List of Practical's

UNIT I

CHAPTER 1

Introduction to Algorithm

Syllabus

Introduction to algorithm, Why to analysis algorithm, Running time analysis, How to Compare Algorithms, Rate of Growth, Commonly Used Rates of Growth, Types of Analysis, Asymptotic Notation, Big-O Notation, Omega-Ω Notation, Theta-Θ Notation, Asymptotic Analysis, Properties of Notations, Commonly used Logarithms and Summations, Performance characteristics of algorithms, Master Theorem for Divide and Conquer, Divide and Conquer Master Theorem: Problems & Solutions, Master Theorem for Subtract and Conquer Recurrences, Method of Guessing and Confirming.

1.1 Introduction

In this chapter we are going to cover what is algorithm, why to analysis algorithms, Running time analysis, How to compare algorithms, Rate of growth, Types of analysis and asymptotic notations.

Syllabus Topic : Introduction to Algorithm

1.1.1 Algorithm

- Now consider a problem of adding two numbers. To perform addition, first we need to identify what we require and how can we perform this operation. Then we follow steps given below :
  1. Start
  2. Take three numbers a, b and c.
  3. Add a and b.
  4. Store the addition in c.
  5. Stop.
- So, what we are doing is, for a given problem, we are providing a step-by-step procedure for solving it. The formal definition of an algorithm can be stated as follows.

"An algorithm is a finite collection of well defined steps designed to solve a particular problem".

Or

"An algorithm is the step-by-step execution of a particular problem".

- An algorithm takes a set of values, as input and produces a value, or a set of values, as output.
- An algorithm may be specified as :
  - In English
  - As a computer program
  - As a pseudo-code
- An algorithm is a well-developed, organized approach to solving a complex problem.
- An algorithm is a set of instructions and instructions are steps.

Program = Data Structure + Algorithms.

1.1.2 Characteristics of Algorithm

An algorithm must have following characteristics.

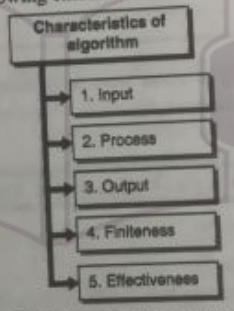


Fig. C1.1: Characteristics of algorithm

➜ **1. Input**

An algorithm must take zero or more inputs that are required for a solution of a problem.

➜ **2. Process**

An algorithm must perform certain operations on the input data.

➜ **3. Output**

An algorithm should produce certain output after processing the inputs.

➜ **4. Finiteness**

An algorithm must terminate after executing certain finite number of steps.

➜ **5. Effectiveness**

An algorithm should be simple and clear. Each step in the algorithm must be unambiguous, feasible and definite.

Syllabus Topic : Why to Analysis Algorithm?

1.2 Why to Analysis Algorithm?

Algorithms are about :

1. Solving problems.
2. Finding the better way to solve a problem without analyzing an algorithms, how do you know whether an algorithm is good or not.

- For example, To sort the numbers, we have many algorithms, like insertion sort, selection sort, quick sort, merge sort etc. But we choose the one which is effective in terms of time and speed consumed.
- In computer science, the analysis of algorithms is the determination of the amount of time, storage and/or other resources necessary to execute them.
- So, the choice of an efficient algorithm is of great importance, which can be made by considering the following factors :
  - Programming requirements of an algorithm.
  - Time requirement of an algorithm.
  - Space requirement of an algorithm.

Syllabus Topic : Running Time Analysis

1.3 Running Time Analysis

- One of the most important aspects of an algorithm is how fast it is. It is the process of determining how processing time increases as the size of the problem increases (input size). Input size is the number of elements in the input.
- It is not very easy to calculate the exact time requirements for any algorithm as it depends upon various factors like machine on which algorithms is to be executed, algorithm itself and input size of the algorithm.

- Because the processor speed in different machines may be different, So, we mainly concentrate to estimate the execution time of an algorithm irrespective to the processor/machine.

Syllabus Topic : How to Compare Algorithms?

1.4 How to Compare Algorithms?

To compare algorithms, consider following factors :

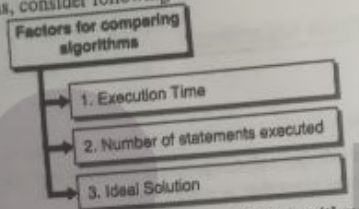


Fig. C1.2 : Factors for comparing algorithms

➔ **1. Execution time**

It is not a good measure as execution times are specific to a particular computer because processor speed in different computer may different.

➔ **2. Number of statements executed**

It is also not a good measure because the number of statements varies with the programming language as well as the style of the individual programmer.

➔ **3. 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.
- Suppose there is a statement x=x + y somewhere in the middle of the program. We wish to determine the total time that statement will spend executing, given some initial state of input data. This requires essentially two items of information, the statement frequency count(i.e. the number of times the statement will be executed) and the time for one execution. The product of these two numbers is the total time.

☞ **Example**

- Consider the three program segments (a), (b), (c).

(a)	(b)	(c)
x = x + y	For i in range(1,n): x = x + y	For i in range(1,n): For j in range(1,n): x = x + y

- Thus for segment (a), the frequency count of this statement is 1.

  For segment (b), the count is n.

  For segment (c), the count is n^2.
- These frequencies 1, n, n^2 are said to be different, increasing order of magnitude. An order of magnitude is a common notation with which we are all familiar. For example, walking, bicycling, riding in a car and flying in an airplane represents increasing orders of magnitude with respect to distance we can travel per hour.
- In connection with algorithm analysis, the order of magnitude of a statement refers to its frequency of execution, while the order of magnitude of an algorithm refers to the sum of the frequencies of all of its statements.
- Given three algorithms for solving the same problem whose order of magnitudes are n, n^2, n^3, naturally we will prefer the first one, since the second and third are progressively slower. For example, if n = 10 then these algorithms will require 10, 100 and 1000 units of time to execute respectively.

Syllabus Topic : Rate of Growth

1.5 Rate of Growth

- Running time analysis is the behavior of the algorithm in terms of input. How the algorithm is behaving when we keep input increased.
- Rate of growth is nothing but the representation of running time and space of an algorithm.
- We want to examine the rate of growth f(n) with respect to standard functions like log n, n^1, n^2, n^3, 2^n etc.

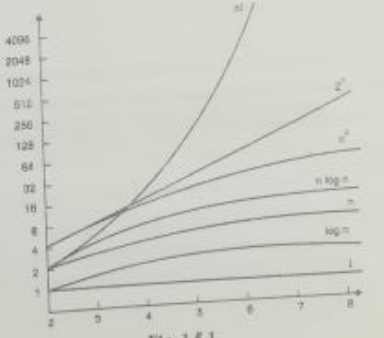


Fig. 1.5.1

Syllabus Topic : Commonly used Rates of Growth

1.6 Commonly Used Rate of Growth

The most common computing times for algorithms are as follows.

Time Complexity	Name	Example
1	Constant	Adding an element to front of linked list.
log n	Logarithmic	Finding an element in a sorted array.
n	Linear	Finding an element in an unsorted array.
n log n	Linear Logarithmic	Sorting n times by divide and conquer.
n^2	Quadratic	Shortest path between two nodes in a graph.
n^3	Cubic	Matrix multiplication.
2^n	Exponential	The towers of Hanoi problem.

Here,

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$.

n	O(log n)	O(n)	O(n log n)	$O(n^2)$	$O(n^3)$
8	3	8	24	64	512
16	4	16	64	256	4096
32	5	32	160	1024	32768
64	6	64	384	4096	262144
128	7	128	896	16384	2097152
256	8	256	2048	262144	67108864

Rate of growth of some standard functions with the size of the input.

Syllabus Topic : Types of Analysis

1.7 Types of Analysis

- To analyze the given algorithm, we need to know with which inputs the algorithm takes less time and with which inputs the algorithm takes a long time. That means we represent the algorithm with multiple expressions : One case which takes less time and another case which takes more time.
- There are three types of analysis.

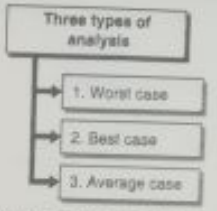


Fig. C1.3 : Types of Analysis

➜ **1. Worst case**

If the running time of the algorithm is longest for all the inputs then it is called worst case. In this type, the key operations are executed maximum number of times.

➜ **2. Best case**

If the running time of the algorithm is shortest for all the inputs then it is called best case. In this type, the key operations are executed minimum number of times.

➜ **3. Average case**

If the running time of the algorithm falls between the worst case and the best case then it is called average case. To calculate the average case complexity of an algorithm, we have to take some assumptions.

☞ **Example**

- For example, searching for an element in an unsorted array of length N.
- If we found the desired element at first position then the number of comparisons will be 1. So it is **Best case.**
- If we found the desired element at last position then the number of comparisons will be N. So it is **Worst case.**
- If we assume it is found roughly in the middle portion of the array then we need to do N/2 comparisons to get the desired element, so it is **Average case.**

Syllabus Topic : Asymptotic Notation

1.8 Asymptotic Notations

- Asymptotic Notations are languages that allows us to analyze an algorithm's running time by identifying its behavior as the input size for the algorithm increases.
- It is used to find out which algorithm is better for the same problem statement.

- Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm:

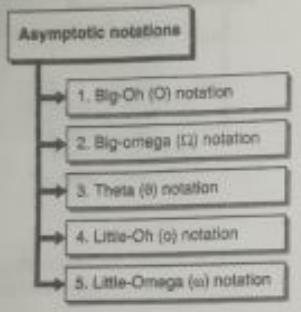


Fig. C1.4 : Asymptotic Notations

- All the above notations are used to express the complexity of an algorithm.
- "Complexity can be defined as the rate at which the storage or time requirement grows as a function of the problem size."
- Complexity is the time and space requirement of the algorithm. If the time and space requirement of the algorithm is more, then complexity of the algorithm is more and if the time and space requirement of the algorithm is less, then complexity of the algorithm is less.
- Out of the two factors, time and space, the space requirement is of the algorithm is not a very important because it is available at very low cost.
- Only the time requirement of the algorithm is considered as an important factor to find the complexity. Because of the importance of time in finding the complexity, it is sometimes termed as **time complexity**.

➜ 1.8.1 Big-Oh(O) Notation

- Big-O notation gives the tight upper bound of the function. It measure the worst case time complexity or the longest amount of time an algorithm possibly take to complete.
- This means, a function f(x) is Big-O of function g(x) and there exists the positive constants c and $n_0$such that,

$$f(n) <= c \times g(n)$$

where $c > 0$ and $n_0 > 1$ the for all $n >= n_0$

$$f(n) = O(g(n))$$

- Here, f(x) and g(x) are the functions of the non-negative integers.

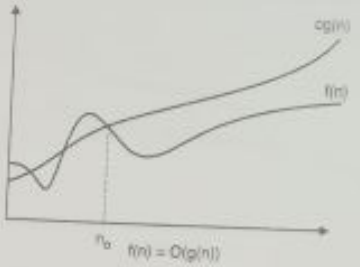


Fig. 1.8.1

Example 1.8.1

Consider the functions, f(n)=2n + 4 and g(n)=n then prove that f(n) = O(g(n)).

Solution :

$$f(n) <= c*g(n) \quad \text{for } n > 0,$$

For some $c > 0$ and $n_0 > 1$, substitute $f(n) = 2n + 4$

$$2n + 4 <= c(n)$$

When can be 2n + 4 less than or equal to c(n).

For above case the values of c can be anything above 3 is better and $n_0 = 3$.

$$2n + 4 <= c(n)$$

$$2n + 4 <= 3n$$

$$4 <= 3n - 2n$$

$$4 <= n$$

It means, for every n>=4 at c=3, f(n) <= c*g(n).

➜ 1.8.2 Big-Omega(Ω) Notation

- Big-Ω notation gives tighter lower bound of the given function. It measures the best case time complexity or the best amount of time an algorithm possibly take to complete.
- This means, a function f(x) is Big-Ω of function g(x) and there exists the positive constants c and $n_0$such that,

$f(n) >= c \times g(n)$ where $c > 0$ and $n_0 > 1$ the for all $n >= n_0$

$$f(n) = \Omega(g(n))$$

Here, f(x) and g(x) are the functions of the non-negative integers.

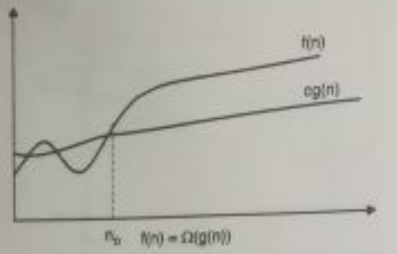


Fig. 1.8.2

Example 1.8.2

Consider the functions, f(n)=3n+2 and g(n)=n then prove that f(n) = Ω(g(n)).

Solution :

$$f(n) > = c*g(n) \quad \text{for } n > 0,$$

For some $c > 0$ and $n_0 > 1$, substitute $f(n) = 3n + 2$

$$3n + 2 < = c(n)$$

When can be 3n + 2greater than or equal to c(n),

For above case the values of c can be anything above 4 is better.

$$3n + 2 > = c(n)$$

$$3n + 2 > = 3n, \quad \text{for } n >= 2$$

Put n = 2 then 8 >= 6.

It means, 3n + 2 is lower bounded.

1.8.3 Theta(Θ) Notation

- Theta(Θ) notation lies between upper bound and lower bound of an algorithm. Using this we can compute the average amount of time taken by any algorithm.
- This means a function f(x) is Theta of function g(x) and there exists three positive constants c1, c2 and $n_0$such that,

$$0 < = c1 * g(n) <= f(n) >= c2 * g(n) \quad \text{for all } n > n_0$$

- Here, f(x) and g(x) are the functions of nonnegative integers.

It may be noted that f(n) = Θ(g(n)) if and only if ,

f(n) = O(g(n)) and f(n) = Ω (g(n)).

- It means after some input n_0, the value of c1 * g(n) should be less than or equal to f(n) and c2 * g(n) should be greater than or equal to f(n).

Example 1.8.3

For example, let f(n)=3n+2 and g(n)=n then prove that f(n) =Θ (g(n)).

Solution :

As 3n+2 > = 3n for all n >=2 and

3n+2 < = 4n for all n<=2

Here, $c_1 = 3$, $c_2 = 4$ and $n_0 = 2$ So f(n) = Θ(g(n)).

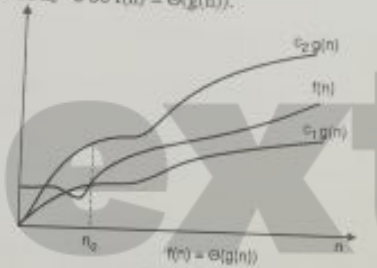


Fig. P.1.8.3

1.8.4 Little-Oh (o) Notation

- Big-O is used as a tight upper-bound on the growth of an algorithm's effort, even though, as written, it can also be a loose upper-bound. "Little-o" (o()) notation is used to describe an upper-bound that cannot be tight.
- We say that f(n) is o(g(n)) if for any real constant $c > 0$, there exists an integer constant $n_0 \geq 1$ such that $f(n) < c * g(n)$ for every integer $n \geq n_0$.

$$0 < = f(n) < c \times g(n)$$

Example

$2n$ is $o(n^2)$ as $2n <= n^2$ for all $n >= 2$ and $c >= 1$.

1.8.5 Little-Omega (ω) Notation

- We use ω notation to denote a lower bound that is not asymptotically tight.
- We say that f(n) is ω(g(n)) if for any real constant $c > 0$, there exists an integer constant $n_0 \geq 1$ such that $f(n) > c \times g(n) \geq 0$ for every integer $n \geq n_0$.

$f(n) > = c \times g(n)$.

Example

$n^2/2$ is $\omega(n)$ as $n^2/2 >= n^2$ for all $n >= 2$ and $c >= 1$.

Syllabus Topic : Asymptotic Analysis

1.9 Asymptotic Analysis

- Asymptotic analysis of an algorithm refers to defining the mathematical boundation / framing of its run time performance. Using Asymptotic Analysis, we can very well conclude the best case, average case and worst case scenario of an algorithm.
- Asymptotic Analysis is input bound that means if there is no input to the algorithm, it is concluded to work in a constant time other than the "input" all other factors are considered constant.
- Asymptotic Analysis refers to computing the running time of any operation in mathematical units of computation. For example, the running time of one operation is computed as $f(n)$ and maybe for another operation it is $g(n^2)$. This means the first operation running time will increase linearly with the increase in n and the second operation will increase exponentially when n increases. Similarly, the running time of both operations will be linearly the same if n is significantly small.

1.9.1 Guidelines for Finding out the Time Complexity of a Piece of Code

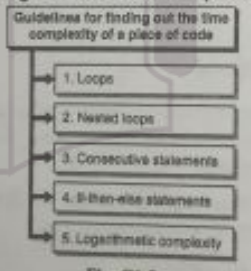


Fig. C1.5

1. Loops

The running time of loop is the running time of the statements inside the loop multiplied by the number of iterations.

```
# executes n times
For i in range (0, n):
    Print ' Current Number:', i          ← Constant time
Therefore, total time = a constant c * n = cn = O(n).
```


2. Nested loops

Analyze inside out. Total running time is the product of the size 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 tim
Therefore, total time = c * n * n
= cn² = O(n²).
```

3. Consecutive statements

Add the time complexities of each statements

```
n = 100
#executes n times
fori in range(0,n):
print 'Current Number:', i                    #constant time
#outer loop executed n times
fori 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
Therefore,
Total time = c0 + c1n + c2n² = O(n²).
```

4. If-then-else statements

Worst-case running time: the test, plus either thenpart or the else part(whichever is the larger).

```
if n == 10:
print "Wrong Value"                  #constant time
print n
else:
fori in range(0,n):                  #n times
print 'Current Number:', I           #constant time
Therefore,
Total time = c0 + c1 * n = o(n).
```

5. Logarithemetic complexity

An algorithm is O(log n) if it takes a constant time to cut the problem size by a fraction (usually by 1/2).

