Table of Contents

0. Organization of Chapters ---- 15
0.1 What Is This Book About? ---- 15
0.2 Should I Buy This Book? ---- 15
0.3 Organization of Chapters ---- 15
0.4 Some Prerequisites ---- 18
1. Introduction ---- 19
1.1 Variables ---- 19
1.2 Data Types ---- 19
1.3 Data Structures ---- 20
1.4 Abstract Data Types (ADTs) ---- 20
1.5 What is an Algorithm? ---- 20
1.6 Why the Analysis of Algorithms? ---- 20
1.7 Goal of the Analysis of Algorithms ---- 20
1.8 What is Running Time Analysis? ---- 20
1.9 How to Compare Algorithms ---- 21
1.10 What is Rate of Growth? ---- 21
1.11 Commonly Used Rates of Growth ---- 21
1.12 Types of Analysis ---- 22
1.13 Asymptotic Notation ---- 22
1.14 Big-O Notation ---- 22
1.15 Omega-Ω Notation ---- 23
1.16 Theta-Θ Notation ---- 24
1.17 Why is it called Asymptotic Analysis? ---- 25
1.18 Guidelines for Asymptotic Analysis ---- 25
1.19 Simplifying properties of asymptotic notations ---- 26
1.21 Commonly used Logarithms and Summations ---- 26
1.21 Master Theorem for Divide and Conquer Recurrences ---- 27
1.22 Divide and Conquer Master Theorem: Problems & Solutions ---- 27
1.23 Master Theorem for Subtract and Conquer Recurrences ---- 28
1.24 Variant of Subtraction and Conquer Master Theorem ---- 28
1.25 Method of Guessing and Confirming ---- 28
1.26 Amortized Analysis ---- 29
1.27 Algorithms Analysis: Problems & Solutions ---- 30
2. Recursion and Backtracking ---- 41
2.1 Introduction ---- 41
2.2 What is Recursion? ---- 41
2.3 Why Recursion? ---- 41
2.4 Format of a Recursive Function ---- 41
2.5 Recursion and Memory (Visualization) ---- 42

2.6 Recursion versus Iteration ---42
2.7 Notes on Recursion ---42
2.8 Example Algorithms of Recursion---43
2.9 Recursion: Problems & Solutions---43
2.10 What is Backtracking?---43
2.11 Example Algorithms of Backtracking ---44
2.12 Backtracking: Problems & Solutions---44

3. Linked Lists--- 47
3.1 What is a Linked List?---47
3.2 Linked Lists ADT---47
3.3 Why Linked Lists? ---47
3.4 Arrays Overview ---47
3.5 Comparison of Linked Lists with Arrays and Dynamic Arrays---48
3.6 Singly Linked Lists---49
3.7 Doubly Linked Lists ---54
3.8 Circular Linked Lists---58
3.9 A Memory-efficient Doubly Linked List---62
3.11 Skip Lists---67
3.12 Linked Lists: Problems & Solutions ---70

4. Stacks --- 90
4.1 What is a Stack? ---90
4.2 How Stacks are Used ---90
4.3 Stack ADT---90
4.4 Applications ---91
4.5 Implementation ---91
4.6 Comparison of Implementations ---95
4.7 Stacks: Problems & Solutions ---95

5. Queues--- 111
5.1 What is a Queue?---111
5.2 How are Queues Used---111
5.3 Queue ADT---111
5.4 Exceptions---111
5.5 Applications ---111
5.6 Implementation ---112
5.7 Queues: Problems & Solutions ---116

6. Trees--- 125
6.1 What is a Tree?---125
6.2 Glossary---125
6.3 Binary Trees---126
6.4 Types of Binary Trees---126
6.5 Properties of Binary Trees---126
6.6 Binary Tree Traversals---128
6.7 Generic Trees (N-ary Trees)---146

6.8 Threaded Binary Tree Traversals (Stack or Queue-less Traversals) ---- 152
6.9 Expression Trees ---- 156
6.10 XOR Trees ---- 158
6.11 Binary Search Trees (BSTs) ---- 159
6.12 Balanced Binary Search Trees ---- 171
6.13 AVL (Adelson-Velskii and Landis) Trees ---- 172
6.14 Other Variations on Trees ---- 186
6.15 Supplementary Questions ---- 189

7. Priority Queues and Heaps ---- 190

7.1 What is a Priority Queue? ---- 190
7.2 Priority Queue ADT ---- 190
7.3 Priority Queue Applications ---- 190
7.4 Priority Queue Implementations ---- 190
7.5 Heaps and Binary Heaps ---- 191
7.6 Binary Heaps ---- 192
7.7 Heapsort ---- 195
7.8 Priority Queues [Heaps]: Problems & Solutions ---- 196

8. Disjoint Sets ADT ---- 208

8.1 Introduction ---- 208
8.2 Equivalence Relations and Equivalence Classes ---- 208
8.3 Disjoint Sets ADT ---- 208
8.4 Applications ---- 208
8.5 Tradeoffs in Implementing Disjoint Sets ADT ---- 209
8.8 Fast UNION Implementation (Slow FIND) ---- 209
8.9 Fast UNION Implementations (Quick FIND) ---- 211
8.10 Summary ---- 212
8.11 Disjoint Sets: Problems & Solutions ---- 212

9. Graph Algorithms ---- 214

9.1 Introduction ---- 214
9.2 Glossary ---- 214
9.3 Applications of Graphs ---- 216
9.4 Graph Representation ---- 216
9.5 Graph Traversals ---- 221
9.6 Topological Sort ---- 226
9.7 Shortest Path Algorithms ---- 228
9.8 Minimal Spanning Trees ---- 233
9.9 Graph Algorithms: Problems & Solutions ---- 236

10. Sorting ---- 252

10.1 What is Sorting? ---- 252
10.2 Why is Sorting Necessary? ---- 252
10.3 Classification of Sorting Algorithms ---- 252
10.4 Other Classifications ---- 252
10.5 Bubble Sort ---- 253

10.6 Selection Sort ---------- 254
10.7 Insertion Sort ---------- 255
10.8 Shell Sort ---------- 256
10.9 Merge Sort ---------- 258
10.11 Quick Sort ---------- 259
10.12 Tree Sort ---------- 263
10.13 Comparison of Sorting Algorithms ---------- 264
10.14 Linear Sorting Algorithms ---------- 264
10.15 Counting Sort ---------- 264
10.16 Bucket Sort (or Bin Sort) ---------- 265
10.17 Radix Sort ---------- 266
10.18 Topological Sort ---------- 266
10.19 External Sorting ---------- 266
10.20 Sorting: Problems & Solutions ---------- 267

11. Searching ---------- 279

11.1 What is Searching? ---------- 279
11.2 Why do we need Searching? ---------- 279
11.3 Types of Searching ---------- 279
11.4 Unordered Linear Search ---------- 279
11.5 Sorted/Ordered Linear Search ---------- 279
11.6 Binary Search ---------- 280
11.7 Interpolation Search ---------- 280
11.8 Comparing Basic Searching Algorithms ---------- 281
11.9 Symbol Tables and Hashing ---------- 282
11.10 String Searching Algorithms ---------- 282
11.11 Searching: Problems & Solutions ---------- 282

12. Selection Algorithms [Medians] ---------- 303

12.1 What are Selection Algorithms? ---------- 303
12.2 Selection by Sorting ---------- 303
12.3 Partition-based Selection Algorithm ---------- 303
12.4 Linear Selection Algorithm - Median of Medians Algorithm ---------- 303
12.5 Finding the K Smallest Elements in Sorted Order ---------- 303
12.6 Selection Algorithms: Problems & Solutions ---------- 303

13. Symbol Tables ---------- 312

13.1 Introduction ---------- 312
13.2 What are Symbol Tables? ---------- 312
13.3 Symbol Table Implementations ---------- 312
13.4 Comparison Table of Symbols for Implementations ---------- 313

14. Hashing ---------- 314

14.1 What is Hashing? ---------- 314
14.2 Why Hashing? ---------- 314
14.3 Hash Table ADT ---------- 314
14.4 Understanding Hashing ---------- 314

14.5 Components of Hashing ---------- 315
14.6 Hash Table ---------- 315
14.7 Hash Function ---------- 316
14.8 Load Factor ---------- 316
14.9 Collisions ---------- 317
14.10 Collision Resolution Techniques ---------- 317
14.11 Separate Chaining ---------- 317
14.12 Open Addressing ---------- 317
14.13 Comparison of Collision Resolution Techniques ---------- 318
14.14 How Hashing Gets O(1) Complexity ---------- 319
14.15 Hashing Techniques ---------- 319
14.16 Problems for which Hash Tables are not suitable ---------- 319
14.17 Bloom Filters ---------- 319
14.18 Hashing: Problems & Solutions ---------- 321

15. String Algorithms ---------- 329
15.1 Introduction ---------- 329
15.2 String Matching Algorithms ---------- 329
15.3 Brute Force Method ---------- 329
15.4 Rabin-Karp String Matching Algorithm ---------- 330
15.5 String Matching with Finite Automata ---------- 331
15.6 KMP Algorithm ---------- 332
15.7 Boyer-Moore Algorithm ---------- 334
15.8 Data Structures for Storing Strings ---------- 334
15.9 Hash Tables for Strings ---------- 335
15.10 Binary Search Trees for Strings ---------- 335
15.11 Tries ---------- 335
15.12 Ternary Search Trees ---------- 337
15.13 Comparing BSTs, Tries and TSTs ---------- 342
15.14 Suffix Trees ---------- 342
15.15 String Algorithms: Problems & Solutions ---------- 344

16. Algorithms Design Techniques ---------- 351
16.1 Introduction ---------- 351
16.2 Classification ---------- 351
16.3 Classification by Implementation Method ---------- 351
16.4 Classification by Design Method ---------- 352
16.5 Other Classifications ---------- 352

17. Greedy Algorithms ---------- 354
17.1 Introduction ---------- 354
17.2 Greedy Strategy ---------- 354
17.3 Elements of Greedy Algorithms ---------- 354
17.4 Does Greedy Always Work? ---------- 354
17.5 Advantages and Disadvantages of Greedy Method ---------- 354
17.6 Greedy Applications ---------- 354

17.7 Understanding Greedy Technique ---- 355
17.8 Greedy Algorithms: Problems & Solutions ---- 357
18. Divide and Conquer Algorithms ---- 362
18.1 Introduction ---- 362
18.2 What is Divide and Conquer Strategy? ---- 362
18.3 Does Divide and Conquer Always Work? ---- 362
18.4 Divide and Conquer Visualization ---- 362
18.5 Understanding Divide and Conquer ---- 363
18.6 Advantages of Divide and Conquer ---- 363
18.7 Disadvantages of Divide and Conquer ---- 363
18.8 Master Theorem ---- 363
18.9 Divide and Conquer Applications ---- 364
18.10 Finding peak element of an array ---- 364
18.11 Finding peak element in two-dimensional array ---- 367
18.12 Divide and Conquer: Problems & Solutions ---- 372
19. Dynamic Programming ---- 386
19.1 Introduction ---- 386
19.2 What is Dynamic Programming Strategy? ---- 386
19.3 Properties of Dynamic Programming Strategy ---- 386
19.4 Greedy vs Divide and Conquer vs DP ---- 386
19.5 Can DP solve all problems? ---- 387
19.6 Dynamic Programming Approaches ---- 387
19.7 Understanding DP Approaches ---- 387
19.8 Examples of DP Algorithms ---- 390
19.9 Longest Common Subsequence ---- 390
19.10 Dynamic Programming: Problems & Solutions ---- 392
20. Complexity Classes ---- 420
20.1 Introduction ---- 420
20.2 Polynomial/Exponential Time ---- 420
20.3 What is a Decision Problem? ---- 420
20.4 Decision Procedure ---- 420
20.5 What is a Complexity Class? ---- 420
20.6 Types of Complexity Classes ---- 421
20.7 Reductions ---- 422
20.8 Complexity Classes: Problems & Solutions ---- 424
21. Miscellaneous Concepts ---- 426
21.1 Introduction ---- 426
21.2 Hacks on Bit-wise Programming ---- 426
References ---- 436

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 omelet, 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 omelet), 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 unambiguous instructions to solve a given problem.

In the traditional study of algorithms, there are two main criteria for judging the merits of algorithms: correctness (does the algorithm give solution to the problem in a finite number of steps?) and efficiency (how much resources (in terms of memory and time) does it take to execute the).

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).

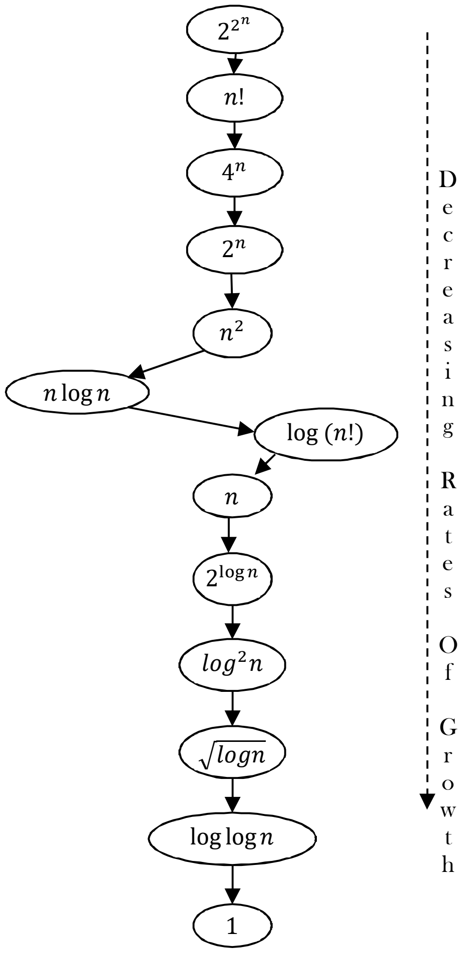
$$Total\ Cost\ =\ cost\_of\_car\ +\ cost\_of\_bicycle$$
$$Total\ Cost\ \approx\ cost\_of\_car\ (approximation)$$

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 (slowest time to complete).
  - Input is the one for which the algorithm runs the slowest.
- Best case
  - Defines the input for which the algorithm takes the least time (fastest time to complete).
  - Input is the one for which the algorithm runs the fastest.
- Average case
  - Provides a prediction about the running time of the algorithm.
  - Run the algorithm many times, using many different inputs that come from some distribution that generates these inputs, compute the total running time (by adding the individual times), and divide by the number of trials.
  - 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.

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

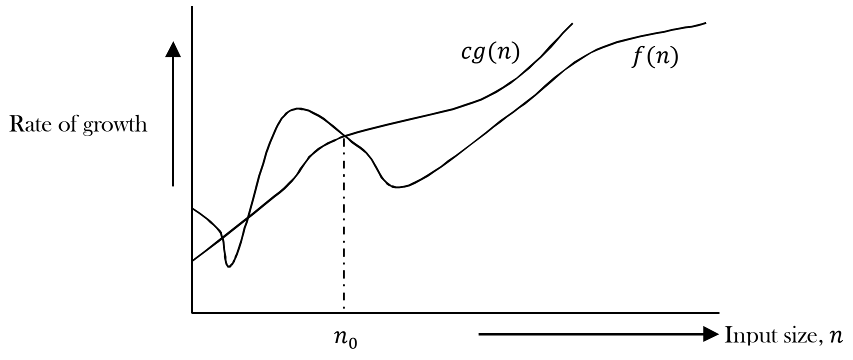
Similarly, for the average case. The expression defines the inputs with which the algorithm takes the average running time (or memory).

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.

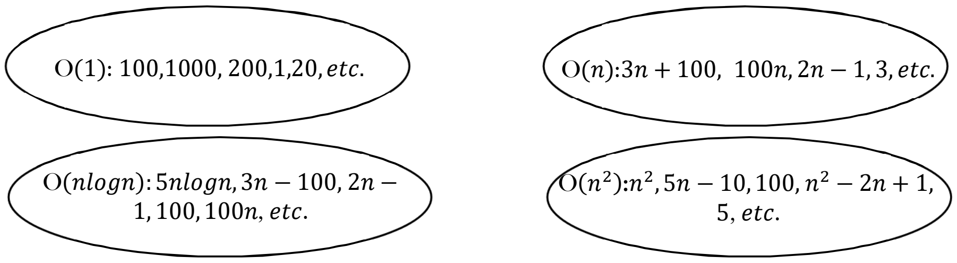


Let us see the O−notation with a little more detail. O−notation defined as $O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$. $g(n)$ is an asymptotic tight upper bound for $f(n)$. Our objective is to give the smallest rate of growth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


$O(1)$: $100, 1000, 200, 1, 20, etc.$
$O(n)$: $3n + 100,\ 100n, 2n - 1, 3, etc.$
$O(nlogn)$: $5nlogn, 3n - 100, 2n - 1, 100, 100n, etc.$
$O(n^2)$: $n^2, 5n - 10, 100, n^2 - 2n + 1, 5, etc.$

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 $n \geq 1$
$\therefore 2n^3 - 2n^2 = O(n^3)$ 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.

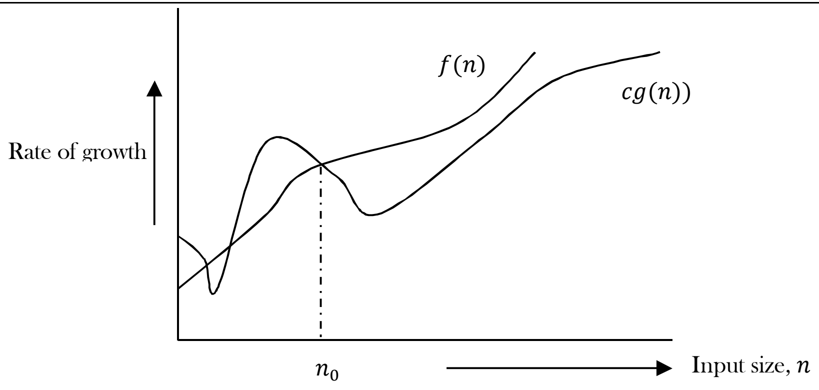
Solution1: $100n + 5 \leq 100n + n = 101n \leq 101n$, for all $n \geq 5$, $n_0 = 5$ and $c = 101$ is a solution.

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 $g(n)$. The Ω notation can be defined as $\Omega(g(n)) = \{f(n)$: 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 $f(n)$.

For example, if $f(n) = 100n^2 + 10n + 50$, $g(n)$ is $\Omega(n^2)$.



Ω Examples

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

Solution: $\exists\ c,\ n_0$ Such that: $0 \le cn^2 \le 5n^2 \Rightarrow cn^2 \le 5n^2 \Rightarrow c = 5$ and $n_0 = 1$

$\therefore 5n^2 = \Omega(n^2)$ with $c = 5$ and $n_0 = 1$

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

Solution: $\exists$ c, n_0 Such that: $0 \le cn^2 \le 100n + 5$

$100n + 5 \le 100n + 5n\ (\forall n \ge 1) = 105n$

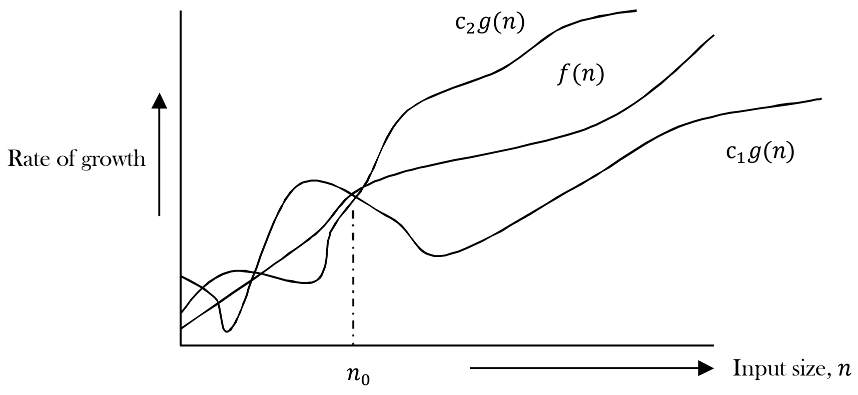
$cn^2 \le 105n \Rightarrow n(cn - 105) \le 0$

Since n is positive $\Rightarrow cn - 105 \le 0 \Rightarrow n \le 105/c$

$\Rightarrow$ 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 $0 \le c_1g(n) \le f(n) \le c_2g(n)$ for all $n \ge n_0\}$. $g(n)$ 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: $\frac{n^2}{5} \le \frac{n^2}{2} - \frac{n}{2} \le n^2$, for all, $n \ge 2$

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

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

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

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

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

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

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

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

Solution: $c_1 logn \le n \le c_2 \log n \Rightarrow c_2 \ge \frac{n}{\log n}, \forall n \ge n_0$ – 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, sep=" ")   #constant time
```

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

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, sep=" ")              #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, sep=" ")  #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 at $(k+1)^{th}$ step we come out of the $loop$. 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 Simplifying properties of asymptotic 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 $g(n) = \Omega(f(n))$.
- If $f(n)$ is in $O(kg(n))$ for any constant $k > 0$, then $f(n)$ is in $O(g(n))$.
- If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$, then $(f_1 + f_2)(n)$ is in $O(\max(g_1(n), g_1(n)))$.
- If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$ then $f_1(n)\, f_2(n)$ is in $O(g_1(n)\, g_1(n))$.

1.21 Commonly used Logarithms and Summations

Logarithms

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

$log\, xy = logx + logy$ $log^k n = (logn)^k$

$log\, logn = log(logn)$ $log \frac{x}{y} = logx - logy$

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

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 Recurrences

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 $T(n) = \Theta(n^{\log_b^a})$
2) If $a = b^k$
   a. If $p > -1$, then $T(n) = \Theta(n^{\log_b^a} \log^{p+1} n)$
   b. If $p = -1$, then $T(n) = \Theta(n^{\log_b^a} \log\log n)$
   c. If $p < -1$, then $T(n) = \Theta(n^{\log_b^a})$
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: $T(n) = 4T(n/2) + n^2 \Rightarrow T(n) = \Theta(n^2 \log n)$ (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 $T(n) = 2T(n/2) + n\log n$
Solution: $T(n) = 2T(n/2) + n\log n \Rightarrow T(n) = \Theta(n\log^2 n)$ (Master Theorem Case 2.a)

Problem-7 $T(n) = 2T(n/2) + n/\log n$
Solution: $T(n) = 2T(n/2) + n/\log n \Rightarrow T(n) = \Theta(n\log\log n)$ (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 $T(n) = 6T(n/3) + n^2 \log n$
Solution: $T(n) = 6T(n/3) + n^2\log n \Rightarrow T(n) = \Theta(n^2 \log n)$ (Master Theorem Case 3.a)

Problem-11 $T(n) = 64T(n/8) - n^2\log n$
Solution: $T(n) = 64T(n/8) - n^2\log n \Rightarrow$ Does not apply (function is not positive)

Problem-12 $T(n) = 7T(n/3) + n^2$
Solution: $T(n) = 7T(n/3) + n^2 => T(n) = \Theta(n^2)$ (Master Theorem Case 3.as)

Problem-13 $T(n) = 4T(n/2) + logn$
Solution: $T(n) = 4T(n/2) + logn => T(n) = \Theta(n^2)$ (Master Theorem Case 1)

Problem-14 $T(n) = 16T(n/4) + n!$
Solution: $T(n) = 16T(n/4) + n! => T(n) = \Theta(n!)$ (Master Theorem Case 3.a)

Problem-15 $T(n) = \sqrt{2}T(n/2) + logn$
Solution: $T(n) = \sqrt{2}T(n/2) + logn => T(n) = \Theta(\sqrt{n})$ (Master Theorem Case 1)

Problem-16 $T(n) = 3T(n/2) + n$
Solution: $T(n) = 3T(n/2) + n => T(n) = \Theta(n^{log3})$ (Master Theorem Case 1)

Problem-17 $T(n) = 3T(n/3) + \sqrt{n}$
Solution: $T(n) = 3T(n/3) + \sqrt{n} => T(n) = \Theta(n)$ (Master Theorem Case 1)

Problem-18 $T(n) = 4T(n/2) + cn$
Solution: $T(n) = 4T(n/2) + cn => T(n) = \Theta(n^2)$ (Master Theorem Case 1)

Problem-19 $T(n) = 3T(n/4) + nlogn$
Solution: $T(n) = 3T(n/4) + nlogn => T(n) = \Theta(nlogn)$ (Master Theorem Case 3.a)

Problem-20 $T(n) = 3T(n/3) + n/2$
Solution: $T(n) = 3T(n/3) + n/2 => T(n) = \Theta(nlogn)$ (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 $T(n) \leq cnlogn$:

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

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.

CHAPTER

TREES 6

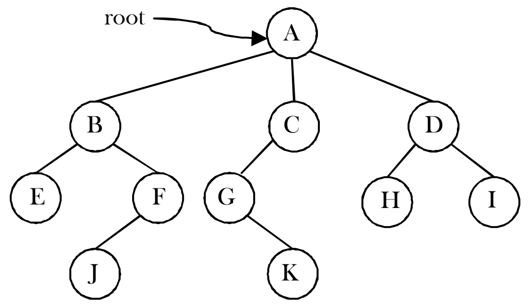
☼ ☼ ☼

6.1 What is a Tree?

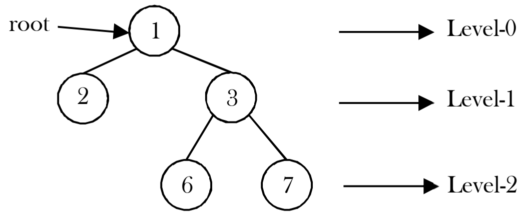
A $tree$ is a data structure similar to a linked list but instead of each node pointing simply to the next node in a linear fashion, each node points to a number of nodes. Tree is an example of non-linear data structures. A $tree$ structure is a way of representing the hierarchical nature of a structure in a graphical form.

In trees ADT (Abstract Data Type), the order of the elements is not important. If we need ordering information linear data structures like linked lists, stacks, queues, etc. can be used.

6.2 Glossary

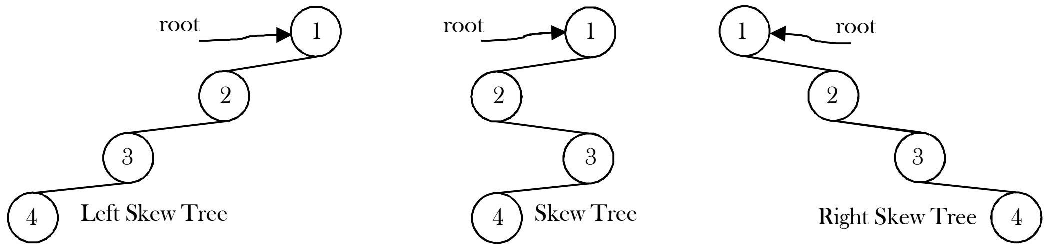


- The $root$ of a tree is the node with no parents. There can be at most one root node in a tree (node A in the above example).
- An $edge$ refers to the link from parent to child (all links in the figure).
- A node with no children is called $leaf$ node (E, J, K, H and I).
- Children of same parent are called $siblings$ (B, C, D are siblings of A, and E, F are the siblings of B).
- A node p is an $ancestor$ of node q if there exists a path from $root$ to q and p appears on the path. The node q is called a $descendant$ of p. For example, A, C and G are the ancestors of K.
- The set of all nodes at a given depth is called the $level$ of the tree (B, C and D are the same level). The root node is at level zero.



- The $depth$ of a node is the length of the path from the root to the node (depth of G is 2, $A - C - G$).
- The $height$ of a node is the length of the path from that node to the deepest node. The height of a tree is the length of the path from the root to the deepest node in the tree. A (rooted) tree with only one node (the root) has a height of zero. In the previous example, the height of B is 2 ($B - F - J$).
- $Height\ of\ the\ tree$ is the maximum height among all the nodes in the tree and $depth\ of\ the\ tree$ is the maximum depth among all the nodes in the tree. For a given tree, depth and height returns the same value. But for individual nodes we may get different results.
- The size of a node is the number of descendants it has including itself (the size of the subtree C is 3).

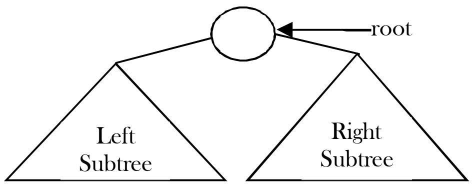
- If every node in a tree has only one child (except leaf nodes) then we call such trees *skew trees*. If every node has only left child then we call them *left skew trees*. Similarly, if every node has only right child then we call them *right skew trees*.

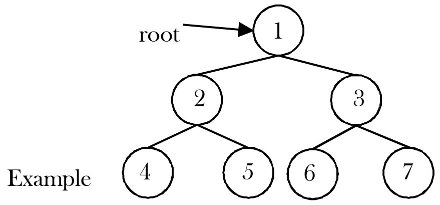


6.3 Binary Trees

A tree is called *binary tree* if each node has zero child, one child or two children. Empty tree is also a valid binary tree. We can visualize a binary tree as consisting of a root and two disjoint binary trees, called the left and right subtrees of the root.

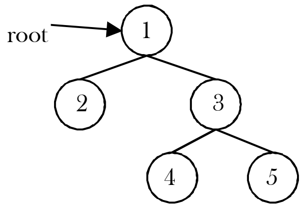
Generic Binary Tree



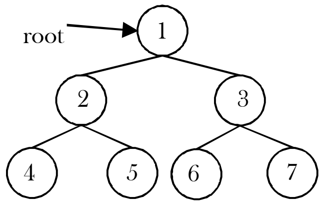


6.4 Types of Binary Trees

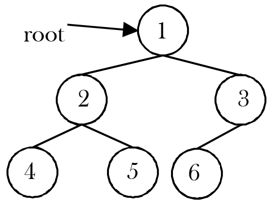
Strict Binary Tree: A binary tree is called *strict binary tree* if each node has exactly two children or no children.



Full Binary Tree: A binary tree is called *full binary tree* if each node has exactly two children and all leaf nodes are at the same level.

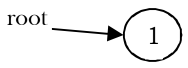


Complete Binary Tree: Before defining the *complete binary tree*, let us assume that the height of the binary tree is h. In complete binary trees, if we give numbering for the nodes by starting at the root (let us say the root node has 1) then we get a complete sequence from 1 to the number of nodes in the tree. While traversing we should give numbering for None pointers also. A binary tree is called *complete binary tree* if all leaf nodes are at height h or $h-1$ and also without any missing number in the sequence.

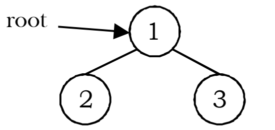


6.5 Properties of Binary Trees

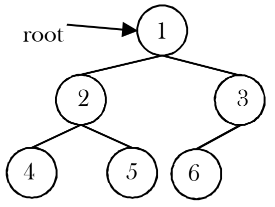
For the following properties, let us assume that the height of the tree is h. Also, assume that root node is at height zero.



Height	Number of nodes at level h
$h = 0$	$2^0 = 1$



$h = 1$ $2^1 = 2$



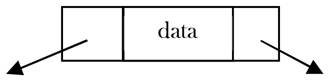
$h = 2$ $2^2 = 4$

From the diagram we can infer the following properties:

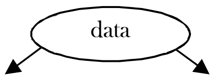
- The number of nodes n in a full binary tree is $2^{h+1} - 1$. Since, there are h levels we need to add all nodes at each level $[2^0 + 2^1 + 2^2 + \cdots + 2^h = 2^{h+1} - 1]$.
- The number of nodes n in a complete binary tree is between 2^h (minimum) and $2^{h+1} - 1$ (maximum). For more information on this, refer to *Priority Queues* chapter.
- The number of leaf nodes in a full binary tree is 2^h.
- The number of None links (wasted pointers) in a complete binary tree of n nodes is $n + 1$.

Structure of Binary Trees

Now let us define structure of the binary tree. One way to represent a node (which contains data) is to have two links which point to left and right children along with data fields as shown below:



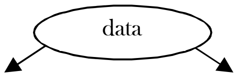
Or

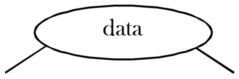


```
'''Binary Tree Class and its methods'''
class BinaryTreeNode:
    def __init__(self, data):
        self.data = data   #root node
        self.left = None  #left child
        self.right = None                #right child
    #set data
    def setData(self, data):
        self.data = data
    #get data
    def getData(self):
        return self.data
    #get left child of a node
    def getLeft(self):
        return self.left
    #get right child of a node
    def getRight(self):
        return self.right
```

```
class BinaryTree:
    def __init__(self, root = None):
        self.root = root
```

Note: In trees, the default flow is from parent to children and it is not mandatory to show directed branches. For our discussion, we assume both the representations shown below are the same.





Operations on Binary Trees

Basic Operations

- Inserting an element into a tree
- Deleting an element from a tree
- Searching for an element
- Traversing the tree

Auxiliary Operations

- Finding the size of the tree
- Finding the height of the tree
- Finding the level which has maximum sum
- Finding the least common ancestor (LCA) for a given pair of nodes, and many more.

Applications of Binary Trees

Following are the some of the applications where *binary trees* play an important role:

- Expression trees are used in compilers.
- Huffman coding trees that are used in data compression algorithms.
- Binary Search Tree (BST), which supports search, insertion and deletion on a collection of items in $O(logn)$ (average).
- Priority Queue (PQ), which supports search and deletion of minimum (or maximum) on a collection of items in logarithmic time (in worst case).

6.6 Binary Tree Traversals

In order to process trees, we need a mechanism for traversing them, and that forms the subject of this section. The process of visiting all nodes of a tree is called *tree traversal*. Each node is processed only once but it may be visited more than once. As we have already seen in linear data structures (like linked lists, stacks, queues, etc.), the elements are visited in sequential order. But, in tree structures there are many different ways.

Tree traversal is like searching the tree, except that in traversal the goal is to move through the tree in a particular order. In addition, all nodes are processed in the *traversal but searching* stops when the required node is found.

Traversal Possibilities

Starting at the root of a binary tree, there are three main steps that can be performed and the order in which they are performed defines the traversal type. These steps are: performing an action on the current node (referred to as "visiting" the node and denoted with "D"), traversing to the left child node (denoted with "L"), and traversing to the right child node (denoted with "R"). This process can be easily described through recursion. Based on the above definition there are 6 possibilities:

1. LDR: Process left subtree, process the current node data and then process right subtree
2. LRD: Process left subtree, process right subtree and then process the current node data
3. DLR: Process the current node data, process left subtree and then process right subtree
4. DRL: Process the current node data, process right subtree and then process left subtree
5. RDL: Process right subtree, process the current node data and then process left subtree
6. RLD: Process right subtree, process left subtree and then process the current node data

Classifying the Traversals

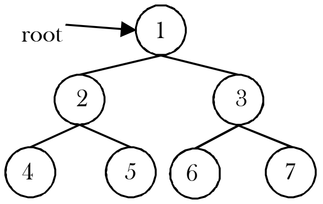
The sequence in which these entities (nodes) are processed defines a particular traversal method. The classification is based on the order in which current node is processed. That means, if we are classifying based on current node (D) and if D comes in the middle then it does not matter whether L is on left side of D or R is on left side of D. Similarly, it does not matter whether L is on right side of D or R is on right side of D. Due to this, the total 6 possibilities are reduced to 3 and these are:

- Preorder Traversal: DLR *or* DRD
- Inorder Traversal: LDR *or* RDL
- Postorder Traversal: LRD *or* RLD

There is another traversal method which does not depend on the above orders and it is:

- Level Order Traversal: This method is inspired from Breadth First Traversal (BFS of Graph algorithms).

Let us use the diagram below for the remaining discussion.



PreOrder Traversal

In preorder traversal, each node is processed before (pre) either of its subtrees. This is the simplest traversal to understand. However, even though each node is processed before the subtrees, it still requires that some information must be maintained while moving down the tree. In the example above, 1 is processed first, then the left subtree, and this is followed by the right subtree.

Therefore, processing must return to the right subtree after finishing the processing of the left subtree. To move to the right subtree after processing the left subtree, we must maintain the root information. The obvious ADT for such information is a stack. Because of its LIFO structure, it is possible to get the information about the right subtrees back in the reverse order.

Preorder traversal is defined as follows:

- Visit the root.
- Traverse the left subtree in Preorder.
- Traverse the right subtree in Preorder.

The nodes of tree would be visited in the order: 1 2 4 5 3 6 7

```
class BinaryTree:
    def __init__(self, root = None):
        self.root = root
    def preOrder(self, root):                    ←DLR      DRL→      def preOrder2(self, root):
        if root == None:                                                 if root == None:
            return                                                           return
        print (root.data, sep= "-->", end="-->")                         print (root.data, sep= "-->", end="-->")
        self.preOrder(root.left)                                         self.preOrder2(root.right)
        self.preOrder(root.right)                                        self.preOrder2(root.left)
```

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

Non-Recursive Preorder Traversal

In the recursive version, a stack is required as we need to remember the current node so that after completing the left subtree we can go to the right subtree. To simulate the same, first we process the current node and before going to the left subtree, we store the current node on stack. After completing the left subtree processing, pop the element and go to its right subtree. Continue this process until stack is nonempty.

