications and each takes constant time giving a $\Theta(n)$ algorithm.

Problem-59 For Problem-58, can we improve the time complexity?

Solution: Refer to the *Divide and Conquer* chapter.

Problem-60 Find the complexity of the below function:

```
def Function(n):
    sum = 0
    for i in range(0, n-1):
        if i > j:
            sum = sum + 1
        else:
            for k in range(0, j):
                sum = sum - 1
```

```
    print (sum)
Function(10)
```

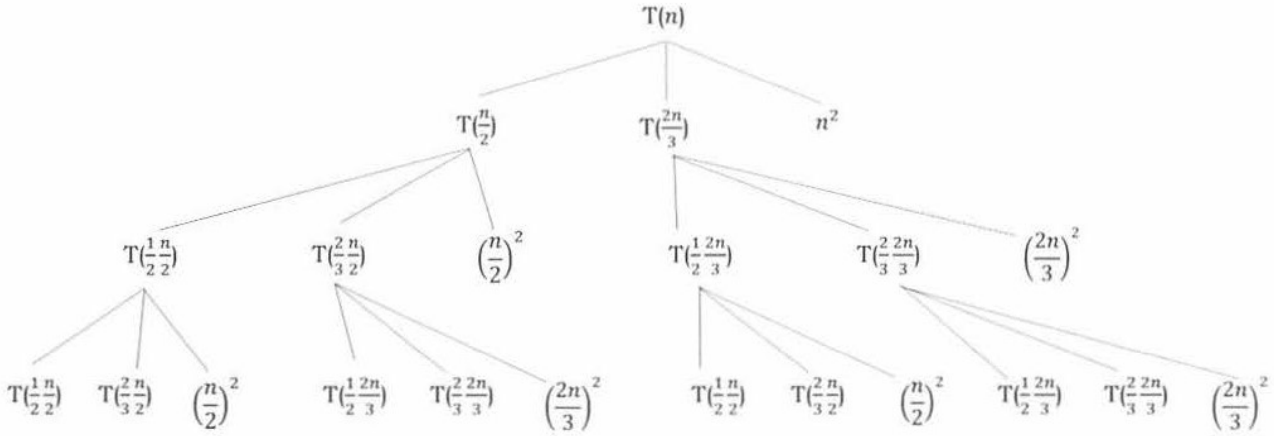
Solution: Consider the *worst – case* and we can ignore the value of j.

```
def Function(n):
    sum = 0
    for i in range(0, n-1):                    # Executes n times
        if i > j:
            sum = sum + 1                      # Executes n times
        else:
            for k in range(0, j):              # Executes n times
                sum = sum - 1
    print (sum)
Function(10)
```

Time Complexity: $O(n^2)$.

Problem-61 Solve the following recurrence relation using the recursion tree method: $T(n)=T(\frac{n}{2})+T(\frac{2n}{3})+n^2$.

Solution: How much work do we do in each level of the recursion tree?



In level 0, we take n^2 time. At level 1, the two subproblems take time:

$$\left(\frac{1}{2}n\right)^2+\left(\frac{2}{3}n\right)^2=\left(\frac{1}{4}+\frac{4}{9}\right)n^2=\left(\frac{25}{36}\right)n^2$$

At level 2 the four subproblems are of size $\frac{n}{2}\frac{n}{2}$, $\frac{2}{3}\frac{n}{2}$, $\frac{1}{2}\frac{2n}{3}$, and $\frac{2}{3}\frac{2n}{3}$ respectively. These two subproblems take time:

$$\left(\frac{1}{4}n\right)^2+\left(\frac{1}{3}n\right)^2+\left(\frac{1}{3}\right)n^2+\left(\frac{4}{9}\right)n^2=\frac{625}{1296}n^2=\left(\frac{25}{36}\right)^2n^2$$

Similarly the amount of work at level k is at most $\left(\frac{25}{36}\right)^k n^2$.

Let $\alpha=\frac{25}{36}$, the total runtime is then:

$$\begin{aligned}
T(n) &\leq \sum_{k=0}^{\infty}\alpha^k n^2 \\
&= \frac{1}{1-\alpha}n^2 \\
&= \frac{1}{1-\frac{25}{36}}n^2 \\
&= \frac{1}{\frac{11}{36}}n^2 \\
&= \frac{36}{11}n^2 \\
&= O(n^2)
\end{aligned}$$

That is, the first level provides a constant fraction of the total runtime.

CHAPTER

2

RECURSION AND BACKTRACKING

2.1 Introduction

In this chapter, we will look at one of the important topics, "*recursion*", which will be used in almost every chapter, and also its relative "*backtracking*".

2.2 What is Recursion?

Any function which calls itself is called *recursive*. A recursive method solves a problem by calling a copy of itself to work on a smaller problem. This is called the recursion step. The recursion step can result in many more such recursive calls.

It is important to ensure that the recursion terminates. Each time the function calls itself with a slightly simpler version of the original problem. The sequence of smaller problems must eventually converge on the base case.

2.3 Why Recursion?

Recursion is a useful technique borrowed from mathematics. Recursive code is generally shorter and easier to write than iterative code. Generally, loops are turned into recursive functions when they are compiled or interpreted.

Recursion is most useful for tasks that can be defined in terms of similar subtasks. For example, sort, search, and traversal problems often have simple recursive solutions.

2.4 Format of a Recursive Function

A recursive function performs a task in part by calling itself to perform the subtasks. At some point, the function encounters a subtask that it can perform without calling itself. This case, where the function does not recur, is called the *base case*. The former, where the function calls itself to perform a subtask, is referred to as the *cursive case*. We can write all recursive functions using the format:

```
if(test for the base case)
    return some base case value
else if(test for another base case)
    return some other base case value
// the recursive case
else
    return (some work and then a recursive call)
```

As an example consider the factorial function: $n!$ is the product of all integers between n and 1. The definition of recursive factorial looks like:

$$n! = 1, \quad \text{if } n = 0$$
$$n! = n * (n-1)! \text{ if } n > 0$$

This definition can easily be converted to recursive implementation. Here the problem is determining the value of $n!$, and the subproblem is determining the value of $(n-1)!$. In the recursive case, when n is greater than 1, the function calls itself to determine the value of $(n-1)!$ and multiplies that with n.

In the base case, when n is 0 or 1, the function simply returns 1. This looks like the following:

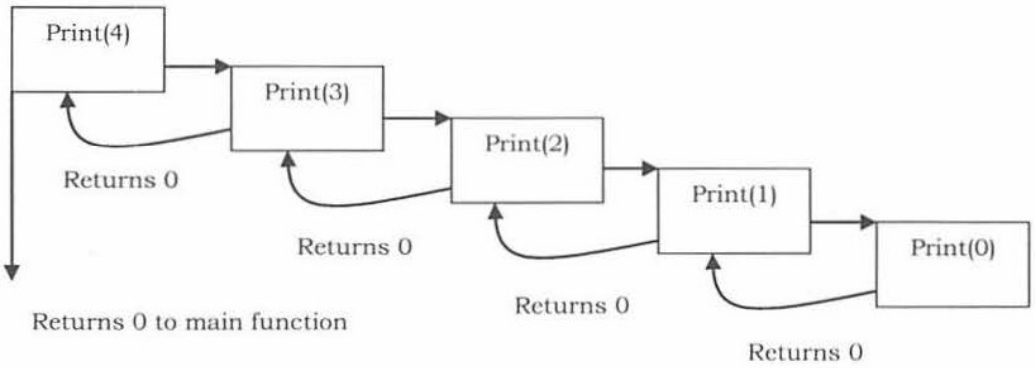
```
// calculates factorial of a positive integer
def factorial(n):
 if n == 0: return 1
 return n*factorial(n-1)
print(factorial(6))
```

2.5 Recursion and Memory (Visualization)

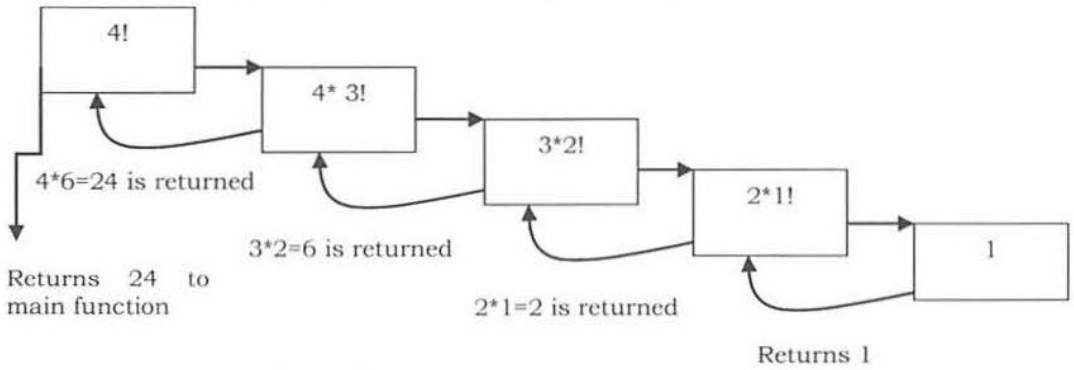
Each recursive call makes a new copy of that method (actually only the variables) in memory. Once a method ends (that is, returns some data), the copy of that returning method is removed from memory. The recursive solutions look simple but visualization and tracing takes time. For better understanding, let us consider the following example.

```
def Print(n):
 if n == 0:                    # this is the terminating base case
    return 0
 else:
        print n
        return Print(n-1)      # recursive call to itself again
print(Print(4))
```

For this example, if we call the print function with n=4, visually our memory assignments may look like:



Now, let us consider our factorial function. The visualization of factorial function with n=4 will look like:



2.6 Recursion versus Iteration

While discussing recursion, the basic question that comes to mind is: which way is better? – iteration or recursion? The answer to this question depends on what we are trying to do. A recursive approach mirrors the problem that we are trying to solve. A recursive approach makes it simpler to solve a problem that may not have the most obvious of answers. But, recursion adds overhead for each recursive call (needs space on the stack frame).

Recursion

- Terminates when a base case is reached.
- Each recursive call requires extra space on the stack frame (memory).
- If we get infinite recursion, the program may run out of memory and result in stack overflow.
- Solutions to some problems are easier to formulate recursively.

Iteration

- Terminates when a condition is proven to be false.
- Each iteration does not require extra space.
- An infinite loop could loop forever since there is no extra memory being created.
- Iterative solutions to a problem may not always be as obvious as a recursive solution.

2.7 Notes on Recursion

- Recursive algorithms have two types of cases, recursive cases and base cases.
- Every recursive function case must terminate at a base case.
- Generally, iterative solutions are more efficient than recursive solutions [due to the overhead of function calls].
- A recursive algorithm can be implemented without recursive function calls using a stack, but it's usually more trouble than its worth. That means any problem that can be solved recursively can also be solved iteratively.
- For some problems, there are no obvious iterative algorithms.
- Some problems are best suited for recursive solutions while others are not.

2.8 Example Algorithms of Recursion

- Fibonacci Series, Factorial Finding
- Merge Sort, Quick Sort
- Binary Search
- Tree Traversals and many Tree Problems: InOrder, PreOrder PostOrder
- Graph Traversals: DFS [Depth First Search] and BFS [Breadth First Search]
- Dynamic Programming Examples
- Divide and Conquer Algorithms
- Towers of Hanoi
- Backtracking Algorithms [we will discuss in next section]

2.9 Recursion: Problems & Solutions

In this chapter we cover a few problems with recursion and we will discuss the rest in other chapters. By the time you complete reading the entire book, you will encounter many recursion problems.

Problem-1 Discuss Towers of Hanoi puzzle.

Solution: The Towers of Hanoi is a mathematical puzzle. It consists of three rods (or pegs or towers) and a number of disks of different sizes which can slide onto any rod. The puzzle starts with the disks on one rod in ascending order of size, the smallest at the top, thus making a conical shape. The objective of the puzzle is to move the entire stack to another rod, satisfying the following rules:

- Only one disk may be moved at a time.
- Each move consists of taking the upper disk from one of the rods and sliding it onto another rod, on top of the other disks that may already be present on that rod.
- No disk may be placed on top of a smaller disk.

Algorithm:

- Move the top $n-1$ disks from *Source* to *Auxiliary* tower,
- Move the n^{th} disk from *Source* to *Destination* tower,
- Move the $n-1$disks from *Auxiliary* tower to *Destination* tower.
- Transferring the top $n-1$ disks from *Source* to *Auxiliary* tower can again be thought of as a fresh problem and can be solved in the same manner. Once we solve *Towers of Hanoi* with three disks, we can solve it with any number of disks with the above algorithm.

```
def TowersOfHanoi(numberOfDisks, startPeg=1, endPeg=3):
    if numberOfDisks:
        TowersOfHanoi(numberOfDisks-1, startPeg, 6-startPeg-endPeg)
        print "Move disk %d from peg %d to peg %d" % (numberOfDisks, startPeg, endPeg)
        TowersOfHanoi(numberOfDisks-1, 6-startPeg-endPeg, endPeg)
TowersOfHanoi(numberOfDisks=4)
```

Problem-2 Given an array, check whether the array is in sorted order with recursion.

Solution:

```
def isArrayInSortedOrder(A):
    # Base case
    if len(A) == 1:
        return True
    return A[0] <= A[1] and isSorted(A[1:])
A = [127, 220, 246, 277, 321, 454, 534, 565, 933]
print(isArrayInSortedOrder(A))
```

Time Complexity: O(n). Space Complexity: O(n) for recursive stack space.

2.10 What is Backtracking?

Backtracking is a form of recursion.The usual scenario is that you are faced with a number of options, and you must choose one of these. After you make your choice you will get a new set of options; just what set of options you get depends on what choice you made. This procedure is repeated over and over until you reach a final state. If you made a good sequence of choices, your final state is a goal state; if you didn't, it isn't. Backtracking is a method of exhaustive search using divide and conquer.

- Sometimes the best algorithm for a problem is to try all possibilities.
- This is always slow, but there are standard tools that can be used to help.
- Tools: algorithms for generating basic objects, such as binary strings [2^n possibilities for n-bit string], permutations [$n!$], combinations [$n!/r!(n-r)!$], general strings [k −ary strings of length n has k^n possibilities], etc...
- Backtracking speeds the exhaustive search by pruning.

2.11 Example Algorithms of Backtracking

- Binary Strings: generating all binary strings
- Generating k −ary Strings
- The Knapsack Problem
- Generalized Strings
- Hamiltonian Cycles [refer *Graphs* chapter]
- Graph Coloring Problem

2.12 Backtracking: Problems & Solutions

Problem-3 Generate all the binary strings with n bits. Assume $A[0..n-1]$ is an array of size n.

Solution:

```
def appendAtBeginningFront(x, L):
    return [x + element for element in L]
def bitStrings(n):
    if n == 0: return []
    if n == 1: return ["0", "1"]
    else:
        return (appendAtBeginningFront("0", bitStrings(n-1)) + appendAtBeginningFront("1", bitStrings(n-1)))
print bitStrings(4)
```

Alternative Approach:

```
def bitStrings(n):
    if n == 0: return []
    if n == 1: return ["0", "1"]
    return [ digit+bitstring for digit in bitStrings(1)
```

```
                for bitstring in bitStrings(n-1)]
print bitStrings(4)
```

Let $T(n)$ be the running time of *binary*(n). Assume function *printf* takes time O(1).

$$T(n) = \begin{cases} c, & \text{if } n < 0 \\ 2T(n-1) + d, & \text{otherwise} \end{cases}$$

Using Subtraction and Conquer Master theorem we get: $T(n) = O(2^n)$. This means the algorithm for generating bit-strings is optimal.

Problem-4 Generate all the strings of length n drawn from $0 \ldots k - 1$.