```
def Logarithms(n):
i =1
whilei<= n:
i= i * 2
printi
Logarithms(100)
```

If we observe carefully, the value of i is doubling every lime. 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 limes.

At k^{th} step $2^k = n$ and we come out of loop.

Taking logarithm on both sides,

```
Log(2^k) = log n
klog2 = logn
k = log2n          //if we assume base-2
Total time = O(log2 n).
```

Syllabus Topic : Properties of Notations

1.10 Properties of Notations

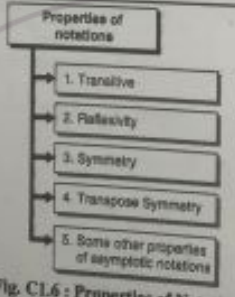


Fig. C1.6 : Properties of Notations

1. Transitive

- If f(n) = Θ(g(n)) and g(n) = Θ(h(n)), then f(n) = Θ(h(n))
- If f(n) = O(g(n)) and g(n) = O(h(n)), then f(n) = O(h(n))
- If f(n) = o(g(n)) and g(n) = o(h(n)), then f(n) = o(h(n))
- If f(n) = Ω(g(n)) and g(n) = Ω(h(n)), then f(n) = Ω(h(n))
- If f(n) = ω(g(n)) and g(n) = ω(h(n)), then f(n) = ω(h(n))

2. Reflexivity

- f(n) = Θ(f(n))
- f(n) = O(f(n))
- f(n) = Ω(f(n))

3. Symmetry

f(n) = Θ(g(n)) if and only if g(n) = Θ(f(n))

4. Transpose Symmetry

- f(n) = O(g(n)) if and only if g(n) = Ω(f(n))
- f(n) = o(g(n)) if and only if g(n) = ω(f(n))

5. Some other properties of asymptotic notations

- If f (n) is O(h(n)) and g(n) is O(h(n)), then f (n) + g(n) is O(h(n)).
- The function $\log_a n$ is $O(\log_b n)$ for any positive numbers a and b ≠ 1.
- $\log_a n$ is O(log n) for any positive a ≠ 1, where log n = $\log_2 n$.

Syllabus Topic : Commonly Used Logarithms and Summations

1.11 Commonly Used Logarithms and Summations

Logarithms

$$\log x^y = y \log x \qquad \log xy = \log x + \log y$$
$$\log n = \log^a{}_{10} \qquad \log^k n = (\log n)^k$$
$$\log n = \log(\log n) \qquad \log x/y = \log x - \log y$$
$$a^{\log_b x} = x^{\log_b a} \qquad \log^x{}_b = \log^x{}_a / \log^b{}_a$$

Arithmetic series

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

Geometric series

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

☞ **Harmonic series**

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

☞ **Other important formulae**

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

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

Syllabus Topic : Performance Characteristics of Algorithms

1.12 Performance Characteristics of Algorithms

Performance analysis of an algorithm depends upon two factors i.e. amount of memory used and amount of compute time consumed on any CPU. Formally they are notified as complexities in terms of :

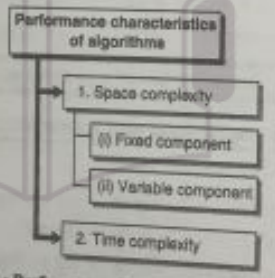


Fig. C1.7 : Performance Characteristics of Algorithms

➔ 1.12.1 Space Complexity

Space Complexity of an algorithm is the amount of memory it needs to run to completion i.e. from start of execution to its termination. Space need by any algorithm is the sum of following components.

(i) Fixed Component

This is independent of the characteristics of the inputs and outputs. This part includes: Instruction Space, Space of simple variables, fixed size component variables and constants variables.

(ii) Variable Component

This consist of the space needed by component variables whose size is dependent on the particular problems instances(Inputs/Outputs) being solved, the space needed by referenced variables and the recursion stack space is one of the most prominent components. Also this included the data structure components like Linked list, heap, trees, graphs etc.

- Therefore the total space requirement of any algorithm 'A' can be provided as :

  Space(A) = Fixed Components(A) + Variable Components(A)

- Among both fixed and variable component the variable part is important to be determined accurately, so that the actual space requirement can be identified for an algorithm 'A'. To identify the space complexity of any algorithm following steps can be followed :

1. Determine the variables which are instantiated by some default values.
2. Determine which instance characteristics should be used to measure the space requirement and this is will be problem specific.
3. Generally the choices are limited to quantities related to the number and magnitudes of the inputs to and outputs from the algorithms.
4. Sometimes more complex measures of the interrelationships among the data items can used.

☞ **Example of Space Complexity**

Space Complexity

```
Algorithm Sum(number, size)                    \\ procedure will produce sum of all numbers
                                               \\provided in 'number' list
{
    result=0.0;
    for count = 1 to size do                   \\will repeat from 1,2,3,4,....size times
        result= result + number[coont];
    return result;
}
```

- In above example, when calculating the space complexity we will be looking for both fixed and variable components.
- Here we have **Fixed components** as 'result','count' and 'size' variable there for total space required is three(3) words.
- **Variable components** is characterized as the value stored in 'size' variable (suppose value store in variable 'size 'is 'n').

- Because this will decide the size of 'number' list and will also drive the for loop. Therefore if the space used by size is one word then the total space required by 'number' variable will be 'n' (value stored in variable 'size').
- Therefore the space complexity can be written as **Space(Sum) = 3 + n;**

1.12.2 Time Complexity

- **Time Complexity** of an algorithm (basically when converted to program) is the amount of computer time it needs to run to completion.
- The time taken by a program is the sum of the compile time and the run/execution time. The compile time is independent of the instance (problem specific) characteristics. following factors effect the time complexity :
  1. Characteristics of compiler used to compile the program.
  2. Computer Machine on which the program is executed and physically clocked.
  3. Multiuser execution system.
  4. Number of program steps.
- Therefore the again the time complexity consist of two components fixed(factor 1 only) and variable/instance(factor 2,3 and 4), so for any algorithm 'A' it is provided as:

  Time (A) = Fixed Time(A) + Instance Time(A)

- Here the number of steps is the most prominent instance characteristics and The number of steps any program statement is assigned depends on the kind of statement like :
  - Comments count as zero steps,
  - An assignment statement which does not involve any calls to other algorithm is counted as one step,
  - For iterative statements we consider the steps count only for the control part of the statement, etc.
- Therefore to calculate total number program of program steps we use following procedure.
- For this we build a table in which we list the total number of steps contributed by each statement. This is often arrived at by first determining the number of steps per execution of the statement and the frequency of each statement executed.
- This procedure is explained using an example.

Example of Time Complexity

Statement	Steps per execution	Frequency	Total Steps
Algorithm Sum(number,size)	0	-	0
{	0	-	0
result=0.0;	1	1	1

Statement	Steps per execution	Frequency	Total Steps
for count = 1 to size do	1	size+1	size + 1
result= result + number[count];	1	size	size
return result;	1	1	1
}	0	-	0
Total			2size + 3

- In above example if you analyze carefully frequency of "for count = 1 to size do" it is 'size +1' this is because the statement will be executed one time more die to condition check for false situation of condition provided in for statement.
- Now once the total steps are calculated they will resemble the instance characteristics in time complexity of algorithm.
- Also the repeated compile time of an algorithm will also be constant every time we compile the same set of instructions so we can consider this time as constant 'C'.
- Therefore the time complexity can be expressed as: **Time(Sum) = C + (2size +3).**
- So in this way both the Space complexity and Time complexity can be calculated. Combination of both complexities comprises the Performance analysis of any algorithm and cannot be used independently.
- Both these complexities also helps in defining parameters on basis of which we optimize algorithms.

Syllabus Topic : Master Theorem for Divide and Conquer

1.13 Master Theorem for Divide and Conquer

- In divide and conquer approach, the problem is divided into smaller sub-problems and then each problem is solved independently. The solution of all sub-problems is finally merged in order to obtain the solution of an original problem.
- The master theorem concerns recurrence relations of the form:

  $T(n) = aT(n/b) + f(n)$ where $a >= 1$ and $b > 1$

- In the application to the analysis of a recursive algorithm, the constants and function take on the following significance:
  - n is the size of the problem.
  - a is the number of subproblems in the recursion.
  - n/b is the size of each subproblem. (Here it is assumed that all subproblems are essentially the same size.)

- f (n) is the cost of the work done outside the recursive calls, which includes the cost of dividing the problem and the cost of merging the solutions to the subproblems.

- 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,

$$T(n) = aT(n/b) + \Theta(n^k \log^p n), \text{ where } a >= 1, b > 1, k >= 0,$$

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 $p >= 0$ then $T(n) = \Theta(n^k \log^p n)$
   b) If $p < 0$ then $T(n) = O(n^k)$

Syllabus Topic : Master Theorem for Divide and Conquer Problems and Solutions

1.13.1 Master Theorem for Divide and Conquer : Problems and Solutions

Example 1.13.1

Find Recurrence for binary search : $T(n) = T(n/2) + 1$.

Solution :

Here, $a = 1, b = 2, k = 0$, Hence $a = b^k$

By case 2 of Master theorem, $T(n) = \Theta(n^0 \log n) = \Theta(\log n)$

Example 1.13.2

Find Recurrence for merge sort: $T(n) = 2 \cdot T(n/2) + 4n$

Solution :

Here, $a = 2, b = 2, k = 1$, Hence $a = b^k$

By Case 2 of Master theorem, $T(n) = \Theta(n \cdot \log n)$

Example 1.13.3

Find Recurrence for binary search : $T(n) = 4\,T(n/2) + n$

Solution :

Here, $a = 4, b = 2, k = 1$, Hence $a > b^k$

By Case 1 of Master theorem, $T(n) = \Theta(n^2)$

Example 1.13.4

Find Recurrence for binary search : $T(n) = 4\,T(n/2) + n^2$

Solution :

Here, $a = 4, b = 2, k = 2$, Hence $a = b^k$

By Case 2(a) of Master theorem, $T(n) = \Theta(n^2 \log n)$

Example 1.13.5

Find Recurrence for binary search : $T(n) = 4\,T(n/2) + n^3$

Solution :

Here, $a = 4, b = 2, k = 3$, Hence $a < b^k$

By Case 3(a) of Master theorem, $T(n) = \Theta(n^3)$

Example 1.13.6

Find Recurrence for binary search : $T(n) = T(n/2) + n^2$

Solution :

Here, $a = 1, b = 2, k = 2$, Hence $a < b^k$

By Case 3(a) of Master theorem, $T(n) = \Theta(n^2)$

Example 1.13.7

Find Recurrence for binary search : $T(n) = 2^n\,T(n/2) + n^n$

Solution :

Does not apply (a is not constant)

Example 1.13.8

Find Recurrence for binary search : $T(n) = 2T(n/2) + n \log n$

Solution :

Here, $a = 2, b = 2, k = 1$, Hence $a = b^k$

By Case 2(a) of Master theorem, $T(n) = \Theta(n \log^2 n)$

Example 1.13.9

Find Recurrence for binary search : $T(n) = 2T(n/2) + n/\log n$

Solution :

Here, $a = 2, b = 2, k = 1$, Hence $a = b^k$

By Case 2(b) of Master theorem,

$$T(n) = \Theta(n \log \log n)$$

Example 1.13.10

Find Recurrence for binary search : $T(n) = 2T(n/4) + n^{0.51}$

Solution : Here, $a = 2$, $b = 4$, $k = 0.51$, Hence $a < b^k$

By Case 3(b) of Master theorem, $T(n) = \Theta(n^{0.51})$

Example 1.13.11

Find Recurrence for binary search : $T(n) = 0.5T(n/2) + 1/n$

Solution :

Does not apply ($a < 1$)

Example 1.13.12

Find Recurrence for binary search : $T(n) = 64T(n/8) - n^2 \log n$

Solution :

Does not apply (function is not positive)

Example 1.13.13

Find Recurrence for binary search : $T(n) = 3T(n/3) + n/2$

Solution :

Here, $a = 3$, $b = 3$, $k = 1$, Hence $a = b^k$

By Case 2(a) of Master theorem, $T(n) = \Theta(n \log n)$

Example 1.13.14

Find Recurrence for binary search : $T(n) = 3T(n/3) + \sqrt{n}$

Solution :

Here, $a = 3$, $b = 3$, $k = 1$, Hence $a = b^k$

By Case 1 of Master theorem, $T(n) = \Theta(n)$

Example 1.13.15

Find Recurrence for binary search : $T(n) = 16T(n/4) + n!$

Solution :

Here, $a = 16$, $b = 4$, $k = 1$, Hence $a > b^k$

By Case 3(a) of Master theorem, $T(n) = \Theta(n!)$

Example 1.13.16

Find Recurrence for binary search : $T(n) = T(n/2) + 4$

Solution :

Here, $a = 1$, $b = 2$, $k = 0$, Hence $a = b^k$

By Case 2 of Master theorem, $T(n) = \Theta(\log n)$

Syllabus Topic : Master Theorem for Subtract and Conquer Recurrences

1.14 Master Theorem for Subtract and Conquer Recurrences

Let $T(n)$ be a function defined on positive n as shown below :

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

for some constants c, a>0, b>0, k>=0and function f(n).

If $f(n)$ is $O(n^k)$, then

1. If $a<1$ then $T(n) = O(n^k)$
2. If $a=1$ then $T(n) = O(n^{k+1})$
3. if $a>1$ then $T(n) = O(n^k a^{n/b})$

Let's try some example :

Example 1.14.1

$T(n) = 3T(n-1)$ when a>0

$= 1$ otherwise

Solution :

Here $a = 3, b = 1$ and $d = 0$ hence $T(n) = O(n^0 3^{n/1})$ $(a > 1)$

that means $T(n) = O(3^n)$.

Example 1.14.2

$T(n) = 5T(n-3) + O(n^2)$ when n>0

$= 1$ otherwise

Solution :

Here $a = 5$, $b = 3$, $d = 2$ hence $T(n) = O(n^2 5^{n/3})$.

Example 1.14.3

$T(n) = 2T(n-1) - 1$, when n>0

$= 1$, when n<=0

Solution :

This recurrence can't be solved using above method since function is not of form

$$T(n) = aT(n-b) + f(n)$$

Syllabus Topic : Method of Guessing and Confirming

1.15 Method of Guessing and Confirming

- Let us discuss about 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 theorems) 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 divide and conquer method (diving the problem into $\sqrt{n}$ sub-problems each with size $\sqrt{n}$). As we can see, the size of the sub-problems at the first level of recursion is n.
- So, let us guess that $T(n) = O(n \log n)$, and then try to prove that our guess is correct.
- Let's start by trying to prove an upper bound $T(n) \le c\, n \log n$:

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

- The last inequality assumes only that $1 \le c.1/2.\log n$. 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 upper bound. Now, let us prove the lower bound for this recurrence.

$$\begin{aligned} T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\ &\ge \sqrt{n}\, k \sqrt{n} \log \sqrt{n} + n \end{aligned}$$

$$\begin{aligned} &= n.\, k \log \sqrt{n} + n \\ &= n.k.1/2.\log n + n \\ &\ge k\, n \log n \end{aligned}$$

- The last inequality assumes only that $1 \ge k.1/2.\log n$. This is incorrect if n is sufficiently large and for any constant k. From the above proof, we can see that our guess is incorrect for lower bound.
- From the above discussion, we understood that $\Theta(n \log n)$ is too big. How about $\Theta(n)$? The lower bound is easy to prove directly :

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

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

$$\begin{aligned} T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\ &\le \sqrt{n}.c.\sqrt{n} + n \\ &= n.\, c + n \\ &= n\,(c+1) \\ &\le c\, n \end{aligned}$$

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

Proving upper bound for $n \sqrt{\log n}$:

$$\begin{aligned} T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\ &\le \sqrt{n}.c.\sqrt{n}\left(\sqrt{\log \sqrt{n}}\right) + n \\ &= n.\, c.1/\sqrt{2}\, \log \sqrt{n} + n \\ &\le c\, n \log \sqrt{n} \end{aligned}$$

Proving lower bound for $n \sqrt{\log n}$:

$$\begin{aligned} T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\ &\ge \sqrt{n}.\, k.\, \sqrt{n}\left(\sqrt{\log \sqrt{n}}\right) + n \\ &= n.\, k.\, 1/\sqrt{2} \log \sqrt{n} + n \\ &\ge k\, n \log \sqrt{n} \end{aligned}$$

The last step doesn't work. So, $\Theta(n \sqrt{\log n})$ doesn't work. What else is between n and n

log n? How about n log log n?

Proving upper bound for n log log n :

$$T(n) = \sqrt{n}\,T(\sqrt{n}) + n$$
$$\leq \sqrt{n}.c.\sqrt{n}\sqrt{n}\ \log\log\sqrt{n} + \sqrt{n} + n$$
$$= n.\ c.\log\log n - c.\ n + n$$
$$\leq c\ n \log\log n,\ \text{if } c \geq 1$$

Proving lower bound for n log log n:

$$T(n) = \sqrt{n}\,T(\sqrt{n}) + n$$
$$\geq \sqrt{n}.\ k.\ \log\log\sqrt{n} + n$$
$$= n.\ k.\log\log n - k.\ n + n$$
$$\geq k\ n \log\log n,\ \text{if } k \leq 1$$

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

Review Questions

Q. 1 What is algorithm ? **(Refer section 1.1.1)**

Q. 2 List and explain characteristics of an algorithm. **(Refer section 1.1.2)**

Q. 3 List and explain various asymptotic notation **(Refer section 1.8).**

Q. 4 Explain different performance characteristics of algorithm. **(Refer sections 1.12, 1.12.1 and 1.12.2)**

❑❑❑

UNIT II

CHAPTER 2

Tree Algorithms

Syllabus

What is a Tree? Glossary, Binary Trees, Types of Binary Trees, Properties of Binary Trees, Binary Tree Traversals, Generic Trees (N-ary Trees), Threaded Binary Tree Traversals, Expression Trees, Binary Search Trees (BSTs), Balanced Binary Search Trees, AVL (Adelson-Velskii and Landis) Trees.

Syllabus Topic : What is a Tree ?

2.1 Tree

- In linear data structure, data is organized in sequential order and in non-linear data structure, data is organized in random order.
- Tree is a very popular data structure used in wide range of applications. A tree data structure can be defined as follows.
- "Tree is a non-linear data structure which organizes data in hierarchical structure."
- Tree represents the nodes connected by edges.

Syllabus Topic : Glossary

2.1.1 Glossary of Tree

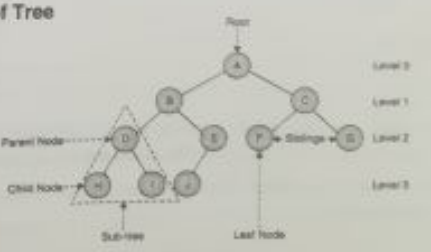


Fig. 2.1.1

Root

- In a tree data structure, the first node is called as **Root Node**. Every tree must have root node.
- Root node is the origin of tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree.

Edge

- An edge is another fundamental part of a tree. In a tree data structure, the connecting link between any two nodes is called as **EDGE**.
- In a tree with 'N' number of nodes there will be a maximum of **'N-1'** number of edges.

Node

Each element in the hierarchical representation is called a node. In the above tree A, B, C, D, E, F, G, H, I, J, K all are nodes.

Path

- Path between any two nodes in a tree is a sequence of distinct nodes in which successive nodes are connected by edges.
- **Length of a Path** is total number of nodes in that path. In below example **the path A - B - E - J has length 4**.

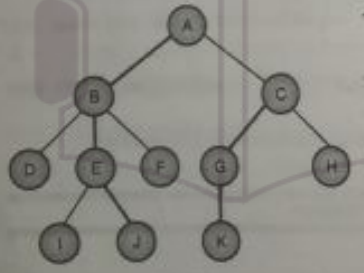


Fig. 2.1.2

- In any tree, 'Path' is a sequence of nodes and edges between to nodes.
- Here, 'path' between A and J is A – B – E– J.
- Here, 'path' between C and K is C – G – K.

Parent

- In a tree data structure, the node which is predecessor of any node is called as **PARENT NODE**. Parent node can also be defined as **"The node which has child / children"**.
- The root node is the only node which does not have a parent. In the above tree, nodes A, B, C, D, E are parent nodes.

Child

- The immediate successor of a node is called as **CHILD Node**.
- In a tree, any parent node can have any number of child nodes. In the above tree, D and E are childrens of node B.

Siblings

- Children of the same Parent are called as **SIBLINGS**. In simple words, the nodes with same parent are called as Sibling nodes.
- In the above tree, F and G are siblings, also D and E are siblings.

Subtree

A subtree is a set of nodes and edges comprised of a parent and all the descendants of that parent.

Leaf node

- The nodes which does not have any child is called as LEAF Node.
- Leaf nodes are also called as terminal nodes.
- In the above tree, F, G, H, I, J are the leaf nodes.

Internal nodes

- A node of a tree that has one or more child is called as **INTERNAL Node**.
- **The root node is also said to be Internal Node** if the tree has more than one node. Internal nodes are also called as **'Non-Terminal'** nodes.

External nodes

The leaf nodes are also called as **External Nodes**. In the above tree, F, G, H, I, J are the external nodes.

Degree of a node

- In a tree data structure, the total number of children of a node is called as **DEGREE** of that Node. In simple words, the Degree of a node is total number of children it has.
  - In above tree degree of a node A is 2.
  - In above tree degree of a node B is 2.
  - In above tree degree of a node C is 2.
  - In above tree degree of a node E is 1.

Degree of a tree

The highest degree of a node among all the nodes in a tree is called as **'Degree of Tree'**. The degree of above tree is 2.

Level

In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on.

Height or Depth of a node

- The height of any node is the length of longest path from that node to a terminal node is called as **HEIGHT** of that Node.
- In a tree, height of the root node is said to be **height of the tree**. In a tree, **height** of all **leaf nodes is '0'**.

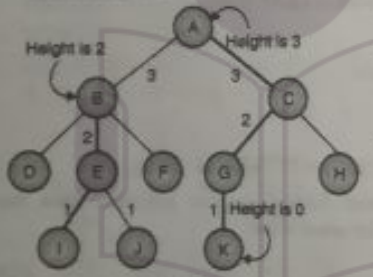


Fig. 2.1.3

- Here, Height of tree is 3
- In any tree, 'Height of Node' is total number of edges from leaf to that node in longest path.
- In any tree, 'Height of tree' is the height of the root node.

Depth of a tree

- The total number of edges from root node to a leaf node at the last level is said to be Depth of the tree.
- Depth or Height is same for the tree but the difference between these is height is always measured from leaf node to root node while depth is measured from root node to leaf node.

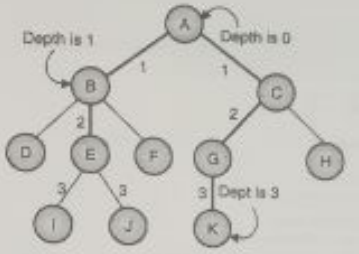


Fig. 2.1.4

- Here, Depth of tree is 3
- In any tree, 'Depth of Node' is total number of edges from edges from root to that node.
- In any tree, 'depth of tree' is total number of edges from root to leaf in the longest path.

Syllabus Topic : Binary Trees

2.2 Binary Trees

- Binary tree is defined on a finite set of nodes that either:

  a. Contains no nodes.

  b. Composed of three disjoint set of nodes: a root node, a binary tree called its left subtree, and a binary tree called its right subtree.
- "A tree in which every node can have a maximum of two children is called as Binary Tree."
- In a binary tree have the property that it can have either 0, 1 or 2 children but not more than 2 children. A binary tree may be empty known as **Null Tree.**

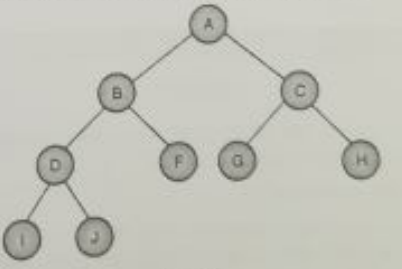


Fig. 2.2.1

How to represent a binary tree ?

A binary tree data structure is represented using two methods. Those methods are as follows.

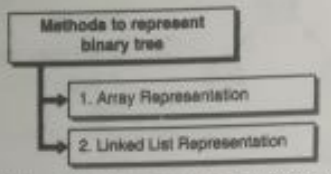


Fig. C2.1 : Methods of representation of binary tree

Consider the following binary tree.

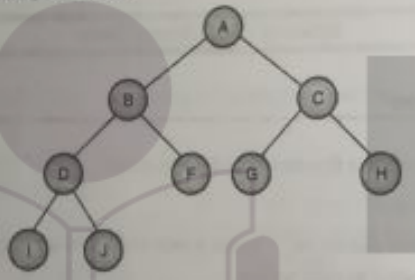


Fig. 2.2.2

1. Array Representation

- In array representation of binary tree, we use a one dimensional array (1-D Array) to represent a binary tree. Consider the above example of binary tree and it is represented as follows.

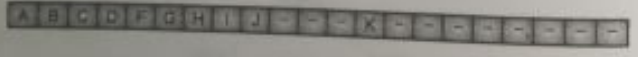


Fig. 2.2.3

- To represent a binary tree of depth 'n' using array representation, we need one dimensional array with a maximum size of $2^{n+1} - 1$.

2. Linked List Representation

- We use double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.

- In this linked list representation, a node has the following structure.

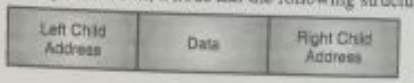


Fig. 2.2.4

- The above example of binary tree represented using Linked list representation is shown as follows.

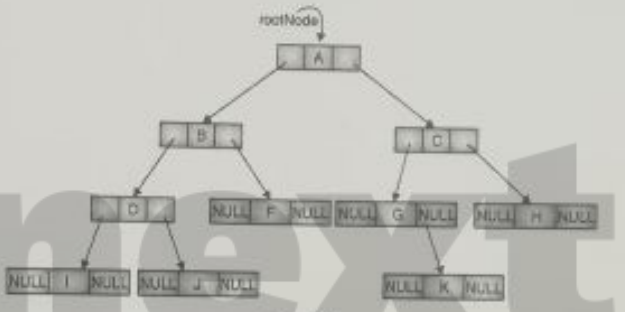


Fig. 2.2.5

Code to define node of Binary tree

- We need to represent a tree node. To do that, we create a new class named Node with 3 attributes.
  - Left node
  - Right node
  - Node's data

