Module 1

Module 2

➜ **(1) Input**

Algorithm may take zero or more input arguments. Depending on the problem, the input may be a scalar, vector, array, tree, graph or some other data structures.

➜ **(2) Output**

Algorithm reads input, processes it and produces at least one output. Depending on the problem being solved, the output may of the form scalar, vector, array, tree, graph or some other data structures.

➜ **(3) Definiteness**

All instructions in algorithm should be unambiguous and simple to interpret. There should not be multiple ways to interpret the same instruction. Instructions should be precise and concise.

➜ **(4) Finiteness**

Every algorithm must terminate after a finite number of steps. If algorithm contains a loop, the upper bound of the loop must be finite. Recursive calls should have a well-defined base case to terminate the algorithm.

➜ **(5) Effectiveness**

The algorithm should be written with a basic set of instructions. The operations should be basic enough to perform exactly using basic set, just like one can perform them with pen and paper. Complex operations should be performed using a combination of basic instruction. For example, multiplication should be performed using loop and addition, sorting should be carried out using looping, comparison, swapping etc.

1.1.3 How to write an Algorithm ?

Q. Discuss the rules to write an algorithm. **(5 Marks)**

Q. What are the general rules followed while writing the algorithm? **(5 Marks)**

- The algorithm basically consists of two parts : Head and body.
- Head part consists of algorithm name, description of the problem being solved, input to the problem and expected output. It may also include the explanation of input argument and output variable. The description provides clear idea to the user about the functionality of the algorithm.
- Body part includes a logical sequence of statements involving various constructs like conditional statements, expressions, loops, breaks, function calls etc.

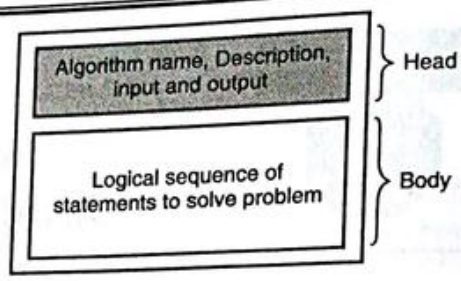


Fig. 1.1.2 : Structure of algorithm

An algorithm is a lucid form of program and it does not have rigid restrictions of syntax. One can write an algorithm using his own terminology.

However, some of the common rules followed for writing algorithms, which are stated below :

1. The algorithm should start with the keyword **Algorithm**, followed by name of algorithm, followed by a list of arguments to be passed to the algorithm.
   Algorithm FIND_SUM(A, B)
2. Comments start with // sign. In very next line, we should specify the description and input-output of the algorithm.
   // Problem Description : Add two integer numbers.
   // Input : Two numbers A and B on which sum is to be performed.
   // Output : Sum of given two numbers.
3. Next, comes body part, which contains various logical statements in proper sequence. These statements may contain control statements, loops, expressions etc.
4. Compound statements are enclosed within the opening and closing curly brace i.e. { … }.
5. Use left arrow for assignment : C ← A + B.
6. Array index usually starts with index 0 or 1.
7. Relational operations are performed using relational operators like <, >, ==, ≠, ≥ and ≤
8. Logical operations are performed using logical operators like **and** (∧), **or** (∨) and ≤ **not** (¬).
9. Input and output are performed using read and write statement.
   read (A) / **read** "A"
   write (A) / **write** "A" or **print** (A) / **print** "A"
10. Control statements are written as follows :

```
if (condition) then Statement end
if (condition) then Statement else Statement end
```

    Multiple statements are enclosed within { … }
11. While loop is written as follows :

```
while (condition) do
{
  Do some work
}
```

    Sometimes curly braces are omitted and block is closed with **end** keyword.

```
while (condition) do
  Do some work
end
```

12. For loop is written as follows :

```
for index ← FirstIndex to LastIndex do
{
  Do some work
}
```

Sometimes curly braces are omitted and block is closed with **end** keyword.

```
for index ← FirstIndex to LastIndex do
  Do some work
end
```

☞ **Examples of how to write algorithms**

Ex. 1.1.1

Write an algorithm for finding the factorial of number n.

Soln. :

```
Algorithm FACTORIAL(n)
// Description : Find factorial of given number
// Input : Number n whose factorial is to be computed
// Output : Factorial of n = n × (n – 1) × ... × 2 × 1

if (n == 1) then
    return 1
else
    return n * FACTORIAL(n – 1)
end
```

Ex. 1.1.2

Write an algorithm to perform matrix multiplication.

Soln. :

```
Algorithm MATRIX_MULTIPLICATION(A, B)
// Description : Perform matrix multiplication of two matrices.
// Input : Two matrices A and B of size n × n.
// Output : Resultant matrix containing multiplication of
            A and B

for i ← 1 to n do
    for j ← 1 to n do
    C[i][j] ← 0
    for k ← 1 to n do
    C[i][j] ← C[i][j] + A[i][k] * B[k][j]
    end
    end
end
```

– However, there is no strict rule to follow these standards. Algorithm syntax may vary from person to person.

Syllabus Topic : Performance Analysis

1.2 Performance Analysis

Syllabus Topic : Space and Time Complexity

1.2.1 Space Complexity

➔ **(May 13, Dec. 13)**

Q. Explain space complexity in detail.
MU - May 2013, Dec. 2013, 3 Marks

Q. What are the basic components which contribute to space complexity? **(4 Marks)**

Q. What do you mean by space complexity of an algorithm? How do we measure the space complexity of an algorithm? Explain with suitable example. **(4 Marks)**

– Space complexity is very important notion of efficiency analysis.

Definition

Problem-solving using computer requires memory to hold temporary data or final result while the program is in execution. The amount of memory required by the algorithm to solve given problem is called **space complexity** of the algorithm.

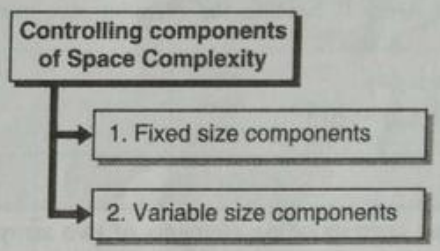


Fig. C1.2 : Controlling components of Space complexity of algorithm

The space complexity of the algorithm is controlled by two components:

➔ **(1) Fixed size components**

It includes the programming part whose memory requirement does not alter on program execution. For example,

- Instructions
- Variables
- Constants
- Arrays

➔ **(2) Variable size components**

It includes the part of the program which whose size depends on the problem being solved. For example,

- Size of loop

- Stack required to handle recursive call
- Dynamic data structures like linked list

– We use the notation S(n) to specify the space complexity of the problem for input size n. The term n is treated as the size of input or the problem size.

– The notion of space complexity is explained with following examples:

Example 1 : Addition of two scalar variables

```
Algorithm ADD_SCALAR(A, B)
// Description : Perform arithmetic addition of two numbers
// Input : Two scalar variables A and B
// Output : variable C, which holds the addition of A and B

C ← A + B
return C
```

The addition of two scalar numbers requires one extra memory location to hold the result. Thus the space complexity of this algorithm is constant, hence S(n) = O(1).

Example 2 : Addition of two arrays

```
Algorithm ADD_ARRAY(A, B)
// Description : Performa element-wise arithmetic addition of
                 two arrays
// Input : Two number arrays A and B
// Output : Array C holding the element-wise sum of array
            A and B
for i ← 1 to n do
    C[i] ← A[i] + B[i]
end
return C
```

– Adding corresponding elements of two arrays, each of size *n* requires extra *n* memory locations to hold the result. As input size *n* increases, required space to hold the result also grows in the linear order of input. Thus the space complexity of above code segment would be S(n) = O(n).

Example 3 : Sum of elements of array

```
Algorithm SUM_ARRAY_ELEMENTS(A)
// Description : Add all elements of array A
// Input : Array A of size n
// Output : Variable Sum which holds the addition of array
            elements.

Sum ← 0
for i ← 1 to n do
    Sum ← Sum + A[i]
end
return Sum
```

– The addition of all array elements requires only one extra variable (i.e. one memory location) denoted as *sum*, this is independent of array size. Thus the space complexity of the algorithm is constant and S(n) = O(1).

1.2.2 Time Complexity

→ **(May 13, Dec. 13)**

Q. Explain time complexity in detail.
MU - May 2013, Dec. 2013, 3 Marks

Q. How do we analyze and measure the time complexity of an algorithm? **(4 Marks)**

Q. What do you mean by time complexity of an algorithm? How do we measure the time complexity of an algorithm? Explain with suitable example. **(4 Marks)**

– Goodness of algorithm is often determined by the time complexity. Time complexity is the most fundamental component of analysis framework.

Definition

The valid algorithm takes a finite amount of time for execution. The time required by the algorithm to solve given problem is called **time complexity** of the algorithm. Time complexity is very useful measure in algorithm analysis.

– Time complexity is not measured in physical clock ticks, rather it is a function of a number of primitive operations. *Primitive operation* is the most frequent operation in algorithm

– We use the notation T(n) to symbolize the time complexity of the problem for input size *n*.

– The notion of time complexity is explained with following examples:

Example 1 : Addition of two scalar variables

```
Algorithm ADD_SCALAR(A, B)
// Description : Perform arithmetic addition of two numbers
// Input : Two scalar variables A and B
// Output : variable C, which holds the addition of A and B

C ← A + B
return C
```

The sum of two scalar numbers requires one addition operation. Thus the time complexity of this algorithm is constant, so T(n) = O(1).

Example 2 : Perform addition of two arrays

```
Algorithm ADD_ARRAY(A, B)
// Description : Performa element-wise arithmetic addition of
    two arrays
// Input : Two number arrays A and B of length n
// Output : Array C holding the element-wise sum of array A
            and B
```

```
for i ← 1 to n do // one initialization, n increment, n comparison
    C[i] ← A[i] + B[i]      // n addition and n assignment
end
return C
```

As it can be observed from above code, addition array elements required iterating loop n times. Variable i is initialized once, the relation between control variable i and n are checked n times, and i is incremented n times. With the loop, addition and assignment operations are performed n time.

Thus the total time of algorithm is measured as,

$$T(n) = 1(\text{initizlization}) + n(\text{copmparison} + \text{increment addition} + \text{assignment}) = 1 + 4n$$

While doing efficiency analysis of the algorithm, we are interested in the order of complexity in term of input size n. What is the relationship between input size n and the number of steps to be performed? So all multiplicative and divisive constants should be dropped. Thus, for given algorithm $T(n) = O(n)$

Example 3 : Sum of elements of array

```
Algorithm SUM_ARRAY_ELEMENTS(A)
// Description : Add all elements of array A
// Input : Array A of size n
// Output : Variable Sum which holds the addition of array elements
Sum ← 0
for i ← 1 to n do
    Sum ← Sum + A[i]
end
```

- The addition of all array elements requires *n* additions (we shall omit the number of comparisons, assignment, initialization etc. to avoid the multiplicative or additive constants). A number of additions depend on the size of the array. It grows in the linear order of input size. Thus the time complexity of above code is $T(n) = O(n)$.
- Time and space complexity of discussed problem is compared in the Table 1.2.1

Table 1.2.1 : Comparison of space and time complexity for various problems

Problem	Space Complexity S(n)	Time Complexity T(n)
Add scalar	O(1)	O(1)
Add two arrays	O(n)	O(n)
Add array elements	O(1)	O(n)

Syllabus Topic : Growth of Function
Big Oh, Omega, Theta Notation

1.2.3 Growth of Function

Q. Define order of growth. List various efficiency classes with example. Show the relationship between efficiency classes. **(6 Marks)**

Definition

The efficiency of the algorithm is expressed in term of input size *n*. The relationship between input size and performance of the algorithm is called **order of growth**.

- Order of growth indicates how quickly the time required by algorithm grows with respect to input size.
- For input size *n*, the algorithm may execute a number of steps in order of log n, n, n^2, n^3 or something else.
- Efficiency classes are categorized into different classes as shown in Table 1.2.2.

Table 1.2.2 : Various efficiency classes

Efficiency class	Order of growth rate	Example
Constant	1	Delete the first node from linked list Remove the largest element from max heap Add two numbers
Logarithmic	log n	Binary search Insert / delete element from binary search tree
Linear	n	Linear search Insert node at the end of linked list Find minimum / maximum element from array
n log n	n log n	Merge sort Binary search Quicksort Heap Sort
Quadratic	n^2	Selection Sort Bubble sort Input 2D array Find maximum element form 2D matrix
Cubic	n^3	Matrix Multiplication
Exponential	2^n	Finding power set of any set Find optimal solution for Knapsack problem Solve TSP using dynamic programming

Efficiency class	Order of growth rate	Example
Factorial	n!	Generating permutations of given set Solve TSP problem using brute force approach

- Efficiency classes are sorted as :
  $O(1) < O(\log^k n) < O(\log (\log n)) < O(\log n) < O((\log n)^k) < O(n) < O(n \log^k n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$.
- These are the widely used classes, there exist many other classes which represent intermediate growth order. Algorithms with running time towards the beginning of the list are considered better. Algorithms having exponential or factorial running time are unacceptable for practical use.
- Effect of input size on growth rate is shown in Table 1.2.3.

Table 1.2.3 : Growth of function

Input Size	Constant (1)	Log (log)	Linear (n)	Quadratic (n^2)	Cubic (n^3)	Exponent (2^n)
1	1	0	1	1	1	2
2	1	1	2	4	8	4
4	1	2	4	16	64	16
8	1	3	8	64	512	256
16	1	4	16	256	4096	65536
32	1	5	32	1024	32768	4294967296

- The relationship between various efficiency classes is shown in Fig. 1.2.1. The growth of log function is very slow compared to linear of a quadratic function. Similar observations can be made for different efficiency classes.
- 2^n, n! or n^n functions grow very quickly. Even for a small problem size (i.e. for small n), such problems require lots of resources.
- Graphical comparison between different efficiency classes is depicted in the Fig. 1.2.1.

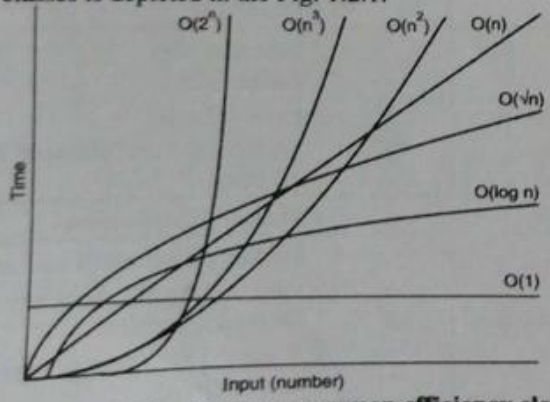


Fig. 1.2.1 : Relation between common efficiency classes

1.2.4 Asymptotic Notation

➔ **(May 13, Dec. 13, May 14, Dec. 15, May 16, Dec. 16)**

Q. Explain Asymptotic notations.
MU - May 2013, Dec. 2013, 5 Marks, May 2016, Dec. 2016, 4 Marks

Q. Explain O, Ω, and θ notations with the help of graph.
MU - May 2014, 3 Marks

Q. Define O, Ω, and θ notations.
MU - Dec. 2015, 3 Marks

Definition

Asymptotic notations are a mathematical tool to find time or space complexity of an algorithm without implementing it in a programming language. This measure is independent of machine-specific constants.
It is a way of describing a major component of the cost of the entire algorithm.

- Machine specific constants involve hardware architecture of the machine, RAM, supported virtual memory, the speed of the processor, available instruction set (RISC or CISC) etc.
- The asymptotic notation does the analysis of algorithm independent of all such parameters.
- Finding minimum element from an array of size *n* takes maximum *n* comparisons.
- The asymptotic complexity of this algorithm is linear (in order of *n*). This linear behavior is the main term in the complexity formula. It says if we double the size of the array, then the numbers of comparisons are roughly doubled.

Definition

The **primitive operation** is the most frequent operation appearing in the algorithm and it depends on the problem. The primitive operation is the major component of cost.

- As discussed earlier, for sorting and searching problems, the primitive operation is a comparison. For adding arrays or matrices, the primitive operation is an addition. For the factorial problem, the primitive operation is multiplication.
- Order of growth of functions is very crucial in performance evaluation of algorithm. Suppose running time of two algorithms A and B are f(n) and g(n), where,
  $f(n) = 2n^2 + 5$
  $g(n) = 10n$
- Here, *n* is the size of problem and polynomials f(n) and g(n) are a number of primitive operations performed by algorithm A and B respectively. For different value of *n*, we have Table 1.2.4.

Table 1.2.4 : Steps comparison of function f(n) and g(n) for different input size

n	1	2	3	4	5	6	7
f(n)	7	13	23	37	55	77	103
g(n)	10	20	30	40	50	60	70

- For small input size, algorithm A may outperform B, but as input size becomes sufficiently large (in this case n = 5), f(n) always runs slower (perform more steps) than g(n). So it's very important to understand growth rate of functions.
- Asymptotic notation describes limiting behaviour of the function. For example, if function $f(n) = 8n^2 + 4n - 32$ then as $n \to \infty$, the term $4n - 32$, becomes insignificant. So the growth of f(n) is limited by the n^2 term.

Following assumptions are made while doing complexity analysis.

☞ **Assumptions**

- The actual cost of operation is not considered.
- Abstract cost c is ignored : $O(c.\ n^2)$ reduces to $O(n^2)$
- Only leading term of polynomial is considered : $O(n^3 + n)$ reduces to $O(n^3)$
- Drop multiplicative or divisive constant if any : $O(2n^2)$ and $O(n^2/2)$ both reduces to $O(n^2)$.
- Various notations like Big Oh (O), Big Omega (Ω), Big Theta (Θ), Little Oh (o), Little Omega (ω) are used to describe the asymptotic running time of the algorithm.

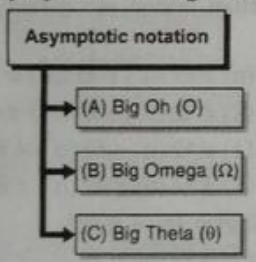


Fig. C1.3 : Different asymptotic notations

→ 1.2.4(A) Big Oh

Q. Define and explain big Oh notations. Give examples. (2 Marks)

- This notation is denoted by 'O', and it is pronounced as **"Big Oh"**. Big Oh notation defines **upper bound** for the algorithm, it means the running time of algorithm cannot be more than its asymptotic upper bound for any random sequence of data.

Definition

Let f(n) and g(n) are two nonnegative functions indicating running time of two algorithms. We say, g(n) is upper bound of f(n) if there exist some positive constants c and n_0 such that $0 \le f(n) \le c \cdot g(n)$ for all $n \ge n_0$. It is denoted as $f(n) = O(g(n))$.

- In Fig. 1.2.2, Horizontal axis represents problem size and the vertical axis represents growth order (steps required to solve the problem of size n) of functions.

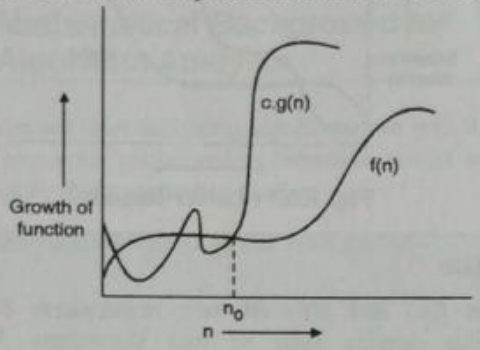


Fig. 1.2.2 : Upper bound

Definition

For small input size, there may be many crossovers between the growth rate of f(n) and c.g(n), but once n becomes significantly large, f(n) grows always slower than c.g(n). This value of n is called **crossover point** and is denoted as n_0.

☞ **Loose bounds**

All the set of functions with growth rate *higher* than its actual bound are called loose upper bound of that function,

$23 = O(n) = O(n^2) = O(n^3) = O(n!)$
$6n + 3 = O(n^2) = O(n^3) = O(n!)$
$3n^2 + 2n + 4 = O(n^3) = O(n!)$
$2n^3 + 4n + 5 = O(2^n) = O(n!)$

☞ **Incorrect bounds**

All the set of functions with a growth rate *lower* than its actual bound are called incorrect bound of that function.

$6n + 3 \neq O(1)$
$3n^2 + 2n + 4 \neq O(n) \neq O(1)$
$2n^3 + 4n + 5 \neq O(n^2) \neq O(n) \neq O(1)$

For fucntion $f(n) = 2n^2 + n + 3$

$f(n) = O(n^2) = O(n^3) = O(2^n) = O(n!) = O(n^n)$
$f(n) \neq O(n\log n) \neq O(n) \neq O(\log n) \neq O(1)$

→ 1.2.4(B) Big Omega

Q. Define and explain Ω notations. Give examples. (2 Marks)

This notation is denoted by 'Ω', and it is pronounced as **"Big Omega"**. Big Omega notation defines **lower bound** for the algorithm. It means the running time of algorithm cannot be less than its asymptotic lower bound for any random sequence of data.

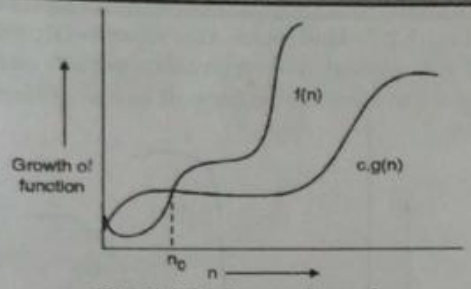


Fig. 1.2.3 : Lower bound

Definition

Let f(n) and g(n) are two nonnegative functions indicating running time of two algorithms. We say the function g(n) is lower bound of function f(n) if there exist some positive constants c and n_0 such that $0 \le c \cdot g(n) \le f(n)$ for all $n \ge n_0$. It is denoted as $f(n) = \Omega(g(n))$.

☞ **Loose bounds**

All the set of functions with growth rate *slower* than its actual bound are called loose lower bound of that function,

$6n + 3 = \Omega(1)$

$3n^2 + 2n + 4 = \Omega(n) = \Omega(1)$

$2n^3 + 4n + 5 = \Omega(n^2) = \Omega(n) = \Omega(1)$

☞ **Incorrect bounds**

All the set of functions with growth rate lower than its actual bound are called incorrect bound of that function.

$23 \neq \Omega(n) \neq \Omega(n^2) \neq \Omega(n^3) \neq \Omega(n!)$

$6n + 3 \neq \Omega(n^2) \neq \Omega(n^3) \neq \Omega(n!)$

$3n^2 + 2n + 4 \neq \Omega(n^3) \neq \Omega(n!)$

$2n^3 + 4n + 5 \neq \Omega(2^n) \neq \Omega(n!)$

For fucntion $f(n) = 2n^2 + n + 3$

$f(n) = \Omega(n^2) = \Omega(n \log n) = \Omega(n)$

$= \Omega(\log n) = \Omega(1)$

$f(n) \neq \Omega(n^3) \neq \Omega(2^n) \neq \Omega(n!) \neq \Omega(n^n)$

→ 1.2.4(C) Big Theta

Q. Define and explain Θ notations. Give examples. **(2 Marks)**

– This notation is denoted by 'Θ', and it is pronounced as **"Big Theta"**. Big Theta notation defines **tight bound** for the algorithm. It means the running time of algorithm cannot be less than or greater than it's asymptotic tight bound for any random sequence of data.

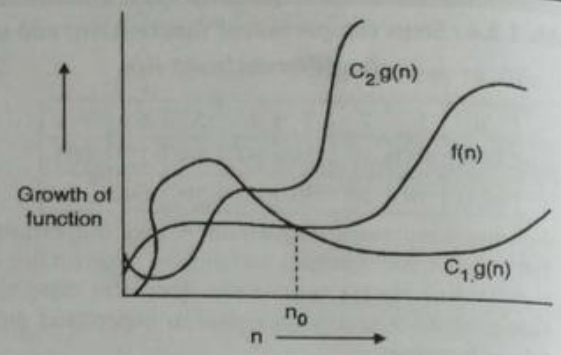


Fig. 1.2.4 : Tight bound

Definition

Let f(n) and g(n) are two nonnegative functions indicating running time of two algorithms. We say the function g(n) is tight bound of function f(n) if there exist some positive constants c_1, c_2, and n_0 such that $0 \le c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n)$ for all $n \ge n_0$. It is denoted as $f(n) = \Theta(g(n))$.

– For big oh, big omega and big theta, g(n) is not a single function but it's a set of functions which always grow quickly, slowly or at the same rate of f(n) respectively for $\forall\, n \ge n_0$.

☞ **Incorrect bounds**

All the set of functions with a growth rate *lower* or *greater* than its actual bound are called incorrect bound of that function.