```
#Pre-order iterative traversal. The nodes' values are appended to the result list in traversal order
def preorderIterative(self, root, result):
    if not root:
        return
    stack = []
    stack.append(root)
    while stack:
        node = stack.pop()
        result.append(node.data)
        if node.right: stack.append(node.right)
        if node.left: stack.append(node.left)
```

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

InOrder Traversal

In Inorder Traversal the root is visited between the subtrees. Inorder traversal is defined as follows:

- Traverse the left subtree in Inorder.
- Visit the root.
- Traverse the right subtree in Inorder.

The nodes of tree would be visited in the order: 4 2 5 1 6 3 7

```
class BinaryTree:
    def __init__(self, root = None):
        self.root = root
    def preOrder(self, root):                    ←LDR      RDL→      def inOrder2(self, root):
        if root == None:                                                 if root == None:
            return                                                           return
        self.inOrder(root.left)                                          self.inOrder2(root.right)
        print (root.data, sep= "-->", end="-->")                         print (root.data, sep= "-->", end="-->")
        self.inOrder(root.right)                                         self.inOrder2(root.left)
```

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

Non-Recursive Inorder Traversal

The Non-recursive version of Inorder traversal is similar to Preorder. The only change is, instead of processing the node before going to left subtree, process it after popping (which is indicated after completion of left subtree processing).

```
# In-order iterative traversal. The nodes' values are appended to the result list in traversal order
def inorderIterative(self, root, result):
    if not root:
        return
    stack = []
    node = root
    while stack or node:
        if node:
            stack.append(node)
            node = node.left
        else:
            node = stack.pop()
            1result.append(node.data)
            node = node.right
```

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

PostOrder Traversal

In postorder traversal, the root is visited after both subtrees. Postorder traversal is defined as follows:

- Traverse the left subtree in Postorder.
- Traverse the right subtree in Postorder.
- Visit the root.

The nodes of the tree would be visited in the order: 4 5 2 6 7 3 1

```
class BinaryTree:
    def __init__(self, root = None):
        self.root = root
    def postOrder(self, root):                          ←LRD      RLD→         def postOrder2(self, root):
        if root == None:                                                           if root == None:
            return                                                                     return
        self.postOrder(root.left)                                                  self.postOrder2(root.right)
        self.postOrder(root.right)                                                 self.postOrder2(root.left)
        print (root.data, sep= "-->", end="-->")                                   print (root.data, sep= "-->", end="-->")
```

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

Non-Recursive Postorder Traversal

In preorder and inorder traversals, after popping the stack element we do not need to visit the same vertex again. But in postorder traversal, each node is visited twice. That means, after processing the left subtree we will visit the current node and after processing the right subtree we will visit the same current node. But we should be processing the node during the second visit. Here the problem is how to differentiate whether we are returning from the left subtree or the right subtree.

We use a $previous$ variable to keep track of the earlier traversed node. Let's assume $current$ is the current node that is on top of the stack. When $previous$ is $current's$ parent, we are traversing down the tree. In this case, we try to traverse to $current's$ left child if available (i.e., push left child to the stack). If it is not available, we look at $current's$ right child. If both left and right child do not exist (ie, $current$ is a leaf node), we print $current's$ value and pop it off the stack.

If prev is $current's$ left child, we are traversing up the tree from the left. We look at $current's$ right child. If it is available, then traverse down the right child (i.e., push right child to the stack); otherwise print $current's$ value and pop it off the stack. If $previous$ is $current's$ right child, we are traversing up the tree from the right. In this case, we print $current's$ value and pop it off the stack.

```
# Post-order iterative traversal. The nodes' values are appended to the result list in traversal order
def postorderIterative(root, result):
    if not root:
        return
    visited = set()
    stack = []
    node = root
    while stack or node:
        if node:
            stack.append(node)
            node = node.left
        else:
            node = stack.pop()
            if node.right and not node.right in visited:
                stack.append(node)
                node = node.right
            else:
                visited.add(node)
                result.append(node.data)
                node = None
```

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

Level Order Traversal

Level order traversal is defined as follows:

- Visit the root.
- While traversing level l, keep all the elements at level $l + 1$ in queue.
- Go to the next level and visit all the nodes at that level.
- Repeat this until all levels are completed.

The nodes of the tree are visited in the order: 1 2 3 4 5 6 7

```
#Implementation with Python module Queue
import Queue
def levelOrder(root, result):
        if root is None:
          return
        q = Queue.Queue()
        q.put(root )
        node = None
        while not q.empty():
          node = q.get()                              # deQueue FIFO
          result.append(node.data)
          if node.left is not None:
            q.put( node.left )
          if node.right is not None:
            q.put( node.right )
    #Implementation with implementation of Queue
    class Queue:
        def __init__(self):
            self.array = []
        def enQueue(self, data):
            self.array.append(data)
        def deQueue(self):
            if len(self.array)==0:
                return None
            return self.array.pop(0)
        def size(self):
            return len(self.array)
        def front(self):
            if len(self.array)==0:
                return None
            return self.array[0]
        def isEmpty(self):
            return len(self.array)==0
    class BTNode:
        def __init__(self, data = None, left = None, right = None):
            self.data = data
            self.left = left
            self.right = right
    class BinaryTree:
        def __init__(self, root = None):
            self.root = root
        def levelOrder(self, root):
            if root == None:
                return
            q = Queue()
            q.enQueue(root)
            while not q.isEmpty():
                temp = q.deQueue()
                print (temp.data, sep= "-->", end="-->")
                if temp.left:
                    q.enQueue(temp.left)
                if temp.right:
                    q.enQueue(temp.right)
```

Time Complexity: O(n).
Space Complexity: O(n). Since, in the worst case, all the nodes on the entire last level could be in the queue simultaneously.

Binary Trees: Problems & Solutions

Problem-1 Give an algorithm for finding maximum element in binary tree.

Solution: One simple way of solving this problem is: find the maximum element in left subtree, find the maximum element in right sub tree, compare them with root data and select the one which is giving the maximum value. This approach can be easily implemented with recursion.

```
maxData = float("-inf")
def findMaxRecursive(root): # maxData is the initially the value of root
        global maxData
        if not root:
            return maxData
        if root.data > maxData:
            maxData = root.data
        findMaxRecursive(root.left)
        findMaxRecursive(root.right)
        return maxData
```

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

Problem-2 Give an algorithm for finding the maximum element in binary tree without recursion.

Solution: Using level order traversal: just observe the element's data while deleting.

```
def findMaxUsingLevelOrder(root):
        if root is None:
            return
        q = Queue()
        q.enQueue( root )
        node = None
        maxElement = 0
        while not q.isEmpty():
            node = q.deQueue()                          # deQueue FIFO
            if maxElement < node.data:
                maxElement = node.data
                if node.left is not None:
                    q.enQueue( node.left )
                if node.right is not None:
                    q.enQueue( node.right )
print (maxElement)
```

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

Problem-3 Give an algorithm for searching an element in binary tree.

Solution: Given a binary tree, return true if a node with data is found in the tree. Recurse down the tree, choose the left or right branch by comparing data with each node's data.

```
def findRecursive(root, data):
        if not root:
            return 0
        if root.data == data:
            return 1
        else:
            temp = findRecursive(root.left, data)
            if temp == 1:
                return temp
            else:
                return findRecursive(root.right, data)
```

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

Problem-4 Give an algorithm for searching an element in binary tree without recursion.

Solution: We can use level order traversal for solving this problem. The only change required in level order traversal is, instead of printing the data, we just need to check whether the root data is equal to the element we want to search.

```
def findUsingLevelOrder(root, data):
        if root is None:
            return -1
        q = Queue()
        q.enQueue( root )
        node = None
        while not q.isEmpty():
                node = q.deQueue()                      # deQueue FIFO
                if data == node.data:
                    return 1
                if node.left is not None:
                    q.enQueue( node.left )
                if node.right is not None:
                    q.enQueue( node.right )
        return 0
```

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

Problem-5 Give an algorithm for inserting an element into binary tree.

Solution: Since the given tree is a binary tree, we can insert the element wherever we want. To insert an element, we can use the level order traversal and insert the element wherever we find the node whose left or right child is None.

```
'''Binary Tree Class and its methods'''
```

```
class BinaryTree:
        def __init__(self, data):
                self.data = data          #root node
                self.left = None          #left child
                self.right = None         #right child
        #set data
        def setData(self, data):
                self.data = data
        #get data
        def getData(self):
                return self.data
        #get left child of a node
        def getLeft(self):
                return self.left
        #get right child of a node
        def getRight(self):
                return self.right
        def insertLeft(self, newNode):
                if self.left == None:
                        self.left = BinaryTree(newNode)
                else:
                        temp = BinaryTree(newNode)
                        temp.left = self.left
                        self.left = temp
        def insertRight(self, newNode):
            if self.right == None:
                self.right = BinaryTree(newNode)
            else:
                temp = BinaryTree(newNode)
                temp.right = self.right
                self.right = temp
# Insert using level order traversal
def insertInBinaryTreeUsingLevelOrder(root, data):
        newNode = BinaryTree(data)
        if root is None:
            root = newNode
            return root
        q = Queue()
        q.enQueue( root )
        node = None
        while not q.isEmpty():
            node = q.deQueue()          # deQueue FIFO
            if data == node.data:
                return root
            if node.left is not None:
                q.enQueue( node.left )
            else:
                node.left = newNode
                return root
            if node.right is not None:
                q.enQueue( node.right )
            else:
                node.right = newNode
        return root
```

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

Problem-6 Give an algorithm for finding the size of binary tree.

Solution: Calculate the size of left and right subtrees recursively, add 1 (current node) and return to its parent.

```
# Compute the number of nodes in a tree.
def findSizeRecursive(root):
        if not root:
            return 0
        return findSizeRecursive(root.left) + findSizeRecursive(root.right) + 1
```

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

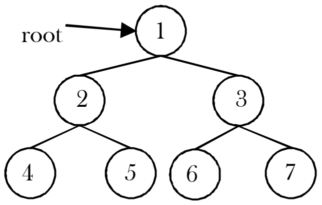
Problem-7 Can we solve Problem-6 without recursion?

Solution: Yes, using level order traversal.

```
def findSizeUsingLevelOrder(root):
        if root is None: return 0
        q = Queue()
        q.enQueue( root )
        node = None
        count = 0
        while not q.isEmpty():
            node = q.deQueue()          # deQueue FIFO
            count += 1
            if node.left is not None:
                q.enQueue( node.left )
            if node.right is not None:
                q.enQueue( node.right )
        return count
```

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

Problem-8 Give an algorithm for printing the level order data in reverse order. For example, the output for the below tree should be: 4 5 6 7 2 3 1



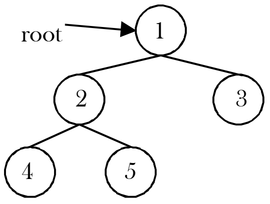
Solution: Maintain the dequeued elements in stack to get the reverse order.

```
def levelOrderTraversalInReverse(root):
        if root is None:  return 0
        q = Queue()
        s = Stack()
        q.enQueue( root )
        node = None
        count = 0
        while not q.isEmpty():
            node = q.deQueue()          # deQueue FIFO
            if node.left is not None:
                q.enQueue( node.left )
            if node.right is not None:
                q.enQueue( node.right )
            s.push(node)
        while(not s.isEmpty()):
            print (s.pop().data)
```

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

Problem-9 Give an algorithm for deleting the tree.

Solution:



To delete a tree, we must traverse all the nodes of the tree and delete them one by one. So which traversal should we use: Inorder, Preorder, Postorder or Level order Traversal?

Before deleting the parent node we should delete its children nodes first. We can use postorder traversal as it does the work without storing anything. We can delete tree with other traversals also with extra space complexity. For the following, tree nodes are deleted in order – 4, 5, 2, 3, 1.

```
def deleteBinaryTree(root):
        if(root == None) :
             return
        deleteBinaryTree(root.left)
        deleteBinaryTree(root.right)
        del root
```

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

Problem-10 Give an algorithm for finding the height (or depth) of the binary tree.

Solution: Recursively calculate height of left and right subtrees of a node and assign height to the node as max of the heights of two children plus 1. This is similar to $PreOrder$ tree traversal (and DFS of Graph algorithms).

```
def maxDepth(root):
        if root == None:
            return 0
        return max(maxDepth(root.left),maxDepth(root.right))+1
```

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

Problem-11 Can we solve Problem-10 without recursion?

Solution: Yes, using level order traversal. This is similar to BFS of Graph algorithms. End of level is identified with None.

```
def maxDepth(root):
        if root == None:
            return 0
        q = []
        q.append([root, 1])
        temp = 0
        while len(q) != 0:
            node, depth = q.pop()
            depth = max(temp, dep)
            if node.left != None:
                    q.append([node.left, depth + 1])
            if node.right != None:
                    q.append([node.right, depth + 1])
        return temp
```

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

Problem-12 Give an algorithm for finding the deepest node of the binary tree.

Solution:

```
def deepestNode(root):
        if root is None:
             return 0
        q = Queue()
        q.enQueue( root )
        node = None
        while not q.isEmpty():
                node = q.deQueue()  # deQueue FIFO
                if node.left is not None:
                        q.enQueue( node.left )
                if node.right is not None:
                        q.enQueue( node.right )
        return node.data
```

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

Problem-13 Give an algorithm for deleting an element (assuming data is given) from binary tree.

Solution: The deletion of a node in binary tree can be implemented as

- Starting at root, find the node which we want to delete.
- Find the deepest node in the tree.
- Replace the deepest node's data with node to be deleted.
- Then delete the deepest node.

Problem-14 Give an algorithm for finding the number of leaves in the binary tree without using recursion.

Solution: The set of nodes whose both left and right children are None are called leaf nodes.

```
def leavesInBinaryTreeUsingLevelOrder(root):
        if root is None:
             return 0
        q = Queue()
        q.enQueue( root )
        node = None
        count = 0
        while not q.isEmpty():
                node = q.deQueue()  # deQueue FIFO
                if node.left is None and node.right is None:
                        count += 1
                else:
                        if node.left is not None:
```

```
                        q.enQueue( node.left )
                if node.right is not None:
                        q.enQueue( node.right )
    return count
```

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

Problem-15 Give an algorithm for finding the number of full nodes in the binary tree without using recursion.

Solution: The set of all nodes with both left and right children are called full nodes.

```
def numberOfFullNodesInBinaryTreeUsingLevelOrder(root):
    if root is None:
        return 0
    q = Queue()
    q.enQueue( root )
    node = None
    count = 0
    while not q.isEmpty():
            node = q.deQueue()  # deQueue FIFO
            if node.left is not None and node.right is not None:
                    count += 1
            if node.left is not None:
                    q.enQueue( node.left )
            if node.right is not None:
                    q.enQueue( node.right )
    return count
```

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

Problem-16 Give an algorithm for finding the number of half nodes (nodes with only one child) in the binary tree without using recursion.

Solution: The set of all nodes with either left or right child (but not both) are called half nodes.

```
def numberOfHalfNodesInBinaryTreeUsingLevelOrder(root):
    if root is None:
        return 0
    q = Queue()
    q.enQueue( root )
    node = None
    count = 0
    while not q.isEmpty():
            node = q.deQueue()  # deQueue FIFO
            if (node.left is None and node.right is not None) or (node.left is not None and node.right is None):
                    count += 1
            if node.left is not None:
                    q.enQueue( node.left )
            if node.right is not None:
                    q.enQueue( node.right )
    return count
```

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

Problem-17 Given two binary trees, return true if they are structurally identical.

Solution:

Algorithm:

- If both trees are None then return true.
- If both trees are not None, then compare data and recursively check left and right subtree structures.

```
# Return true if they are structurally identical.
def areStructurullySameTrees(root1, root2):
    if (not root1.left) and not (root1.right) and (not root2.left) and not (root2.right) and root1.data == root2.data:
            return True
    if (root1.data != root2.data) or (root1.left and not root2.left) or \
            (not root1.left and root2.left) or (root1.right and not root2.right) or (not root1.right and root2.right):
            return False
    left = areStructurullySameTrees(root1.left, root2.left) if root1.left and root2.left else True
    right = areStructurullySameTrees(root1.right, root2.right) if root1.right and root2.right else True
    return left and right
```

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

Problem-18 Give an algorithm for finding the diameter of the binary tree. The diameter of a tree (sometimes called the *width*) is the number of nodes on the longest path between two leaves in the tree.

Solution: To find the diameter of a tree, first calculate the diameter of left subtree and right subtrees recursively. Among these two values, we need to send maximum value along with current level $(+1)$.

```
    ptr = 0
    def diameter(root):
            global ptr
            if(not root) :
                    return 0
            left = diameter(root.left)
            right = diameter(root.right)
            if(left + right > ptr):
                ptr = left + right
            return max(left, right)+1
    #Alternative Coding
    def diameter(root):
            if (root == None):
                    return 0
            lHeight = height(root.eft)
            rHeight = height(root.right)
            lDiameter = diameter(root.left)
            rDiameter = diameter(root.right)
            return max(lHeight + rHeight + 1, max(lDiameter, rDiameter))
    # The function Compute the "height" of a tree. Height is the number of nodes along
    # the longest path from the root node down to the farthest leaf node.
    def height(root):
            if (root == None) :
                    return 0
```

There is another solution and the complexity is $O(n)$. The main idea of this approach is that the node stores its left child's and right child's maximum diameter if the node's child is the "root", therefore, there is no need to recursively call the height method. The drawback is we need to add two extra variables in the node class.

```
    def findMaxLen(root):
            nMaxLen = 0
            if (root == None): return 0
            if (root.left == None):
                    root.nMaxLeft = 0
            if (root.right == None):
                    root.nMaxRight = 0
            if (root.left != None):
                    findMaxLen(root.left)
            if (root.right != None):
                    findMaxLen(root.right)
            if (root.left != None):
                    nTempMaxLen = 0
                    nTempMaxLen = max(root.left.nMaxLeft, root.left.nMaxRight)
                    root.nMaxLeft = nTempMaxLen + 1
            if (root.right != None):
                    nTempMaxLen = 0
                    nTempMaxLen = max(root.right.nMaxLeft, root.right.nMaxRight)
                    root.nMaxRight = nTempMaxLen + 1
            if (root.nMaxLeft + root.nMaxRight > nMaxLen):
                    nMaxLen = root.nMaxLeft + root.nMaxRight
            return nMaxLen
```

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

Problem-19 Give an algorithm for finding the level that has the maximum sum in the binary tree.

Solution: The logic is very much similar to finding the number of levels. The only change is, we need to keep track of the sums as well.

```
    def findLevelWithMaxSum(root):
            if root is None:
                    return 0
            q = Queue()
            q.enQueue( root )
            q.enQueue( None )
            node = None
            level = maxLevel= currentSum = maxSum = 0
```

```
        while not q.isEmpty():
                node = q.deQueue()              # deQueue FIFO
                # If the current level is completed then compare sums
                if(node == None):
                        if(currentSum> maxSum):
                                maxSum = currentSum
                                maxLevel = level
                        currentSum = 0
                        #place the indicator for end of next level at the end of queue
                        if not q.isEmpty():
                                q.enQueue( None )
                                level += 1
                else:
                        currentSum += node.data
                        if node.left is not None:
                                q.enQueue( node.left )
                        if node.right is not None:
                                q.enQueue( node.right )
        return maxLevel
```

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

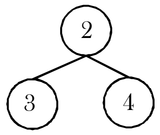
Problem-20 Given a binary tree, print out all its root-to-leaf paths.

Solution: Refer to comments in functions.

```
   def pathsAppender(root, path, paths):
        if not root: return 0
        path.append(root.data)
        paths.append(path)
        pathsAppender(root.left, path+[root.data], paths)
        pathsAppender(root.right, path+[root.data], paths) # make sure it can be executed!
   def pathsFinder(root):
      paths = []
      pathsAppender(root, [], paths)
      print ('paths:', paths)
```

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

Problem-21 Given a binary tree containing digits from 0-9 only, each root-to-leaf path could represent a number. An example is the root-to-leaf path 1->2->3 which represents the number 123. Find the total sum of all root-to-leaf numbers. For example,

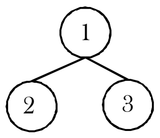


The root-to-leaf path 1->2 represents the number 23.
The root-to-leaf path 1->3 represents the number 24.
Return the sum = 23 + 24 = 47.

Solution:

```
   def sumNumbers(self, root):
        if not root:
                return 0
        current=0
        sum=[0]
        self.calSum(root, current, sum)
        return sum[0]
   def calSum(self, root, current, sum):
        if not root:
                return
        current=current*10+root.data
        if not root.left and not root.right:
                sum[0]+=current
                return
        self.calSum(root.left, current, sum)
        self.calSum(root.right,current, sum)
```

Problem-22 Given a binary tree, find the maximum path sum. The path may start and end at any node in the tree. For example: Given the below binary tree,



Solution:

```
def treeMaximumSumPath(node, isLeft=True, Lpath={}, Rpath={}):
    if isLeft:
        # left sub-tree
        if not node.left:
            Lpath[node.id] = 0
            return 0
        else:
            Lpath[node.id] = node.data + max(treeMaximumSumPath(node.left, True, Lpath, Rpath), \
                                                treeMaximumSumPath(node.left, False, Lpath, Rpath))
            return Lpath[node.id]
    else:
        # right sub-tree
        if not node.right:
            Rpath[node.id] = 0
            return 0
        else:
            Rpath[node.id] = node.data + max( treeMaximumSumPath(node.right, True, Lpath, Rpath), \
                                                treeMaximumSumPath(node.right, False, Lpath, Rpath))
            return Rpath[node.id]
def maxSumPath(root):
    Lpath = {}
    Rpath = {}
    treeMaximumSumPath(root, True, Lpath, Rpath)
    treeMaximumSumPath(root, False, Lpath, Rpath)
    print ('Left-path:', Lpath)
    print ('Right-path:', Rpath)
    path2sum = dict((i, Lpath[i]+Rpath[i]) for i in Lpath.keys())
    i = max(path2sum, key=path2sum.get)
    print ('The path going through node', i, 'with max sum', path2sum[i])
    return path2sum[i]
```

Problem-23 Give an algorithm for checking the existence of path with given sum. That means, given a sum, check whether there exists a path from root to any of the nodes.

Solution: For this problem, the strategy is: subtract the node value from the sum before calling its children recursively, and check to see if the sum is 0 when we run out of tree.

```
def pathFinder(root, val, path, paths):
    if not root:
        return False
    if not root.left and not root.right:
        if root.data == val:
            path.append(root.data)
            paths.append(path)
            return True
        else:
            return False
    left = pathFinder(root.left, val-root.data, path+[root.data], paths)
    right = pathFinder(root.right, val-root.data, path+[root.data], paths) # make sure it can be executed!
    return left or right
def hasPathWithSum(root, val):
    paths = []
    pathFinder(root, val, [], paths)
    print ('sum:', val)
    print ('paths:', paths)
```

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

Problem-24 Give an algorithm for finding the sum of all elements in binary tree.

Solution: Recursively, call left subtree sum, right subtree sum and add their values to current nodes data.

```
    def sumInBinaryTreeRecursive(root):
        if(root == None) :
            return 0
        return root.data+sumInBinaryTreeRecursive(root.left) + sumInBinaryTreeRecursive(root.right)
```

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

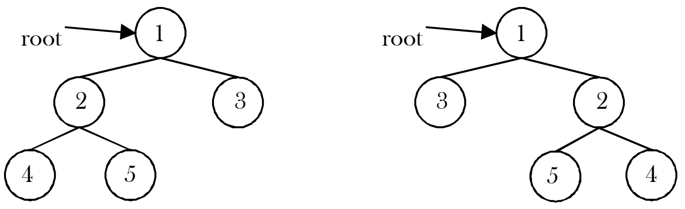
Problem-25 Can we solve Problem-24 without recursion?

Solution: We can use level order traversal with simple change. Every time after deleting an element from queue, add the node's data value to sum variable.

```
    def sumInBinaryTreeLevelOrder(root):
        if root is None:  return 0
        q = Queue()
        q.enQueue( root )
        node = None
        sum = 0
        while not q.isEmpty():
           node = q.deQueue()            # deQueue FIFO
           sum += node.data
           if node.left is not None:
                 q.enQueue( node.left )
           if node.right is not None:
                 q.enQueue( node.right )
        return sum
```

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

Problem-26 Give an algorithm for converting a tree to its mirror. Mirror of a tree is another tree with left and right children of all non-leaf nodes interchanged. The trees below are mirrors to each other.



Solution:

```
    def mirrorOfBinaryTree(root):
        if(root != None):
          mirrorOfBinaryTree(root.left)
          mirrorOfBinaryTree(root.right)
          # swap the pointers in this node
          temp  = root.left
          root.left  = root.right
          root.right = temp
        return root
```

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

Problem-27 Given two trees, give an algorithm for checking whether they are mirrors of each other.

Solution:

```
    def areMirrors(root1, root2):
        if(root1 == None and root2 == None):
          return 1
        if(root1 == None or root2 == None):
          return 0
        if(root1.data != root2.data):
          return 0
        else:
          return areMirrors(root1.left, root2.right) and areMirrors(root1.right, root2.left)
```

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

Problem-28 Give an algorithm for finding LCA (Least Common Ancestor) of two nodes in a Binary Tree.

Solution:

```
    def lca(root, alpha, beta):
      if not root:
            return None
      if root.data == alpha or root.data == beta:
            return root
```

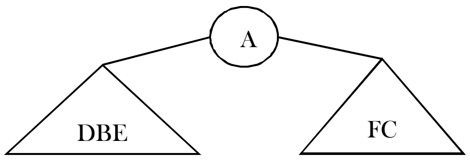
```
        left = lca(root.left, alpha, beta)
        right = lca(root.right, alpha, beta)
        if left and right:
            # alpha & beta are on both sides
            return root
        else:
            # EITHER alpha/beta is on one side OR alpha/beta is not in L&R subtrees
            return left if left else right
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$ for recursion.

Problem-29 Give an algorithm for constructing binary tree from given Inorder and Preorder traversals.

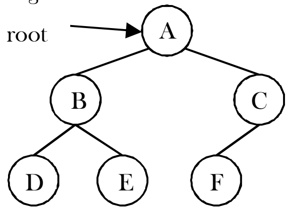
Solution: Let us consider the traversals below:

```
Inorder sequence:   D B E A F C
Preorder sequence: A B D E C F
```



In a Preorder sequence, leftmost element denotes the root of the tree. So we know 'A' is the root for given sequences. By searching 'A' in Inorder sequence we can find out all elements on the left side of 'A', which come under the left subtree, and elements on the right side of 'A', which come under the right subtree. So we get the structure as seen below.

We recursively follow the above steps and get the following tree.



Algorithm: BuildTree()

1 Select an element from $Preorder$. Increment a $Preorder$ index variable ($preOrderIndex$ in code below) to pick next element in next recursive call.
2 Create a new tree node ($newNode$) from heap with the data as selected element.
3 Find the selected element's index in Inorder. Let the index be $inOrderIndex$.
4 Call BuildBinaryTree for elements before $inOrderIndex$ and make the built tree as left subtree of $newNode$.
5 Call BuildBinaryTree for elements after $inOrderIndex$ and make the built tree as right subtree of $newNode$.
6 return $newNode$.

```
class TreeNode:
    def __init__(self, data):
        self.val = data
        self.left = None
        self.right = None
class Solution:
    def buildTree(self, preorder, inorder):
            if not inorder:
                    return None # inorder is empty
                    root = TreeNode(preorder[0])
                    rootPos = inorder.index(preorder[0])
                    root.left = self.buildTree(preorder[1 : 1 + rootPos], inorder[ : rootPos])
                    root.right = self.buildTree(preorder[rootPos + 1 : ], inorder[rootPos + 1 : ])
          return root
# Alternative coding
class Solution2:
    def buildTree(self, preorder, inorder):
        return self.buildTreeRecursive(preorder, inorder, 0, 0, len(preorder))
    def buildTreeRecursive(self, preorder, inorder, indPre, indIn, element):
        if element==0:
            return None
        solution = TreeNode(preorder[indPre])
        numElementsLeftSubtree = 0
        for i in range(indIn, indIn+element):
            if inorder[i] == preorder[indPre]:
```

```
            break
          numElementsLeftSubtree += 1
      solution.left = self.buildTreeRecursive(preorder, inorder, indPre+1, indIn, numElementsLeftSubtree)
      solution.right = self.buildTreeRecursive(preorder, inorder, indPre+numElementsLeftSubtree+1,\
              indIn+numElementsLeftSubtree+1, element-1-numElementsLeftSubtree)
      return solution
```

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

Problem-30 If we are given two traversal sequences, can we construct the binary tree uniquely?

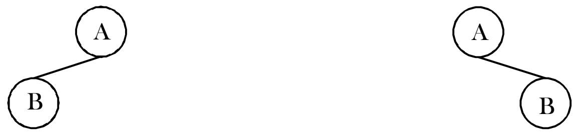
Solution: It depends on what traversals are given. If one of the traversal methods is *Inorder* then the tree can be constructed uniquely, otherwise not. Therefore, the following combinations can uniquely identify a tree:

- Inorder and Preorder
- Inorder and Postorder
- Inorder and Level-order

The following combinations do not uniquely identify a tree.

- Postorder and Preorder
- Preorder and Level-order
- Postorder and Level-order

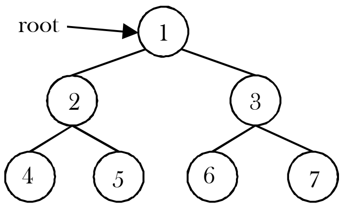
For example, Preorder, Level-order and Postorder traversals are the same for the above trees:



Preorder Traversal = AB Postorder Traversal = BA Level-order Traversal = AB

So, even if three of them (PreOrder, Level-Order and PostOrder) are given, the tree cannot be constructed uniquely.

Problem-31 Give an algorithm for printing all the ancestors of a node in a Binary tree. For the tree below, for 7 the ancestors are 1 3 7.