```
class Node(object)
#Tree node: left and right child + data which can be any object.
        def __init__(self, data):
        #Node Constructor
        #@param data node data object
        self.left = None
        self.right = None
        self.data = data
```

2.2.1 Implementation of Binary Tree Class and its Methods

```
class BinaryTree():
def __init__(self,root):
self.left = None
self.right = None
self.root = rootid
def getLeftChild(self):
returnself.left
def getRightChild(self):
returnself.right
defsetNodeValue(self,value):
self. rootid = value
def getNodeValue(self):
returnself.rootid
def insertRight(self,newNode):
ifself.right == None:
self.right = BinaryTree(newNode)
else:
tree = BinaryTree(newNode)
tree.right = self.right
self.right = tree
def insertLeft(self,newNode):
ifself.left == None:
self.left = BinaryTree(newNode)
else:
tree = BinaryTree(newNode)
tree.left = self.left
self.left = tree
def printTree(tree):
if tree != None:
printTree(tree.getLeftChild())
print(tree.getNodeValue())
printTree(tree.getRightChild())
```



```
# test tree
deftestTree():
myTree = BinaryTree("Maud")
myTree.insertLeft("Bob")
myTree.insertRight("Tony")
myTree.insertRight("Steven")
printTree(myTree)
```

2.2.2 Applications of Binary Trees

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

Syllabus Topic : Types of Binary Trees

2.3 Types of Binary Trees

There are different types of binary trees :

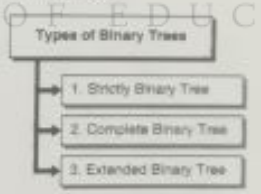


Fig. C2.2 : Types of Binary Trees

➜ 1. Strictly Binary Tree

- A binary tree is a strictly binary tree in which every internal node should have exactly two children or none.
- Strictly binary tree is used to represent algebraic expressions where non-leaf nodes represent operators and leaf nodes represent operands.
- Strictly binary tree data structure is used to represent mathematical expressions.

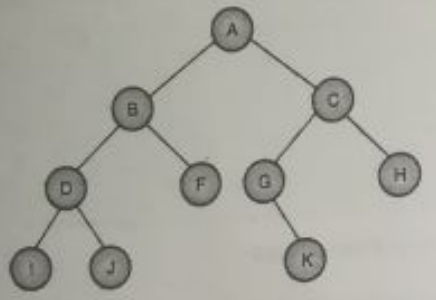


Fig. 2.3.1

2. Complete Binary Tree

- A binary tree in which every internal node has exactly two children that means all internal nodes have degree 2 and all leaf nodes are at same level is called Complete Binary Tree.
- It is also termed as Perfect Binary Tree.
- In complete binary tree, if there are n nodes at level l then at level i+1, there are 2^i nodes.

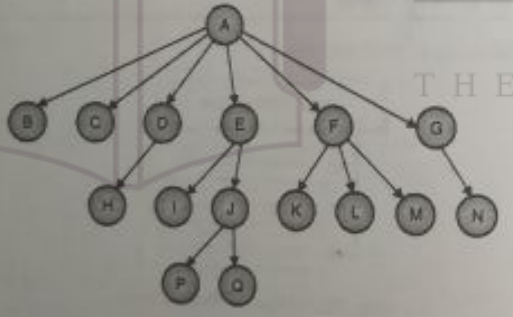


Fig. 2.3.2

3. Extended Binary Tree

- An extended binary tree is a transformation of any binary tree into a complete binary tree. This transformation consists of replacing every null subtree of the original tree with "special nodes". These special nodes are called as dummy nodes.
- The full binary tree obtained by adding dummy nodes to a binary tree is called as Extended Binary Tree.

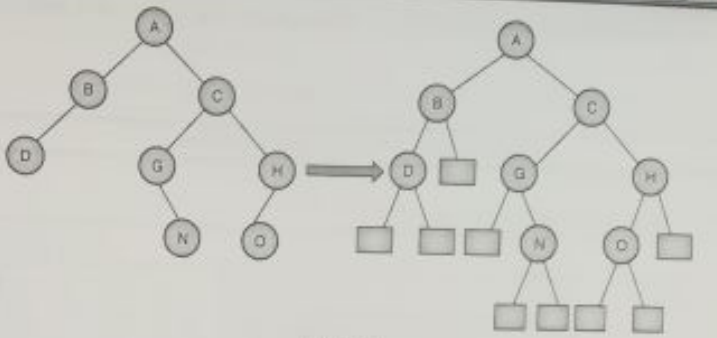


Fig. 2.3.3

- In above tree the nodes which are represented by rectangle are the dummy nodes.

Syllabus Topic : Properties of Binary Trees

2.4 Properties of Binary Trees

- A binary tree with n nodes has exactly n – 1 nodes.
- In a binary tree every node except the root node has exactly one parent.
- In a binary tree, there is exactly one path connecting any two nodes in the tree.
- The maximum number of nodes in a binary tree of height h is $2^{h+1} - 1$. For example, number of nodes in a binary tree of height 4 will be $2^{4+1} - 1 = 31$.
- The minimum number of nodes in a binary tree of height h is h+1.

(a) Full tree (b) Complete tree

Fig. 2.4.1

- Number of leaf nodes in a complete binary tree is (n + 1)/2.
- In a complete binary tree, Number of external nodes = Number of internal nodes + 1.

- In a complete binary tree, if there are n nodes at level 'l' then at level 'l+1', there are $2n$ nodes.
- A full binary tree. is a binary tree in which each node has exactly zero or two children.

Syllabus Topic : Binary Trees Traversals

2.5 Binary Trees Traversals

Traversal is a process to visit all the nodes of a tree. There are three types of binary tree traversals.

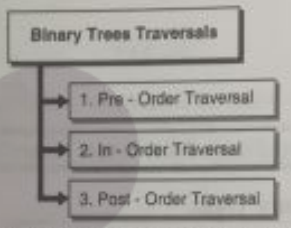


Fig. C2.3 : Types of binary tree traversals

Consider the following binary tree.

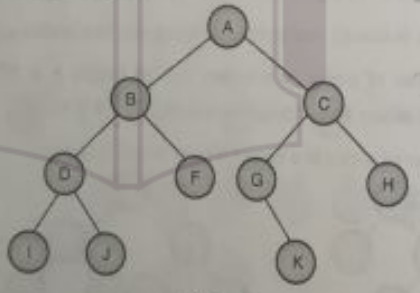


Fig. 2.5.1

Let's explore each tree traversal method one by one.

2.5.1 Pre - Order Traversal

- This traversal is also known as depth-first order.
- In this traversal technique the traversal order is root-left-right i.e.
  - Process data of root node

  - First, traverse left subtree completely
  - Then, traverse right subtree.
- In the above example if we perform the preorder traversal then first we start from its root which is A. Next we move to the left subtree of A, so we visit left child B. Again B has D and F as its left and right child reply, so first we visit left child D.
- Again D has I and J as its left and right child reply, so we visit its left child I which is the leftmost child and then we move to the right child J because I does not have any child. Next we visit the B's right child which is F. With this we have completed root and left parts of node A.
- Now we go for A's right child which is C so visit it. Again C has lest and right child, so move to the left child of C which is G. But G have only right child K, it does not have any left child. So we visit G's right child K. Next move to the right child of C because we have done with G. Visit C's right child H which is the right most child in the tree. So we stop the process.
- Pre-Order Traversal for above example :

  A - B - D - I - J - F - C - G - K – H

Code to implement pre-order traversal

(i) Recursive Preorder Traversal

```
#Pre-order recursive traversal. The nodes' values are appended to the result #list in traversal order
defpreorderRecursive(root, result):
if not root:
return
result.append(root.data)
preorderRecursive(root. left, result)
preorderRecursive(root.right, result)
```

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

(ii) Non-Recursive Preorder Traversal

```
#Pre-order iterative traversal. The nodes' values are appended to the result list in traversal order
def preorder_iterative(root, result):
if not root:
return
stack= []
stack.append(root)
```

```
while stack:
node =stack.pop()
result.append(node.data)
ifnode.right: stack.append(node.right)
ifnode.left: stack.append(node.left)
```

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

➜ 2.5.2 In - Order Traversal

- This traversal is also known as depth-first order.
- In this traversal technique the traversal order is left-root-right i.e.
  - First process left subtree (before processing root node)
  - Then, process current root node
  - Process right subtree.
- In the above example of binary tree, first we start from the left child of root node 'A' which is B but again B has left child D, so we move to D. Again D has left child I so we visit I because this is the leftmost child in a tree (it does not have any child).
- Then visit its root node D and then visit its right child J. With this we have completed the left part of node B. Then Visit B. Next move to B's right child which is F, visit F. With this we have done with A's left part.
- Now visit A and move to its right part. A has C as its right child. Again C have left child G and G have only right child K.G does not have any left child, so visit G. then visit K. with this we have done with left part of C.
- Then visit C and move to the right part of C. C has right child H which is the rightmost child so visit H and stop the process.
- In-Order Traversal for above example is :

  I - D - J - B - F - A - G - K - C – H

☛ Code to implement In-order traversal

(i) Recursive inorder Traversal

```
# In-order recursive traversal. The nodes' values are appended to the result #list in traversal order
def inorderRecursive(root, result):
if not root:
return
inorderRecursive(root. left, result)
result.append(root.data)
inorderRecursive(root.right, result)
```


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

(ii) Non-Recursive inorder Traversal

```
# In-order iterative traversal. The nodes' values are appended to the result list in traversal order
def inorderiterative(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()
result.append(node.data)
node = node.right
```

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

➜ 2.5.3 Post - Order Traversal

- In this traversal technique the traversal order is left-right-root.
  - Process data of left subtree
  - First, traverse right subtree
  - Then, traverse root node.
- In post-order traversal, the left subtree is traversed first in post-order traversal then the right subtree is traversed in post-order traversal and then root is traversed.
- In the above example, first we try to visit left child of root node 'A', but A's left child is a root node for left subtree. So we try to visit its (B's) left child 'D' and again D is a root for subtree with nodes D, I and J.
- So we try to visit its left child 'I' and it is the left most child. So first we visit 'I' then go for itsright child 'J'and later we visit root node 'D'. With this we have completed the left part of node B.
- Then visit B's right child 'F' and then visit 'B'. With this we have completed left part of node A. With this we have completed left parts of root node A. Then we go for right part of the node A. In right of A again there is a subtree with root C.

- So go for left child of C and again it is a subtree with root G. But G does not have left part but it has right child so first visit that right child 'K' and then visit node 'G'. With this we have completed the left part of node C.
- Then visit C's right child 'H' which is the right most child in the tree and then visit node 'C'. And finally visit the root node 'A'.so we stop the process.
- Post-Order Traversal for above example binary tree is :

  I - J - D - F - B - K - G - H - C – A

Code to implement post-order traversal

(i) Recursive postorder Traversal

```
# Post-order recursive traversal. The nodes' values are appended to the result #list in traversal order
def postorderRecursive(root, result):
if not root:
return
postorderRecursive(root.left, result)
postorderRecursive(root. right, result)
result.append(root.data)
```

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

(ii) Non-Recursive postorder Traversal

```
# 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).

Program 2.5.1

Write a program Python program for tree traversals.

Solution :

Python program for tree traversals

```
# A class that represents an individual node in a Binary Tree
class Node:
    def _init_(self,key):
        self.left = None
        self.right = None
        self.val = key
# A function to do inorder tree traversal
def printInorder(root):
    if root:
        # First recur on left child
        printInorder(root.left)
        # then print the data of node
        print(root.val),
        # now recur on right child
        printInorder(root.right)
# A function to do postorder tree traversal
def printPostorder(root):
    if root:
        # First recur on left child
        printPostorder(root.left)
        # the recur on right child
        printPostorder(root.right)
```

```
        # now print the data of node
        print(root.val),
# A function to do postorder tree traversal
def printPreorder(root):
    if root:
        # First print the data of node
        print(root.val),
        # Then recur on left child
        printPreorder(root.left)
        # Finally recur on right child
        printPreorder(root.right)
# Driver code
root = Node(1)
root.left  = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
print "Preorder traversal of binary tree is"
printPreorder(root)
print "\nInorder traversal of binary tree is"
printInorder(root)
print "\nPostorder traversal of binary tree is"
printPostorder(root)
```

Syllabus Topic : Generic Trees(N-ary Trees)

2.6 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 in Fig. 2.6.1

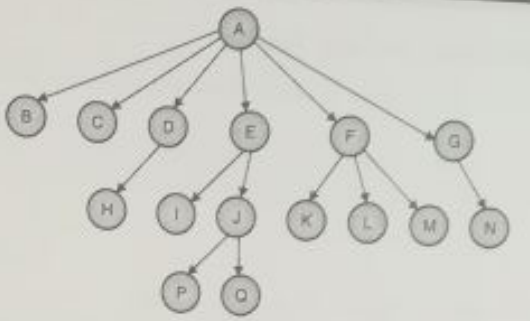


Fig. 2.6.1

Representation of 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 follows.

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

2.6.1 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.

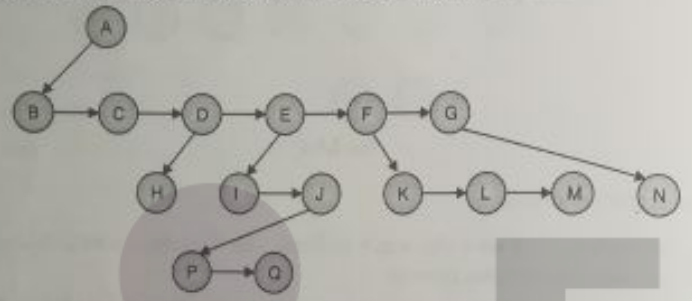


Fig. 2.6.2

- 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 shown in Fig. 2.6.3.

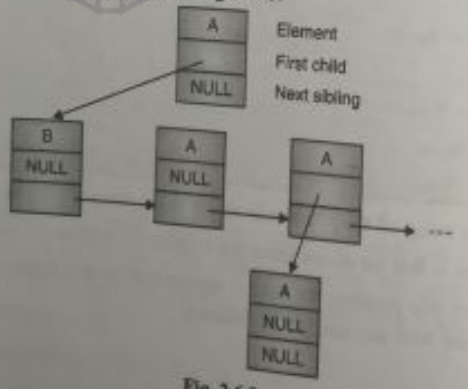


Fig. 2.6.3

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

```
#Node of a Generic Tree
classTreeNode:
#constructor
definit_ (self, data=None, next .. None):
self.data= data
sdf.firstChild = None
self.nextSibling .. None
```

Syllabus Topic : Threaded Binary Tree Traversals

2.7 Threaded Binary Tree Traversals

- The basic difference between a Binary tree and The Threaded Binary tree is that in Binary trees the nodes are null if there is no left or right child associated with it.
- That means in the linked list representation of binary tree, each node contains two pointers.
- The first pointer points to the left child and second pointer points to the right child but most of pointers are NULL because of the absence of left and right child and so there is no way to traverse back.
- So binary trees have a lot of wasted space. But in threaded binary tree we have threads associated with the nodes.
- That means they either are linked to the predecessor or successor in the inorder traversal of the nodes.
- This helps us to traverse further or backward in the inorder traversal fashion.
- "Threaded Binary Tree is also a binary tree in which all left child pointers that are NULL (in Linked list representation) points to its in-order predecessor, and all right child pointers that are NULL (in Linked list representation) points to its in-order successor."

Note : If there is no in-order predecessor or in-order successor, then it point to root node.

Why do we need Threaded Binary Tree ?

- Binary trees have a lot of wasted space: the leaf nodes each have 2 null pointers. We can use these pointers to help us in inorder traversals.
- Threaded binary tree makes the tree traversal faster since we do not need stack or recursion for traversal.
- Consider the following binary tree (Fig. 2.7.1).

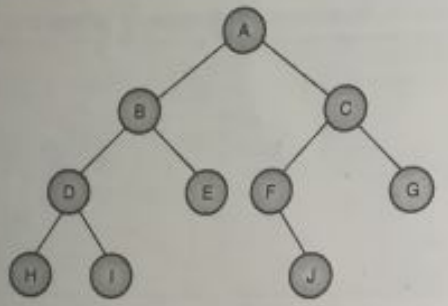


Fig. 2.7.1

- To convert above binary tree into threaded binary tree, first find the in-order traversal of that tree.
- In-order traversal of above binary tree.

  H - D - I - B - E - A - F - J - C - G
- Above example binary tree become as follows after converting into threaded binary tree.

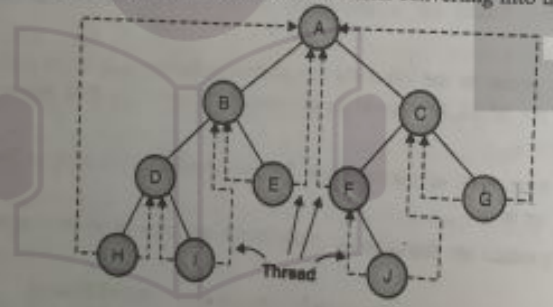


Fig. 2.7.2

In Fig. 2.7.2, threads are indicated with dotted links.

2.7.1 Types of Threaded Binary Trees

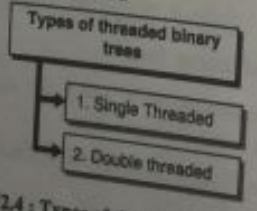


Fig. C2.4 : Types of threaded binary trees

➜ **1. Single Threaded**

Each node is threaded towards either the in-order predecessor or successor (left or right) means all right null pointers will point to inorder successor OR all left null pointers will point to inorder predecessor.

➜ **2. Double threaded**

Each node is threaded towards both the in-order predecessor and successor (left and right) means all right null pointers will point to inorder successor **AND** all left null pointers will point to inorder predecessor.

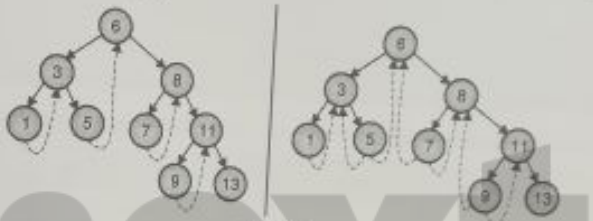


Fig. 2.7.3

2.7.2 Predecessor and Successor

When you do the inorder traversal of a binary tree, the neighbors of given node are called **Predecessor** (the node lies behind of given node) and **Successor** (the node lies ahead of given node).