$23 \neq \Theta(n) \neq \Theta(n^2) \neq \Theta(n^3) \neq \Theta(n!)$

$6n+3 \neq \Theta(1) \neq \Theta(n^2) \neq \Theta(n^3) \neq \Theta(n!)$

$3n^2 + 2n + 4 \neq \Theta(1) \neq \Theta(n) \neq \Theta(n^3) \neq \Theta(n!)$

$2n^3 + 4n + 5 \neq \Theta(1) \neq \Theta(n) \neq \Theta(n^2) \neq \Theta(2^n) \neq \Theta(n!)$

For fucntion $f(n) = 2n^2 + n + 3$

$f(n) \neq \Theta(1) \neq \Theta(n)$

$\neq \Theta(n^3) \neq \Theta(2^n) \neq \Theta(n!) \neq \Theta(n^n)$

Loose bound does not exist for tight bound.

Ex. 1.2.1 **MU - May 2014, 7 Marks**

Represent the following function using Big oh, Omega and Theta Notations.

(i) $(n) = 3n + 2$ (ii) $T(n) = 10n^2 + 2n + 1$

Soln. :

(A) Big oh (upper bound)

(i) $T(n) = 3n + 2$

To find upper bound of f(n), we have to find c and n_0 such that $0 \le f(n) \le c \cdot g(n)$ for all $n \ge n_0$

$0 \le f(n) \le c \cdot g(n)$

$0 \le 3n + 2 \le c \cdot g(n)$

$0 \le 3n + 2 \le 3n + 2n$, for all $n \ge 1$ (There can be such infinite possibilities)

$0 \le 3n + 2 \le 5n$

So, $c = 5$ and $g(n) = n$, $n_0 = 1$

(ii) $T(n) = 10n^2 + 2n + 1$

To find upper bound of f(n), we have to find c and n_0 such that $0 \le f(n) \le c \cdot g(n)$ for all $n \ge n_0$

$0 \le f(n) \le c \cdot g(n)$

$0 \le 10n^2 + 2n + 1 \le c \cdot g(n)$

$0 \le 10n^2 + 2n + 1 \le 10n^2 + 2n^2 + n^2$, for all $n \ge 1$

$0 \le 10n^2 + 2n + 1 \le 13n^2$

So, $c = 13$, $g(n) = n^2$ and $n_0 = 1$

(B) Lower Bound

(i) $T(n) = 3n + 2$

To find lower bound of f(n), we have to find c and n_0 such that $0 \le c.g(n) \le f(n)$ for all $n \ge n_0$

$0 \le c \cdot g(n) \le f(n)$

$0 \le c \cdot g(n) \le 3n + 2$

$0 \le 3n \le 3n + 2 \rightarrow$ true, for all $n \ge n_0$

$f(n) = \Omega(g(n)) = \Omega(n)$ for $c = 3$, $n_0 = 1$

(ii) $T(n) = 10n^2 + 2n + 1$

To find lower bound of f(n), we have to find c and n_0 such that $0 \le c.g(n) \le f(n)$ for all $n \ge n_0$

$0 \le c \cdot g(n) \le f(n)$

$0 \le c \cdot g(n) \le 10n^2 + 2n + 1$

$0 \le 10n^2 \le 10n^2 + 2n + 1, \rightarrow$ true, for all $n \ge 1$

$f(n) = \Omega(g(n)) = \Omega(n^2)$ for $c = 3$, $n_0 = 1$

(C) Tight bound

(i) $T(n) = 3n + 2$

To find tight bound of f(n), we have to find c_1, c_2 and n_0 such that,

$0 \le c_1 \cdot g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0$

$0 \le c_1 \cdot g(n) \le 3n + 2 \le c_2 \cdot g(n)$

$0 \le 3n \le 3n + 2 \le 5n$, for all $n \ge 1$

Above inequality is true and there exists such infinite inequaities.

So, $f(n) = \Theta(g(n)) = \Theta(n)$ for $c_1 = 3$, $c_2 = 5$, $n_0 = 1$

(ii) $T(n) = 10n^2 + 2n + 1$

To find tight bound of f(n), we have to find c_1, c_2 and n_0 such that,

$0 \le c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n)$ for all $n \ge n_0$

$0 \le c_1 \cdot g(n) \le 10n^2 + 2n + 1 \le c_2 \cdot g(n)$

$0 \le 10n^2 \le 10n^2 + 2n + 1 \le 13n^2$, for all $n \ge 1$

Above inequality is true and there exists such infinite inequaities. So,

$f(n) = \Theta(g(n)) = \Theta(n^2)$ for $c_1 = 10$, $c_2 = 13$, $n_0 = 1$

Syllabus Topic : Mathematical Background for Algorithm Analysis

1.3 Mathematical Background for Algorithm Analysis

Before we start analyzing algorithm, we will first look at some important mathematical formulas, which will help us to simplify the computation further.

Mathematics to simplify the summation

$$\sum_{i=1}^{n} 1 = 1 + 1 + 1 + \ldots + 1 = n = O(n)$$

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \ldots + n = \sum n = \frac{n(n+1)}{2}$$

$$= \frac{n^2 + n}{2} = O(n^2)$$

$$\sum_{i=1}^{n} i^k = 1 + 2^k + 3^k + \ldots + n^k = \frac{n^k}{k+1} = O(n^{k+1})$$

$$\sum_{i=1}^{n} k^i = k + k^2 + k^3 + \ldots + k^n = \frac{k^{n+1}}{k-1} = O(k^n)$$

$$\sum_{i=k}^{n} 1 = n - k + 1$$

1.3.1 Framework for Analysis of Non-Recursive Algorithms

Q. Discuss the general plan for analyzing efficiency of non-recursive algorithm. **(6 Marks)**

Q. Explain the framework of efficiency analysis of non-recursive algorithms with suitable examples. **(6 Marks)**

- Finding complexity of the non-recursive algorithm is simpler than that of recursive algorithms. A number of primitive operations define the complexity of the algorithm. By following below steps we can find the complexity of non-recursive algorithms :
  - Determine **size of problem** / input
  - Find out **primitive** / elementary operation
  - Find **count of primitive operations** for best, worst or average case.
  - Simplify the summation by **dropping multiplicative and divisive constants** of highest degree polynomial term in sum.

Ex. 1.3.1

Determine the complexity to find the sum of elements of the array.

Soln. :

```
Algorithm SUM_ARRAY(A, n)
// Description : Find the sum of elements of array
// Input : Array A of length n
// Output : Variable Sum which holds the summation of array
            elements
```

```
Sum ← 0
for i ← 1 to n do
    Sum ← Sum + A[i]
end
print "Sum of Array Elements: ", Sum
```

Iterate over length of array A

Add each element in partial sum

Step 1 : Size of problem is *n* because length of array is *n*

Step 2 : Primitive operation is addition:
Sum = Sum + A[i]

Step 3 : For loop iterates, *n* time and hence addition is performed *n* times, so T(n) = O(n)

$$T(n) = \sum_{i=1}^{n} 1 = 1 + 1 + 1 + \ldots n \text{ times} = 0(n)$$

Ex. 1.3.2

Find complexity of bubble sort

Soln. :

```
Algorithm BUBBLE_SORT(A, n)
// Description : Sort the given numerical data
// Input : Array A of randomly placed n element
// Output : Sorted sequence of input data
for i ← 1 to n do
    for j ← 1 to n – i – 1 do
    if A[j] > A[j + 1] then
    Swap(A[i], A[j])
    end
    end
end
```

Move largest element at the end of unsorted list

Step 1 : Size of problem is n

Step 2 : Primitive operation is comparison

Step 3 : For each instance of outer loop, inner loop iterate (n – i) times.

For i=1, inner loop does n – 1 comparisons, for i=2, inner loop does n – 2 comparisons and so on,

$$T(n) = (n-1) + (n-2) + \ldots + 3 + 2 + 1$$

$$= \sum_{i=1}^{n-1} i = 1 + 2 + 3 + \ldots + (n-1)$$

$$= \sum (n-1) = \frac{(n-1) \cdot n}{2} = \frac{n^2 - n}{2} = O(n^2)$$

Ex. 1.3.3

Write an algorithm for searching an element in array of size n. Calculate complexity of this algorithm.

Soln. :

- We will discuss and derive the complexity of linear search technique to search an element from an array of size n.
- Linear search is a very simple way of searching an element from the list. Let *Key* is the element that we want to search. The *key* element is compared with one by one all index locations of A. Algorithm halts in two cases : *Key* element is found or entire array is scanned.
- Algorithm for linear search is shown below :

```
Algorithm LINEAR_SEARCH(A, Key)
// Description : Perform linear search on array A to search
                 element Key
// Input : Array of length n and Key to be searched
// Output : Success / failure message
flag ← 0
for i ←1 to n do
    if Key == A[i] then
    print "Element Found on Location", i
    flag ← 1
    break
    end
end

if flag == 0 then
    print "Element Not Found"
end
```

Set flag if Key is found

Key is found, so stop further search

Flag wont set if Key not found

Complexity analysis

Best case

The algorithm needs a minimum number of comparisons if the key element is on the first position. In the best case, the size of input array does not matter. In the best case, the algorithm does only one comparison irrespective of array length. Hence the running time of the linear search in the best case is, T(n) = O(1).

Worst case

The algorithm does a maximum number of comparisons if the key element is on the last position of the array or it is not present at all. The entire array needs to be scanned. Numbers of comparisons linearly grow with the size of the input. Hence the running time of linear search in worst case is, T(n) = O(n)

Average case

The average case occurs when an element is neither on the first location nor at last. The key element may be near to the beginning of array or it may be towards the end, or it

may be somewhere near the middle. So on an average, the algorithm does (n / 2) comparisons.

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

The time complexity of all three cases is depicted in the following table.

Best case	Average case	Worst case
O(1)	O(n)	O(n)

Ex. 1.3.4

Write an algorithm to find Max Element from an unsorted array of size n. Calculate complexity of this algorithm.

Soln. :

To find the maximum element, first Max is set to A[1]. Then Max is compared with each element of array. If A[i] > Max, algorithm updates value of Max and sets it to A[i]. The process is repeated over the length of array. The pseudo code of the process is given below :

```
Algorithm FIND_MAX(A)
// Description: Find the maximum element from given array//
    Input : Array A of length n
// Output : Variable Max holding the maximum element of
    array A
Max ← A[1]
for i ← 2 to n do
    if Max < A[i] then
    Max ← A[i]
    end
end
print "Maximum Element of Array A is", Max
```

Update Max if it is smaller than A[i]

Complexity analysis

In every iteration, algorithm does one comparison and problem size is reduced by 1. Recurrence for this problem is formulated as,

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

Let us solve this using iteration method,

$$T(n-1) = T(n-2) + 1$$
$$\Rightarrow T(n) = [T(n-2) + 1] + 1 = T(n-2) + 2$$
$$T(n-2) = T(n-3) + 1$$
$$\Rightarrow T(n) = [T(n-3) + 1] + 2 = T(n-3) + 3$$

After k iterations,

$$T(n) = T(n-k) + k$$

For $k = n$

$$T(n) = T(n-n) + n = T(0) + n$$

Cost for solving problem of size 0 is definitely 0, so $T(0) = 0$

Hence, $T(n) = O(n)$

Ex. 1.3.5

Write an algorithm to delete an element from a linked list. Also, mention the worst case running time for this operation.

Soln. :

To delete the element from the linked list, we should traverse the list to search the element to be deleted. The list will be traversed until a node with the specified element is found or end of the list is reached.

Algorithm for the given operation is described below:

```
Algorithm DELETE_NODE(HEAD, Key)
// Description: Delete node from the linked list having value
    Key
// Input: Linked list starting at HEAD and Key
// Output: Linked list after deletion of node with value Key

Temp = HEAD          // TEMP points to the first node in list

// Iterate until reach to second last node or the node before the
    node to be deleted
while Temp → Next →Next ≠ NULL && Temp → Next →
    Data ≠ Key do
    Temp = Temp → Next
end

// If node to be deleted is not the last node
if Temp → Next → Data == Key && Temp → Next → Next
    ≠ NULL then
    Hold = Temp → Next
    Temp → Next = Hold → Next
    Free(Hold)

// If node to be deleted is last node
else if Temp → Next → Data == Key && Temp → Next →
    Next == NULL then
    Hold = Temp → Next
    Temp → Next = NULL
    Free(Hold)

// If node with given key does not exist
else
    print "Node not found"
end
```

Go up to second last node

Stop on node before the node to be deleted

The problem is similar to linear search. On each iteration, problem size is reduced by 1, and one comparison is done. Thus, the recurrence for the above algorithm can be formulated as,

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

Let us solve this using iteration method,

$$T(n-1) = T(n-2) + 1$$
$$\Rightarrow T(n) = [T(n-2) + 1] + 1 = T(n-2) + 2$$
$$T(n-2) = T(n-3) + 1$$
$$\Rightarrow T(n) = [T(n-3) + 1] + 2 = T(n-3) + 3$$

After k iterations,

$$T(n) = T(n-k) + k$$

For $k = n$

$T(n) = T(n-n) + n = T(0) + n$

Cost for solving problem of size 0 is definitely 0, so $T(0) = 0$

Hence, $T(n) = O(n)$

Worst case for the problem occurs when the node to be deleted is at the end of the list or it is not present at all. For the list of n nodes, the worst case running time would be $T(n) = O(n)$.

Ex. 1.3.6

Consider the following algorithm.

```
ALGORITHM sum (n)
// Input : A non-negative integer n
S ← 0
for i = 1 to n do
S ← S + i
return S.
```

i) What does this algorithm compute ?

ii) What is its basic operation ?

iii) How many times the basic operations executed ?

iv) What is the efficiency class of this algorithm ?

v) Suggest an improved algorithm and indicate its efficiency class. If you cannot do it, try to prove that it cannot be done.

Soln. :

(i) This algorithm computes the summation of numbers from 1 to n

(ii) The basic operation is an addition, i.e. computing sum

(iii) Basic operation executes n times

(iv) As it performs n basic operations, efficiency class of this algorithm is O(n) .

(v) An improved version of the algorithm :

```
Algorithm IMPROVED_SUM(n)
// Description: Add first n numbers
// Input: Number n
// Output: Variable Sum holding the summation of first
           n numbers

Sum ← n * (n + 1) / 2
return Sum
```

The complexity of this algorithm is O(1), it performs only one computation to find the sum of first n natural numbers.

For n = 6,

Using algorithm SUM() : 1 + 2 + 3 + 4 + 5 + 6 = 21

Using Algorithm IMPROVED_SUM() :

n*(n + 1) / 2 = (6 * 7) / 2 = 21

1.3.2 Framework for Analysis of Recursive Algorithms

Q. Explain the framework of efficiency analysis of recursive algorithms with suitable examples. (5 Marks)

- By following steps, we can find the complexity of recursive algorithms:
- Determine size of problem
- Identify primitive operation
- Find count of primitive operation in each call
- Set up and solve recurrence equation using appropriate method.

Ex. 1.3.7

Write a recursive algorithm for Tower of Hanoi problem, set up its recurrence and solve it.

Soln. :

- The tower of Hanoi is very well known the recursive problem. The problem is based on 3 *pegs* (source, auxiliary and destination) and n disks. Tower of Hanoi is the problem of shifting all n disks from source peg to destination peg using auxiliary peg with following constraints :
- Only one disk can be moved at a time.
- A Larger disk cannot be placed on smaller disk.
- The initial and final configuration of the disks is shown in Fig. P. 1.3.7(a) and Fig. P. 1.3.7(b), respectively.

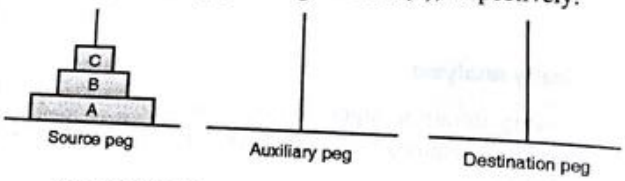


Fig. P. 1.3.7(a) : Initial state of Tower of Hanoi

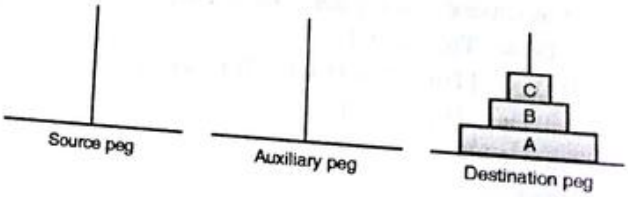
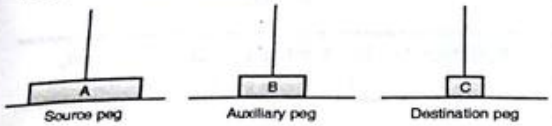


Fig. P. 1.3.7(b) : Final state of Tower of Hanoi

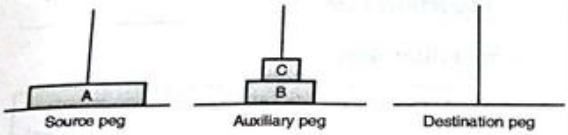
- There can be n number of disks on source peg. Let's trace the problem for n=3 disks. The trace of the solution is :

Step 1 : Move disk C from the *src* peg to *dst* peg

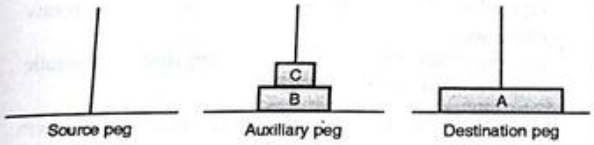
Step 2 : Move disk B from the *src* peg to *aux* peg



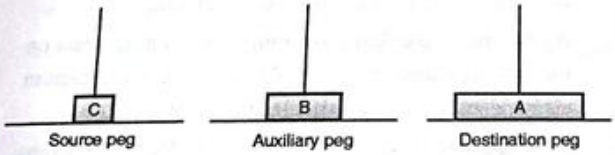
Step 3 : Move disk C from the *dst* peg to *aux* peg



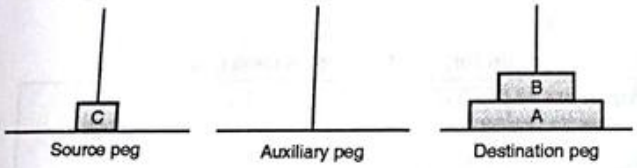
Step 4 : Move disk A from the *src* peg to *dst* peg



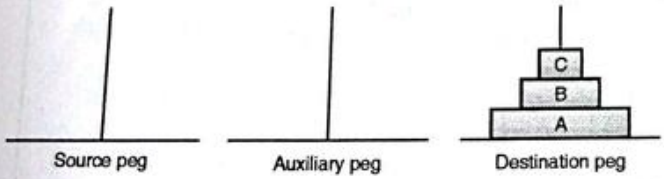
Step 5 : Move disk C from the *aux* peg to *src* peg



Step 6 : Move disk B from the *aux* peg to *dst* peg



Step 7 : Move disk C from the *src* peg to *dst* peg



Recursive approach is the best suitable for solving this problem. The recursive formulation for tower of Hanoi is given as,

HANOI(source, aux, dest, n) =

$$\begin{cases} \text{move from src to dst} & \text{if } n = 1 \\ \text{HANOI (src, dst, aux, } n-1) \\ \text{HANOI (src, aux, dst, 1)} \\ \text{HANOI(aux, src, dst, } n-1) & \text{otherwise} \end{cases}$$

```
Algorithm HANOI(src, aux, dest, n)
// Description: Move n disks from source peg to destination
   peg
// Input: 3 pages, and n disks on source peg
// Output: n disks on destination peg

if n = = 1 then
    Move disk from src to dest
else
    HANOI(src, dest, aux, n – 1)
    HANOI(src, aux, dst, 1)
    HANOI(aux, src, dest, n – 1)
end
```

Step 1 : Size of problem is *n*

Step 2 : Primitive operation is to move disk from one peg to another peg

Step 3 : Every call makes other two recursive calls with problem size n – 1. And each call corresponds to one primitive operation, so recurrence for this problem can be set up as follow :

$$T(n) = 2T(n-1) + 1 \quad \ldots (1)$$

Let us solve this recurrence using forward and backward substitution:

Substitute n by n – 1 in Equation (1),

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

By putting this value back in Equation (1),

$$\begin{aligned} T(n) &= 2[2T(n-2) + 1] + 1 \\ &= 2^2T(n-2) + 2 + 1 \\ &= 2^2T(n-2) + (2^2 - 1) \quad \ldots(2) \end{aligned}$$

Similarly, replace n by n – 2 in Equation (1),

$$T(n-2) = 2T(n-3) + 1,$$

From Equation (2),

$$\begin{aligned} T(n) &= 2^2[2T(n-3) + 1] + 2 + 1 \\ &= 2^3T(n-3) + 2^2 + 2 + 1 \\ &= 2^3T(n-3) + (2^3 - 1) \end{aligned}$$

In general,

$$T(n) = 2^kT(n-k) + (2^k - 1)$$

By putting k = n – 1,

$$T(n) = 2^{n-1}[T(1)] + (2^{n-1} - 1)$$

T(1) indicates problem of size 1. To shift 1 disk from source to destination peg takes only one move, so T(1) = 1.

$T(n) = 2^{n-1} + (2^{n-1} - 1)$
$= 2^n - 1$
Thus, $T(n) = O(2^n)$

Ex. 1.3.8
Setup and solve a Recurrence relation for the number of calls made by F(n), the recursive algorithms for computing n!

OR
Write an algorithm to find factorial using recursion. Find the time complexity.

Soln. :

Factorial of number n is computed as : $n! = n * (n - 1) * (n - 2) * \ldots * 2 * 1$

Recursive algorithm for finding factorial of any number is described below:

```
Algorithm FACTORIAL(n)
// Description: Find factorial of given number
// Input: Integer value n
// Output: Factorial of number n

if n == 0 or n == 1 then
    return 1              // Base case
else
    return n * (n – 1)    // Recursive case
end
```

For each call, problem size reduces by one and each call performs one multiplication. We can setup the recurrence as follow:

New problem of size (n – 1)

Cost of one multiplication

$T(n) = T(n-1) + 1$ (Inductive case) ... (1)
$T(n) = n$, if $n = 0$ or $n = 1$ (Base case)

Let use solves this recurrence using two methods:

Forward substitution

From Equation (1),
$T(2) = T(1) + 1 = 1 + 1 = 2$
$T(3) = T(2) + 1 = 2 + 1 = 3$
After k steps
$T(k) = k$
For $k = n$,
$T(n) = n = O(n)$

Backward substitution

To find $T(n)$, we need to find $T(n - 1)$. To solve $T(n - 1)$, let us put $n = n - 1$ in Equation (1).

$T(n-1) = T(n-1-1) + 1 = T(n-2) + 1$
So, $T(n) = T(n-2) + 1 + 1 = T(n-2) + 2$
Similarly,
$T(n-2) = T(n-2-1) + 1 = T(n-3) + 1$
So, $T(n) = T(n-3) + 2 + 1 = T(n-3) + 3$
After k steps,
$T(k) = T(n-k) + k$
For $k = n$,
$T(n) = T(n-n) + n = T(0) + n = n = O(n)$

Syllabus Topic : Analysis of Selection Sort, Insertion Sort

1.4 Analysis of Selection Sort and Insertion Sort

1.4.1 Selection Sort

Q. Explain selection sort and derive its complexity. **(10 Marks)**

Like bubble sort, selection sort is also comparison based in place algorithm. Selection sort is simple and it has obvious advantage of minimum swaps among all algorithms. For list of size n, selection sort performs maximum
$(n - 1)$ swaps. However, it's running time is quadratic and hence not accepted for a large list.

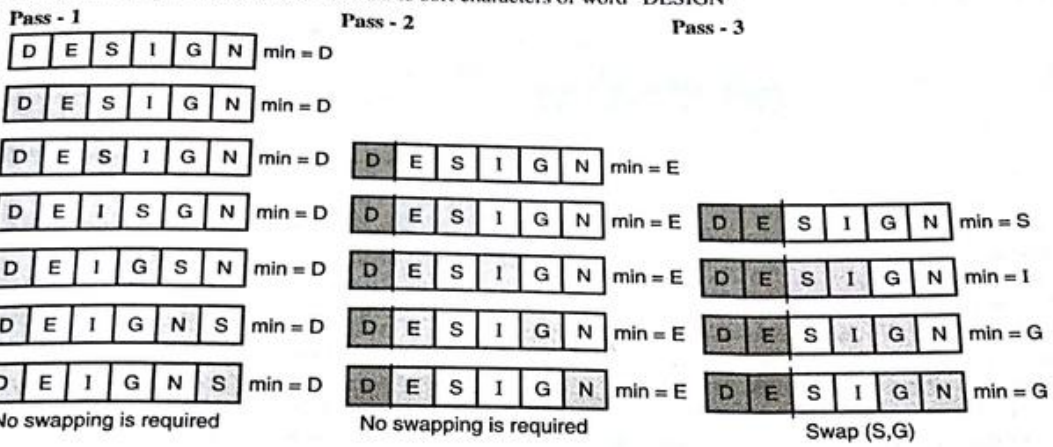
- In every iteration i, Selection sort finds the minimum element from the unsorted list and swap it with an i^{th} element in the list. At the end of the i^{th} pass, i elements get sorted. Sorting starts from the beginning.
- At the end of the first pass, minimum element seats on the first location, in the second pass, second minimum element seats on the second location and so on.
- If the list is reverse sorted, bubble sort does n – 1 swaps in the first pass, whereas selection sort does only one swap.
- Algorithm for selection sort is shown below :

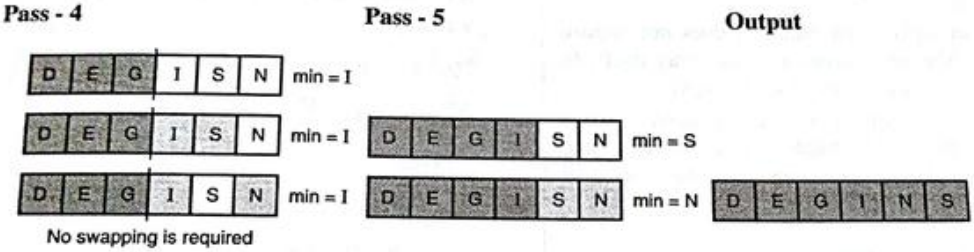
```
Algorithm SELECTION_SORT(A)
// A is an array of size n
for i ← 1 to n – 1 do
    min ← i
    for j ← i + 1 to n do
        if ( A[j] < A[min]) do
            min ← j
        end
    end
    swap(A[i], A[min])
end
```

Ex. 1.4.1

Sort the letters of word "DESIGN" in alphabetical order using selection sort.