Solution: Let us assume we keep current k-ary string in an array $A[0..n-1]$. Call function *k-string*(n, k):

```
def rangeToList(k):
    result = []
    for i in range(0,k):
            result.append(str(i))
    return result
def baseKStrings(n,k):
    if n == 0: return []
    if n == 1: return rangeToList(k)
    return [ digit+bitstring for digit in baseKStrings(1,k)
                    for bitstring in baseKStrings(n-1,k)]
print baseKStrings(4,3)
```

Let $T(n)$ be the running time of $k - string(n)$. Then,

$$T(n) = \begin{cases} c, & if\ n < 0 \\ kT(n-1) + d, & otherwise \end{cases}$$

Using Subtraction and Conquer Master theorem we get: $T(n) = O(k^n)$.

Note: For more problems, refer to *String Algorithms* chapter.

Problem-5 Solve the recurrence $T(n) = 2T(n - 1) + 2^n$.

Solution: At each level of the recurrence tree, the number of problems is double from the previous level, while the amount of work being done in each problem is half from the previous level. Formally, the i^{th} level has 2^i problems, each requiring 2^{n-i} work. Thus the i^{th} level requires exactly 2^n work. The depth of this tree is n, because at the i^{th} level, the originating call will be $T(n - i)$. Thus the total complexity for $T(n)$ is $T(n2^n)$.

Problem-6 **Finding the length of connected cells of 1s (regions) in an matrix of 0s and 1s**: Given a matrix, each of which may be 1 or 0. The filled cells that are connected form a region. Two cells are said to be connected if they are adjacent to each other horizontally, vertically or diagonally. There may be several regions in the matrix. How do you find the largest region (in terms of number of cells) in the matrix?

Sample Input: 11000 Sample Output: 5
01100
00101
10001
01011

Solution: The simplest idea is: for each location traverse in all 8 directions and in each of those directions keep track of maximum region found.

```
def getval(A, i, j, L,  H):
    if (i< 0 or i >= L or j< 0 or j >= H):
            return 0
    else:
            return A[i][j]
def findMaxBlock(A, r, c, L, H, size):
    global maxsize
    global cntarr
    if ( r >= L or c >= H):
            return
    cntarr[r][c]=1
    size += 1
    if  (size > maxsize):
```

```
            maxsize = size
    #search in eight directions
    direction=[[-1,0],[-1,-1],[0,-1],[1,-1],[1,0],[1,1],[0,1],[-1,1]];
    for i in range(0,7):
            newi =r+direction[i][0]
            newj=c+direction[i][1]
            val=getval (A, newi, newj, L, H)
            if (val>0  and (cntarr[newi][newj]==0)):
                    findMaxBlock(A, newi, newj, L, H, size)
    cntarr[r][c]=0
def getMaxOnes(A, rmax, colmax):
    global maxsize
    global size
    global cntarr
    for i in range(0,rmax):
            for j in range(0,colmax):
                    if (A[i][j] == 1):
                            findMaxBlock(A, i, j, rmax, colmax, 0)
    return maxsize
zarr=[[1,1,0,0,0],[0,1,1,0,1],[0,0,0,1,1],[1,0,0,1,1],[0,1,0,1,1]]
rmax = 5
colmax =  5
maxsize=0
size=0
cntarr=rmax*[colmax*[0]]
print ("Number of maximum 1s are ")
print getMaxOnes(zarr, rmax, colmax)
```

CHAPTER

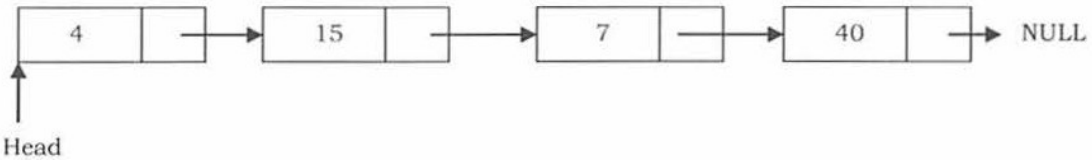
3

LINKED LISTS

3.1 What is a Linked List?

A linked list is a data structure used for storing collections of data. A linked list has the following properties.

- Successive elements are connected by pointers
- The last element points to NULL
- Can grow or shrink in size during execution of a program
- Can be made just as long as required (until systems memory exhausts)
- Does not waste memory space (but takes some extra memory for pointers)



3.2 Linked Lists ADT

The following operations make linked lists an ADT:

Main Linked Lists Operations

- Insert: inserts an element into the list
- Delete: removes and returns the specified position element from the list

Auxiliary Linked Lists Operations

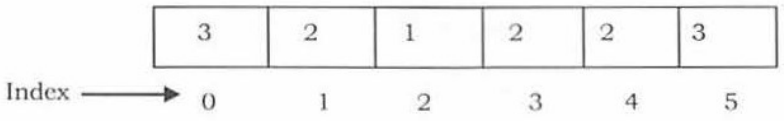
- Delete List: removes all elements of the list (dispose of the list)
- Count: returns the number of elements in the list
- Find n^{th} node from the end of the list

3.3 Why Linked Lists?

There are many other data structures that do the same thing as linked lists. Before discussing linked lists it is important to understand the difference between linked lists and arrays. Both linked lists and arrays are used to store collections of data, and since both are used for the same purpose, we need to differentiate their usage. That means in which cases *arrays* are suitable and in which cases *linked lists* are suitable.

3.4 Arrays Overview

One memory block is allocated for the entire array to hold the elements of the array. The array elements can be accessed in constant time by using the index of the particular element as the subscript.



Why Constant Time for Accessing Array Elements?

To access an array element, the address of an element is computed as an offset from the base address of the array and one multiplication is needed to compute what is supposed to be added to the base address to get the memory address of the element. First the size of an element of that data type is calculated and then it is multiplied with the index of the element to get the value to be added to the base address.

This process takes one multiplication and one addition. Since these two operations take constant time, we can say the array access can be performed in constant time.

Advantages of Arrays

- Simple and easy to use
- Faster access to the elements (constant access)

Disadvantages of Arrays

- **Fixed size:** The size of the array is static (specify the array size before using it).
- **One block allocation**: To allocate the array itself at the beginning, sometimes it may not be possible to get the memory for the complete array (if the array size is big).
- **Complex position-based insertion**: To insert an element at a given position, we may need to shift the existing elements. This will create a position for us to insert the new element at the desired position. If the position at which we want to add an element is at the beginning, then the shifting operation is more expensive.

Dynamic Arrays

Dynamic array (also called *growable array, resizable array, dynamic table,* or *array list*) is a random access, variable-size list data structure that allows elements to be added or removed.

One simple way of implementing dynamic arrays is to initially start with some fixed size array. As soon as that array becomes full, create the new array double the size of the original array. Similarly, reduce the array size to half if the elements in the array are less than half.

Note: We will see the implementation for *dynamic arrays* in the *Stacks*, *Queues* and *Hashing* chapters.

Advantages of Linked Lists

Linked lists have both advantages and disadvantages. The advantage of linked lists is that they can be *expanded* in constant time. To create an array, we must allocate memory for a certain number of elements. To add more elements to the array, we must create a new array and copy the old array into the new array. This can take a lot of time.

We can prevent this by allocating lots of space initially but then we might allocate more than we need and waste memory. With a linked list, we can start with space for just one allocated element and *add* on new elements easily without the need to do any copying and reallocating.

Issues with Linked Lists (Disadvantages)

There are a number of issues with linked lists. The main disadvantage of linked lists is *access time* to individual elements. Array is random-access, which means it takes O(1) to access any element in the array. Linked lists take $O(n)$ for access to an element in the list in the worst case. Another advantage of arrays in access time is *spacial locality* in memory. Arrays are defined as contiguous blocks of memory, and so any array element will be physically near its neighbors. This greatly benefits from modern CPU caching methods.

Although the dynamic allocation of storage is a great advantage, the *overhead* with storing and retrieving data can make a big difference. Sometimes linked lists are *hard to manipulate*. If the last item is deleted, the last but one must then have its pointer changed to hold a NULL reference. This requires that the list is traversed to find the last but one link, and its pointer set to a NULL reference.

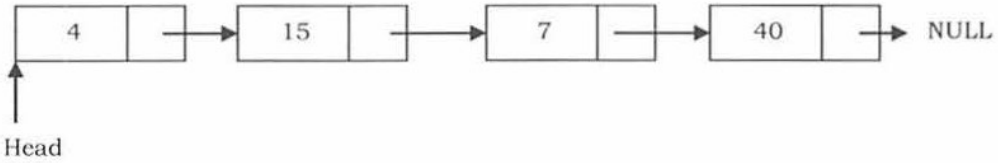
Finally, linked lists waste memory in terms of extra reference points.

3.5 Comparison of Linked Lists with Arrays and Dynamic Arrays

Parameter	Linked list	Array	Dynamic array
Indexing	O(*n*)	O(1)	O(1)
Insertion/deletion at beginning	O(1)	O(*n*), if array is not full (for shifting the elements)	O(*n*)
Insertion at ending	O(*n*)	O(1), if array is not full	O(1), if array is not full O(*n*), if array is full
Deletion at ending	O(*n*)	O(1)	O(*n*)
Insertion in middle	O(*n*)	O(*n*), if array is not full (for shifting the elements)	O(*n*)
Deletion in middle	O(*n*)	O(*n*), if array is not full (for shifting the elements)	O(*n*)
Wasted space	O(*n*)	0	O(*n*)

3.6 Singly Linked Lists

Generally "linked list" means a singly linked list. This list consists of a number of nodes in which each node has a *next* pointer to the following element. The link of the last node in the list is NULL, which indicates the end of the list.



Following is a type declaration for a linked list of integers:

```
#Node of a Singly Linked List
class Node:
    #constructor
    def __init__(self):
        self.data = None
        self.next = None
    #method for setting the data field of the node
    def setData(self,data):
        self.data = data
    #method for getting the data field of the node
    def getData(self):
        return self.data
    #method for setting the next field of the node
    def setNext(self,next):
        self.next = next
    #method for getting the next field of the node
    def getNext(self):
        return self.next
    #returns true if the node points to another node
    def hasNext(self):
            return self.next != None
```

Basic Operations on a List

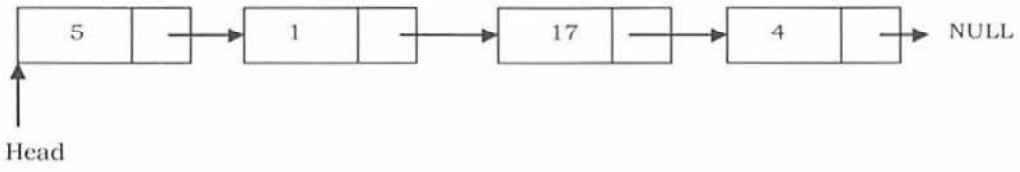
- Traversing the list
- Inserting an item in the list
- Deleting an item from the list

Traversing the Linked List

Let us assume that the *head* points to the first node of the list. To traverse the list we do the following.

- Follow the pointers.

- Display the contents of the nodes (or count) as they are traversed.
- Stop when the next pointer points to NULL.



The ListLength() function takes a linked list as input and counts the number of nodes in the list. The function given below can be used for printing the list data with extra print function.

```
def listLength(self):
    current = self.head
    count = 0

    while current != None:
        count = count + 1
        current = current.getNext()
    return count
```

Time Complexity: O(n), for scanning the list of size n.
Space Complexity: O(1), for creating a temporary variable.

Singly Linked List Insertion

Insertion into a singly-linked list has three cases:

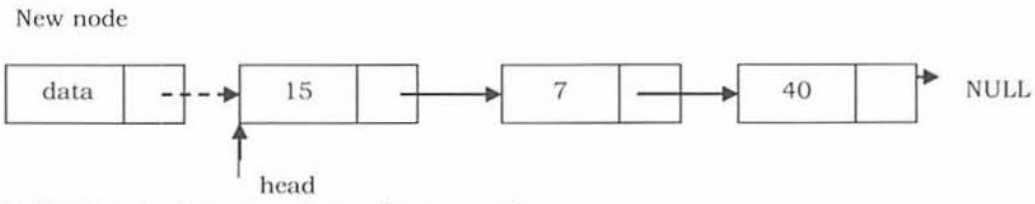
- Inserting a new node before the head (at the beginning)
- Inserting a new node after the tail (at the end of the list)
- Inserting a new node at the middle of the list (random location)

Note: To insert an element in the linked list at some position p, assume that after inserting the element the position of this new node is p.

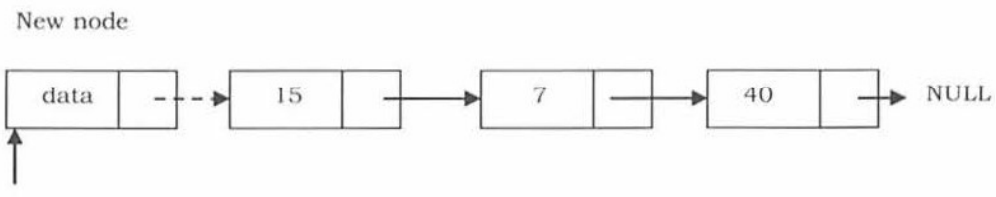
Inserting a Node in Singly Linked List at the Beginning

In this case, a new node is inserted before the current head node. *Only one next pointer* needs to be modified (new node's next pointer) and it can be done in two steps:

- Update the next pointer of new node, to point to the current head.



- Update head pointer to point to the new node.



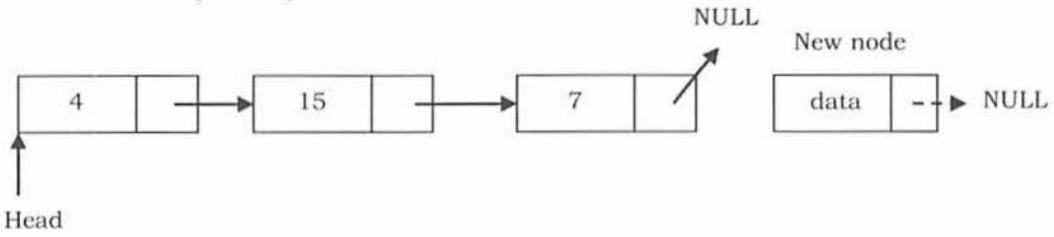

```
#method for inserting a new node at the beginning of the Linked List (at the head)
def insertAtBeginning(self,data):
    newNode = Node()
    newNode.setData(data)

    if self.length == 0:
```

```
        self.head = newNode
    else:
        newNode.setNext(self.head)
        self.head = newNode
    self.length += 1
```

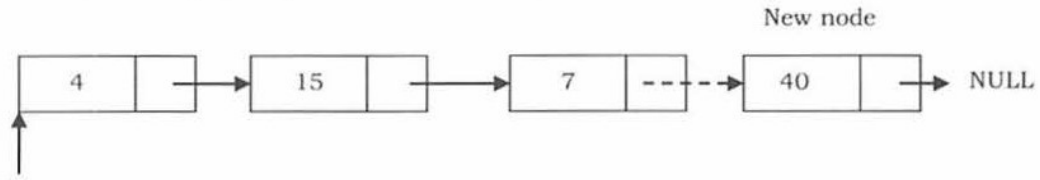
Inserting a Node in Singly Linked List at the Ending

In this case, we need to modify *two next pointers* (last nodes next pointer and new nodes next pointer).

- New nodes next pointer points to NULL.



- Last nodes next pointer points to the new node.