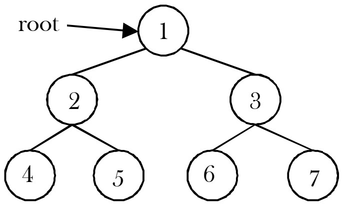


Solution: Apart from the Depth First Search of this tree, we can use the following recursive way to print the ancestors.

```
def printAllAncestors(root, node):
        if(root == None):
                return 0
        if(root.left == node or root.right == node or  printAllAncestors(root.left, node) or printAllAncestors(root.right, node)):
                print(root.data)
                return 1
        return 0
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$ for recursion.

Problem-32 **Zigzag Tree Traversal:** Give an algorithm to traverse a binary tree in Zigzag order. For example, the output for the tree below should be: 1 3 2 4 5 6 7



Solution: This problem can be solved easily using two stacks. Assume the two stacks are: *currentLevel* and *nextLevel*. We would also need a variable to keep track of the current level order (whether it is left to right or right to left).

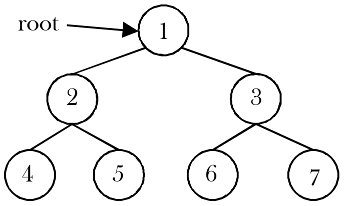
We pop from *currentLevel* stack and print the node's value. Whenever the current level order is from left to right, push the node's left child, then its right child, to stack *nextLevel*. Since a stack is a Last In First Out (*LIFO*) structure, the next time that nodes are popped off nextLevel, it will be in the reverse order.

On the other hand, when the current level order is from right to left, we would push the node's right child first, then its left child. Finally, don't forget to swap those two stacks at the end of each level (*i.e.*, when *currentLevel* is empty).

```
def zigZagTraversal(self, root):
        result = []
        currentLevel =[]
        if root != None:
            currentLevel.append(root)
        leftToRight = True
        while len(currentLevel)>0:
            levelresult = []
            nextLevel = []
            while len(currentLevel)>0:
                node = currentLevel.pop()
                levelresult.append(node.val)
                if leftToRight:
                    if node.left != None:
                        nextLevel.append(node.left)
                    if node.right != None:
                        nextLevel.append(node.right)
                else:
                    if node.right != None:
                        nextLevel.append(node.right)
                    if node.left != None:
                        nextLevel.append(node.left)
            currentLevel = nextLevel
            result.append(levelresult)
            leftToRight = not leftToRight
        return result
```

Time Complexity: $O(n)$. Space Complexity: Space for two stacks = $O(n) + O(n) = O(n)$.

Problem-33 Give an algorithm for finding the vertical sum of a binary tree. For example,
The tree has 5 vertical lines
Vertical-1: nodes-4 => vertical sum is 4
Vertical-2: nodes-2 => vertical sum is 2
Vertical-3: nodes-1,5,6 => vertical sum is $1 + 5 + 6 = 12$
Vertical-4: nodes-3 => vertical sum is 3
Vertical-5: nodes-7 => vertical sum is 7
We need to output: 4 2 12 3 7

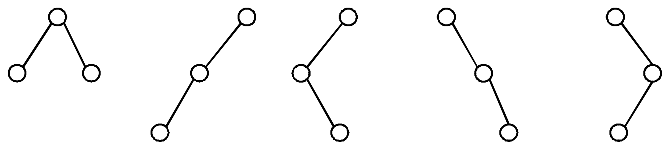


Solution: We can do an inorder traversal and hash the column. We call $vertical\_sum\_in\_binary\_tree(root, 0)$ which means the root is at column 0. While doing the traversal, hash the column and increase its value by $root \rightarrow data$.

```
hashTable = {}
def verticalSumInBinaryTree(root, column):
        if not root:
            return
        if not column in hashTable:
            hashTable[column] = 0
        hashTable[column] = hashTable[column] + root.data
        verticalSumInBinaryTree(root.left, column - 1)
        verticalSumInBinaryTree(root.right, column + 1)
verticalSumInBinaryTree(root, 0)
print (hashTable)
```

Problem-34 How many different binary trees are possible with n nodes?

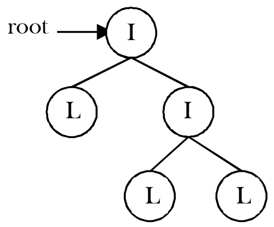
Solution: For example, consider a tree with 3 nodes ($n = 3$). It will have the maximum combination of 5 different (i.e., $2^3 - 3 = 5$) trees.



In general, if there are n nodes, there exist $2^n - n$ different trees.

Problem-35 Given a tree with a special property where leaves are represented with 'L' and internal node with 'I'. Also, assume that each node has either 0 or 2 children. Given preorder traversal of this tree, construct the tree.

Example: Given preorder string => ILILL



Solution: First, we should see how preorder traversal is arranged. Pre-order traversal means first put root node, then pre-order traversal of left subtree and then pre-order traversal of right subtree. In a normal scenario, it's not possible to detect where left subtree ends and right subtree starts using only pre-order traversal. Since every node has either 2 children or no child, we can surely say that if a node exists then its sibling also exists. So every time when we are computing a subtree, we need to compute its sibling subtree as well.

Secondly, whenever we get 'L' in the input string, that is a leaf and we can stop for a particular subtree at that point. After this 'L' node (left child of its parent 'L'), its sibling starts. If 'L' node is right child of its parent, then we need to go up in the hierarchy to find the next subtree to compute.

Keeping the above invariant in mind, we can easily determine when a subtree ends and the next one starts. It means that we can give any start node to our method and it can easily complete the subtree it generates going outside of its nodes. We just need to take care of passing the correct start nodes to different sub-trees.

```
i = 0
def buildTreeFromPreOrder(A):
        global i
        if(A == None or i >= len(A)):                # Boundary Condition
                return None
        newNode = BinaryTree(A[i])
        newNode.data = A[i]
        newNode.left = newNode.right = None
        if(A[i] == "L"):                             # On reaching leaf node, return
                return newNode
        i += 1                                       # Populate left sub tree
        newNode.left = buildTreeFromPreOrder(A)
        i += 1                                       # Populate right sub tree
        newNode.right = buildTreeFromPreOrder(A)
        return newNode
root = buildTreeFromPreOrder(["I","I","L","I","L","L","I","L","L"])
postorderRecursive(root)
```

Time Complexity: $O(n)$.

Problem-36 Given a binary tree with three pointers (left, right and nextSibling), give an algorithm for filling the *nextSibling* pointers assuming they are None initially.

Solution: We can use simple queue (similar to the solution of Problem-11). Let us assume that the structure of binary tree is:

```
def fillNextSiblingsWithLevelOrderTraversal(root):
        if root is None: return 0
        q = Queue()
        q.enQueue( root )
        node = None
        count = 0
        while not q.isEmpty():
                node = q.deQueue()              # deQueue FIFO
                node.nextSibling = q.queueFront()
                if node.left is not None:
                        q.enQueue( node.left )
                if node.right is not None:
                        q.enQueue( node.right )
```

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

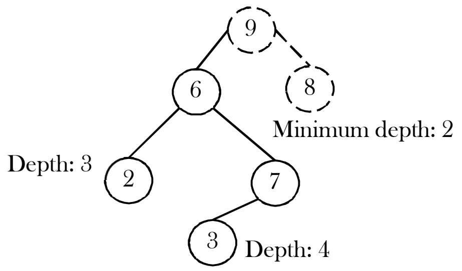
Problem-37 Is there any other way of solving Problem-36?

Solution: The trick is to re-use the populated *nextSibling* pointers. As mentioned earlier, we just need one more step for it to work. Before we pass the *left* and *right* to the recursion function itself, we connect the right child's *nextSibling* to the current node's nextSibling left child. In order for this to work, the current node *nextSibling* pointer must be populated, which is true in this case.

```
def fillNextSiblings(root):
        if (root == None): return
        if root.left:
             root.left.nextSibling = root.right
        if root.right:
             if root.nextSibling:
                     root.right.nextSibling = root.nextSibling.left
             else:
                     root.right.nextSibling = None
        fillNextSiblings(root.left)
        fillNextSiblings(root.right)
```

Time Complexity: $O(n)$.

Problem-38 **Minimum depth of a binary tree:** Given a binary tree, find its minimum depth. The minimum depth of a binary tree is the number of nodes along the shortest path from the root node down to the nearest leaf node. For example, minimum depth of the following binary tree is 3.



Solution: The algorithm is similar to the algorithm of finding depth (or height) of a binary tree, except here we are finding minimum depth. One simplest approach to solve this problem would be by using recursion. But the question is when do we stop it? We stop the recursive calls when it is a leaf node or *None*.

Algorithm: Let *root* be the pointer to the root node of a subtree.

- If the *root* is equal to *None*, then the minimum depth of the binary tree would be 0.
- If the *root* is a leaf node, then the minimum depth of the binary tree would be 1.
- If the *root* is not a leaf node and if left subtree of the *root* is None, then find the minimum depth in the right subtree. Otherwise, find the minimum depth in the left subtree.
- If the *root* is not a leaf node and both left subtree and right subtree of the *root* are not *None*, then recursively find the minimum depth of left and right subtree. Let it be *leftSubtreeMinDepth* and *rightSubtreeMinDepth* respectively.
- To get the minimum height of the binary tree rooted at root, we will take minimum of *leftSubtreeMinDepth* and *rightSubtreeMinDepth* and 1 for the *root* node.

```
class Solution:
    def minimumDepth(self, root):
        # If root (tree) is empty, minimum depth would be 0
        if root is None:
            return 0
        # If root is a leaf node, minimum depth would be 1
        if root.left is None and root.right is None:
            return 1
        # If left subtree is None, find minimum depth in right subtree
        if root.left is None:
            return self.minimumDepth(root.right)+1
        # If right subtree is None, find minimum depth in left subtree
        if root.right is None:
            return self.minimumDepth(root.left) +1
        # Get the minimum depths of left and right subtrees and add 1 for current level.
        return min(self.minimumDepth(root.left), self.minimumDepth(root.right)) + 1
# Approach two
class Solution:
    def minimumDepth(self, root):
        if root == None:
            return 0
        if root.left == None or root.right == None:
            return self.minimumDepth(root.left) + self.minimumDepth(root.right)+1
        return min(self.minimumDepth(root.right), self.minimumDepth(root.left))+1
```

Time complexity: O(n), as we are doing pre order traversal of tree only once. Space complexity: O(n), for recursive stack space.

Solution with level order traversal: The above recursive approach may end up with complete traversal of the binary tree even when the minimum depth leaf is close to the root node. A better approach is to use level order traversal. In this algorithm, we will traverse the binary tree by keeping track of the levels of the node and closest leaf node found till now.

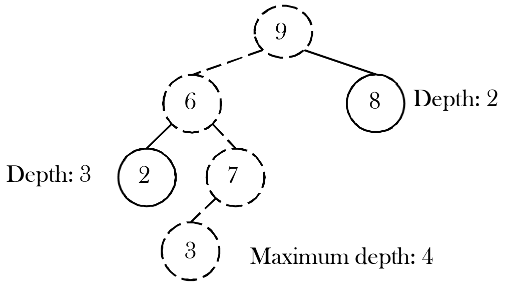
Algorithm: Let *root* be the pointer to the root node of a subtree at level L.

- If root is equal to *None*, then the minimum depth of the binary tree would be 0.
- If root is a leaf node, then check if its level(L) is less than the level of closest leaf node found till now. If yes, then update closest leaf node to current node and return.
- Recursively traverse left and right subtree of node at level L + 1.

```
class Solution:
    def minimumDepth(self, root):
        if root is None: return 0
        queue = []
        queue.append((root, 1))
        while queue:
            current, depth = queue.pop(0)
            if current.left is None and current.right is None:
                return depth
            if current.left:
                queue.append((current.left, depth+1))
            if current.right:
                queue.append((current.right, depth+1))
```

Time complexity: O(n), as we are doing lever order traversal of the tree only once. Space complexity: O(n), for queue.

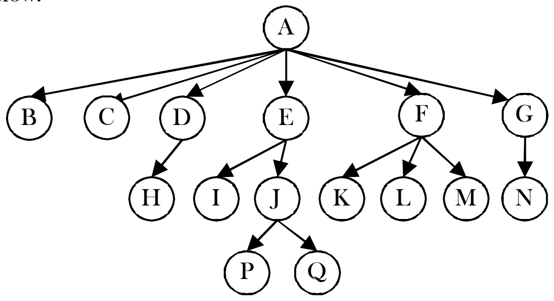
Similar question: Maximum depth of a binary tree: Given a binary tree, find its maximum depth. The maximum depth of a binary tree is the number of nodes along the shortest path from the root node down to the farthest leaf node. For example, maximum depth of following binary tree is 4. Careful observation tells us that it is exactly same as finding the depth (or height) of the tree.



6.7 Generic Trees (N-ary Trees)

In the previous section we discussed binary trees where each node can have a maximum of two children and these are represented easily with two pointers. But suppose if we have a tree with many children at every node and also if we do not know how many children a node can have, how do we represent them?

For example, consider the tree shown below.



How do we represent the tree?

In the above tree, there are nodes with 6 children, with 3 children, with 2 children, with 1 child, and with zero children (leaves). To present this tree we have to consider the worst case (6 children) and allocate that many child pointers for each node. Based on this, the node representation can be given as:

```
#Node of a Generic Tree
class TreeNode:
  def __init__(self, data=None, next=None):          #constructor
```

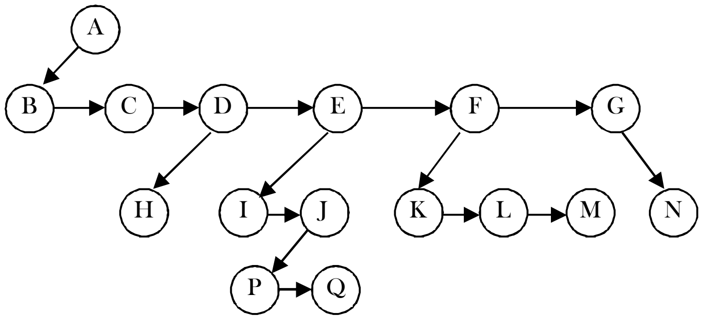
```
        self.data = data
        self.firstChild = None
        self.secondChild = None
        self.thirdChild = None
        self.fourthChild = None
        self.fifthChild = None
        self.sixthChild = None
```

Since we are not using all the pointers in all the cases, there is a lot of memory wastage. Another problem is that we do not know the number of children for each node in advance. In order to solve this problem we need a representation that minimizes the wastage and also accepts nodes with any number of children.

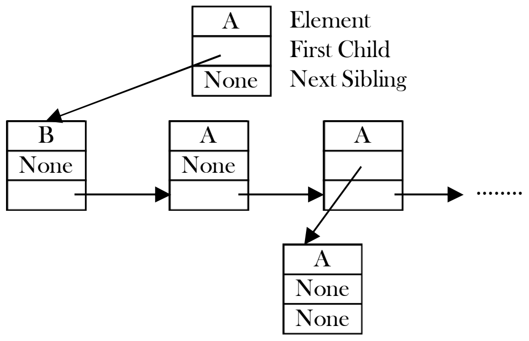
Representation of Generic Trees

Since our objective is to reach all nodes of the tree, a possible solution to this is as follows:

- At each node link children of same parent (siblings) from left to right.
- Remove the links from parent to all children except the first child.



What these above statements say is if we have a link between children then we do not need extra links from parent to all children. This is because we can traverse all the elements by starting at the first child of the parent. So if we have a link between parent and first child and also links between all children of same parent then it solves our problem. This representation is sometimes called first child/next sibling representation. First child/next sibling representation of the generic tree is shown above. The actual representation for this tree is:



Based on this discussion, the tree node declaration for general tree can be given as:

```
    # Node of a Generic Tree
    class TreeNode:
        def __init__(self, data=None, next=None):        #constructor
            self.data = data
            self.firstChild = None
            self.nextSibling = None
```

Note: Since we are able to convert any generic tree to binary representation, in practice we use binary trees. We can treat all generic trees with a first child/next sibling representation as binary trees.

Generic Trees: Problems & Solutions

Problem-39 Implement simple generic tree which allows us to add children and also prints the path from root to leaves (nodes without children) for every node.

Solution:

```
import string
class GenericTree:    # Generic n-ary tree node object
  """ Children are additive; no provision for deleting them.
    The birth order of children is recorded: 0 for the first child added, 1 for the second, and so on.
      GenericTree(parent, value=None)    Constructor
```

```
        parent          If this is the root node, None, otherwise the parent's GenericTree object.
        childList       List of children, zero or more GenericTree objects.
        value           Value passed to constructor; can be any type.
        birthOrder      If this is the root node, 0, otherwise the index of this child in the parent's .childList
        nChildren()     Returns the number of self's children.
        nthChild(n)     Returns the nth child; raises IndexError if n is not a valid child number.
        fullPath():     Returns path to self as a list of child numbers.
        nodeId():       Returns path to self as a NodeId.
    """
    def __init__ ( self, parent, value=None ):
        self.parent     =  parent
        self.value      =  value
        self.childList  =  []
        if  parent is None:
            self.birthOrder  =  0
        else:
            self.birthOrder  =  len(parent.childList)
            parent.childList.append ( self )
    def nChildren ( self ):
        return len(self.childList)
    def nthChild ( self, n ):
        return self.childList[n]
    def fullPath ( self ):
        result  =  []
        parent  =  self.parent
        kid     =  self
        while  parent:
            result.insert ( 0, kid.birthOrder )
            parent, kid  =  parent.parent, parent
        return result
    def nodeID ( self ):
        fullPath  =  self.fullPath()
        return nodeID( fullPath )
class NodeId:
    def __init__ ( self, path ):
        self.path  =  path
    def __str__ ( self ):
        L  =  map ( str, self.path )
        return string.join ( L, "/" )
    def find ( self, node ):
        return self.__reFind ( node, 0 )
    def __reFind ( self, node, i ):
        if  i >= len(self.path):
            return node.value       # We're there!
        else:
            childNo  =  self.path[i]
        try:
            child  =  node.nthChild ( childNo )
        except IndexError:
            return None
        return self.__reFind ( child, i+1 )
    def isOnPath ( self, node ):
        if  len(nodePath) > len(self.path):
            return 0        # Node is deeper than self.path
        for  i in range(len(nodePath)):
            if  nodePath[i] != self.path[i]:
                return 0    # Node is a different route than self.path
        return 1
```

Problem-40 Given a tree, give an algorithm for finding the sum of all the elements of the tree.

Solution: The solution is similar to what we have done for simple binary trees. That means, traverse the complete list and keep on adding the values. We can either use level order traversal or simple recursion.

```
    def findSum(root):
        if(root == None):  return 0
        return root.data + findSum(root.firstChild) + findSum(root.nextSibling)
```

Time Complexity: O(n). Space Complexity: O(1) (if we do not consider stack space), otherwise O(n).

Note: All problems which we have discussed for binary trees are applicable for generic trees also. Instead of left and right pointers we just need to use firstChild and nextSibling.

Problem-41 For a 4-ary tree (each node can contain maximum of 4 children), what is the maximum possible height with 100 nodes? Assume height of a single node is 0.

Solution: In 4-ary tree each node can contain 0 to 4 children, and to get maximum height, we need to keep only one child for each parent. With 100 nodes, the maximum possible height we can get is 99.

If we have a restriction that at least one node has 4 children, then we keep one node with 4 children and the remaining nodes with 1 child. In this case, the maximum possible height is 96. Similarly, with n nodes the maximum possible height is $n - 4$.

Problem-42 For a 4-ary tree (each node can contain maximum of 4 children), what is the minimum possible height with n nodes?

Solution: Similar to the above discussion, if we want to get minimum height, then we need to fill all nodes with maximum children (in this case 4). Now let's see the following table, which indicates the maximum number of nodes for a given height.

Height, h	Maximum Nodes at height, $h = 4^h$	Total Nodes height $h = \frac{4^{h+1}-1}{3}$
0	1	1
1	4	1+4
2	4 × 4	1+ 4 × 4
3	4 × 4 × 4	1+ 4 × 4 + 4 × 4 × 4

For a given height h the maximum possible nodes are: $\frac{4^{h+1}-1}{3}$. To get minimum height, take logarithm on both sides:

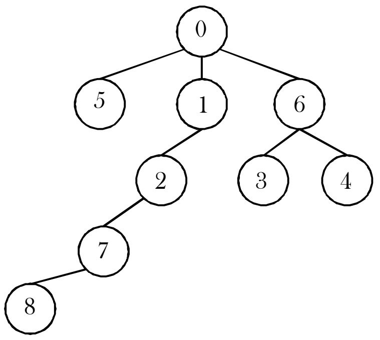
$$n = \frac{4^{h+1}-1}{3} \Longrightarrow 4^{h+1} = 3n + 1 \Longrightarrow (h+1)log4 = \log(3n+1) \Longrightarrow h + 1 = \log_4(3n+1) \Longrightarrow h = \log_4(3n+1) - 1$$

Problem-43 Given a parent array P, where $P[i]$ indicates the parent of i^{th} node in the tree (assume parent of root node is indicated with -1). Give an algorithm for finding the height or depth of the tree.

Solution: For example: if the P is

-1	0	1	6	6	0	0	2	7
0	1	2	3	4	5	6	7	8

Its corresponding tree is:



From the problem definition, the given array represents the parent array. That means, we need to consider the tree for that array and find the depth of the tree. The depth of this given tree is 4. If we carefully observe, we just need to start at every node and keep going to its parent until we reach -1 and also keep track of the maximum depth among all nodes.

```
def depthInGenericTree(P):
        maxDepth =-1
        currentDepth =-1
        for i in range (0, len(P)):
                currentDepth = 0
                j = i
                while(P[j] != -1):
                        currentDepth += 1
                        j = P[j]
                if(currentDepth > maxDepth):
                        maxDepth = currentDepth
        return maxDepth
P=[-1, 0, 1, 6, 6, 0, 0, 2, 7]
print ("Depth of given Generic Tree is:", depthInGenericTree(P))
```

Time Complexity: O(n^2). For skew trees we will be re-calculating the same values. Space Complexity: O(1).

Note: We can optimize the code by storing the previous calculated nodes' depth in some hash table or other array. This reduces the time complexity but uses extra space.

Problem-44 Given a node in the generic tree, give an algorithm for counting the number of siblings for that node.

Solution: Since tree is represented with the first child/next sibling method, the tree structure can be given as:

```
class GenericTreeNode:
        def __init__(self, data):
            self.data = data                #root node
            self.firstChild = None                  #left child
            self.nextSibling = None         #right child
```

For a given node in the tree, we just need to traverse all its next siblings.

```
def siblingsCount(current):
        count = 0
        while(current):
            count += 1
            current = current.nextSibling
        return count
```

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

With generic tree representation, we can count the siblings of a given node with code below.

```
def siblingsCount ( self ):
    if  parent is None: return 1
    else:
          return self.parent.nChildren
```

Problem-45 Given a node in the generic tree, give an algorithm for counting the number of children for that node.

Solution: With tree is represented as first child/next sibling method; for a given node in the tree, we just need to point to its first child and keep traversing all its next siblings.

```
def childrenCount(current):
        count = 0
        current = current.firstChild
        while(current):
            count += 1
            current = current.nextSibling
        return count
```

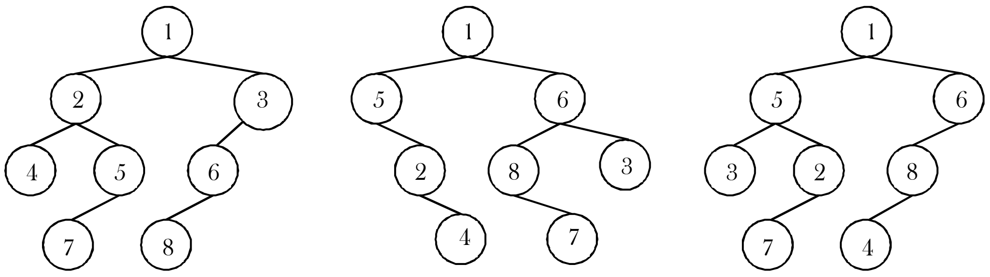
Time Complexity: O(n). Space Complexity: O(1).

With generic tree representation, we can count the children of a given node with code below.

```
def childrenCount ( self ):
    return len(self.childList)
```

Problem-46 Given two trees how do we check whether the trees are isomorphic to each other or not?

Solution: Two binary trees $root1$ and $root2$ are isomorphic if they have the same structure. The values of the nodes does not affect whether two trees are isomorphic or not. In the diagram below, the tree in the middle is not isomorphic to the other trees, but the tree on the right is isomorphic to the tree on the left.

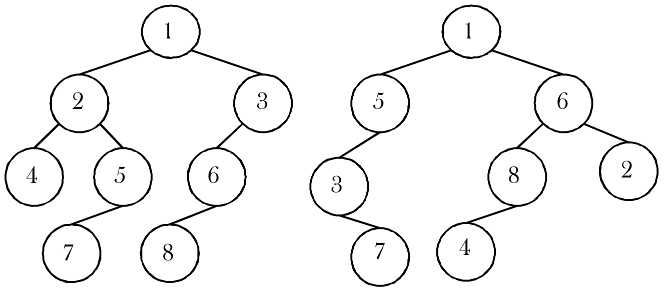


```
def isIsomorphic(root1, root2):
        if(not root1 and not root2):
            return 1
        if((not root1 and root2) or (root1 and not root2)):
            return 0
        return (isIsomorphic(root1.left, root2.left) and isIsomorphic(root1.right, root2.right))
```

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

Problem-47 Given two trees how do we check whether they are quasi-isomorphic to each other or not?

Solution:



Two trees $root1$ and $root2$ are quasi-isomorphic if $root1$ can be transformed into $root2$ by swapping the left and right children of some of the nodes of $root1$. Data in the nodes are not important in determining quasi-isomorphism; only the shape is important. The trees below are quasi-isomorphic because if the children of the nodes on the left are swapped, the tree on the right is obtained.

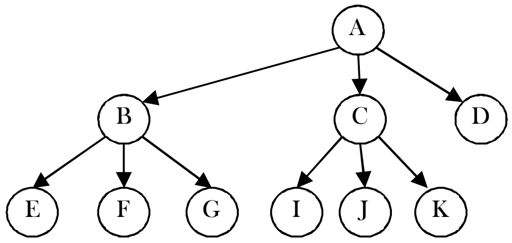
```
def quasiIsomorphic(root1, root2):
        if(not root1 and not root2): return 1
        if((not root1 and root2) or (root1 and not root2)):return 0
        return (quasiIsomorphic(root1.left, root2.left) and quasiIsomorphic(root1.right, root2.right)
                or quasiIsomorphic(root1.right, root2.left) and quasiIsomorphic(root1.left, root2.right))
```

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

Problem-48 A full k –ary tree is a tree where each node has either 0 or k children. Given an array which contains the preorder traversal of full k –ary tree, give an algorithm for constructing the full k –ary tree.

Solution: In k –ary tree, for a node at i^{th} position its children will be at $k * i + 1$ to $k * i + k$. For example, the example below is for full 3-ary tree.

As we have seen, in preorder traversal first left subtree is processed then followed by root node and right subtree. Because of this, to construct a full k-ary, we just need to keep on creating the nodes without bothering about the previous constructed nodes. We can use this trick to build the tree by using one global index.



The declaration for k-ary tree can be given as:

```
class KaryTreeNode:
    def __init__ ( self, k, data=None ):
        self.data      =  data
        self.childList  =  []

def buildKaryTree(A, k):
        n = len(A)
        if n <= 0:
                return None
        index = 0
        root = KaryTreeNode(None, A[0])
        if(not root):
                print("Memory Error")
                return
        Q = Queue()
        if(Q == None):
                return None
        Q.enQueue(root)
        while(not Q.isEmpty()):
                temp = Q.deQueue()
                for i in range(0,k):
                        index += 1
                        if index <  n:
                                temp.childList.insert(i,KaryTreeNode(None, A[index]))
                                Q.enQueue(temp.childList[i])
        return root
def preorderRecursive(kroot):
        if not kroot:
                return
        print (kroot.data)
        for node in kroot.childList:
                preorderRecursive(node)
A=[1,2,3,4,5,6,7,8,9,10,11,12,13]
kroot = buildKaryTree(A, 3)
preorderRecursive(kroot)
```

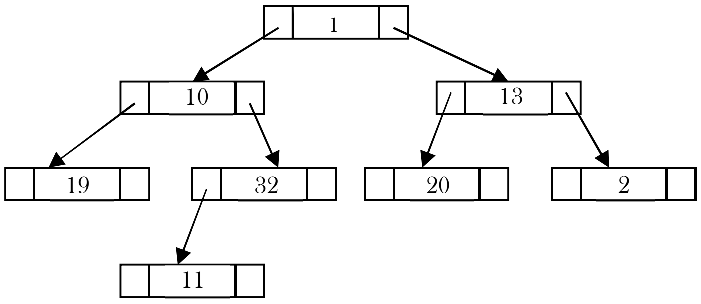
Time Complexity: $O(n)$, where n is the size of the pre-order array. This is because we are moving sequentially and not visiting the already constructed nodes.

6.8 Threaded Binary Tree Traversals (Stack or Queue-less Traversals)

In earlier sections we have seen that, *preorder, inorder, and postorder* binary tree traversals used stacks and *level order* traversals used queues as an auxiliary data structure. In this section we will discuss new traversal algorithms which do not need both stacks and queues. Such traversal algorithms are called *threaded binary tree traversals* or *stack/queue less traversals*.

Issues with Regular Binary Tree Traversals

- The storage space required for the stack and queue is large.
- The majority of pointers in any binary tree are None. For example, a binary tree with n nodes has $n + 1$ None pointers and these were wasted.



- It is difficult to find successor node (preorder, inorder and postorder successors) for a given node.

Motivation for Threaded Binary Trees

To solve these problems, one idea is to store some useful information in None pointers. If we observe the previous traversals carefully, stack/queue is required because we have to record the current position in order to move to the right subtree after processing the left subtree. If we store the useful information in None pointers, then we don't have to store such information in stack/queue.

The binary trees which store such information in None pointers are called *threaded binary trees*. From the above discussion, let us assume that we want to store some useful information in None pointers. The next question is what to store?

The common convention is to put predecessor/successor information. That means, if we are dealing with preorder traversals, then for a given node, None left pointer will contain preorder predecessor information and None right pointer will contain preorder successor information. These special pointers are called *threads*.

Classifying Threaded Binary Trees

The classification is based on whether we are storing useful information in both None pointers or only in one of them.

- If we store predecessor information in None left pointers only then we can call such binary trees *left threaded binary trees*.
- If we store successor information in None right pointers only then we can call such binary trees *right threaded binary trees*.
- If we store predecessor information in None left pointers and successor information in None right pointers, then we can call such binary trees *fully threaded binary trees* or simply *threaded binary trees*.

Note: For the remaining discussion we consider only (*fully*) *threaded binary trees*.

Types of Threaded Binary Trees

Based on above discussion we get three representations for threaded binary trees.

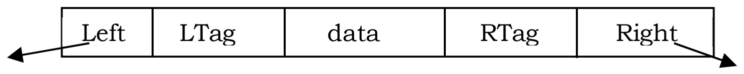
- *Preorder Threaded Binary Trees*: None left pointer will contain PreOrder predecessor information and None right pointer will contain PreOrder successor information.
- *Inorder Threaded Binary Trees*: None left pointer will contain InOrder predecessor information and None right pointer will contain InOrder successor information.

- *Postorder Threaded Binary Trees*: None left pointer will contain PostOrder predecessor information and None right pointer will contain PostOrder successor information.

Note: As the representations are similar, for the remaining discussion we will use InOrder threaded binary trees.

Threaded Binary Tree structure

Any program examining the tree must be able to differentiate between a regular *left/right* pointer and a *thread.* To do this, we use two additional fields in each node, giving us, for threaded trees, nodes of the following form:



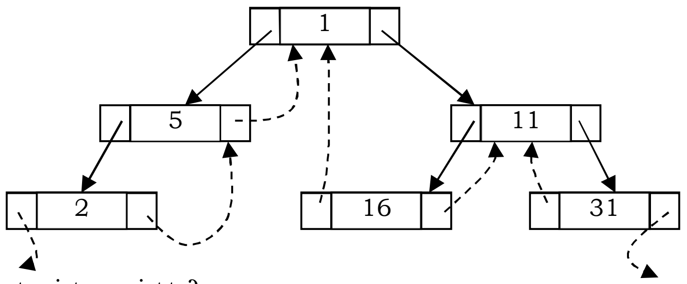
```
'''Threaded Binary Tree Class and its methods'''
class ThreadedBinaryTree:
    def __init__(self, data):
        self.data = data        #data
        self.left = None        #left child
        self.LTag = None
        self.right = None       #right child
        self.RTag = None
```

Difference between Binary Tree and Threaded Binary Tree Structures

	Regular Binary Trees	Threaded Binary Trees
if LTag == 0	None	left points to the in-order predecessor
if LTag == 1	left points to the left child	left points to left child
if RTag == 0	None	right points to the in-order successor
if RTag == 1	right points to the right child	right points to the right child

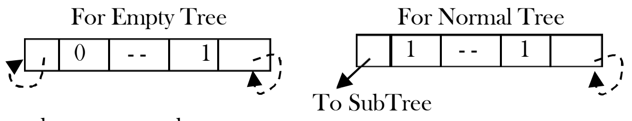
Note: Similarly, we can define preorder/postorder differences as well.

As an example, let us try representing a tree in inorder threaded binary tree form. The tree below shows what an inorder threaded binary tree will look like. The dotted arrows indicate the threads. If we observe, the left pointer of left most node (2) and right pointer of right most node (31) are hanging.

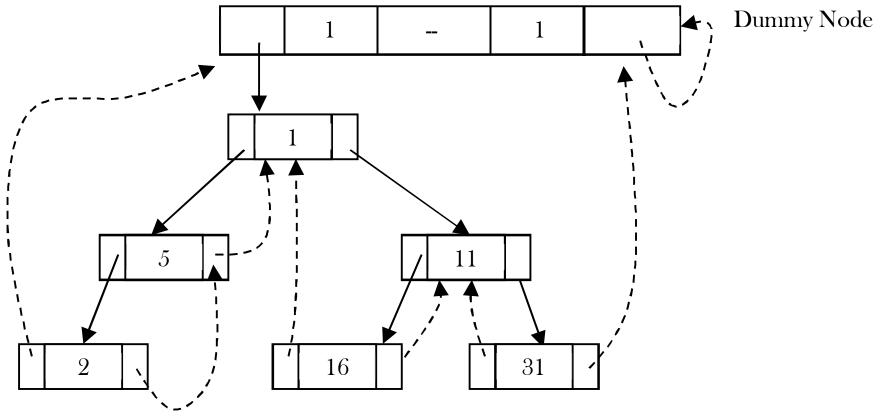


What should leftmost and rightmost pointers point to?

In the representation of a threaded binary tree, it is convenient to use a special node *Dummy* which is always present even for an empty tree. Note that right tag of Dummy node is 1 and its right child points to itself.



With this convention the above tree can be represented as:



Finding Inorder Successor in Inorder Threaded Binary Tree

To find inorder successor of a given node without using a stack, assume that the node for which we want to find the inorder successor is P.

Strategy: If P has a no right subtree, then return the right child of P. If P has right subtree, then return the left of the nearest node whose left subtree contains P.