Soln. : The following figure shows the simulation to sort characters of word "DESIGN"





Complexity analysis

- Similarly bubble sort, selection sort also dose the same number of comparisons. It iterates both loops irrespective of input data pattern.
- Unlike bubble sort, selection sort cannot detect sorted sequence. So running time of selection sort in best, average and worst case is $O(n^2)$. We can come to this conclusion by adding number of comparisons just like bubble sort, but here we use recurrence equation to derive the complexity.
- Let's assume T(n) defines the running time to solve the problem of size n. In selection sort, after each iteration, one element gets sorted and problem size reduces by one.
- For each problem of size n, inner loop iterates n times. So recurrence equation for selection sort can be written as,
- $T(n) = T(n-1) + n$

Put $n = n - 1$ in above equation,

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

Put value of $T(n-1)$ in previous equation of $T(n)$.

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

Put $n = n - 2$ in above equation,

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

Put value of $T(n-2)$ in previous equation of $T(n)$.

$$\therefore T(n) = T(n-3) + (n-2) + (n-1) + n$$

After k iterations,

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

$$T(n) = T(n-k) + (n-k+1) + (n-k+2) + \ldots + (n-1) + n$$

When k approaches to n,

$$T(n) = T(0) + 1 + 2 + 3 + \ldots + (n-1) + n$$

$T(0) = 0$, because it is running time of problem of size zero, and no effort is needed to solve this problem.

$$T(n) = 1 + 2 + 3 + \ldots + n$$

$$= \sum n = \frac{n(n+1)}{2} = \frac{n^2+n}{2} = \frac{n^2}{2} + \frac{n}{2} = O$$

$$\left(\max\left(\frac{n^2}{2}, \frac{n}{n}\right)\right) = O\left(\frac{n^2}{2}\right)$$

$$T(n) = O(n^2)$$

Best case	Average case	Worst case
$O(n^2)$	$O(n^2)$	$O(n^2)$

1.4.2 Insertion Sort

→ (May 17)

Q. Explain insertion sort and derive its complexity. **MU - May 2017, 10 Marks**

- We will analyze insertion sort algorithm in depth and for rest of the algorithms we will find out its running time from its structure.
- It's obvious observation that to sort hundred elements will take more time than just sorting five elements. Running time is function of input size.
- Insertion sort uses the analogy of sorting cards in hand. It works in a way people are used to sort cards. One card is removed at a time from the deck and it is inserted at correct location in hand. Upcoming cards are processed in same way.
- To insert new card, all the cards in hand having value larger than new card are shifted on right side by one. New card is inserted on space created after moving some k cards on right side.
- Insertion sort in inplace algorithm, it does not require extra memory. Sorting is done in input array itself. In iteration k, first k elements are always sorted.
- Running time is the number of steps required to solve problem on RAM model. Each instruction may take different amount of time. Let us consider the cost of i^{th} instruction is c_i.

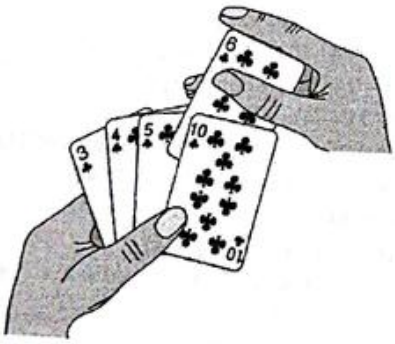


Fig. 1.4.1 : Process of insertion sort

☞ **Algorithm**

Algorithm Insertion Sort (A) *// A is array of size n*	Cost	Time
for $j \leftarrow 2$ to n do	c_1	n
$key \leftarrow A[j]$	c_2	$n-1$
$i \leftarrow j-1$	c_3	$n-1$
while ($i > 0$ && $A[i] > key$)	c_4	$\sum_{j=2}^{n} t_j$
$A[i+1] \leftarrow A[i]$	c_5	$\sum_{j=2}^{n} (t_j - 1)$
$i \leftarrow i-1$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
end	–	–
$A[i+1] \leftarrow key$	c_7	$n-1$
end	–	–

☞ **Complexity Analysis**

- Size of input array is n. Total time taken by algorithm is the summation of time taken by each of its instruction.

$$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot \left(\sum_{j=2}^{n} t_j\right) + c_5 \cdot \sum_{j=2}^{n} (t_j - 1) + c_6 \cdot \sum_{j=2}^{n} (t_j - 1) + c_7 \cdot (n-1)$$

- **Best case analysis :** Best case gives the lower bound of running time of algorithm, means for any other data sequence of data, running time cannot be less than its best case running time. Best case for insertion sort occurs when data is already sorted. For this case, condition in while loop will never satisfies and hence $t_j = 1$, so

$$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \sum_{j=2}^{n} 1 + c_5 \sum_{j=2}^{n} + 0 + c_6 \sum_{j=2}^{n} 0 + c_7 \cdot (n-1)$$

Where,

$$\sum_{j=2}^{n} 1 = 1 + 1 + \ldots + 1 \text{ (n - 1 times)} = n-1$$

$$= c_1 \cdot n + c_2 \cdot n - c_2 + c_3 \cdot n - c_3 + c_4 \cdot n - c_4 + c_7 \cdot n - c_7$$

$$= (c_1 + c_2 + c_3 + c_7)\, n - (c_3 + c_4 + c_4 + c_7)$$

$$= a\,n + b \text{ Which is linear function of n}$$

$$= O(n)$$

- **Worst case analysis :** The worst-case running time gives an upper bound of running time for any input. It indicates that for any arbitrary sequence of input data, running time of algorithm cannot get worse than its worst case running time. Worst case for insertion sort occurs when data is sorted in reverse order. So we must have to compare A[j] with each element of sorted array A[1 ... j – 1].

So $t_j = j$,

$$\sum_{j=2}^{n} j = 2 + 3 + 4 + \ldots + n$$

$$= (1 + 2 + 3 + \ldots + n) - 1 = \sum n - 1$$

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

and, $\sum_{j=2}^{n} (j-1) = 1 + 2 + 3 + \ldots + n - 1$

$= \Sigma (n-1) = \frac{n(n-1)}{2}$

$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot \left(\sum_{j=2}^{n} j\right)$

$+ c_5 \cdot \sum_{j=2}^{n} (j-1) + c_6 \cdot \sum_{j=2}^{n} (j-1)$

$+ c_7 \cdot (n-1)$

$= c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot \left(\frac{n(n+1)}{2} - 1\right) + c_5 \cdot \frac{n(n-1)}{2}$

$+ c_6 \cdot \frac{n(n-1)}{2} + c_7 \cdot (n-1)$

$= \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right) n^2 +$

$\left(c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7\right)$

$n - (c_2 + c_3 + c_4 + c_7)$

$= an^2 + bn + c$

which is quadratic function of n

$= O(n^2)$

- **Average case analysis :** Average case is often roughly as bad as worst case. On an average, half of the elements are greater than A[j] and other is less than that. So, $t_j = (j / 2)$. It again turns out to be quadratic function of n. Average case running time of insertion sort is $O(n^2)$.

Best case	Average case	Worst case
O(n)	$O(n^2)$	$O(n^2)$

Ex. 1.4.2

Sort the letters of word "EXAMPLE" in alphabetical order using insertion sort.

Soln. :

Initial array of alphabets

E	X	A	M	P	L	E

- One by one element is scanned from array. In iteration k, if incoming character is greater than previous element, then copy newly read character at location k.
- If k^{th} element is smaller than $(k-1)^{st}$ element, then keep movie elements on right side until we get element smaller than newly coming element. Insert new element in created free space. Step by step simulation of insertion sort is shown here :

Step 1 : Read E. E is the only character, so no shifting or swapping is required.

E	X	A	M	P	L	E

Step 2 : Read X. E and X are in order, so no shifting or swapping is required.

E	X	A	M	P	L	E

Step 3 : Read A. A and X are out of order, so move X on 3rd position in array. Compare A with E. Again, A and E are out order, so move E on 2nd position in array. Now there are no more elements left in array, so insert A on 1st vacant position

E	X	A	M	P	L	E

Step 4 : Read M. M and X are out of order, so move X on 4th position in array. Compare M with E. They are in order, so insert M on 3rd vacant position

A	E	X	M	P	L	E

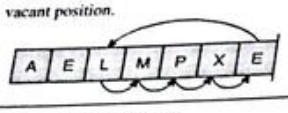
Step 5 : Read P. P and X are out of order, so move X on 5th position in array. Compare P with M. They are in order, so insert P on 4th vacant position

A	E	M	X	P	L	E

Step 6 : Read L. L is less than X, P and M, so move all three characters on right by position and insert L on 3rd vacant position.

A	E	M	P	X	L	E

Step 7 : Read E. E is less than X, P, M and L, so move all four characters on right by position and insert E on 3rd vacant position.

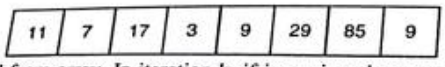


Step 8 : No more elements are left in the array. The array is sorted.

Ex. 1.4.3 **MU - Dec. 2014, 10 Marks**

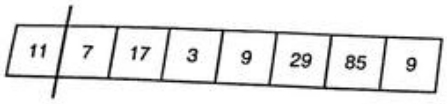
To sort the given set of number using insertion sort and also show the result of each pass. < 11, 7, 17, 3, 9, 29, 85, 9 >

Soln. : Initial array of numbers

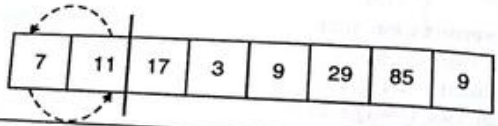


- *One by one element is scanned from array. In iteration k, if incoming character is greater than previous element, then copy newly read character at location k.*
- *If k^{th} element is smaller than $(k-1)^{st}$ element, then keep movie elements on right side until we get element smaller than newly coming element. Insert new element in created free space. Step by step simulation of insertion sort is shown here :*

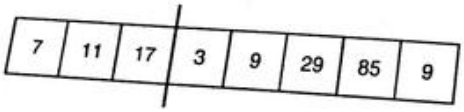
Step 1 : Read 11. 11 is the only number, so shifting or swapping is not required.



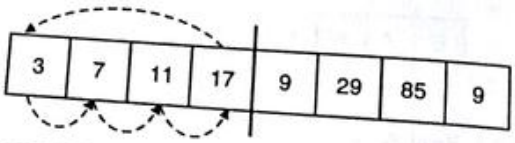
Step 2 : Read 7. 7 and 11 are out of order, so move 11 on 2nd position in array. Now there are no more elements left in array on left of side, so insert 7 on 1st vacant position



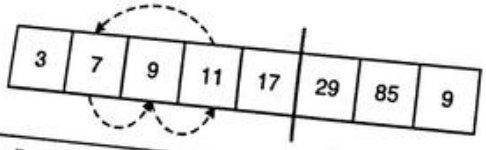
Step 3 : Read 17. It is in correct position, so shifting or swapping is not required.



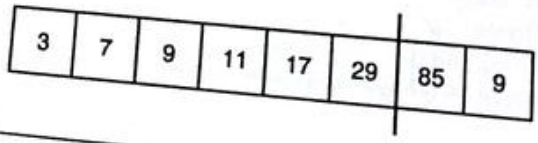
Step 4 : Read 3. It is not in correct position, so shift elements greater than 3 on the right side by one position until the correct location for 3 is found.



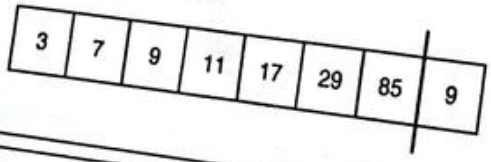
Step 5 : Read 9. It is not in correct position, so shift elements greater than 9 on the right side by one position until the correct location for 9 is found.



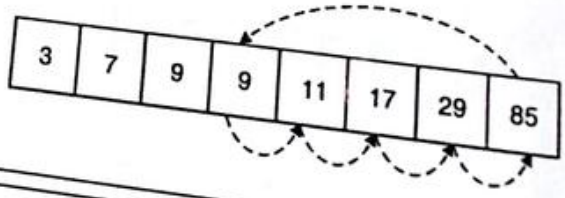
Step 6 : Read 29. It is in correct position, so shifting or swapping is not required.



Step 7 : Read 85. It is in correct position, so shifting or swapping is not required.



Step 8 : Read 9. It is not in correct position, so shift elements greater than 9 on the right side by one position until the correct location for 9 is found.



1.5 Exam Pack (University and Review Questions)

☞ **Syllabus Topic : Space and Time Complexity**

Q. Explain space complexity in detail. *(Ans. : Refer section 1.2.1)* (3 Marks) **(May 2013, Dec. 2013)**

Q. What are the basic components which contribute to space complexity? *(Ans. : Refer section 1.2.1)*(4 Marks)

Q. What do you mean by space complexity of an algorithm? How do we measure the space complexity of an algorithm? Explain with suitable example. *(Ans. : Refer section 1.2.1)* (4 Marks)

Q. Explain time complexity in detail. *(Ans. : Refer section 1.2.2)* (3 Marks) **(May 2013, Dec. 2013)**

Q. How do we analyze and measure the time complexity of an algorithm? *(Ans. : Refer section 1.2.2)*(4 Marks)

Q. What do you mean by time complexity of an algorithm? How do we measure the time complexity of an algorithm? Explain with suitable example. *(Ans. : Refer section 1.2.2)*(4 Marks)

☞ **Syllabus Topic : Growth of Function - Big Oh, Omega, Theta Notation**

Q. Define order of growth. List various efficiency classes with example. Show the relationship between efficiency classes. *(Ans. : Refer section 1.2.3)*(6 Marks)

Q. Explain Asymptotic notations. *(Ans. : Refer section 1.2.4)* (5/4 Marks) **(May 2013, Dec. 2013, May 2016)**

Q. Explain O, Ω, and θ notations with the help of graph. *(Ans. : Refer section 1.2.4)* (3 Marks) **(May 2014)**

Q. Define O, Ω, and θ notations. *(Ans. : Refer section 1.2.4)* (3 Marks) **(Dec. 2015)**

Q. Define and explain big Oh notations. Give examples. *(Ans. : Refer section 1.2.4(A))* (2 Marks)

Q. Define and explain Ω notations. Give examples. *(Ans. : Refer section 1.2.4(B))* (2 Marks)

Q. Define and explain Θ notations. Give examples. *(Ans. : Refer section 1.2.4(C))* (2 Marks)

Ex. 1.2.1 (7 Marks) **(May 2014)**

☞ **Syllabus Topic : Mathematical Background for Algorithm Analysis**

Q. Discuss the general plan for analyzing efficiency of non-recursive algorithm. *(Ans. : Refer section 1.3.1)* (6 Marks)

Q. Explain the framework of efficiency analysis of non-recursive algorithms with suitable examples. *(Ans. : Refer section 1.3.1)* (6 Marks)

Q. Explain the framework of efficiency analysis of recursive algorithms with suitable examples. *(Ans. : Refer section 1.3.2)* (5 Marks)

☞ **Syllabus Topic : Analysis of Selection Sort, Insertion Sort**

Q. Explain selection sort and derive its complexity. *(Ans. : Refer section 1.4.1)* (10 Marks)

Q. Explain insertion sort and derive its complexity. *(Ans. : Refer section 1.4.2)* (10 Marks) **(May 2017)**

Ex. 1.4.3 (10 Marks) **(Dec. 2014)**

❑❑❑

Module 1

CHAPTER 2

Divide and Conquer

Syllabus Topics

General method, Analysis of Merge sort, Analysis of Quick sort, Analysis of Binary search, Finding the minimum and maximum algorithm and analysis, Strassen's matrix multiplication.

- *In this chapter, we will discuss an interesting problem-solving technique called divide and conquer. The technique divides the larger problem into smaller sub-problems, and solution of the original large problem is obtained by combining a solution of smaller subproblems. Such algorithms are a prime candidate for parallel implementation. We will discuss and compare the performance of various problems solved using traditional method and using divide and conquer approach. Divide and conquer method easily outperforms the traditional algorithms.*

Syllabus Topic : General Method

2.1 General Method

→ **(May 13, Dec. 13, May 14)**

Q. Explain Divide and Conquer strategy. List any Four examples that can be solved by divide and conquer. **MU - May 2013, Dec. 2013, 6 Marks**

Q. Comment on the module of computation : Divide and Conquer. **MU - May 2014, 5 Marks**

2.1.1 Introduction

Q. Explain the concept of divide and conquer. **(5 Marks)**

- Divide and conquer strategy operates in three stages :

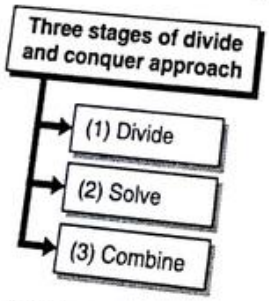


Fig. C2.1 : Divide and conquer algorithm stages

→ **(1) Divide :** Recursively divide the problem into smaller subproblems.

→ **(2) Solve :** Subproblems are solved independently.

→ **(3) Combine :** Combine solutions of subproblems in-order to derive the solution of the original big problem.

- Subproblems are similar to the original problem with smaller arguments, hence such problems can be easily solved using recursion.
- When subproblem hits to smallest possible size, it is solved and the results are recursively combined to generate a solution of the original bigger problem.
- Divide and conquer is multi-branched, top-down recursive approach. Each branch indicates one subproblem and it calls itself with the smaller argument.
- Understanding and designing of divide and conquer algorithms needs skill and good reasoning.
- Fig. 2.1.1 shows the graphical representation of divide and conquer strategy. Subproblems may not be of size exactly n/2.

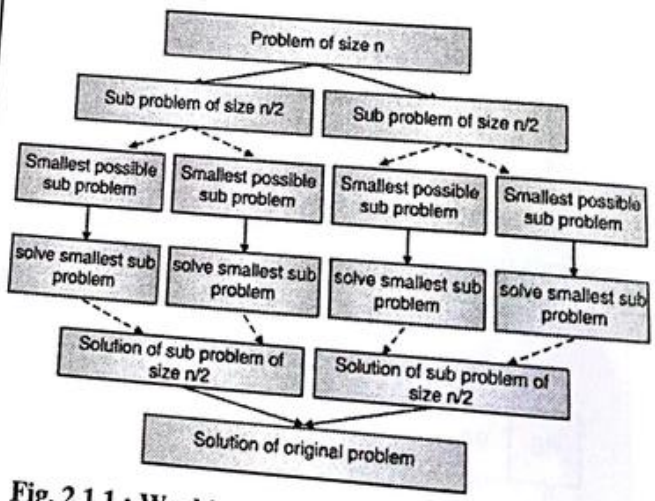


Fig. 2.1.1 : Working principle of divide and conquer

Applications

Q. Enlist few problems that can be solved using divide and conquer approach. (3 Marks)

Many computer science problems are effectively solved using divide and conquer. Few of them are listed here:

1. Finding exponential of the number
2. Multiplying large numbers
3. Multiplying matrices (Strassen's algorithm)
4. Sorting elements (Quicksort and merge sort)
5. Searching element from the list (Binary search)
6. Discrete Fourier Transform
7. Closest pair problem
8. Max-min problem

2.1.2 Control Abstraction

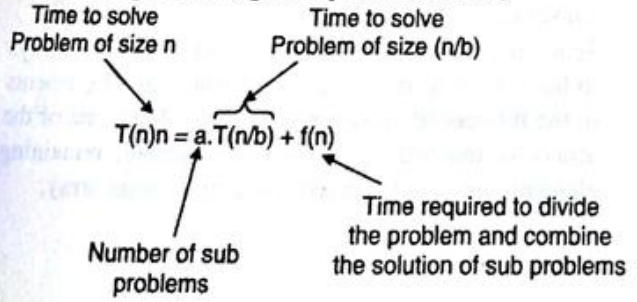
➔ **(May 13, Dec. 13)**

Q. Write control abstraction (General method) for divide and conquer. MU - May 2013, Dec. 2013, 4 Marks

- As we discussed, divide and conquer approach works in three stages :
  1. Recursively divide the problem into smaller subproblems
  2. Subproblems are solved independently
  3. Combine solutions of subproblems in-order to derive the solution of the original big problem.
- Control abstraction for divide and conquer (DC) approach is given below :

```
Algorithm DC(P)
// P is the problem to be solved

if P is small enough then
    return Solution of P
else
    divide larger problem P into k smaller subproblems P1,
    P2, ..., Pk
    solve each small subproblem Pi using DC strategy
    return combined solution (DC(P1), DC(P2), ..., DC(Pk))
end
```

- Each subproblem in divide and conquer are independent and hence sub-problems may be solved multiple times.
- If we create **a** problems, each of size **n/b**, and if division/combination cost is **f(n)**, the time complexity of such problem is given by the recurrence,



2.1.3 Efficiency Analysis

Q. Derive a general equation to find the complexity of divide and conquer approach. (5 Marks)

- In general form, the time complexity of problem solved using divide and conquer approach is given by following recurrence equation:

$$T(n) = \begin{cases} g(n) & \text{, if n is too small} \\ T(n_1) + T(n_2) \ldots + T(n_k) + f(n) & \text{, otherwise} \end{cases}$$

- T(n) is the total time required to solve the problem of size *n*. The g(n) is the cost of solving the very small problem. It denotes the complexity of solving the base case. $T(n_i)$ is the cost of solving subproblem of size n_i. The function f(n) represents the time required to divide the problem and combine the solution of subproblems.
- Generalized recurrence for this strategy is written as T(n) = a.T(n/b) + f(n), where **a** is the number of sub problems, **n/b** is the size of each sub problem and **f(n)** is the cost of division or combination. Thus,

$$T(n) = a.T(n/b) + f(n)$$

- Assume that we have at least one problem, so $a \geq 1$, and size of each problem is n/2, so b = 2,

Assume $n = b^k$, where k = 1, 2, 3, 4, ...

$$T(b^k) = a.T(b^k/b) + f(b^k)$$
$$= a.T(b^{k-1}) + f(b^k) \quad \ldots (2.1.1)$$

- To compute $T(b^{k-1})$, replace k by k – 1 in Equation (2.1.1).

$$\therefore T(b^{k-1}) = a.T(b^{k-2}) + f(b^{k-1})$$
$$\Rightarrow T(b^k) = a[a.T(b^{k-2}) + f(b^{k-1})] + f(b^k)$$
$$= a^2.T(b^{k-2}) + af(b^{k-1})] + f(b^k) \quad \ldots (2.1.2)$$

By substituting k = k – 2 in Equation (2.1.1).

$$T(b^{k-2}) = a.T(b^{k-3}) + f(b^{k-2})$$

Substitute $T(b^{k-2})$ in Equation (2.1.2)

$$\therefore T(b^k) = a^2[a.T(b^{k-3}) + f(b^{k-2})] + af(b^{k-1}) + f(b^k)$$
$$= a^3.T(b^{k-3}) + a^2f(b^{k-2}) + af(b^{k-1}) + f(b^k)$$

After k iterations,

$$T(b^k) = a^k.T(b^{k-k}) + a^{k-1}f(b) + a^{k-2}f(b^2) + \ldots + a^0f(b^k)$$
$$= a^k.T(1) + a^{k-1}f(b) + a^{k-2}f(b^2) + \ldots + a^0f(b^k)$$
$$= a^kT(1) + \frac{a^k}{a} f(b) + \frac{a^k}{a^2} f(b^2) + \ldots + \frac{a^k}{a^k} f(b^k)$$
$$= a^k \left[T(1) + \sum_{i=1}^{k} f(b^i)/a^i \right]$$

$n = b^k$, so $k = \log_b n$

$$T(n) = a^{\log_b n} \left[T(1) + \sum_{i=1}^{\log_b n} f(b^i)/a^i \right]$$

By property of logarithm,

$$x^{\log_b y} = y^{\log_b x}$$

$$T(n) = n^{\log_b a} \left[T(1) + \sum_{i=1}^{\log_b n} f(b^i)/a^i \right] \quad \ldots (2.1.3)$$

- Thus, the complexity of problem depends on a number of problems **a**, size of the problem **(n/b)** and the division/combination cost **f(n)**.