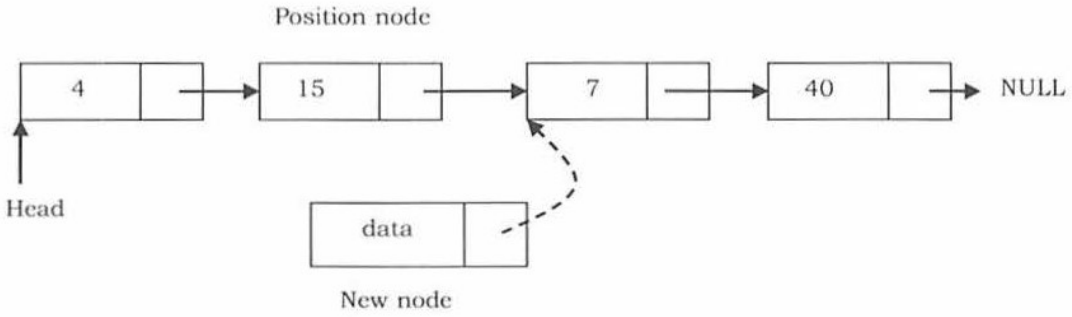



```
#method for inserting a new node at the end of a Linked List
def insertAtEnd(self,data):
    newNode = Node()
    newNode.setData(data)
    current = self.head
    while current.getNext() != None:
        current = current.getNext()
    current.setNext(newNode)
    self.length += 1
```

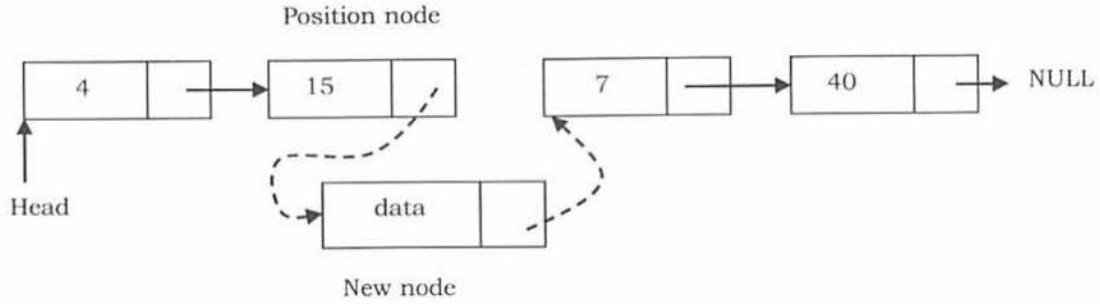
Inserting a Node in Singly Linked List at the Middle

Let us assume that we are given a position where we want to insert the new node. In this case also, we need to modify two next pointers.

- If we want to add an element at position 3 then we stop at position 2. That means we traverse 2 nodes and insert the new node. For simplicity let us assume that the second node is called *position* node. The new node points to the next node of the position where we want to add this node.



- Position nodes next pointer now points to the new node.



Let us write the code for all three cases. We must update the first element pointer in the calling function, not just in the called function. For this reason we need to send a double pointer. The following code inserts a node in the singly linked list.

```
#Method for inserting a new node at any position in a Linked List
def insertAtPos(self,pos,data):
 if pos > self.length or pos < 0:
    return None
 else:
    if pos == 0:
          self.insertAtBeg(data)
    else:
          if pos == self.length:
             self.insertAtEnd(data)
          else:
             newNode = Node()
             newNode.setData(data)
             count = 0
             current = self.head
             while count < pos-1:
                   count += 1
                   current = current.getNext()
             newNode.setNext(current.getNext())
             current.setNext(newNode)
             self.length += 1
```

Note: We can implement the three variations of the *insert* operation separately.

Time Complexity: O(n), since, in the worst case, we may need to insert the node at the end of the list.
Space Complexity: O(1), for creating one temporary variable.

Singly Linked List Deletion

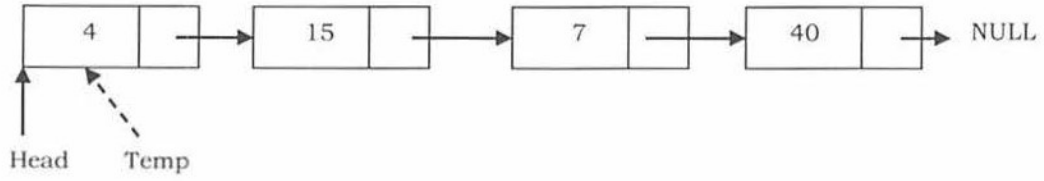
Similar to insertion, here we also have three cases.

- Deleting the first node
- Deleting the last node
- Deleting an intermediate node.

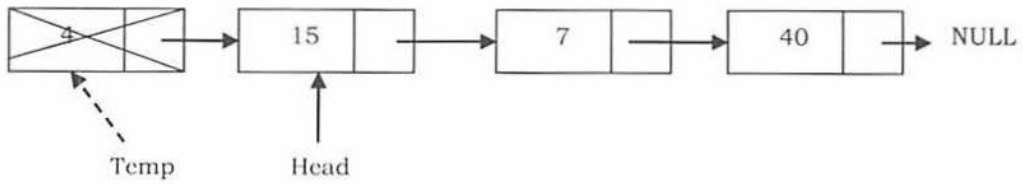
Deleting the First Node in Singly Linked List

First node (current head node) is removed from the list. It can be done in two steps:

- Create a temporary node which will point to the same node as that of head.



- Now, move the head nodes pointer to the next node and dispose of the temporary node.

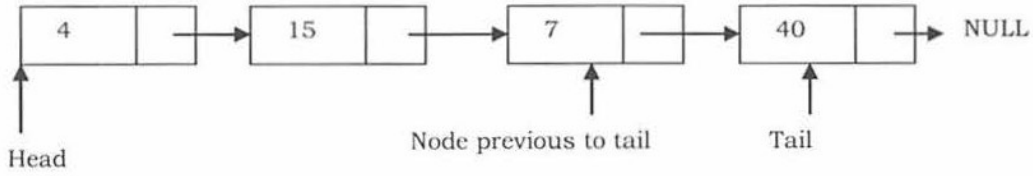



```
#method to delete the first node of the linked list
def deleteFromBeginning(self):
    if self.length == 0:
        print "The list is empty"
    else:
        self.head = self.head.getNext()
        self.length -= 1
```

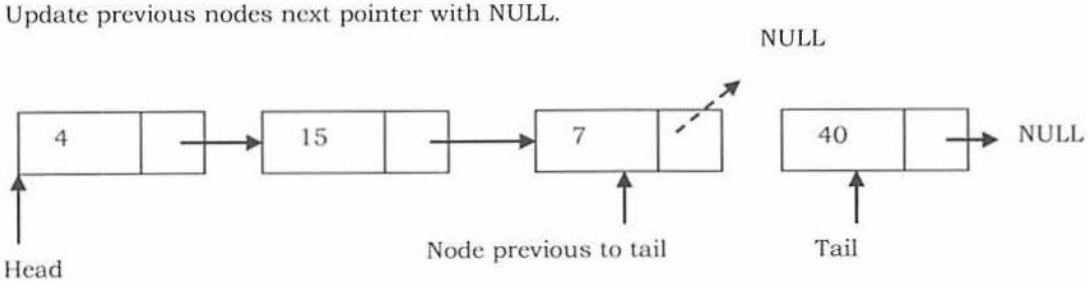
Deleting the Last Node in Singly Linked List

In this case, the last node is removed from the list. This operation is a bit trickier than removing the first node, because the algorithm should find a node, which is previous to the tail. It can be done in three steps:

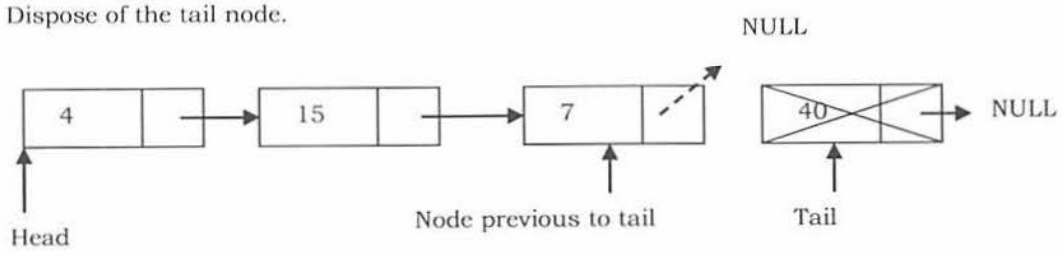
- Traverse the list and while traversing maintain the previous node address also. By the time we reach the end of the list, we will have two pointers, one pointing to the *tail* node and the other pointing to the node *before* the tail node.



- Update previous nodes next pointer with NULL.



- Dispose of the tail node.

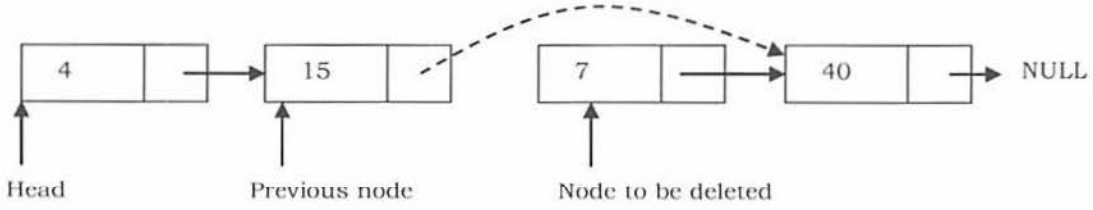



```
#Method to delete the last node of the linked list
def deleteLastNodeFromSinglyLinkedList(self):
    if self.length == 0:
        print "The list is empty"
    else:
        currentnode = self.head
        previousnode = self.head
        while currentnode.getNext() != None:
            previousnode = currentnode
            currentnode = currentnode.getNext()
        previousnode.seNext(None)
        self.length -= 1
```

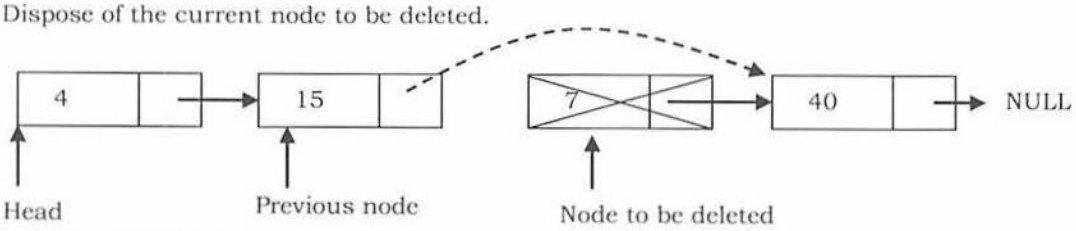
Deleting an Intermediate Node in Singly Linked List

In this case, the node to be removed is *always located between* two nodes. Head and tail links are not updated in this case. Such a removal can be done in two steps:

- Similar to the previous case, maintain the previous node while traversing the list. Once we find the node to be deleted, change the previous node's next pointer to the next pointer of the node to be deleted.



- Dispose of the current node to be deleted.




```
#Delete with node from linked list
def deleteFromLinkedListWithNode(self, node):
    if self.length == 0:
        raise ValueError("List is empty")
    else:
        current = self.head
        previous = None
        found = False
        while not found:
            if current == node:
                found = True
            elif current is None:
                raise ValueError("Node not in Linked List")
            else:
                previous = current
                current = current.getNext()
    if previous is None:
        self.head = current.getNext()
    else:
        previous.seNext(current.getNext())
    self.length -= 1
#Delete with data from linked list
def deleteValue(self,value):
    currentnode = self.head
    previousnode = self.head
    while currentnode.next != None or currentnode.value != value:
        if currentnode.value == value:
            previousnode.next = currentnode.next
            self.length -= 1
            return
        else:
            previousnode = currentnode
            currentnode = currentnode.next
    print "The value provided is not present"
#Method to delete a node at a particular position
def deleteAtPosition(self,pos):
    count = 0
```

```
        currentnode = self.head
        previousnode = self.head
        if pos > self.length or pos < 0:
            print "The position does not exist. Please enter a valid position"
        else:
            while currentnode.next != None or count < pos:
                count = count + 1
                if count == pos:
                    previousnode.next = currentnode.next
                    self.length -= 1
                    return
                else:
                    previousnode = currentnode
                    currentnode = currentnode.next
```

Time Complexity: O(n). In the worst case, we may need to delete the node at the end of the list.
Space Complexity: O(1), for one temporary variable.

Deleting Singly Linked List

Python is garbage-collected, so if you reduce the size of your list, it will reclaim memory.

```
def clear( self ) :
    self.head = None
```

Time Complexity: O(1). Space Complexity: O(1)

3.7 Doubly Linked Lists

The *advantage* of a doubly linked list (also called *two – way linked list*) is that given a node in the list, we can navigate in both directions. A node in a singly linked list cannot be removed unless we have the pointer to its predecessor. But in a doubly linked list, we can delete a node even if we don't have the previous node's address (since each node has a left pointer pointing to the previous node and can move backward).

The primary *disadvantages* of doubly linked lists are:

- Each node requires an extra pointer, requiring more space.
- The insertion or deletion of a node takes a bit longer (more pointer operations).

Similar to a singly linked list, let us implement the operations of a doubly linked list. If you understand the singly linked list operations, then doubly linked list operations are obvious. Following is a type declaration for a doubly linked list of integers:

```
class Node:
    # If data is not given by user,its taken as None
    def __init__(self, data=None, next=None, prev=None):
        self.data = data
        self.next = next
        self.prev = prev
    #method for setting the data field of the node
    def setData(self,data):
        self.data = data
    #method for getting the data field of the node
    def getData(self):
        return self.data
    #method for setting the next field of the node
    def setNext(self,next):
        self.next = next
    #method for getting the next field of the node
    def getNext(self):
        return self.next
    #returns true if the node points to another node
    def hasNext(self):
```

```
            return self.next != None
        #method for setting the next field of the node
        def setPrev(self,prev):
            self.prev = prev
        #method for getting the next field of the node
        def getPrev(self):
            return self.prev
        #returns true if the node points to another node
        def hasPrev(self):
            return self.prev != None
        # __str__ returns string equivalent of Object
        def __str__(self):
    return "Node[Data = %s]" % (self.data,)
```

Doubly Linked List Insertion

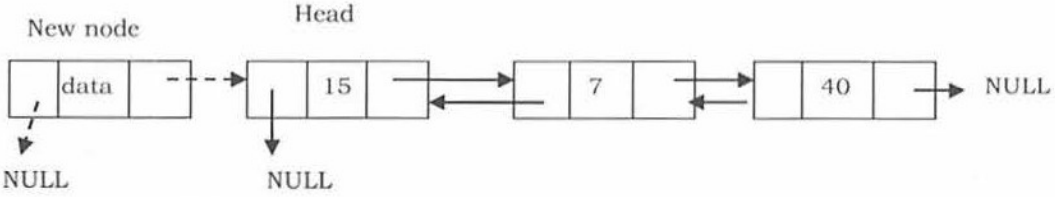
Insertion into a doubly-linked list has three cases (same as a singly linked list):

- Inserting a new node before the head.
- Inserting a new node after the tail (at the end of the list).
- Inserting a new node at the middle of the list.

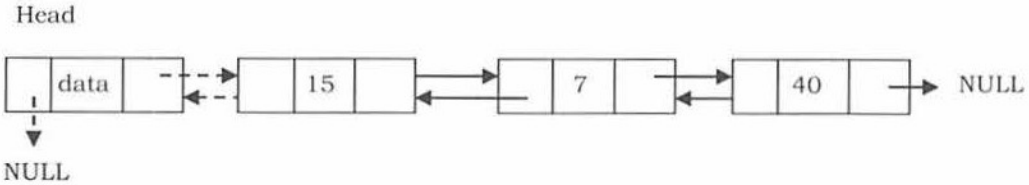
Inserting a Node in Doubly Linked List at the Beginning

In this case, new node is inserted before the head node. Previous and next pointers need to be modified and it can be done in two steps:

- Update the right pointer of the new node to point to the current head node (dotted link in below figure) and also make left pointer of new node as NULL.



- Update head node's left pointer to point to the new node and make new node as head.