```
def inorderSuccessor(P):
        if(P.RTag == 0):  return P.right
        else:
            Position = P.right
            while(Position.LTag == 1):
                    Position = Position.left
                    return Position
```

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

Inorder Traversal in Inorder Threaded Binary Tree

We can start with $dummy$ node and call inorderSuccessor() to visit each node until we reach $dummy$ node.

```
def inorderTraversal(root):
        P = inorderSuccessor(root)
        while(P != root):
                    P = inorderSuccessor(P)
                  print (P.data)
```

Alternative coding:

```
def inorderTraversal(root):
        P = root
        while (1):
            P = inorderSuccessor(P)
            if(P == root):
                return
            print (P.data)
```

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

Finding PreOrder Successor in InOrder Threaded Binary Tree

Strategy: If P has a left subtree, then return the left child of P. If P has no left subtree, then return the right child of the nearest node whose right subtree contains P.

```
def preorderSuccessor(P):
        if(P.LTag == 1):  return P.left
        else :
            Position = P
            while(Position.RTag == 0):
                        Position = Position.right
            return Position.right
```

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

PreOrder Traversal of InOrder Threaded Binary Tree

As in inorder traversal, start with $dummy$ node and call preorderSuccessor() to visit each node until we get $dummy$ node again.

```
def preorderTraversal(root):
        P = preorderSuccessor(root)
        while(P != root) :
                    P = preorderSuccessor(P)
                    print (P.data)
```

Alternative coding:

```
def preorderTraversal(root) :
        P = root
        while(1):
            P = preorderSuccessor(P)
            if(P == root): return
            print (P.data)
```

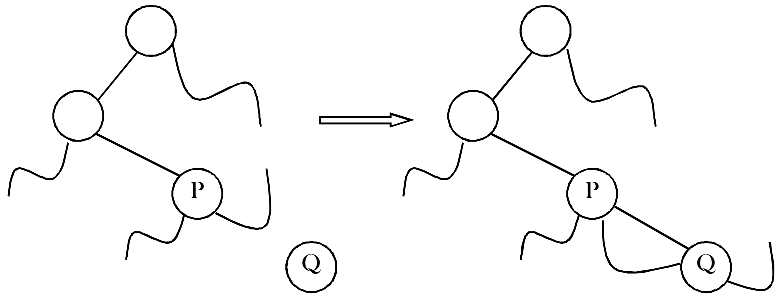
Time Complexity: O(n). Space Complexity: O(1).

Note: From the above discussion, it should be clear that inorder and preorder successor finding is easy with threaded binary trees. But finding postorder successor is very difficult if we do not use stack.

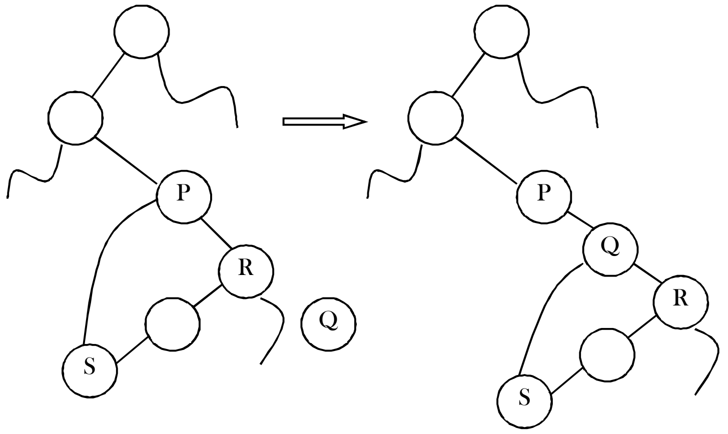
Insertion of Nodes in InOrder Threaded Binary Trees

For simplicity, let us assume that there are two nodes P and Q and we want to attach Q to right of P. For this we will have two cases.

- Node P does not have right child: In this case we just need to attach Q to P and change its left and right pointers.



- Node P has right child (say, R): In this case we need to traverse $R's$ left subtree and find the left most node and then update the left and right pointer of that node (as shown below).



```
def insertRightInInorderTBT(P, Q):
        Q.right = P.right
        Q.RTag = P.RTag
        Q.left = P
        Q.LTag = 0
        P.right = Q
        P.RTag = 1
        if(Q.RTag == 1) :                              #Case-2
                Temp = Q.right
                while(Temp.LTag):
                    Temp = Temp.left
                Temp.left = Q
```

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

Threaded Binary Trees: Problems & Solutions

Problem-49 For a given binary tree (not threaded) how do we find the preorder successor?

Solution: For solving this problem, we need to use an auxiliary stack S. On the first call, the parameter node is a pointer to the head of the tree, and thereafter its value is None. Since we are simply asking for the successor of the node we got the last time we called the function.

It is necessary that the contents of the stack S and the pointer P to the last node "visited" are preserved from one call of the function to the next; they are defined as static variables.

```
# pre-order successor for an unthreaded binary tree
def preorderSuccessor(node):
        S = Stack()
        if(node != None):
            P = node
        if(P.left != None):
            push(S,P)
            P = P.left
        else :
            while (P.right == None):
                P = pop(S)
                P = P.right
        return P
```

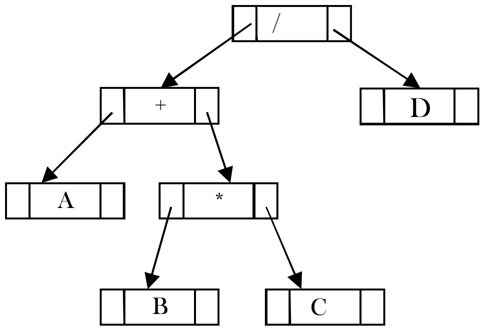
Problem-50 For a given binary tree (not threaded) how do we find the inorder successor?

Solution: Similar to the above discussion, we can find the inorder successor of a node as:

```
    # In-order successor for an unthreaded binary tree
    def inorderSuccessor(node):
        S = Stack()
        if(node != None):
            P = node
        if(P.right == None):
            P = pop(S)
        else :
            P = P.right
            while (P.left != None):
                push(S, P)
            P = P.left
        return P
```

6.9 Expression Trees

A tree representing an expression is called an *expression tree*. In expression trees, leaf nodes are operands and non-leaf nodes are operators. That means, an expression tree is a binary tree where internal nodes are operators and leaves are operands. An expression tree consists of binary expression. But for a u-nary operator, one subtree will be empty. The figure below shows a simple expression tree for (A + B * C) / D.

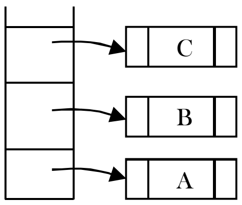


Algorithm for Building Expression Tree from Postfix Expression

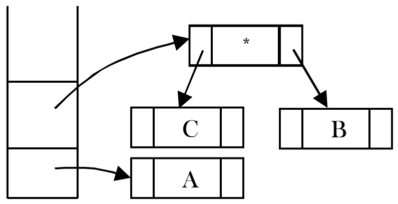
```
operatorPrecedence = { '(' : 0,  ')' : 0,  '+' : 1,  '-' : 1,  '*' : 2,  '/' : 2 }
def postfixConvert(infix):
    stack = []
    postfix = []
    for char in infix:
        if char not in operatorPrecedence:
            postfix.append(char)
        else:
            if len(stack) == 0:
                stack.append(char)
            else:
                if char == "(":
                    stack.append(char)
                elif char == ")":
                    while stack[len(stack) - 1] != "(":
                        postfix.append(stack.pop())
                    stack.pop()
                elif operatorPrecedence[char] > operatorPrecedence[stack[len(stack) - 1]]:
                    stack.append(char)
                else:
                    while len(stack) != 0:
                        if stack[len(stack) - 1] == '(':
                            break
                        postfix.append(stack.pop())
                    stack.append(char)
    while len(stack) != 0:
        postfix.append(stack.pop())
    return postfix
class Node(object):
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
class ExressionTree(object):
    def __init__(self, root = None):
        self.__root = root
    def inorder(self):
        self.__inorderHelper(self.__root)
    def __inorderHelper(self, node):
        if node:
            self.__inorderHelper(node.left)
            print (node.value)
            self.__inorderHelper(node.right)
    def preorder(self):
        self.__preorderUtil(self.__root)
    def __preorderUtil(self, node):
        if node:
            print (node.value)
            self.__preorderUtil(node.left)
            self.__preorderUtil(node.right)
    def postorder(self):
        self.__postorderUtil(self.__root)
    def __postorderUtil(self, node):
        if node:
            self.__postorderUtil(node.left)
            self.__postorderUtil(node.right)
            print (node.value)
def buildExpressionTree(infix):
    postfix = postfixConvert(infix)
      stack = []
    for char in postfix:
        if char not in operatorPrecedence:
            node = Node(char)
            stack.append(node)
        else:
            node = Node(char)
            right = stack.pop()
            left = stack.pop()
            node.right = right
            node.left = left
            stack.append(node)
    return ExressionTree(stack.pop())
print ("In Order:")
buildExpressionTree("(5+3)*6").inorder()
print ("Post Order:")
buildExpressionTree("(5+3)*6").postorder()
print ("Pre Order:")
buildExpressionTree("(5+3)*6").preorder()
```

Example: Assume that one symbol is read at a time. If the symbol is an operand, we create a tree node and push a pointer to it onto a stack. If the symbol is an operator, pop pointers to two trees T_1 and T_2 from the stack (T_1 is popped first) and form a new tree whose root is the operator and whose left and right children point to T_2 and T_1 respectively. A pointer to this new tree is then pushed onto the stack.

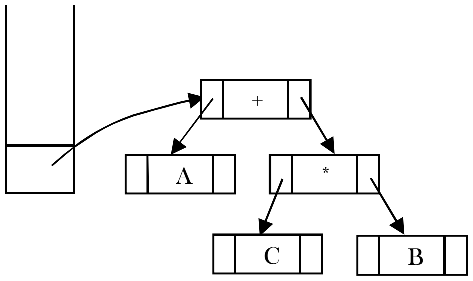
As an example, assume the input is A B C * + D /. The first three symbols are operands, so create tree nodes and push pointers to them onto a stack as shown below.



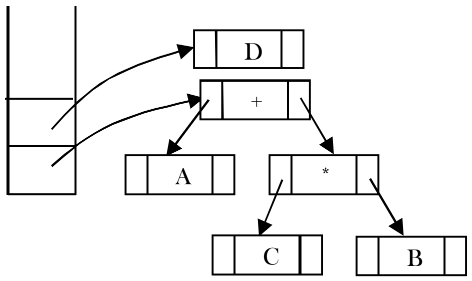
Next, an operator '*' is read, so two pointers to trees are popped, a new tree is formed and a pointer to it is pushed onto the stack.



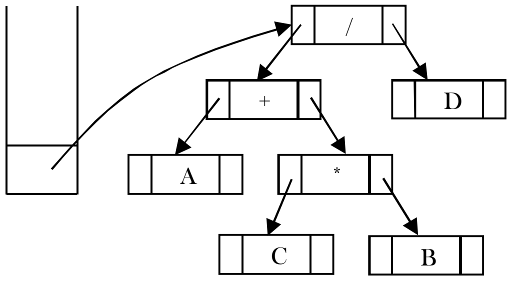
Next, an operator '+' is read, so two pointers to trees are popped, a new tree is formed and a pointer to it is pushed onto the stack.



Next, an operand 'D' is read, a one-node tree is created and a pointer to the corresponding tree is pushed onto the stack.



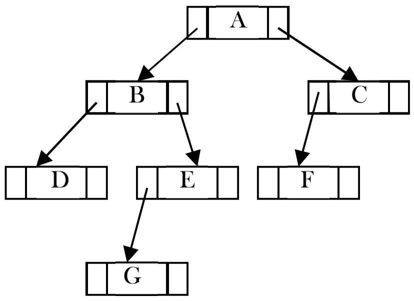
Finally, the last symbol ('/') is read, two trees are merged and a pointer to the final tree is left on the stack.



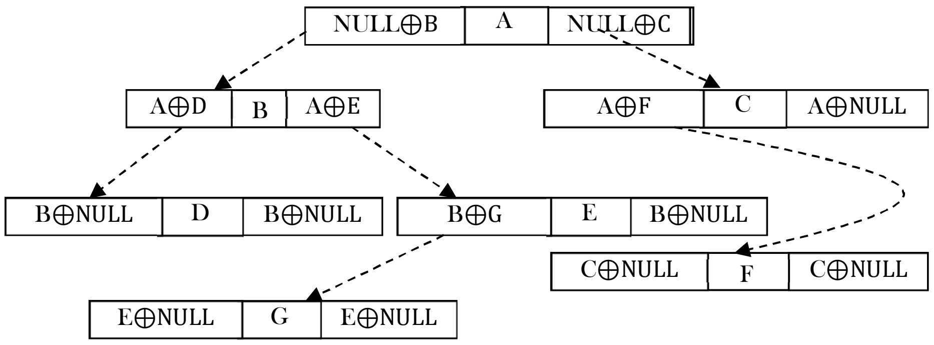
6.10 XOR Trees

This concept is similar to *memory efficient doubly linked lists* of *Linked Lists* chapter. Also, like threaded binary trees this representation does not need stacks or queues for traversing the trees. This representation is used for traversing back (to parent) and forth (to children) using ⊕ operation. To represent the same in XOR trees, for each node below are the rules used for representation:

- Each nodes left will have the ⊕ of its parent and its left children.
- Each nodes right will have the ⊕ of its parent and its right children.
- The root nodes parent is NULL and also leaf nodes children are NULL nodes.



Based on the above rules and discussion, the tree can be represented as:



The major objective of this presentation is the ability to move to parent as well to children. Now, let us see how to use this representation for traversing the tree. For example, if we are at node B and want to move to its parent node A, then we just need to perform ⊕ on its left content with its left child address (we can use right child also for going to parent node).

Similarly, if we want to move to its child (say, left child D) then we have to perform ⊕ on its left content with its parent node address. One important point that we need to understand about this representation is: When we are at node B, how do we know the address of its children D? Since the traversal starts at node root node, we can apply ⊕ on root's left content with NULL. As a result we get its left child, B. When we are at B, we can apply ⊕ on its left content with A address.

6.11 Binary Search Trees (BSTs)

Why Binary Search Trees?

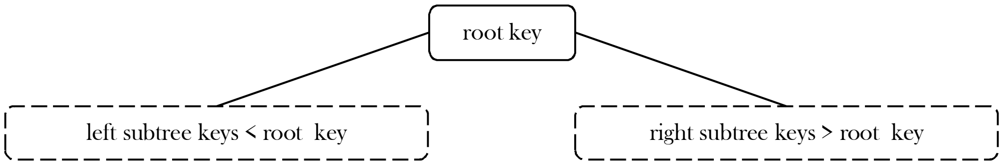
In previous sections we have discussed different tree representations and in all of them we did not impose any restriction on the nodes data. As a result, to search for an element we need to check both in left subtree and in right subtree. Due to this, the worst case complexity of search operation is O(n).

In this section, we will discuss another variant of binary trees: Binary Search Trees (BSTs). As the name suggests, the main use of this representation is for *searching*. In this representation we impose restriction on the kind of data a node can contain. As a result, it reduces the worst case average search operation to O($logn$).

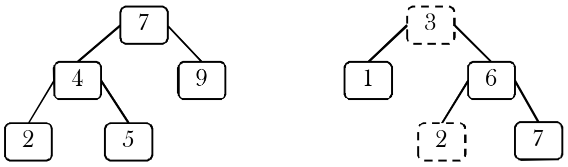
Binary Search Tree Property

In binary search trees, all the left subtree elements should be less than root data and all the right subtree elements should be greater than root data. This is called binary search tree property. Note that, this property should be satisfied at every node in the tree.

- The left subtree of a node contains only nodes with keys less than the nodes key.
- The right subtree of a node contains only nodes with keys greater than the nodes key.
- Both the left and right subtrees must also be binary search trees.



Example: The left tree is a binary search tree and the right tree is not a binary search tree (at node 6 it's not satisfying the binary search tree property).



Binary Search Tree Declaration

There is no difference between regular binary tree declaration and binary search tree declaration. The difference is only in data but not in structure. But for our convenience we change the structure name as:

```
'''Binary Search Tree Class and its methods'''
class BSTNode:
        def __init__(self, data):
            self.data = data            #root node
        self.left = None            #left child
        self.right = None           #right child
    #set data
    def setData(self, data):
        self.data = data
    #get data
    def getData(self):
        return self.data
    #get left child of a node
    def getLeft(self):
        return self.left
    #get right child of a node
    def getRight(self):
        return self.right
```

Operations on Binary Search Trees

Main operations: Following are the main operations that are supported by binary search trees:

- Find/ Find Minimum / Find Maximum element in binary search trees
- Inserting an element in binary search trees
- Deleting an element from binary search trees

Auxiliary operations: Checking whether the given tree is a binary search tree or not

- Finding k^{th}-smallest element in tree
- Sorting the elements of binary search tree and many more

Important Notes on Binary Search Trees

- Since root data is always between left subtree data and right subtree data, performing inorder traversal on binary search tree produces a sorted list.
- While solving problems on binary search trees, first we process left subtree, then root data, and finally we process right subtree. This means, depending on the problem, only the intermediate step (processing root data) changes and we do not touch the first and third steps.
- If we are searching for an element and if the left subtree root data is less than the element we want to search, then skip it. The same is the case with the right subtree.. Because of this, binary search trees take less time for searching an element than regular binary trees. In other words, the binary search trees consider either left or right subtrees for searching an element but not both.
- The basic operations that can be performed on binary search tree (BST) are insertion of element, deletion of element, and searching for an element. While performing these operations on BST the height of the tree gets changed each time. Hence there exists variations in time complexities of best case, average case, and worst case.
- The basic operations on a binary search tree take time proportional to the height of the tree. For a complete binary tree with node n, such operations runs in O(logn) worst-case time. If the tree is a linear chain of n nodes (skew-tree), however, the same operations takes O(n) worst-case time.

Finding an Element in Binary Search Trees

Find operation is straightforward in a BST. Start with the root and keep moving left or right using the BST property. If the data we are searching is same as nodes data then return current node.

If the data we are searching is less than nodes data then search left subtree of current node; otherwise search right subtree of current node. If the data is not present, end up in a *None* link.

```
def find( root, data ):
    currentNode = root
    while currentNode is not None and data != currentNode.data:
        if data > currentNode.data:
            currentNode = currentNode.right
        else:
            currentNode = currentNode.left
    return currentNode
```

Time Complexity: O(n), in worst case (when BST is a skew tree). Space Complexity: O(n), for recursive stack.

Non recursive version of the above algorithm can be given as:

```
#Search the key from node, iteratively
def find(root, data):
    currentNode = root
    while currentNode:
        if data == currentNode.data:
```

```
                    return currentNode
            if data < currentNode.data:
                    currentNode = currentNode.left
            else:
                    currentNode = currentNode.right
        return None
```

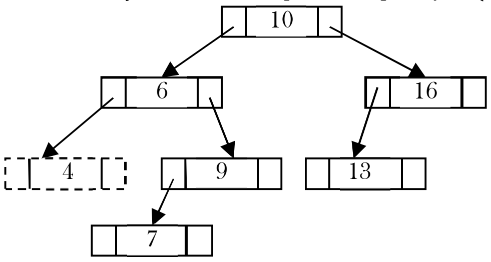
Time Complexity: O(n). Space Complexity: O(1).

Finding Minimum Element in Binary Search Trees

In BSTs, the minimum element is the left-most node, which does not have left child. In the BST below, the minimum element is **4**.

```
def findMin(root):
        currentNode = root
        if currentNode.left is None:
            return currentNode
        else:
          return findMin(currentNode.left)
```

Time Complexity: O(n), in worst case (when BST is a *left skew* tree). Space Complexity: O(n), for recursive stack.



Non recursive version of the above algorithm can be given as:

```
def findMin(root):
        currentNode = root
        if currentNode is None:
            return None
        while currentNode.left is not None:
            currentNode = currentNode.left
        return currentNode
```

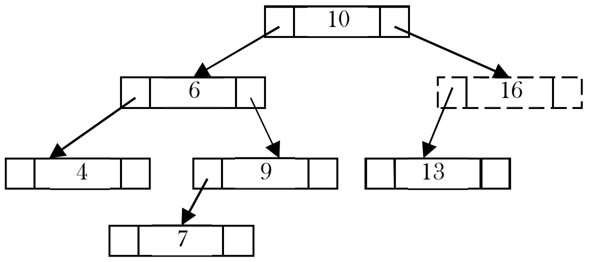
Time Complexity: O(n). Space Complexity: O(1).

Finding Maximum Element in Binary Search Trees

In BSTs, the maximum element is the right-most node, which does not have right child. In the BST below, the maximum element is **16**.

```
#Search the key from node, iteratively
def findMax(root):
        currentNode = root
        if currentNode.right is None: return currentNode
        else:
            return findMax(currentNode.right)
```

Time Complexity: O(n), in worst case (when BST is a *right skew* tree). Space Complexity: O(n), for recursive stack.



Non recursive version of the above algorithm can be given as:

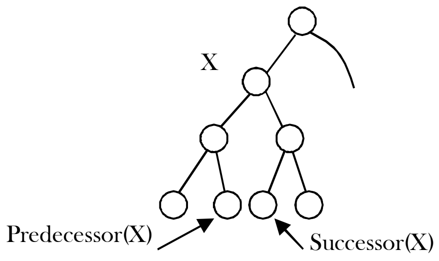
```
 def findMax(root):
        currentNode = root
        if currentNode is None:  return None
        while currentNode.right is not None:
            currentNode = currentNode.right
        return currentNode
```

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

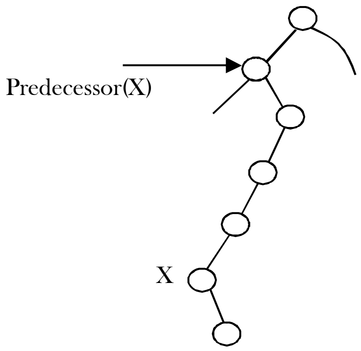
Where is Inorder Predecessor and Successor?

Where is the inorder predecessor and successor of node X in a binary search tree assuming all keys are distinct?

If X has two children then its inorder predecessor is the maximum value in its left subtree and its inorder successor the minimum value in its right subtree.



If it does not have a left child, then a node's inorder predecessor is its first left ancestor.

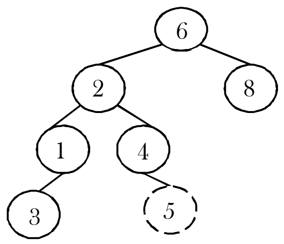


```
#Successror of a node in BST
 def successorBST(root):
        temp = None
        if root.right:
            temp = root.right
            while temp.left:
                    temp = s.left
        return temp
 # Predecessor of a node in BST
 def predecessorBST(root):
        temp = None
        if root.left:
            temp = root.left
            while temp.right:
                    temp = temp.right
        return temp
```

Inserting an Element from Binary Search Tree

To insert $data$ into binary search tree, first we need to find the location for that element. We can find the location of insertion by following the same mechanism as that of $find$ operation. While finding the location, if the $data$ is already there then we can simply neglect and come out. Otherwise, insert $data$ at the last location on the path traversed.

As an example let us consider the following tree. The dotted node indicates the element (5) to be inserted. To insert 5, traverse the tree using $find$ function. At node with key 4, we need to go right, but there is no subtree, so 5 is not in the tree, and this is the correct location for insertion.



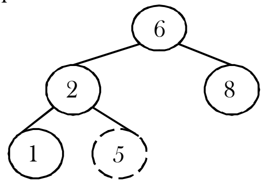
```
def insertNode(root, node):
    if root is None:
        root = node
    else:
        if root.data > node.data:
            if root.left == None:
                root.left = node
            else:
                insertNode(root.left, node)
        else:
            if root.right == None:
                root.right = node
            else:
                insertNode(root.right, node)
```

Note: In the above code, after inserting an element in subtrees, the tree is returned to its parent. As a result, the complete tree will get updated.
Time Complexity:O(n). Space Complexity:O(n), for recursive stack. For iterative version, space complexity is O(1).

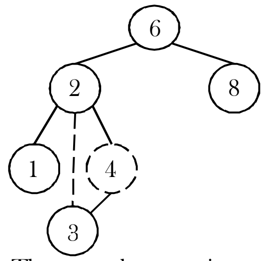
Deleting an Element from Binary Search Tree

The delete operation is more complicated than other operations. This is because the element to be deleted may not be the leaf node. In this operation also, first we need to find the location of the element which we want to delete. Once we have found the node to be deleted, consider the following cases:

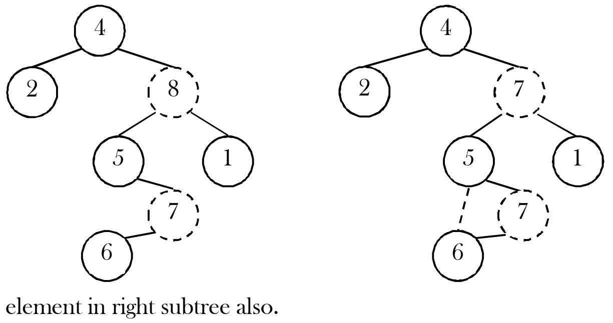
- If the element to be deleted is a leaf node: return NULL to its parent. That means make the corresponding child pointer NULL. In the tree below to delete 5, set NULL to its parent node 2.



- If the element to be deleted has one child: In this case we just need to send the current node's child to its parent. In the tree below, to delete 4, 4 left subtree is set to its parent node 2.



- If the element to be deleted has both children: The general strategy is to replace the key of this node with the largest element of the left subtree and recursively delete that node (which is now empty). The largest node in the left subtree cannot have a right child, so the second *delete* is an easy one. As an example, let us consider the following tree. In the tree below, to delete 8, it is the right child of the root. The key value is 8. It is replaced with the largest key in its left subtree (7), and then that node is deleted as before (second case).



Note: We can replace with minimum element in right subtree also.

```
def deleteNode(root, data):
    """ delete the node with the given data and return the root node of the tree """
    if root.data == data:
            # found the node we need to delete
            if root.right and root.left:
                    # get the successor node and its parent
                    [psucc, succ] = findMin(root.right, root)
```

```
                # splice out the successor (we need the parent to do this)
                if psucc.left == succ:
                    psucc.left = succ.right
                else:
                    psucc.right = succ.right
                # reset the left and right children of the successor
                succ.left = root.left
                succ.right = root.right
                return succ
            else:
                # "easier" case
                if root.left:
                    return root.left          # promote the left subtree
                else:
                    return root.right         # promote the right subtree
        else:
            if root.data > data:      # data should be in the left subtree
                if root.left:
                    root.left = deleteNode(root.left, data)
                # else the data is not in the tree
            else:     # data should be in the right subtree
                if root.right:
                    root.right = deleteNode(root.right, data)
        return root
    def findMin(root, parent):
        """ return the minimum node in the current tree and its parent """
        # we use an ugly trick: the parent node is passed in as an argument
        # so that eventually when the leftmost child is reached, the
        # call can return both the parent to the successor and the successor
        if root.left:
            return findMin(root.left, root)
        else:
            return [parent, root]
```

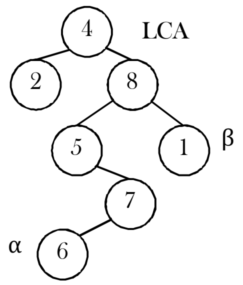
Time Complexity: O(n). Space Complexity: O(n) for recursive stack. For iterative version, space complexity is O(1).

Binary Search Trees: Problems & Solutions

Note: For ordering related problems with binary search trees and balanced binary search trees, Inorder traversal has advantages over others as it gives the sorted order.

Problem-51 Given pointers to two nodes in a binary search tree, find the lowest common ancestor (LCA). Assume that both values already exist in the tree.

Solution:



The main idea of the solution is: while traversing BST from root to bottom, the first node we encounter with value between α and β, i.e., $\alpha < node \rightarrow data < \beta$, is the Least Common Ancestor(LCA) of α and β (where $\alpha < \beta$). So just traverse the BST in pre-order, and if we find a node with value in between α and β, then that node is the LCA.

If its value is greater than both α and β, then the LCA lies on the left side of the node, and if its value is smaller than both α and β, then the LCA lies on the right side.

```
def lca(root, a, b):
    if a <= root.data <= b or b <= root.data <= a:
        return root
    if a < root.data and b < root.data:
        return lca(root.left, a, b)
    if a > root.data and b > root.data:
        return lca(root.right, a, b)
```

Time Complexity: O(n). Space Complexity: O(n), for skew trees.

Problem-52 Give an algorithm for finding the shortest path between two nodes in a BST.

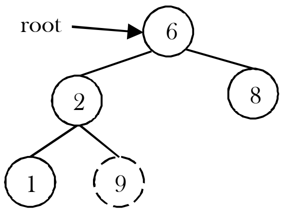
Solution: It's nothing but finding the LCA of two nodes in BST.

Problem-53 Give an algorithm for counting the number of BSTs possible with n nodes.

Solution: This is a DP problem. Refer to chapter on *Dynamic Programming* for the algorithm.

Problem-54 Give an algorithm to check whether the given binary tree is a BST or not.

Solution:



Consider the following simple program. For each node, check if the node on its left is smaller and check if the node on its right is greater. This approach is wrong as this will return true for binary tree below. Checking only at current node is not enough.

```
def isBST(root):
    if root == None:
      return 1
    # false if left is > than root
    if root.left is not None and root.left.data > root.data:
      return 0
    # false if right is < than root
    if root.right is not None and root.right.data < root.data:
      return 0
    # false if, recursively, the left or right is not a BST
    if not isBST(root.left) or not isBST(root.right):
      return 0
    # passing all that, it's a BST
    return 1
```

Time Complexity: O(n), but algorithm is incorrect. Space Complexity: O(n), for runtime stack space.

Problem-55 Can we think of getting the correct algorithm?

Solution: For each node, check if max value in left subtree is smaller than the current node data and min value in right subtree greater than the node data. It is assumed that we have helper functions *FindMin*() and *FindMax*() that return the min or max integer value from a non-empty tree.

```
# Returns true if a binary tree is a binary search tree
def isBST(root):
    if root is None:
      return 1
    # false if the max of the left is > than root
    if root.left is not None and FindMax(root.left) > root.data:
      return 0
    # false if the min of the right is <= than root
    if root.right is not None and FindMin(root.right) < root.data:
      return 0
    # false if, recursively, the left or right is not a BST
    if not isBST(root.left) or not isBST(root.right):
      return 0
    # passing all that, it's a BST
    return 1
```

Time Complexity: O(n^2). In a BST, we spend O(n) time (in the worst case) for finding the maximum element in left subtree and O(n) time for finding the minimum element in right subtree. In the above algorithm, for every element we keep finding the maximum element in left subtree and minimum element in right subtree which would cost O($2n$)≈O(n). Since there were n such elements, the overall time complexity is O(n^2).
Space Complexity: O(n) for runtime stack space.

Problem-56 Can we improve the complexity of Problem-55?

Solution: Yes. We can improve the time complexity of previous algorithm. A better solution is to look at each node only once. The trick is to write a utility helper function isBST(root, min, max) that traverses down the tree keeping track of the narrowing min and max allowed values as it goes, looking at each node only once. The initial values for min and max should be $float("-infinity")$, and $float("infinity")$, — they narrow from there.

```
def isBST(root, min, max):
        if root is None:
            return 1
        if root.data <= min or root.data >= max:
            return 0
        result = isBST(root.left, min, root.data)
        result = result and isBST(root.right, root.data, max)
        return result
print(isBST(root, float("-infinity"), float("infinity")))
```

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

Problem-57 Can we further improve the complexity of Problem-55?

Solution: We can further improve the solution by using in-order traversal. The idea behind this solution is that in-order traversal of BST produces sorted lists. While traversing the BST in in-order, at each node check the condition that its key value should be greater than the key value of its previous visited node. Also, we need to initialize the previous value with possible minimum integer value (say, $float("-infinity")$).

```
previousValue = float("infinity")
def isBST4(root, previousValue):
        if root is None:
            return True
        if not isBST4(root.left, previousValue):
            return False
        if root.data < previousValue:
            return False
        previousValue = root.data
        return isBST4(root.right, previousValue)
```

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

Problem-58 Give an algorithm for converting BST to circular DLL with space complexity $O(1)$.

Solution: Convert left and right subtrees to DLLs and maintain end of those lists. Then, adjust the pointers.