2.1.4 Sorting → (May 15)

Q. Which are different factors considered for sorting elements. **MU - May 2015, 5 Marks**

- ***Sorting*** *is a process of arranging elements in certain order. Numeric data may be sorted in ascending or descending order. Alphabets or strings may be sorted in lexicographical order. Sorting is intensively studied and explored area of computer science. In many applications, sorting is the inherent need.*
- *Large classes of algorithms are available for sorting. All have certain limitations and fix application domain. No algorithm serves all the purposes. However, running time is the most commonly used measure to select the best algorithm.*
- *If numbers of elements are small enough to sort in main memory, sorting is called* ***internal sorting****. If numbers of elements are large such that some of them reside on external storage during sorting procedure, it is called* ***external sorting****.*
- *Sorting algorithm should possess following properties :*

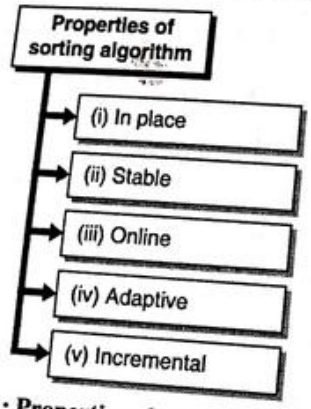


Fig. C2.2 : Properties of sorting algorithm

→ **(i) In place**

Only require constant additional space to sort the algorithm

→ **(ii) Stable**

Does not alter the relative position of same elements after sorting

→ **(iii) Online**

Sort the data as it arrives

→ **(iv) Adaptive**

Performance of algorithm vary with input pattern

→ **(v) Incremental**

Build sorted sequence one number at a time

Syllabus Topic : Analysis of Merge Sort

2.2 Merge Sort

Q. Analyze merge sort and find time complexity of merge sort. **(5 Marks)**

Merge sort sorts the list using divide and conquer strategy. It works on following two basic principles :

- Sorting smaller list is faster than sorting the larger list.
- Combining two sorted sublists is faster than that of two unsorted lists.

Merge sort works in three stages :

- **Divide :** Recursively divide the single list into two sublists until each sublist contains 1 element.
- **Conquer :** Sort both sublists of parent list. List of 1 element is sorted.
- **Combine :** Recursively combine sorted sublists into the list of size n is produced.
- Merge sort divides the list of size *n* into two sublists, each of size n/2. This subdivision continues until problem size becomes one. After hitting to the problem size one, conquer phase starts. Fig. 2.2.1 shows the divide phase of merge sort.

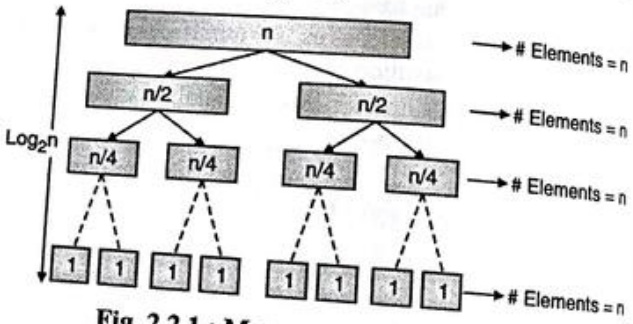


Fig. 2.2.1 : Merge sort - division stage

Definition

Two-way merge is the process of merging two sorted arrays of size m and n into a single sorted array of size (m + n).

- Merge sort compares two arrays, each of size one by performing the 2-way merge. The sorted sequence is kept in a new array of size two.
- In next step, two sorted arrays, each of size two are merged into a single array of size four. This process continues till solution of the entire problem is constructed.
- Sentinel symbol is inserted at the end of both the arrays to be merged to inform the algorithm that all elements in the referenced array are processed. When one of the arrays is reached to its Sentinel symbol, remaining elements are simply copied into a final larger array.

Algorithm for merge sort

Q. Given a sequence of n-elements A[1] ..... A[n], assume that they are split into 2 sets A[1] ..... A[n/2] and A[n/2 + 1] ... A[n], each set is individually sorted and the resulting sequence is merged to produce a single sorted sequence of n elements.

Using the divide and conquer strategy, write a Merge sort algorithm to sort the sequence in non-decreasing order. **(6 Marks)**

Q. Write an algorithm for sorting 'n' numbers using merge sort. **(6 Marks)**

Algorithm for merge sort is described below :

```
Algorithm MERGE_SORT(A, low, high)
// Description : Sort elements of array A using Merge sort
// Input : Array of size n, low ← 1 and high ← n
// Output : Sorted array B

if low < high then
    mid ← floor(low + high) / 2
    MERGE_SORT(A, low, mid)

    MERGE_SORT (A, mid + 1, high)
    COMBINE(A, low, mid, high)
            // Merge two sorted sublists
end
```

Recursively sort first sub list

Recursively sort second sub list

Procedure COMBINE performs a two-way merge to produce a sorted array of size n from two sorted arrays of size n /2. The subroutine works as follow:

```
COMBINE (A, low, mid, high)
l1 ← mid – low + 1          // Size of 1st array
l2 ← high – mid             // Size of 2nd array
for i ← 1 to l1 do
    LEFT [i] ← A [low + i–1]  // Copy 1st sub list in array
    LEFT
end
for j ← 1 to l2 do
    RIGHT [j] ← A [mid + j]   // Copy 2nd sub list in array
    RIGHT
End

// Insert sentinel symbol at the end of each array
LEFT [l1 + 1] ← ∞
RIGHT [l2 + 1] ← ∞

// Start two-way merge process
i ← 1, j ← 1
for k ← low to high do
                    // Perform 2-way merge on LEFT & RIGHT
    if LEFT [i] ≤ RIGHT [j] then
        // Smaller element is in LEFT sub list, so copy it in B
        B[k] ← LEFT [i]
        i ← i + 1
    else
        // Smaller element is in RIGHT sub list, so copy it in
        B
        B[k] ← RIGHT[j]
        j ← j + 1
    end
end
```

Two-way merge

Ex. 2.2.1

Simulate merge sort on data sequence

<77, 22, 33, 44, 11, 55, 66>

Soln. :

Let us understand the working of merge sort through graphical representation and call sequences. Consider the sequence of the element as shown below:

77	22	33	44	11	55	66
0	1	2	3	4	5	6
↑						↑
low						high

Simulation of merge sort for given array is depicted in the Fig. P. 2.2.1. The number in a circle on left or right corner indicates the call sequence. COMBINE is called as soon as two sublists of size 1 are created.

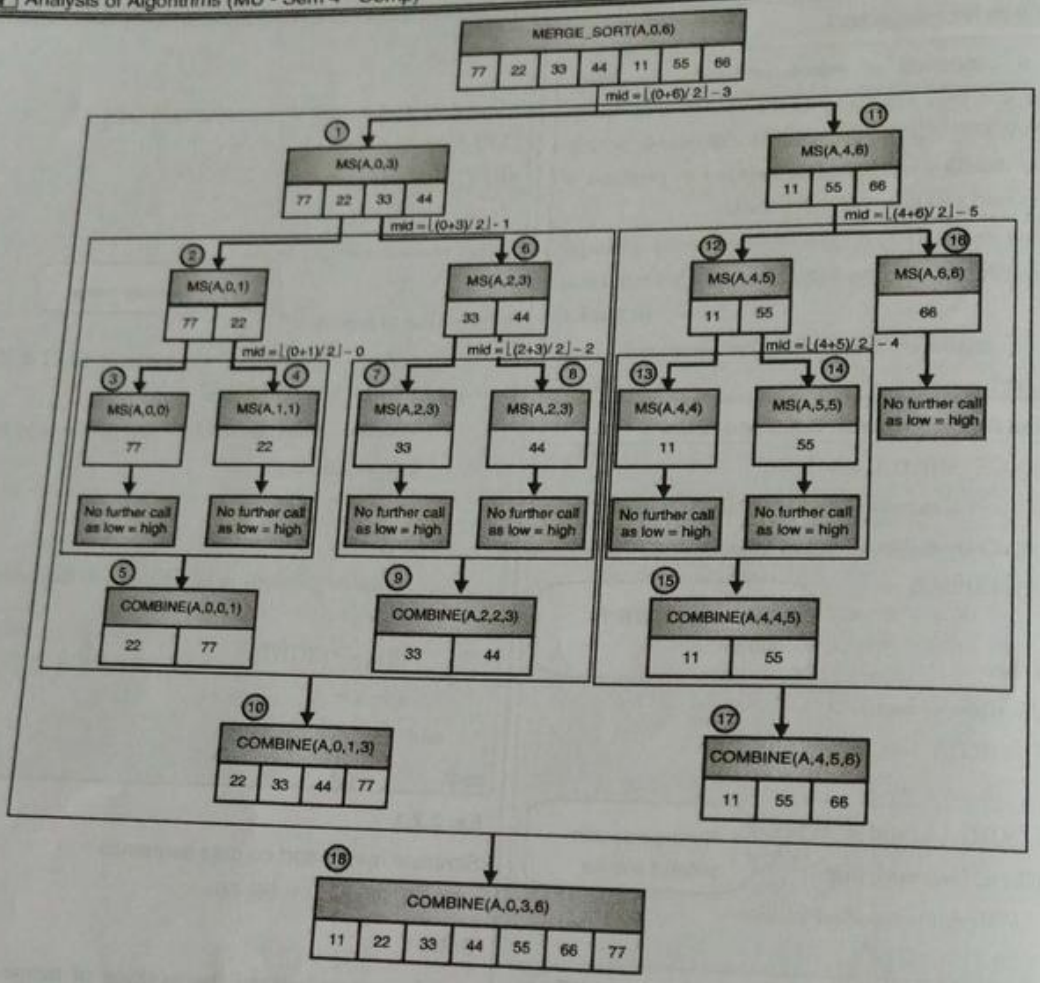
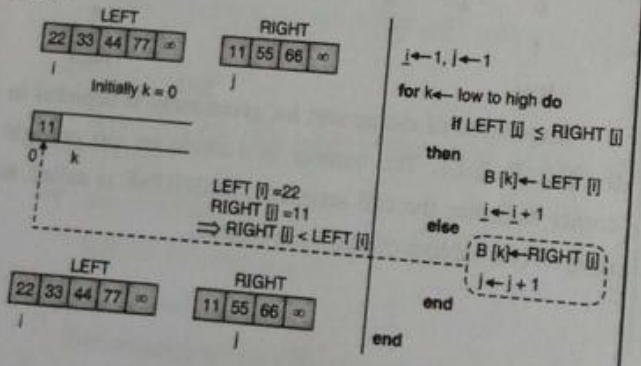


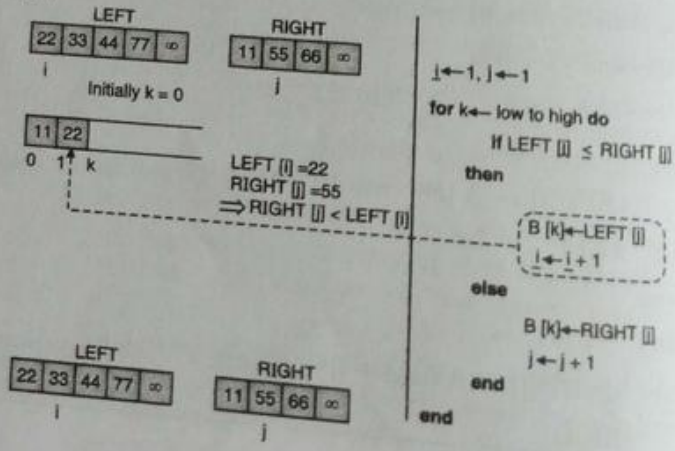
Fig. P. 2.2.1

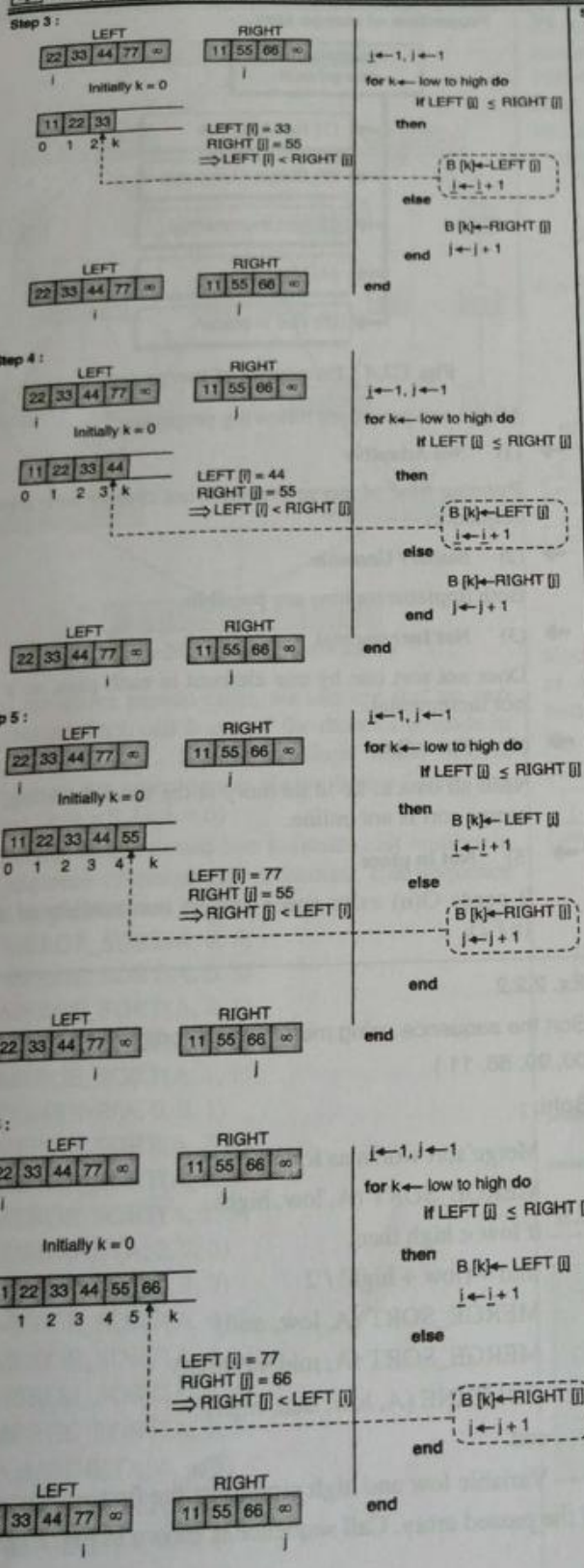
Let us see how COMBINE does two-way merge of two array. Consider the two sorted sub lists LEFT = {22, 33, 44, 77} and RIGHT = {11, 55, 66}.

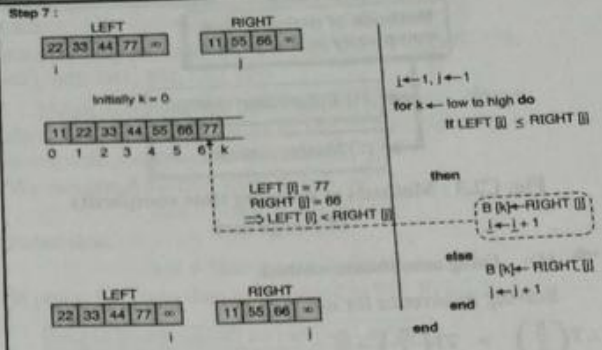
Step 1 :



Step 2 :







Finally, the sorted sequence of two sub lists LEFT = {22, 33, 44, 77} and RIGHT = {11, 55, 66} would be B={11, 22, 33, 44, 55, 66, 77}.

Complexity analysis

Q. State time complexity of merge sort. **(4 Marks)**

Q. Formulate recurrence for merge sort and find the time complexity of it. **(4 Marks)**

- On each recursive call, the list is divided into two sublists, and problem size reduces by half. After sorting two sublists of size (n / 2), combine procedure takes n comparisons to form the sorted list of size n. Total running time of merge sort is computed by summing complexity of following three steps:

  (1) Divide : This step computes the middle index of the array, which can be done in constant time. Thus,
  $D(n) = \Theta(1)$.

  (2) Conquer : We recursively solve two subproblems, each of size (n / 2), which contributes $2T(n/2)$ to the running time.

  (3) Combine : COMBINE procedure merges two sublists, each of size n/2 and does n comparisons, thus $C(n) = \Theta(n)$.

- Hence, T(n) = T(Conquer) + T(Divide) + T(Combine)

$$T(n) = \begin{cases} 0 & , \text{if } n \le 1 \\ T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + D(n) + C(n) & , \text{else} \end{cases}$$

Where first $T\left(\frac{n}{2}\right)$ = Cost for solving left sublist

And second $T\left(\frac{n}{2}\right)$ = Cost for solving right sublist

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) + \Theta(1)$$

$$= 2T\left(\frac{n}{2}\right) + n \quad \ldots(2.2.1)$$

We will derive the time complexity of merge sort using two methods.

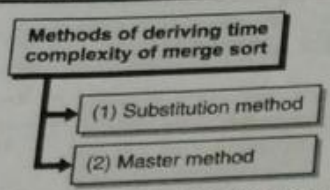


Fig. C2.3 : Methods of deriving time complexity of merge sort

→ **(1) Using substitution method**

Solving recurrence for n/2,

$$\therefore T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + \frac{n}{2}$$

Substitute it in Equation (2.2.1)

$$\therefore T(n) = 2\left(2 \cdot T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n$$

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

After k substitutions,

$$T(n) = 2^k \cdot T\left(\frac{n}{2^k}\right) + kn$$

Suppose, $n = 2^k$, so $k = \log_2 n$

$$T(n) = n \cdot T\left(\frac{n}{n}\right) + \log_2 n \cdot n$$

$$= n \cdot T(1) + n \cdot \log_2 n$$

But, $T(1) = 0$

// Base case: no comparison is needed for problem of size 1

So, $T(n) = O(n \cdot \log_2 n)$

→ **(2) Using Master method**

Recurrence of the merge sort, $T(n) = 2T\left(\frac{n}{2}\right) + n$ is of the form $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$.

Comparing both the recurrence,

$a = 2$, $b = 2$ and $f(n) = n$ with $d = 1$, where d is the degree of polynomial function f(n).

$b^d = 2^1 = 2$

Here, $a = b^d$, so from the Case 1 of Variant I of master method,

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

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

Whether the list is already sorted, inverse sorted or randomly shuffled, all three steps must be performed. Merge sort cannot detect if the list is sorted. So numbers of comparisons are same for all three cases.

The time complexity of merge sort for all three cases is stated in the following table :

Best case	Average case	Worst case
$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$

Properties of merge sort

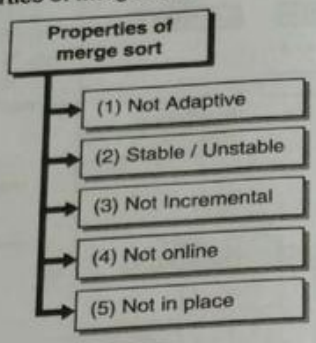


Fig. C2.4 : Properties of merge sort

Merge sort possesses following properties:

→ **(1) Not Adaptive**

Running time of merge sort does not change with input sequence.

→ **(2) Stable / Unstable**

Both implementations are possible.

→ **(3) Not Incremental**

Does not sort one by one element in each pass, so it is not incremental.

→ **(4) Not online**

Need all data to be in memory at the time of sorting, so merge sort is not online.

→ **(5) Not in place**

It needs O(n) extra space to sort two sublists of size (n / 2).

Ex. 2.2.2

Sort the sequence using merge sort algorithm : A {33, 22, 44, 00, 99, 88, 11 }

Soln. :

Merge sort works as follow :

```
MERGE_SORT (A, low, high) :
if low < high then,
mid = (low + high) / 2
MERGE_SORT (A, low, mid)
MERGE_SORT (A, mid +1, high)
COMBINE (A, low, mid, high)
end
```

Variable low and high represents the first and last index of the passed array. Call sequence is shown in Fig. P. 2.2.2.

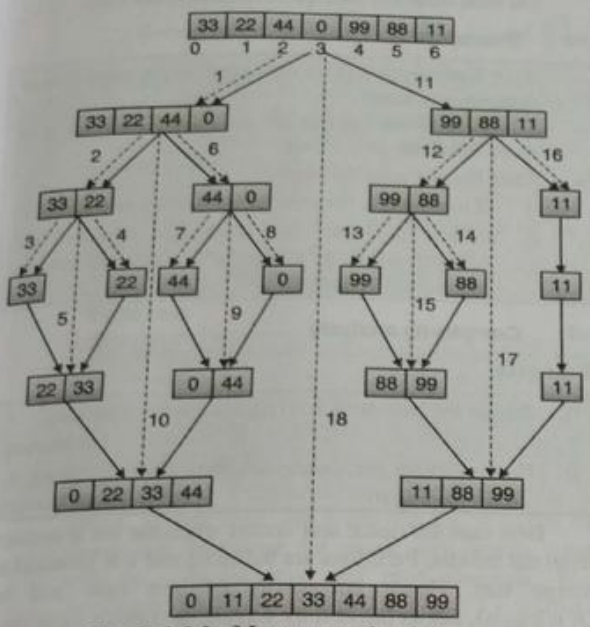


Fig. P. 2.2.2 : Merge sort simulation

From the above pseudo code, we can see that no two calls are parallel. Each call is one of the three calls made in MERGE_SORT(A, low, high) procedure stated above. Calling sequence for given data is shown below (according to the picture, low = 0, high = 6) :

The number beside dotted line indicates call number in the entire sequence of merge sort procedure. Call sequence for given example is shown below:

Step-0 : MERGE_SORT(A, 0, 6)
Step-1 : MERGE_SORT(A, 0, 3)
Step-2 : MERGE_SORT(A, 0, 1)
Step-3 : MERGE_SORT(A, 0, 0)
Step-4 : MERGE_SORT(A, 1, 1)
Step-5 : COMBINE(A, 0, 0, 1)
Step-6 : MERGE_SORT(A, 2, 3)
Step-7 : MERGE_SORT(A, 2, 2)
Step-8 : MERGE_SORT(A, 3, 3)
Step-9 : COMBINE (A, 2, 2, 3)
Step-10 : COMBINE (A, 0, 1, 3)
Step-11 : MERGE_SORT(A, 4, 6)
Step-12 : MERGE_SORT(A, 4, 5)
Step-13 : MERGE_SORT(A, 4, 4)
Step-14 : MERGE_SORT(A, 5, 5)
Step-15 : COMBINE (A, 4, 4, 5)
Step-16 : MERGE_SORT(A, 6, 6)
Step-17 : COMBINE (A, 4, 5, 6)
Step-18 : COMBINE (A, 0, 3, 6)

Ex. 2.2.3

Sort the following elements using merge sort: 410, 385, 279, 752, 451, 523, 961, 354, 550, 620

Soln. : Merge sort is divide and conquer approach. Merge sort algorithm divides the list in to two halves. Splitting is continuing until problem size reaches to 1.

We computed the mid value as follow:

$$mid = ceil(low + high) / 2$$

rather than,

$$mid = floor(low + high) / 2$$

Divide phase for given data is described in Fig. P. 2.2.3.

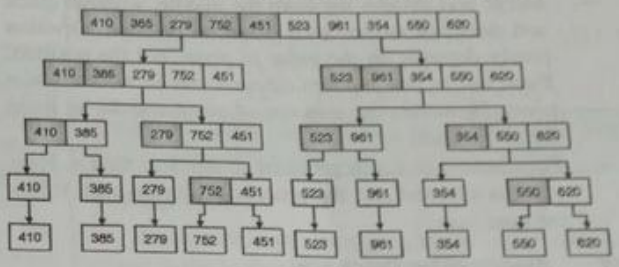


Fig. P. 2.2.3 : Divide stage

When problem size becomes trivial (i.e. n = 1), the algorithm starts solving the problem and merging solutions of smaller subproblems. The solution is constructed in a bottom-up approach. Conquer procedure is described in Fig. P. 2.2.3(a).