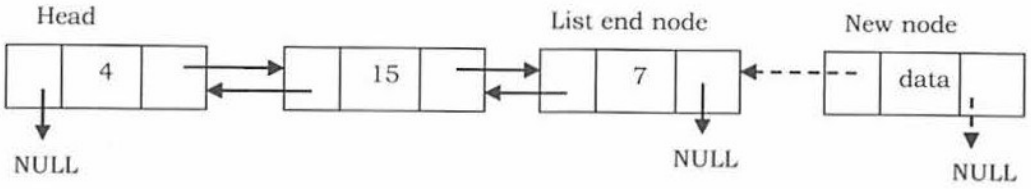



```
def insertAtBeginning(self, data):
    newNode = Node(data, None, None)
    if (self.head == None):    # To imply that if head == None
        self.head = self.tail = newNode
    else:
        newNode.setPrev(None)
        newNode.setNext(self.head)
        self.head.setPrev(newNode)
        self.head = newNode
```

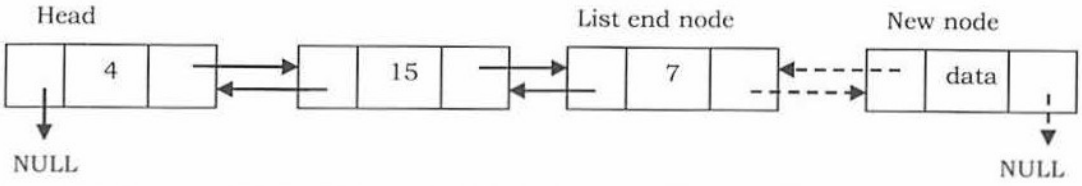
Inserting a Node in Doubly Linked List at the Ending

In this case, traverse the list till the end and insert the new node.

- New node right pointer points to NULL and left pointer points to the end of the list.



- Update right pointer of last node to point to new node.

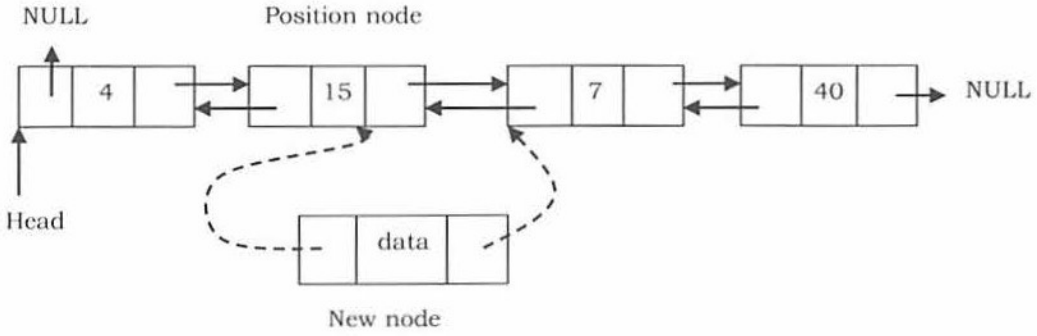



```
def insertAtEnd(self, data):
    if (self.head == None):    # To imply that if head == None
        self.head = Node(data)
        self.tail = self.head
    else:
        current = self.head
        while(current.getNext() != None):
            current = current.getNext()
        current.setNext(Node(data, None, current))
        self.tail = current.getNext()
```

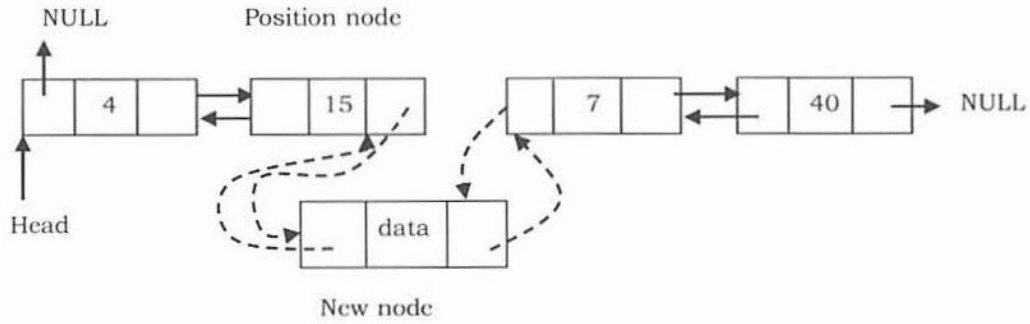
Inserting a Node in Doubly Linked List at the Middle

As discussed in singly linked lists, traverse the list to the position node and insert the new node.

- *New node* right pointer points to the next node of the *position node* where we want to insert the new node. Also, *new node* left pointer points to the *position node*.



- Position node right pointer points to the new node and the *next node* of position node left pointer points to new node.



Now, let us write the code for all of these three cases. We must update the first element pointer in the calling function, not just in the called function. For this reason we need to send a double pointer. The following code inserts a node in the doubly linked list.

```
def getNode(self, index):
    currentNode = self.head
    if currentNode == None:
        return None
    i = 0
    while i < index and currentNode.getNext() is not None::
        currentNode = currentNode.getNext()
        if currentNode == None:
            break
        i += 1
    return currentNode
def insertAtGivenPosition(self, index, data):
    newNode = Node(data)
    if self.head == None or index == 0:
        self.insertAtBeginning(data)
    elif index > 0:
        temp = self.getNode(index)
        if temp == None or temp.getNext() == None:
            self.insert(data)
        else:
            newNode.setNext(temp.getNext())
            newNode.setPrev(temp)
            temp.getNext().setPrev(newNode)
            temp.setNext(newNode)
```

Time Complexity: O(n). In the worst case, we may need to insert the node at the end of the list.
Space Complexity: O(1), for a temporary variable.

Doubly Linked List Deletion

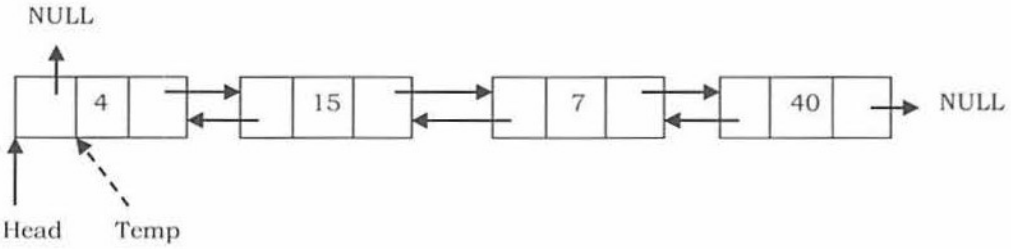
Similar to singly linked list deletion, here we have three cases:

- Deleting the first node
- Deleting the last node
- Deleting an intermediate node

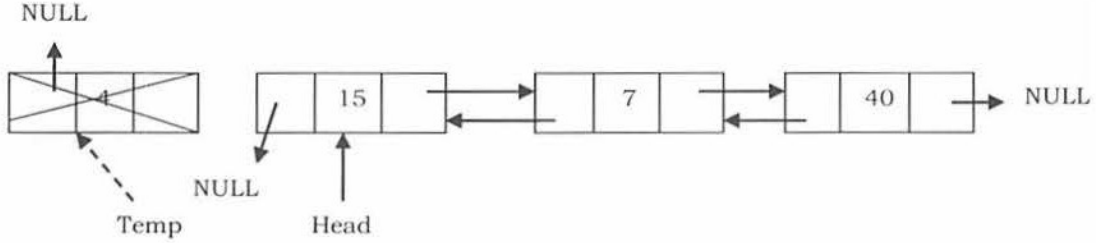
Deleting the First Node in Doubly Linked List

In this case, the first node (current head node) is removed from the list. It can be done in two steps:

- Create a temporary node which will point to the same node as that of head.



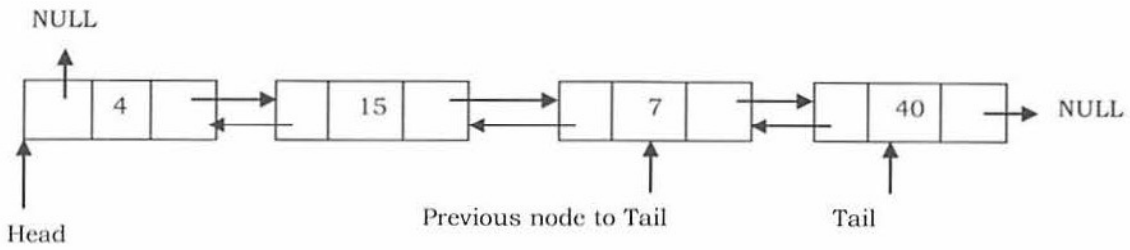
- Now, move the head nodes pointer to the next node and change the heads left pointer to NULL. Then, dispose of the temporary node.



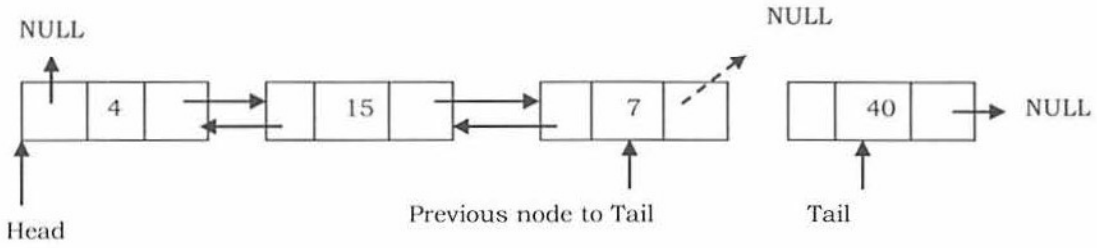
Deleting the Last Node in Doubly Linked List

This operation is a bit trickier, than removing the first node, because the algorithm should find a node, which is previous to the tail first. This can be done in three steps:

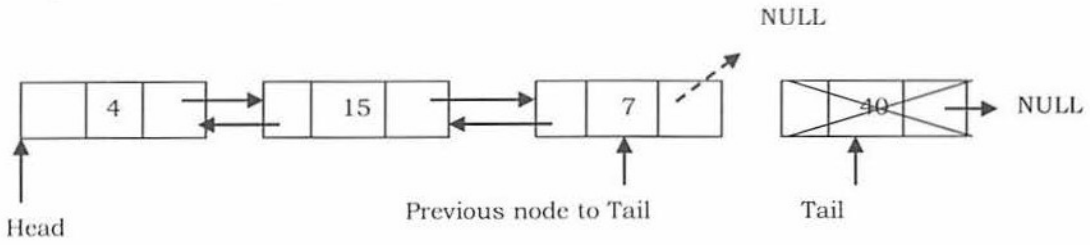
- Traverse the list and while traversing maintain the previous node address also. By the time we reach the end of the list, we will have two pointers, one pointing to the tail and the other pointing to the node before the tail.



- Update the next pointer of previous node to the tail node with NULL.



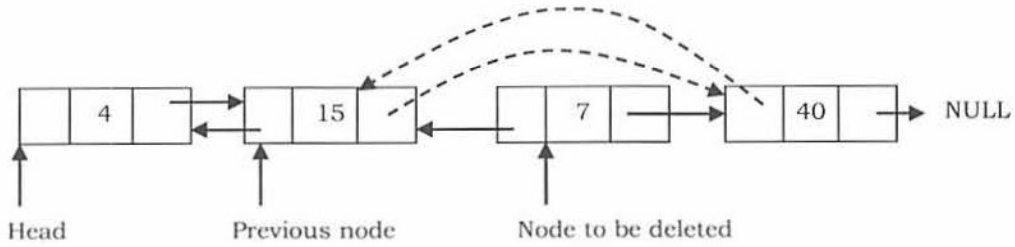
- Dispose of the tail node.



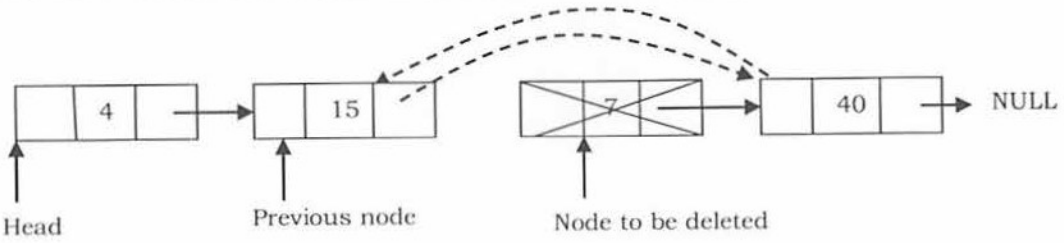
Deleting an Intermediate Node in Doubly Linked List

In this case, the node to be removed is *always located between* two nodes, and the head and tail links are not updated. The removal can be done in two steps:

- Similar to the previous case, maintain the previous node while also traversing the list. Upon locating the node to be deleted, change the previous node's next pointer to the next node of the node to be deleted.



- Dispose of the current node to be deleted.




```
#Deleting element at given position
def getNode(self, index):
    currentNode = self.head
    if currentNode == None:
        return None
    i = 0
    while i <= index:
        currentNode = currentNode.getNext()
        if currentNode == None:
            break
        i += 1
    return currentNode

def delteAtGivenPosition(self, index):
    temp = self.getNode(index)
    if temp:
        temp.getPrev.setNext(temp.getNext())
        if temp.getNext():
            temp.getNext().setPrev(temp.getPrev())
        temp.setPrev(None)
        temp.setNext(None)
        temp.setData(None)

#Deleting with given data
def deleteWithData(self, data):
    temp = self.head
    while temp is not None:
        if temp.getData() == data:
            # if it's not the first element
            if temp.getNext() is not None:
                temp.getNext().setNext(temp.getNext())
                temp.getNext().setPrev(temp.getPrev())
            else:
                # otherwise we have no prev (it's None), head is the next one, and prev becomes None
                self.head = temp.getNext()
                temp.getNext().setPrev(None)
        temp = temp.getNext()
```

Time Complexity: O(n), for scanning the complete list of size n.
Space Complexity: O(1), for creating one temporary variable.

3.8 Circular Linked Lists

In singly linked lists and doubly linked lists, the end of lists are indicated with NULL value. But circular linked lists do not have ends. While traversing the circular linked lists we should be careful; otherwise we will be traversing the list infinitely. In circular linked lists, each node has a successor. Note that unlike singly linked lists, there is no node with NULL pointer in a circularly linked list. In some situations, circular linked lists are useful. There is no difference in the node declaration of circular linked lists compared to singly linked lists.

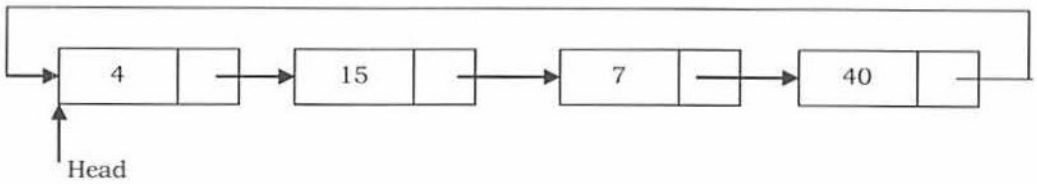
For example, when several processes are using the same computer resource (CPU) for the same amount of time, we have to assure that no process accesses the resource before all other processes do (round robin algorithm). The following is a type declaration for a circular linked list:

```
#Node of a Circular Linked List
class Node:
    #constructor
    def __init__(self):
```

```
        self.data = None
        self.next = None
    #method for setting the data field of the node
    def setData(self,data):
        self.data = data
    #method for getting the data field of the node
    def getData(self):
        return self.data
    #method for setting the next field of the node
    def setNext(self,next):
        self.next = next
    #method for getting the next field of the node
    def getNext(self):
        return self.next
    #returns true if the node points to another node
    def hasNext(self):
            return self.next != None
```

In a circular linked list, we access the elements using the *head* node (similar to *head* node in singly linked list and doubly linked lists).