☞ **Example**

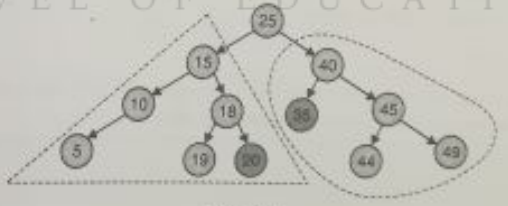


Fig. 2.7.4

2.7.3 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 :

Left	LTag	Data	RTag	Right

Fig. 2.7.5

```
#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
```

2.7.4 Difference between Binary Tree and Threaded Binary Tree Structures

	Regular Binary Tree	Threaded Binary Tree
If LTag = 0	NULL	Left points to the inorder predecessor
If LTag = 1	Left points to the left child	Left points to the left child
If RTag = 0	NULL	Right points to the inorder successor
If RTag = 1	Right points to the right child	Right points to the right child

2.7.5 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 light child of I. 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):
    else:
    return P.right
    Position = P.right
    while(Position.LTag == 1):
    Position = Position.left
    return Position
```


2.7.6 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)
printP.data
```

Syllabus Topic : Expression Trees

2.8 Expression Tree

- Expression tree is a binary tree in which each internal node corresponds to operator and each leaf node corresponds to operand.
- Consider the expression : ((3+ 5) × (5 + 9))
- It can be represented as a binary tree.

Fig. 2.8.1

- To convert Infix Expression into Postfix Expression using a stack data structure, We can use the following steps :
  1. Read all the symbols one by one from left to right in the given Infix Expression.
  2. If the reading symbol is operand, then directly print it to the result (Output).
  3. If the reading symbol is left parenthesis '(', then Push it on to the Stack.
  4. If the reading symbol is right parenthesis ')', then Pop all the contents of stack until respective left parenthesis is popped and print each popped symbol to the result.
  5. If the reading symbol is operator (+ , - , * , / etc.), then Push it on to the Stack. However, first pop the operators which are already on the stack that have higher or equal precedence than current operator and print them to the result.

Example

A + (B * C – (D / E – F) * G) * H

Stack	Input	Output
Empty	A+(B*C-(D/E-F)*G)*H	-
Empty	+(B*C-(D/E-F)*G)*H	A
+	(B*C-(D/E-F)*G)*H	A
+(	B*C-(D/E-F)*G)*H	A
+(	*C-(D/E-F)*G)*H	AB
+(*	C-(D/E-F)*G)*H	AB
+(*	-(D/E-F)*G)*H	ABC
+(-	(D/E-F)*G)*H	ABC*
+(-(	D/E-F)*G)*H	ABC*
+(-(	/E-F)*G)*H	ABC*D
+(-(/	E-F)*G)*H	ABC*D
+(-(/	-F)*G)*H	ABC*DE
+(-(-	F)*G)*H	ABC*DE/
+(-(-	F)*G)*H	ABC*DE/
+(-(-	)*G)*H	ABC*DE/F
+(-	*G)*H	ABC*DE/F-
+(-*	G)*H	ABC*DE/F-
+(-*	)*H	ABC*DE/F-G
+	*H	ABC*DE/F-G*-
+*	H	ABC*DE/F-G*-
+*	End	ABC*DE/F-G*-H
Empty	End	ABC*DE/F-G*-H*+

Program 2.8.1

Write a program for Building Expression Tree from Postfix Expression.

Solution :

```
# An expression tree node
class Et:
    # Constructor to create a node
```



```
    def __init__(self , value):
        self.value = value
        self.left = None
        self.right = None
# A utility function to check if 'c' is an operator
def isOperator(c):
    if (c == '+' or c == '-' or c == '*'
        or c == '/' or c == '^'):
        return True
    else:
        return False
# A utility function to do inorder traversal
def inorder(t):
    if t is not None:
        inorder(t.left)
        print t.value,
        inorder(t.right)
# Returns root of constructed tree for given postfix expression
def constructTree(postfix):
    stack = []
    # Traverse through every character of input expression
    for char in postfix :
        # if operand, simply push into stack
        if not isOperator(char):
            t = Et(char)
            stack.append(t)
        # Operator
        else:
            # Pop two top nodes
            t = Et(char)
            t1 = stack.pop()
            t2 = stack.pop()
            # make them children
            t.right = t1
```

```
            t.left = t2
            # Add this subexpression to stack
            stack.append(t)
    # Only element will be the root of expression tree
    t = stack.pop()
    return t
# Driver program to test above
postfix = "ab+ef*g*-"
r = constructTree(postfix)
print "Infix expression is"
inorder(r)
```

Syllabus Topic : Binary Search Trees (BSTs)

2.9 Binary Search Trees (BSTs)

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties :

- The left sub-tree of a node has a key less than or equal to its parent node's key.
- The right sub-tree of a node has a key greater than to its parent node's key.
- Both the left and right subtrees must also be binary search trees.
- "Binary Search Tree is a binary tree in which every node contains only smaller values in its left subtree and only larger values in its right subtree."
- Fig. 2.9.1 shows a pictorial representation of BST.

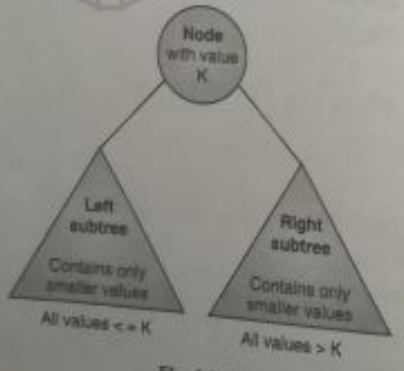


Fig. 2.9.1

Example

Fig. 2.9.2

We observe that the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.

Binary Search Tree Declaration

```
#Binary Search Tree Class and its methods
class BSTNode:
def init. (self, data):
self.data = data
self.left = None
self. right = None
#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):
returnself.left
#get right child of a node
def getRight(self):
return self.right
```

2.9.1 Operations on Binary Search Trees

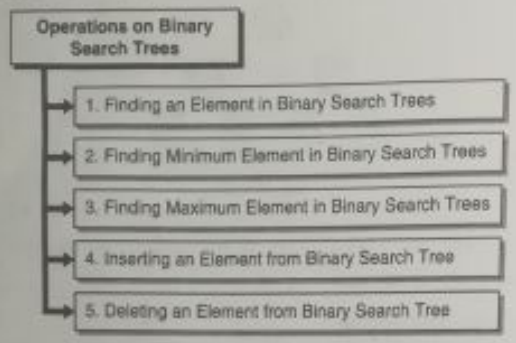


Fig. C2.5 : Operations on Binary Search tree

➜ 1. Finding an Element in Binary Search Trees

To search a given key in Binary Search Tree, we first compare it with root, if the key is present at root, we return root. If key is greater than root's key, we recur for right subtree of root node. Otherwise we recur for left subtree.

```
def find( root, data ):
currentNode = root
whilec urrentNode is not None and data != currentNode.getData():
if data>currentNode.getData():
currentNode = currentNode.getRight()
else:
currentNode = currentNode.getLeft()
return currentNode
```

➜ 2. Finding Minimum Element in Binary Search Trees

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

```
deffindMin(root):
currentNode = root
if currentNode.getLeft() == None:
return currentNode
else:
return findMin(currentNode.getLeft())
```


➜ 3. Finding Maximum Element in Binary Search Trees

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

```
#Search the key from node, iteratively
deffindMax(root):
currentNode = root
if currentNode.getRight() == None:
return currenLNode
else:
return findMax(currentNode.getRight())
```

➜ 4. Inserting an Element from Binary Search Tree

- When looking for a place to insert a new key, we traverse the tree from root to leaf, making comparisons to key stored in the nodes of the tree and deciding, based on the comparisons, to continue searching in the left or right subtree.
- In other words, we examine the root and recursively insert the new node to the left subtree if its key is less than that of root, or the right subtree if its key is greater than or equal to that of root.
- Code to insert a node in BST

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

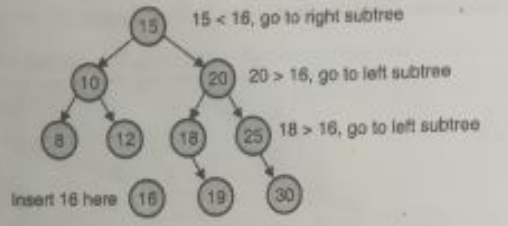


Fig. 2.9.3

5. Deleting an Element from Binary Search Tree

Deleting a node from Binary search tree has following three cases.

- **Case 1 :** Deleting a Leaf node (A node with no children)
- **Case 2 :** Deleting a node with one child
- **Case 3 :** Deleting a node with two children

1. Node to be deleted is leaf: Simply remove from the tree.

Fig. 2.9.4

2. Node to be deleted has only one child: Copy the child to the node and delete the child.

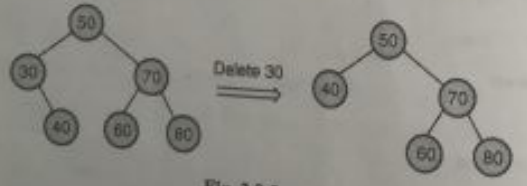


Fig. 2.9.5

3. Node to be deleted has two children: Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.

Fig. 2.9.6

The important thing to note is, inorder successor is needed only when right child is not empty. In this particular case, inorder successor can be obtained by finding the minimum value in right child of the node.

Code to delete a node in BST

```
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
ifroot.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)
ifpsucc.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
ifroot.left:
return root.left
# promote the left subtree
else:
```

```
return root.right
# promote the right subtree
else:
if root == data > data:
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
ifroot.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.data,root)
else:
return [parent, root)
```

Syllabus Topic : Balanced Binary Search Trees

2.10 Balanced Binary Search Trees

Binary search trees are a nice idea, but they fail to accomplish our goal of doing lookup insertion and deletion each in time $O(\log_2(n))$, when there are n items in the tree. Imagine starting with an empty tree and inserting 1, 2, 3 and 4, in that order.

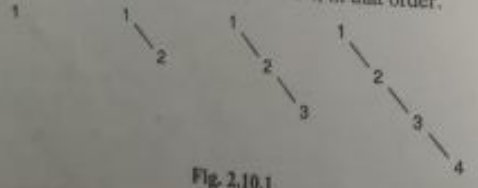


Fig. 2.10.1

- You do not get a branching tree, but a linear tree. All of the left subtrees are empty. Because of this behavior, in the worst case each of the operations (lookup, insertion and deletion) takes time $\Theta(n)$. From the perspective of the worst case, we might as well be using a linked list and linear search.
- That bad worst case behavior can be avoided by using an idea called height balancing, sometimes called AVL trees.

2.10.1 Height Balanced Trees

- The height of a node in a tree is the length of the longest path from that node downward to a leaf, counting both the start and end vertices of the path.
- The height of a leaf is 1. The height of a nonempty tree is the height of its root.
- For example, tree has height 3. (There are 3 equally long paths from the root to a leaf. One of them is (30 18 24).) The height of an empty tree is defined to be 0.

Fig. 2.10.2

- We define the balance factor for a node as the difference between the height of the left subtree and the height of the right subtree.
  balanceFactor=height(leftSubTree)−height(rightSubTree)
- We will define a tree to be balance if the balance factor is -1, 0, or 1.

Example

Fig. 2.10.3 : Tree

- In the above tree, balanced factor of each node is as follows :

  balanceFactor(24) = height(leftSubTree) − height(rightSubTree)
  = 0 − 0 = 0

balanceFactor(36) = 0 – 0 = 0

balanceFactor(51) = 0 – 0 = 0

balanceFactor(18) = 0 – -1 = -1

balanceFactor(50) = 1 – 1 = 0

balanceFactor(30) = 2 – 2 = 0

- So the above tree is balanced binary search tree because the balanced factor of each node is – 1. 0, or 1.

Syllabus Topic : AVL (Adelson and Velski and Landis) Trees

2.11 AVL (Adelson, Velski and Landis) Trees

- AVL tree is also a binary search tree but it is a balanced tree. The technique of balancing the height of binary trees was developed by Adelson, Velskii and Landis and hence given the short form as AVL tree or Balanced Binary Tree.
- A binary tree is said to be balanced, if the difference between the heights of left and right subtrees of every node in the tree is either -1, 0 or +1.
- An AVL tree is defined as follows :

  "An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1."

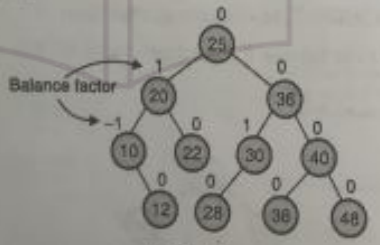


Fig. 2.11.1

- The above tree is a binary search tree and every node is satisfying balance factor condition. So this tree is said to be an AVL tree.

AVL Tree 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 == None:
return 0
else:
leftH = self.recHeight(r.left)
rightH = self. recHeight(r.right)
if leftH>rightH:
return 1 +leftH
else:
return 1 +rightH
```

2.11.1 AVL Tree Rotations

- In AVL tree, after performing every operation like insertion and deletion we need to check the balance factor of every node in the tree.
- If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced.
- Rotation operations are used to make a tree balanced.
- There are **four** rotations and they are classified into **two** types :
  1. Single rotations
     1. Left roatation (LL rotation)
     2. Right roatation (RR rotation)
  2. Double rotations
     1. Left Right rotation (LR rotation)
     2. Right Left rotation (RL rotation)

2.11.1(A) Single Rotations

(a) Single Left Rotation (LL Rotation)

- If a tree becomes unbalanced, when a node is inserted into the right subtree of the right child, then we perform a single left rotation .

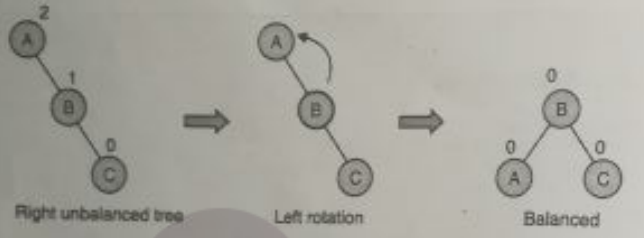


Fig. 2.11.2

- In our example, node **A** has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making **A** the left-subtree of B.

(b) Single Right rotation(RR Rotation)

- AVL tree may become unbalanced, if a node is inserted in the left subtree of the left child. The tree then needs a right rotation.

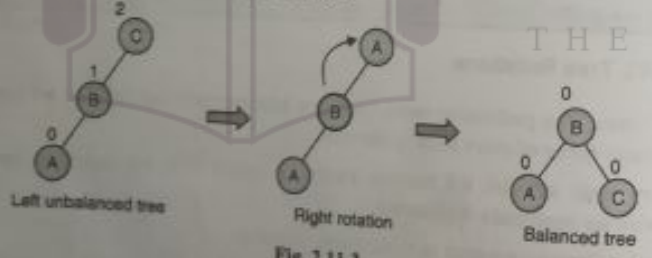


Fig. 2.11.3

- The unbalanced node becomes the right child of its left child by performing a right rotation.

2.11.1(B) Double Rotations

(a) Left-Right rotation(LR rotation)

- The LR Rotation is combination of single left rotation followed by single right rotation.

- In LR Rotation, first every node moves one position to left then one position to right from the current position. To understand LR Rotation, let us consider following insertion operations into an AVL Tree,

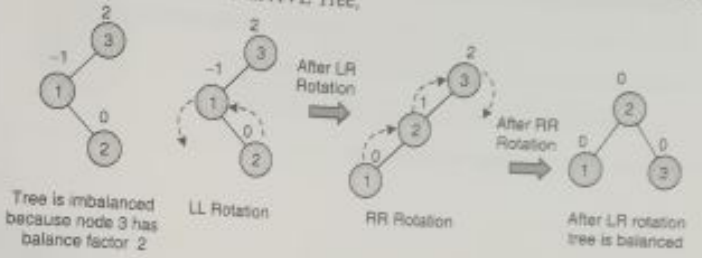


Fig. 2.11.4

(b) Right Left Rotation (RL Rotation)

- The RL Rotation is combination of single right rotation followed by single left rotation.
- In RL Rotation, first every node moves one position to right then one position to left from the current position.
- To understand RL Rotation, let us consider following insertion operations into an AVL Tree :

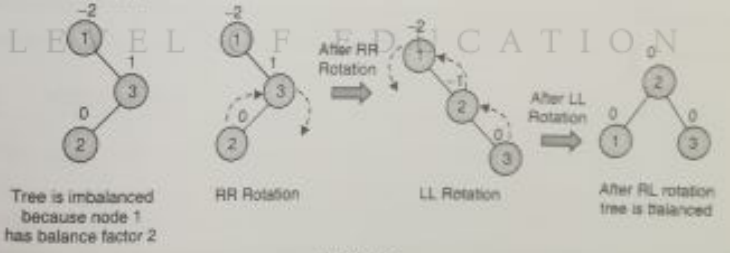


Fig. 2.11.5

Review Questions

Q. 1 What is tree ? **(Refer section 2.1)**

Q. 2 How represent binary tree? **(Refer section 2.2)**

Q. 3 Explain different types of binary tree. **(Refer section 2.3)**

Q. 4 Explain pre-order, in-order and post-order traversals with example. **(Refer sections 2.5.1, 2.5.2 and 2.5.3)**

Q. 5 Write short note on threaded binary tree traversals. **(Refer section 2.7)**

Q. 6 Define predecessor and successor. **(Refer section 2.7.2)**

Q. 7 Write short note on binary reach trees. **(Refer section 2.9)**

Q. 8 Describe AVL tree. **(Refer section 2.11)**

❑❑❑

CHAPTER 3

UNIT II

Graph and Selection Algorithms

Syllabus

Graph Algorithms : Introduction, Glossary, Applications of Graphs, Graph Representation, Graph Traversals, Topological Sort, Shortest Path Algorithms, Minimal Spanning Tree.

Selection Algorithms : What are Selection Algorithms? Selection by Sorting, Partition-based Selection Algorithm, Linear Selection Algorithm - Median of Medians Algorithm, Finding the K Smallest Elements in Sorted Order.

Syllabus Topic : Graph Algorithm : Introduction

3.1 Introduction

- This chapter introduces an important non-linear data structure called Graph.
- It has applications in various fields like electrical and electronics engineering, computer science, games and puzzles, Geographical Information System etc.

Syllabus Topic : Glossary

3.1.1 Glossary of Graph

- Graph is a non linear data structure. A Graph G consists of finite set of vertices V and finite set of edges E which can be denoted by G = (V, E).
- Where, V represent the entities which has names and other attributes.
- Edges E represent the links that connect the vertices.

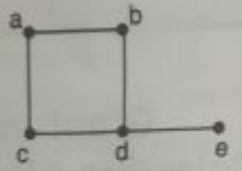


Fig. 3.1.1

In the above graph,

V = {a, b, c, d, e}

E = {ab, ac, bd, cd, de}

3.1.2 Graph Terminology

Vertex

A individual data element of a graph is called as Vertex. **Vertex** is also known as **node**. In above example graph, a, b, c, d and e are known as vertices.

Edge

An edge is a connecting link between two vertices. **Edge** is also known as **Arc**. An edge is represented as (startingVertex, endingVertex). In the above graph (ab), (ac), (cd), (bd), (de) are edges.

Types of Edges

1. **Undirected Edge :** An undirected edge is a bidirectional edge. If there is a undirected edge between vertices a and b then edge (ab) is equal to edge (ba).
2. **Directed Edge :** A directed edge is a unidirectional edge. If there is a directed edge between vertices a and b then edge (ab) is not equal to edge (ba).
3. **Weighted Edge :** A weighted edge is an edge with cost on it.

Undirected Graph

An undirected graph is a graph in which all the edges are bi-directional i.e. there is no direction associated with the edges.

Directed Graph

A directed graph is a graph in which all the edges are uni-directional i.e. direction is associated with the edges.

Fig. 3.1.2 : Undirected graph

Fig. 3.1.3 : Directed graph

Weighted Graph

The graph in which weight is associate with every edge is a weighted graph.

Fig. 3.1.4

Adjacent

If there is an edge between vertices A and B then both A and B are said to be adjacent.

Origin

If an edge is directed, its first endpoint is said to be origin of it.

Destination

If an edge is directed, its first endpoint is said to be origin of it and the other endpoint is said to be the destination of the edge.

Outgoing Edge

A directed edge is said to be outgoing edge on its origin vertex.

Incoming Edge

A directed edge is said to be incoming edge on its destination vertex.

Fig. 3.1.5

In the above graph,

Vertex	Number of outgoing edges	Number of incoming edges
1	2	0
2	0	2
3	2	2
4	1	1

Degree of a vertex

Total number of edges connected to a vertex is said to be degree of that vertex.

Indegree of a vertex : Total number of edges terminating at the vertex is said to be indegree of that vertex.

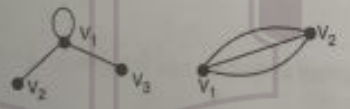
Outdegree of a vertex : Total number of edges starting from the vertex is said to be outdegree of that vertex.

Self Loop

If the starting and terminating vertices of an edge are same, then that edge is termed as self loop.

Parallel edges or multiple edges

If there are multiple edges between a pair of vertices, then such edges are called as parallel or multiple edges.



(a) Self loop (b) Parallel edges

Fig. 3.1.6

Simple Graph

A graph is said to be simple if there are no parallel and self-loop edges.

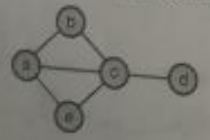


Fig. 3.1.7

Complete Graph

A graph in which every vertex is connected with every other vertex i.e. each vertex is adjacent to all other vertices in the graph, then that graph is called a complete graph.

Fig. 3.1.8

Multigraph

The graph which contains multiple edges between a pair of vertices is called a multigraph.

Fig. 3.1.9

Path

A path is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex.

Cycle

A cycle is a path where the first and last vertices are the same. A simple cycle is a cycle with no repeated vertices or edges (except the first and last vertices).

Directed Acyclic Graph(DAG)

A directed acyclic graph [DAG] is a directed graph with no cycles.

Syllabus Topic : Applications of Graph

3.1.3 Applications of Graph

1. Representing relationships between components in electronic circuits.
2. Transportation networks: Highway network, Flight network.

3. Computer networks: Local area network, Internet, Web
4. Databases: For representing ER (Entity Relationship) diagrams in databases, for representing dependency of tables in databases

Syllabus Topic : Graph Representation

3.2 Graph Representation

Graph data structure is represented using following representations :

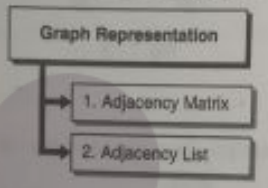
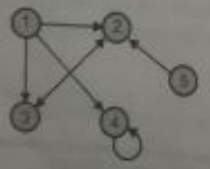


Fig. C3.1: Graph representation

➜ 3.2.1 Adjacency Matrix

- Adjacency Matrix is a 2D array of size V × V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j.
- Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.
- For example, consider the following graph representation.



	1	2	3	4	5
1	0	1	1	1	0
2	0	0	1	0	0
3	0	1	0	0	0
4	0	0	0	1	0
5	0	1	0	0	0

Fig. 3.2.1

Graph Declaration for Adjacency Matrix

To represent graphs, we need the number of vertices, the number of edges and also their interconnections. So, the graph can be declared as:


```
class Vertex:
def init_(self, node):
self.id = node
# Mark all nodes unvisited
self.visited =False
def addNeighbor(self, neighbor, G):
G.addEdge(self.id, neighbor)
def getConnections(self, G):
return G.adjMatrix[self. id]
def getVertexID(self):
return self.id
def setVertexJD(self, id):
self.id= id
def setVisited(self):
self.visited =True
def _str_(self):
return str(self. id)
class Graph:
def init (self, numVertices, cost= 0):
self.adjMatrix = [[-1-]*numVerticesfor _ in range(numVertices)]
self.numVertices= numVertices
self.vertices = []
for i in range(O,numVertices):
newVertex = Vertex(i)
self.vertices.append(newVertex)
```

➜ 3.2.2 Adjacency List

- An array of linked lists is used. Size of the array is equal to number of vertices. Let the array be array[].
- An entry array[i] represents the linked list of vertices adjacent to the i^{th} vertex. This representation can also be used to represent a weighted graph. The weights of edges can be stored in nodes of linked lists.

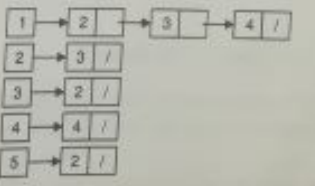


Fig. 3.2.2

Graph Declaration for Adjacency List

```
class Vertex:
def __init__(self, node):
self.id = node
self.adjacent = {}
# Set distance to infinity for all nodes
self.distance = sys.maxint
Mark all nodes unvisited
self.visited = False
# Predecessor
self.previous = None
    class Graph:
def init_(self):
self.vertDictionary = {}
self.numVertices = 0
```

Syllabus Topic : Graph Traversal

3.3 Graph Traversal

- Graph traversal is technique used for searching a vertex in a graph. Graph traversal means visiting every vertex and edge exactly once in a well-defined order.
- While using certain graph algorithms, you must ensure that each vertex of the graph is visited exactly once.There are two graph traversal techniques and they are as follows,

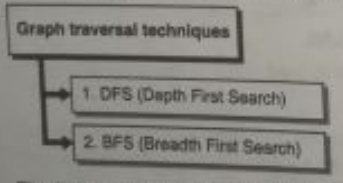


Fig. C3.2 : Graph traversal techniques

3.3.1 DFS (Depth First Search)

Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

- **Undiscovered state :** The initial state of vertex.

- **Discovered state :** The vertex is encountered but not yet processed.
- **Processed state :** The vertex has been visited.

Steps for DFS

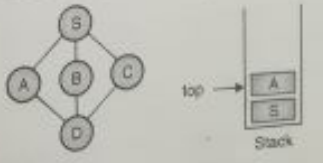
Step 1 : Initially all the vertices of graph G are set to Undiscovered.

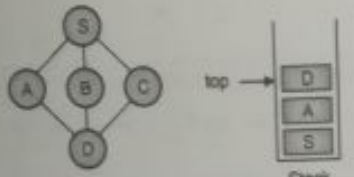
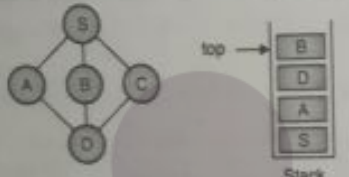
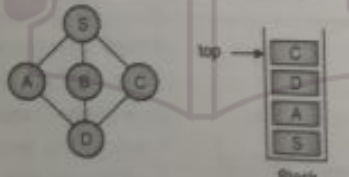
Step 2 : Change the state of starting vertex of the graph to Discovered, and put it in the stack.

Step 3 : Repeat steps 4 to 5 while the stack is not empty.

Step 4 : Remove a vertex say v which is at the top of the stack and change its state to processed.

Step 5 : Repeat for all undiscovered vertices u of vertex v u is set to the status Discovered and pushed to the stack.

Step	Traversal	Description
1.		Initialize the stack.
2.		Mark S as visited and put it onto the stack. Explore any unvisited adjacent node from S. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.
3.		Mark A as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both S and D are adjacent to A but we are concerned for unvisited nodes only.

Step	Traversal	Description
4.		Visit D and mark it as visited and put onto the stack. Here, we have B and C nodes, which are adjacent to D and both are unvisited. However, we shall again choose in an alphabetical order.
5.		We choose B, mark it as visited and put onto the stack. Here B does not have any unvisited adjacent node. So, we pop B from the stack.
6.		We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find D to be on the top of the stack.
7.		Only unvisited adjacent node is from D is C now. So we visit C, mark it as visited and put it onto the stack.

DFS algorithm

- Initially all vertices are marked unvisited (false). The DFS algorithm starts at a vertex u in the graph.
- By starting at vertex u it considers the edges from u to other vertices. If the edge leads to an already visited vertex, then backtrack to current vertex u.
- If an edge leads to an unvisited vertex, then go to that vertex and start processing from that vertex. That means the new vertex becomes the current vertex.
- Follow this process until we reach the dead-end. At this point start backtracking. The process terminates when backtracking leads back to the start vertex. The algorithm based on this mechanism is given below: assume visited[] is a global array.


```
def dfs(G, currentVert, visited):
visited[currentVert] =True                  # Mark the visited node
print "traversal: " + currentVert.getVertexID()
for nbr in currentVert.getConnections():    # Take a neighboring node
if nbr not in visited:                      #Check whether the neighbor node is already
visited
dfs(G, nbr, visited)                        # Recursively traverse the neighboring node

def DFSTraversal(G):
visited = {}                                # Dictionary to mark the visited nodes
for currentVert in G:                       # G contains vertex objects
if currentVert not in visited:              # Start traversing from the root node only if
                                            #its not visited
dfs(G, currentVert, visited)                #For a connected graph this is called only once
```

DFS program

```
# Python program to print DFS traversal from a given graph
from collections import defaultdict

# This class represents a directed graph using
# adjacency list representation
class Graph:

    # Constructor
    def __init__(self):

        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    # A function used by DFS
    def DFSUtil(self,v,visited):

        # Mark the current node as visited and print it
        visited[v]= True
        print v,
```

```
    # Recur for all the vertices adjacent to this vertex
        for i in self.graph[v]:
            if visited[i] == False:
                self.DFSUtil(i, visited)

    # The function to do DFS traversal. It uses
    # recursive DFSUtil()
    def DFS(self,v):

    # Mark all the vertices as not visited
        visited = [False]*(len(self.graph))

    # Call the recursive helper function to print
    # DFS traversal
        self.DFSUtil(v,visited)

# Driver code
# Create a graph given in the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print "Following is DFS from (starting from vertex 2)"
g.DFS(2)
```

➜ 3.3.2 BFS (Breadth First Search)

- BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layer wise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.
- As the name BFS suggests, you are required to traverse the graph breadthwise as follows :
  1. First move horizontally and visit all the nodes of the current layer
  2. Move to the next layer

- **Undiscovered state :** the initial state of vertex.
- **Discovered state :** the vertex is encountered but not yet processed.
- **Processed state :** the vertex has been visited.

☞ Steps for BFS

Step 1 : Initially all the vertices of graph G are set to Undiscovered.

Step 2 : Change the state of starting vertex of the graph to Discovered, and put it in the queue.

Step 3 : Repeat steps 4 to 5 while the queue is not empty.

Step 4 : Remove a vertex say v which is at the front of the queue and change its state to processed.

Step 5 : Repeat for all undiscovered vertices u of vertex v, u is set to the status Discovered and added to the queue.

☞ Example

Consider the following example.

Fig. 3.3.1

Step 1 : Select the vertex 1 as starting point (visit 1). Insert 1 into the queue.

Queue : | 1 | | | | |

Step 2 : Visit all adjacent vertices of 1 which are not visited(2, 3). Insert newly visited vertices in the queue and delete 1 from the queue.

Queue : | | 2 | 3 | | |

Step 3 : Visit all adjacent vertices of 2 which are not visited (4, 5). Insert newly visited vertices in the queue and delete 2 from the queue.

Queue : | | | 3 | 4 | 5 |

Step 4 : Visit all adjacent vertices of 3 which are not visited (there is no vertex). Delete 3 from the queue.

Queue : | | | 3 | 4 | 5 |

Step 5 : Visit all adjacent vertices of 4 which are not visited (there is no vertex). Delete 4

from the queue.

Queue : | | | | 4 | 5 |

Step 6 : Visit all adjacent vertices of 5 which are not visited (there is no vertex). Delete 5 from the queue.

Queue : | | | | | 5 |

Queue became empty. So stop the BFS process.

BFS Traversal is: 1 2 3 4 5.

BFS algorithm

- Assume that initially all vertices are marked unvisited (false). Vertices that have been processed and removed from the queue are marked visited (true).
- We use a queue to represent the visited set as it will keep the vertices in the order or when they were first visited. The implementation for the above discussion can be give n as:

```
def BFSTraversal(G,s):
start = G.getVertex(s)
start.setDistance(0)
start.setPrevious(None)
vertQueue = Queue()
vertQueue.enQueue(start)
while (vertQueue.size> 0):
currentVert = vertQueue.deQueue()
print current Vert.getVertexID()
for nbr in currentVert.getConnections():
if (nbr.getColor() == 'white'):
nbr.setColor('gray')
nbr.set_Distance(currentVert.getDistance() + 1)
nbr.setPrevious(currentVert)
vertQueue.enQueue(nbr)
currentVert.setColor('black')
def BFS(G):
for v in G:
if (v.getColor() == 'white'):
BFSTraversal(G, v .get VertexID())
```

BFS program

```
# Program to print BFS traversal from a given source
# vertex. BFS(int s) traverses vertices reachable
# from s.
```



```
from collections import defaultdict

# This class represents a directed graph using adjacency
# list representation
class Graph:

    # Constructor
    def __init__(self):

        # default dictionary to store graph
        self.graph = defaultdict(list)

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    # Function to print a BFS of graph
    def BFS(self, s):

        # Mark all the vertices as not visited
        visited = [False]*(len(self.graph))

        # Create a queue for BFS
        queue = []

        # Mark the source node as visited and enqueue it
        queue.append(s)
        visited[s] = True
        while queue:
            # Dequeue a vertex from queue and print it
            s = queue.pop(0)
            print s,
            # Get all adjacent vertices of the dequeued
            # vertex s. If a adjacent has not been visited,
            # then mark it visited and enqueue it
            for i in self.graph[s]:
                if visited[i] == False:
                    queue.append(i)
```

```
                visited[i] = True

# Driver code
# Create a graph given in the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)
print "Following is Breadth First Traversal (starting from vertex 0)"
g.BFS(0)
```

Syllabus Topic : Topological Sort

3.4 Topological Sort

- Topological sort is an algorithm for a Directed Acyclic Graph (DAG).
- Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge uv, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

Working of Topological Sort

- Initially, indegree is computed for all vertices, starting with the vertices which are having indegree 0. To keep track of vertices with indegree zero we can use a queue.
- All vertices of indegree 0 are placed on queue. While the queue is not empty, a vertex v is removed, and all edges adjacent to v have their indegrees decremented. A vertex is put on the queue as soon as its indegree falls to 0.
- The topological ordering is the order in which the vertices DeQueue.
- The time complexity of this algorithm is O(|E| + |V|) if adjacency lists are used.

Consider following example:

Fig. 3.4.1

1. Compute the indegrees

   V1: 0

   V2: 1

   V3: 2

   V4: 2

   V5: 2

2. Find a vertex with indegree 0: V1, Insert it in the queue.

V1				

3. Output V1 , remove V1 from the queue and update the indegrees.Remove edges: (V1,V2), (V1, V3) and(V1,V4)Updated indegrees:

   V2: 0

   V3: 1

   V4: 1

   V5: 2

4. Now vertex V2 is having indegree 0. Insert V2 in the queue.

V2				

5. Output V2 , remove V2 from the queue and update the indegrees:

   Remove edges: (V2,V4) , (V2, V5)

   Updated indegrees:

   V3: 1

   V4: 0

   V5: 1

6. Now vertex V4 is having indegree 0. Insert V4 in the queue.

V4				

7. Output V4 , remove V4 from the queue and update the indegrees:

   Remove edges: (V4,V5) , (V4, V3)

   Updated indegrees:

   V3: 0

   V5: 0

8. Now vertex V5 and V3 both are having indegree 0. There is no any other remaining vertex or edge. So stop the process.

   Finally, the topological sort is as : V1, V2, V4, V3, V5.

3.4.1 Topological Sort Algorithm

- Initially, indegree is computed for all vertices, starting with the vertices which are having indegree 0. To keep track of vertices with indegree zero we can use a queue. All vertices of indegree 0 are placed on queue.
- While the queue is not empty, a vertex v is removed, and all edges adjacent to v have their indegrees decremented. A vertex is put on the queue as soon as its indegree falls to 0. The topological ordering is the order in which the vertices DeQueue.
- The time complexity of this algorithm is O(E + V) if adjacency lists are used.

```
class Vertex:
def __init__(self, node):
self.id = node
self.adjacent= {}
# Set distance to infinity for all nodes
self.distance= sys.maxint
# Mark all nodes unvisited
self.visited = False
# Predecessor
self.previous = None
#InDegree Count
self.inDegree = 0
#OutDegree Count
self.outDegree =0
#.....

class Graph:
    def __init__(self):
self.vertDictionary = {}
self.numVertices = 0
#.....

def topologicalSort(G):
"""Perform a topological sort of the nodes. If the graph has a cycle,
throw a Graph Topological Exception with the list or successfully
ordered nodes."""
# Topologically sorted list of the nodes (result)
topologicalList = []
#Queue (fifo list) of the nodes with inDegree 0
```



```
remainingInDegree = {}
nodes= G.getVertices()
for v in G:
indegree = v.getInDegree()
if indegree == 0:
topologicalQueue.append(v)
else:
remainingInDegree[v] = indegree
#Remove nodes with inDegree 0 and decrease the inDegree of their sons
while len(topologicalQueue):
# Remove the first node with degree 0
node= topologicalQueue.pop(0)
topologicalList.append(node)
# Decrease the in Degree of the sons
for son in node.getConnections():
son.setInDegree(son.getInDegree() -1)
if son.getInDegree() == 0:
topologicalQueue.append(son)
#If not all nodes were covered, the graph must have a cycle
# Raise a GraphTopographicalException
if len(topologicalList) != len(nodes):
raise GraphTopologicalException(topologicalList)


#Printing the topological order
while len(topologicalList):
node = topologicalList.pop(0)
print node.getVertexID()
```

Syllabus Topic : Shortest Path Algorithms

3.5 Shortest Path Algorithms

- Finding the shortest path is one of the basic problems encountered in many graph applications. Given a graph G = (V, E) and a distinguished vertex s, we need to find the shortest path from s to every other vertex in G.

- Generally, there can exist more than one path between a particular pair of vertices. There are different variations of the shortest path problem which vary in terms of specifications of the source vertex and the destination vertex of a path.
  1. Shortest path can be found from one particular source vertex to all other vertices. This is known as **Single source shortest path problem(Dijkstra's Algorithm).**
  2. Shortest path can be found from all possible source vertices to all other destination vertices. This is known as **All pairs shortest path problem(Floyd Warshall's Algorithm).**

3.5.1 Single Source Shortest Path Problem (Dijkstra's Algorithm)

- Dijkstra's algorithm finds the shortest path from source vertex v to all other vertices in the graph.
- It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined.
- It works in directed or undirected graph and all edges in the graph must have non-negative weight.
- Graph should not contain any cycle.
- Dijkstra's algorithm maintains a distance parameter for each vertex of the graph. These distances are initially set to infinity for all vertices except the source vertex s. This is done not to imply there is an infinite distance, but to note that those vertices have not yet been visited.
- The algorithm involves repeated iteration of the following process. At each iteration, we have a tree T rooted at s. For the first iteration, the tree will be the single vertex s, and the distance to it will be zero.
- For subsequent iterations (after the first), the next vertex v to be added to the tree will be the closest unvisited vertex to s (this will be easy to find). Once we found v, we add v to the tree, and check for each of its neighbors u, if the path from s – v path along with the edge uv gives a shorter distance for u than the one we have stored for u. If yes, the distance of u is updated to the new smaller value.
- We also maintain for each vertex v a parent vertex, which determines the last edge that is used to get to v from s.

Dijkstra's algorithm

```
import heap q
def dijkstrastra(G, source):
print "Dijkstra's shortest path"
# Set the distance for the source node to zero
source.setDistance(0)
```



```
# Put tuple pair into the priority queue
unvisitedQueue = [(v.getDistance(),v) for v in G]
heapq.heapify(unvisitedQueue)
while len(unvisitedQueue):
   #Pops a vertex with the smallest distance
uv = heapq.heappop(unvisitedQueue)
current = uv[1]
current.setVisited()
#for next in v.adjacent:
for next in current.adjacent:
# if visited, skip
if next.visited:
continue
newDist = current.getDistance() + current.getWeight(next)
if newDist < next.getDistance():
next.setDistance(newDist)
next.setPrevious(current)
print 'Updated : current = %s next= %s newDist = %s'
%(current.getVertexID(), next.getVertexID(), next.getDistance())
else:
print 'Not updated : current = %s next = %s newDist = %s'
%(current.getVertexID(), next.getVertexID(), next.getDistance())
#Rebuild heap
# 1. Pop every item
while len(unvisitedQueue):
heapq.heappop(unvisitedQueue)
# 2. Put all vertices not visited into the queue
unvisitedQueue = [(v.getDistance(),v) for v in G if not v.visited]
heapq. heapify(unvisitedQueue)
```

Example

- Consider the following example:

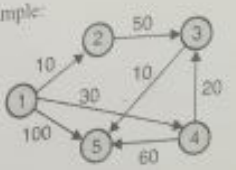


Fig. 3.5.1

Initially

$$S = \{1\}, \tilde{D}[2] = 10, \tilde{D}[3] = \infty, \tilde{D}[4] = 30, \tilde{D}[5] = 100$$

Iteration 1

Select w = 2, so that S = {1, 2}

D[3] = min(∞, D[2] + C[2, 3]) = 60

D[4] = min(30, D[2] + C[2, 4]) = 30

D[5] = min(100, D[2] + C[2, 5]) = 100

Iteration 2

Select w = 4, so that S = {1, 2, 4}

D[3] = min(60, D[4] + C[4, 3]) = 50

D[5] = min(100, D[4] + C[4, 5]) = 90

Iteration 3

Select w = 3, so that S = {1, 2, 4, 3}

D[5] = min(90, D[3] + C[3, 5]) = 60

Iteration 4

Select w = 5, so that S = {1, 2, 4, 3, 5}

D[2] = 10

D[3] = 50

D[4] = 30

D[5] = 60

3.5.2 All Pairs Shortest Path Problem (Bellman-Fordalgorithm)

- Dijkstra algorithm doesn't work for Graphs with negative weight edges, Bellman-Ford works for such graphs.
- Bellman-Ford algorithm is used to find all shortest path in a graph from one source to all other nodes.
- This algorithm works if there are no negative-cost cycles. Each vertex can DeQueue at most |V| times, so the running time is O(|E|.|V|) if adjacency lists are used.
- Bellman-Ford algorithm has more running time than Dijkstra's algorithm.
- This algorithm takes two nodes as arguments and an edge connecting these nodes.

- If the distance from the source to the first node (A) plus the edge length is less than distance to the second node, than the first node is denoted as the predecessor of the second node and the distance to the second node is recalculated (distance (A) + edge.length). Otherwise no changes are applied.

Bellman-Ford Algorithm