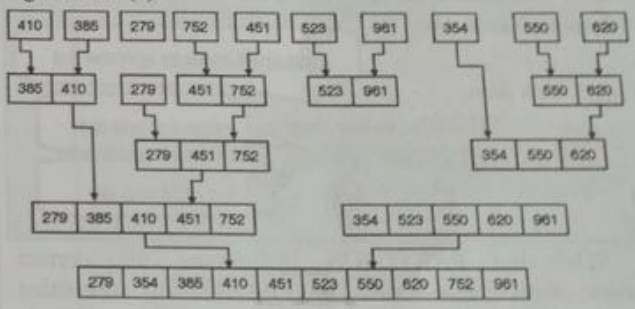


Fig. P. 2.2.3(a) : Combine stage

Syllabus Topic : Analysis of Quick Sort

2.3 Quick Sort

Q. Explain quick sort algorithm. Evaluate complexity with suitable example. **(10 Marks)**
Q. Explain quick sort algorithm using divide and conquer approach. **(6 Marks)**
Q. Discuss partition exchange sort and analyze it. **(5 Marks)**

– Quick sort was invented by Tony Hoare. It is divide and conquer based approach. Quick sort is also known as **partition exchange sort**.

- In every iteration, quick sort selects one element as a **pivot** and moves pivot to the correct location. Fixing the pivot on its correct location divides the list into two sub list of equal or unequal size.

Sublist 1	P	Sublist 2
$x < P$		$x \geq P$

- Each sublist is solved recursively. Unlike merge sort, here partitioning of the list is carried out dynamically. Merge sort partitions array based on the *position* of array elements, while quick sort does it based on *actual value* of elements.
- Merge sort divides list from the middle, whereas quick sort does not ensure such balanced division. Division purely depends on the *value* of pivot, not the position. First, last or any random element can be selected as a pivot. However, the position of pivot should be fixed for all iteration.
- Each sublist is a new problem of size less than n. New pivot is selected and the process continues until it hits the base case.

☞ Algorithm for Quick sort

Q. Write an algorithm for quick sort. **(3 Marks)**

Algorithm for quick sort is shown below :

```
Algorithm QUICKSORT (A, low, high)
// Description : Sort array A using Quick sort
// Input : Unsorted array A of size n, low = 0, high = n - 1
// Output : Sorted array A

if low ≤ high then
    q ← PARTITION (a, low, high)      // Find pivot index to split the list
    QUICKSORT (A, low, q – 1)         // Sort left sub list
    QUICKSORT (A, q + 1, high)        // Sort right sub list
end
```

Subroutine PARTITION determines the correct position of pivot in a sorted array. PARTITION subroutine scans the array from left and right. Using low index, it finds element greater than the pivot and using a high index it finds the element smaller or equal to pivot.

The pseudo code of PARTITION subroutine is given below:

```
PARTITION (A, low, high)

x ← A[high]  // x is pivot element i.e. last element of list
i ← low – 1
for j ← low to high – 1 do
    if A[j] ≤ x then
    i ← i + 1
        swap (A[i], A[j])
    end
end
swap (A[i + 1], A[high])
return (i + 1)          // Correct index of pivot after sorting
```

On each recursive call, quick sort does following:

☞ Procedure

- Scan array from left to right until an element greater than pivot is found.
- Scan array from right to left until an element equal or smaller than pivot is found.
- After finding such elements,
  - If low < high, then exchange A[low] and A[high].
  - If low ≥ high, then exchange A[pivot] and A[high], which will split the list into two sublists. Solve them recursively.

☞ Complexity analysis

Best case

Q. Derive the best case time complexity of quick sort. **(6 Marks)**

Q. Prove that for the Quick sort: Best Case efficiency is $T(N) = O(n \log n)$ **(5 Marks)**

Best case for quick sort occurs when the list is divided from the middle. Partitions are balanced and it is identical to merge sort. Hence complexity of best case will be $O(n \log_2 n)$. Let us prove it by solving recurrence equation.

$$T(1) = 0$$

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \quad \ldots(2.3.1)$$

$2T\left(\frac{n}{2}\right)$ is the time required to solve two problems of size (n/2)

$O(n)$ is the time required to fix the position of the pivot. Fixing of pivot requires a scan of the entire array.

The recurrence of the quick sort for the best case is identical to the recurrence of merge sort.

We will derive the time complexity of merge sort using two methods:

(1) Substitution method and (2) Master method

(1) Using substitution method

Solving recurrence for n/2,

$$\therefore T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + \frac{n}{2}$$

Substitute it in Equation (2.3.1)

$$\therefore T(n) = 2\left(2 \cdot T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n = 2^2 \cdot T\left(\frac{n}{2^2}\right) + 2n.$$

After k substitutions,

$$T(n) = 2^k \cdot T\left(\frac{n}{2^k}\right) + kn$$

Suppose, $n = 2^k$, so $k = \log_2 n$

$$T(n) = n \cdot T\left(\frac{n}{n}\right) + \log_2 n \cdot n$$

$$= n \cdot T(1) + n \cdot \log_2 n$$

But, $T(1) = 0$

// Base case: no comparison is needed for problem of size 1

So, $T(n) = O(n \cdot \log_2 n)$

(2) Using Master method

Recurrence of the merge sort, $T(n) = 2T\left(\frac{n}{2}\right) + n$ is of the form $T(n) = aT\left(\frac{n}{b}\right) + f(n)$.

Comparing both the recurrence,

$a = 2$, $b = 2$ and $f(n) = n$ with $d = 1$, where d is the degree of polynomial function f(n).

$b^d = 2^1 = 2$

Here, $a = b^d$, so from the Case 1 of Variant I of master method,

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

Worst case

Q. Derive worst-case time complexity of quick sort. Prove that worst-case efficiency of quick sort is $O(n^2)$. **(5 Marks)**

Q. Prove that for the Quick sort: Worst Case efficiency is $T(N) = O(n^2)$ **(5 Marks)**

Worst case for quick sort occurs when the list is already sorted. In that case, the pivot will be at either end. The worst case behaviour for quick sort creates subproblems with (n – 1) elements and zero elements. Worst case partitioning of the list is shown in Fig. 2.3.1. In this case, the height of tree would be n.

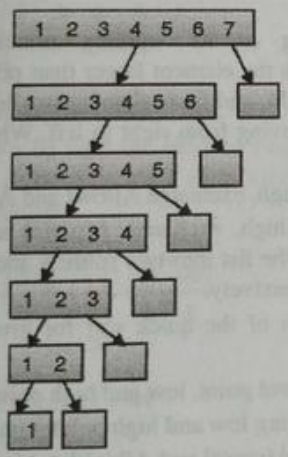


Fig. 2.3.1 : Worst case scenario for quick sort

Combination cost for quick sort is constant because each call fixes the correct position of the pivot and no extra work is required. To divide the list, algorithm scans the array and determines the correct position of the pivot, so division cost of QUICK_SORT is linear, i.e $\Theta(n)$. In the worst case, one of the sublists has size 0, and other has size (n – 1), so in next call, the algorithm has to solve a problem of size (n – 1).

Thus,

$T(0) = 1$ (Base case)

$T(n) = T(\text{conquer}) + T(\text{divide}) + T(\text{combine})$

$\therefore T(n) = T(n-1) + \Theta(n) + \Theta(1)$

$= T(n-1) + n$ ...(2.3.2)

Using iterative approach,

Substitute n by (n – 1) in Equation (2.3.2),

$T(n-1) = T(n-2) + (n-1)$ ...(2.3.3)

$\Rightarrow T(n) = T(n-2) + (n-1) + n$ ... (2.3.4)

Substitute n by (n – 1) in Equation (2.3.3),

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

From Equation (2.3.4),

$\therefore T(n) = T(n-3) + (n-2) + (n-1) + n$

: :

After k iterations,

$T(n) = T(n-k) + (n-k+1) + (n-k+2) + \ldots + (n-1) + n$

Let $k = n$,

$\therefore T(n) = T(0) + 1 + 2 + 3 + \ldots + (n-2) + (n-1) + n$

$= 1 + 2 + 3 + \ldots + n = \Sigma n$

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

Note : In worst case, quick sort behaves similar to selection sort.

Average case

For quick sort, average case running time is much closer to the best case.

That is,

$T(n) = O(n \log_2 n)$

The time complexity of all three cases is stated in the following table:

Best case	Average case	Worst case
$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$

☞ **Randomization**

➔ **(May 15)**

Q. Explain randomized version of Quick sort and derive its complexity. **MU - May 2015. 10 Marks**

Q. Explain randomized quick sort. **(3 Marks)**

Q. How to achieve O(n log n) time complexity in the worst case for quick sort. **(3 Marks)**

- Worst case for quick sort depends upon how we select pivot element. Worst case for quick sort occurs when the array is already sorted and first or the last element is used as a pivot. We can add randomization to an algorithm in order to obtain better performance.
- In the worst case, selection of first or last element as pivot partitions list in (n – 1) and 0 size sublists. By choosing random element as a pivot, partitions can roughly be balanced, and performance close to merge sort may be achieved.
- Selection of pivot point position is a key factor in the performance of a quick sort.

Few choices for pivot selection

1. The Select middle element of the list as a pivot.
2. Select random element from the list as a pivot.
3. Use median as a pivot.

Algorithm for randomized quick sort is shown below :

```
Algorithm RANDOMIZED_QUICKSORT (A, low, high)
// Description : Sort array A using Quick sort
// Input : Unsorted array A of size n, low = 0, high = n - 1
// Output : Sorted array A
                                  Find pivot index to split the list
if low ≤ high then
   q ← RANDOMIZED-PARTITION (a, low, high)
   QUICKSORT (A, low, q - 1)      Sort left sub list
   QUICKSORT (A, q + 1, high)     Sort right sub list
end
```

The pseudo code of RANDOMIZED_PARTITION subroutine is given below:

```
RANDOMIZED_PARTITION (A, low, high)

i ← RANDOM(low, high)
Exchange(A[i], A[high])
return PARTITION(A, low, high)
    // Correct index of pivot after sorting
```

- The subroutine PARTITION is discussed in regular version of quick sort.

Time Complexity

- If every time we select first or last element as pivot, then sorted list will be divided in two lists of size 0 and n – 1 for sorted data. But randomized version selects the pivot as above stated, which avoids pivot being always first or last. Lists after division will be no more biased now. On average we can consider that list will be divided in approximately two halves on each division, which leads to the recurrence $T(n) = 2T(n/2) + n$. The solution to the recurrence is discussed here:

Solving recurrence for n/2,

$$\therefore T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + \frac{n}{2}$$

Substitute it in Equation (2.3.1)

$$\therefore T(n) = 2\left(2 \cdot T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n$$

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

After k substitutions,

$$T(n) = 2^k \cdot T\left(\frac{n}{2^k}\right) + kn$$

Suppose, $n = 2^k$, so $k = \log_2 n$

$$T(n) = n \cdot T\left(\frac{n}{n}\right) + \log_2 n \cdot n$$

$$= n \cdot T(1) + n \cdot \log_2 n$$

But, $T(1) = 0$

// Base case: no comparison is needed for problem of size 1

So, $T(n) = O(n \cdot \log_2 n)$

Properties of quick sort

(1) In place : It uses O(logn) amount of extra memory.
(2) Quick sort is massively recursive.
(3) In the best case, quick sort is extremely fast.
(4) It is very complex.

Q. Why quick sort is better than merge sort?

Even though worst case running time of quick sort is $O(n^2)$, quick sort is considered better than merge sort due to many other factors. The complexity of sorting algorithm is measured by a number of comparison operations performed by the algorithm.

In general, quick sort does fewer comparisons than merge sort. Quick sort is in place algorithm and requires O(logn) extra space compared to O(n) space of merge sort. Quick sort also exhibits good cache locality, which minimizes the number of memory access. In addition, by introducing randomized version, worst case running time of quick sort can be made $O(n\log_2 n)$.

Ex. 2.3.1

Trace quick-sort for the data set : A = {44, 22, 33, 77, 11, 55, 66}

Soln. :

According to its working principle, quick sort algorithm finds the element larger than pivot while moving from left to right, and finds element smaller than or equal to pivot while moving from right to left. When such elements are found,

- If low < high, exchange A[low] and A [high].
- If low ≥ high, exchange A[pivot] and A[high]. This will split the list into two sublists and both sublists are sorted recursively.
- Simulation of the quick sort for given data is shown below :

Step 1 : Fix pivot point, low and high pointers.
Keep moving low and high pointer until
A[low] > A[pivot] and A[high] ≤ A[pivot].
And if low < high then swap A[low] and A[high]
low ≥ high then swap A[pivot] and A[high]

Initially,

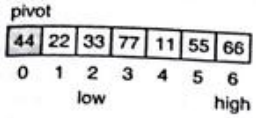
44	22	33	77	11	55	66
0	1	2	3	4	5	6

Step 2 : A[low] = 22 and A[pivot] = 44
A[low] < A[pivot] ⇒ low = low + 1
= 1 + 1 = 2

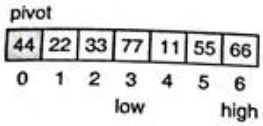
pivot

44	22	33	77	11	55	66
0	1	2	3	4	5	6
	low					high

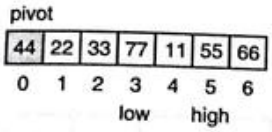
Step 3 : A[low] = 33 and A[pivot] = 44
A[low] < A[pivot] ⇒ low = low + 1
= 2 + 1 = 3



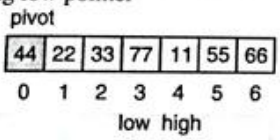
Step 4 : A[low] = 77 and A[pivot] = 44
A[low] > A[pivot] ⇒ fix low check for high pointer
A[high] = 66 and A[pivot] = 44
A[high > A[pivot] ⇒ high
= high – 1 = 6 – 1 = 5



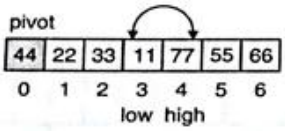
Step 5 : A[high] = 55 and A[pivot] = 44
A[high] > A[pivot] ⇒ high =
high – 1 = 5 – 1 = 4



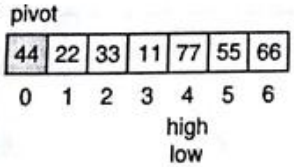
Step 6 : A[high] = 11 and A[pivot] = 44
A[high] ≤ A[pivot] ⇒ fix high and perform swap
Here, low < high, so swap A[low] with A[high] and again start moving low pointer



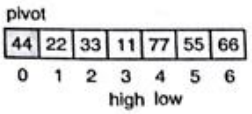
Step 7 : A[low] = 11 and A[pivot] = 44
A[low] < A[pivot] ⇒ low =
low + 1 = 3 + 1 = 4



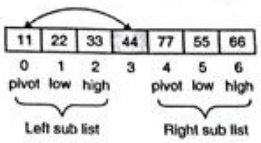
Step 8 :
A[low] = 77 and A[pivot] = 44
A[low] > A[pivot] ⇒ fix low and check high
A[high] = 77 and A[pivot] = 44
A[high] > A[pivot] ⇒ high
= high – 1 = 4 – 1 = 3



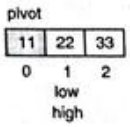
Step 9 : A[high] = 11 and A[pivot] = 44
A[high] ≤ A[pivot] ⇒ fix high and perform swap
Here, low ≥ high so swap A[pivot] with A[high] divide list in two sub lists and reset pointers. Recursively sort both the sub lists.



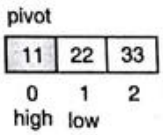
Step 10 : Let us first sort the *left* sub list
A[low] = 22 and A[pivot] = 11
A[low] > A[pivot], so fix low and check high pointer
A[high] = 33 and A[pivot] = 11
A[high] > A[pivot] ⇒ high
= high – 1 = 2 – 1 = 1



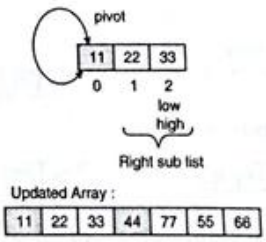
Step 11 : A[high] = 22 and A[pivot] = 11
A[high] > A[pivot] ⇒ high =
high – 1 = 1 – 1 = 0



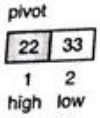
Step 12 : A[high] = 11 and A[pivot] = 11
A[high] ≤ A[pivot] ⇒ so fix high and perform swap
Here, low ≥ high so swap A[pivot] with A[high], divide list in two sub lists and reset pointers. Recursively sort both the sub lists



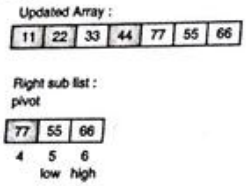
Step 13 : A[low] = 33 and A[pivot] = 22
A[low] > A[pivot] ⇒ so fix low and check high pointer
A[high] = 33 and A[pivot] = 22
A[right] > A[pivot] ⇒ high = high – 1 = 2 – 1 = 1



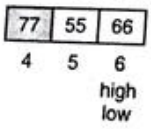
Step 14 : A[high] = 22 and A[pivot] = 22
A[high] ≤ A[pivot], so fix high and perform swap
Here, low ≥ high so swap A[pivot] with A[high], divide list in two sub lists. Left sub list has size zero and right sub list has only one element {33}, so no further process is required.



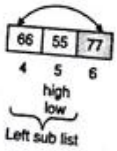
Step 15 : Let us now sort the right sub list of original array derived in step 9
A[low] = 55 and A[pivot] = 77
A[low] < A[pivot], low = low + 1 = 5 + 1 = 6

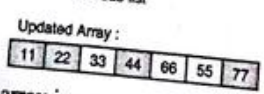


Step 16 : A[low] = 66 and A[pivot] = 77 A[low] < A[pivot], low = low + 1 = 6 + 1 = 7
low > high so stop, and check for high index
A[high] = 66 and A[pivot] = 77
A[high] < A[pivot], so fix high and perform swap
Here, low ≥ high so swap A[pivot] with A[high], divide list in two sub lists and reset pointers. Right sub list has size zero. So only left sub list needs to be sorted.

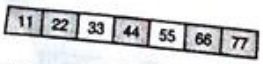


Step 17 : A[low] = 55 and A[pivot] = 66
A[low] < A[pivot], low = low + 1 = 5 + 1 = 6
low > high so stop, and check for high index
A[high] = 55 and A[pivot] = 66
A[high] < A[pivot], so fix high and perform swap
Here, low ≥ high so swap A[pivot] with A[high], divide list in two sub lists and reset pointers. Right sub list has size zero and right sub list is of size 1, so sorting is not required at all.





The final sorted array is :



– *Gray cells indicate movement of the pivot.*

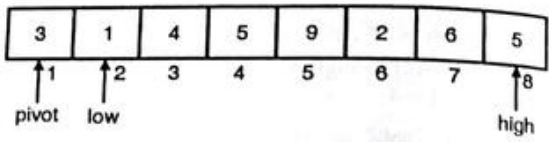
Ex. 2.3.2
Trace quick-sort for the data set
A = {3, 1, 4, 5, 9, 2, 6, 5}

Soln. :

– According to its working principle, quick sort algorithm finds the element larger than pivot while moving from left to right, and finds element smaller than or equal to pivot while moving from right to left. When such elements are found,
  - If low < high, exchange A[low] and A [high].
  - If low ≥ high, exchange A[pivot] and A[high]. This will split the list into two sublists and both sublists are sorted recursively.

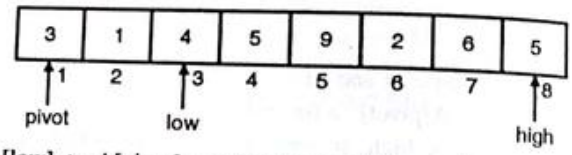
– Let's assume that first element of the list is the pivot. Initially, we will set low = pivot + 1 and high = last index.

Step 1 :



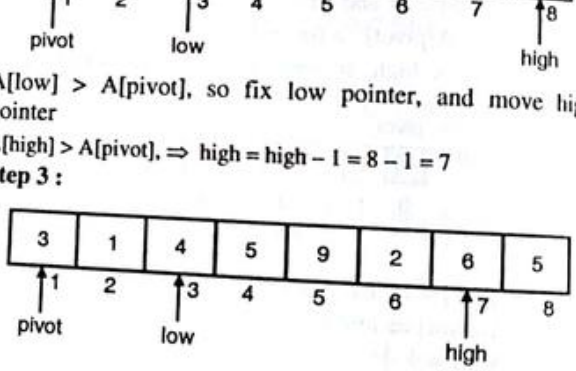
A[low] < A[pivot] ⇒ low = low + 1 = 2 + 1 = 3

Step 2 :

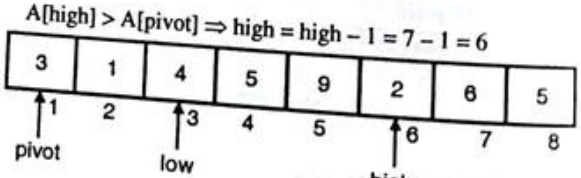


A[low] > A[pivot], so fix low pointer, and move high pointer
A[high] > A[pivot], ⇒ high = high – 1 = 8 – 1 = 7

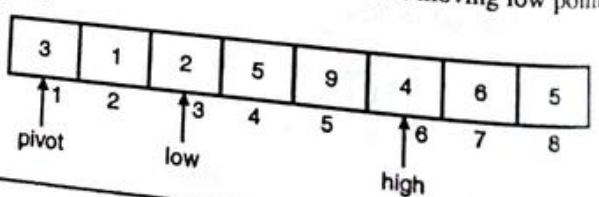
Step 3 :



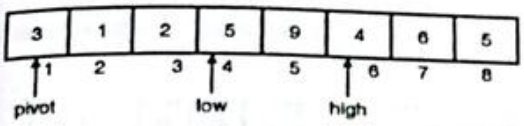
A[high] > A[pivot] ⇒ high = high – 1 = 7 – 1 = 6



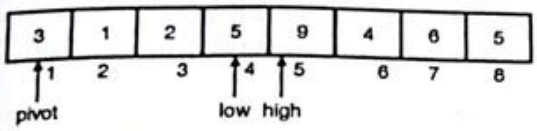
A[high] ≤ A[pivot], so fix high. As low < high exchange A[low] with A[high] and start moving low pointer again.



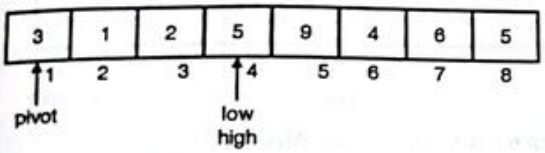
A[low] ≤ A[pivot], ⇒ low = low + 1 = 3 + 1 = 4



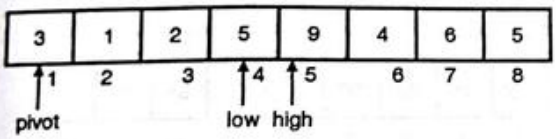
A[low] > A[pivot], so fix low and check for high
A[high] > A[pivot] ⇒ high = high – 1 = 6 – 1 = 5



A[high] > A[pivot] ⇒ high = high – 1 = 5 – 1 = 4

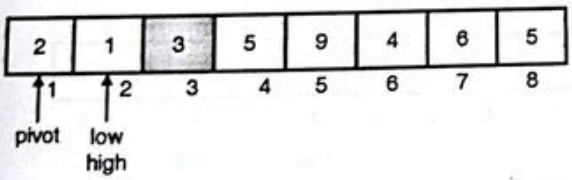


A[high] > A[pivot] ⇒ high = high – 1 = 4 – 1 = 3

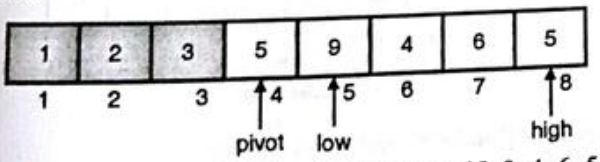


- A[high] ≤ A[pivot] and high < low, so exchange A[pivot] with A[high] and split the list into two sublists.
- Left sub list {2, 1} and right sub list {5, 9, 4, 6, 5} are recursively processed in similar way.

Reassign value of low and high in first sublist.

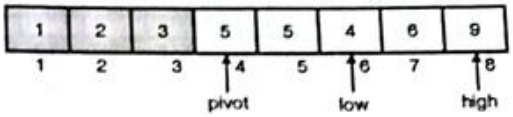


In first sub list {2, 1}, A[low] < A[pivot], but low cannot go ahead as it is pointing to last element in the list. So fix low and check for high. A[high] ≤ A[pivot], and high = low, so exchange A[high] and A[pivot] and divide the list. But new sub list has only one element so no need to sort it.

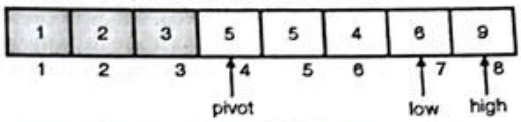


Repeat the procedure for second sublist {5, 9, 4, 6, 5}. Pivot is the first element in the list, low is next to it and high points to the last element.

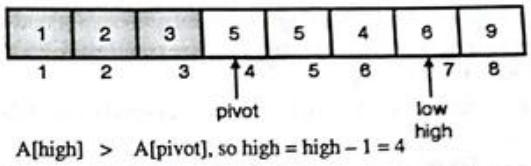
A[low] > A[pivot], so fix low and check for high. A[high] = A[pivot] and low < high, so exchange A[low] and A[high]. Increment low.