Counting Nodes in a Circular List

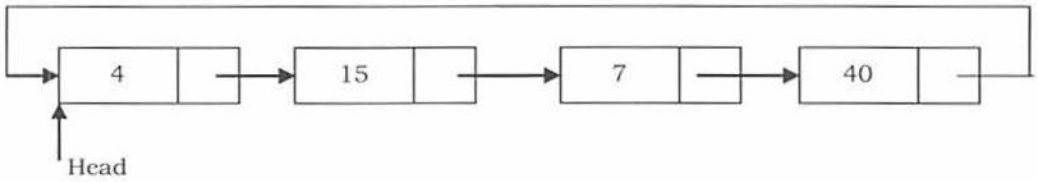


The circular list is accessible through the node marked *head*. To count the nodes, the list has to be traversed from the node marked *head*, with the help of a dummy node *current*, and stop the counting when *current* reaches the starting node *head*. If the list is empty, *head* will be NULL, and in that case set *count* = 0. Otherwise, set the current pointer to the first node, and keep on counting till the current pointer reaches the starting node.

```
#This method would be a member of other class (say, CircularList)
def circularListLength(self):
    currentNode = self.head
    if currentNode == None:
            return 0
    count = 1
    currentNode = currentNode.getNext()
    while currentNode !=  self.head:
            currentNode = currentNode.getNext()
            count =  count + 1
    retur  count
```

Time Complexity: O(n), for scanning the complete list of size n. Space Complexity: O(1), for temporary variable.

Printing the Contents of a Circular List



We assume here that the list is being accessed by its *head* node. Since all the nodes are arranged in a circular fashion, the *tail* node of the list will be the node previous to the *head* node. Let us assume we want to print the

contents of the nodes starting with the *head* node. Print its contents, move to the next node and continue printing till we reach the *head* node again.

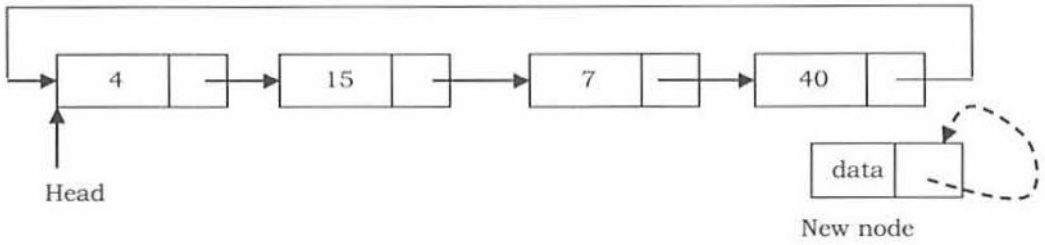
```
def printCircularList(self):
    currentNode = self.head
    if currentNode == None:   return 0
    print (currentNode.getData())
    currentNode = currentNode.getNext()
    while currentNode !=  self.head:
            currentNode = currentNode.getNext()
            print (currentNode.getData())
```

Time Complexity: O(n), for scanning the complete list of size n. Space Complexity: O(1), for temporary variable.

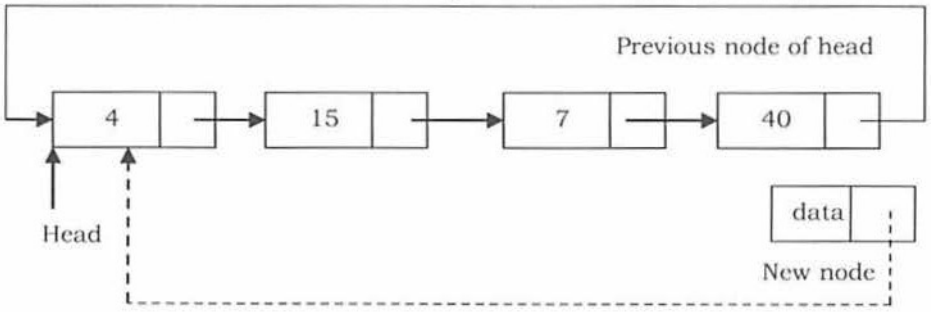
Inserting a Node at the End of a Circular Linked List

Let us add a node containing *data*, at the end of a list (circular list) headed by *head*. The new node will be placed just after the tail node (which is the last node of the list), which means it will have to be inserted in between the tail node and the first node.

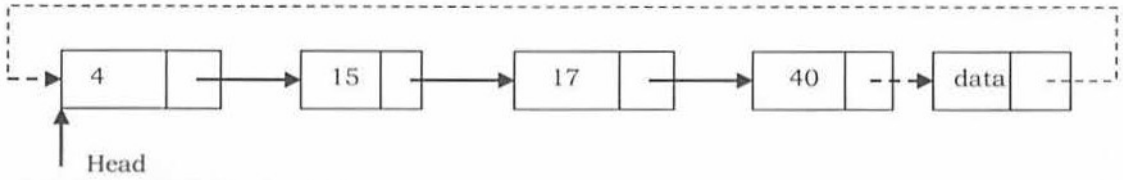
- Create a new node and initially keep its next pointer pointing to itself.



- Update the next pointer of the new node with the head node and also traverse the list to the tail. That means in a circular list we should stop at the node whose next node is head.



- Update the next pointer of the previous node to point to the new node and we get the list as shown below.




```
def insertAtEndInCLL (self, data):
    current = self.head
    newNode = Node()
    newNode.setData(data)
    while current.getNext != self.head:
            current = current.getNext()
    newNode.setNext(newNode)
    if self.head == None:
            self.head = newNode;
    else:
```

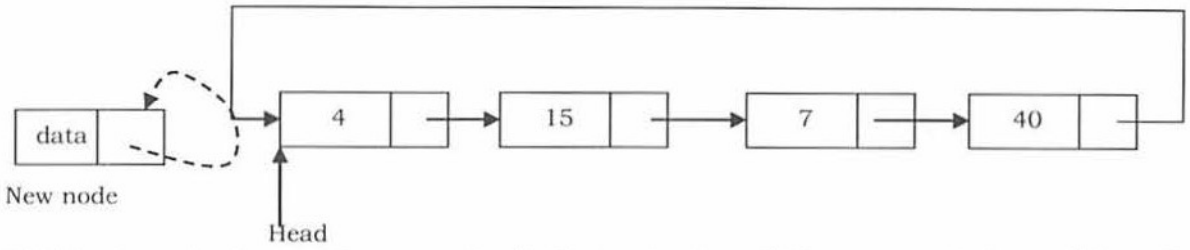
```
            newNode.setNext(self.head)
            current.setNext(newNode)
```

Time Complexity: O(n), for scanning the complete list of size n. Space Complexity: O(1), for temporary variable.

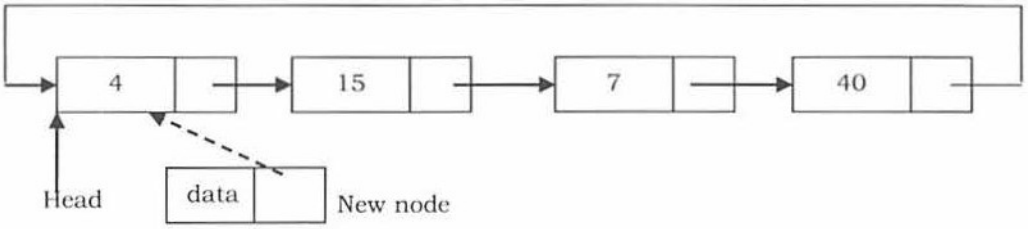
Inserting a Node at the Front of a Circular Linked List

The only difference between inserting a node at the beginning and at the end is that, after inserting the new node, we just need to update the pointer. The steps for doing this are given below:

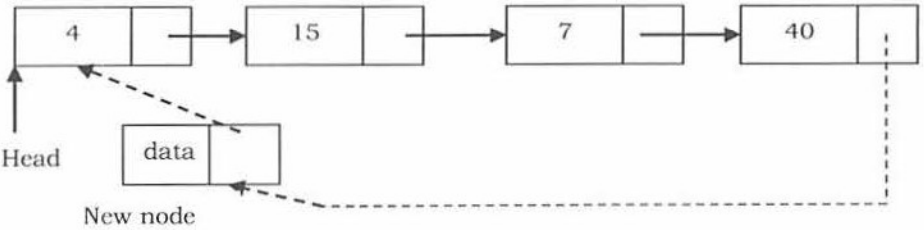
- Create a new node and initially keep its next pointer pointing to itself.



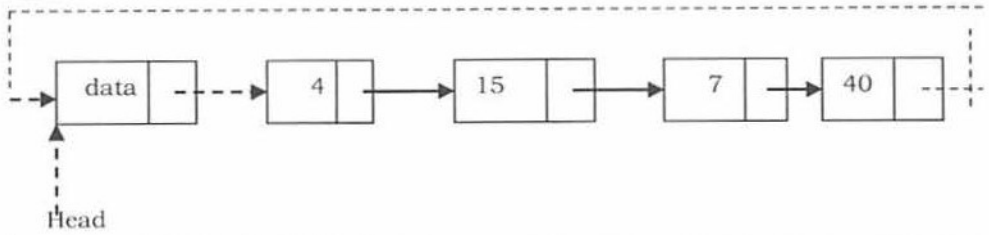
- Update the next pointer of the new node with the head node and also traverse the list until the tail. That means in a circular list we should stop at the node which is its previous node in the list.



- Update the previous head node in the list to point to the new node.



- Make the new node as the head.




```
def insertAtBeginInCLL (self, data):
    current = self.head
    newNode = Node()
    newNode.setData(data)
    while current.getNext != self.head:
            current = current.getNext()
    newNode.setNext(newNode)
    if self.head == None:
            self.head = newNode;
    else:
            newNode.setNext(self.head)
            current.setNext(newNode)
```

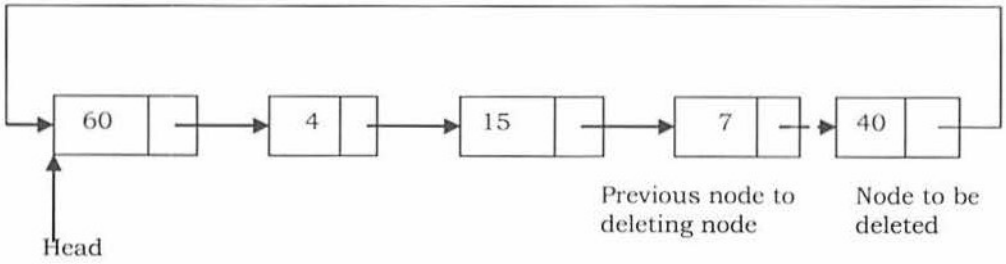
```
            self.head = newNode
```

Time Complexity: O(n), for scanning the complete list of size n. Space Complexity: O(1), for temporary variable.

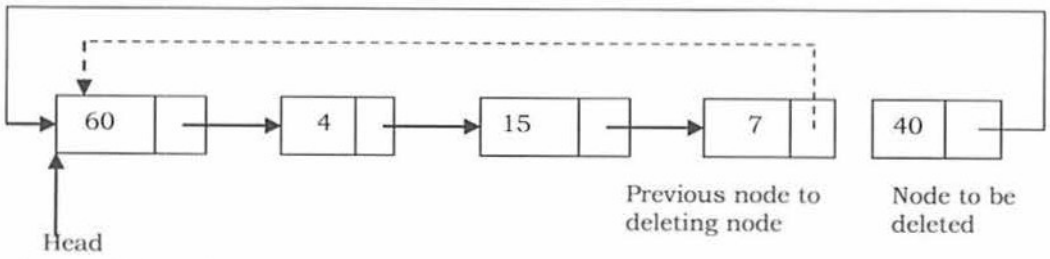
Deleting the Last Node in a Circular List

The list has to be traversed to reach the last but one node. This has to be named as the tail node, and its next field has to point to the first node. Consider the following list. To delete the last node 40, the list has to be traversed till you reach 7. The next field of 7 has to be changed to point to 60, and this node must be renamed *pTail*.

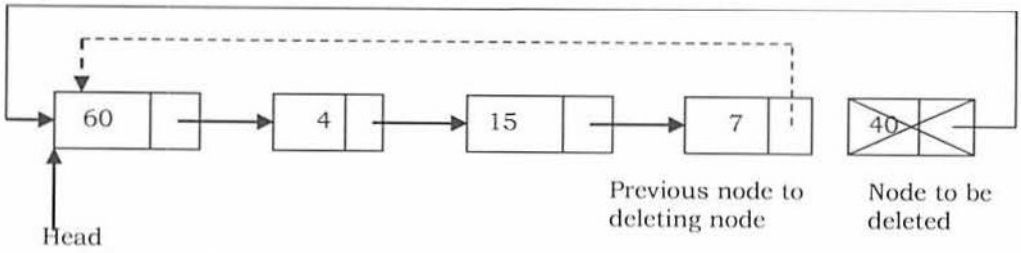
- Traverse the list and find the tail node and its previous node.



- Update the tail node's previous node pointer to point to head.



- Dispose of the tail node.



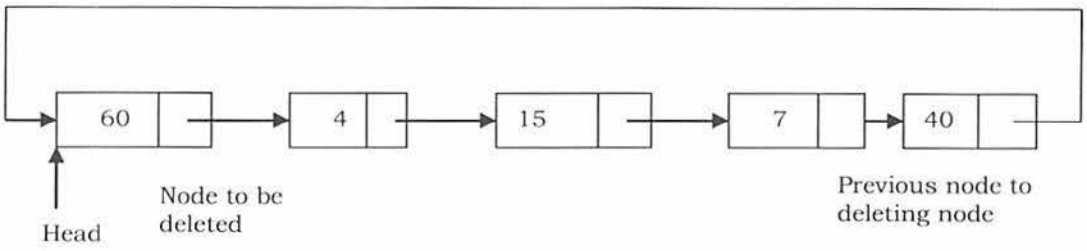

```
def deleteLastNodeFromCLL (self):
    temp = self.head
    current = self.head
    if self.head == None:
        print ("List Empty")
        return
    while current.getNext() != self.head:
        temp = current;
        current = current.getNext()
    temp.setNext(current.getNext())
    return
```

Time Complexity: O(n), for scanning the complete list of size n. Space Complexity: O(1)

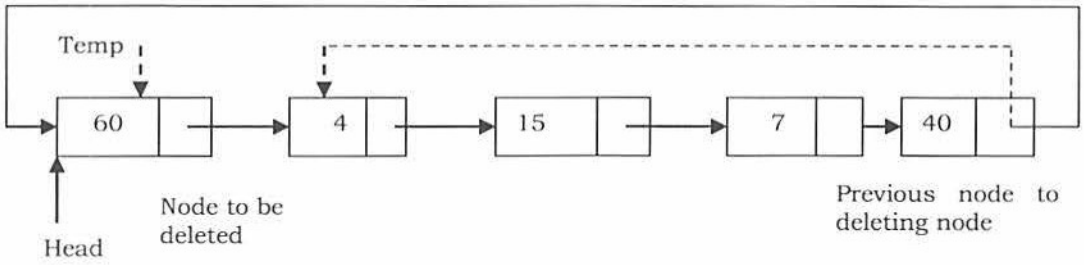
Deleting the First Node in a Circular List

The first node can be deleted by simply replacing the next field of the tail node with the next field of the first node.

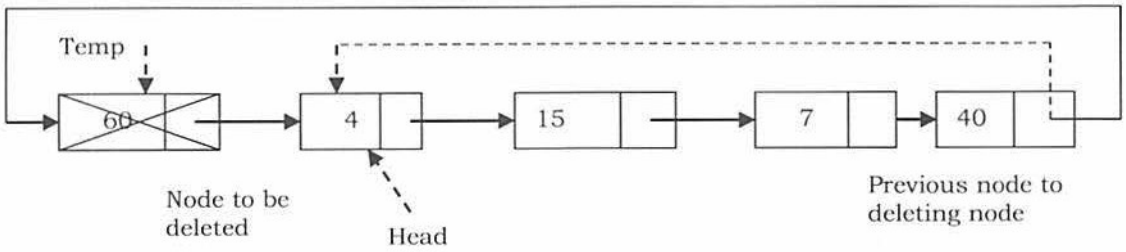
- Find the tail node of the linked list by traversing the list. Tail node is the previous node to the head node which we want to delete.