```
import sys
def BellmanFord(G, source):
destination = {}
predecessor = {}
for node in G:
destination[node] = sys.maxint # We start admiting that the rest of nodes are very very far
predecessor[node] =None
destination[source] = 0 # For the source we know how to reach
for i in range(len(G)- 1):
  for u in G:
    for v in G[u]: #For each neighbour of u
    # If the distance between the node and the neighbour is lower than the one I have now
    if destination[v] > destination[u] + G[u][v]
    # Record this lower distance
    Destination[v] = destination[u] + G[u][v]
    Predecessor[v] = u
# Step 3: check for negative-weight cycles
for u in G:
for v in G[u]:
assert destination[v] <= destination[u] + G[u][v]
return destination, predecessor
if __name__ == '__main__':
G = {
'A': {'B': - 1, 'C': 4},
'B': {'C': 3, 'D': 2, 'E       ': 2},
'C': {},
'D': {'B': 1, 'C': 5},
'E': {'D': -3}
}
print BellmanFord(G, 'A')
```

Example

Consider following example.

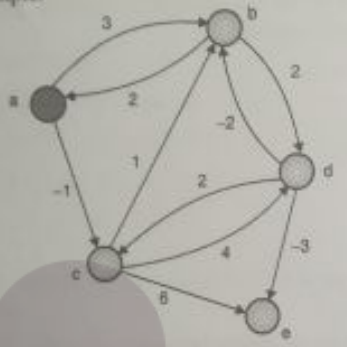


Fig. 3.5.2

Step 1 : Take the 0th iteration:

Let the given source vertex be a. Initialize all distances as infinite, except the distance to source itself.

x	u	Distance d[u] ← min {d[u], d[x] + c(x, u)}	a	b	c	d	e
			0	∞	∞	∞	0
a,	b	d[b] ← min {∞, 0 + 3} = 3	0	3	∞	∞	∞
a,	c	d[c] ← min {∞, 0 – 1} = –1	0	3	–1	∞	∞
b,	a	d[a] ← min {0, 3 + 2} = 0	0	3	–1	∞	∞
b,	d	d[d] ← min {∞, 3 + 2} = 5	0	3	–1	5	∞
c,	b	d[b] ← min {3, –1 + 1} = 0	0	0	–1	5	∞
c,	d	d[d] ← min {5, –1 + 4} = 3	0	0	–1	3	∞
c,	e	d[e] ← min {∞, –1 + 6} = 5	0	0	–1	3	5
d,	b	d[b] ← min {0, –3 + 2} = 0	0	0	–1	3	5
d,	c	d[c] ← min {–1, 3 + 2} = –1	0	0	–1	3	5
d	e	d[e] ← min {5, 3 – 3} = 0	0	0	–1	3	0

Iteration	Dist(x)				
	a	b	c	d	e
0	0	∞	∞	∞	∞

Step 2 : Take the 1st iteration:

Take one vertex at a time say A and edges which are outgoing from the vertex A.

Iteration	Dist(x)				
	a	b	c	d	e
0	0	∞	∞	∞	∞
1	0	3	–1	∞	∞

x	u	Distance d[u] ← min {d[u], d[x] + c(x, u)}	a	b	c	d	e
			0	0	–1	3	0
a,	b	d[b] ← min {0, 0 + 3} = 0	0	0	∞	3	0
a,	c	d[c] ← min {–1, 0 – 1} = –1	0	0	–1	3	0
b,	a	d[a] ← min {0, 0 + 2} = 0	0	0	–1	3	0
b,	d	d[d] ← min {3, 0 + 2} = 2	0	0	–1	2	0
c,	b	d[b] ← min {0, –1 + 1} = 0	0	0	–1	2	0
c,	d	d[d] ← min {2, –1 + 4} = 2	0	0	–1	2	0
c,	e	d[e] ← min {0, –1 + 6} = 0	0	0	–1	2	0
d,	b	d[b] ← min {0, –2 + 2} = 0	0	0	–1	2	0
d,	c	d[c] ← min {–1, 2 + 2} = –1	0	0	–1	2	0
d	e	d[e] ← min {0, 2 – 3} = –1	0	0	–1	2	–1

Step 3 : Take the next iteration and perform the same way:

Iteration	Dist(x)				
	a	b	c	d	e
0	0	∞	∞	∞	∞
1	0	3	–1	∞	∞
2	0	0	–1	2	0
3	0	0	–1	2	–1

Syllabus Topic : Minimal Spanning Tree

3.6 Minimal Spanning Tree

In a connected undirected graph G = (V, E), if we cover all the vertices but not all edges, also no cycle is formed while covering all the vertices then such a component of a graph is called a spanning tree.

- We can also define a spanning tree of a connected graph G as a subgraph of G that contains all the vertices.

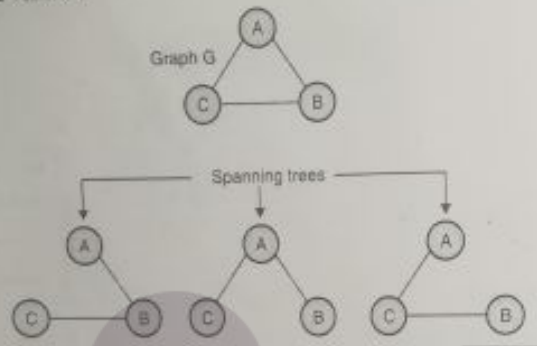


Fig. 3.6.1

- There are two famous algorithms for this problem:
  1. Kruskal's algorithm 2. Prim's algorithm

3.6.1 Kruskal's Algorithm

- Kruskal's algorithm uses the greedy approachto find the minimum cost spanning tree.
- In this algorithm, all the edges of the graph G are ordered in increasing order of their weight.
- Add all the edges one by one in increasing order skipping those edges whose addition would create a cycle.

Algorithm

```
def kruskal(G):
edges = []
for v in G:
makeSet(v.getVertexID())
for w in v.getConnections():
vid = v.getVertexID()
wid = w.getVertexID()
edges.append((v.getWeight(w),vid, wid))
edges.sort()
minimumSpanningTree = set()
for edge in edges:
weight, vertice1, vertice2 = edge
if find(vertice1) != find(vertice2):
union(vertice1, vertice2)
minimumSpanningTree.add(edge)
return minimumSpanningTree
```


Example

- Let us consider the following example :

Fig. 3.6.2

Step 1 : Remove all loops and Parallel Edges from the given graph. In case of parallel edges, keep the one which has the least cost associated and remove all others.

Fig. 3.6.3

Step 2 : Arrange all the edges in increasing order.

Edge	B, D	D, T	A, C	C, D	B, C	B, T	A, B	A, S	S, C
weight	2	2	3	3	4	5	6	7	8

Step 3 : Add edges one by one in increasing order but avoid those edges which can create a cycle. So, we will add the following edges.

Edge	weight
(B, D)	2
(D, T)	2
(A, C)	3
(C, D)	3
(A, S)	7

We have not added the edges (B, C), (B, T), (A, B) and (S, C) because if we try to add these edges, it will create a cycle. So, ignore such edges.

we now have minimum cost spanning tree having cost as follows:

$$2 + 2 + 3 + 3 + 7 = 17$$

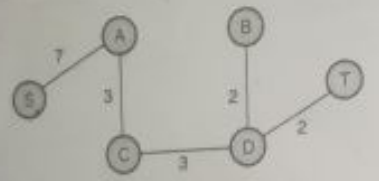


Fig. 3.6.4

3.6.2 Prim's Algorithm

- This algorithm is used to find minimum spanning tree for connected weighted undirected graph. Prim's algorithm starts by choosing an arbitrary vertex node of the graph.
- At each step, a new node will be added to the tree.
- This algorithm stops when all the nodes from the graph are added to the tree.

Algorithm

```
def Prims(G, source):
print "Dijkstra Modified for Prim"
# Set the distance for the source node to zero
source.setDistance(0)
# Put tuple pair into the priority queue
unvisitedQueue = [(v.getDistance(),v) for v in G]
heapq.heapify(unvisitedQueue)
while len(unvisitedQueue):
  #Pops a vertex with the smallest distance
  uv = heapq.heappop(unvisitedQueue)
  current = uv[1]
  current.setVisited()
  #for next in v.adjacent:
  for next in current.adjacent:
    # if visited, skip
    if next. visited:
      continue
    newCost = current.getWeight(next)
    if newCost < next.getDistance():
      next.setDistance(current.getWeight(next))
      next.setPrevious(current)
      print 'Updated : current = %s next = %s newCost = %s'
            %(current.getVertexID(), next.getVertexID(), next.getDistance())
```



```
else:
print 'Not updated : current %s next = %s newCost %s'
    % (current.getVertexID(), next.getVertexID(), next.getDistance())
# Rebuild heap
# 1. Pop every item
while len(unvisitedQueue):
    heapq.heappop(unvisitedQueue)
# 2. Put all vertices not visited into the queue
  unvisitedQueue = [(v.getDistance(),v) for v in G if not v.visited]
      heapq.heapify(unvisitedQueue)
```

Example

- To understand the prim's algorithm consider the following example:

Fig. 3.6.5

First choose any arbitrary node as root node. In this case, we choose S node as the root node of Prim's spanning tree. Now,

Step 1 : S = { NULL } , G = {S, A, B, C, D, T}

Now check which are the outgoing edges of S and chose the one which have less cost. s = {S},

G = {A, B, C, D, T }
= min{(S, A), (S, B)} = min{7, 8} = 7

We choose the edge (S,A) as it has lesser value.

Step 2 : s = {S, A},

G = {B, C, D, T }
= min{(S, C), (A, C), (A, B)} = min{8, 3, 6} = 3

We choose the edge (A, C) as it has lesser value.

Step 3 : s = {S, A, C},

G = { B, D, T}
= min{(S, C), (B, C), (A, B), (C, D)}
= min{8, 4, 6, 3} = 3

We choose the edge (C, D) as it has lesser value.

Step 4 : S = {S, A, C, D},

G = { B, T}

= min{(S, C), (C, B), (A, B), (D, B), (D, T), (C, D)}

= min{8, 4, 6, 2, 2, 3} = 2

We choose the edge (D, B) as it has lesser value.

Step-5: S = {S, A, C, D, B},

G = {T}

= min{(S, C), (C, B), (A, B), (D, T), (C, D), (B, T)}

= min{8, 4, 6, 2, 3, 5} = 2

We choose the edge (D, T) as it has lesser value.

Fig. 3.6.6

Syllabus Topic : Algorithms : What are selection algorithms?

3.7 Selection Algorithms

- Selection algorithm is an algorithm for finding the k^{th} smallest/largest number in a list (also called as k^{th} order statistics).
- This includes finding the minimum, maximum, and median elements. For finding the k^{th} order statistic, there are multiple solutions which provide different complexities, and in this chapter we will enumerate those possibilities.

Algorithm 3.7.1

Write an algorithm to find the largest element in an array A of size n.

Solution :

```
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).
```


Algorithm 3.7.2

Write an algorithm to find the smallest and largest elements in an array II 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).
```

Syllabus Topic : Selection by Sorting

3.8 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(n log n) time (for sorting) where n is the length of the input list.

Syllabus Topic : Partition based Selection Algorithm

3.9 Partition based Selection Algorithm

- **Quick select** is a selection algorithm to find the k^{th} smallest element in an unordered list. It is related to the quicksort sorting algorithm.
- Quickselect uses the same approach as quicksort, it chooses one element as a pivot and partition the data in two based on the pivot, accordingly as less than or greater than the pivot.
- However, instead of recursing into both sides, as in quicksort, quickselect only recourses into one side – the side with the element it is searching for. This reduces the average complexity from O(n log n) to O(n), with a worst case of $O(n^2)$.

Algorithm for Quickselect

- Suppose we are given an unsorted sequence S of n comparable elements together with an integer $k \in [1,n]$. At a high level, the quick-select algorithm for finding the k^{th} smallest element in S. We pick a "pivot" element from S at random and use this to subdivide S into three subsequences L, E, and G, storing the elements of S less than, equal to, and greater than the pivot, respectively.
- We determine which of these subsets contains the desired element, based on the value of k and the sizes of those subsets. We then recur on the appropriate subset, noting that the desired element's rank in the subset may differ from its rank in the full set.

```
def quick_select(S, k):
""" Return the k^th smallest element of list S, for k from 1 to len(S)."""
if len(S) == 1:
  return S[0]
pivot = random.choice(S)          # pick random pivot element from S
L = [x for x in S if x < pivot]   # elements less than pivot
E = [x for x in S if x == pivot]  # elements equal to pivot
G = [x for x in S if pivot < x]   # elements greater than pivot
if k <= len(L):
return quick_select(L, k)         # kth smallest lies in L
```



```
elif k <= len(L) + len(E):
return pivot                      # kth smallest equal to pivot
else:
  j = k - len(L) - len(E)         # new selection parameter
return quick_select(G, j)         # kth smallest is jth in G
```

Syllabus Topic : Linear Selection Algorithm : Median of Medians Algorithm

3.10 Linear Selection Algorithm - Median of Medians Algorithm

- The median of medians is an approximate selection algorithm, frequently used to supply a good pivot for an exact selection algorithm.
- Median of medians finds an approximate median in linear time only.
- The **median-of-medians algorithm** chooses the pivot in the following way :
  1. Divide the list into sublists of length five. (Note that the last sublist may have length less than five.)
  2. Sort each sublist and determine its median directly.
  3. Use the median of medians algorithm to recursively determine the median of the set of all medians from the previous step.
  4. Use the median of the medians from step 3 as the pivot.

Median of Medians algorithm

```
CHUNK_SIZE = 5
def kthByMedianOfMedian(unsortedList, k):
if len(unsortedList) <= CHUNK_SIZE:
return get_kth(unsortedList, k)
chunks = splitIntoChunks(unsortedList, CHUNK_SIZE)
medians_list = []
for chunk in chunks:
median_chunk = get_median(chunk)
medians_list.append(median_chunk)
size = len(medians_list)
mom = kthByMedianOfMedian(medians_list, size / 2 + (size % 2))
smaller, larger = splitListByPivot(unsortedList, mom)
```

```
valuesBeforeMom = len(smaller)
if valuesBeforeMom == (k - 1):
return mom
elif valuesBeforeMom > (k - 1 ):
return kthByMedianOfMedian(smaller, k)
else:
return kthByMedianOfMedian(larger, k - valuesBeforeMom - 1)
```

Syllabus Topic : Finding The K Smallest Elements in Sorted Order

3.11 Finding the K Smallest Elements in Sorted Order

Given a set of n elements from a totally-ordered domain, find the k smallest elements, and list them in sorted order.

```
class Solution
def find MedianSortedArrays(self, A, B):
#comparing middle elements of A and B, which we identify as Ai and Bj. If Ai is between Bj
and Bj-1, we have just found the i+j+1 smallest element. Therefore, if we choose i and j such
that i+j = k-1, we are able to find the k-th smallest element.
def findkth(a,b,k):
        lena = len(a)
        lenb = len(b)
if lena > lenb:
return findkth(b,a,k)
if a == []: return b[k-1]
if k == 1: return min(a[0],b[0])
        parta = min(k/2, lena)
        partb = k - parta
if a[parta-1] < b[partb-1]: #delete impossible value from a
return findkth(a[parta:], b, k-parta)
else:                       #delete impossible value from b
return findkth(a, b[partb:], k-partb)
    length = len(A)+len(B)
if length % 2 == 0:
```



```
return (findkth(A,B,length/2)+findkth(A,B,length/2+1))*0.5
else:
return findkth(A,B,length/2+1)*1.0
```

Review Questions

Q. 1 List the applications of graph. (Refer section 3.1.3)

Q. 2 Explain adjacency matrix and adjacency list. (Refer sections 3.2.1 and 3.2.2)

Q. 3 Write short note on : DFS and BFS (Refer sections 3.3.1 and 3.3.2)

Q. 4 Write DFS algorithm. (Refer section 3.3.1)

Q. 5 Explain working of topological sort. (Refer section 3.4)

Q. 6 Write Dijkstra's algorithm. (Refer section 3.5.1)

Q. 7 Explain Kruskal and Prim's algorithms. (Refer sections 3.6.1 and 3.6.2)

❑❑❑

UNIT III

CHAPTER 4

Algorithms Design Techniques & Greedy Algorithms

Syllabus

Algorithms Design Techniques : Introduction, Classification, Classification by Implementation Method, Classification by Design Method

Greedy Algorithms : Introduction, Greedy Strategy, Elements of Greedy Algorithms, Advantages and Disadvantages of Greedy Method, Greedy Applications, Understanding Greedy Technique.

Syllabus Topic : Algorithms Design Techniques - Introduction

4.1 Introduction

- In the previous chapters, we had seen many algorithms to solved different types of problem.
- Generally we need to look for the similarity of current problem to other problem for which we have solutions. This help for future reference.
- In this chapter, we will see different ways of classifying the algorithm and in subsequent chapters.
- It is covering Greedy, Divide and Conquer, Dynamic Programming.

Syllabus Topic : Classification

→ 4.1.1 Classification of Algorithms

Following are the ways of classifying algorithms.

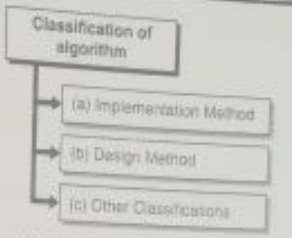


Fig. C4.1 : Classification of Algorithms

Syllabus Topic : Classification by Implementation Method

→ 4.1.1(a) Classification by Implementation Method

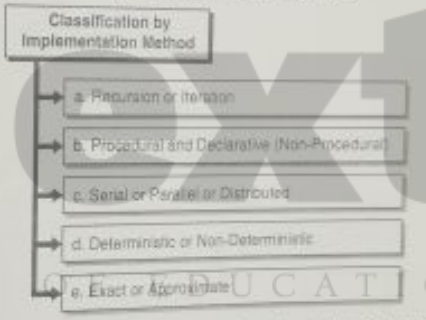


Fig. C4.2 : Classification by Implementation Method

→ a. Recursion or Iteration

- A recursion algorithm is the one that calls itself repeatedly until condition is true. It is common method used in programming language like C, C++, java etc. specifically use to calculate factorial of given number.
- Iterative algorithm use constructs like loops and sometimes data structure like Stack and Queue to solve the problem.
- Some problem like Tower of Hanoi is easily done by recursive method. Every recursive method has iterative method and vice versa.

→ b. Procedural and Declarative (Non-Procedural)

In non-procedural language like SQL, we only tell what to do but not tell how to do it. Where as in procedural language like C, PHP, Java etc. , we tell what to do and as well as how to do it.

c. Serial or Parallel or Distributed

- Computer executes instruction one after the other in serial order. These are called as serial algorithms.
- Computer executes some of instruction simultaneously using threads, this call as parallel algorithm.
- If the parallel algorithms are distributed on different machine then we call as Distributed algorithms.

d. Deterministic or Non-Deterministic

Deterministic algorithms solve the problem with a predefined process, where as non-deterministic algorithms guess the best solution at each step through the use of heuristics or random techniques.

e. Exact or Approximate

The algorithms for which we are able to find the optimal solutions are called Exact Algorithms. If we do not have optimal solution to problem, then we called it as approximate algorithms.

Syllabus Topic : Classifications by Design Method

4.1.1(b) Classifications by Design Method

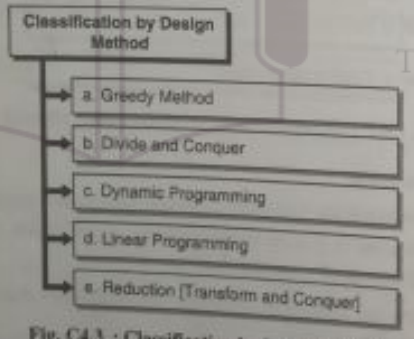


Fig. C4.3 : Classification by Design Method

a. Greedy Method

- Greedy algorithm works in stages. In each stage, a decision is made that good is good at that time. Without bothering about future consequences, this means that some 'local best' is chosen.
- It assumes that local best solution selection is best and also makes for global optimal solution.

b. Divide and Conquer

The divide and conquer method solves 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.

c. Dynamic Programming

- Dynamic programming (DP) and memorization 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, where as in DP there will be an overlap of sub-problems.
- By using memorization [maintaining a table for already solved sub problems].

d. Linear Programming

- 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.

e. 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.

4.1.1(c) Other Classifications of Algorithm

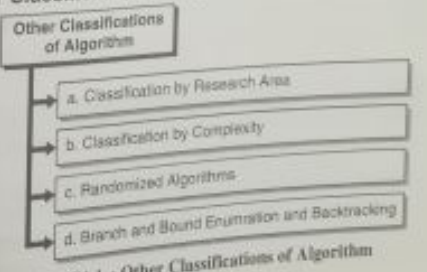


Fig. C4.4 : Other Classifications of Algorithm

➜ **a. 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.

➜ **b. 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.

➜ **c. Randomized Algorithms**

A few algorithms make choices randomly. For some problems, the fastest solutions must involve randomness. Example: Quick Sort.