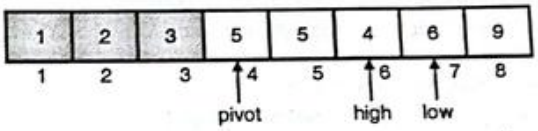
A[low] < A[pivot], so low = low + 1 = 7



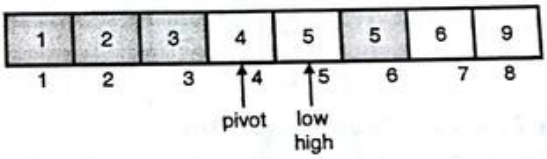
A[low] > A[pivot], so fix low and move high



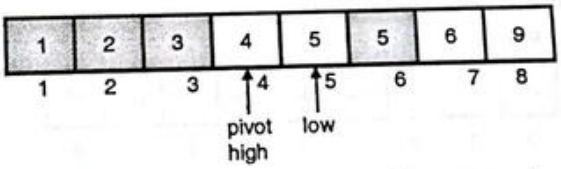
A[high] > A[pivot], so high = high – 1 = 4



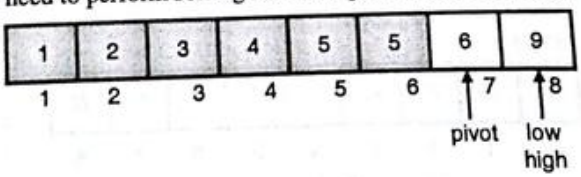
A[high] < A[pivot], and high < low, so exchange A[pivot] and A[high] and split list in two sub lists



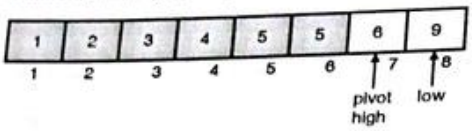
Repeat the same procedure for both new sub lists. In sub list {4, 5}, A[low] > A[pivot], so fix low, and check for high. A[high] > A[pivot], so high = high – 1 = 4



A[high] = A[pivot], and high < low, so exchange A[high] and A[pivot]. Pivot will set on correct location and divide the list into two sublists, one with element {5} and another list is null. As there is only one element in list {5}, no need to perform sorting on it. Set pointers on list {6, 9}



In sub list {4, 5}, A[low] > A[pivot], so fix low, and check for high. A[high] > A[pivot], so high = high – 1 = 7



– A[high] = A[pivot], and high < low, so exchange A[high] and A[pivot]. Pivot will set on correct location and divide the list into two sublists, one with element {9} and another list is null. As there is only one element in list {9}, no need to perform sorting on it. The list is sorted.

Ex. 2.3.3 **MU - May 2013. 10 Marks**

Sort following numbers using Quick sort algorithm. Show all passes of execution. Also state the time complexity. 65, 70, 75, 80, 85, 60, 55, 50, 45

Soln. : Simulation of the quick sort for given data is shown below :

Step 1 : Fix pivot point, low and high pointers.

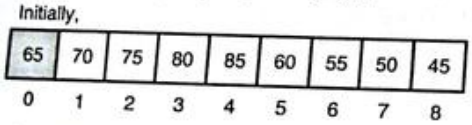
Keep moving low and high pointer until A[low] > A[pivot] and

A[high] ≤ A[pivot]. And if

low < high then swap A[low] and A[high]

low ≥ high then swap A[pivot] and A[high]

Initially,



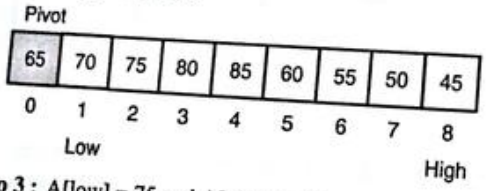
Step 2 : A[low] = 70 and A[pivot] = 65

A[low] > A[pivot] ⇒ fix low and check for high pointer

A[high] = 45 and A[pivot] = 65

A[high] ≤ A[pivot] ⇒ fix high and perform swap

Here, low < high, so swap A[low] with A[high] and again start moving low pointer

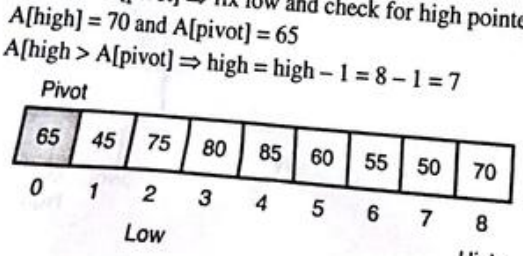


Step 3 : A[low] = 75 and A[pivot] = 65

A[low] > A[pivot] ⇒ fix low and check for high pointer

A[high] = 70 and A[pivot] = 65

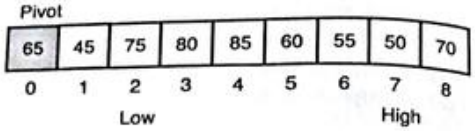
A[high > A[pivot] ⇒ high = high – 1 = 8 – 1 = 7



Step 4 : A[high] = 50 and A[pivot] = 65

A[high] ≤ A[pivot] ⇒ fix high and perform swap

Here, low < high, so swap A[low] with A[high] and again start moving low pointer

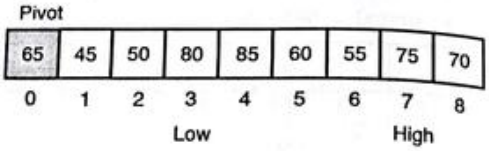


Step 5 : A[low] = 80 and A[pivot] = 65

A[low] > A[pivot] ⇒ fix low and check for high pointer

A[high] = 75 and A[pivot] = 65

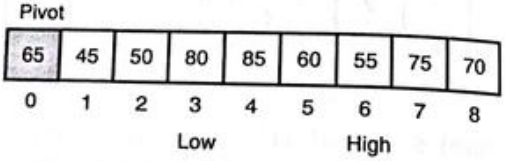
A[high > A[pivot] ⇒ high = high – 1 = 7 – 1 = 6



Step 6 : A[high] = 55 and A[pivot] = 65

A[high] ≤ A[pivot] ⇒ fix high and perform swap

Here, low < high, so swap A[low] with A[high] and again start moving low pointer

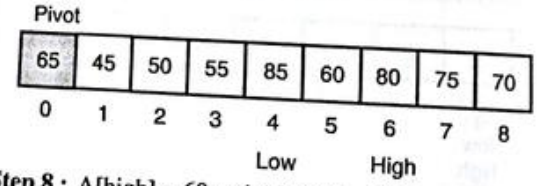


Step 7 : A[low] = 85 and A[pivot] = 65

A[low] > A[pivot] ⇒ fix low and check high

A[high] = 80 and A[pivot] = 65

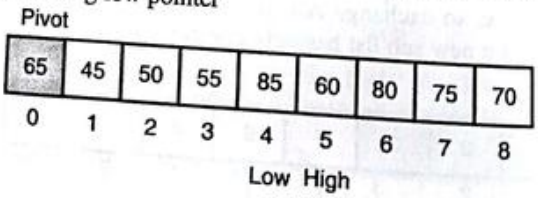
A[high] > A[pivot] ⇒ high = high – 1 = 6 – 1 = 5



Step 8 : A[high] = 60 and A[pivot] = 65

A[high] ≤ A[pivot] ⇒ fix high and perform swap

Here, low < high, so swap A[low] with A[high] and again start moving low pointer

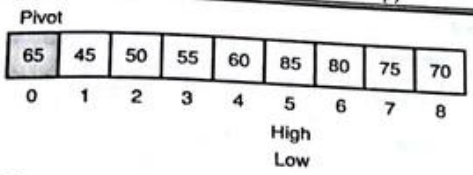


Step 9 : A[low] = 85 and A[pivot] = 65

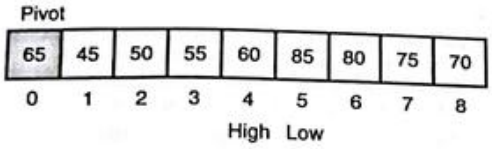
A[low] > A[pivot], so fix low and check high pointer

A[high] = 85 and A[pivot] = 65

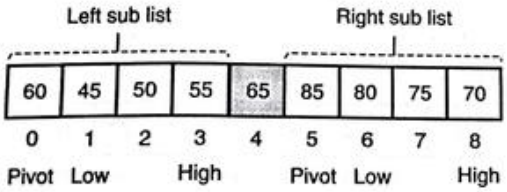
A[high] > A[pivot] ⇒ high = high – 1 = 5 – 1 = 4



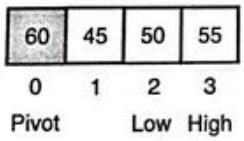
Step 10 :
A[high] = 60 and A[pivot] = 65
A[high] ≤ A[pivot] ⇒ so fix high and perform swap
Here, low ≥ high so swap A[pivot] with A[high], divide list in two sub lists and reset pointers. Recursively sort both the sub lists



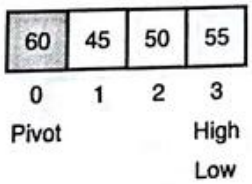
Step 11 : Let us first sort left sublist
A[low] = 45 and A[pivot] = 60
A[low] < A[pivot] ⇒ low = low + 1 = 1 + 1 = 2



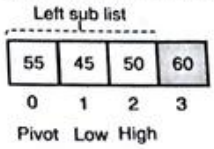
Step 12 :
A[low] = 50 and A[pivot] = 60
A[low] < A[pivot] ⇒ low = low + 1 = 2 + 1 = 3



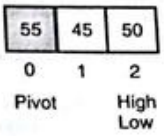
Step 13 : A[low] = 55 and A[pivot] = 60
A[low] < A[pivot] ⇒ low = low + 1 but not more elements are left in the list, so check high pointer
A[high] = 55 and A[pivot] = 60
A[high] ≤ A[pivot] ⇒ fix high and perform swap
Here, low == high, so swap A[pivot] with A[high] and split the list.



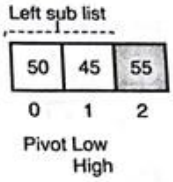
Step 14 : Right sub list is empty. For left sub list,
A[low] = 45 and A[pivot] = 55
A[low] < A[pivot], so low = low + 1 = 1 + 1 = 2



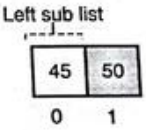
Step 15 :



Step 16 :

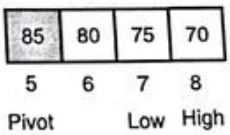


Step 17 :

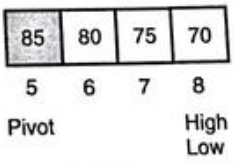


Left sublist shown in step 11 is now sorted. In a similar way, right sublist will be sorted now. The output of each step is shown.

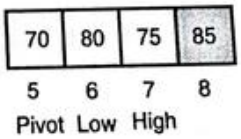
Step 18 :



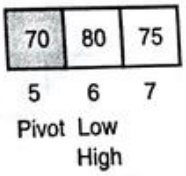
Step 19 :



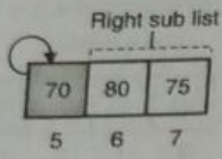
Step 20 :



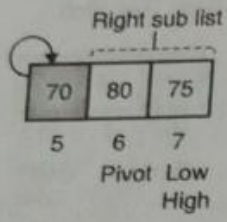
Step 21 :



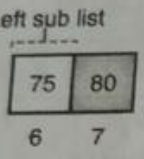
Step 22 :



Step 23 :



Step 24 :



Final sorted array would be :

45	50	55	60	65	70	75	80	85

– Gray cells indicate movement of the pivot.

Ex. 2.3.4 **MU - May 2016, 10 Marks**

Sort the following numbers using Quick Sort. Also, derive the time complexity of Quick Sort. 50, 31, 71, 38, 77, 81, 12, 33

Soln. :

Step 1 : Fix pivot point, low and high pointers.

Keep moving low and high pointer until A[low] > A[pivot] and A[high] ≤ A[pivot]. And if

low < high then swap A[low] and A[high]

low ≥ high then swap A[pivot] and A[high]

Initally,

50	31	71	38	77	81	12	33
0	1	2	3	4	5	6	7
Pivot	Low						High

Step 2 :

50	31	71	38	77	81	12	33
0	1	2	3	4	5	6	7
Pivot		Low					High

Step 3 :

50	31	33	38	77	81	12	71
0	1	2	3	4	5	6	7
Pivot			Low				High

Step 4 :

50	31	33	38	77	81	12	31
0	1	2	3	4	5	6	7
Pivot				Low			High

Step 5 :

50	31	33	38	77	81	12	31
0	1	2	3	4	5	6	7
Pivot				Low			High

Step 6 :

50	31	33	38	12	81	77	71
0	1	2	3	4	5	6	7
Pivot				Low		High	

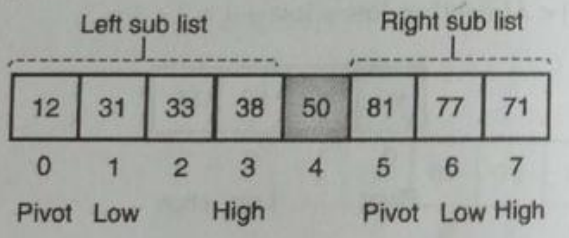
Step 7 :

50	31	33	38	12	81	77	71
0	1	2	3	4	5	6	7
Pivot					Low	High	

Step 8 :

50	31	33	38	12	81	77	71
0	1	2	3	4	5	6	7
Pivot					Low		
					High		

Step 9 :



Let first sort the left sublist

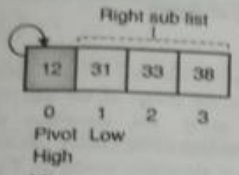
Step 10 :

12	31	33	38
0	1	2	3
Pivot	Low	High	

Step 11 :

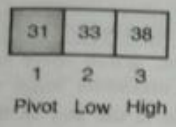
12	31	33	38
0	1	2	3
Pivot	Low		
	High		

Step 12 :

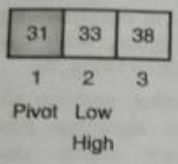


Lets sort this right sublist

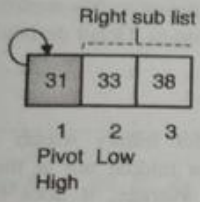
Step 13 :



Step 14 :

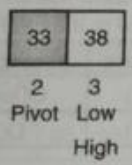


Step 15 :

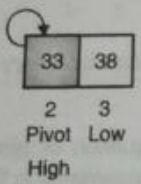


Let us solve this right sublist first

Step 16 :

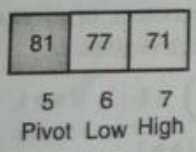


Step 17 :

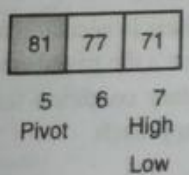


Left sub list of step 9 is sorted. Let's sort the right sublist generated in step 9.

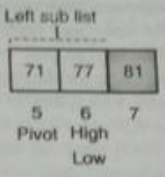
Step 18 :



Step 19 :

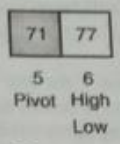


Step 20 :

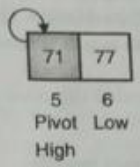


Let us sort left sublist.

Step 21 :

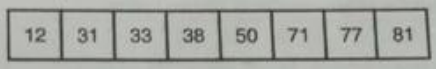


Step 22 :



The list is sorted.

The final array will be:



Syllabus Topic : Analysis of Binary Search

2.4 Binary Search

→ (Dec. 15, May 17)

Q. Implement the binary search and derive its complexity. **MU - Dec. 2015, 10 Marks**

Q. Write the algorithm and derive the complexity of Binary Search Algorithm. **MU - May 2017, 5 Marks**

Q. Explain binary search using divide and conquer approach. **(6 Marks)**

- Searching is very common operation in computer science applications. Searching is the problem of finding an element from given set of data. We will discuss two approaches, linear search, and binary search. Binary search is divide and conquer approach. The discussion will prove that binary search outperforms linear search.
- Binary search is efficient than linear search. For binary search, the array must be sorted, which is not required in case of linear search.
- Binary search is divide and conquer based search technique. In each step, algorithm divides the list into two halves and check if the element to be searched is on upper or lower half of the array. If the element is found on middle, algorithm returns.

- Let us assign minimum and maximum index of the array to variables *low* and *high*, respectively. The middle index is computed as *(low + high)/2*.
- In every iteration, the algorithm compares the middle element of the array with a key to be searched. Initial range of search is A[0] to A[n - 1]. When the key is compared with A [*mid*], there are three possibilities :

1. The array is already sorted, so if key < A[mid] then key cannot be present in bottom half of the array. Search space for problem will be reduced by setting the *high* index to (mid – 1). New search range would be A[*low*] to A[*mid* – 1].
2. If key > A[*mid*] then the key cannot be present in upper half of the array. Search space for problem will be reduced by moving the *low* index to (mid + 1). New search range would be A[*mid* + 1] to A[*high*]
3. If key = A[*mid*], the search is successful and algorithm halts.

This process is repeated till index *low* is less than *high* or element is found.

☞ Algorithm for binary search

Q. Write an algorithm for binary search. (3 Marks)

Algorithm for binary search is shown below :

```
Algorithm BINARY_SEARCH(A, Key)
// Description : Perform binary search on array A
// Input : Sorted array A of size n and Key to be searched
// Output : Success / Failure

low ← 1
high ← n
while low < high do
    mid ← (low + high) / 2
    if A[mid] == key then
        return mid
    else if A[mid] < key then
        low ← mid + 1
    else
        high ← mid – 1
    end
end

return 0
```

- Binary search reduces search space by half in every iteration. In a linear search, search space was reduced by one only.
- If there are n elements in the array, binary search, and linear search has to search among (n / 2) and (n – 1) elements respectively in the second iteration. In the third iteration, the binary search has to scan only (n / 4) elements, whereas linear search has to scan (n – 2) elements. This shows that binary search would hit the bottom very quickly.

☞ Complexity analysis

Best case

In binary search, the key is first compared with the middle element of the array. If the key is in the middle position of the array, the algorithm does only one comparison, irrespective of the size of the array. Hence, best case running time of algorithm would be, T(n) = 1.

Worst case

Q. Find worst-case efficiency of binary search. (4 Marks)

In every iteration, search space of binary search is reduced by half, so maximum $\log_2 n$ array divisions are possible. If the key is at the leaf of the tree or it is not present at all, then algorithm does $\log_2 n$ comparisons, which is maximum. Numbers of comparisons grow in logarithmic order of input size. Hence, worst case running time of algorithm would be, $T(n) = O(\log_2 n)$.

After every iteration, problem size is reduced by a factor of 2, and the algorithm does one comparison. Recurrence of binary search can be written as $T(n) = T(n/2) + 1$. Solution to this recurrence leads to same running time, i.e. $O(\log_2 n)$. Detail derivation is discussed in Ex. 2.4.1.

Average case

Average case for binary search occurs when key element is neither on middle nor at the leaf level of the search tree. On an average, it does half of the $\log_2 n$ comparisons, which will turn out as $T(n) = O(\log_2 n)$.

The complexity of linear search and binary search for all three cases is compared in the following table.

	Best case	Average case	Worst case
Binary Search	O(1)	$O(\log_2 n)$	$O(\log_2 n)$
Linear Search	O(1)	O(n)	O(n)

Ex. 2.4.1

Write the recurrence relation for binary search and solve it.

OR

Write the recurrence equation of binary search and prove that running time of the recurrence is $O(\log_2 n)$.

Soln. :

In every iteration, binary search does one comparison and creates a new problem of size n/2. So recurrence equation of binary search is given as,

$$T(n) = T\left(\frac{n}{2}\right) + 1, \quad \text{if } n > 1$$

$$T(n) = 1, \quad \text{if } n = 1$$

Only one comparison is needed when there is only one element in the array, that's the trivial case.

This is the boundary condition for recurrence. Let us solve this by iterative approach,

$$T(n) = T\left(\frac{n}{2}\right) + 1 \quad \ldots(1)$$

Substitute n by n/2 in Equation (1) to find T(n/2)

$$T\left(\frac{n}{2}\right) = T\left(\frac{n}{4}\right) + 1 \quad \ldots(2)$$

Substitute value of T(n/2) in Equation (1),

$$\therefore T(n) = T\left(\frac{n}{2^2}\right) + 1 \quad \ldots(3)$$

Substitute n by n/2 in Equation (2) to find T(n/4),

$$T\left(\frac{n}{4}\right) = T\left(\frac{n}{8}\right) + 1$$

Substitute value of T(n/4) in Equation (3),

$$\therefore T(n) = T\left(\frac{n}{2^3}\right) + 3$$

:

:

After k iterations,

$$\therefore T(n) = T\left(\frac{n}{2^k}\right) + k$$

Suppose, $n = 2^k$, So $k = \log_2 n$

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

$$= T(1) + k$$

So, $T(n) = 1 + k = 1 + \log_2 n$

$T(n) = O(\log_2 n)$

Ex. 2.4.2

Find element 33 from array A {11, 22, 33, 44, 55, 66, 77, 88} using binary search.3

Soln. :

The array is already sorted, so we can directly apply binary search on A. Here Key = 33.

Iteration 1: Initial search space

11	22	33	44	55	66	77	88
1	2	3	4	5	6	7	8

low = 1, high = 8,

mid = (low + high) / 2 = (1 + 8)/ 2 = 9/2 = 4

A[mid] = A[4] = 44

As Key < A[4], update high

high = mid – 1 = 4 – 1 = 3

Iteration 2 : New search space

11	22	33
1	2	3

low = 1, high = 3,

mid = (low + high) / 2

= (1 + 3)/ 2= 4/2 = 2

A[mid] = A[2] = 22

As Key > A[2], update low

low = mid + 1= 2 + 1 = 3

Iteration 3 : New search space

low = 3, high = 3,

mid = (low + high) / 2 = (3 + 3)/ 2 = 6/2 = 3

A[mid] = A[3] = 33

Key = A[3],

Element found, so return mid

33
3

Ex. 2.4.3

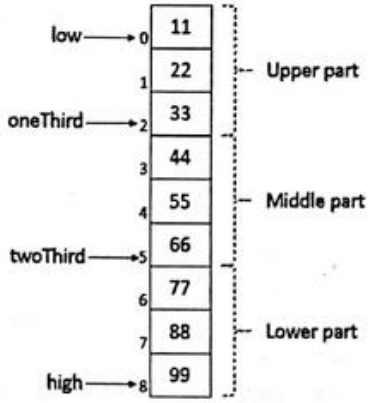
Consider the binary search algorithm so that it splits the input not into two sets of almost equal sizes, but into three sets of sizes approximately one third, write down the recurrence for this ternary search algorithm and find the asymptotic complexity of this algorithm.

OR

A binary search splits an array into two parts. If it splits an array into three parts, write down recurrence relation and complexity.

Soln. :

Ternary search divides the array into three parts, so instead of mid index, we would compute oneThird and twoThird indices of the array as shown in the figure. The array is divide in three parts: upper, middle and lower.



low = 0

high = 8

oneThird = (low + high) / 3 = (0+8) / 3 = 2

two Third = 2/3 * (high - low) / 3 = 2/3*(8) / 3 = 5

- Initially, *low* and *high* indices are pointing to minimum and maximum index of an array.
- If the key is less than the A[oneThird], then it will be definitely in the upper part of the array, so *high* index is updated to oneThird – 1.
- If key is greater than A[twoThird], then it will be definitely in lower part of the array, so *low* index is updated to twoThird + 1
- And if the key is between A[oneThird] and A[twoThird] then *low* and *high* indices will be updated to (oneThird + 1) and (twoThird - 1) respectively.

- Pseudo code for ternary search is given below :

```
Algorithm TERNARY_SEARCH(A, Key)
// Description: Perform ternary search on array A
// Input: Sorted array A of size n and Key to be searched
// Output: Success / Failure

low ← 1
high ← n

while low ≤ high do
    oneThird ← floor(high – low) / 3
    twoThird ← floor(2/3 * (high - low))

    if A[oneThird] == key then
        return oneThird
    else if A[twoThird] == key then
        return twoThird
    else if key < A[oneThird] then
        high ← oneThird - 1
    else if key > A[oneThird] && key < A[twoThird] then
        low ← oneThird + 1
        high ← twoThird - 1
    else
        low ← twoThird + 1
    end
end
return 0
```

Key is in upper part

Key is in middle part

Key is in lower part

There are two cases of ternary search:

Case 1 : Two comparisons fix the position in one of the parts of the array, and the size of the new problem would be n/3. Recurrence for this case would be $T(n) = T(n/3) + 2$

Case 2 : After single comparison, list is divided into two parts of size 1/3 and 2/3 respectively. In the worst case, the key will be in the larger part of size 2/3. Recurrence for this case would be $T(n) = T(2n/3) + 1$.