```
def BSTToDLL(root):
      ''' main function to take the root of the BST and return the head of the doubly linked list '''
      prev = None
      head = None
      BSTToDoublyList(root, prev, head)
      return head
def BSTToDoublyList(root, prev, head):
      if (not root): return
      BSTToDoublyList(root.left, prev, head)
      # current node's left points to previous node
      root.left = prev
      prev.right = root                     # Previous node's right points to current node
      head = root                           # If previous is None that current node is head
      right = root.right                    # Saving right node
      #Now we need to make list created till now as circular
      head.left = root
      root.right = head
      #For right-subtree/parent, current node is in-order predecessor
      prev = root
      BSTToDoublyList(right, prev, head)
```

Time Complexity: $O(n)$.

Problem-59 For Problem-58, is there any other way of solving it?

Solution: Yes. There is an alternative solution based on the divide and conquer method which is quite neat. As evident, the function considers 4 major cases:

1. When the current node(root) is a leaf node

2. When there exists no left child
3. When there exists no right child
4. When there exists both left and right child

```
def BSTToDLL(root):
    ''' main function to take the root of the BST and return the head of the doubly linked list  '''
    #for leaf Node return itself
    if root.left == root and root.right == root:
        return root
    elif root.left == root:                    # No left subtree exist
        h2 = BSTToDLL(root.right)
        root.right = h2
        h2.left.right = root
        root.left = h2.left
        h2.left = root
        return root
    elif root.right == root:                          # No right subtree exist
        h1 = BSTToDLL(root.left)
        root.left = h1.left
        h1.left.right = root
        root.right = h1
        h1.left = root
        return h1
    else:                                             # Both left and right subtrees exist
        h1 = BSTToDLL(root.left)
        h2 = BSTToDLL(root.right)
        l1 = h1.left                                  # Find last nodes of the lists
        l2 = h2.left
        h1.left = l2
        l2.right = h1
        l1.right = root
        root.left = l1
        root.right = h2
        h2.left = root
        return h1
```

Time Complexity: $O(n)$.

Problem-60 Given a sorted doubly linked list, give an algorithm for converting it into balanced binary search tree.

Solution: Find the middle node and adjust the pointers.

```
def DLLToBalancedBST(head):
        if( not head or not head.next):
                return head
        # Refer Linked Lists chapter for this function. We can use two-pointer logic to find the middle node
        temp = FindMiddleNode(head)
        p = head
        while(p.next != temp):
                p = p.next
        p.next = None
        q = temp.next
        temp.next = None
        temp.prev = DLLToBalancedBST(head)
        temp.next = DLLToBalancedBST(q)
        return temp
```

Time Complexity: $2T(n/2) + O(n)$ [for finding the middle node] = $O(nlogn)$.

Note: For *FindMiddleNode* function refer *Linked Lists* chapter.

Problem-61 Given a sorted array, give an algorithm for converting the array to BST.

Solution: If we have to choose an array element to be the root of a balanced BST, which element should we pick? The root of a balanced BST should be the middle element from the sorted array. We would pick the middle element from the sorted array in each iteration. We then create a node in the tree initialized with this element. After the element is chosen, what is left? Could you identify the sub-problems within the problem?

There are two arrays left – the one on its left and the one on its right. These two arrays are the sub-problems of the original problem, since both of them are sorted. Furthermore, they are subtrees of the current node's left and right child.

The code below creates a balanced BST from the sorted array in $O(n)$ time (n is the number of elements in the array). Compare how similar the code is to a binary search algorithm. Both are using the divide and conquer methodology.

```
def  buildBST(A, left, right) :
        if(left > right):
                return None
        newNode = Node()
        if(not newNode) :
                print("Memory Error")
                return
        if(left == right):
                newNode.data = A[left]
                newNode.left = None
                newNode.right = None
        else :
                mid = left + (right-left)/ 2
                newNode.data = A[mid]
                newNode.left = buildBST(A, left, mid - 1)
                newNode.right = buildBST(A, mid + 1, right)
        return newNode
if __name__ == "__main__":
        #create the sample BST
        A= [2, 3, 4, 5, 6, 7]
        root = buildBST(A, 0, len(A)-1)
        print ("\ncreating BST")
        printBST(root)
```

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

Problem-62 Given a singly linked list where elements are sorted in ascending order, convert it to a height balanced BST.

Solution: A naive way is to apply the Problem-60 solution directly. In each recursive call, we would have to traverse half of the list's length to find the middle element. The run time complexity is clearly O($nlogn$), where n is the total number of elements in the list. This is because each level of recursive call requires a total of $n/2$ traversal steps in the list, and there are a total of $logn$ number of levels (ie, the height of the balanced tree).

Problem-63 For Problem-62, can we improve the complexity?

Solution: Hint: How about inserting nodes following the list order? If we can achieve this, we no longer need to find the middle element as we are able to traverse the list while inserting nodes to the tree.

Best Solution: As usual, the best solution requires us to think from another perspective. In other words, we no longer create nodes in the tree using the top-down approach. Create nodes bottom-up, and assign them to their parents. The bottom-up approach enables us to access the list in its order while creating nodes [42].

Isn't the bottom-up approach precise? Any time we are stuck with the top-down approach, we can give bottom-up a try. Although the bottom-up approach is not the most natural way we think, it is helpful in some cases. However, we should prefer top-down instead of bottom-up in general, since the latter is more difficult to verify.

Below is the code for converting a singly linked list to a balanced BST. Please note that the algorithm requires the list length to be passed in as the function parameters. The list length can be found in O(n) time by traversing the entire list once. The recursive calls traverse the list and create tree nodes by the list order, which also takes O(n) time. Therefore, the overall run time complexity is still OO(n).

```
def sortedListToBST(head, start, end):
        if(start > end):
                return None
        # same as (start+end)/2, avoids overflow
        mid = start + (end - start) // 2
        left = sortedListToBST(head, start, mid-1)
        root = BSTNode(head.data)
        head = head.next
        print ("root data mid:",mid, root.data)
        root.left = left
        root.right = sortedListToBST(head, mid+1, end)
        return root
def converTsortedListToBST(head, n) :
        return sortedListToBST(head, 0, n-1)
```

Problem-64 Give an algorithm for finding the k^{th} smallest element in BST.

Solution: The idea behind this solution is that, inorder traversal of BST produces sorted lists. While traversing the BST in inorder, keep track of the number of elements visited.

```
count=0
def kthSmallestInBST(root, k):
        global count
        if(not root):
            return None
        left = kthSmallestInBST(root.left, k)
        if( left ):
           return left
        count += 1
        if(count == k):
                return root
        return kthSmallestInBST(root.right, k)
```

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

Problem-65 **Floor and ceiling:** If a given key is less than the key at the root of a BST then the floor of the key (the largest key in the BST less than or equal to the key) must be in the left subtree. If the key is greater than the key at the root, then the floor of the key could be in the right subtree, but only if there is a key smaller than or equal to the key in the right subtree; if not (or if the key is equal to the the key at the root) then the key at the root is the floor of the key. Finding the ceiling is similar, interchanging right and left. For example, if the sorted with input array is {1, 2, 8, 10, 10, 12, 19}, then

For $x = 0$: floor doesn't exist in array, ceil = 1, For $x = 1$: floor = 1, ceil = 1
For $x = 5$: floor = 2, ceil = 8, For $x = 20$: floor = 19, ceil doesn't exist in array

Solution: The idea behind this solution is that, inorder traversal of BST produces sorted lists. While traversing the BST in inorder, keep track of the values being visited. If the roots data is greater than the given value then return the previous value which we have maintained during traversal. If the roots data is equal to the given data then return root data.

```
def floorInBSTUtil(root, data):
        if(not root):
           return sys.maxint
        if(root.data == data) :
           return root.data
        if(data < root.data ):
           return floorInBSTUtil(root.left, data)
        floor = floorInBSTUtil(root.right, data)
        if floor <=  data:
           return floor
        else: return root.data
```

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

For ceiling, we just need to call the right subtree first, followed by left subtree.

```
def ceilInBST(root, data):
        # Base case
        if( root == None ):
           return -sys.maxint
        # Found equal data
        if( root.data == data ):
           return root.data
        # If root's data is smaller, ceil must be in right subtree
        if( root.data < data ):
           return ceilInBST(root.right, data)
        # Else, either left subtree or root has the ceil data
        ceil = ceilInBST(root.left, data)
        if ceil >= data:
           return ceil
        else: return root.data
```

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

Problem-66 Give an algorithm for finding the union and intersection of BSTs. Assume parent pointers are available (say threaded binary trees). Also, assume the lengths of two BSTs are m and n respectively.

Solution: If parent pointers are available then the problem is same as merging of two sorted lists. This is because if we call inorder successor each time we get the next highest element. It's just a matter of which inorderSuccessor to call.

Time Complexity: $O(m + n)$. Space complexity: $O(1)$.

Problem-67 For Problem-66, what if parent pointers are not available?

Solution: If parent pointers are not available, the BSTs can be converted to linked lists and then merged.

1 Convert both the BSTs into sorted doubly linked lists in $O(n + m)$ time. This produces 2 sorted lists.
2 Merge the two double linked lists into one and also maintain the count of total elements in $O(n + m)$ time.
3 Convert the sorted doubly linked list into height balanced tree in $O(n + m)$ time.

Problem-68 For Problem-69, can we still think of an alternative way to solve the problem?

Solution: Yes, by using inorder traversal.

- Perform inorder traversal on one of the BSTs.
- While performing the traversal store them in table (hash table).
- After completion of the traversal of first BST, start traversal of second BST and compare them with hash table contents.

Time Complexity: $O(m + n)$. Space Complexity: $O(Max(m,n))$.

Problem-69 Given a BST and two numbers $K1$ and $K2$, give an algorithm for printing all the elements of BST in the range $K1$ and $K2$.

Solution:

```
def rangePrinter(root, K1, K2):
        if not root:
            return
        if K1 <= root.data <= K2:
            print(root.data)
        if root.data < K1:
            return rangePrinter(root.right,K1, K2)
        if root.data > K2:
            return rangePrinter(root.left, K1, K2)
```

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

Problem-70 For Problem-69, is there any alternative way of solving the problem?

Solution: We can use level order traversal: while adding the elements to queue check for the range.

```
import Queue
def rangePrinter(root, K1, K2):
        if root is None:
            return
        q = Queue.Queue()
        q.put(root )
        temp= None
        while not q.empty():
                temp = q.get()                          # deQueue FIFO
                if K1 <= root.data <= K2:
                        print(root.data)
                if temp.left is not None and temp.data  >= K1:
                        q.put(temp.left)
                if temp.right is not None and temp.data  <= K2:
                        q.put(temp.right)
```

Time Complexity: $O(n)$. Space Complexity: $O(n)$, for queue.

Problem-71 For Problem-69, can we still think of alternative way for solving the problem?

Solution: First locate $K1$ with normal binary search and after that use InOrder successor until we encounter $K2$. For algorithm, refer to problems section of threaded binary trees.

Problem-72 Given root of a Binary Search tree, trim the tree, so that all elements returned in the new tree are between the inputs A and B.

Solution: It's just another way of asking Problem-69.

```
def trimBST(root, minVal, maxVal):
        if not root:
            return
        root.left = trimBST(root.left, minVal, maxVal)
        root.right = trimBST(root.right, minVal, maxVal)
        if minVal<=root.data<=maxVal:
            return root
        if root.data<minVal:
            return root.right
        if root.data>maxVal:
            return root.left
```

Problem-73 Given two BSTs, check whether the elements of them are the same or not.
For example: two BSTs with data 10 5 20 15 30 and 10 20 15 30 5 should return true and the dataset with 10 5 20 15 30 and 10 15 30 20 5 should return false. **Note:** BSTs data can be in any order.

Solution: One simple way is performing an inorder traversal on first tree and storing its data in hash table. As a second step, perform inorder traversal on second tree and check whether that data is already there in hash table or not (if it exists in hash table then mark it with -1 or some unique value).

During the traversal of second tree if we find any mismatch return false. After traversal of second tree check whether it has all -1s in the hash table or not (this ensures extra data available in second tree).

Time Complexity: $O(max(m, n))$, where m and n are the number of elements in first and second BST. Space Complexity: $O(max(m, n))$. This depends on the size of the first tree.

Problem-74 For Problem-73, can we reduce the time complexity?

Solution: Instead of performing the traversals one after the other, we can perform $in - order$ traversal of both the trees in parallel. Since the $in - order$ traversal gives the sorted list, we can check whether both the trees are generating the same sequence or not.

Time Complexity: $O(max(m, n))$. Space Complexity: $O(1)$. This depends on the size of the first tree.

Problem-75 For the key values $1 \ldots n$, how many structurally unique BSTs are possible that store those keys.

Solution: Strategy: consider that each value could be the root. Recursively find the size of the left and right subtrees.

```
def countTrees(n) :
    if (n <= 1):
        return 1
    else :
            # there will be one value at the root, with whatever remains on the left and right
            # each forming their own subtrees. Iterate through all the values that could be the root...
            sum = 0
            for root in range(1,n+1):
                left = countTrees(root - 1)
                right = countTrees(n - root)
                # number of possible trees with this root == left*right
                sum += left*right
                return(sum)
```

Problem-76 Given a BST of size n, in which each node r has an additional field $r \rightarrow size$, the number of the keys in the sub-tree rooted at r (including the root node r). Give an $O(h)$ algorithm $GreaterthanConstant(r, k)$ to find the number of keys that are strictly greater than k (h is the height of the binary search tree).

Solution:

```
def greaterThanConstant (r, k):
    keysCount = 0
    while (r):
        if (k < r.data):
            keysCount = keysCount + r.right.size + 1
            r = r.left
        else if (k > r.data):
            r = r.right
        else:
            keysCount = keysCount + r.right.size
            break
    return keysCount
```

The suggested algorithm works well if the key is a unique value for each node. Otherwise when reaching k=$r.data$, we should start a process of moving to the right until reaching a node y with a key that is bigger then k, and then we should return $keysCount + y.size$. Time Complexity: $O(h)$ where h=$O(n)$ in the worst case and $O(logn)$ in the average case.

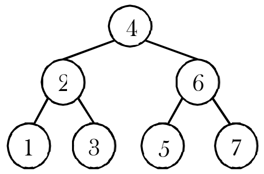
6.12 Balanced Binary Search Trees

In earlier sections we have seen different trees whose worst case complexity is $O(n)$, where n is the number of nodes in the tree. This happens when the trees are skew trees. In this section we will try to reduce this worst case complexity to $O(logn)$ by imposing restrictions on the heights.

In general, the height balanced trees are represented with $HB(k)$, where k is the difference between left subtree height and right subtree height. Sometimes k is called $balance\ factor$.

Full Balanced Binary Search Trees

In $HB(k)$, if $k = 0$ (if balance factor is zero), then we call such binary search trees as $full$ balanced binary search trees. That means, in $HB(0)$ binary search tree, the difference between left subtree height and right subtree height should be at most zero. This ensures that the tree is a full binary tree. For example,



Note: For constructing $HB(0)$ tree refer to $Problems$ section.

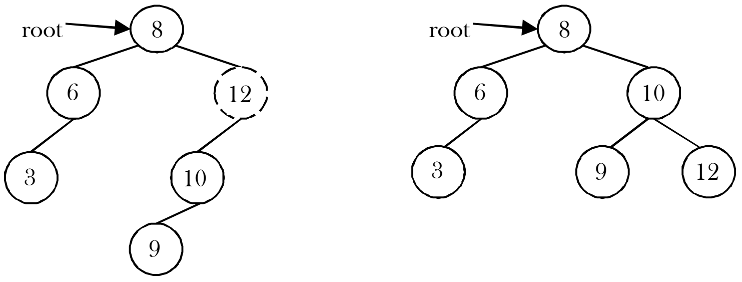
6.13 AVL (Adelson-Velskii and Landis) Trees

In $HB(k)$, if $k = 1$ (if balance factor is one), such a binary search tree is called an $AVL\ tree$. That means an AVL tree is a binary search tree with a $balance$ condition: the difference between left subtree height and right subtree height is at most 1.

Properties of AVL Trees

A binary tree is said to be an AVL tree, if:

- It is a binary search tree, and
- For any node X, the height of left subtree of X and height of right subtree of X differ by at most 1.



As an example, among the above binary search trees, the left one is not an AVL tree, whereas the right binary search tree is an AVL tree.

Minimum/Maximum Number of Nodes in AVL Tree

For simplicity let us assume that the height of an AVL tree is h and $N(h)$ indicates the number of nodes in AVL tree with height h. To get the minimum number of nodes with height h, we should fill the tree with the minimum number of nodes possible. That means if we fill the left subtree with height $h-1$ then we should fill the right subtree with height $h-2$. As a result, the minimum number of nodes with height h is:

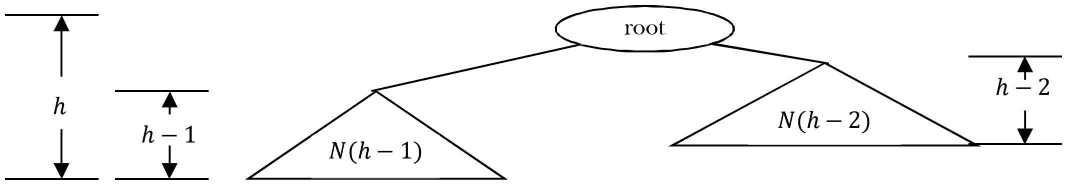
$$N(h) = N(h-1) + N(h-2) + 1$$

In the above equation:

- $N(h-1)$ indicates the minimum number of nodes with height $h-1$.
- $N(h-2)$ indicates the minimum number of nodes with height $h-2$.
- In the above expression, "1" indicates the current node.

We can give $N(h-1)$ either for left subtree or right subtree. Solving the above recurrence gives:

$$N(h) = O(1.618^h) \Longrightarrow h = 1.44logn \approx O(logn)$$



Where n is the number of nodes in AVL tree. Also, the above derivation says that the maximum height in AVL trees is $O(logn)$. Similarly, to get maximum number of nodes, we need to fill both left and right subtrees with height $h-1$. As a result, we get:

$$N(h) = N(h-1) + N(h-1) + 1 = 2N(h-1) + 1$$

The above expression defines the case of full binary tree. Solving the recurrence we get:

$$N(h) = O(2^h) \Longrightarrow h = logn \approx O(logn)$$

∴ In both the cases, AVL tree property is ensuring that the height of an AVL tree with n nodes is $O(logn)$.

AVL Tree Declaration

Since AVL tree is a BST, the declaration of AVL is similar to that of BST. But just to simplify the operations, we also include the height as part of the declaration.

```
class AVLNode:
    def __init__(self,data,balanceFactor,left,right):
        self.data = data
        self.balanceFactor = 0
        self.left = left
        self.right = right
```

Finding the Height of an AVL tree

```
def height(self):
        return self.recHeight(self.root)
def recHeight(self,root):
        if root is None:
                return 0
        else:
                leftH = self.recHeight(r.left)
                rightH = self.recHeight(r.right)
                if leftH>rightH:
                        return 1+leftH
                else:
                        return 1+rightH
```

Time Complexity: $O(1)$.

Rotations

When the tree structure changes (e.g., with insertion or deletion), we need to modify the tree to restore the AVL tree property. This can be done using single rotations or double rotations. Since an insertion/deletion involves adding/deleting a single node, this can only increase/decrease the height of a subtree by 1. So, if the AVL tree property is violated at a node X, it means that the heights of left(X) and right(X) differ by exactly 2. This is because, if we balance the AVL tree every time, then at any point, the difference in heights of left(X) and right(X) differ by exactly 2. Rotations is the technique used for restoring the AVL tree property. This means, we need to apply the rotations for the node X.

Observation: One important observation is that, after an insertion, only nodes that are on the path from the insertion point to the root might have their balances altered, because only those nodes have their subtrees altered. To restore the AVL tree property, we start at the insertion point and keep going to the root of the tree.

While moving to the root, we need to consider the first node that is not satisfying the AVL property. From that node onwards, every node on the path to the root will have the issue. Also, if we fix the issue for that first node, then all other nodes on the path to the root will automatically satisfy the AVL tree property. That means we always need to care for the first node that is not satisfying the AVL property on the path from the insertion point to the root and fix it.

Types of Violations

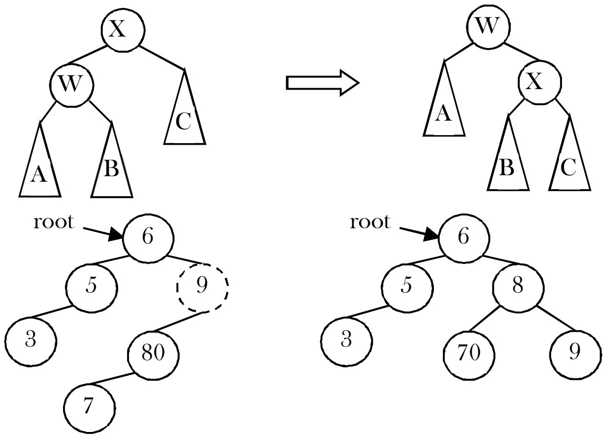
Let us assume the node that must be rebalanced is X. Since any node has at most two children, and a height imbalance requires that $X's$ two subtree heights differ by two, we can observe that a violation might occur in four cases:

1. An insertion into the left subtree of the left child of X.
2. An insertion into the right subtree of the left child of X.
3. An insertion into the left subtree of the right child of X.
4. An insertion into the right subtree of the right child of X.

Cases 1 and 4 are symmetric and easily solved with single rotations. Similarly, cases 2 and 3 are also symmetric and can be solved with double rotations (needs two single rotations).

Single Rotations

Left Left Rotation (LL Rotation) [Case-1]: In the case below, node X is not satisfying the AVL tree property. As discussed earlier, the rotation does not have to be done at the root of a tree. In general, we start at the node inserted and travel up the tree, updating the balance information at every node on the path.

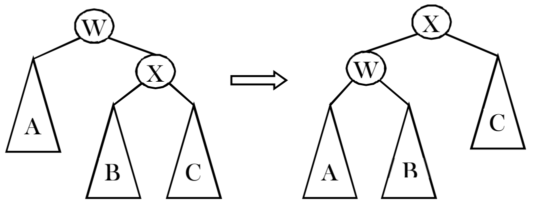


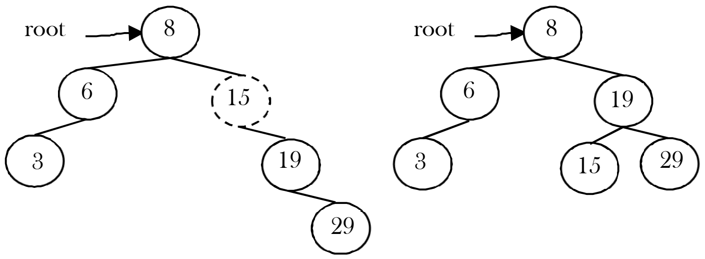
For example, in the figure above, after the insertion of 7 in the original AVL tree on the left, node 9 becomes unbalanced. So, we do a single left-left rotation at 9. As a result we get the tree on the right.

```
def singleLeftRotate(self,root):
        W = root.left
        root.left = W.right
        W.right = root
        return W
```

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

Right Right Rotation (RR Rotation) [Case-4]: In this case, node *X* is not satisfying the AVL tree property.





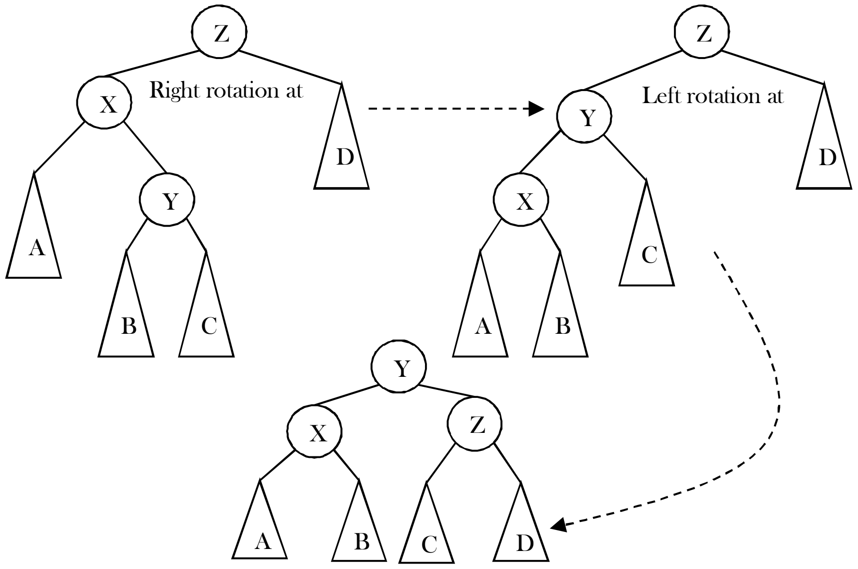
For example, in the above figure, after the insertion of 29 in the original AVL tree on the left, node 15 becomes unbalanced. So, we do a single right-right rotation at 15. As a result we get the tree on the right.

```
def singleRightRotate(self,root):
        X = root.right
        root.right = X.left
        X.left = root
        return X
```

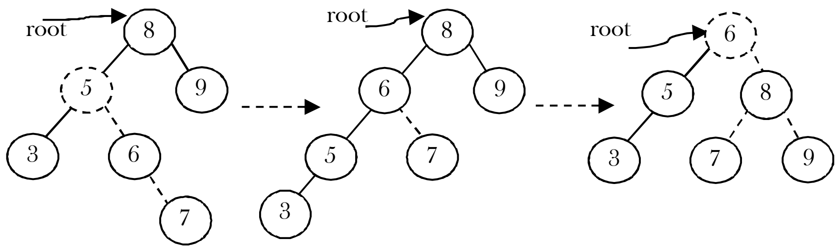
Time Complexity: O(1). Space Complexity: O(1).

Double Rotations

Left Right Rotation (LR Rotation) [Case-2]: For case-2 and case-3 single rotation does not fix the problem. We need to perform two rotations.



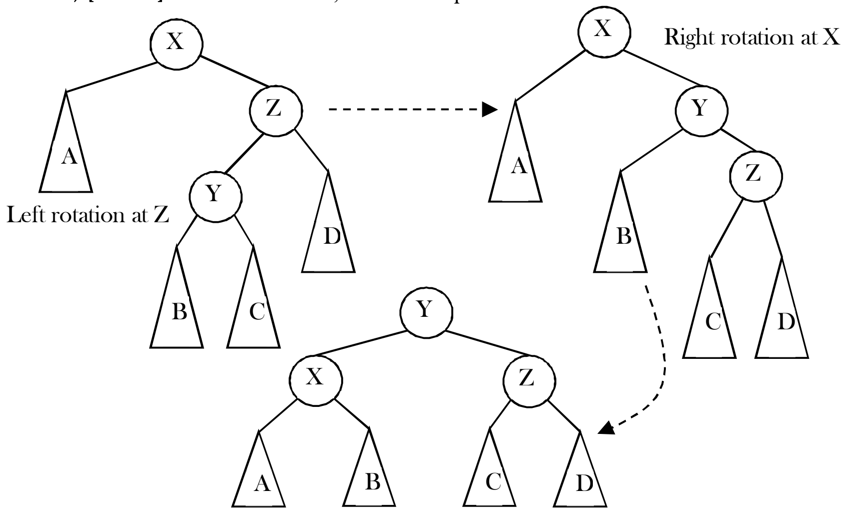
As an example, let us consider the following tree: Insertion of 7 is creating the case-2 scenario and right side tree is the one after double rotation.



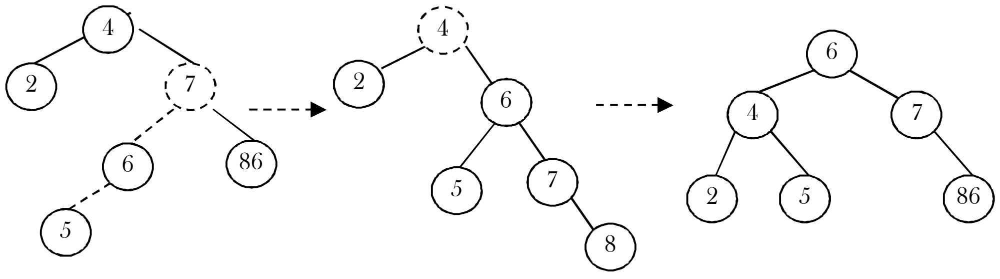
Code for left-right double rotation can be given as:

```
def rightLeftRotate(self,root):
    X = root.left
    if X.balanceFactor == -1:
        root.balanceFactor = 0
        X.balanceFactor = 0
        root = self.singleLeftRotate(root)
    else:
        Y = X.right
        if Y.balanceFactor == -1:
            root.balanceFactor = 1
            X.balanceFactor = 0
        elif Y.balanceFactor == 0:
            root.balanceFactor = 0
            X.balanceFactor = 0
        else:
            root.balanceFactor = 0
            X.balanceFactor = -1
        Y.balanceFactor = 0
        root.left = self.singleRightRotate(X)
        root = self.singleLeftRotate(root)
    return root
```

Right Left Rotation (RL Rotation) [Case-3]: Similar to case-2, we need to perform two rotations to fix this scenario.



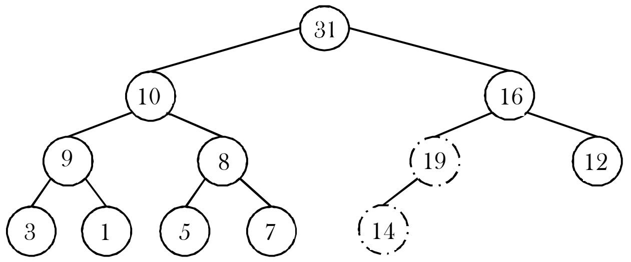
As an example, let us consider the following tree: The insertion of 6 is creating the case-3 scenario and the right side tree is the one after the double rotation.



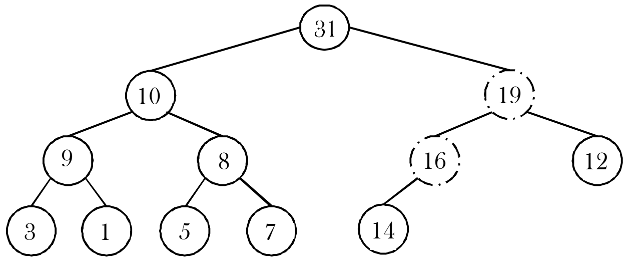
As an example, let us consider the following tree: The insertion of 6 is creating the case-3 scenario and the right side tree is the one after the double rotation.

```
def rightLeftRotate(self, root):
```

In order to heapify this element (19), we need to compare it with its parent and adjust them. Swapping 19 and 14 gives:



Again, swap 19 and16:



Now the tree is satisfying the heap property. Since we are following the bottom-up approach we sometimes call this process *percolate up*.

```
def insert(self,k):
    self.heapList.append(k)
    self.size = self.size + 1
    self.percolateUp(self.size)
```

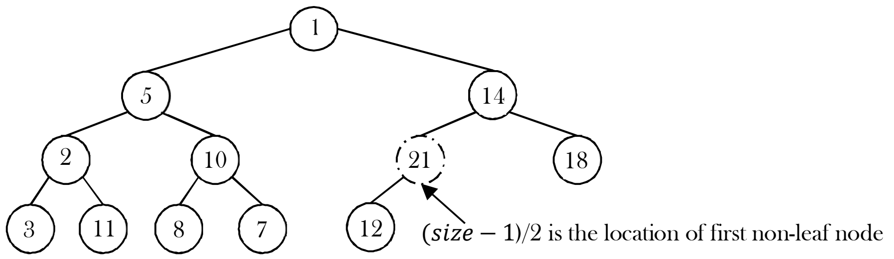
Time Complexity: O($logn$). The explanation is the same as that of the *Heapify* function.

Heapifying the Array

One simple approach for building the heap is, take n input items and place them into an empty heap. This can be done with n successive inserts and takes O($nlogn$) in the worst case. This is due to the fact that each insert operation takes O($logn$).

To finish our discussion of binary heaps, we will look at a method to build an entire heap from a list of keys. The first method you might think of may be like the following. Given a list of keys, you could easily build a heap by inserting each key one at a time. Since you are starting with a list of one item, the list is sorted and you could use binary search to find the right position to insert the next key at a cost of approximately O($logn$) operations. However, remember that inserting an item in the middle of the list may require O(n) operations to shift the rest of the list over to make room for the new key. Therefore, to insert n keys into the heap would require a total of O($nlogn$) operations. However, if we start with an entire list then we can build the whole heap in O(n) operations.

Observation: Leaf nodes always satisfy the heap property and do not need to care for them. The leaf elements are always at the end and to heapify the given array it should be enough if we heapify the non-leaf nodes. Now let us concentrate on finding the first non-leaf node. The last element of the heap is at location $h \rightarrow size - 1$, and to find the first non-leaf node it is enough to find the parent of the last element.



```
def buildHeap(self,A):
    i = len(A) // 2
    self.size = len(A)
    self.heapList = [0] + A[:]
    while (i > 0):
        self.percolateDown(i)
        i = i - 1
```

Time Complexity: The linear time bound of building heap can be shown by computing the sum of the heights of all the nodes. For a complete binary tree of height h containing $n = 2^{h+1} - 1$ nodes, the sum of the heights of the nodes is $n - h - 1 = n - logn - 1$ (for proof refer to *Problems Section*). That means, building the heap operation can be done in linear time (O(n)) by applying a *PercolateDown* function to the nodes in reverse level order.

7.7 Heapsort

One main application of heap ADT is sorting (heap sort). The heap sort algorithm inserts all elements (from an unsorted array) into a heap, then removes them from the root of a heap until the heap is empty. Note that heap sort can be done in place with the array to be sorted.

Instead of deleting an element, exchange the first element (maximum) with the last element and reduce the heap size (array size). Then, we heapify the first element. Continue this process until the number of remaining elements is one.