- Create a temporary node which will point to the head. Also, update the tail nodes next pointer to point to next node of head (as shown below).



- Now, move the head pointer to next node. Create a temporary node which will point to head. Also, update the tail nodes next pointer to point to next node of head (as shown below).




```
def deleteFrontNodeFromCLL (self):
 current = self.head
 if self.head == None:
    print ("List Empty")
    return
 while current.getNext() != self.head:
    current = current.getNext()
 current.setNext(self.head.getNext())
 self.head = self.head.getNext()
 return
```

Time Complexity: O(n), for scanning the complete list of size n. Space Complexity: O(1)

Applications of Circular List

Circular linked lists are used in managing the computing resources of a computer. We can use circular lists for implementing stacks and queues.

3.9 A Memory-efficient Doubly Linked List

In conventional implementation, we need to keep a forward pointer to the next item on the list and a backward pointer to the previous item. That means elements in doubly linked list implementations consist of data, a pointer to the next node and a pointer to the previous node in the list as shown below.

Conventional Node Definition

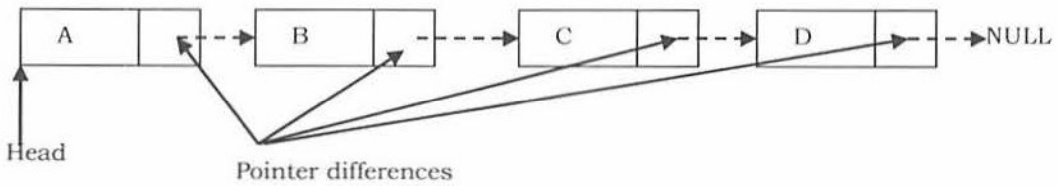
```
#Node of a Singly Linked List
class Node:
   #constructor
   def __init__(self):
      self.data = None
```

```
        self.next = None
    #method for setting the data field of the node
    def setData(self,data):
        self.data = data
    #method for getting the data field of the node
    def getData(self):
        return self.data
    #method for setting the next field of the node
    def setNext(self,next):
        self.next = next
    #method for getting the next field of the node
    def getNext(self):
        return self.next
    #returns true if the node points to another node
    def hasNext(self):
            return self.next != None
```

Recently a journal (Sinha) presented an alternative implementation of the doubly linked list ADT, with insertion, traversal and deletion operations. This implementation is based on pointer difference. Each node uses only one pointer field to traverse the list back and forth.

New Node Definition

```
class Node:
    #constructor
    def __init__(self):
        self.data = None
        self.ptrdiff = None
    #method for setting the data field of the node
    def setData(self,data):
        self.data = data
    #method for getting the data field of the node
    def getData(self):
        return self.data
    #method for setting the pointer difference field of the node
    def setPtrDiff(self, prev, next):
        self.ptrdiff = prev ^ next
    #method for getting the next field of the node
    def getPtrDiff(self):
        return self.ptrdiff
```



The *ptrdiff* pointer field contains the difference between the pointer to the next node and the pointer to the previous node. The pointer difference is calculated by using exclusive-or (⊕) operation.

$$ptrdiff = pointer\ to\ previous\ node \oplus pointer\ to\ next\ node.$$

The *ptrdiff* of the start node (head node) is the ⊕ of NULL and *next* node (next node to head). Similarly, the *ptrdiff* of end node is the ⊕ of *previous* node (previous to end node) and NULL. As an example, consider the following linked list.

In the example above,

- The next pointer of A is: NULL ⊕ B
- The next pointer of B is: A ⊕ C
- The next pointer of C is: B ⊕ D
- The next pointer of D is: C ⊕ NULL

Why does it work?

To find the answer to this question let us consider the properties of ⊕:

$X \oplus X = 0$
$X \oplus 0 = X$
$X \oplus Y = Y \oplus X$ (symmetric)
$(X \oplus Y) \oplus Z = X \oplus (Y \oplus Z)$ (transitive)

For the example above, let us assume that we are at C node and want to move to B. We know that C's *ptrdiff* is defined as B ⊕ D. If we want to move to B, performing ⊕ on C's *ptrdiff* with D would give B. This is due to the fact that

$$(B \oplus D) \oplus D = B \text{ (since, } D \oplus D=0)$$

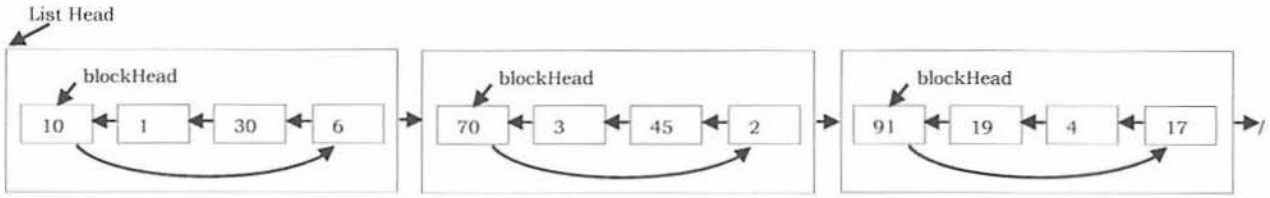
Similarly, if we want to move to D, then we have to apply ⊕ to C's *ptrdiff* with B to give D.

$$(B \oplus D) \oplus B = D \text{ (since, } B \oplus B=0)$$

From the above discussion we can see that just by using a single pointer, we can move back and forth. A memory-efficient implementation of a doubly linked list is possible with minimal compromising of timing efficiency.

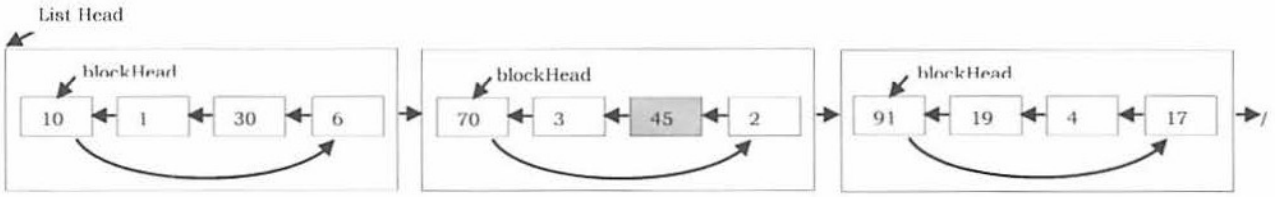
3.10 Unrolled Linked Lists

One of the biggest advantages of linked lists over arrays is that inserting an element at any location takes only O(1) time. However, it takes O(*n*) to search for an element in a linked list. There is a simple variation of the singly linked list called *unrolled linked lists*. An unrolled linked list stores multiple elements in each node (let us call it a block for our convenience). In each block, a circular linked list is used to connect all nodes.



Assume that there will be no more than n elements in the unrolled linked list at any time. To simplify this problem, all blocks, except the last one, should contain exactly $\lfloor\sqrt{n}\rfloor$ elements. Thus, there will be no more than $\lfloor\sqrt{n}\rfloor$ blocks at any time.

Searching for an element in Unrolled Linked Lists

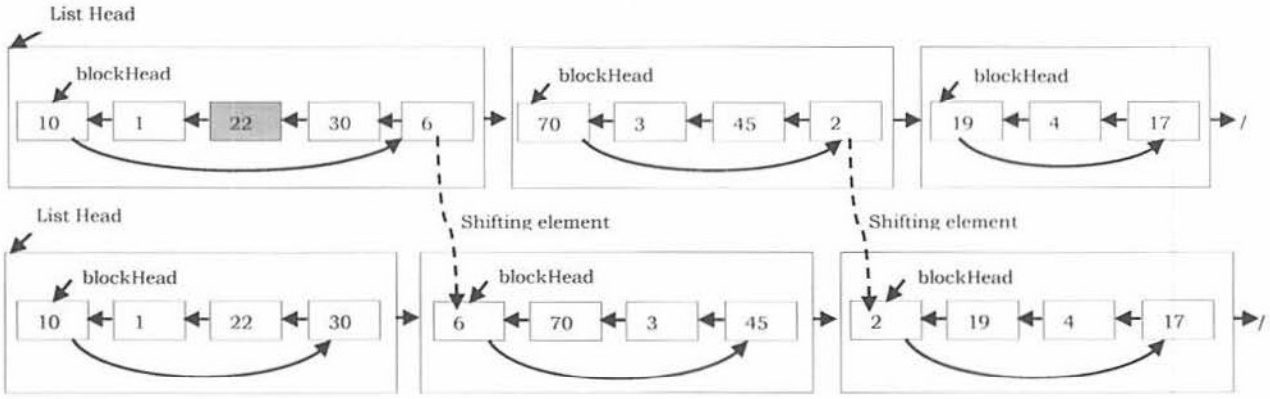


In unrolled linked lists, we can find the k^{th} element in $O(\sqrt{n})$:

1. Traverse the *list of blocks* to the one that contains the k^{th} node, i.e., the $\left\lfloor \frac{k}{\lfloor\sqrt{n}\rfloor} \right\rfloor^{\text{th}}$ block. It takes $O(\sqrt{n})$ since we may find it by going through no more than $\sqrt{n}$ blocks.
2. Find the $(k \bmod \lfloor\sqrt{n}\rfloor)^{\text{th}}$ node in the circular linked list of this block. It also takes $O(\sqrt{n})$ since there are no more than $\lfloor\sqrt{n}\rfloor$ nodes in a single block.

Inserting an element in Unrolled Linked Lists

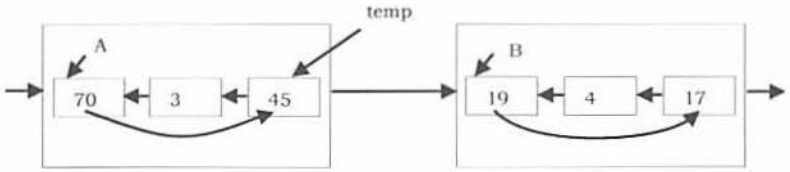
When inserting a node, we have to re-arrange the nodes in the unrolled linked list to maintain the properties previously mentioned, that each block contains $\lfloor\sqrt{n}\rfloor$ nodes. Suppose that we insert a node x after the i^{th} node, and x should be placed in the j^{th} block. Nodes in the j^{th} block and in the blocks after the j^{th} block have to be shifted toward the tail of the list so that each of them still have $\lfloor\sqrt{n}\rfloor$ nodes. In addition, a new block needs to be added to the tail if the last block of the list is out of space, i.e., it has more than $\lfloor\sqrt{n}\rfloor$ nodes.



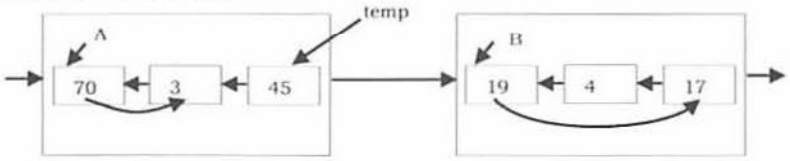
Performing Shift Operation

Note that each *shift* operation, which includes removing a node from the tail of the circular linked list in a block and inserting a node to the head of the circular linked list in the block after, takes only O(1). The total time complexity of an insertion operation for unrolled linked lists is therefore $O(\sqrt{n})$; there are at most $O(\sqrt{n})$ blocks and therefore at most $O(\sqrt{n})$ shift operations.

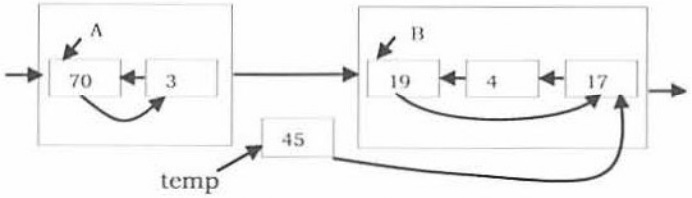
1. A temporary pointer is needed to store the tail of *A*.



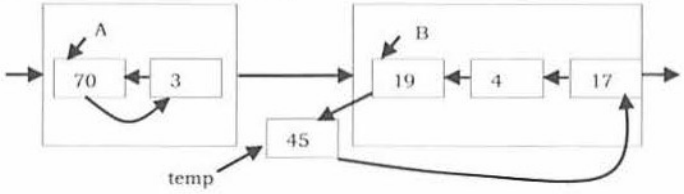
2. In block *A*, move the next pointer of the head node to point to the second-to-last node, so that the tail node of *A* can be removed.



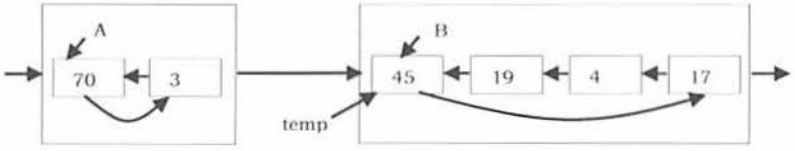
3. Let the next pointer of the node, which will be shifted (the tail node of *A*), point to the tail node of *B*.



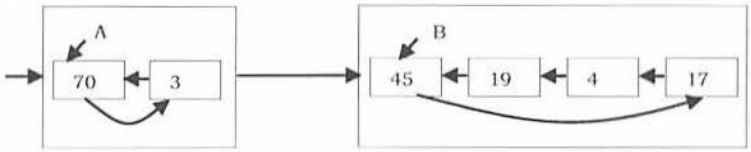
4. Let the next pointer of the head node of *B* point to the node temp points to.



5. Finally, set the head pointer of *B* to point to the node *temp* points to. Now the node temp points to becomes the new head node of *B*.



6. *temp* pointer can be thrown away. We have completed the shift operation to move the original tail node of *A* to become the new head node of *B*.



Performance

With unrolled linked lists, there are a couple of advantages, one in speed and one in space. First, if the number of elements in each block is appropriately sized (e.g., at most the size of one cache line), we get noticeably better cache performance from the improved memory locality. Second, since we have $O(n/m)$ links, where n is the number of elements in the unrolled linked list and m is the number of elements we can store in any block, we can also save an appreciable amount of space, which is particularly noticeable if each element is small.

Comparing Doubly Linked Lists and Unrolled Linked Lists

To compare the overhead for an unrolled list, elements in doubly linked list implementations consist of data, a pointer to the next node, and a pointer to the previous node in the list, as shown below.

```
class Node:
  # If data is not given by user,its taken as None
  def __init__(self, data=None, next=None, prev=None):
    self.data = data
    self.next = next
    self.prev = prev
```

Assuming we have 4 byte pointers, each node is going to take 8 bytes. But the allocation overhead for the node could be anywhere between 8 and 16 bytes. Let's go with the best case and assume it will be 8 bytes. So, if we want to store 1K items in this list, we are going to have 16KB of overhead.

Now, let's think about an unrolled linked list node (let us call it *LinkedBlock*). It will look something like this:

```
class LinkedBlock:
  def __init__(self, nextBlock=None, blockHead=None):
    self.next = nextBlock
    self.head = blockHead
    self.nodeCount = 0
```

Therefore, allocating a single node (12 bytes + 8 bytes of overhead) with an array of 100 elements (400 bytes + 8 bytes of overhead) will now cost 428 bytes, or 4.28 bytes per element. Thinking about our 1K items from above, it would take about 4.2KB of overhead, which is close to 4x better than our original list. Even if the list becomes severely fragmented and the item arrays are only 1/2 full on average, this is still an improvement. Also, note that we can tune the array size to whatever gets us the best overhead for our application.