Solving the recurrence $T(n) = T(2n/3) + 1$.

The recurrence $T(n) = T(2n/3) + 1$ is of the form $T(n) = aT(n/b) + f(n)$, with $a = 1$, $b = 3/2$ and $f(n) = 1$ with degree of polynomial $d = 0$.

$b^d = (3/2)^0 = 1$

Here, $a = b^d$, so from master method $T(n) = \Theta(n^d \log n)$

$= \Theta(n^0 \log n)$

$= \Theta(\log n)$

Thus, worst case running time of ternary search is $\Theta(\log n)$.

Ex. 2.4.4 MU - May 2015, 10 Marks

Suppose you are given n number of coins, in that one coin is faulty, its weight is less than standard coin weight. To find the faulty coin in a list using the proper searching method. What will be the complexity of the searching method?

Soln. :

- We will discuss two cases of coin arrangement.

Case I : List of coin is sorted according to their weight

(A) List is sorted in ascending order of weight

- In this case, there is no need of sorting the list. We can directly perform the binary search. Binary search divides the list into halves in each iteration. In the worst case, it performs O(logn) comparisons.
- But the linear search needs only one comparison for searching as the faulty coin is in the first position, so it takes O(1) time to find the faulty coin.

(B) List is sorted in descending order of weight

- List needs to be sorted first for binary search, which takes O(nlong) time. And binary search itself runs in O(logn) time. Over all, it runs in O(nlogn + logn) time.
- This would be the worst case for linear search as the faulty coin is at the end of list, and it will take O(n) time.

Case II : List of coin is not sorted according to their weight

- To perform the searching using binary search, we first need to sort the list. Divide and conquer based sorting techniques (like merge sort, quick sort) takes nlogn time for sorting.
- The binary search runs in logn time, so overall finding faulty coin takes O(nlogn + logn) time. Linear search does not require sorting of data, in the worst case, it performs O(n) comparisons.
- So in any case, linear search outperforms binary search in particular given case.

Syllabus Topic : Finding Minimum and Maximum Algorithm and Analysis

2.5 Min-Max Problem

➔ **(May 14, Dec. 14, May 16, May 17)**

Q. Write a Min-Max function to find minimum and maximum value from given a set of values using divide and conquer. Also, drive its complexities.
MU - May 2014, Dec. 2014, 10 Marks

Q. Write an algorithm to find the minimum and maximum value using divide and conquer and also derive its complexity.
MU - May 2016, 10 Marks

Q. Write an algorithm for finding maximum and minimum number from given set.
MU - May 2017, 5 Marks

Q. Design and analyze a divide and conquer algorithm for finding minimum and maximum number in the array of n-numbers that uses (3n/2) – 2 comparisons for any n.
(6 Marks)

Q. Design and analyze a divide and conquer algorithm for finding a maximum and minimum element in a list L[1 : n].
(6 Marks)

- Max Min problem is to find a maximum and minimum element from given array.
- In the traditional approach, the maximum and minimum element can be found by comparing each element and updating Max and Min values as and when required.
- This approach is simple but it does (n – 1) comparisons for finding max and the same number of comparisons for finding the min. It results in total 2(n – 1) comparisons.
- Using divide and conquer approach, we can reduce the number of comparisons.
- Divide and conquer approach for Max. Min problem works in three stages.
  - If a_1 is the only element in the array, a_1 is the maximum and minimum.
  - If the array contains only two elements a_1 and a_2, then the single comparison between two elements can decide minimum and maximum of them.
  - If there are more than two elements, algorithm divides the array from middle and creates two subproblems. Both subproblems are treated as an independent problem and the same recursive process is applied to them. This division continues until subproblem size becomes one or two.
- After solving two subproblems, their minimum and maximum numbers are compared to build the solution of the large problem. This process continues in a bottom-up fashion to build solution of a parent problem.

```
Algorithm DC_MAXMIN (A, low, high)
// Description: Find minimum and maximum element from
     array using divide and conquer approach
// Input: Array A of length n, and indices low = 0 and high =
     n - 1
// Output: (min, max) variables holding minimum and
     maximum element of array
if n == 1 then                          // Array contain only one element
    return (A[1], A[1])
else if n == 2 then                     // Array contain only two elements
    if A[1] < A[2] then
        return (A[1], A[2])
    else
        return (A[2], A[1])
else                                    // Solve left sub problem
    mid ← (low + high) / 2
    [LMin, LMax] = DC_MAXMIN (A, low, mid)
    [RMin, RMax] = DC_MAXMIN (A, mid + 1, high)
                                        // Solve right sub problem
    If LMax > RMax then                 // Combine solution
        max ← LMax
    else
        max ← RMax
    end
    if LMin < RMin then                 // Combine solution
        min ← LMin
    else
        min ← RMin
    end
    return (min, max)
end
```

☞ **Complexity analysis**

The conventional algorithm takes 2(n – 1) comparisons in worst, best and average case.

DC_MAXMIN does two comparisons two determine minimum and maximum element and creates two problems of size n/2, so the recurrence can be formulated as

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

Let us solve this equation using interactive approach.

$$T(n) = 2T\left(\frac{n}{2}\right) + 2 \qquad \ldots(2.5.1)$$

By substituting n by (n / 2) in Equation (2.5.1)

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + 2$$

$$T(n) = 2\left[2T\left(\frac{n}{4}\right) + 2\right] + 2$$

$$= 4T\left(\frac{n}{4}\right) + 4 + 2 \qquad \ldots(2.5.2)$$

By substituting n by $\frac{n}{4}$ in Equation (2.5.1),

$$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + 2$$

Substitute it in Equation (2.5.2),

$$\therefore T(n) = 4\left[2T\left(\frac{n}{8}\right) + 2\right] + 4 + 2$$

$$= 8T\left(\frac{n}{8}\right) + 8 + 4 + 2$$

$$= 2^3 T\left(\frac{n}{2^3}\right) + 2^3 + 2^2 + 2^1$$

$$\vdots$$

After k – 1 iterations

$$\therefore T(n) = 2^{k-1} T\left(\frac{n}{2^{k-1}}\right) + 2^{k-1} + 2^{k-2} + \ldots + 2^3 + 2^2 + 2^1$$

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

$= 2^{k-1} T\left(\frac{n}{2^{k-1}}\right) + (2^k - 2)$ (Simplifying series)

Let $n = 2^k \Rightarrow 2^{k-1} = \left(\frac{n}{2}\right)$

$\therefore T(n) = \left(\frac{n}{2}\right) T\left(\frac{2^k}{2^{k-1}}\right) + (n-2)$

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

For n = 2, T(n) = 1 (two elements require only 1 comparison to determine min-max)

$T(n) = \left(\frac{n}{2}\right) + (n-2) = \left(\frac{3n}{2} - 2\right)$

It can be observed that divide and conquer approach does only $\left(\frac{3n}{2} - 2\right)$ comparisons compared to 2(n − 1) comparisons of the conventional approach.

For any random pattern, this algorithm takes the same number of comparisons.

Syllabus Topic : Strassen's Matrix Multiplication

2.6 Strassen's Matrix Multiplication

➔ **(Dec. 13, May 15, Dec. 15, May 16, May 17)**

Q. Write a short note on Strassen's matrix multiplication.
MU - Dec. 2013, May 2015, May 2016, May 2017, 10 Marks

Q. Explain the Strassen's Matrix multiplication.
MU - Dec. 2015, 10 Marks

Strassen has proposed divide and conquer strategy based algorithm, which takes less numbers of multiplications compare to this traditional way of matrix multiplication. Using Strassen's method, multiplication operation is defined as,

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$C_{11} = S_1 + S_4 - S_5 + S_7$

$C_{12} = S_3 + S_5$

$C_{21} = S_2 + S_4$

$C_{22} = S_1 + S_3 - S_2 + S_6$

Where,

$S_1 = (A_{11} + A_{22}) * (B_{11} + B_{22})$

$S_2 = (A_{21} + A_{22}) * B_{11}$

$S_3 = A_{11} * (B_{12} - B_{22})$

$S_4 = A_{22} * (B_{21} - B_{11})$

$S_5 = (A_{11} + A_{12}) * B_{22}$

$S_6 = (A_{21} - A_{11}) * (B_{11} + B_{12})$

$S_7 = (A_{12} - A_{22}) * (B_{21} + B_{22})$

Let us check if it same as conventional approach.

$C_{12} = S_3 + S_5 = A_{11} * (B_{12} - B_{22}) + (A_{11} + A_{12}) * B_{22}$

$= A_{11} * B_{12} - A_{11} * B_{22} + A_{11} * B_{22} + A_{12} * B_{22}$

$= A_{11} * B_{12} + A_{12} * B_{22}$

This is same as C_{12} derived using conventional approach. Similarly we can derive all C_{ij} for Strassen's matrix multiplication. Algorithm for Strassen's multiplication is shown below :

```
Algorithm STRASSEN_MAT_MUL (int *A, int *B, int *C,
    int n)
// A and B are input matrices
// C is the output matrix
// All matrices are of size n × n

if n == 1 then
    *C = *C + (*A) * (*B)
else
    STRASSEN_MAT_MUL (A, B, C, n/4)
    STRASSEN_MAT_MUL (A, B + (n/4), C + (n/4), n/4)
    STRASSEN_MAT_MUL (A + 2 * (n/4), B, C + 2 * (n/4),
     n/4)
    STRASSEN_MAT_MUL (A + 2 * (n/4), B + (n/4),
     C + 3 * (n/4), n/4)
    STRASSEN_MAT_MUL (A + (n/4), B + 2 * (n/4), C,
     n/4)
    STRASSEN_MAT_MUL (A + (n/4), B + 3 * (n/4),
     C + (n/4), n/4)
    STRASSEN_MAT_MUL (A + 3 * (n/4), B + 2 * (n/4),
     C + 2 * (n/4), n/4)
    STRASSEN_MAT_MUL (A + 3 * (n/4), B + 3 * (n/4),
     C + 3 * (n/4), n/4)
end
```

☞ **Complexity Analysis**

Conventional approach performs eight multiplications to multiply two matrices of size 2 × 2. Whereas Strassen's approach performs seven multiplications on problem of size 1 × 1, which in turn finds the multiplication of 2 × 2 matrices using addition. In short, to solve problem of size n, Strassen's approach creates seven problems of size (n − 2). Recurrence equation for Strassen's approach is given as,

$$T(n) = 7.T\left(\frac{n}{2}\right)$$

Two matrices of size 1 × 1 needs only one multiplication, so base case would be, T (1) = 1.

Let us find the solution using iterative approach. By substituting n = (n / 2) in above equation,

$$T\left(\frac{n}{2}\right) = 7.T\left(\frac{n}{4}\right)$$

$$\therefore T(n) = 7^2.T\left(\frac{n}{2^2}\right)$$

.
.
.

$$T(n) = 7^k.T\left(\frac{n}{2^k}\right)$$

Let's assume

$$n = 2^k$$

$\therefore k = \log_2 n$

$$T(2^k) = 7^k . T\left(\frac{2^k}{2^k}\right) = 7^k . T(1) = 7^k = 7^{\log_2 n}$$
$$= n^{\log_2 7} = n^{2.81} < n^3$$

Thus, running time of Strassen's matrix multiplication algorithm $O(n^{2.81})$, which is less then cubic order of traditional approach. The difference between running time becomes significant when n is large.

Example of Strassen's matrix multiplication is discussed below:

Ex. 2.6.1

Multiply given two matrices A and B using Strassen's approach, where

$$A = \begin{bmatrix} 0 & 0 & 3 & 4 \\ 5 & 6 & 0 & 0 \\ 0 & 0 & 6 & 5 \\ 4 & 3 & 0 & 0 \end{bmatrix}, \text{ and } B = \begin{bmatrix} 0 & 0 & 6 & 5 \\ 4 & 3 & 0 & 0 \\ 0 & 0 & 3 & 4 \\ 5 & 6 & 0 & 0 \end{bmatrix}$$

Soln. :

$$\text{Let } C = A \times B = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$C = \begin{bmatrix} \begin{bmatrix} 0 & 0 \\ 5 & 6 \end{bmatrix} & \begin{bmatrix} 3 & 4 \\ 0 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 \\ 4 & 3 \end{bmatrix} & \begin{bmatrix} 6 & 5 \\ 0 & 0 \end{bmatrix} \end{bmatrix} \times \begin{bmatrix} \begin{bmatrix} 0 & 0 \\ 4 & 3 \end{bmatrix} & \begin{bmatrix} 6 & 5 \\ 0 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 \\ 5 & 6 \end{bmatrix} & \begin{bmatrix} 3 & 4 \\ 0 & 0 \end{bmatrix} \end{bmatrix}$$

$$S_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22})$$
$$= \left(\begin{bmatrix} 0 & 0 \\ 5 & 6 \end{bmatrix} + \begin{bmatrix} 6 & 5 \\ 0 & 0 \end{bmatrix}\right) \times \left(\begin{bmatrix} 0 & 0 \\ 4 & 3 \end{bmatrix} + \begin{bmatrix} 3 & 4 \\ 0 & 0 \end{bmatrix}\right)$$
$$= \begin{bmatrix} 6 & 5 \\ 5 & 6 \end{bmatrix} \times \begin{bmatrix} 3 & 4 \\ 4 & 3 \end{bmatrix}$$
$$= \begin{bmatrix} 18+20 & 24+15 \\ 15+24 & 20+18 \end{bmatrix} = \begin{bmatrix} 38 & 39 \\ 39 & 38 \end{bmatrix}$$

$$S_2 = (A_{21} + A_{22}) * B_{11}$$
$$= \left(\begin{bmatrix} 0 & 0 \\ 4 & 3 \end{bmatrix} + \begin{bmatrix} 6 & 5 \\ 0 & 0 \end{bmatrix}\right) \times \begin{bmatrix} 0 & 0 \\ 4 & 3 \end{bmatrix}$$
$$= \begin{bmatrix} 6 & 5 \\ 4 & 3 \end{bmatrix} \times \begin{bmatrix} 0 & 0 \\ 4 & 3 \end{bmatrix} = \begin{bmatrix} 20 & 15 \\ 12 & 9 \end{bmatrix}$$

$$S_3 = A_{11} * (B_{12} - B_{22})$$
$$= \begin{bmatrix} 0 & 0 \\ 5 & 6 \end{bmatrix} \times \left(\begin{bmatrix} 6 & 5 \\ 0 & 0 \end{bmatrix} - \begin{bmatrix} 3 & 4 \\ 0 & 0 \end{bmatrix}\right)$$
$$= \begin{bmatrix} 0 & 0 \\ 5 & 6 \end{bmatrix} \times \begin{bmatrix} 3 & 1 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 15 & 5 \end{bmatrix}$$

$$S_4 = A_{22} * (B_{21} - B_{11})$$
$$= \begin{bmatrix} 6 & 5 \\ 0 & 0 \end{bmatrix} \times \left(\begin{bmatrix} 0 & 0 \\ 5 & 6 \end{bmatrix} - \begin{bmatrix} 0 & 0 \\ 4 & 3 \end{bmatrix}\right)$$
$$= \begin{bmatrix} 6 & 5 \\ 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 0 \\ 1 & 3 \end{bmatrix} = \begin{bmatrix} 5 & 15 \\ 0 & 0 \end{bmatrix}$$

$$S_5 = (A_{11} + A_{12}) * B_{22}$$
$$= \left(\begin{bmatrix} 0 & 0 \\ 5 & 6 \end{bmatrix} + \begin{bmatrix} 3 & 4 \\ 0 & 0 \end{bmatrix}\right) \times \begin{bmatrix} 3 & 4 \\ 0 & 0 \end{bmatrix}$$
$$= \begin{bmatrix} 3 & 4 \\ 5 & 6 \end{bmatrix} \times \begin{bmatrix} 3 & 4 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 9 & 12 \\ 15 & 20 \end{bmatrix}$$

$$S_6 = (A_{21} - A_{11}) * (B_{11} + B_{12})$$
$$= \left(\begin{bmatrix} 0 & 0 \\ 4 & 3 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 5 & 6 \end{bmatrix}\right) \times \left(\begin{bmatrix} 0 & 0 \\ 4 & 3 \end{bmatrix} + \begin{bmatrix} 6 & 5 \\ 0 & 0 \end{bmatrix}\right)$$
$$= \begin{bmatrix} 0 & 0 \\ -1 & -3 \end{bmatrix} \times \begin{bmatrix} 6 & 5 \\ 4 & 3 \end{bmatrix}$$
$$= \begin{bmatrix} 0 & 0 \\ -18 & -14 \end{bmatrix}$$

$$S_7 = (A_{12} - A_{22}) * (B_{21} + B_{22})$$
$$= \left(\begin{bmatrix} 3 & 4 \\ 0 & 0 \end{bmatrix} - \begin{bmatrix} 6 & 5 \\ 0 & 0 \end{bmatrix}\right) \times \left(\begin{bmatrix} 0 & 0 \\ 5 & 6 \end{bmatrix} + \begin{bmatrix} 3 & 4 \\ 0 & 0 \end{bmatrix}\right)$$
$$= \begin{bmatrix} -3 & -1 \\ 0 & 0 \end{bmatrix} \times \begin{bmatrix} 3 & 4 \\ 5 & 6 \end{bmatrix}$$
$$= \begin{bmatrix} -14 & -18 \\ 0 & 0 \end{bmatrix}$$

$$C_{11} = S_1 - S_4 - S_5 + S_7$$
$$= \begin{bmatrix} 38 & 39 \\ 39 & 38 \end{bmatrix} + \begin{bmatrix} 5 & 15 \\ 0 & 0 \end{bmatrix} - \begin{bmatrix} 9 & 12 \\ 15 & 20 \end{bmatrix} + \begin{bmatrix} -14 & -18 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 20 & 24 \\ 24 & 18 \end{bmatrix}$$

$$C_{12} = S_3 + S_5 = \begin{bmatrix} 0 & 0 \\ 15 & 5 \end{bmatrix} + \begin{bmatrix} 9 & 12 \\ 15 & 20 \end{bmatrix}$$
$$= \begin{bmatrix} 9 & 30 \\ 15 & 25 \end{bmatrix}$$

$$C_{21} = S_2 + S_4 = \begin{bmatrix} 20 & 15 \\ 12 & 9 \end{bmatrix} + \begin{bmatrix} 5 & 15 \\ 0 & 0 \end{bmatrix}$$
$$= \begin{bmatrix} 25 & 30 \\ 12 & 9 \end{bmatrix}$$

$$C_{22} = S_1 + S_3 - S_2 + S_6$$
$$= \begin{bmatrix} 38 & 39 \\ 39 & 38 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 15 & 5 \end{bmatrix} - \begin{bmatrix} 20 & 15 \\ 12 & 9 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ -18 & -14 \end{bmatrix} = \begin{bmatrix} 18 & 24 \\ 24 & 20 \end{bmatrix}$$

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$
$$= \begin{bmatrix} \begin{bmatrix} 20 & 24 \\ 24 & 18 \end{bmatrix} & \begin{bmatrix} 9 & 12 \\ 30 & 25 \end{bmatrix} \\ \begin{bmatrix} 25 & 30 \\ 12 & 9 \end{bmatrix} & \begin{bmatrix} 18 & 24 \\ 24 & 20 \end{bmatrix} \end{bmatrix}$$
$$= \begin{bmatrix} 20 & 24 & 9 & 12 \\ 24 & 18 & 30 & 25 \\ 25 & 30 & 18 & 24 \\ 12 & 9 & 24 & 20 \end{bmatrix}$$

2.7 Exam Pack (University and Review Questions)

☞ **Syllabus Topic : General Method**

Q. Explain Divide and Conquer strategy. List any Four examples that can be solved by divide and conquer. *(Ans. : Refer section 2.1)* (6 Marks)

(May 2013, Dec. 2013)

Q. Comment on the module of computation : Divide and Conquer. *(Ans. : Refer section 2.1)*(5 Marks) **(May 2014)**

Q. Explain the concept of divide and conquer. *(Ans. : Refer section 2.1.1)*(5 Marks)

Q. Enlist few problems that can be solved using divide and conquer approach. *(Ans. : Refer section 2.1.1)*(3 Marks)

Q. Write control abstraction (General method) for divide and conquer. *(Ans. : Refer section 2.1.2)* (4 Marks) **(May 2013, Dec. 2013)**

Q. Derive a general equation to find the complexity of divide and conquer approach. *(Ans. : Refer section 2.1.3)* (5 Marks)

Q. Which are different factors considered for sorting elements. *(Ans. : Refer section 2.1.4)* (5 Marks) **(May 2015)**

☞ **Syllabus Topic : Analysis of Merge Sort**

Q. Analyze merge sort and find time complexity of merge sort. *(Ans. : Refer section 2.2)*(5 Marks)

Q. Given a sequence of n-elements A[1] ..... A[n], assume that they are split into 2 sets A[1] ..... A[n/2] and A[n/2 + 1] ... A[n], each set is individually sorted and the resulting sequence is merged to produce a single sorted sequence of n elements.
Using the divide and conquer strategy, write a Merge sort algorithm to sort the sequence in non-decreasing order.
(Ans. : Refer section 2.2) (6 Marks)

Q. Write an algorithm for sorting 'n' numbers using merge sort. *(Ans. : Refer section 2.2)*(6 Marks)

Q. State time complexity of merge sort. *(Ans. : Refer section 2.2)* (4 Marks)

Q. Formulate recurrence for merge sort and find the time complexity of it. *(Ans. : Refer section 2.2)*(4 Marks)

☞ **Syllabus Topic : Analysis of Quick Sort**

Q. Explain quick sort algorithm. Evaluate complexity with suitable example. *(Ans. : Refer section 2.3)* (10 Marks)

Q. Explain quick sort algorithm using divide and conquer approach. *(Ans. : Refer section 2.3)* (6 Marks)

Q. Discuss partition exchange sort and analyze it. *(Ans. : Refer section 2.3)* (5 Marks)

Q. Write an algorithm for quick sort. *(Ans. : Refer section 2.3)* (3 Marks)

Q. Derive the best case time complexity of quick sort. *(Ans. : Refer section 2.3)*(6 Marks)

Q. Prove that for the Quick sort: Best Case efficiency is T(N) = O(nlog n) *(Ans. : Refer section 2.3)*(5 Marks)

Q. Explain randomized version of Quick sort and derive its complexity. *(Ans. : Refer section 2.3)* (10 Marks) **(May 2015)**

Q. Explain randomized quick sort. *(Ans. : Refer section 2.3)* (3 Marks)

Q. How to achieve O(n log n) time complexity in the worst case for quick sort. *(Ans. : Refer section 2.3)* (3 Marks)

Q. Why quick sort is better than merge sort? *(Ans. : Refer section 2.3)*(3 Marks)

Ex. 2.3.3 (10 Marks) **(May 2013)**

Ex. 2.3.4 (10 Marks) **(May 2016)**

☞ **Syllabus Topic : Analysis of Binary Search**

Q. Implement the binary search and derive its complexity. *(Ans. : Refer section 2.4)*(10 Marks)

Q. Write the algorithm and derive the complexity of Binary Search Algorithm. *(Ans. : Refer section 2.4)* (5 Marks) **(May 2017)**

Q. Explain binary search using divide and conquer approach. (Ans. : Refer section 2.4) (6 Marks)

Q. Find worst-case efficiency of binary search. *(Ans. : Refer section 2.4)*(4 Marks)

Ex. 2.4.4 (10 Marks) **(May 2015)**

☞ **Syllabus Topic : Finding Minimum and Maximum Algorithm and Analysis**

Q. Write a Min-Max function to find minimum and maximum value from given a set of values using divide and conquer. Also, drive its complexities. *(Ans. : Refer section 2.5)* (10 Marks) **(May 2014, Dec. 2014)**

Q. Write an algorithm to find the minimum and maximum value using divide and conquer and also derive its complexity. *(Ans. : Refer section 2.5)* (10 Marks) **(May 2016)**

Q. Write an algorithm for finding maximum and minimum number from given set.
(Ans. : Refer section 2.5)(5 Marks) **(May 2017)**

Q. Design and analyze a divide and conquer algorithm for finding minimum and maximum number in the array of n-numbers that uses (3n/2) – 2 comparisons for any n.
(Ans. : Refer section 2.5)(6 Marks)

Q. Design and analyze a divide and conquer algorithm for finding a maximum and minimum element in a list L[1 : n]. *(Ans. : Refer section 2.5)*(6 Marks)

☞ **Syllabus Topic : Strassen's Matrix Multiplication**

Q. Write a short note on Strassen's matrix multiplication. *(Ans. : Refer section 2.6)* (10 Marks)
(May 2015, May 2016, May 2017)

Q. Explain the Strassen's Matrix multiplication.
(Ans. : Refer section 2.6) (10 Marks) **(Dec. 2015)**

❑❑❑

Module 1

CHAPTER 3

Recurrence

Syllabus Topics

The substitution method - Recursion tree method - Master method.