➜ **d. Branch and Bound Enumration and Backtracking**

These were used in Artificial Intelligence and we do not need to explore these fully. For the Backtracking method refer to the Recursion and Backtracking chapter.

Syllabus Topic : Greedy Algorithms : Introduction

4.2 Introduction of Greedy Algorithms

- A greedy algorithm, as the name suggests, always makes the choice that seems to be the best at that moment. This means that it makes a locally-optimal choice in the hope that this choice will lead to a globally-optimal solution.
- The Greedy technique is best suited for looking at the immediate situation.

4.2.1 Greedy Strategy

- A greedy algorithm is an algorithmic paradigm that follows the problem solving heuristic of making the locally optimal choice at each stage with the aim of finding a global optimum.
- In many problems, a greedy strategy does not in general produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that help to produce a global optimal solution in a reasonable time.
- For example, a greedy strategy for the travelling salesman problem (which is of a high computational complexity) is that: "At each stage visit an unvisited city nearest to the current city". This will need not find a best solution, but terminates in a reasonable number of steps; finding an optimal solution typically requires unreasonably many steps.

- That overall means Greedy algorithms try to find a localized optimum solution, which may eventually lead to globally optimized solutions.

Syllabus Topic : Elements of Greedy Algorithms

4.2.2 Elements of Greedy Algorithms

- The greedy algorithm work on two properties.

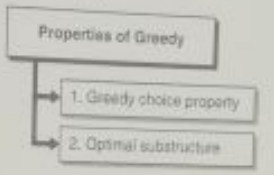


Fig. C4.5 : Properties of Greedy

➜ **1. Greedy choice property**

- We can make whatever choice seems best at the moment and then solve the subproblems that arise later. 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 subproblem.
- 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.

➜ **2. Optimal substructure**

"A problem exhibits optimal substructure if an optimal solution to the problem contains optimal solutions to the sub-problems." That means we can solve subproblems and build up the solutions to solve larger problems.

Syllabus Topic : Advantages and Disadvantages of Greedy Method

4.2.3 Advantages and Disadvantages of Greedy Method

Advantages of Greedy Method

- The Greedy method is that it is straight forward, easy to understand and easy to code.

- Finding solution is quite easy with a greedy algorithm for a problem.
- Analyzing the run time for greedy algorithms will generally be much easier than for other techniques (like Divide and conquer).
- Once we make a decision, we do not have to spend time re-examining the already computed values.

Disadvantages of Greedy Method

- The difficult part is that for greedy algorithms you have to work much harder to understand correctness issues.
- Even with the correct algorithm, it is hard to prove why it is correct. Proving that a greedy algorithm is correct is more of an art than a science.
- In many cases there is no guaranteed that making locally optimal improvements in a locally optimal solution gives the optimal global solution.

Syllabus Topic : Greedy Applications

4.3 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

Syllabus Topic : Understanding Greedy Technique

4.4 Understanding Greedy Technique

To understand Greedy Algorithm lets go through with following examples.

4.4.1 Counting Coins

This problem is to count to a desired value by choosing the least possible coins and the greedy approach forces the algorithm to pick the largest possible coin. If we are provided coins of ₹1, ₹ 5, ₹10 and ₹ 20 (Yes, We've ₹ 20 coins : D) and we are asked to count ₹ 36 then the greedy procedure will be –

Select one ₹ 20 coin, the remaining count is 16

- Then select one ₹ 10 coin, the remaining count is 6
- Then select one ₹ 5 coin, the remaining count is 1
- And finally, the selection of one ₹ 1 coins solves the problem

36 – 20 = 16 (20)
16 – 10 = 6 (20) (10)
6 – 5 = 1 (20) (10) (5)
1 – 1 = 16 (20) (10) (5) (1)

Fig. 4.4.1

- Though, it seems to be working fine, for this count we need to pick only 4 coins. But if we slightly change the problem then the same approach may not be able to produce the same optimum result.
- For the currency system, where we have coins of ₹ 1, ₹ 6, ₹ 10 and ₹ 20 value, counting coins for value 36 will be absolutely optimum but for count like 32, it may use more coins than necessary.
- For example, the greedy approach will use 20 + 10 + 1 + 1, total 4 coins. Whereas the same problem could be solved by using only 3 coins (20 + 6 + 6)

4.4.2 Finding Largest Sum

Fig. 4.4.2

- If we've a goal of reaching the largest-sum, at each step, the greedy algorithm will choose what appears to be the optimal immediate choice, so it will choose 12 instead of 3 at the second step, and will not reach the best solution, which contains 99.

- Hence, we may conclude that the greedy approach picks an immediate optimized solution and may fail where global optimization is a major concern.

4.4.3 Huffman Coding Algorithm

- Given a set of n characters from Let alphabet 'A' each character c € A/and their associated frequency freq(c).Find a binary code for each character c € 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 lo 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 frequency 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.

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).

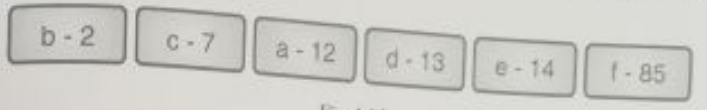


Fig. 4.4.3

- 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:

Fig. 4.4.4

- Repeat this process until only one tree is left.

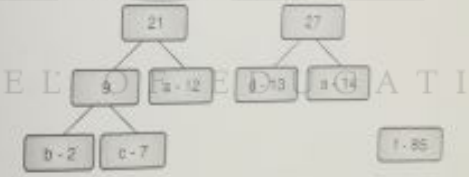


Fig. 4.4.5

Fig. 4.4.6

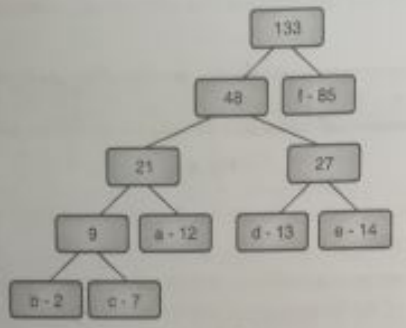


Fig. 4.4.7

- Once the tree is built, each leaf node corresponds to a letter with n 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.

Character	Frequency
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)
for pair in lo[1:]
  pair[1] = '0' + pair[1]
for pair in hi[1:]
  pair[1] = '1' + pair[1]
heappush(heap,[lo[0]+hi[0]]+lo[1:]+hi[1:])
return sorted(heappop(heap)[1:],key=lambda p:[len(p[-1]), p])
inputText = "this is an example for Huffman encoding"
characterFrequency = defaultdict(int)
for character in inputText :
characterFrequency[character] + = 1
huffCodes = HuffmanEncode (characterFrequency)
print "Symbol \tFrequency\tHuffman code "
for p in huffCodes :
print %s\t\t\t%s\t\t\t%s"% (p[0], characterFrequency[p[0]],p[1])
```

Time Complexity: O(nlogn), since there will be one build heap, 2n - 2 delete mins, and n - 2 inserts, on a priority queue that never has more than n elements. Refer to the priority Queues chapter for details.

4.4.4 Problem

The following algorithm gives the optimal solution?

Algorithm : Merge the files in pairs. That means after the first step, the algorithm produces the n/2 intermediate files. For the next step, we need to consider these intermediate files and merge them in pairs and keep going.

Note : Sometimes this algorithm is called 2-way merging. Instead of two files at a time, if we merge K files at a time then we call is K -way merging.

- Solution: This algorithm will not produce the optimal solution and consider the previous example for a counter example. As per the above algorithm, we need to merge the first pair of files (10 and 5 size files), the second pair of files (100 and 50) and the third pair of files (20 and 15). As a result we gel the following list of files. {15, 150, 35}
- Similarly, merge the output in pairs and this step produces [below, the third element does not have a pair element, so keep it the same
- Finally, 165,35}, {185}
- The total cost of merging = Cost of all merging operations = 15 + 150 + 35 + 165 + 185 = 550. This is much more than 395 (of the previous problem). So, the given algorithm is not giving the best (optimal) solution.

Review Questions

Q. 1 Describe classifications of algorithm in detail. **(Refer sections 4.1.1, 4.1.1(a), 4.1.1(b), 4.1.1(c))**

Q. 2 Explain properties of greedy algorithms. **(Refer section 4.2.2)**

Q. 3 Write short note on advantages and disadvantages of greedy method. **(Refer section 4.2.3)**

Q. 4 Explain Huffman coding algorithm with example. **(Refer section 4.4.3)**

UNIT III

CHAPTER 5

Divide and Conquer Algorithms

Syllabus

Introduction, What is Divide and Conquer Strategy? Divide and Conquer Visualization, Understanding Divide and Conquer, Advantages of Divide and Conquer, Disadvantages of Divide and Conquer, Master Theorem, Divide and Conquer Applications

Syllabus Topic : Introduction

5.1 Introduction

In the Greedy chapter, we have seen that for many problems the Greedy strategy failed to provide optimalsolutions. From those problems, there are some that can be easily solved by using the Divide and Conquer(D & C) technique. Divide and Conquer is a popular technique for algorithm design.

Syllabus Topic : What is Divide and Conquer Strategy?

5.1.1 Divide and Conquer Strategy

- This paradigm, **divide-and-conquer**, breaks a problem into subproblems that are similar to the original problem, recursively solves the subproblems, and finally combines the solutions to the subproblems to solve the original problem.
- Because divide-and-conquer solves subproblems recursively, each subproblem must be smaller than the original problem, and there must be a base case for subproblems.
- A divide-and-conquer algorithm can be described using three parts :
  1. Divide the problem into a number of subproblems that are smaller instances of the same problem.

2. Conquer the subproblems by solving them recursively. If they are small enough, solve the subproblems as base cases.
3. Combine the solutions to the subproblems into the solution for the original problem.

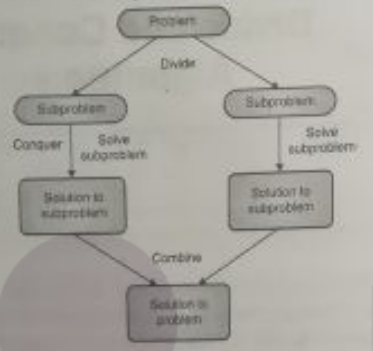


Fig. 5.1.1

- If we expand out two more recursive steps, it looks like this.

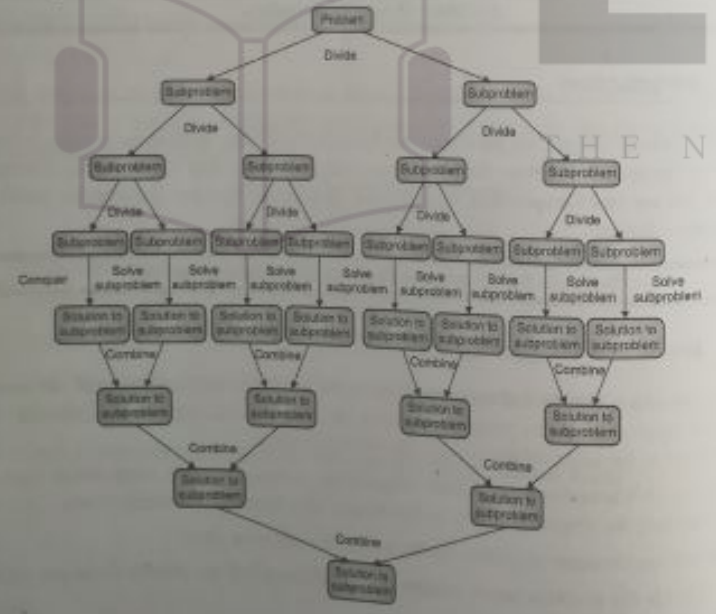


Fig. 5.1.2

- Because divide-and-conquer creates at least two subproblems, a divide-and-conquer algorithm makes multiple recursive calls.
- All the time it is not possible to solve all the problems with the Divide and Conquer technique. As per the definition of D and C, the recursion solves the subproblems which are of the same type.

Syllabus Topic : Divide and Conquer Visualization

5.2 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.

```
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 )
    )
)
```

Syllabus Topic : Understanding Divide and Conquer

5.2.1 Understanding Divide and Conquer

- For a clear understanding of D and 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 become enemy of each 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.

Syllabus Topic : Advantages of Divide and Conquer

5.2.2 Advantages of Divide and Conquer

- The first, and probably most recognizable benefit of the divide and conquer paradigmis the fact that it allows us to solve difficult and often impossible looking problems such as the Tower of Hanoi, which is a mathematical game or puzzle. Being given a difficult problem can often be discouraging if there is no idea how to go about solving it.
- The divide and conquer method, it reduces the degree of difficulty since it divides the problem into sub problems that are easily solvable and usually runs faster than other algorithms would.
- It often plays a part in finding other efficient algorithms, and in fact it was the central role in finding the quick sort and merge sort algorithms.
- It also uses memory caches effectively. The reason for this is the fact that when the sub problems become simple enough, they can be solved within a cache, without having to access the slower main memory, which saves time and makes the algorithm more efficient.
- And in some cases, it can even produce more precise outcomes in computations with rounded arithmetic than iterative methods would.
- Packaged with all of these advantages, however, are some weaknesses in the process.

Syllabus Topic : Disadvantages of Divide and Conquer

5.2.3 Disadvantages of Divide and Conquer

- One of the most common issues with this sort of algorithm is the fact that the recursion is slow, which in some cases outweighs any advantages of this divide and conquer process.
- Another concern with it is the fact that sometimes it can become more complicated than a basic iterative approach, especially in cases with a large n.
- In other words, if someone wanted to add a large amount of numbers together, if they just create a simple loop to add them together, it would turn out to be a much simpler approach

than it would be to divide the numbers up into two groups, add these groups recursively, and then add the sums of the two groups together.

- Another downfall is that sometimes once the problem is broken down into sub problems, the same sub problem can occur many times.
- In cases like these, it can often be easier to identify and save the solution to the repeated subproblem, which is commonly referred to as memorization.
- And the last recognizable implementation issue is that these algorithms can be carried out by a non-recursive program that will store the different sub problems in things called explicit stacks, which gives more freedom in deciding just which order the sub problems should be solved.
- These implementation issues do not make this process a bad decision when it comes to solving difficult problems, but rather this paradigm is the basis of many frequently used algorithms

Syllabus Topic : Master Theorem

5.3 Master Theorem

- We assume a divide and conquer algorithm in which a problem with input size n is always divided into a subproblems, each with input size n/b. Here a and b are integer constants with $a \geq 1$ and $b > 1$.
- We assume n is a power of b, say $n = b^k$.
- Otherwise at some stage we will not be able to divide the sub-problem size exactly by b.
- However, the Master Theorem still holds if n is not a power of b, and the subproblem input sizes are [n/b] or [n/b]

  Note $k = \log_b(n)$
- The recurrence for the running time is :

  $$T(n) = a\,T(n/b) + f(n),\ T(1) = d$$
- Here f(n) represents the divide and combine time (i.e., the non-recursive time). f(n) may involve θ, e.g., $f(n) = \theta(n^2)$.

  We define $E = \log_b(a)$.

  E is called the critical exponent. (It strongly influences the solution) By definition, $b^E = a$.

  Note that $a^k = n^E$

  Why ? $a^k = (b^E)^k = (b^k)^E = n^E$.
- We can write down the total time to solve all sub-problems at a given depth in the recursion tree.

Depth of recursion	Size of sub-problems	Number of sub-problems	Total (non-recursive) time at this depth is roughly proportional to
0	n	1	f(n)
1	n/b	a	af (n/b)
2	n/b^2	a^2	$a^2 f(n/b^2)$
3	n/b^3	a^3	$a^3 f(n/b^3)$
⋮	⋮	⋮	⋮
k – 2	n/b^{k-2}	a^{k-2}	$a^{k-2} f(n/b^{k-2})$
k – 1	$n/b^{k-1} = b$	a^{k-1}	$a^{k-1} f(n/b^{k-1}) = \theta(n^E)$
k	$n/b^k = 1$	$a^k = n^E$	$a^k d = O(n^E)$

$T(n)$ = Sum of terms in rightmost column above

$= f(n) + af(n/b) + a^2 f(n/b)^2 + \ldots + a^{k-1} f(n/b^{k-1}) + a^k d$

The critical functions in determining T(n) are :

(i) f(n) : (the non-recursive time at depth 0)

(ii) n^E : (the non-recursive time at depth k, or k – 1).

Clearly : $T(n) \geq \theta(\max(n^E, f(n)))$.

On the other hand, if the terms in the right hand column of the table either increase as we move down, or decrease as we move down,

then : $T(n) \leq \theta(\max(n^E, f(n)) \cdot \log_b(n))$.

We will see that, if one of n^E and f(n) grows much more rapidly than the other, then $T(n) \leq \theta$ (more rapidly growing function).

(1) $f(n)$ in $O(n^{E-\epsilon})$ for fixed $\epsilon > 0$ implies $T(n) = \theta(n^E)$.

(2) $f(n)$ in $\theta(n^E)$ implies $T(n) = \theta(n^E \log_b(n))$.

(3) $f(n)$ in $\Omega(n^{E+\epsilon})$ for fixed $\epsilon > 0$ implies $T(n) = \theta(f(n))$.

Actually, (3) requires an additional hypothesis, that typically holds.

Note none of these cases may apply. For example, if $f(n) = n^E \log_b(n)$, we are between cases (2) and (3) : neither case holds.

Syllabus Topic : Divide and Conquer Applications

5.4 Divide and Conquer Applications

Following are some problems, which are solved using divide and conquer approach.

1. Finding the maximum and minimum of a sequence of numbers
2. Strassen's matrix multiplication
3. Merge sort
4. Binary search

Let us consider a simple problem that can be solved by divide and conquer technique.

Example 5.4.1

Statement : The Max-Min Problem in algorithm analysis is finding the maximum and minimum value in an array.

Solution :

To find the maximum and minimum numbers in a given array numbers[] of size n, the following algorithm can be used. First we are representing the naive method and then we will present divide and conquer approach.

Naïve Method

Naïve method is a basic method to solve any problem. In this method, the maximum and minimum number can be found separately. To find the maximum and minimum numbers, the following straightforward algorithm can be used.

Algorithm : Max-Min-Element (numbers[])

```
max := numbers[1]
min := numbers[1]
for i = 2 to n do
if numbers[i] > max then
max := numbers[i]
if numbers[i] < min then
min := numbers[i]
return (max, min)
```

Analysis

- The number of comparison in Naive method is 2n - 2.
- The number of comparisons can be reduced using the divide and conquer approach. Following is the technique.

Divide and Conquer Approach

In this approach, the array is divided into two halves. Then using recursive approach maximum and minimum numbers in each halves are found. Later, return the maximum of two maxima of each half and the minimum of two minima of each half.

In this given problem, the number of elements in an array is y−x+1y−x+1, where y is greater than or equal to x.

Max–Min(x, y)Max–Min(x, y) will return the maximum and minimum values of an array numbers[x...y]numbers[x...y].

Algorithm

```
Max - Min(x, y)
if x – y ≤ 1 then
return (max(numbers[x], numbers[y]), min((numbers[x], numbers[y]))
else
  (max1, min1) = maxmin(x, ⌊((x + y)/2)⌋)
  (max2, min2) = maxmin(⌊((x + y)/2) + 1⌋,y)
return (max(max1, max2), min(min1, min2))
```

Analysis

Let T(n) be the number of comparisons made by Max – Min(x,y) Max – Min(x,y), where the number of elements n = y – x + 1n = y – x + 1.

If ***T(n)*** represents the numbers, then the recurrence relation can be represented as

$$T(n) = \begin{cases} T\left(\frac{1}{2}\right) + T\left(\frac{1}{2}\right) + 2 & \text{for } n > 2 \\ 1 & \text{for } n = 2 \\ 0 & \text{for } n = 1 \end{cases}$$

Let us assume that n is in the form of power of 2. Hence, n = 2k where k is height of the recursion tree. So,

$$\begin{aligned} T(n) &= 2.T(n^2) + 2 = 2.(2.T(n^4) + 2) + 2.... \\ &= 3n^2 - 2T(n) = 2.T(n^2) + 2 = 2.(2.T(n^4) + 2) + 2..... \\ &= 3n^2 - 2 \end{aligned}$$

Compared to Naive method, in divide and conquer approach, the number of comparisons is less. However, using the asymptotic notation both of the approaches are represented by O(n).

This chapter, we will discuss merge sort and analyze its complexity.

Example 5.4.2

Statement : The problem of sorting a list of numbers lends itself immediately to a divide-and-conquer strategy: split the list into two halves, recursively sort each half, and then merge the two sorted sub-lists.

Solution :

In this algorithm, the numbers are stored in an array numbers[]. Here, p and q represents the start and end index of a sub-array.

Algorithm