```
def heapSort( A ):
  # convert A to heap
  length = len( A ) - 1
  leastParent = length / 2
  for i in range ( leastParent, -1, -1 ):
    percolateDown( A, i, length )
  # flatten heap into sorted array
  for i in range ( length, 0, -1 ):
    if A[0] > A[i]:
      swap( A, 0, i )
      percolateDown( A, 0, i - 1 )
 #Modfied percolateDown to skip the sorted elements
def percolateDown( A, first, last ):
  largest = 2 * first + 1
  while largest <= last:
    # right child exists and is larger than left child
    if ( largest < last ) and ( A[largest] < A[largest + 1] ):
      largest += 1
    # right child is larger than parent
    if A[largest] > A[first]:
      swap( A, largest, first )
      # move down to largest child
      first = largest
      largest = 2 * first + 1
    else:
      return # force exit
def swap( A, x, y ):
  temp = A[x]
  A[x] = A[y]
  A[y] = temp
```

Time complexity: As we remove the elements from the heap, the values become sorted (since maximum elements are always $root$ only). Since the time complexity of both the insertion algorithm and deletion algorithm is O($logn$) (where n is the number of items in the heap), the time complexity of the heap sort algorithm is O($nlogn$).

7.8 Priority Queues [Heaps]: Problems & Solutions

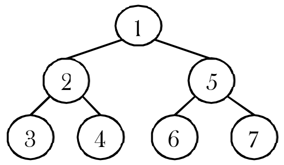
Problem-1 What are the minimum and maximum number of elements in a heap of height h?

Solution: Since heap is a complete binary tree (all levels contain full nodes except possibly the lowest level), it has at most $2^{h+1} - 1$ elements (if it is complete). This is because, to get maximum nodes, we need to fill all the h levels completely and the maximum number of nodes is nothing but the sum of all nodes at all h levels.

To get minimum nodes, we should fill the $h - 1$ levels fully and the last level with only one element. As a result, the minimum number of nodes is nothing but the sum of all nodes from $h - 1$ levels plus 1 (for the last level) and we get $2^h - 1 + 1 = 2^h$ elements (if the lowest level has just 1 element and all the other levels are complete).

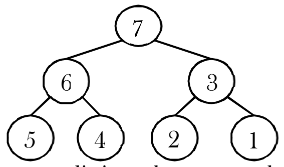
Problem-2 Is there a min-heap with seven distinct elements so that the preorder traversal of it gives the elements in sorted order?

Solution: Yes. For the tree below, preorder traversal produces ascending order.



Problem-3 Is there a max-heap with seven distinct elements so that the preorder traversal of it gives the elements in sorted order?

Solution: Yes. For the tree below, preorder traversal produces descending order.



Problem-4 Is there a min-heap/max-heap with seven distinct elements so that the inorder traversal of it gives the elements in sorted order?

Implementation

```
def mergeSort(A):
    if len(A)>1:
        mid = len(A)//2
        lefthalf = A[:mid]
        righthalf = A[mid:]
        mergeSort(lefthalf)
        mergeSort(righthalf)
        i = j = k = 0
        while i<len(lefthalf) and j<len(righthalf):
            if lefthalf[i]<righthalf[j]:
                A[k]=lefthalf[i]
                i=i+1
            else:
                A[k]=righthalf[j]
                j=j+1
            k=k+1
        while i<len(lefthalf):
            A[k]=lefthalf[i]
            i=i+1
            k=k+1
        while j<len(righthalf):
            A[k]=righthalf[j]
            j=j+1
            k=k+1
A = [534,246,933,127,277,321,454,565,220]
mergeSort(A)
print(A)
```

Analysis

In merge-sort the input array is divided into two parts and these are solved recursively. After solving the subarrays, they are merged by scanning the resultant subarrays. In merge sort, the comparisons occur during the merging step, when two sorted arrays are combined to output a single sorted array. During the merging step, the first available element of each array is compared and the lower value is appended to the output array. When either array runs out of values, the remaining elements of the opposing array are appended to the output array.

How do determine the complexity of merge-sort? We start by thinking about the three parts of divide-and-conquer and how to account for their running times. We assume that we're sorting a total of n elements in the entire array.

The divide step takes constant time, regardless of the subarray size. After all, the divide step just computes the midpoint mid of the indices $left$ and $right$. Recall that in big-Θ notation, we indicate constant time by Θ(1).

The conquer step, where we recursively sort two subarrays of approximately $\frac{n}{2}$ elements each, takes some amount of time, but we'll account for that time when we consider the subproblems. The combine step merges a total of n elements, taking $\Theta(n)$ time.

If we think about the divide and combine steps together, the Θ(1) running time for the divide step is a low-order term when compared with the $\Theta(n)$ running time of the combine step. So let's think of the divide and combine steps together as taking $\Theta(n)$ time. To make things more concrete, let's say that the divide and combine steps together take cn time for some constant c.

Let us assume $T(n)$ is the complexity of merge-sort with n elements. The recurrence for the merge-sort can be defined as:

$$T(n) = 2T(\frac{n}{2}) + \Theta(n)$$

Using master theorem, we could derive $T(n) = \Theta(\,nlogn)$

For merge-sort there is no running time difference between best, average and worse cases as the division of input arrays happen irrespective of the order of the elements. Above merge sort algorithm uses an auxiliary space of $O(n)$ for left and right subarrays together. Merge-sort is a recursive algorithm and each recursive step puts another frame on the run time stack. Sorting 32 items will take one more recursive step than 16 items, and it is in fact the size of the stack that is referred to when the space requirement is said to be $O(logn)$.

Worst case complexity : $\Theta(nlogn)$
Best case complexity : $\Theta(nlogn)$
Average case complexity : $\Theta(nlogn)$
Space complexity: $\Theta(logn)$ for runtime stack space and $O(n)$ for the auxiliary space

10.11 Quick Sort

Quick sort is the famous algorithm among comparison-based sorting algorithms. Like merge sort, quick sort uses divide-and-conquer technique, and so it's a recursive algorithm. The way that quick sort uses divide-and-conquer is a little different from how merge sort does. The quick sort uses divide and conquer technique to gain the same advantages as the merge sort, while not using additional storage. As a trade-off, however, it is possible that the list may not be divided into half. When this happens, we will see that the performance is diminished.

It sorts in place and no additional storage is required as well. The slight disadvantage of quick sort is that its worst-case performance is similar to the average performances of the bubble, insertion or selection sorts (i.e., O(n^2)).

Divide and conquer strategy

A quick sort first selects an element from the given list, which is called the $pivot$ value. Although there are many different ways to choose the pivot value, we will simply use the $first$ item in the list. The role of the pivot value is to assist with splitting the list into two sublists. The actual position where the pivot value belongs in the final sorted list, commonly called the $partition$ point, will be used to divide the list for subsequent calls to the quick sort.

All elements in the first sublist are arranged to be smaller than the $pivot$, while all elements in the second sublist are arranged to be larger than the $pivot$. The same partitioning and arranging process is performed repeatedly on the resulting sublists until the whole list of items are sorted.

Let us assume that array A is the list of elements to be sorted, and has the lower and upper bounds low and $high$ respectively. With this information, we can define the divide and conquer strategy as follows:

$Divide$: The list $A[low \ldots high]$ is partitioned into two non-empty sublists $A[low \ldots q]$ and $A[q+1 \ldots high]$, such that each element of $A[low \ldots q]$ is less than or equal to each element of $A[q+1 \ldots high]$. The index q is computed as part of partitioning procedure with the first element as $pivot$.

$Conquer$: The two sublists $A[low \ldots q]$ and $A[q+1 \ldots high]$ are sorted by recursive calls to quick sort.

Algorithm

The recursive algorithm consists of four steps:

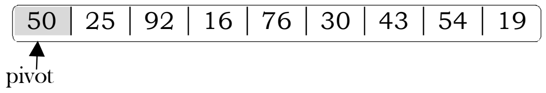
1) If there are one or no elements in the list to be sorted, return.
2) Pick an element in the list to serve as the $pivot$ point. Usually the first element in the list is used as a $pivot$.
3) Split the list into two parts - one with elements larger than the $pivot$ and the other with elements smaller than the $pivot$.
4) Recursively repeat the algorithm for both halves of the original list.

In the above algorithm, the important step is partitioning the list into two sublists. The basic steps to partition a list are:

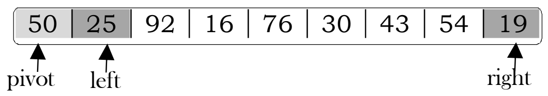
1. Select the first element as a $pivot$ in the list.
2. Start a pointer (the $left$ pointer) at the second item in the list.
3. Start a pointer (the $right$ pointer) at the last item in the list.
4. While the value at the $left$ pointer in the list is lesser than the $pivot$ value, move the $left$ pointer to the right (add 1). Continue this process until the value at the $left$ pointer is greater than or equal to the $pivot$ value.
5. While the value at the $right$ pointer in the list is greater than the $pivot$ value, move the $right$ pointer to the left (subtract 1). Continue this process until the value at the $right$ pointer is lesser than or equal to the $pivot$ value.
6. If the $left$ pointer value is greater than or equal to the $right$ pointer value, then swap the values at these locations in the list.
7. If the $left$ pointer and $right$ pointer don't meet, go to step 1.

Example

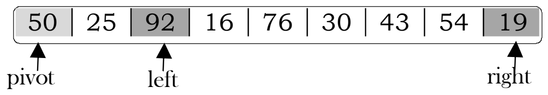
Following example shows that 50 will serve as our first pivot value. The partition process will happen next. It will find the $partition$ point and at the same time move other items to the appropriate side of the list, either lesser than or greater than the $pivot$ value.



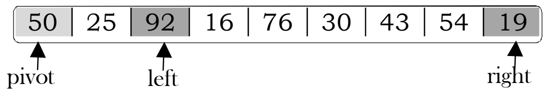
Partitioning begins by locating two position markers—let's call them $left$ and $right$—at the beginning and end of the remaining items in the list (positions 1 and 8 in figure). The goal of the partition process is to move items that are on the wrong side with respect to the pivot value while converging on the split point also. The figure given below shows this process as we locate the position of 50.



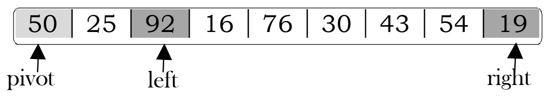
25 < 50, move $left$ pointer to right:



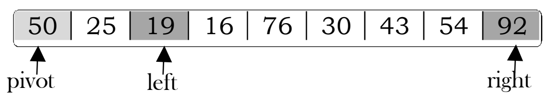
92 > 50, stop from moving $left$ pointer:



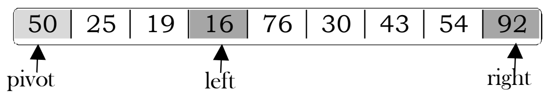
19 < 50, stop from moving $right$ pointer:



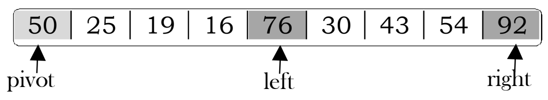
Swap 19 and 92:



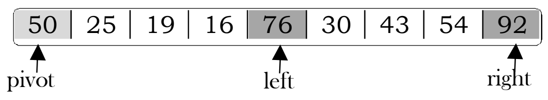
Now, continue moving left and right pointers. 19 < 50, move $left$ pointer to right:



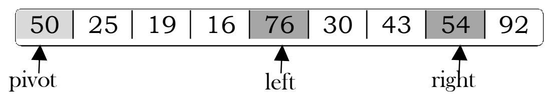
16 < 50, move $left$ pointer to right:



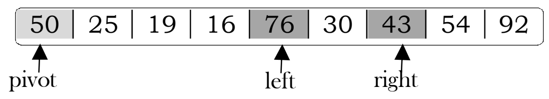
76 > 50, stop from moving $left$ pointer:



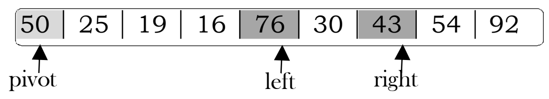
92 > 50, move $right$ pointer to left:



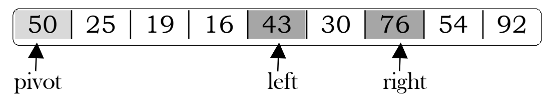
54 > 50, move $right$ pointer to left:



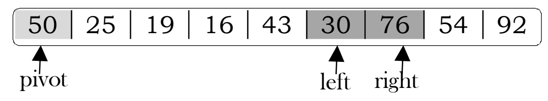
43 < 50, stop from moving $right$ pointer:



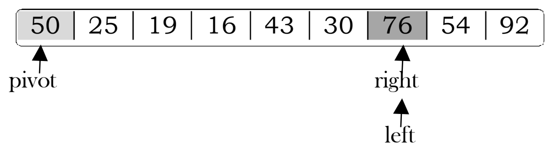
Swap 76 and 43:



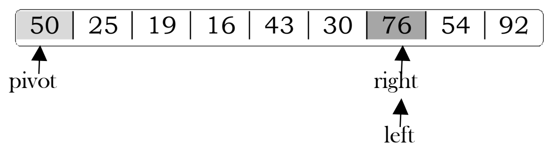
43 < 50, move $left$ pointer to right:



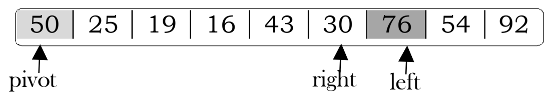
30 < 50, move $left$ pointer to right:



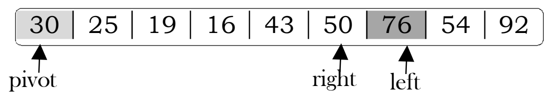
76 > 50, stop from moving $left$ pointer:



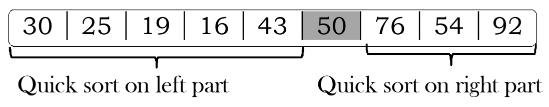
76 > 50, move $right$ pointer to left:



At the point where right becomes less than left, we stop. The position of right is now the $partition$ point. The $pivot$ value can be exchanged with the contents of the $partition$ point. In addition, all the items to the left of the split point are less than the pivot value, and all the items to the right of the split point are greater than the pivot value. Now, we can exchange these two elements 50 and 30. Element 50 is now in correct position.



The list can now be divided at the partition point and the quick sort can be invoked recursively on the two halves.



Repeat the process for the two sublists.

Implementation

```
def quick_sort(A,low,high):
   if low<high:
       partition_point = partition(A,low,high)
       quick_sort(A,low,partition_point-1)
       quick_sort(A,partition_point+1,high)
def partition(A,low,high):
   pivot = A[low]
   left = low+1
   right = high
   done = False
   while not done:
       while left <= right and A[left] <= pivot:
           left = left + 1
       while A[right] >= pivot and right >= left:
           right = right -1
       if right < left:
           done = True
       else:
           temp = A[left]
           A[left] = A[right]
           A[right] = temp
   temp = A[low]
   A[low] = A[right]
   A[right] = temp
   return right
A = [50,25,92,16,76,30,43,54,19]
quick_sort(A,0,len(A)-1)
              print(A)
```

Analysis

Let us assume that T(n) be the complexity of Quick sort and also assume that all elements are distinct. Recurrence for $T(n)$ depends on two subproblem sizes which depend on partition element. If pivot is i^{th} smallest element then exactly $(i - 1)$ items will be in left part and $(n - i)$ in right part. Let us call it as i −split. Since each element has equal probability of selecting it as pivot the probability of selecting i^{th} element is $\frac{1}{n}$.

Best Case: Each partition splits array in halves and gives

$$T(n) = 2T(n/2) + \Theta(n) = \Theta(n\log n), \text{[using } \textit{Divide and Conquer} \text{ master theorem]}$$

Worst Case: Each partition gives unbalanced splits and we get

$$T(n) = T(n - 1) + \Theta(n) = \Theta(n^2)[\textit{using Subtraction and Conquer master theorem}]$$

The worst-case occurs when the list is already sorted and last element chosen as pivot.

Average Case: In the average case of Quick sort, we do not know where the split happens. For this reason, we take all possible values of split locations, add all their complexities and divide with n to get the average case complexity.

$$T(n) = \sum_{i=1}^{n} \frac{1}{n}(runtime\ with\ i - split) + n + 1$$

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

//since we are dealing with best case we can assume $T(n-i)$ and $T(i-1)$ are equal

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

$$= \frac{2}{n}\sum_{i=0}^{n-1} T(i) + n + 1$$

Multiply both sides by n.

$$n\mathrm{T}(n) = 2\sum_{i=0}^{n-1} T(i) + n^2 + n$$

Same formula for $n - 1$.

$$(n-1)T(n-1) = 2\sum_{i=0}^{n-2} T(i) + (n-1)^2 + (n-1)$$

Subtract the $n - 1$ formula from n.

$$nT(n) - (n-1)T(n-1) = 2\sum_{i=0}^{n-1} T(i) + n^2 + n - (2\sum_{i=0}^{n-2} T(i) + (n-1)^2 + (n-1))$$

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

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

Divide with $n(n+1)$.

$$\begin{aligned}
\frac{T(n)}{n+1} &= \frac{T(n-1)}{n} + \frac{2}{n+1} \\
&= \frac{T(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\
&. \\
&. \\
&= \mathrm{O}(1) + 2\textstyle\sum_{i=3}^{n}\frac{1}{i} \\
&= \mathrm{O}(1) + \mathrm{O}(2logn) \\
\frac{T(n)}{n+1} &= \mathrm{O}(logn) \\
T(n) &= \mathrm{O}((n+1)\ logn) = \mathrm{O}(nlogn)
\end{aligned}$$

Time Complexity, $T(n) = \mathrm{O}(nlogn)$.

Performance

Worst case Complexity: $\mathrm{O}(n^2)$
Best case Complexity: $\mathrm{O}(nlogn)$
Average case Complexity: $\mathrm{O}(nlogn)$
Worst case space Complexity: $\mathrm{O}(1)$

Randomized Quick sort

In average-case behavior of Quick sort, we assume that all permutations of the input numbers are equally likely. However, we cannot always expect it to hold. We can add randomization to an algorithm in order to reduce the probability of getting worst case in Quick sort.

There are two ways of adding randomization in Quick sort: either by randomly placing the input data in the array or by randomly choosing an element in the input data for pivot. The second choice is easier to analyze and implement. The change will only be done at the $partition$ algorithm.

In normal Quick sort, $pivot$ element was always the leftmost element in the list to be sorted. Instead of always using $A[low]$ as $pivot$, we will use a randomly chosen element from the subarray $A[low..high]$ in the randomized version of Quick sort. It is done by exchanging element $A[low]$ with an element chosen at random from $A[low..high]$. This ensures that the $pivot$ element is equally likely to be any of the $high - low + 1$ elements in the subarray.

Since the pivot element is randomly chosen, we can expect the split of the input array to be reasonably well balanced on average. This can help in preventing the worst-case behavior of quick sort which occurs in unbalanced partitioning.

Even though the randomized version improves the worst case complexity, its worst case complexity is still $\mathrm{O}(n^2)$. One way to improve $Randomized - Quick\ sort$ is to choose the pivot for partitioning more carefully than by picking a random element from the array. One common approach is to choose the pivot as the median of a set of 3 elements randomly selected from the array.

10.12 Tree Sort

Tree sort uses a binary search tree. It involves scanning each element of the input and placing it into its proper position in a binary search tree. This has two phases:

- First phase is creating a binary search tree using the given array elements.
- Second phase is traversing the given binary search tree in inorder, thus resulting in a sorted array.

Performance

The average number of comparisons for this method is $\mathrm{O}(nlogn)$. But in worst case, the number of comparisons is reduced by $\mathrm{O}(n^2)$, a case which arises when the sort tree is skew tree.

Selection Algorithms [Medians]

CHAPTER 12

12.1 What are Selection Algorithms?

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 findingthe k^{th} order statistic, there are multiple solutions which provide different complexities, and in this chapter we will enumerate those possibilities.

12.2 Selection by Sorting

A selection problem can be converted to a sorting problem. In this method, we first sort the input elements and then get the desired element. It is efficient if we want to perform many selections.

For example, let us say we want to get the minimum element. After sorting the input elements we can simply return the first element (assuming the array is sorted in ascending order). Now, if we want to find the second smallest element, we can simply return the second element from the sorted list.

That means, for the second smallest element we are not performing the sorting again. The same is also the case with subsequent queries. Even if we want to get k^{th} smallest element, just one scan of the sorted list is enough to find the element (or we can return the k^{th}-indexed value if the elements are in the array).

From the above discussion what we can say is, with the initial sorting we can answer any query in one scan, $O(n)$. In general, this method requires $O(nlogn)$ time (for *sorting*), where n is the length of the input list. Suppose we are performing n queries, then the average cost per operation is just $\frac{n\,logn}{n} \approx O(logn)$. This kind of analysis is called *amortized* analysis.

12.3 Partition-based Selection Algorithm

For the algorithm check Problem-6. This algorithm is similar to Quick sort.

12.4 Linear Selection Algorithm - Median of Medians Algorithm

Worst-case performance	$O(n)$
Best-case performance	$O(n)$
Worst-case space complexity	$O(1)$ auxiliary

Refer to Problem-11.

12.5 Finding the K Smallest Elements in Sorted Order

For the algorithm check Problem-6. This algorithm is similar to Quick sort.

12.6 Selection Algorithms: Problems & Solutions

Problem-1 Find the largest element in an array A of size n.

Solution: Scan the complete array and return the largest element.

```
def findLargestInArray(A):
    max = 0
    for number in A:
        if number > max:
            max = number
        return max
    print(findLargestInArray([2,1,5,234,3,44,7,6,4,5,9,11,12,14,13]))
```

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

Note: Any deterministic algorithm that can find the largest of n keys by comparison of keys takes at least $n - \mathbf{1}$ comparisons.

Problem-2 Find the smallest and largest elements in an array A of size n.

Solution:

```
def findSmallestAndLargestInArray(A):
        max = 0
        min = 0
        for number in A:
                if number > max:
                        max = number
                elif number < min:
                        min = number
        print("Smallest: %d", min)
        print("Largest: %d", max )
findSmallestAndLargestInArray([2,1,5,234,3,44,7,6,4,5,9,11,12,14,13])
```

Time Complexity - O(n). Space Complexity - O(1). The worst-case number of comparisons is $2(n-1)$.

Problem-3 Can we improve the previous algorithms?

Solution: Yes. We can do this by comparing in pairs.

```
def findMinMaxWithPairComparisons(A):
        ## for an even-sized Aray
        _max = A[0]
        _min = A[0]
        for indx in range(0, len(A), 2):
            first = A[indx]
            second = A[indx+1]
            if (first < second):
                if first < _min: _min = first
                if second > _max: _max = second
            else:
                if second < _min: _min = second
                if first > _max: _max = first
        print(_min)
        print(_max)
findMinMaxWithPairComparisons([2,1,5,234,3,44,7,6,4,5,9,11,12,14,13,19])
```

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

Number of comparisons: $\begin{cases} \frac{3n}{2} - 2, if\ n\ is\ even \\ \frac{3n}{2} - \frac{3}{2}\ if\ n\ is\ odd \end{cases}$

Summary:

Straightforward comparison – $2(n-1)$ comparisons
Compare for min only if comparison for max fails
Best case: increasing order – $n-1$ comparisons
Worst case: decreasing order – $2(n-1)$ comparisons
Average case: $3n/2 - 1$ comparisons

Note: For divide and conquer techniques refer to *Divide and Conquer* chapter.

Problem-4 Give an algorithm for finding the second largest element in the given input list of elements.

Solution: Brute Force Method

Algorithm:

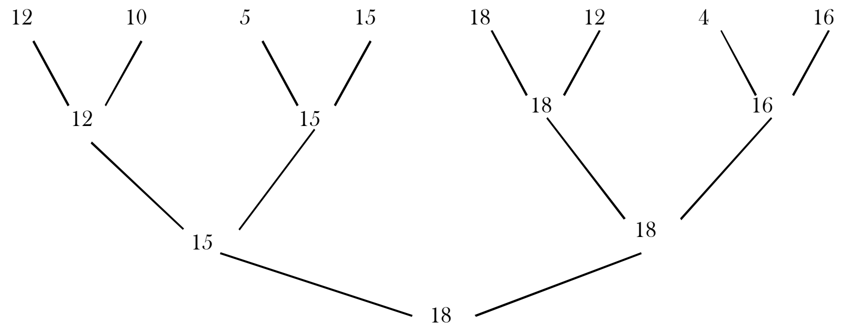
- Find largest element: needs $n-1$ comparisons
- Delete (discard) the largest element
- Again find largest element: needs $n-2$ comparisons

Total number of comparisons: $n-1+n-2 = 2n-3$

Problem-5 Can we reduce the number of comparisons in Problem-4 solution?

Solution: The Tournament method: For simplicity, assume that the numbers are distinct and that n is a power of 2. We pair the keys and compare the pairs in rounds until only one round remains. If the input has eight keys, there are four comparisons in the first round, two in the second, and one in the last. The winner of the last round is the largest key. The figure below shows the method.

The tournament method directly applies only when n is a power of 2. When this is not the case, we can add enough items to the end of the array to make the array size a power of 2. If the tree is complete then the maximum height of the tree is $logn$. If we construct the complete binary tree, we need $n - 1$ comparisons to find the largest. The second largest key has to be among the ones that were lost in a comparison with the largest one. That means, the second largest element should be one of the opponents of the largest element. The number of keys that are lost to the largest key is the height of the tree, i.e. $logn$ [if the tree is a complete binary tree]. Then using the selection algorithm to find the largest among them, take $logn - 1$ comparisons. Thus the total number of comparisons to find the largest and second largest keys is $n + logn - 2$.



```
def secondSmallestInArray(A):
        comparisonCount = 0
        # indexes that are to be compared
        idx = range(0,len(A))
        # list of knockout for all elements
        knockout = [[] for i in idx]
        # play tournaments, until we have only one node left
        while len(idx) > 1:
            # index of nodes that win this tournament
            idx1 = []
            # nodes in idx odd, if yes then last automatically goes to next round
            odd = len(idx) % 2
            # iterate over even indexes, as we do a paired tournament
            for i in xrange(0, len(idx) - odd, 2):
                    firstIndex = idx[i]
                    secondIndex = idx[i+1]
                    comparisonCount += 1
                    # perform tournament
                    if A[firstIndex] < A[secondIndex]:
                        # firstIndex qualifies for next round
                        idx1.append(firstIndex)
                        # add A[secondIndex] to knockout list of firstIndex
                        knockout[firstIndex].append(A[secondIndex])
                    else:
                        idx1.append(secondIndex)
                        knockout[secondIndex].append(A[firstIndex])
            if odd == 1:
                    idx1.append(idx[i+2])
            # perform new tournament
            idx = idx1
        print ("Smallest element =", A[idx[0]])
        print ("Total comparisons =", comparisonCount)
        print ("Nodes knocked off by the smallest =", knockout[idx[0]], "\n")
        # compute second smallest
        a = knockout[idx[0]]
        if len(a) > 0:
            v = a[0]
            for i in xrange(1,len(a)):
                    comparisonCount += 1
                    if v > a[i]: v = a[i]
            print ("Second smallest element =", v)
```

```
          print ("Total comparisons =", comparisonCount)
    A = [2, 4, 3, 7, 3, 0, 8, 4, 11, 1]
    print(secondSmallestInArray(A))
```

Problem-6 Find the k-smallest elements in an array S of n elements using partitioning method.

Solution: Brute Force Approach: Scan through the numbers k times to have the desired element. This method is the one used in bubble sort (and selection sort), every time we find out the smallest element in the whole sequence by comparing every element. In this method, the sequence has to be traversed k times. So the complexity is O($n \times k$).

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

Solution: Yes. Sort and take the first k elements.

1. Sort the numbers.
2. Pick the first k elements.

The time complexity calculation is trivial. Sorting of n numbers is of O($nlogn$) and picking k elements is of O(k). The total complexity is O($nlogn + k$) $=$ O($nlogn$).

Problem-8 Can we use the *tree sorting* technique for solving Problem-6?

Solution: Yes.

1. Insert all the elements in a binary search tree.
2. Do an InOrder traversal and print k elements which will be the smallest ones. So, we have the k smallest elements.

The cost of creation of a binary search tree with n elements is O($nlogn$) and the traversal up to k elements is O(k). Hence the complexity is O($nlogn + k$) $=$ O($nlogn$).

Disadvantage: If the numbers are sorted in descending order, we will be getting a tree which will be skewed towards the left. In that case, the construction of the tree will be $0 + 1 + 2 + \ldots + (n-1) = \frac{n(n-1)}{2}$ which is O(n^2). To escape from this, we can keep the tree balanced, so that the cost of constructing the tree will be only $nlogn$.

Problem-9 Can we improve the *tree sorting* technique for solving Problem-6?

Solution: Yes. Use a smaller tree to give the same result.

1. Take the first k elements of the sequence to create a balanced tree of k nodes (this will cost $klogk$).
2. Take the remaining numbers one by one, and
   a. If the number is larger than the largest element of the tree, return.
   b. If the number is smaller than the largest element of the tree, remove the largest element of the tree and add the new element. This step is to make sure that a smaller element replaces a larger element from the tree. And of course the cost of this operation is $logk$ since the tree is a balanced tree of k elements.

Once Step 2 is over, the balanced tree with k elements will have the smallest k elements. The only remaining task is to print out the largest element of the tree.

Time Complexity:

1. For the first k elements, we make the tree. Hence the cost is $klogk$.
2. For the rest $n - k$ elements, the complexity is O($logk$).

Step 2 has a complexity of $(n-k)\,logk$. The total cost is $klogk + (n-k)\,logk = nlogk$ which is O($nlogk$). This bound is actually better than the ones provided earlier.

Problem-10 Can we use the partitioning technique for solving Problem-6?

Solution: Yes.

Algorithm

1. Choose a pivot from the array.
2. Partition the array so that: $A[low...pivotpoint - 1] <= pivotpoint <= A[pivotpoint + 1..high]$.
3. if $k < pivotpoint$ then it must be on the left of the pivot, so do the same method recursively on the left part.
4. if $k = pivotpoint$ then it must be the pivot and print all the elements from low to $pivotpoint$.
5. if $k > pivotpoint$ then it must be on the right of pivot, so do the same method recursively on the right part.

The input data can be any iterable. The randomization of pivots makes the algorithm perform consistently even with unfavorable data orderings.

```
import random
def kthSmallest(data, k):
  "Find the nth rank ordered element (the least value has rank 0)."
  data = list(data)
  if not 0 <= k < len(data):
    raise ValueError('not enough elements for the given rank')
  while True:
    pivot = random.choice(data)
```

```
        pcount = 0
        under, over = [], []
        uappend, oappend = under.append, over.append
        for elem in data:
            if elem < pivot:
                uappend(elem)
            elif elem > pivot:
                oappend(elem)
            else:
                pcount += 1
        if k < len(under):
            data = under
        elif k < len(under) + pcount:
            return pivot
        else:
            data = over
            k -= len(under) + pcount
print(kthSmallest([2,1,5,234,3,44,7,6,4,5,9,11,12,14,13], 5))
```

Time Complexity: $O(n^2)$ in worst case as similar to Quicksort. Although the worst case is the same as that of Quicksort, this performs much better on the average [$O(nlogk)$ – Average case].

Problem-11 Find the k^{th}-smallest element in an array S of n elements in best possible way.

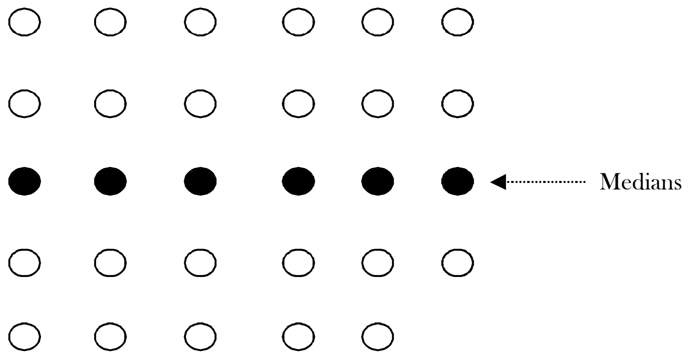
Solution: This problem is similar to Problem-6 and all the solutions discussed for Problem-6 are valid for this problem. The only difference is that instead of printing all the k elements, we print only the k^{th}element. We can improve the solution by using the *median of medians* algorithm. Median is a special case of the selection algorithm. The algorithm Selection(A, k) to find the k^{th} smallest element from set A of n elements is as follows:

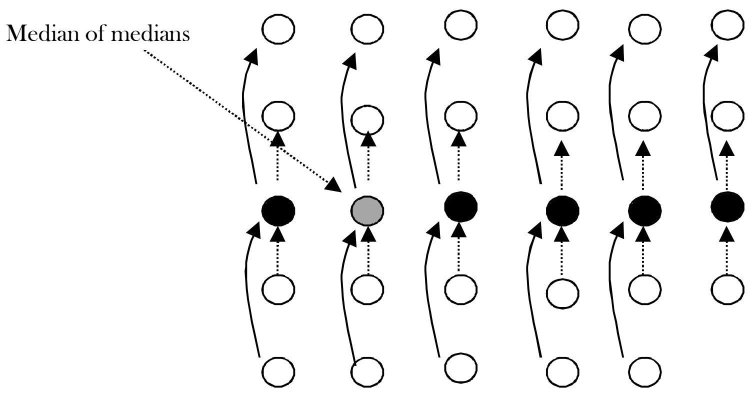
Algorithm: $Selection$(A, k)

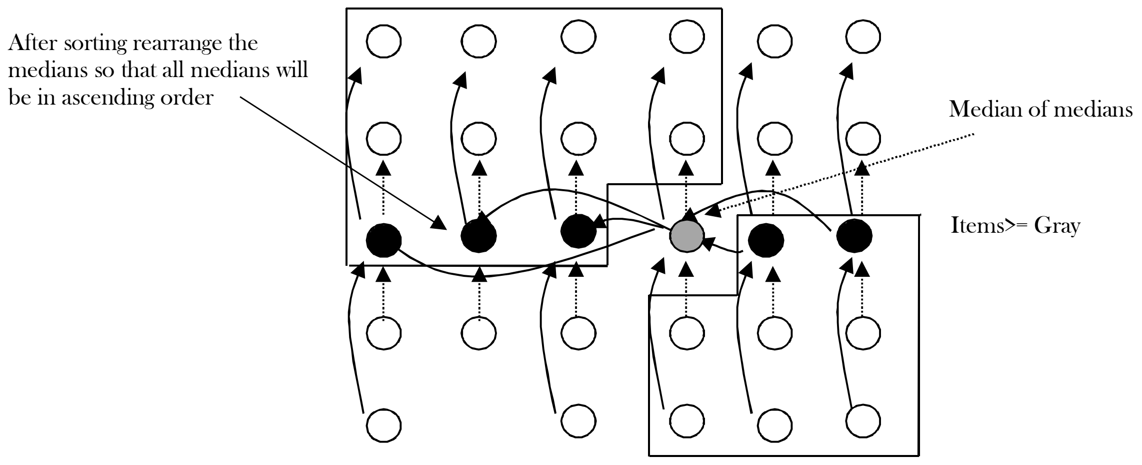
```
1.  Partition $A$ into $ceil\left(\frac{length(A)}{5}\right)$ groups, with each group having five items (the last group may have fewer items).
2.  Sort each group separately (e.g., insertion sort).
3.  Find the median of each of the $\frac{n}{5}$ groups and store them in some array (let us  say $A'$).
4.  Use $Selection$ recursively to find the median of $A'$ (median of medians). Let us asay the median of medians is $m$.
            $m = Selection(A', \frac{\frac{length(A)}{5}}{2})$
5.  Let $q\ =\ \#$ elements of $A$ smaller than $m$
6.  If($k\ ==\ q\ +\ 1$)
        return $m$
    # Partition with pivot
7.  Else partition $A$ into $X$ and $Y$
    $X$ = {items smaller than $m$}
    $Y$ = {items larger than $m$}

    # Next,form a subproblem
8.  If($k\ < q\ +\ 1$)
        return Selection(X, k)
9.  Else
        return Selection(Y, k – (q+1))
```

Before developing recurrence, let us consider the representation of the input below. In the figure, each circle is an element and each column is grouped with 5 elements. The black circles indicate the median in each group of 5 elements. As discussed, sort each column using constant time insertion sort.






After sorting rearrange the medians so that all medians will be in ascending order
Median of medians
Items>= Gray

In the figure above the gray circled item is the median of medians (let us call this m). It can be seen that at least $1/2$ of 5 element group medians $\leq m.$ Also, these $1/2$ of 5 element groups contribute 3 elements that are $\leq\ m$ except 2 groups [last group which may contain fewer than 5 elements, and other group which contains m]. Similarly, at least $1/2$ of 5 element groups contribute 3 elements that are $\geq\ m$ as shown above. $1/2$ of 5 element groups contribute 3 elements, except 2 groups gives: $3(\lceil\frac{1}{2}\lceil\frac{n}{5}\rceil\rceil$-2) $\approx \frac{3n}{10} - 6$. The remaining are $n\ -(\frac{3n}{10} - 6) \approx \frac{7n}{10} + 6$. Since $\frac{7n}{10} + 6$ is greater than $\frac{3n}{10} - 6$ we need to consider $\frac{7n}{10} + 6$ for worst.

Components in recurrence:

- In our selection algorithm, we choose m, which is the median of medians, to be a pivot, and partition A into two sets X and Y. We need to select the set which gives maximum size (to get the worst case).
- The time in function $Selection$ when called from procedure $partition.$ The number of keys in the input to this call to $Selection$ is $\frac{n}{5}$.
- The number of comparisons required to partition the array. This number is $length(S)$, let us say $n.$

We have established the following recurrence: $T(n)\ = T\left(\frac{n}{5}\right) + \Theta(n) +\ Max\{T(X), T(Y)\}$

From the above discussion we have seen that, if we select median of medians m as pivot, the partition sizes are: $\frac{3n}{10} - 6$ and $\frac{7n}{10} + 6$. If we select the maximum of these, then we get:

$$\begin{aligned} T(n) &= T\left(\frac{n}{5}\right) + \Theta(n) + T\left(\frac{7n}{10} + 6\right) \\ &\approx T\left(\frac{n}{5}\right) + \Theta(n) + T\left(\frac{7n}{10}\right) + O(1) \\ &\leq c\frac{7n}{10} + c\frac{n}{5} + \Theta(n) + O(1) \end{aligned}$$

Finally, $T(n)\ = \Theta(n)$.