Implementation

```
#Node of a Singly Linked List
class Node:
 #constructor
 def __init__(self):
        self.value = None
        self.next = None

#Node of a Singly Linked List
class LinkedBlock:
 #constructor
 def __init__(self):
        self.head = None
        self.next = None
        nodeCount = 0

blockSize = 2
blockHead = None

#create an empty block
```

```
def newLinkedBlock():
 block=LinkedBlock()
 block.next=None
 block.head=None
 block.nodeCount=0
 return block

#create a node
def newNode(value):
 temp=Node()
 temp.next=None
 temp.value=value
 return temp

def searchElements(blockHead, k):
 #find the block
 j=(k+blockSize-1)//blockSize #k-th node is in the j-th block
 p=blockHead
 j -= 1
 while(j):
        p=p.next
        j -= 1

 fLinkedBlock=p

 #find the node
 q=p.head
 k=k%blockSize
 if(k==0):
        k=blockSize
 k = p.nodeCount+1-k
 k -= 1
 while (k):
        q=q.next
        k -= 1

 fNode=q

 return fLinkedBlock, fNode

#start shift operation from block *p
def shift(A):
 B = A
 global blockHead
 while(A.nodeCount > blockSize): #if this block still have to shift
        if(A.next==None): #reach the end. A little different
                A.next=newLinkedBlock()
                B=A.next
                temp=A.head.next
                A.head.next=A.head.next.next
                B.head=temp
                temp.next=temp
                A.nodeCount -= 1
                B.nodeCount += 1
        else:
                B=A.next
                temp=A.head.next
                A.head.next=A.head.next.next
                temp.next=B.head.next
                B.head.next=temp
                B.head=temp
                A.nodeCount -= 1
                B.nodeCount += 1
 A=B

def addElement(k, x):
 global blockHead
 r = newLinkedBlock()
```

```
    p = Node()
    if(blockHead == None): #initial, first node and block
            blockHead=newLinkedBlock()
            blockHead.head=newNode(x)
            blockHead.head.next=blockHead.head
            blockHead.nodeCount += 1
    else:
            if(k==0): #special case for k=0.
                    p=blockHead.head
                    q=p.next
                    p.next=newNode(x)
                    p.next.next=q
                    blockHead.head=p.next
                    blockHead.nodeCount += 1
                    shift(blockHead)
            else:
                    r, p = searchElements(blockHead, k)
                    q = p
                    while(q.next != p):
                            q=q.next
                    q.next=newNode(x)
                    q.next.next=p
                    r.nodeCount += 1
                    shift(r)
    return blockHead
def searchElement(blockHead, k):
    q, p = searchElements(blockHead, k)
    return p.value
blockHead = addElement(0,11)
blockHead = addElement(0,21)
blockHead = addElement(1,19)
blockHead = addElement(1,23)
blockHead = addElement(2,16)
blockHead = addElement(2,35)
searchElement(blockHead, 1)
```

3.11 Skip Lists

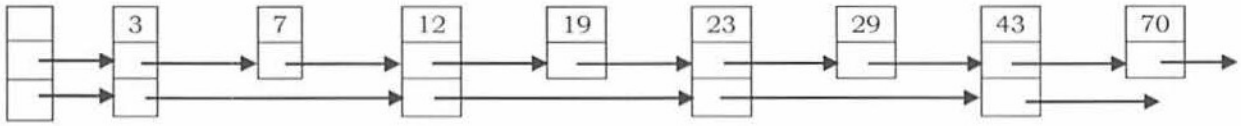
Binary trees can be used for representing abstract data types such as dictionaries and ordered lists. They work well when the elements are inserted in a random order. Some sequences of operations, such as inserting the elements in order, produce degenerate data structures that give very poor performance. If it were possible to randomly permute the list of items to be inserted, trees would work well with high probability for any input sequence. In most cases queries must be answered on-line, so randomly permuting the input is impractical. Balanced tree algorithms re-arrange the tree as operations are performed to maintain certain balance conditions and assure good performance.

Skip list is a data structure that can be used as an alternative to balanced binary trees (refer to *Trees* chapter). As compared to a binary tree, skip lists allow quick search, insertion and deletion of elements. This is achieved by using probabilistic balancing rather than strictly enforce balancing. It is basically a linked list with additional pointers such that intermediate nodes can be skipped. It uses a random number generator to make some decisions.

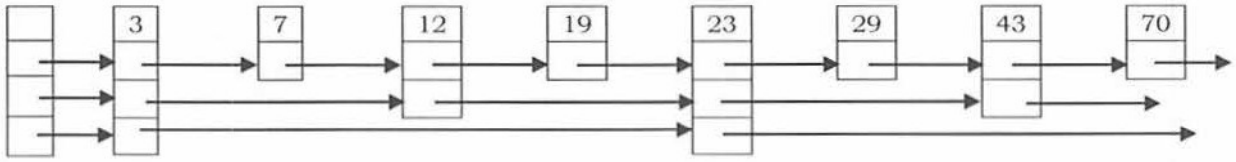
In an ordinary sorted linked list, search, insert, and delete are in $O(n)$ because the list must be scanned node-by-node from the head to find the relevant node. If somehow we could scan down the list in bigger steps (skip down, as it were), we would reduce the cost of scanning. This is the fundamental idea behind Skip Lists.

Skip Lists with One Level

Skip Lists with Two Levels



Skip Lists with Three Levels



This section gives algorithms to search for, insert and delete elements in a dictionary or symbol table. The Search operation returns the contents of the value associated with the desired key or failure if the key is not present. The Insert operation associates a specified key with a new value (inserting the key if it had not already been present). The Delete operation deletes the specified key. It is easy to support additional operations such as "find the minimum key" or "find the next key".

Each element is represented by a node, the level of which is chosen randomly when the node is inserted without regard for the number of elements in the data structure. A level i node has i forward pointers, indexed 1 through i. We do not need to store the level of a node in the node. Levels are capped at some appropriate constant *MaxLevel*. The level of a list is the maximum level currently in the list (or 1 if the list is empty). The header of a list has forward pointers at levels one through MaxLevel. The forward pointers of the header at levels higher than the current maximum level of the list point to NULL.

Initialization

An element NIL is allocated and given a key greater than any legal key. All levels of all skip lists are terminated with NIL. A new list is initialized so that the level of the list is equal to 1 and all forward pointers of the list's header point to NIL.

Search for an element

We search for an element by traversing forward pointers that do not overshoot the node containing the element being searched for. When no more progress can be made at the current level of forward pointers, the search moves down to the next level. When we can make no more progress at level 1, we must be immediately in front of the node that contains the desired element (if it is in the list).

Insertion and Deletion Algorithms

To insert or delete a node, we simply search and splice. A vector update is maintained so that when the search is complete (and we are ready to perform the splice), update[i] contains a pointer to the rightmost node of level i or higher that is to the left of the location of the insertion/deletion. If an insertion generates a node with a level greater than the previous maximum level of the list, we update the maximum level of the list and initialize the appropriate portions of the update vector. After each deletion, we check if we have deleted the maximum element of the list and if so, decrease the maximum level of the list.

Choosing a Random Level

Initially, we discussed a probability distribution where half of the nodes that have level i pointers also have level i+1 pointers. To get away from magic constants, we say that a fraction p of the nodes with level i pointers also have level i+1 pointers. (for our original discussion, p = 1/2). Levels are generated randomly by an algorithm. Levels are generated without reference to the number of elements in the list

Performance

In a simple linked list that consists of n elements, to perform a search n comparisons are required in the worst case. If a second pointer pointing two nodes ahead is added to every node, the number of comparisons goes down to $n/2 + 1$ in the worst case. Adding one more pointer to every fourth node and making them point to the fourth node ahead reduces the number of comparisons to $\lceil n/2 \rceil + 2$. If this strategy is continued so that every

node with i pointers points to $2 * i - 1$ nodes ahead, O($logn$) performance is obtained and the number of pointers has only doubled ($n + n/2 + n/4 + n/8 + n/16 + ... = 2n$).

The find, insert, and remove operations on ordinary binary search trees are efficient, O($logn$), when the input data is random; but less efficient, O(n), when the input data is ordered. Skip List performance for these same operations and for any data set is about as good as that of randomly-built binary search trees - namely O($logn$).

Comparing Skip Lists and Unrolled Linked Lists

In simple terms, Skip Lists are sorted linked lists with two differences:

- The nodes in an ordinary list have one next reference. The nodes in a Skip List have many *next* references (also called *forward* references).
- The number of *forward* references for a given node is determined probabilistically.

We speak of a Skip List node having levels, one level per forward reference. The number of levels in a node is called the *size* of the node. In an ordinary sorted list, insert, remove, and find operations require sequential traversal of the list. This results in O(n) performance per operation. Skip Lists allow intermediate nodes in the list to be skipped during a traversal - resulting in an expected performance of O($logn$) per operation.

Implementation

```
import random
import math
class Node(object):
    def __init__(self, data, level=0):
        self.data = data
        self.next = [None] * level
    def __str__(self):
        return "Node(%s,%s)" % (self.data, len(self.next))
    __repr__ = __str__
class SkipList(object):
    def __init__(self, max_level=8):
        self.max_level = max_level
        n = Node(None, max_level)
        self.head = n
        self.verbose = False
    def randomLevel(self, max_level):
        num = random.randint(1, 2**max_level - 1)
        lognum = math.log(num, 2)
        level = int(math.floor(lognum))
        return max_level - level
    def updateList(self, data):
        update = [None] * (self.max_level)
        n = self.head
        self._n_traverse = 0
        level = self.max_level - 1
        while level >= 0:
            if self.verbose and \
                n.next[level] != None and n.next[level].data >= data:
                print 'DROP down from level', level + 1
            while n.next[level] != None and n.next[level].data < data:
                self._n_traverse += 1
                if self.verbose:
                    print 'AT level', level, 'data', n.next[level].data
                n = n.next[level]
            update[level] = n
            level -= 1
        return update
    def find(self, data, update=None):
        if update is None:
            update = self.updateList(data)
        if len(update) > 0:
```

```
            candidate = update[0].next[0]
            if candidate != None and candidate.data == data:
                return candidate
        return None

    def insertNode(self, data, level=None):
        if level is None:
            level = self.randomLevel(self.max_level)
        node = Node(data, level)
        update = self.updateList(data)
        if self.find(data, update) == None:
            for i in range(level):
                node.next[i] = update[i].next[i]
                update[i].next[i] = node
def printLevel(sl, level):
    print 'level %d:' % level,
    node = sl.head.next[level]
    while node:
        print node.data, '=>',
        node = node.next[level]
    print 'END'
x = SkipList(4)
for i in range(0, 20, 2):
    x.insertNode(i)
printLevel(x, 0)
printLevel(x, 1)
printLevel(x, 2)
```

3.12 Linked Lists: Problems & Solutions

Problem-1 Implement Stack using Linked List.

Solution: Refer to *Stacks* chapter.

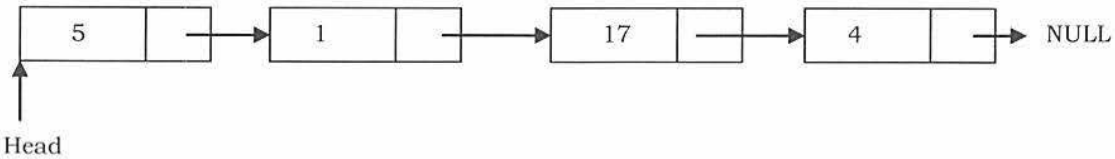
Problem-2 Find n^{th} node from the end of a Linked List.

Solution: Brute-Force Method: Start with the first node and count the number of nodes present after that node. If the number of nodes is $< n-1$ then return saying "fewer number of nodes in the list". If the number of nodes is $> n-1$ then go to next node. Continue this until the numbers of nodes after current node are $n-1$.

Time Complexity: $O(n^2)$, for scanning the remaining list (from current node) for each node.
Space Complexity: $O(1)$.

Problem-3 Can we improve the complexity of Problem-2?

Solution: Yes, using hash table. As an example consider the following list.



In this approach, create a hash table whose entries are < *position of node, node address* >. That means, key is the position of the node in the list and value is the address of that node.

Position in List	Address of Node
1	Address of 5 node
2	Address of 1 node
3	Address of 17 node
4	Address of 4 node

By the time we traverse the complete list (for creating the hash table), we can find the list length. Let us say the list length is M. To find n^{th} from the end of linked list, we can convert this to $M-n+1^{th}$ from the beginning. Since we already know the length of the list, it is just a matter of returning $M-n+1^{th}$ key value from the hash table.

Time Complexity: Time for creating the hash table, $T(m) = O(m)$.
Space Complexity: Since we need to create a hash table of size m, $O(m)$.

Problem-4 Can we use Problem-3 approach for solving Problem-2 without creating the hash table?

Solution: Yes. If we observe the Problem-3 solution, what we are actually doing is finding the size of the linked list. That means we are using the hash table to find the size of the linked list. We can find the length of the linked list just by starting at the head node and traversing the list. So, we can find the length of the list without creating the hash table. After finding the length, compute $M - n + 1$ and with one more scan we can get the $M - n + 1^{th}$ node from the beginning. This solution needs two scans: one for finding the length of the list and the other for finding $M - n + 1^{th}$ node from the beginning.

Time Complexity: Time for finding the length + Time for finding the $M\text{-}n+1^{th}$ node from the beginning. Therefore, $T(n = O(n) + O(n) \approx O(n)$.

Space Complexity: O(1). Hence, no need to create the hash table.

Problem-5 Can we solve Problem-2 in one scan?

Solution: Yes. Efficient Approach: Use two pointers *pNthNode* and *pTemp*. Initially, both point to head node of the list. *pNthNode* starts moving only after *pTemp* has made n moves. From there both move forward until *pTemp* reaches the end of the list. As a result *pNthNode* points to n^{th} node from the end of the linked list.

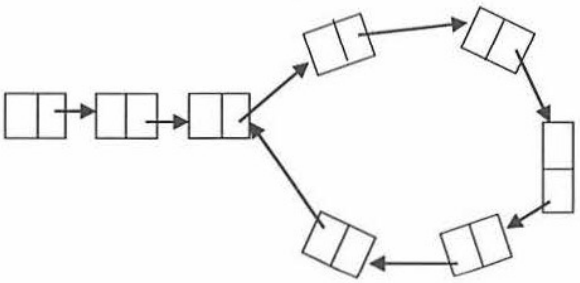
Note: At any point of time both move one node at a time.

```
def nthNodeFromEnd( self, n ):
    if 0 > n:
        return None
    # count k units from the self.head.
    temp = self.head
    count = 0
    while count < n and None != temp:
      temp = temp.next
      count += 1
    # if the LinkedList does not contain k elements, return None
    if count < n or None == temp:
      return None
    # keeping tab on the nth element from temp, slide temp until
    # temp equals self.tail. Then return the nth element.
    nth = self.head
    while None != temp.next:
      temp = temp.next
      nth = nth.next
    return nth
```

Time Complexity: $O(n)$. Space Complexity: O(1).

Problem-6 Check whether the given linked list is either NULL-terminated or ends in a cycle (cyclic).

Solution: Brute-Force Approach. As an example, consider the following linked list which has a loop in it. The difference between this list and the regular list is that, in this list, there are two nodes whose next pointers are the same. In regular singly linked lists (without a loop) each node's next pointer is unique. That means the repetition of next pointers indicates the existence of a loop.



One simple and brute force way of solving this is, start with the first node and see whether there is any node whose next pointer is the current node's address. If there is a node with the same address then that indicates that some other node is pointing to the current node and we can say a loop exists.

Continue this process for all the nodes of the linked list.

Does this method work? As per the algorithm, we are checking for the next pointer addresses, but how do we find the end of the linked list (otherwise we will end up in an infinite loop)?

Note: If we start with a node in a loop, this method may work depending on the size of the loop.

Problem-7 Can we use the hashing technique for solving Problem-6?

Solution: Yes. Using Hash Tables we can solve this problem.