```
Merge-Sort (numbers[], p, r)
if p < r then
q = ⌊(p + q) / 2⌋
Merge-Sort (numbers[], p, q)
  Merge-Sort (numbers[], q + 1, r)
  Merge (numbers[], p, q, r)
Function: Merge (numbers[], p, q, r)
n1 = q – p + 1
n2 = r – q
declare leftnums[1...n1 + 1] and rightnums[1...n2 + 1] temporary arrays
for i = 1 to n1
leftnums[i] = numbers[p + i - 1]
for j = 1 to n2
rightnums[j] = numbers[q + j]
leftnums[n1 + 1] = ∞
rightnums[n2 + 1] = ∞
i = 1
j = 1
for k = p to r
if leftnums[i] ≤ rightnums[j]
numbers[k] = leftnums[i]
  i = i + 1
else
numbers[k] = rightnums[j]
  j = j + 1
```

Analysis

Let us consider, the running time of Merge-Sort as ***T(n)***. Hence,

$$T(n) = \begin{cases} c & \text{if } n \le 1 \\ 2xT\left(\frac{n}{2}\right) + dxn & \text{otherwise} \end{cases}$$

Where c and b are constants

Therefore, using this recurrence relation,

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

As $i = \log n$,

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

$$= c.n + d.n.\log n$$

Therefore, $T(n) = O(n \log n)$

Example

In the following example, we have shown Merge-Sort algorithm step by step. First, every iteration array is divided into two sub-arrays, until the sub-array contains only one element. When these sub-arrays cannot be divided further, then merge operations are performed.

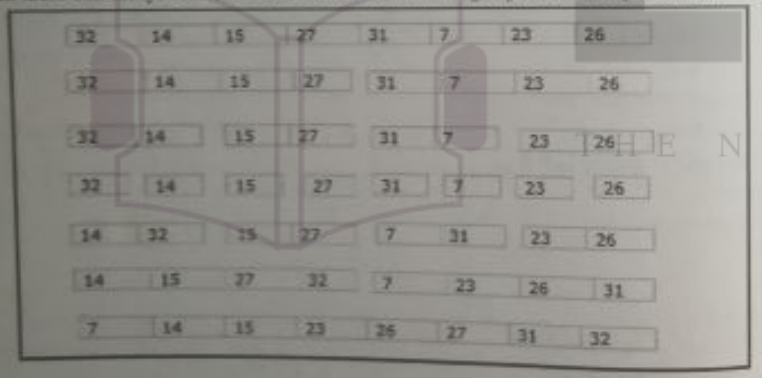


Fig. P. 5.4.2

Example 5.4.3

Let we consider an algorithm 'A' which solves problems by dividing them into five sub problems of half the size, recursively solving each sub problem, and then combining the solutions in linear time. What is the complexity of this algorithm?

Solution :

Let us assume that the input size is 'n' and T(n) defines the solution to the given problem.

As per the description, the algorithm divides the problem into 'S' sub problems with each of size (n/2). So we need to solve 5T(n/2) subproblems. After solving these sub problems, the given array (linear time) is scanned to combine these

Solution :

The total recurrence algorithm for this problem can be given as:

$$T(n) = 5T(n/2) + O(n).$$

Using the Master theorem (of D &C), we get the complexity as

$$O(n^{\log(5/2)}) = O(n^{2+}) = O(n^2).$$

Example 5.4.4

Similar to above Problem, An algorithm B solves problems of size n by recursively solving two sub problems of size n-1 and then combining the solutions in constant time. What is the complexity of this algorithm?

Solution :

Let us assume that the input size is n and T(n) defines the solution to the given problem. As per the description of algorithm we divide this problem into 2 sub problems with each of size n - 1. So we have to solve 2T(n - 1) subproblems. After solving these sub problems, the algorithm takes only a constant time to combine these solutions. The total recurrence algorithm for this problem can be given as:

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

Using Master theorem (of Divide and Conquer), we get the complexity as $O(n^{\log 2/2^0}) = O(2^n)$. (Refer to Introduction chapter for more details)

Example 5.4.5

Write a recurrence and solve it.

```
def function(n):
if(n > 1):
print((*n))
function(n/2)
function(n/2)
```

Solution :

Let us assume that input size is n and T(n) defines the solution to the given problem. As per the given code, after printing the character and dividing the problem into 2 subproblems with each of size (n/2) and solving them. So we need to solve 2T(n/2) subproblems. After solving these subproblems, the algorithm is not doing anything for combining the solutions. The total recurrence algorithm for this problem can be given as:

$$T(n) = 2T(n/2) + O(1)$$

Using Master theorem (of D & C), we get the complexity as $O(n^{\log(2/2)}) = O(n^1) = O(n)$.

Example 5.4.6

Stock Pricing Problem

Consider the stock price of Career Monk.com in n consecutive days. That means the input consists of an array with stock prices of the company. We know that the stock price will not be the same on all the days. In the input stock prices there may be dates where the stock is high when we can sell the current holdings, and there may be days when we can buy the stock. Now our problem is to find the day on which we can buy the stock and the day on which we can sell the stock so that we can make maximum profit.

Solution :

As given in the problem, let us assume that the input is an array with stock prices [integers]. Let us say the given array is A[1], ... A[n]. From this array we have to find two days [one for buy and one for sell] in such a way that we can make maximum profit. Also, another point to make is that the buy date should be before sell date. One simple approach is to look at all possible buy and sell dates.

```
def calculateProfitWhenBuyingNow(A, index):
buyingPrice = A[index]
maxProfit = 0
sellAt = index
for i in range(index + 1, len(A)):
sellingPrice = A[i]
profit = sellingPrice-buyingPrice
if profit >maxProfit:
maxProfit = profit
sellAt = i
return maxProfit, sellAt
#check all possible buying times
def StockStrategyBruteForce(A):
maxProfit =None
buy = None
sell = None
for index, item in enumerate(A):
profit, sellAt = calculateProfitWhenBuyingNow(A , index)
if (maxProfit is None) or (profit >maxProfit):
maxProfit = profit
buy= index
sell = sellAt
return maxProfit, buy, sell
```

The two nested loops take $n(n + 1)/2$ computations, so this takes time $\Theta(n^2)$.

Review Questions

Q.1 Write short note on Divide and Conquer strategy. (Refer section 5.1.1)

Q.2 Explain advantages of divide and conquer. (Refer section 5.2.2)

Q.3 Explain disadvantages of divide and conquer. (Refer section 5.2.3)

Q.4 Describe master theorem in detail. (Refer section 5.3)

❑❑❑

UNIT III

CHAPTER 6

Dynamic Programming

Syllabus

Introduction, What is Dynamic Programming Strategy? Properties of Dynamic Programming Strategy, Problems which can be solved using Dynamic Programming, Dynamic Programming Approaches, Examples of Dynamic Programming Algorithms, Understanding Dynamic Programming, Longest Common Subsequence.

Syllabus Topic : Introduction

6.1 Introduction

Invented in 1950 by Richard Bellman an U.S mathematician as a general method for optimizing multistage decision processes. It is a planning concept, which is a technique for solving problems with overlapping sub problems.

Syllabus Topic : What is Dynamic Programming Strategy ?

6.1.1 Dynamic Programming Strategy

- Dynamic Programming is a powerful technique that allows one to solve many different types of problems in time O(n2) or O(n3) for which a naive approach would take exponential time.
- It is used in optimization. Dynamic Programming solves problems by combining the solutions of different sub problem. Moreover, Dynamic Programming algorithm it is technique of reuse solution of problem which already solved.

Syllabus Topic : Properties of Dynamic Programming Strategy

6.1.2 Properties of Dynamic Programming Strategy

Overlapping sub-problems and optimal substructure are the two properties of problem tell us that given problem can be solved using dynamic programming.

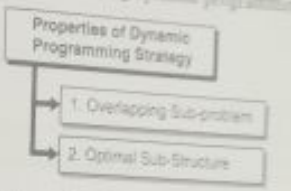


Fig. C6.1 : Properties of dynamic programming strategy

➜ **1. Overlapping Sub-problem**

Overlapping sub-problem means combines solutions to sub-problems For example, Binary Search does not have overlapping sub-problem. Whereas recursive program of Fibonacci numbers have many overlapping sub-problems.

➜ **2. Optimal Sub-Structure**

Optimal Sub-Structure means, if the optimal solution of the given problem can be obtained using optimal solutions of its sub-problems.

6.1.3 Ways to Solve Problem using dynamic Programming

There are two ways to solve problem using dynamic programming.

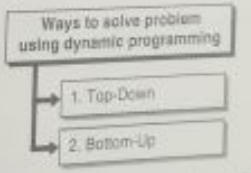


Fig. C6.2 : Ways to solve problem using dynamic programming

➜ **1. Top-Down**

Start solving the given problem by breaking it down. If you see that the problem has been solved already, then just return the saved answer. If it has not been solved, solve it and save the answer. This is usually easy to think of and very intuitive. This is referred to as **Memoization**.

2. Bottom-Up

- Analyze the problem and see the order in which the sub-problems are solved and start solving from the trivial subproblem, up towards the given problem.
- In this process, it is guaranteed that the subproblems are solved before solving the problem. This is referred to as **Dynamic Programming**.
- For example If a node **B** lies in the shortest path from a source node **A** to destination node **C**, then the shortest path from **A** to **C** is the combination of the shortest path from **A** to **B** , and the shortest path from **B** to **C**.

Syllabus Topic : Problems which can be solved using Dynamic Programming

6.2 Problems to Solved using Dynamic Programming

- Longest Common Subsequence
- Knapsack
- Matrix-chain multiplication.
- Bellman-Pond algorithm
- Floyd's All-Pairs shortest path algorithm
- Chain matrix multiplication
- Subset Sum
- Travelling salesman problem, and many more.

Syllabus Topic : Dynamic Programming Approaches

6.3 Dynamic Programming Approach

Dynamic Programming algorithm is designed using the following four steps –

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution, typically in a bottom-up fashion.
- Construct an optimal solution from the computed information.

Syllabus Topic : Understanding Dynamic Programming

6.4 Understanding Dynamic Programming

To understand in better way let consider following Fibonacci series example.

the Fibonacci sequence of integers: 1, 1, 2, 3, 5, 8, 13, 21, 33, 54,...

following is recurrence relation for the same.

$$F(n) = \begin{cases} 1 & n = 0 \\ 1 & n = 1 \\ F(n-1) + (n-2) & n > 1 \end{cases}$$

Program

```
def fibonacci(n):
if(n <=1):
return n
else:
return(fibonacci(n-1) + fibonacci(n-2))
n =int(input("Enter number of terms:"))
print("Fibonacci sequence:")
for i in range(n):
printfibonacci(i)
```

Solving above recurrence give:

$$T(n) = T(n-1) + T(n-2) + \frac{1}{2} = (1+\sqrt{5})^n = 2^n = O(2^n)$$

- Recall definition of Fibonacci numbers:

$$f(0) = 0$$
$$f(1) = 1$$
$$f(n) = f(n-1) + f(n-2)$$
$$Fib(4) = Fib(3) + Fib(2)$$
$$= (Fib(2) + Fib(1)) + Fib(2)$$
$$= ((Fib(1) + Fib(0)) + Fib(1)) + Fib(2)$$
$$= ((Fib(1) + Fib(0)) + Fib(1)) + (Fib(1) + Fib(0))$$

- Here, call to Fib(1) and Fib(0) is made multiple times.In the case of Fib(100) these calls would be count for million times.
- Hence there is lots of wastage of resources (CPU cycles and Memory for storing information on stack).

- In dynamic Programming all the subproblems are solved even those which are not needed, but in recursion only required subproblem are solved. So solution by dynamic programming should be properly framed to remove this ill-effect.
- For example, in combinatorics, C(n.m) = C(n-1,m) + C(n-1,m-1).

```
          1
        1   1
      1   2   1
    1   3   3   1
  1   4   6   4   1
1   5  10  10   5   1
```

- In simple solution, one would have to construct the whole Pascal triangle to calculate C(5,4) but recursion could save a lot of time.
- Dynamic programming and recursion work in almost similar way in the case of non overlapping subproblem. In such problem other approaches could be used like "divide and conquer".

Syllabus Topic : Examples of Dynamic Programming Algorithms

6.5 Examples of Dynamic Programming Algorithms

6.5.1 Factorial Problem

- n! is the product of all integers between 11 and 1. The definition of recursive factorial can be give n as:

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

$1! = 1$

$0! = 1$

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 - 1)!.

In the recursive case, when n is greater than 1, the function call itself to find the value of (n - 1)! and multiplies that with n. In the base case, when n is 0 or 1, the function simply returns 1.

Program

```
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: $T(n) = n \times T(n - 1) = O(n)$
Time Complexity: O(n). Space Complexity: O(n), recursive calls need a stack of size 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(rnax(m, n))
- Recursive calls need a stack of size equal to the maximum of m and n.

Improving

Now let us see how Dynamic problem reduces the complexity. From the above it can be seen that **fact(n)** is calculated from fact(n - 1) and n and nothing else. Rather than 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:

Program

```
factTable = n
def factorial(n):
try: return factTablefnl
exceptKeyError:
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.

6.5.2 Knapsack Problem

- In 0-1 Knapsack, items cannot be broken which means the thief should take the item as a whole or should leave it. This is reason behind calling it as 0-1 Knapsack.
- Hence, in case of 0-1 Knapsack, the value of xi can be either 0or 1, where other constraints remain the same.
- 0-1 Knapsack cannot be solved by Greedy approach. Greedy approach does not ensure an optimal solution. In many instances, Greedy approach may give an optimal solution.

The following examples will establish our statement.

Example 1

- Let us consider that the capacity of the knapsack is W = 25 and the items are as shown in the following table.

Item	A	B	C	D
Profit	24	18	18	10
Weight	24	10	10	7

- Without considering the profit per unit weight (pi/wi), if we apply Greedy approach to solve this problem, first item A will be selected as it will contribute maximum profit among all the elements.
- After selecting item A, no more item will be selected. Hence, for this given set of items total profit is 24. Whereas, the optimal solution can be achieved by selecting items, B and C, where the total profit is 18 + 18 = 36.

Example 2

- Instead of selecting the items based on the overall benefit, in this example the items are selected based on ratio pi/wi. Let us consider that the capacity of the knapsack is W = 30 and the items are as shown in the following table.

Item	A	B	C
Price	100	280	120
Weight	10	40	20
Ratio	10	7	6

- Using the Greedy approach, first item A is selected. Then, the next item B is chosen. Hence, the total profit is 100 + 280 = 380. However, the optimal solution of this instance can be achieved by selecting items, B and C, where the total profit is 280 + 120 = 400.
- Hence, it can be concluded that Greedy approach may not give an optimal solution.
- To solve 0-1 Knapsack, Dynamic Programming approach is required.

Problem Statement

A thief is robbing a store and can carry a maximal weight of Winto his knapsack. There are n items and weight of i^{th} item is w_i and the profit of selecting this item is p_i. What items should the thief take?

Dynamic-Programming Approach

- Let *i* be the highest-numbered item in an optimal solution S for W dollars. Then $S = S - \{i\}$ is an optimal solution for $W - w_i$ dollars and the value to the solution S is V_i plus the value of the sub-problem.
- We can express this fact in the following formula: define c[i, w] to be the solution for items 1,2, … , i and the maximum weight w.

The algorithm takes the following inputs

- The maximum weight W
- The number of items n
- The two sequences $v = <v_1, v_2, \ldots, v_n>$ and $w = <w_1, w_2, \ldots, w_n>$

```
Dynamic-0-1-knapsack (v, w, n, W)
for w = 0 to W do
c[0, w] = 0
for i = 1 to n do
c[i, 0] = 0
for w = 1 to W do
if w_i ≤ w then
if v_i + c[i-1, w-w_i] then
c[i, w] = v_i + c[i-1, w-w_i]
else c[i, w] = c[i-1, w]
else
c[i, w] = c[i-1, w]
```

- The set of items to take can be deduced from the table, starting at **c[n, w]** and tracing backwards where the optimal values came from.
- If c[i, w] = c[i-1, w], then item i is not part of the solution, and we continue tracing with **c[i-1, w]**. Otherwise, item i is part of the solution, and we continue tracing with **c[i-1, w-W]**.

Analysis

This algorithm takes $\theta(n, w)$ times as table c has $(n + 1).(w + 1)$ entries, where each entry requires $\theta(1)$ time to compute.

Syllabus Topic : Longest Common Subsequence Problems

6.6 Longest Common Subsequence Problems

The longest common subsequence problem is finding the longest sequence which exists in both the given strings.

Subsequence

Let us consider a sequence $S = <s_1, s_2, s_3, s_4, \ldots, s_n>$.

A sequence $Z = <z_1, z_2, z_3, z_4, \ldots, z_m>$ over S is called a subsequence of S, if and only if it can be derived from S deletion of some elements.

Common Subsequence

Suppose, X and Y are two sequences over a finite set of elements. We can say that Z is a common subsequence of X and Y, if Z is a subsequence of both X and Y.

Longest Common Subsequence

- If a set of sequences are given, the longest common subsequence problem is to find a common subsequence of all the sequences that is of maximal length.
- The longest common subsequence problem is a classic computer science problem, the basis of data comparison programs such as the diff-utility, and has applications in bioinformatics. It is also widely used by revision control systems, such as SVN and Git, for reconciling multiple changes made to a revision-controlled collection of files.

6.6.1 Naïve Method

- Let X be a sequence of length m and Y a sequence of length n. Check for every subsequence of X whether it is a subsequence of Y, and return the longest common subsequence found.
- There are 2m subsequences of X. Testing sequences whether or not it is a subsequence of Y takes O(n) time. Thus, the naive algorithm would take O(n2m) time.

6.6.2 Dynamic Programming

- Let $X = < x_1, x_2, x_3, \ldots, x_m >$ and $Y = < y_1, y_2, y_3, \ldots, y_n >$ be the sequences. To compute the length of an element the following algorithm is used.

- In this procedure, table **C[m,n]** is computed in row major order and another table **B[m,n]** is computed to construct optimal solution.

Algorithm : LCS-Length-Table-Formulation (X, Y)

```
m := length(X)
n := length(Y)
for i = 1 to m do
C[i, 0] := 0
for j = 1 to n do
C[0, j] := 0
for i = 1 to m do
for j = 1 to n do
if x_i = y_j
C[i, j] := C[i - 1, j - 1] + 1
B[i, j] := 'D'
else
if C[i -1, j] ≥ C[i, j -1]
C[i, j] := C[i - 1, j] + 1
B[i, j] := 'U'
else
C[i, j] := C[i, j - 1] + 1
B[i, j] := 'L'
return C and B
Algorithm: Print-LCS (B, X, i, j)
if i = 0 and j = 0
return
if B[i, j] = 'D'
  Print-LCS(B, X, i-1, j-1)
Print(x_i)
else if B[i, j] = 'U'
  Print-LCS(B, X, i-1, j)
else
  Print-LCS(B, X, i, j-1)
```

This algorithm will print the longest common subsequence of **X** and **Y**.

6.6.3 Analysis

To populate the table, the outer **for** loop iterates ***m*** times and the inner **for** loop iterates ***n*** times. Hence, the complexity of the algorithm is $O(m, n)$, where ***m*** and ***n*** are the length of two strings.

Example

- In this example, we have two strings $X = BACDB$ and $Y = BDCB$ to find the longest common subsequence.
- Following the algorithm LCS-Length-Table-Formulation (as stated above), we have calculated table C (shown on the left hand side) and table B (shown on the right hand side).
- In table B, instead of 'D', 'L' and 'U', we are using the diagonal arrow, left arrow and up arrow, respectively. After generating table B, the LCS is determined by function LCS-Print. The result is BCB.

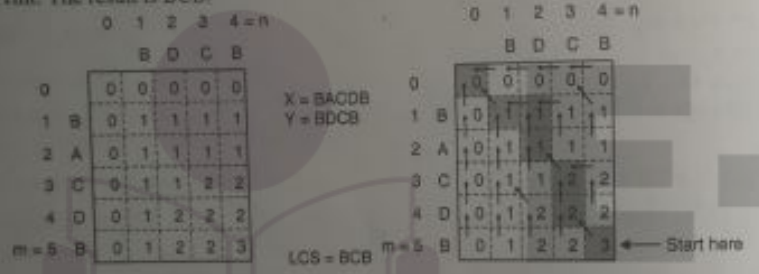


Fig. 6.6.1

- The Knapsack Problem Imagine you have a homework assignment with different parts labelled A through G.
- Each part has a "value" (in points) and a "size" (time in hours to complete). For example, say the values and times for our assignment are: A B C D E F G value 7 9 5 12 14 6 12 time 3 4 2 6 7 3 5 Say you have a total of 15 hours: which parts should you do?
- If there was partial credit that was proportional to the amount of work done (e.g., one hour spent on problem C earns you 2.5 points) then the best approach is to work on problems in order of points/hour (a greedy strategy).
- But, what if there is no partial credit? In that case, which parts should you do, and what is the best total value possible?2 The above is an instance of the knapsack problem, formally defined as follows:

  In the knapsack problem we are given a set of n items, where each item i is specified by a size is and a value vi . We are also given a size bound S (the size of our knapsack).
- The goal is to find the subset of items of maximum total value such that sum of their sizes is at most S (they all fit into the knapsack).

Review Questions

Q. 1 Explain the properties of dynamic programming strategy. **(Refer section 6.1.2)**

Q. 2 Write factorial problem. **(Refer section 6.5.1)**

Q. 3 Explain Knapsack problem. **(Refer section 6.5.2)**

❑❑❑