```
CHUNK_SIZE = 5
def kthByMedianOfMedian(unsortedList, k):
    if len(unsortedList) <= CHUNK_SIZE:
        return getKth(unsortedList, k)
    chunks = splitIntoChunks(unsortedList, CHUNK_SIZE)
    mediansList = []
    for chunk in chunks:
```

CHAPTER

15

STRING ALGORITHMS

15.1 Introduction

To understand the importance of string algorithms let us consider the case of entering the URL (Uniform 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 the command line interface (in both $Windows$ and $UNIX$). After typing the prefix of the directory name, if we press the tab button, we 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 a $string\ matching$ problem. After discussing various string matching algorithms, we will look at different data structures for storing strings.

15.2 String Matching Algorithms

In this section, we concentrate on checking whether a pattern P is a substring of another string T (T stands for text) or not. Since we are trying to check a fixed string P, sometimes these algorithms are called $exact\ string\ matching$ algorithms. To simplify our discussion, let us assume that the length of given text T is n and the length of the pattern P which we are trying to match has the length m. That means, T has the characters from 0 to $n - 1$ ($T[0 \dots n - 1]$) and P has the characters from 0 to $m - 1$ ($P[0 \dots m - 1]$). This algorithm is implemented in $C++$ as $strstr()$.

In the subsequent sections, we start with the brute force method and gradually move towards better algorithms.

- Brute Force Method
- Rabin-Karp String Matching Algorithm
- String Matching with Finite Automata
- KMP Algorithm
- Boyer-Moore Algorithm
- Suffix Trees

15.3 Brute Force Method

In this method, for each possible position in the text T we check whether the pattern P matches or not. Since the length of T is n, we have $n - m + 1$ possible choices for comparisons. This is because we do not need to check the last $m - 1$ locations of T as the pattern length is m. The following algorithm searches for the first occurrence of a pattern string P in a text string T..

Algorithm

```
def strStrBruteForce(str, pattern):
    if not pattern: return 0
    for i in range(len(str)-len(pattern)+1):
        stri = i; patterni = 0
        while stri < len(str) and patterni < len(pattern) and str[stri] == pattern[patterni]:
            stri += 1
            patterni += 1
        if patterni == len(pattern): return i
    return -1

print (strStrBruteForce("xxxxyzabcdabcdefabc", "abc"))
```

Time Complexity: O($(n - m + 1) \times m$) ≈O($n \times m$). Space Complexity: O(1).

15.4 Rabin-Karp String Matching Algorithm

In this method, we will use the hashing technique and instead of checking for each possible position in T, we check only if the hashing of P and the hashing of m characters of T give the same result.

Initially, apply the hash function to the first m characters of T and check whether this result and P's hashing result is the same or not. If they are not the same, then go to the next character of T and again apply the hash function to m characters (by starting at the second character). If they are the same then we compare those m characters of T with P.

Selecting Hash Function

At each step, since we are finding the hash of m characters of T, we need an efficient hash function. If the hash function takes O(m) complexity in every step, then the total complexity is O($n \times m$). This is worse than the brute force method because first we are applying the hash function and also comparing.

Our objective is to select a hash function which takes O(1) complexity for finding the hash of m characters of T every time. Only then can we reduce the total complexity of the algorithm. If the hash function is not good (worst case), the complexity of the Rabin-Karp algorithm is O($n - m + 1) \times m$) ≈O($n \times m$). If we select a good hash function, the complexity of the Rabin-Karp algorithm complexity is O($m + n$). Now let us see how to select a hash function which can compute the hash of m characters of T at each step in O(1).

For simplicity, let's assume that the characters used in string T are only integers. That means, all characters in $T \in \{0, 1, 2, \ldots, 9\}$. Since all of them are integers, we can view a string of m consecutive characters as decimal numbers. For example, string $'61815'$ corresponds to the number 61815. With the above assumption, the pattern P is also a decimal value, and let us assume that the decimal value of P is p. For the given text $T[0..n-1]$, let $t(i)$ denote the decimal value of length$-m$ substring $T[i..i+m-1]$ for $i = 0,1,\ldots,n-m-1$. So, $t(i) == p$ if and only if $T[i..i+m-1] == P[0..m-1]$.

We can compute p in O(m) time using Horner's Rule as:

$$p = P[m-1] + 10(P[m-2] + 10(P[m-3] + \ldots + 10\,(P[1] + 10\,P[0])\ldots))$$

The code for the above assumption is:

```
value = 0
for i in range (0, m-1):
    value = value * 10
    value = value + P[i]
```

We can compute all $t(i)$, for $i = 0,1,\ldots,n-m-1$ values in a total of O(n) time. The value of $t(0)$ can be similarly computed from $T[0..m-1]$ in O(m) time. To compute the remaining values $t(0), t(1), \ldots, t\,(n-m-1)$, understand that $t(i+1)$ can be computed from $t(i)$ in constant time.

$$t(i+1) = 10 * (t(i) - 10^{m-1} * T[i]) + T[i + m - 1]$$

For example, if T = "123456" and $m = 3$

$$t(0) = 123$$
$$t(1) = 10 * (123 - 100 * 1) + 4 = 234$$

Step by Step explanation

First : remove the first digit : $123 - 100 * 1 = 23$
Second: Multiply by 10 to shift it : $23 * 10 = 230$
Third: Add last digit : $230 + 4 = 234$

The algorithm runs by comparing, $t(i)$ with p. When $t(i) == p$, then we have found the substring P in T, starting from position i..

```
def RabinKarp(text, pattern):
    if pattern == None or text == None:
        return -1
    if pattern == "" or text == "":
        return -1
    if len(pattern) > len(text):
        return -1
    hashText = Hash(text, len(pattern))
    hashPattern = Hash(pattern, len(pattern))
    hashPattern.update()
    for i in range(len(text)-len(pattern)+1):
        if hashText.hashedValue() == hashPattern.hashedValue():
            if hashText.text() == pattern:
                return i
            hashText.update()
        return -1
    class Hash:
        def __init__(self, text, size):
            self.str  = text
            self.hash = 0
            for i in xrange(0, size):
                self.hash += ord(self.str[i])
            self.init = 0
            self.end  = size
        def update(self):
            if self.end <= len(self.str) -1:
                self.hash -= ord(self.str[self.init])
                self.hash += ord(self.str[self.end])
                self.init += 1
                self.end  += 1
        def hashedValue(self):
            return self.hash
        def text(self):
            return self.str[self.init:self.end]
    print (RabinKarp("3141592653589793", "26"))
```

15.5 String Matching with Finite Automata

In this method we use the finite automata which is the concept of the Theory of Computation (ToC). Before looking at the algorithm, first let us look at the definition of finite automata.

Finite Automata

A finite automaton F is a 5-tuple $(Q, q_0, A, \sum, \delta)$, where

- Q is a finite set of states
- $q_0 \in Q$ is the start state
- $A \subseteq Q$ is a set of accepting states
- $\sum$ is a finite input alphabet
- δ is the transition function that gives the next state for a given current state and input

How does Finite Automata Work?

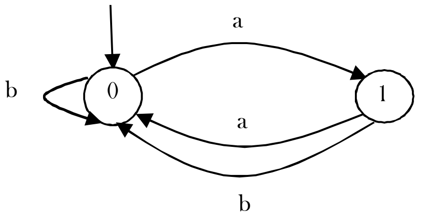
- The finite automaton F begins in state q_0
- Reads characters from $\sum$ one at a time
- If F is in state q and reads input character a, F moves to state $\delta(q, a)$
- At the end, if its state is in A, then we say, F accepted the input string read so far
- If the input string is not accepted it is called the rejected string

Example: Let us assume that $Q = \{0,1\}, q_0 = 0, A = \{1\}, \Sigma = \{a, b\}$. $\delta(q, a)$ as shown in the transition table/diagram. This accepts strings that end in an odd number of $a's$; e.g., $abbaaa$ is accepted, aa is rejected.

Input

State	a	b
0	1	0
1	0	0

Transition Function/Table



Important Notes for Constructing the Finite Automata

For building the automata, first we start with the initial state. The FA will be in state k if k characters of the pattern have been matched. If the next text character is equal to the pattern character c, we have matched $k + 1$ characters and the FA enters state $k + 1$. If the next text character is not equal to the pattern character, then the FA go to a state $0, 1, 2, \ldots, or\ k$, depending on how many initial pattern characters match the text characters ending with c.

Matching Algorithm

Now, let us concentrate on the matching algorithm.

- For a given pattern $P[0..m-1]$, first we need to build a finite automaton F
    - The state set is $Q = \{0, 1, 2, \ldots, m\}$
    - The start state is 0
    - The only accepting state is m
    - Time to build F can be large if Σ is large
- Scan the text string $T[0..n-1]$ to find all occurrences of the pattern $P[0..m-1]$
- String matching is efficient: $\Theta(n)$
    - Each character is examined exactly once
    - Constant time for each character
    - But the time to compute δ (transition function) is $O(m|\Sigma|)$. This is because δ has $O(m|\Sigma|)$ entries. If we assume $|\Sigma|$ is constant then the complexity becomes $O(m)$.

Algorithm:

```
# Input: Pattern string P[0..m-1], δ and F   # Goal: All valid shifts displayed
def finiteAutomataStringMatcher(P,m, F, δ):
    q = 0
    for i in range(0,m):
     q = δ(q,T[i])
        if(q == m):
            print("Pattern occurs with shift: ", i-m)
```

Time Complexity: $O(m)$.

15.6 KMP Algorithm

As before, let us assume that T is the string to be searched and P is the pattern to be matched. This algorithm was presented by Knuth, Morris and Pratt. It takes $O(n)$ time complexity for searching a pattern. To get $O(n)$ time complexity, it avoids the comparisons with elements of T that were previously involved in comparison with some element of the pattern P.

The algorithm uses a table and in general we call it *prefix function* or *prefix table* or *fail function* F. First we will see how to fill this table and later how to search for a pattern using this table. The prefix function F for a pattern stores the knowledge about how the pattern matches against shifts of itself. This information can be used to avoid useless shifts of the pattern P. It means that this table can be used for avoiding backtracking on the string TT.

Prefix Table

```
def prefixTable(pattern):
    m = len(pattern)
    F = [0] * m
    k = 0
    for q in range(1, m):
        while k > 0 and pattern[k] != pattern[q]:
            k = F[k - 1]
        if pattern[k] == pattern[q]:
            k = k + 1
        F[q] = k
    return F
```

As an example, assume that $P = a\ b\ a\ b\ a\ c\ a$. For this pattern, let us follow the step-by-step instructions for filling the prefix table F. Initially: $m = length[P] = 7, F[0] = 0$ and $F[1] = 0$.

Step 1: $i = 1, j = 0, \mathrm{F}[1] = 0$

	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0					

Step 2: $i = 2, j = 0, \mathrm{F}[2] = 1$

	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0	1				

Step 3: $i = 3, j = 1, \mathrm{F}[3] = 2$

	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0	1	2			

Step 4: $i = 4, j = 2, \mathrm{F}[4] = 3$

	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0	1	2	3		

Step 5: $i = 5, j = 3, \text{F}[5] = 1$

	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0	1	2	3	0	

Step 6: $i = 6, j = 1, \text{F}[6] = 1$

	0	1	2	3	4	5	6
P	a	b	a	b	a	c	a
F	0	0	1	2	3	0	1

At this step the filling of the prefix table is complete.

Matching Algorithm

The KMP algorithm takes pattern P, string T and prefix function F as input, and finds a match of P in T.

```
def KMP(text, pattern):
    n = len(text); m = len(pattern)
    F = prefixTable(pattern)
    q = 0
    for i in range(n):
        while q > 0 and pattern[q] != text[i]:
            q = F[q - 1]
        if pattern[q] == text[i]:
            q = q + 1
        if q == m:
            return i - m + 1
    return -1
print (KMP("bacbababacaca", "ababaca"))
```

Time Complexity: $\text{O}(m + n)$, where m is the length of the pattern and n is the length of the text to be searched. Space Complexity: $\text{O}(m)$.

Now, to understand the process let us go through an example. Assume that $T = b\,a\,c\,b\,a\,b\,a\,b\,a\,b\,a\,c\,a\,c\,a$ & $P = a\,b\,a\,b\,a\,c\,a$. Since we have already filled the prefix table, let us use it and go to the matching algorithm. Initially: $n = size\ of\ T = 15$; $m = size\ of\ P = 7$.

Step 1: $i = 0,\ j = 0$, comparing $P[0]$ with $T[0]$. $P[0]$ does not match with $T[0]$. P will be shifted one position to the right.

T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P	a	b	a	b	a	c	a								

Step 2: $i = 1,\ j = 0$, comparing $P[0]$ with $T[1]$. $P[0]$ matches with $T[1]$. Since there is a match, P is not shifted.

T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P		a	b	a	b	a	c	a							

Step 3: $i = 2,\ j = 1$, comparing $P[1]$ with $T[2]$. $P[1]$ does not match with $T[2]$. Backtracking on P, comparing $P[0]$ and $T[2]$.

T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P		a	b	a	b	a	c	a							

Step 4: $i = 3,\ j = 0$, comparing $P[0]$ with $T[3]$. $P[0]$ does not match with $T[3]$.

T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P				a	b	a	b	a	c	a					

Step 5: $i = 4,\ j = 0$, comparing $P[0]$ with $T[4]$. $P[0]$ matches with $T[4]$.

T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P					a	b	a	b	a	c	a				

Step 6: $i = 5,\ j = 1$, comparing $P[1]$ with $T[5]$. $P[1]$ matches with T[5].

T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P					a	b	a	b	a	c	a				

Step 7: $i = 6,\ j = 2$, comparing $P[2]$ with $T[6]$. $P[2]$ matches with $T[6]$.

T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P					a	b	a	b	a	c	a				

Step 8: $i = 7,\ j = 3$, comparing $P[3]$ with $T[7]$. $P[3]$ matches with $T[7]$.

T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P					a	b	a	b	a	c	a				

Step 9: $i = 8,\ j = 4$, comparing $P[4]$ with $T[8]$. $P[4]$ matches with $T[8]$.

T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P					a	b	a	b	a	c	a				

Step 10: $i = 9,\ j = 5$, comparing $P[5]$ with $T[9]$. $P[5]$ does not match with $T[9]$. Backtracking on P, comparing $P[4]$ with $T[9]$ because after mismatch $j = F[4] = 3$.

T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P					a	b	a	b	a	c	a				

Comparing $P[3]$ with $T[9]$.

T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P							a	b	a	b	a	c	a		

Step 11: $i = 10,\ j = 4$, comparing $P[4]$ with $T[10]$. $P[4]$ matches with $T[10]$.

T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P							a	b	a	b	a	c	a		

Step 12: $i = 11,\ j = 5$, comparing $P[5]$ with $T[11]$. $P[5]$ matches with $T[11]$.

T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P							a	b	a	b	a	c	a		

Step 13: $i = 12,\ j = 6$, comparing $P[6]$ with $T[12]$. $P[6]$ matches with $T[12]$.

T	b	a	c	b	a	b	a	b	a	b	a	c	a	c	a
P							a	b	a	b	a	c	a		

Pattern P has been found to completely occur in string T. The total number of shifts that took place for the match to be found are: $i - m = 13 - 7 = 6$ shifts.

Notes:

- KMP performs the comparisons from left to right
- KMP algorithm needs a preprocessing (prefix function) which takes O(m) space and time complexity
- Searching takes O($n + m$) time complexity (does not depend on alphabet size)

15.7 Boyer-Moore Algorithm

Like the KMP algorithm, this also does some pre-processing and we call it $last\ function$. The algorithm scans the characters of the pattern from right to left beginning with the rightmost character. During the testing of a possible placement of pattern P in T, a mismatch is handled as follows: Let us assume that the current character being matched is $T[i] = c$ and the corresponding pattern character is $P[j]$. If c is not contained anywhere in P, then shift the pattern P completely past $T[i]$. Otherwise, shift P until an occurrence of character c in P gets aligned with $T[i]$. This technique avoids needless comparisons by shifting the pattern relative to the text.

The $last$ function takes O($m + |\Sigma|$) time and the actual search takes O(nm) time. Therefore the worst case running time of the Boyer-Moore algorithm is O($nm + |\Sigma|$). This indicates that the worst-case running time is quadratic, in the case of $n == m$, the same as the brute force algorithm.

- The Boyer-Moore algorithm is very fast on the large alphabet (relative to the length of the pattern).
- For the small alphabet, Boyer-Moore is not preferable.
- For binary strings, the KMP algorithm is recommended.
- For the very shortest patterns, the brute force algorithm is better.

15.8 Data Structures for Storing Strings

If we have a set of strings (for example, all the words in the dictionary) and a word which we want to search in that set, in order to perform the search operation faster, we need an efficient way of storing the strings. To store sets of strings we can use any of the following data structures.

- Hashing Tables
- Binary Search Trees
- Tries
- Ternary Search Trees

ALGORITHMS DESIGN TECHNIQUES

CHAPTER 16

16.1 Introduction

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 in getting the solution easily. In this chapter, we will see different ways of classifying the algorithms and in subsequent chapters we will focus on a few of them (Greedy, Divide and Conquer, Dynamic Programming).

16.2 Classification

There are many ways of classifying algorithms and a few of them are shown below:

- Implementation Method
- Design Method
- Other Classifications

16.3 Classification by Implementation Method

Recursion or Iteration

A $recursive$ algorithm is one that calls itself repeatedly until a base condition is satisfied. It is a common method used in functional programming languages like $C, C++$, etc.

$Iterative$ algorithms use constructs like loops and sometimes other data structures like stacks and queues to solve the problems.

Some problems are suited for recursive and others are suited for iterative. For example, the $Towers\ of\ Hanoi$ problem can be easily understood in recursive implementation. Every recursive version has an iterative version, and vice versa.

Procedural or Declarative (Non-Procedural)

In $declarative$ programming languages, we say what we want without having to say how to do it. With $procedural$ programming, we have to specify the exact steps to get the result. For example, SQL is more declarative than procedural, because the queries don't specify the steps to produce the result. Examples of procedural languages include: C, PHP, and PERL.

Serial or Parallel or Distributed

In general, while discussing the algorithms we assume that computers execute one instruction at a time. These are called $serial$ algorithms.

$Parallel$ algorithms take advantage of computer architectures to process several instructions at a time. They divide the problem into subproblems and serve them to several processors or threads. Iterative algorithms are generally parallelizable.

If the parallel algorithms are distributed on to different machines then we call such algorithms $distributed$ algorithms.

Deterministic or Non-Deterministic

$Deterministic$ algorithms solve the problem with a predefined process, whereas $non-deterministic$ algorithms guess the best solution at each step through the use of heuristics.

Exact or Approximate

As we have seen, for many problems we are not able to find the optimal solutions. That means, the algorithms for which we are able to find the optimal solutions are called $exact$ algorithms. In computer science, if we do not have the optimal solution, we give approximation algorithms.

Approximation algorithms are generally associated with NP-hard problems (refer to the $Complexity\ Classes$ chapter for more details).

16.4 Classification by Design Method

Another way of classifying algorithms is by their design method.

Greedy Method

Greedy algorithms work in stages. In each stage, a decision is made that is good at that point, without bothering about the future consequences. Generally, this means that some *local best* is chosen. It assumes that the local best selection also makes for the *global* optimal solution.

Divide and Conquer

The D & C strategy solves a problem by:

1) Divide: Breaking the problem into sub problems that are themselves smaller instances of the same type of problem.
2) Recursion: Recursively solving these sub problems.
3) Conquer: Appropriately combining their answers.

Examples: merge sort and binary search algorithms.

Dynamic Programming

Dynamic programming (DP) and memoization work together. The difference between DP and divide and conquer is that in the case of the latter there is no dependency among the sub problems, whereas in DP there will be an overlap of sub-problems. By using memoization [maintaining a table for already solved sub problems], DP reduces the exponential complexity to polynomial complexity (O(n^2), O(n^3), etc.) for many problems.

The difference between dynamic programming and recursion is in the memoization of recursive calls. When sub problems are independent and if there is no repetition, memoization does not help, hence dynamic programming is not a solution for all problems.

By using memoization [maintaining a table of sub problems already solved], dynamic programming reduces the complexity from exponential to polynomial.

Linear Programming

Linear programming is not a programming language like C++, Java, or Visual Basic. Linear programming can be defined as:

A method to allocate scarce resources to competing activities in an optimal manner when the problem can be expressed using a linear objective function and linear inequality constraints.

A linear program consists of a set of variables, a linear objective function indicating the contribution of each variable to the desired outcome, and a set of linear constraints describing the limits on the values of the variables. The *solution* to a linear program is a set of values for the problem variables that results in the best – *largest or smallest* – value of the objective function and yet is consistent with all the constraints. Formulation is the process of translating a real-world problem into a linear program. Once a problem has been formulated as a linear program, a computer program can be used to solve the problem. In this regard, solving a linear program is relatively easy. The hardest part about applying linear programming is formulating the problem and interpreting the solution. In linear programming, there are inequalities in terms of inputs and *maximizing* (or *minimizing*) some linear function of the inputs. Many problems (example: maximum flow for directed graphs) can be discussed using linear programming.

Reduction [Transform and Conquer]

In this method we solve a difficult problem by transforming it into a known problem for which we have asymptotically optimal algorithms. In this method, the goal is to find a reducing algorithm whose complexity is not dominated by the resulting reduced algorithms. For example, the selection algorithm for finding the median in a list involves first sorting the list and then finding out the middle element in the sorted list. These techniques are also called *transform and conquer*.

16.5 Other Classifications

Classification by Research Area

In computer science each field has its own problems and needs efficient algorithms. Examples: search algorithms, sorting algorithms, merge algorithms, numerical algorithms, graph algorithms, string algorithms, geometric algorithms, combinatorial algorithms, machine learning, cryptography, parallel algorithms, data compression algorithms, parsing techniques, and more.

Classification by Complexity

In this classification, algorithms are classified by the time they take to find a solution based on their input size. Some algorithms take linear time complexity (O(n)) and others take exponential time, and some never halt. Note that some problems may have multiple algorithms with different complexities.

Randomized Algorithms

A few algorithms make choices randomly. For some problems, the fastest solutions must involve randomness. Example: Quick Sort.

Branch and Bound Enumeration and Backtracking

These were used in Artificial Intelligence and we do not need to explore these fully. For the Backtracking method refer to the *Recusion and Backtracking* chapter.

Note: In the next few chapters we discuss the Greedy, Divide and Conquer, and Dynamic Programming] design methods. These methods are emphasized because they are used more often than other methods to solve problems.

CHAPTER 17

Greedy Algorithms

☼ ☼ ☼

17.1 Introduction

Let us start our discussion with simple theory that will give us an understanding of the Greedy technique. In the game of *Chess*, every time we make a decision about a move, we have to also think about the future consequences. Whereas, in the game of *Tennis* (or *Volleyball*), our action is based on the immediate situation. This means that in some cases making a decision that looks right at that moment gives the best solution (*Greedy*), but in other cases it doesn't. The Greedy technique is best suited for looking at the immediate situation.

17.2 Greedy Strategy

Greedy algorithms work in stages. In each stage, a decision is made that is good at that point, without bothering about the future. This means that some *local best* is chosen. It assumes that a local good selection makes for a global optimal solution.

17.3 Elements of Greedy Algorithms

The two basic properties of optimal Greedy algorithms are:

1) Greedy choice property
2) Optimal substructure

Greedy choice property

This property says that the globally optimal solution can be obtained by making a locally optimal solution (Greedy). The choice made by a Greedy algorithm may depend on earlier choices but not on the future. It iteratively makes one Greedy choice after another and reduces the given problem to a smaller one.

Optimal substructure

A problem exhibits optimal substructure if an optimal solution to the problem contains optimal solutions to the subproblems. That means we can solve subproblems and build up the solutions to solve larger problems.

17.5 Advantages and Disadvantages of Greedy Method

The main advantage of the Greedy method is that it is straightforward, easy to understand and easy to code. In Greedy algorithms, once we make a decision, we do not have to spend time re-examining the already computed values. Its main disadvantage is that for many problems there is no greedy algorithm. That means, in many cases there is no guarantee that making locally optimal improvements in a locally optimal solution gives the optimal global solution.

17.6 Greedy Applications

- Sorting: Selection sort, Topological sort
- Priority Queues: Heap sort
- Huffman coding compression algorithm
- Prim's and Kruskal's algorithms
- Shortest path in Weighted Graph [Dijkstra's]
- Coin change problem

- Fractional Knapsack problem
- Disjoint sets-UNION by size and UNION by height (or rank)
- Job scheduling algorithm
- Greedy techniques can be used as an approximation algorithm for complex problems

17.7 Understanding Greedy Technique

For better understanding let us go through an example.

Huffman Coding Algorithm

Definition

Given a set of n characters from the alphabet A [each character $c \in A$] and their associated frequency $freq(c)$, find a binary code for each character $c \in A$, such that $\sum_{c \in A} freq(c)|binarycode(c)|$ is minimum, where $|binarycode(c)|$ represents the length of binary code of character c. That means the sum of the lengths of all character codes should be minimum [the sum of each character's frequency multiplied by the number of bits in the representation].

The basic idea behind the Huffman coding algorithm is to use fewer bits for more frequently occurring characters. The Huffman coding algorithm compresses the storage of data using variable length codes. We know that each character takes 8 bits for representation. But in general, we do not use all of them. Also, we use some characters more frequently than others. When reading a file, the system generally reads 8 bits at a time to read a single character. But this coding scheme is inefficient. The reason for this is that some characters are more frequently used than other characters. Let's say that the character $'e'$ is used 10 times more frequently than the character $'q'$. It would then be advantageous for us to instead use a 7 bit code for e and a 9 bit code for q because that could reduce our overall message length.

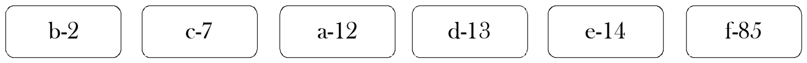
On average, using Huffman coding on standard files can reduce them anywhere from 10% to 30% depending on the character frequencies. The idea behind the character coding is to give longer binary codes for less frequent characters and groups of characters. Also, the character coding is constructed in such a way that no two character codes are prefixes of each other.

An Example

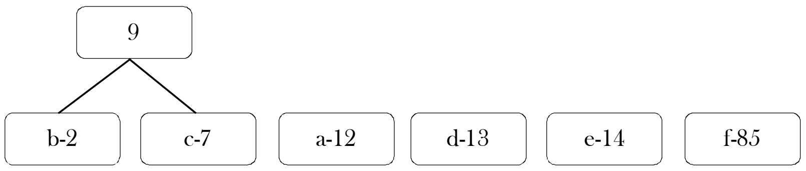
Let's assume that after scanning a file we find the following character frequencies:

Character	Frequency
a	12
b	2
c	7
d	13
e	14
f	85

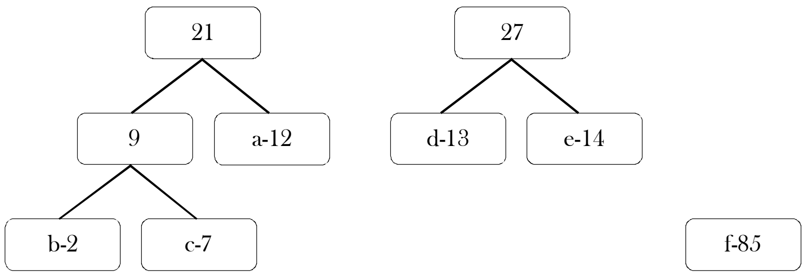
Given this, create a binary tree for each character that also stores the frequency with which it occurs (as shown below).

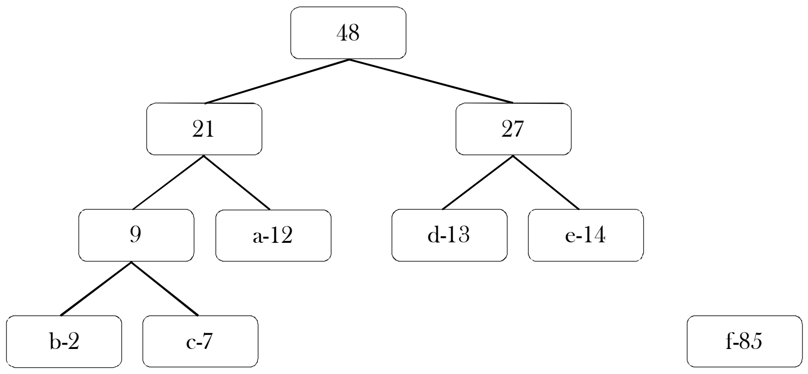


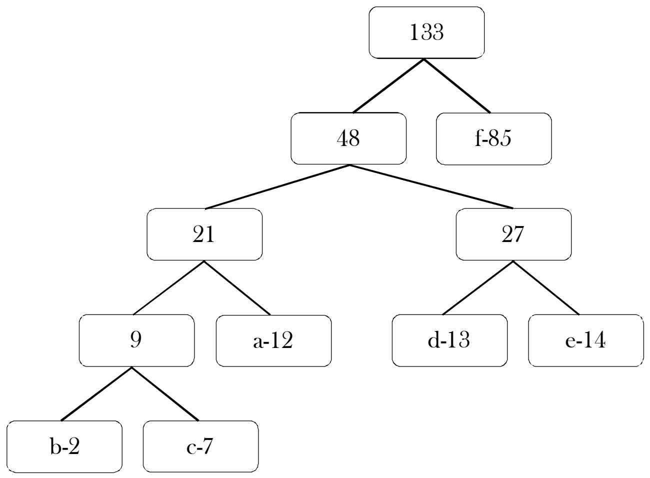
The algorithm works as follows: In the list, find the two binary trees that store minimum frequencies at their nodes. Connect these two nodes at a newly created common node that will store no character but will store the sum of the frequencies of all the nodes connected below it. So our picture looks like this:



Repeat this process until only one tree is left:







Once the tree is built, each leaf node corresponds to a letter with a code. To determine the code for a particular node, traverse from the root to the leaf node. For each move to the left, append a 0 to the code, and for each move to the right, append a 1. As a result, for the above generated tree, we get the following codes:

Letter	Code
a	001
b	0000
c	0001
d	010
e	011
f	1

Calculating Bits Saved

Now, let us see how many bits that Huffman coding algorithm is saving. All we need to do for this calculation is see how many bits are originally used to store the data and subtract from that the number of bits that are used to store the data using the Huffman code. In the above example, since we have six characters, let's assume each character is stored with a three bit code. Since there are 133 such characters (multiply total frequencies by 3), the total number of bits used is $3 * 133 = 399$. Using the Huffman coding frequencies we can calculate the new total number of bits used:

Letter	Code	Frequency	Total Bits
a	001	12	36
b	0000	2	8
c	0001	7	28
d	010	13	39
e	011	14	42
f	1	85	85
Total			238

Thus, we saved $399 - 238 = 161$ bits, or nearly 40% of the storage space.