Algorithm:

- Traverse the linked list nodes one by one.
- Check if the address of the node is available in the hash table or not.
- If it is already available in the hash table, that indicates that we are visiting the node that was already visited. This is possible only if the given linked list has a loop in it.
- If the address of the node is not available in the hash table, insert that node's address into the hash table.
- Continue this process until we reach the end of the linked list *or* we find the loop.

Time Complexity: $O(n)$ for scanning the linked list. Note that we are doing a scan of only the input.
Space Complexity: $O(n)$ for hash table.

Problem-8 Can we solve Problem-6 using the sorting technique?

Algorithm:

- Traverse the linked list nodes one by one and take all the next pointer values into an array.
- Sort the array that has the next node pointers.
- If there is a loop in the linked list, definitely two next node pointers will be pointing to the same node.
- After sorting if there is a loop in the list, the nodes whose next pointers are the same will end up adjacent in the sorted list.
- If any such pair exists in the sorted list then we say the linked list has a loop in it.

Time Complexity: $O(nlogn)$ for sorting the next pointers array.
Space Complexity: $O(n)$ for the next pointers array.

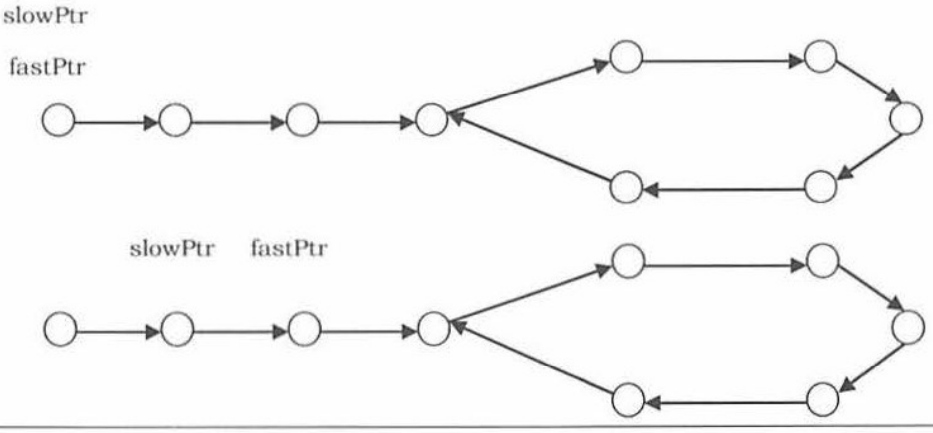
Problem with the above algorithm: The above algorithm works only if we can find the length of the list. But if the list has a loop then we may end up in an infinite loop. Due to this reason the algorithm fails.

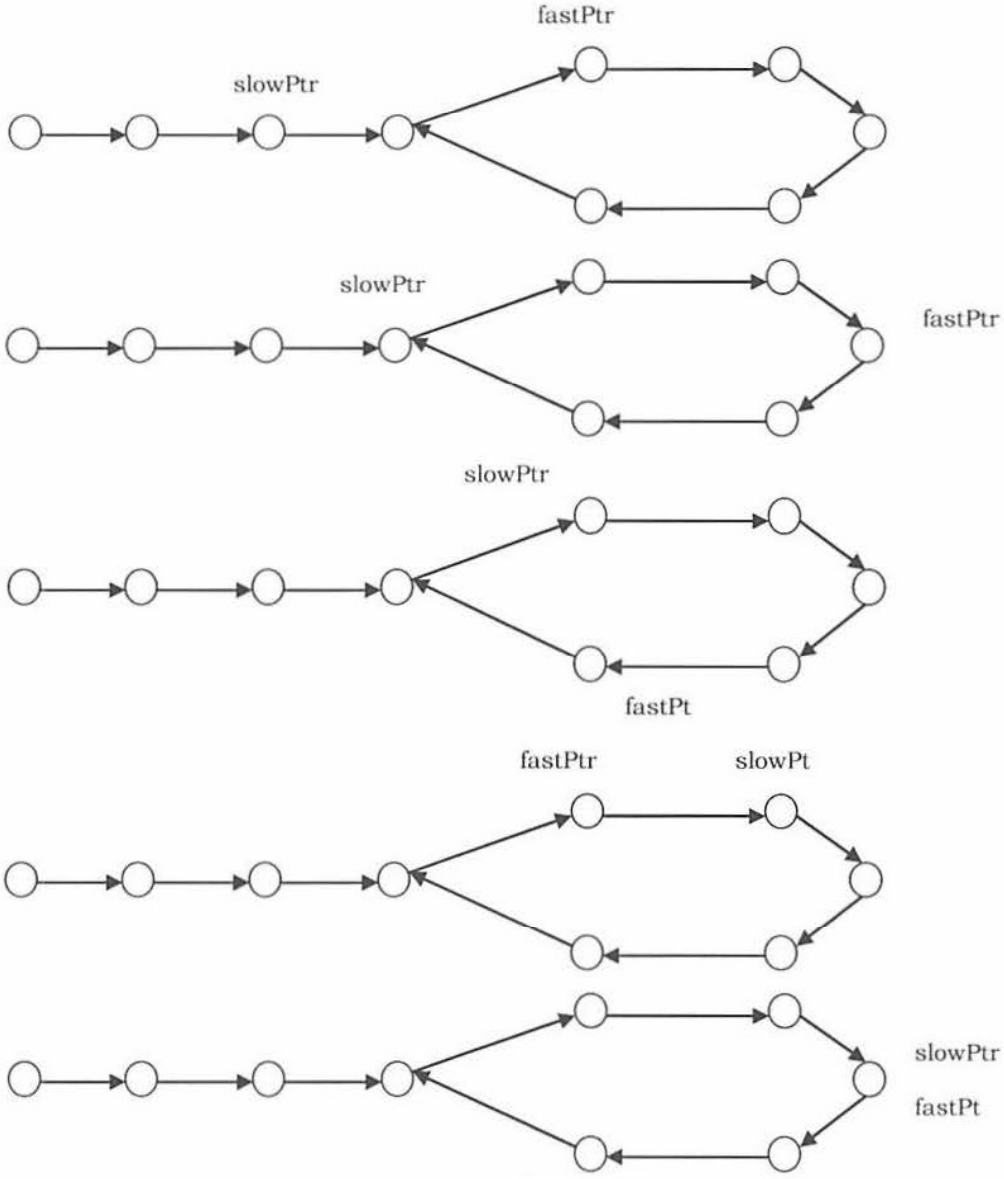
Problem-9 Can we solve the Problem-6 in $O(n)$?

Solution: Yes. Efficient Approach (Memoryless Approach): This problem was solved by *Floyd*. The solution is named the Floyd cycle finding algorithm. It uses *two* pointers moving at different speeds to walk the linked list. Once they enter the loop they are expected to meet, which denotes that there is a loop. This works because the only way a faster moving pointer would point to the same location as a slower moving pointer is if somehow the entire list or a part of it is circular. Think of a tortoise and a hare running on a track. The faster running hare will catch up with the tortoise if they are running in a loop.

As an example, consider the following example and trace out the Floyd algorithm. From the diagrams below we can see that after the final step they are meeting at some point in the loop which may not be the starting point of the loop.

Note: *slowPtr* (*tortoise*) moves one pointer at a time and *fastPtr* (*hare*) moves two pointers at a time.






```
def detectCycle(self):
    fastPtr = self.head
    slowPtr = self.head

    while (fastPtr and slowPtr):
        fastPtr = fastPtr.getNext()
        if (fastPtr == slowPtr):
            return True

        if fastPtr == None:
            return False

        fastPtr = fastPtr.getNext()
        if (fastPtr == slowPtr):
            return True

        slowPtr = slowPtr.getNext()
```

Time Complexity: O(n). Space Complexity: O(1).

Problem-10 We are given a pointer to the first element of a linked list L. There are two possibilities for L, it either ends (snake) or its last element points back to one of the earlier elements in the list (snail). Give an algorithm that tests whether a given list L is a snake or a snail.

Solution: It is the same as Problem-6.

Problem-11 Check whether the given linked list is NULL-terminated or not. If there is a cycle find the start node of the loop.

Solution: The solution is an extension to the solution in Problem-9. After finding the loop in the linked list, we initialize the *slowPtr* to the head of the linked list. From that point onwards both *slowPtr* and *fastPtr* move only one node at a time. The point at which they meet is the start of the loop. Generally we use this method for removing the loops. Let x and y be travelers such that y is walking twice as fast as x (i.e. $y = 2x$). Further, let s be the place where x and y first started walking at the same time. Then when x and y meet again, the distance from s to the start of the loop is the exact same distance from the present meeting place of x and y to the start of the loop.

```
def detectCycleStart( self ) :
  if None == self.head or None == self.head.next:
    return None
  # slow and fast both started at head after one step,
  # slow is at self.head.next and fast is at self.head.next.next
  slow = self.head.next
  fast = slow.next
  # each keep walking until they meet again.
  while slow != fast:
    slow = slow.next
    try:
      fast = fast.next.next
    except AttributeError:
      return None # no cycle if NoneType reached
  # from self.head to beginning of loop is same as from fast to beginning of loop
  slow = self.head
  while slow != fast:
    slow = slow.next
    fast = fast.next
  return slow # beginning of loop
```

Time Complexity: O(n). Space Complexity: O(1).

Problem-12 From the previous discussion and problems we understand that the meeting of tortoise and hare concludes the existence of the loop, but how does moving the tortoise to the beginning of the linked list while keeping the hare at the meeting place, followed by moving both one step at a time, make them meet at the starting point of the cycle?

Solution: This problem is at the heart of number theory. In the Floyd cycle finding algorithm, notice that the tortoise and the hare will meet when they are $n \times L$, where L is the loop length. Furthermore, the tortoise is at the midpoint between the hare and the beginning of the sequence because of the way they move. Therefore the tortoise is $n \times L$ away from the beginning of the sequence as well.

If we move both one step at a time, from the position of the tortoise and from the start of the sequence, we know that they will meet as soon as both are in the loop, since they are $n \times L$, a multiple of the loop length, apart. One of them is already in the loop, so we just move the other one in single step until it enters the loop, keeping the other $n \times L$ away from it at all times.

Problem-13 In Floyd cycle finding algorithm, does it work if we use steps 2 and 3 instead of 1 and 2?

Solution: Yes, but the complexity might be high. Trace out an example.

Problem-14 Check whether the given linked list is NULL-terminated. If there is a cycle, find the length of the loop.

Solution: This solution is also an extension of the basic cycle detection problem. After finding the loop in the linked list, keep the *slowPtr* as it is. The *fastPtr* keeps on moving until it again comes back to *slowPtr*. While moving *fastPtr*, use a counter variable which increments at the rate of 1.

```
def findLoopLength( self ):
  if None == self.head or None == self.head.next:
    return 0
  # slow and fast both started at head after one step,
  # slow is at self.head.next and fast is at self.head.next.next
  slow = self.head.next
  fast = slow.next
```

```
    # each keep walking until they meet again.
    while slow != fast:
      slow = slow.next
      try:
        fast = fast.next.next
      except AttributeError:
        return 0 # no cycle if NoneType reached
    loopLength = 0
    slow = slow.next
    while slow != fast:
      slow = slow.next
      loopLength = loopLength + 1
    return loopLength
```

Time Complexity: O(n). Space Complexity: O(1).

Problem-15 Insert a node in a sorted linked list.

Solution: Traverse the list and find a position for the element and insert it.

```
def orderedInsert(self,item):
    current = self.head
    previous = None
    stop = False
    while current != None and not stop:
        if current.getData() > item:
            stop = True
        else:
            previous = current
            current = current.getNext()
    temp = Node(item)
    if previous == None:
        temp.setNext(self.head)
        self.head = temp
    else:
        temp.setNext(current)
        previous.setNext(temp)
```

Time Complexity: O(n). Space Complexity: O(1).

Problem-16 Reverse a singly linked list.

Solution: This algorithm reverses this singly linked list in place, in O(n). The function uses three pointers to walk the list and reverse link direction between each pair of nodes.

```
# Iterative version
def reverseList(self):
        last = None
        current = self.head
        while(current is not None):
                nextNode = current.getNext()
                current.setNext(last)
                last = current
                current = nextNode
        self.head = last
```

Time Complexity: O(n). Space Complexity: O(1).

Recursive version: We can find it easier to start from the bottom up, by asking and answering tiny questions (this is the approach in The Little Lisper):

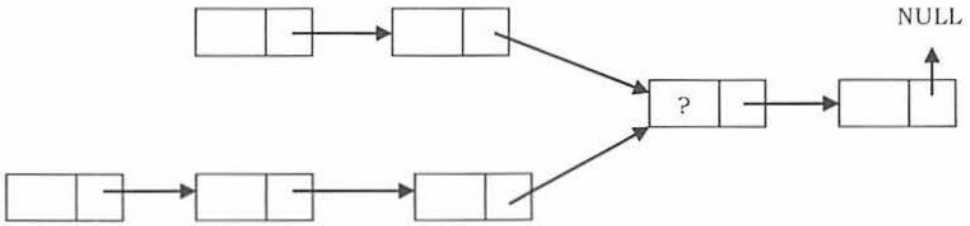
- What is the reverse of NULL (the empty list)? NULL.
- What is the reverse of a one element list? The element itself.
- What is the reverse of an n element list? The reverse of the second element followed by the first element.

```
def reverseRecursive( self, n ) :
        if None != n:
```

```
        right = n.getNext()
        if self.head != n:
                n.setNext(self.head)
                self.head = n
        else:
                n.setNext(None)
        self.reverseRecursive( right )
```

Time Complexity: O(n). Space Complexity: O(n), for recursive stack.

Problem-17 Suppose there are two singly linked lists both of which intersect at some point and become a single linked list. The head or start pointers of both the lists are known, but the intersecting node is not known. Also, the number of nodes in each of the lists before they intersect is unknown and may be different in each list. *List*1 may have n nodes before it reaches the intersection point, and *List*2 might have m nodes before it reaches the intersection point where m and n may be $m = n, m < n$ or $m > n$. Give an algorithm for finding the merging point.



Solution: Brute-Force Approach: One easy solution is to compare every node pointer in the first list with every other node pointer in the second list by which the matching node pointers will lead us to the intersecting node. But, the time complexity in this case will be O(mn) which will be high.

Time Complexity: O(mn). Space Complexity: O(1).

Problem-18 Can we solve Problem-17 using the sorting technique?

Solution: No. Consider the following algorithm which is based on sorting and see why this algorithm fails.

Algorithm:

- Take first list node pointers and keep them in some array and sort them.
- Take second list node pointers and keep them in some array and sort them.
- After sorting, use two indexes: one for the first sorted array and the other for the second sorted array.
- Start comparing values at the indexes and increment the index according to whichever has the lower value (increment only if the values are not equal).
- At any point, if we are able to find two indexes whose values are the same, then that indicates that those two nodes are pointing to the same node and we return that node.

Time Complexity: Time for sorting lists + Time for scanning (for comparing)
$= O(mlogm) + O(nlogn) + O(m + n)$ We need to consider the one that gives the maximum value.

Space Complexity: O(1).

Any problem with the above algorithm? Yes. In the algorithm, we are storing all the node pointers of both the lists and sorting. But we are forgetting the fact that there can be many repeated elements. This is because after the merging point, all node pointers are the same for both the lists. The algorithm works fine only in one case and it is when both lists have the ending node at their merge point.

Problem-19 Can we solve Problem-17 using hash tables?

Solution: Yes.

Algorithm:

- Select a list which has less number of nodes (If we do not know the lengths beforehand then select one list randomly).
- Now, traverse the other list and for each node pointer of this list check whether the same node pointer exists in the hash table.
- If there is a merge point for the given lists then we will definitely encounter the node pointer in the hash table.

```
def findIntersectingNode( self, list1, list2 ):
    intersect = {}
    t = list1
```