```
from heapq import heappush, heappop, heapify
from collections import defaultdict

def HuffmanEncode(characterFrequency):
    heap = [[freq, [sym, ""]] for sym, freq in characterFrequency.items()]
    heapify(heap)
    while len(heap) > 1:
        lo = heappop(heap)
        hi = heappop(heap)
```

CHAPTER 18

DIVIDE AND CONQUER ALGORITHMS

18.1 Introduction

In the *Greedy* chapter, we have seen that for many problems the Greedy strategy failed to provide optimal solutions. Among those problems, there are some that can be easily solved by using the *Divide and Conquer* (D & C) technique. Divide and Conquer is an important algorithm design technique based on recursion.

The D & C algorithm works by recursively breaking down a problem into two or more sub problems of the same type, until they become simple enough to be solved directly. The solutions to the sub problems are then combined to give a solution to the original problem.

18.2 What is Divide and Conquer Strategy?

The D & C strategy solves a problem by:

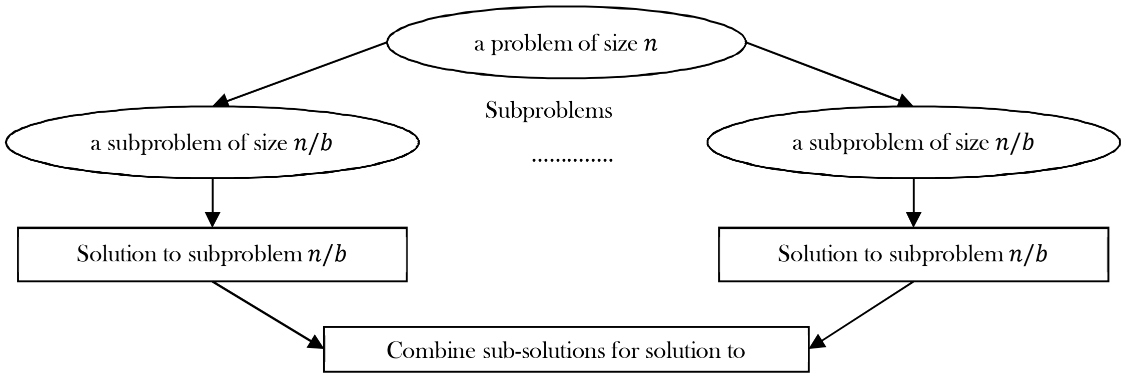
1) *Divide*: Breaking the problem into subproblems that are themselves smaller instances of the same type of problem.
2) *Conquer*: Conquer the subproblems by solving them recursively.
3) *Combine*: Combine the solutions to the subproblems into the solution for the original given problem.

18.3 Does Divide and Conquer Always Work?

It's not possible to solve all the problems with the Divide & Conquer technique. As per the definition of D & C, the recursion solves the subproblems which are of the same type. For all problems it is not possible to find the subproblems which are the same size and D & C is not a choice for all problems.

18.4 Divide and Conquer Visualization

For better understanding, consider the following visualization. Assume that n is the size of the original problem. As described above, we can see that the problem is divided into sub problems with each of size n/b (for some constant b). We solve the sub problems recursively and combine their solutions to get the solution for the original problem.


a problem of size n
Subproblems
a subproblem of size n/b
....
a subproblem of size n/b
Solution to subproblem n/b
Solution to subproblem n/b
Combine sub-solutions for solution to

```
DivideAndConquer ( P ):
    if( small ( P ) ):
        # P is very small so that a solution is obvious
        return solution ( n )
        divide the problem P into k sub problems P1, P2, ..., Pk
        return (
            Combine (
                DivideAndConquer ( P1 ),
                DivideAndConquer ( P2 ),
                ...
                DivideAndConquer ( Pk ))
```

18.5 Understanding Divide and Conquer

For a clear understanding of D & C, let us consider a story. There was an old man who was a rich farmer and had seven sons. He was afraid that when he died, his land and his possessions would be divided among his seven sons, and that they would quarrel with one another.

So he gathered them together and showed them seven sticks that he had tied together and told them that anyone who could break the bundle would inherit everything. They all tried, but no one could break the bundle. Then the old man untied the bundle and broke the sticks one by one. The brothers decided that they should stay together and work together and succeed together. The moral for problem solvers is different. If we can't solve the problem, divide it into parts, and solve one part at a time.

In earlier chapters we have already solved many problems based on D & C strategy: like Binary Search, Merge Sort, Quick Sort, etc.... Refer to those topics to get an idea of how D & C works. Below are a few other real-time problems which can easily be solved with D & C strategy. For all these problems we can find the subproblems which are similar to the original problem.

- Looking for a name in a phone book: We have a phone book with names in alphabetical order. Given a name, how do we find whether that name is there in the phone book or not?
- Breaking a stone into dust: We want to convert a stone into dust (very small stones).
- Finding the exit in a hotel: We are at the end of a very long hotel lobby with a long series of doors, with one door next to us. We are looking for the door that leads to the exit.
- Finding our car in a parking lot.

18.6 Advantages of Divide and Conquer

Solving difficult problems: D & C is a powerful method for solving difficult problems. As an example, consider the Tower of Hanoi problem. This requires breaking the problem into subproblems, solving the trivial cases and combining the subproblems to solve the original problem. Dividing the problem into subproblems so that subproblems can be combined again is a major difficulty in designing a new algorithm. For many such problems D & C provides a simple solution.

Parallelism: Since D & C allows us to solve the subproblems independently, this allows for execution in multi-processor machines, especially shared-memory systems where the communication of data between processors does not need to be planned in advance, because different subproblems can be executed on different processors.

Memory access: D & C algorithms naturally tend to make efficient use of memory caches. This is because once a subproblem is small, all its subproblems can be solved within the cache, without accessing the slower main memory.

18.7 Disadvantages of Divide and Conquer

One disadvantage of the D & C approach is that recursion is slow. This is because of the overhead of the repeated subproblem calls. Also, the D & C approach needs stack for storing the calls (the state at each point in the recursion). Actually this depends upon the implementation style. With large enough recursive base cases, the overhead of recursion can become negligible for many problems.

Another problem with D & C is that, for some problems, it may be more complicated than an iterative approach. For example, to add n numbers, a simple loop to add them up in sequence is much easier than a D & C approach that breaks the set of numbers into two halves, adds them recursively, and then adds the sums.

18.8 Master Theorem

As stated above, in the D & C method, we solve the sub problems recursively. All problems are generally defined in terms of recursive definitions. These recursive problems can easily be solved using Master theorem. For details on Master theorem, refer to the *Introduction to Analysis of Algorithms* chapter. Just for continuity, let us reconsider the Master theorem. 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 the complexity can be directly given as:

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

18.9 Divide and Conquer Applications

- Binary Search
- Merge Sort and Quick Sort
- Median Finding
- Min and Max Finding
- Matrix Multiplication
- Closest Pair problem
- Finding peak an one-dimentional and two-dimentional arrays

18.10 Finding peak element of an array

Given an input array A, find a peak element and return its index. In an array, a peak element is an element that is greater than its neighbors. We need to find the index i of the peak element A[i] where A[i] ≥ A[i − 1] and A[i] ≥ A[i + 1]. The array may contain multiple peaks, in that case return the index to any one of the peaks. For the first element there won't be previous element; hence assume A[-1] = -∞. On the similar lines, for the last element there won't be next element; assume A[n] = -∞.

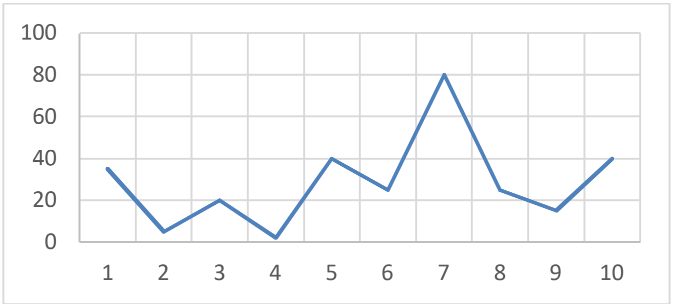
Examples

To understand the problem statement well, let us plot the elements with array indexes as X-axis and values of corresponding indexes as Y-axis.

Example-1:

Input array	35, 5, 20, 2, 40, 25, 80, 25, 15, 40
Possible peaks	35, 20, 40, 80, 40

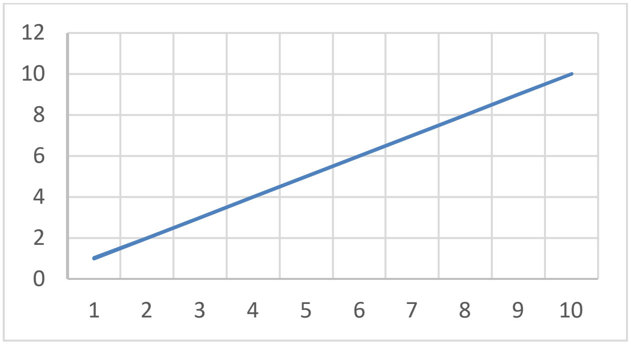
In the following graph, we can observe that elements 20, 40, and 80 are greater than its neighbors. Element 40 is the last element and is greater than its previous element 15.



Example-2:

Input array	1, 2, 3, 4, 5, 6, 7, 8, 9, 10
Possible peaks	10

Since the elements were in ascending order, only the last element is satisfying the peak element definition.



Observations

- If array has all the same elements, every element is a peak element.
- Every array has a peak element.
- Array might have many peak elements but we are finding only one.
- If array is in ascending order then last element of the array will be the peak element.
- If array is in descending order then first element of the array will be the peak element.

CHAPTER 19

Dynamic Programming

☼ ☼ ☼

19.1 Introduction

In this chapter, we will try to solve few of the problems for which we failed to get the optimal solutions using other techniques (say, *Greedy* and *Divide* & *Conquer* approaches). Dynamic Programming is a simple technique but it can be difficult to master. Being able to tackle problems of this type would greatly increase your skill.

Dynamic programming (usually referred to as DP) is a very powerful technique to solve a particular class of problems. It demands very elegant formulation of the approach and simple thinking and the coding part is very easy. The idea is very simple, if you have solved a problem with the given input, then save the result for future reference, so as to avoid solving the same problem again. Simply, we need to remember the past.

One easy way to identify and master DP problems is by solving as many problems as possible. The term DP is not related to coding, but it is from literature, and means filling tables.

19.2 What is Dynamic Programming Strategy?

Dynamic programming is typically applied to *optimization problems*. In such problems there can be many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value. We call such a solution an optimal solution to the problem, as opposed to the optimal solution, since there may be several solutions that achieve the optimal value.

The development of a dynamic-programming algorithm can be broken into a sequence of four steps.

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Steps 1-3 form the basis of a dynamic-programming solution to a problem. Step 4 can be omitted if only the value of an optimal solution is required. When we do perform step 4, we sometimes maintain additional information during the computation in step 3 to ease the construction of an optimal solution.

If the given problem can be broken up into smaller sub-problems and these smaller subproblems are in turn divided into still-smaller ones, and in this process, if you observe some over-lapping subproblems, then it's a big hint for DP. Also, the optimal solutions to the subproblems contribute to the optimal solution of the given problem.

19.3 Properties of Dynamic Programming Strategy

The two dynamic programming properties which can tell whether it can solve the given problem or not are:

- *Optimal substructure*: An optimal solution to a problem contains optimal solutions to sub problems.
- *Overlapping sub problems*: A recursive solution contains a small number of distinct sub problems repeated many times.

19.4 Greedy vs Divide and Conquer vs DP

All algorithmic techniques construct an optimal solution of a subproblem based on optimal solutions of smaller subproblems.
Greedy algorithms are one which finds optimal solution at each and every stage with the hope of finding global optimum at the end. The main difference between DP and greedy is that, the choice made by a greedy algorithm may depend on choices made so far but not on future choices or all the solutions to the sub problem. It iteratively makes one greedy choice after another, reducing each given problem into a smaller one. In other words, a greedy algorithm never reconsiders its choices. This is the main difference from dynamic programming, which is exhaustive and is guaranteed to find the solution. After every stage, dynamic programming makes decisions based on all the decisions made in the previous stage, and may reconsider the previous stage's algorithmic path to solution.

The main difference between dynamic programming and divide and conquer is that in the case of the latter, sub problems are independent, whereas in DP there can be an overlap of sub problems.

19.5 Can DP solve all problems?

Like greedy and divide and conquer techniques, DP cannot solve every problem. There are problems which cannot be solved by any algorithmic technique [greedy, divide and conquer and DP]. The difference between DP and straightforward recursion is in memoization of recursive calls. If the sub problems are independent and there is no repetition then DP does not help. So, dynamic programming is not a solution for all problems.

19.6 Dynamic Programming Approaches

Dynamic programming is all about ordering computations in a way that we avoid recalculating duplicate work. In dynamic programming, we have a main problem, and subproblems (subtrees). The subproblems typically repeat and overlap. The major components of DP are:

- Overlapping subproblems: Solves sub problems recursively.
- Storage: Store the computed values to avoid recalculating already solved subproblems.

By using extra storage, DP reduces the exponential complexity to polynomial complexity (O(n^2), O(n^3), etc.) for many problems.

Basically, there are two approaches for solving DP problems:

- Top-down approach [Memoization]
- Bottom-up approach [Tabulation]

These approaches were classified based on the way we fill the storage and reuse them.

$$Dynamic\ Programming\ =\ Overlapping\ subproblems\ +\ \text{Memoization or}\ Tabulation$$

19.7 Understanding DP Approaches

Top-down Approach [Memoization]

In this method, the problem is broken into sub problems; each of these subproblems is solved; and the solutions remembered, in case they need to be solved. Also, we save each computed value as the final action of the recursive function, and as the first action we check if pre-computed value exists.

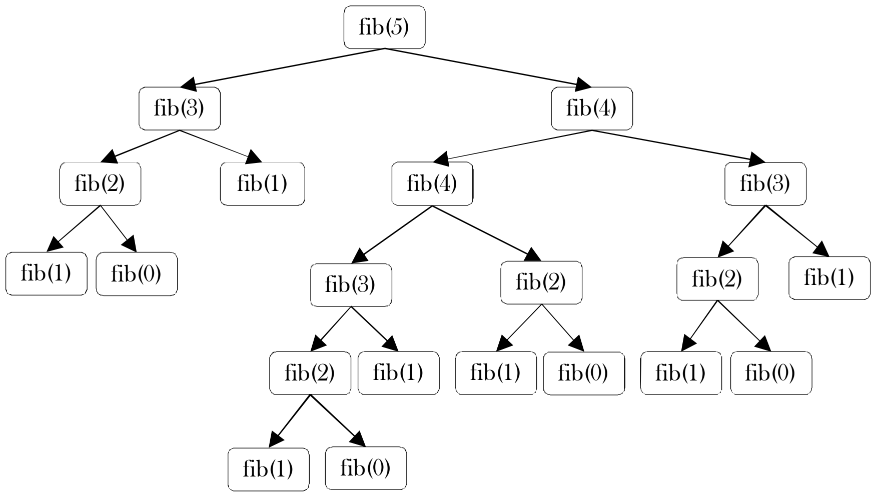
Bottom-up Approach [Tabulation]

In this method, we evaluate the function starting with the smallest possible input argument value and then we step through possible values, slowly increasing the input argument value. While computing the values we store all computed values in a table (memory). As larger arguments are evaluated, pre-computed values for smaller arguments can be used.

Example: Fibonacci Series

Let us understand how DP works through an example; Fibonacci series. In Fibonacci series, the current number is the sum of previous two numbers. The Fibonacci series is defined as follows:

$$\begin{aligned} Fib(\text{n}) &= 0, && for\ n = 0 \\ &= 1, && for\ n = 1 \\ &= Fib(\text{n}-1) + Fib(\text{n}-2), && for\ n > 1 \end{aligned}$$



Calling $fib(5)$ produces a call tree that calls the function on the same value many times:

$fib(5)$
$fib(4) + fib(3)$
$(fib(3) + fib(2)) + (fib(2) + fib(1))$

$((fib(2) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))$
$(((fib(1) + fib(0)) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))$

In the above example, $fib(2)$ was calculated three times (overlapping of subproblems). If n is big, then many more values of fib (subproblems) are recalculated, which leads to an exponential time algorithm. Instead of solving the same subproblems again and again we can store the previous calculated values and reduce the complexity.

The recursive implementation can be given as:

```
def fibo(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else: return fibo(n-1)+fibo(n-2)
print (fibo(10))
```

Solving the above recurrence gives:

$$T(n) = T(n-1) + T(n-2) + 1 \approx \left(\frac{1+\sqrt{5}}{2}\right)^n \approx 2^n = O(2^n)$$

Memoization Solution [Top-down]

$Memoization$ works like this: Start with a recursive function and add a table that maps the function's parameter values to the results computed by the function. Then if this function is called twice with the same parameters, we simply look up the answer in the table. In this method, we preserve the recursive calls and use the values if they are already computed. The implementation for this is given as:

```
fibTable = {1:0, 2:1}
def fibo(n):
  if n <= 2:
    return 1
  if n in fibTable:
    return fibTable[n]
  else:
    fibTable[n] = fibo(n-1) + fibo(n-2)
    return fibTable[n]
print(fibo(10))
```

Tabulation Solution [Bottom-up]

The other approach is bottom-up. Now, we see how DP reduces this problem complexity from exponential to polynomial. This method start with lower values of input and keep building the solutions for higher values.

```
def fibo(n):
  fibTable = [0,1]
  for i in range(2,n+1):
    fibTable.append(fibTable[i-1] + fibTable[i-2])
  return fibTable[n]
print(fibo(10))
```

Note: For all problems, it may not be possible to find both top-down and bottom-up programming solutions.

Both versions of the Fibonacci series implementations clearly reduce the problem complexity to $O(n)$. This is because if a value is already computed then we are not calling the subproblems again. Instead, we are directly taking its value from the table.

Time Complexity: $O(n)$. Space Complexity: $O(n)$, for table.

Further Improving

One more observation from the Fibonacci series is: The current value is the sum of the previous two calculations only. This indicates that we don't have to store all the previous values. Instead, if we store just the last two values, we can calculate the current value. The implementation for this is given below:

```
def fibo(n):
  a, b = 0, 1
  for i in range(n):
        a, b = b, a + b
  return a
print(fibo(10))
```

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

Note: This method may not be applicable (available) for all problems.

Observations

While solving the problems using DP, try to figure out the following:

- See how the problems are defined in terms of subproblems recursively.
- See if we can use some table [memoization] to avoid the repeated calculations.

Example: Factorial of a Number

As another example, consider the factorial problem: $n!$ is the product of all integers between n and 1. The definition of recursive factorial can be given as:

$$n! = n * (n-1)!$$
$$1! = 1$$
$$0! = 1$$

This definition can easily be converted to implementation. Here the problem is finding the value of $n!$, and the sub-problem is finding the value of $(n-l)!$. In the recursive case, when n is greater than 1, the function calls itself to find the value of $(n-l)!$ and multiplies that with n. In the base case, when n is 0 or 1, the function simply returns 1.

```
def factorial(n):
  if n == 0: return 1
  return n*factorial(n-1)
print(factorial(6))
```

The recurrence for the above implementation can be given as:

$$T(n) = n \times T(n-1) \approx O(n)$$

Time Complexity: $O(n)$. Space Complexity: $O(n)$, recursive calls need a stack of size n.

In the above recurrence relation and implementation, for any n value, there are no repetitive calculations (no overlapping of sub problems) and the factorial function is not getting any benefits with dynamic programming. Now, let us say we want to compute a series of $m!$ for some arbitrary value m. Using the above algorithm, for each such call we can compute it in $O(m)$. For example, to find both $n!$ and $m!$ we can use the above approach, wherein the total complexity for finding $n!$ and $m!$ is $O(m+n)$.

Time Complexity: $O(n+m)$. Space Complexity: $O(\max(m,n))$, recursive calls need a stack of size equal to the maximum of m and n.

Improving with Dynamic Programming

Now let us see how DP reduces the complexity. From the above recursive definition it can be seen that $fact(n)$ is calculated from $fact(n$ - 1) and n and nothing else. Instead of calling $fact(n)$ every time, we can store the previous calculated values in a table and use these values to calculate a new value. This implementation can be given as:

```
factTable = {}
def factorial(n):
   try:
      return factTable[n]
   except KeyError:
      if n == 0:
         factTable[0] = 1
         return 1
      else:
         factTable[n] = n * factorial(n-1)
         return factTable[n]
print(factorial(10))
```

For simplicity, let us assume that we have already calculated $n!$ and want to find $m!$. For finding $m!$, we just need to see the table and use the existing entries if they are already computed. If $m < n$ then we do not have to recalculate $m!$. If $m > n$ then we can use $n!$ and call the factorial on the remaining numbers only.

The above implementation clearly reduces the complexity to $O(max\ (m,n))$. This is because if the $fact$(n) is already there, then we are not recalculating the value again. If we fill these newly computed values, then the subsequent calls further reduce the complexity.

Time Complexity: $O(max\ (m,n))$. Space Complexity: $O(max\ (m,n))$ for table.

Bottom-up versus Top-down Programming

With *tabulation* (bottom-up), we start from smallest instance size of the problem, and *iteratively* solve bigger problems using solutions of the smaller problems (i.e. by reading from the table), until we reach our starting instance.

With *memoization* (top-down) we start right away at original problem instance, and solve it by breaking it down into smaller instances of the same problem (*recursion*). When we have to solve smaller instance, we first check in a look-up table to see if we already solved it. If we did, we just read it up and return value without solving it again and branching into recursion. Otherwise, we solve it recursively, and save result into table for further use.

In bottom-up approach, the programmer has to select values to calculate and decide the order of calculation. In this case, all subproblems that might be needed are solved in advance and then used to build up solutions to larger problems.

In top-down approach, the recursive structure of the original code is preserved, but unnecessary recalculation is avoided. The problem is broken into subproblems, these subproblems are solved and the solutions remembered, in case they need to be solved again.

Recursion with memoization is better whenever the state is sparse space (number of different subproblems are less). In other words, if we don't actually need to solve all smaller subproblems but only some of them. In such cases the recursive implementation can be much faster. Recursion with memoization is also better whenever the state space is irregular, i.e., whenever it is hard to specify an order of evaluation iteratively. Recursion with memoization is faster because only subproblems that are necessary in a given problem instance are solved.

Tabulation methods are better whenever the state space is dense and regular. If we need to compute the solutions to all the subproblems anyway, we may as well do it without all the function calling overhead. An additional advantage of the iterative approach is that we are often able to save memory by forgetting the solutions to subproblems we won't need in the future. For example, if we only need row k of the table to compute row $k + 1$, there is no need to remember row $k - 1$ anymore. On the flip side, in tabulation method, we solve all subproblems in spite of the fact that some subproblems may not be needed for a given problem instance.

19.8 Examples of DP Algorithms

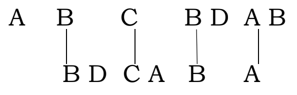
- Many string algorithms including longest common subsequence, longest increasing subsequence, longest common substring, edit distance.
- Algorithms on graphs can be solved efficiently: Bellman-Ford algorithm for finding the shortest distance in a graph, Floyd's All-Pairs shortest path algorithm, etc.
- Chain matrix multiplication
- Subset Sum
- 0/1 Knapsack
- Travelling salesman problem, and many more

19.9 Longest Common Subsequence

Given two strings: string X of length m [$X(1..m)$], and string Y of length n [$Y(1..n)$], find the longest common subsequence: the longest sequence of characters that appear left-to-right (but not necessarily in a contiguous block) in both strings. For example, if X = "ABCBDAB" and Y = "BDCABA", the LCS(X, Y) = {"BCBA", "BDAB", "BCAB"}. We can see there are several optimal solutions.

Brute Force Approach: One simple idea is to check every subsequence of $X[1..m]$ (m is the length of sequence X) to see if it is also a subsequence of $Y[1..n]$ (n is the length of sequence Y). Checking takes O(n) time, and there are 2^m subsequences of X. The running time thus is exponential O($n.2^m$) and is not good for large sequences.

Recursive Solution: Before going to DP solution, let us form the recursive solution for this and later we can add memoization to reduce the complexity. Let's start with some simple observations about the LCS problem. If we have two strings, say "ABCBDAB" and "BDCABA", and if we draw lines from the letters in the first string to the corresponding letters in the second, no two lines cross:



From the above observation, we can see that the current characters of X and Y may or may not match. That means, suppose that the two first characters differ. Then it is not possible for both of them to be part of a common subsequence - one or the other (or maybe both) will have to be removed. Finally, observe that once we have decided what to do with the first characters of the strings, the remaining sub problem is again a LCS problem, on two shorter strings. Therefore we can solve it recursively.

The solution to LCS should find two sequences in X and Y and let us say the starting index of sequence in X is i and the starting index of sequence in Y is j. Also, assume that $X[i \dots m]$ is a substring of X starting at character i and going until the end of X, and that $Y[j \dots n]$ is a substring of Y starting at character j and going until the end of Y.

Based on the above discussion, here we get the possibilities as described below:

1) If $X[i] == Y[j] : 1 + LCS(i + 1, j + 1)$
2) If $X[i] \neq Y[j]$: $LCS(i, j + 1)$ // skipping j^{th} character of Y
3) If $X[i] \neq Y[j]$: $LCS(i + 1, j)$ // skipping i^{th} character of X

In the first case, if $X[i]$ is equal to $Y[j]$, we get a matching pair and can count it towards the total length of the LCS. Otherwise, we need to skip either i^{th} character of X or j^{th} character of Y and find the longest common subsequence. Now, $LCS(i, j)$ can be defined as:

$$LCS(i,j) = \begin{cases} 0, & if\ i = m\ or\ j = n \\ Max\{LCS(i, j+1), LCS(i+1, j)\}, & if\ \text{X[i]} \neq \text{Y[j]} \\ 1 + LCS[i+1, j+1], & if\ \text{X[i]} == \text{Y[j]} \end{cases}$$

LCS has many applications. In web searching, if we find the smallest number of changes that are needed to change one word into another. A $change$ here is an insertion, deletion or replacement of a single character.

```
def LCSLength(X, Y):
    if not X or not Y:
        return ""
    x, m, y, n = X[0], X[1:], Y[0], Y[1:]
    if x == y:
        return x + LCSLength(m, n)
    else: return max(LCSLength(X, n), LCSLength(m, Y), key=len)
print (LCSLength('thisisatest', 'testingLCS123testing'))
```

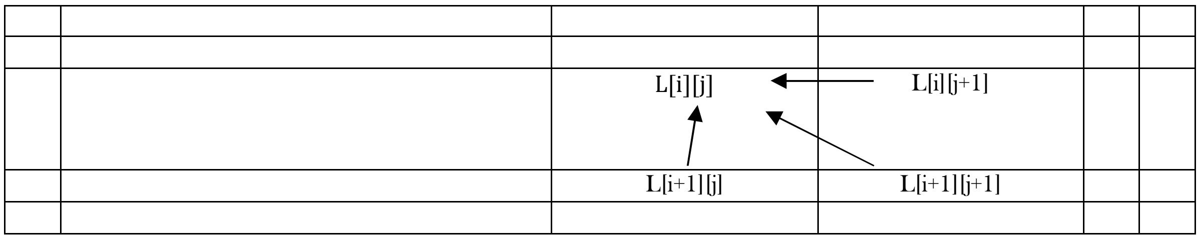
This is a correct solution but it is very time consuming. For example, if the two strings have no matching characters, the last line always gets executed which gives (if $m == n$) close to $O(2^n)$.

DP Solution: Adding Memoization: The problem with the recursive solution is that the same subproblems get called many different times. A subproblem consists of a call to LCSLength, with the arguments being two suffixes of X and Y, so there are exactly $(i+1)(j+1)$ possible subproblems (a relatively small number). If there are nearly $\mathbf{2^n}$ recursive calls, some of these subproblems must be being solved over and over.

The DP solution is to check, whenever we want to solve a sub problem, whether we've already done it before. So we look up the solution instead of solving it again. Implemented in the most direct way, we just add some code to our recursive solution. To do this, look up the code. This can be given as:

```
def LCSLength(X, Y):
    Table = [[0 for j in range(len(Y)+1)] for i in range(len(X)+1)]
    # row 0 and column 0 are initialized to 0 already
    for i, x in enumerate(X):
        for j, y in enumerate(Y):
            if x == y:
                Table[i+1][j+1] = Table[i][j] + 1
            else:
                Table[i+1][j+1] = max(Table[i+1][j], Table[i][j+1])
    # read the substring out from the matrix
    result = ""
    x, y = len(X), len(Y)
    while x != 0 and y != 0:
        if Table[x][y] == Table[x-1][y]:
            x -= 1
        elif Table[x][y] == Table[x][y-1]:
            y -= 1
        else:
            assert X[x-1] == Y[y-1]
            result = X[x-1] + result
            x -= 1
            y -= 1
    return result
print (LCSLength('thisisatest', 'testingLCS123testing'))
```

First, take care of the base cases. We have created an LCS table with one row and one column larger than the lengths of the two strings. Then run the iterative DP loops to fill each cell in the table. This is like doing recursion backwards, or bottom up.



The value of $LCS[i][j]$ depends on 3 other values ($LCS[i+1][j+1]$, $LCS[i][j+1]$ and $LCS[i+1][j]$), all of which have larger values of i or j. They go through the table in the order of decreasing i and j values. This will guarantee that when we need to fill in the value of $LCS[i][j]$, we already know the values of all the cells on which it depends.

Time Complexity: $O(mn)$, since i takes values from 1 to m and and j takes values from 1 to n. Space Complexity: $O(mn)$.

Note: In the above discussion, we have assumed $LCS(i,j)$ is the length of the LCS with $X[i \dots m]$ and $Y[j \dots n]$. We can solve the problem by changing the definition as $LCS(i,j)$ is the length of the LCS with $X[1 \dots i]$ and $Y[1 \dots j]$.

Printing the subsequence: The above algorithm can find the length of the longest common subsequence but cannot give the actual longest subsequence. To get the sequence, we trace it through the table. Start at cell $(0, 0)$. We know that the value of $LCS[0][0]$ was the maximum of 3 values of the neighboring cells. So we simply recompute $LCS[0][0]$ and note which cell gave the maximum value. Then we move to that cell (it will be one of $(1, 1)$, $(0, 1)$ or $(1, 0)$) and repeat this until we hit the boundary of the table. Every time we pass through a cell (i, j) where $X[i] == Y[j]$, we have a matching pair and print $X[i]$. At the end, we will have printed the longest common subsequence in O(mn) time.

An alternative way of getting path is to keep a separate table for each cell. This will tell us which direction we came from when computing the value of that cell. At the end, we again start at cell $(0, 0)$ and follow these directions until the opposite corner of the table. From the above examples, I hope you understood the idea behind DP. Now let us see more problems which can be easily solved using the DP technique.

Note: As we have seen above, in DP the main component is recursion. If we know the recurrence then converting that to code is a minimal task. For the problems below, we concentrate on getting the recurrence.

19.10 Dynamic Programming: Problems & Solutions

Problem-1 Convert the following recurrence to code.

$$T(0) = T(1) = 2$$
$$T(n) = \sum_{i=1}^{n-1} 2 \times T(i) \times T(i-1), \text{for } n > 1$$

Solution: The code for the given recursive formula can be given as:

```
def f(n) :
    sum = 0
    if(n==0 or n==1):
        return 2
    # Recursive case
    for i in range(1, n):
        sum += 2 * f(i) * f(i-1)
    return sum
```

Problem-2 Can we improve the solution to Problem-1 using memoization of DP?

Solution: Yes. Before finding a solution, let us see how the values are calculated.

$$T(0) = T(1) = 2$$
$$T(2) = 2 * T(1) * T(0)$$
$$T(3) = 2 * T(1) * T(0) + 2 * T(2) * T(1)$$
$$T(4) = 2 * T(1) * T(0) + 2 * T(2) * T(1) + 2 * T(3) * T(2)$$

From the above calculations it is clear that there are lots of repeated calculations with the same input values. Let us use a table for avoiding these repeated calculations, and the implementation can be given as:

```
def f2(n) :
    T = [0] * (n+1)
    T[0] = T[1] = 2
    for i in range(2, n+1):
        T[i] = 0
        for j in range(1, i):
            T[i] +=2 * T[j] * T[j-1]
    return T[n]
print (f2(4))
```

Time Complexity: O(n^2), two for loops. Space Complexity: O(n), for table.

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

Solution: Yes, since all sub problem calculations are dependent only on previous calculations, code can be modified as:

```
def f(n):
    T = [0] * (n+1)
    T[0] = T[1] = 2
    T[2] = 2 * T[0] * T[1]
    for i in range(3, n+1):
        T[i]=T[i-1] + 2 * T[i-1] * T[i-2]
    return T[n]
print (f(4))
```

Time Complexity: O(n), since only one for loop. Space Complexity: O(n).