☞ **Introduction**

Q. Define recurrence equation. What is the use of it? (4 Marks)

Definition

Recurrence equation recursively defines a sequence of function with different argument. Behavior of recursive algorithm is better represented using recurrence equations.

- Recurrence are normally of the form:

$$T(n) = T(n-1) + f(n), \text{ for } n > 1. \quad \ldots(3.1.1)$$
$$T(n) = 0, \text{ for } n = 0$$

- The function $f(n)$ may represent constant or any polynomial in n.
- Equation (3.1.1) is called recurrence equation. T(n) is interpreted as the time required to solve the problem of size n. On recursively solving T(n) for $n = n - 1$, recurrence will hit to the base case $T(n) = 0$, for $n = 0$. Values will be back propagated and the final value of T(n) is computed.
- Recurrence of linear search / factorial : $T(n) = T(n-1) + 1$
- Recurrence of selection / bubble sort : $T(n) = T(n-1) + n$
- Let us discuss various methods to solve the recurrence equation.

☞ **Use**

- Recurrence relation can effectively represent the running time of recursive algorithms.
- Time complexity of certain recurrence can be easily solved using master methods.

Syllabus Topic : The Substitution Method

3.1 The Substitution Method

Q. Explain substitution method with example. (10 Marks)

Linear homogeneous recurrence of polynomial order greater than 2 hardly arises in practice. In this section, we shall discuss two unfolding methods which are widely used to solve a large class of recurrence.

It is also known **iteration methods.** There are two ways to solve such equations.

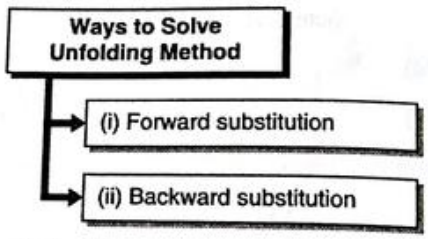


Fig. C3.1 : Ways to solve unfolding methods

➜ **(i) Forward substitution**

Forward substitution method finds the solution of the smallest problem using base condition. A solution of the bigger problem is obtained using the previously computed solution of the smaller problem. This process is repeated until the solution for problem n is achieved.

Ex. 3.1.1

Solve the recurrence of linear search using forward substitution method.

Soln. :

Recurrence of linear search is,

$$T(n) = T(n-1) + 1, \text{ for } n > 0 \text{ and}$$
$$T(n) = 0, \text{ for } n = 0 \text{ (Base condition)}$$

Inductive case of the recurrence is

$$T(n) = T(n-1) + 1 \quad \ldots(1)$$

Forward substitution finds the solution of T(n) by solving the progressively bigger problems, starting from smallest possible case.

Here, the base case is

$$T(n) = 0, \text{ for } n = 0$$
$$\Rightarrow T(0) = 0$$

...(2)

Put n = 1 in Equation (1) to solve problem of size 1.

$\Rightarrow T(1) = T(1-1) + 1 = T(0) + 1 = 0 + 1$

$\therefore T(1) = 1$ (From Equation (2))

For n = 2,

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

$\therefore T(2) = 2$

For n = 3,

$T(3) = T(3-1) + 1 = T(2) + 1 = 2 + 1$

$\therefore T(3) = 3$

After k substitutions,

$T(k) = k$,

And for k = n,

$T(n) = O(n)$

(ii) Backward substitution

This method substitutes the value of n by n – 1 recursively, to solve smaller and smaller problem. It works exactly in reverse order of forward substitution.

Ex. 3.1.2

Solve the recurrence of linear search using forward substitution method.

Soln. :

Recurrence of linear search is,

$T(n) = T(n-1) + 1$, for $n > 0$ and

$T(n) = 0$, for $n = 0$ (Base condition)

Inductive case of the recurrence is

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

The problem can be solved if T(n – 1) is known. To find T(n – 1), substitute n = n – 1 in Equation (1),

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

Replace this T(n – 1) in Equation (1)

$\therefore T(n) = [T(n-2) + 1] + 1$

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

The problem can be solved if T(n – 2) is known. To find T(n – 2), substitute n = n – 1 in Equation (2),

$T(n-2) = T(n-3) + 1$ ...(3)

Replace this T(n – 2) in Equation (2)

$\therefore T(n) = [T(n-3) + 1] + 2 = T(n-3) + 3$

After k substitutions,

$T(n) = T(n-k) + k$,

To match the base case, assume that k = n,

$\therefore T(n) = T(n-n) + n = T(0) = n = 0 + n$ (From base case)

$\therefore T(n) = O(n)$

Ex. 3.1.3

Solve recurrence equation T (n) = T (n – 1) + n using forward substitution and backward substitution method.

Soln. : Backward substitution

Given the recurrence,

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

Replace n by n – 1 in Equation (1),

$\Rightarrow T(n-1) = T(n-2) + (n-1)$ ...(2)

Put it in Equation (1),

$\therefore T(n) = [T(n-2) + (n-1)] + n$

Replace n by n – 1 in Equation (2)

$T(n-2) = T(n-3) + (n-2)$ ...(3)

$\Rightarrow T(n) = T(n-3) + (n-2) + (n-1) + n$ ...(4)

After k substitutions,

$T(n) = T(n-k) + (n-k+1) + (n-k+2) + \ldots (n-1) + n$

When k approaches to n,

$T(n) = T(0) + 1 + 2 + 3 + \ldots + (n-1) + n$

$\Rightarrow T(n) = \Sigma n = n(n+1)/2 = (n^2/2) + (n/2)$

So, $T(n) = O(n^2)$

Forward substitution

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

Forward substitution finds the solution by replacing the solution of smaller problem into large problem.

For n = 0, $T(1) = T(0) + 1 = 1$

For n = 2, $T(2) = T(1) + 2 = 1 + 2 = 3 = 1 + 2$

For n = 3, $T(3) = T(2) + 3 = 3 + 3 = 6 = 1 + 2 + 3$

For n = 4, $T(4) = T(3) + 4 = 6 + 4 = 10 = 1 + 2 + 3 + 4$.

After k substitutions,

For n = k, $T(n) = T(k-1) + k = 1 + 2 + 3 + \ldots + k$

When k approaches to n,

$T(n) = T(n-1) + n = 1 + 2 + 3 + \ldots + n = \Sigma n$

So, $T(n) = O(n^2)$

Ex. 3.1.4

Solve following recurrence using iteration method.

$T(n) = T(n-1) + n^4$

Soln. : Given that, $T(n) = T(n-1) + n^4$

Substitute n by n – 1

$\therefore T(n-1) = T(n-2) + (n-1)^4$

$\therefore T(n) = [T(n-2) + (n-1)^4] + n^4$

Substituting n by n – 2 in main recurrence,

$T(n-2) = T(n-3) + (n-2)^4$

$\therefore T(n) = [T(n-3) + (n-2)^4] + (n-1)^4 + n^4$

$\vdots$

After k iterations,

$T(n) = T(n-k) + (n-k+1)^4 + (n-k+2)^4 + \ldots + n^4$

For k = n

$T(n) = T(n-n) + T(n-n+1)^4 + (n-n+2)^4 + \ldots + n^4$

$= T(0) + 1^4 + 2^4 + 3^4 + \ldots + n^4 = \sum_{i=1}^{n} i^4$

$\therefore T(n) = \Theta(n^5)$

Ex. 3.1.5

Solve following recurrence using iteration

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

Soln. :

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

Substitute $n = n-1$ in this equation

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

$T(n) = 2[2T(n-2)]$

$= 4T(n-2) = 2^2 \cdot T(n-2)$

Substitute $n = n-2$ in main recurrence,

$\therefore T(n-2) = 2T(n-3)$

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

After k iterations

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

When $k = n-1$

$\therefore T(n) = 2^{n-1} \cdot T(n-n+1) = 2^{n-1} \cdot T(1)$

$\therefore T(n) = O(2^{n-1})$ $(\because T(1) = 1)$

Ex. 3.1.6

Find out the time complexity for the recurrence equation as follows :

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

(ii) $T(n) = 2T(n/2) + n$

Also explain the above equations belong to which searching/sorting algorithms.

Soln. :

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

Given recurrence is of binary search.

In every iteration, binary search does one comparison and creates the new problem of size n/2. So recurrence equation of binary search is given as,

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

$T(n) = 1$, $n = 1$ i.e. only one comparison is needed when there is only one element in the array, that's the trivial case. This is the boundary condition for recurrence. Let us solve this by iterative approach,

$$T(n) = T\left(\frac{n}{2}\right) + 1 \quad \ldots(1)$$

Put n = n/2 in Equation (1) to find T(n/2)

$$T\left(\frac{n}{2}\right) = T\left(\frac{n}{4}\right) + 1 \quad \ldots(2)$$

Substitute value of T(n/2) in Equation (1),

$$\therefore T(n) = T\left(\frac{n}{2^2}\right) + 2 \quad \ldots(3)$$

Put n = n/2 Equation (2) to find T(n/4),

$$T\left(\frac{n}{4}\right) = T\left(\frac{n}{8}\right) + 1$$

Substitute value of T(n/4) in Equation (3),

$$\therefore T(n) = T\left(\frac{n}{2^3}\right) + 3$$

.

.

After k iterations,

$$\therefore T(n) = T\left(\frac{n}{2^k}\right) + k$$

Suppose, $n = 2^k$, So $k = \log_2 n$

$$T(n) = T\left(\frac{2^k}{2^k}\right) + k = T(1) + k$$

$T(1) = 1$

So, $T(n) = 1 + k = 1 + \log_2 n$

$T(n) = O(\log_2 n)$

(ii) $T(n) = 2T(n/2) + n$

Given recurrence is of merge sort.

In merge sort, after each iteration, the list is divided into two parts, and problem size reduces by half. Unlike binary search, in merge sort, we will get two sub problems after divide stage. After sorting two sub lists of size (n / 2), combine procedure takes n comparisons. So, total running time of merge sort is given by

$$T(n) = \begin{cases} 0 & , \text{if } n \le 1 \\ T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + D(n) + C(n) & , \text{else} \end{cases}$$

Where first $T\left(\frac{n}{2}\right)$ = cost for solving left sub list

And second $T\left(\frac{n}{2}\right)$ = cost for solving right sub list

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) + \Theta(1)$$

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

$$\therefore T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + \frac{n}{2}$$

$$\therefore T(n) = 2\left(2 \cdot T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n$$

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

$$\vdots$$

$$= 2^k \cdot T\left(\frac{n}{2^2}\right) + kn$$

Suppose, $n = 2^k$, so $k = \log_2 n$

$$T(n) = n \cdot T\left(\frac{n}{n}\right) + \log_2 n \cdot n = n \cdot T(1) + n \cdot \log_2 n$$

But, $T(1) = 0$, because no comparison is needed to sort the list of size 1.

$$T(n) = O(n \cdot \log_2 n)$$

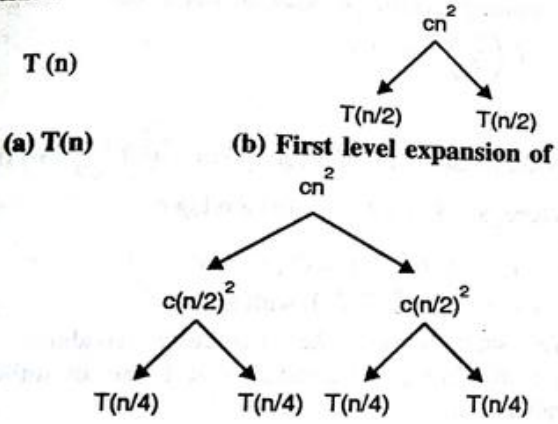
Syllabus Topic : Recursion Tree Method

3.2 Recursion Tree

Q. What is recursion tree? Describe. **(5 Marks)**

Drawing a solution by guess is difficult for complex recurrence. Recurrence tree method provides effective and yet simple way of solving the recurrences. Ultimately, recurrence is the set of functions, each branch in recurrence tree represents the cost of solving one problem from the family of problems belonging to given recurrence. Summation of cost across all branches at level i give us the total cost at level i. And summing cost of all levels, we get the total cost for solving the recurrence.

Let us consider the recurrence $T(n) = 2T(n/2) + \Theta(n^2)$. Let us derive the recurrence tree step by step as shown in Fig. 3.2.1. Problem is divided in to two sub problems, each of size n/2. The cost of solving problem of size n is n^2. We shall keep exploring the tree until it hits the boundary condition.



(a) T(n) **(b) First level expansion of T(n)**

(c) Second level expansion of T(n)

Fig. 3.2.1 : Recurrence tree formulation

At every level, problem is being reduced by factor of size 2. Size of sub problem at level i is $n/2^i$. Hence, size of sub problem becomes 1 when $n/2^i = 1$. Thus, tree has $\log_2 n + 1$ level. (depth vary from 0 to $\log_2 n$). Fully expanded tree is shown in Fig. 3.2.2.

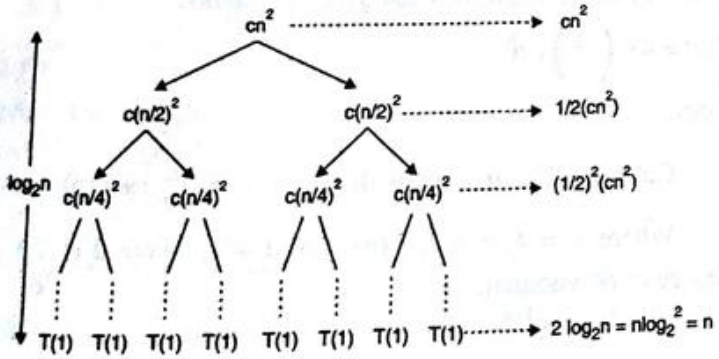


Fig. 3.2.2 : Fully expanded recurrence tree

We will compute the cost at each level in above tree. As it is a binary tree, number of nodes at depth i would be 2^i.

Cost at depth i would be $c.(n/2^i)^2$. Cost of base problem has size 1, so we can consider the cost of T(1) as constant. Height of the tree is $\log_2 n$. By summing up the cost at every level, we can find the cost of whole problem.

$$T(n) = cn^2 + \left(\frac{1}{2}\right) cn^2 + \left(\frac{1}{2}\right)^2 cn^2 + \ldots. + \left(\frac{1}{2}\right)^{\log_2 n - 1} n^2 + \Theta(n)$$

$$= \sum_{i=0}^{\log_2 n - 1} \left(\frac{1}{2}\right)^i cn^2 + \Theta(n)$$

$$< \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i cn^2 + \Theta(n)$$

$$= cn^2 \frac{1}{1-\left(\frac{1}{2}\right)} + \Theta(n)$$

$$= 2n^2 + \Theta(n)$$

$$T(n) = O(n^2)$$

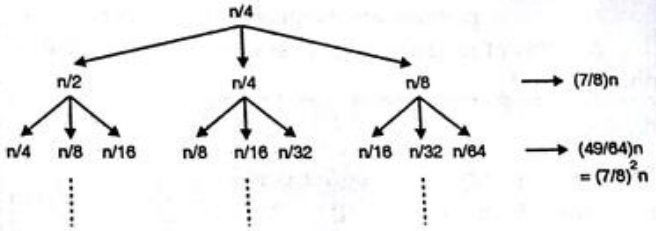
Ex. 3.2.1

Solve the following recurrence :

$T(n) = T(n/2) + T(n/4) + T(n/8) + n$.

Soln. :

Recurrence tree for given recurrence equation is shown below :



$$T(n) = T(n/2) + T(n/4) + T(n/8) + n$$

$$T(n) \leq n + (7/8)n + (7/8)^2 n + (7/8)^3 n + \ldots + (7/8)^h n,$$

where, h is the height of tree

$$= n \sum_{i=0}^{h} \left(\frac{7}{8}\right)^i$$

$$\text{Hence, } T(n) \leq n \sum_{i=0}^{\infty} \left(\frac{7}{8}\right)^i$$

$$= n \frac{1}{1-\left(\frac{7}{8}\right)} = 8n$$

Thus, $T(n) = O(n)$

Syllabus Topic : Master Method

3.3 Master Method

→ (May 15, May 17)

Q. Explain three cases of master theorem. **MU - May 2015, 5 Marks**

Q. Explain masters method with example. **MU - May 2017, 5 Marks**

- Divide and conquer strategy uses recursion. The time complexity of the recursive program is described using recurrence. In the previous chapter, we have studied various methods for solving the recurrence. The master theorem is often suitable to find the time complexity of recursive problems.
- Master method is used to quickly solve the recurrence of the form $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$. Master method finds the solution without substituting the values of T(n/b). In above equation,

  n = Size of the problem

  a = Number of sub problems created in recursive solution

  n/b = Size of each sub problem

  f(n) = Work done outside recursive call. This includes cost of division of problem and merging of the solution.

Solution of recurrence equation using master method is obtained as,

Variant I

If f(n) is the polynomial of degree d and

1. $T(n) = \Theta(n^d \log n)$ If $a = b^d$
2. $T(n) = \Theta\left(n^{\log_b^a}\right)$ If $a > b^d$
3. $T(n) = \Theta(n^d)$ If $a < b^d$

Variant II

1. If $f(n) = O\left(n^{\log_b^{a-\epsilon}}\right)$ for some constant $\epsilon > 0$, then $T(n) = \Theta\left(n^{\log_b^a}\right)$
2. If $f(n) = \Theta\left(n^{\log_b^a}\right)$, then $T(n) = \Theta\left(n^{\log_b^a} \log n\right)$
3. If $f(n) = \Omega\left(n^{\log_b^{a+\epsilon}}\right)$, for some constant $\epsilon > 0$, and if $a.f(n/b) \le c.f(n)$ for some constant $c < 1$, then $T(n) = \Theta(f(n))$

Examples of Master method

Ex. 3.3.1

Solve following recurrence equation using master method :

$T(n) = T\left(\frac{2n}{3}\right) + 1$

Soln. :

Given equation is of the form $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$

Where, $a = 1$, $b = \frac{3}{2}$ and $f(n) = 1$

$$n^{\log_b^a} = n^{\log_{3/2}^1} = n^0 = 1$$

Here, $f(n) = \Theta(n^{\log_b^a})$, so solution of recurrence would be,

$$T(n) = \Theta(n^{\log_b^a} \cdot \log^n) = \Theta(\log^n)$$

Ex. 3.3.2

Solve following recurrence equation using master method :

$T(n) = 3T\left(\frac{n}{4}\right) + n \log n$

Soln. :

Given equation is of the form $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$

Where, $a = 3$, $b = 4$ and $f(n) = n \log n$

$$O(n^{\log_b^a}) = O(n^{\log_4^3}) = O(n^{0.793})$$

Since, $f(n) = \Omega(n^{0.793+\epsilon})$, with $\epsilon \approx 0.2$

We shall check the regularity condition, i.e. $af(n/b) \le c.f(n)$ for some constant $c < 1$ and all sufficient large value of n.

$af(n/b) = 3(n/4)\log(n/4)$

$3(n/4)\log(n/4) \le (3/4)n\log n = c.f(n)$, with $c = 3/4$

Thus, it satisfies the condition 3 of master theorem, and hence

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

Ex. 3.3.3

Solve following recurrence using master method.

$T(n) = 4T\left(\frac{n}{2}\right) + n$

Soln. :

Compare this equation with $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$

Where, $a = 4$, $b = 2$, $f(n) = n$, $d = 1$, where d is the degree of polynomial.

$$b^d = 2^1$$

Here, $a = b^d$

So, $T(n) = \Theta(n^d \log n) = \Theta(n \log n)$.

Ex. 3.3.4

$T(n) = 8T\left(\frac{n}{2}\right) + n^d$

Soln. : Compare this equation with $T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$

Here, $a = 8$, $b = 2$ and $f(n) = n^2$, $d = 2$

Checking relation between a and b^d

As $a > b^d$ (i.e $8 > 2^2$), solution to this Equation is given as,

$T(n) = \left(n^{\log_b^a}\right) = \Theta\left(n^{\log_2^8}\right) = \Theta\left(n^{\log_2^{2^3}}\right) = \Theta\left(n3^{\log_2^2}\right) = \Theta(n^3)$

Ex. 3.3.5

Solve the following recurrence using Master method : $T(n) = 4T(n/3) + n^2$.

Soln. : Compare given recurrence with equation with $T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$

Here, $a = 4$, $b = 3$ and $f(n) = n^2$, $d = 2$

Checking relation between a and b^d

As $a < b^d$ (i.e $4 < 3^2$), solution to this Equation is given as,

$T(n) = \Theta(n^d) = \Theta(n^2)$

Ex. 3.3.6

Solve the following recurrences using the Master method : $T(1) = 0$. $T(n) = 9\,T(n/3) + n^3 \log n$; $n > 1$.

Soln. : Given equation is of the form

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

Where, $a = 9$, $b = 3$ and $f(n) = n^3 \log n$

$n^{\log_b^a} = n^{\log_3^9} = n^{\log_3^{3^2}} = n^{2\log_3^3} = n^2$

Since, $f(n) = \Omega(n^{2+\epsilon})$, with $\epsilon \approx 1$

We shall check the regularity condition, i.e. $af(n/b) \leq c.f(n)$ for some constant $c < 1$ and all sufficient large value of n.

$af(n/b) = 9(n/3)^3 \log(n/3)$

$9(n/3)^3 \log(n/3) \leq (1/3)\, n^3 \log n = c.f(n)$, with $c = 1/4$

Thus, it satisfies the condition 3 of master theorem, and hence

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

Ex. 3.3.7

Consider the recurrences. $T(n) = 3\,T(n/3) + cn$, and $T(n) = 5\,T(n/4) + n^2$ where c is constant and n is the number of inputs. Find the asymptotic bounds.

Soln. :

(1) $T(n) = 3\,T(n/3) + cn$

Compare given recurrence with equation with

$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$

Here, $a = 3$, $b = 3$ and $f(n) = n$, $d = 1$

Checking relation between a and b^d

As $a = b^d$ (i.e $3 < 3^1$), solution to this Equation is given as,

$T(n) = \Theta(n^d \log n) = \Theta(n^1 \log n) = \Theta(n \log n)$

(2) $T(n) = T(n) = 5\,T(n/4) + n^2$

Compare given recurrence with equation with

$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$

Here, $a = 5$, $b = 4$ and $f(n) = n^2$, $d = 2$

Checking relation between a and b^d

As $a < b^d$ (i.e $5 < 4^2$), solution to this Equation is given as,

$T(n) = \Theta(n^d) = \Theta(n^2)$

Ex. 3.3.8 **MU - Dec. 2015. 5 Marks**

Find the complexity of given recurrence relation

(i) $T(n) = 4T(n/2) + n^2$

(ii) $T(n) = 2T(n/2) + n^3$

Soln. : Given equations are of the form $T(n) = aT(n/b) + f(n)$. It can be easily solved using master's method.

(i) $T(n) = 4T(n/2) + n^2$

Here, $a = 4$, $b = 2$ and $f(n) = n^2$, $d = 2$

$b^d = 2^2 = 4$

From the master theory, here $a = b^d$ so

$T(n) = \Theta(n^d \log n) = \Theta(n^2 \log n)$

(ii) $T(n) = 2T(n/2) + n^3$

Here, $a = 2$, $b = 2$, $f(n) = n^3$ and $d = 3$.

$b^d = 2^3 = 8$

From master theorem, as $a < b^d$, $T(n) = \Theta(n^d) = \Theta(n^3)$

3.4 Exam Pack (University and Review Questions)

☞ **Syllabus Topic : The Substitution Method**

Q. Explain substitution method with example. *(Ans. : Refer section 3.1)*(10 Marks)

☞ **Syllabus Topic : Recursion Tree Method**

Q. What is recursion tree? Describe. *(Ans. : Refer section 3.2)* (5 Marks)

☞ **Syllabus Topic : Master Method**

Q. Explain three cases of master theorem. *(Ans. : Refer section 3.3)* (5 Marks) **(May 2015)**

Q. Explain masters method with example. *(Ans. : Refer section 3.3)* (5 Marks) **(May 2017)**

Ex. 3.3.8 (5 Marks) **(Dec. 2015)**

Module 2

CHAPTER 4

Dynamic Programming

Syllabus Topics

General Method, Multistage graphs, single source shortest path, all pair shortest path, Assembly-line scheduling, 0/1 knapsack, Travelling salesman problem, Longest common subsequence.

Syllabus Topic : General Method

4.1 General Method

- Dynamic programming was invented by U.S. mathematician Richard Bellman in 1950. Like greedy algorithms, dynamic programming is also used to solve optimization problems. But unlike greedy approach, dynamic programming always ensures optimal / best solution.

Definition

A feasible solution is a solution that satisfies constraints of the problem. When the problem has multiple feasible solutions with different cost, the solution with the minimum cost or maximum profit is called **optimal solution.**

- Cost metric depends on the problem. For sorting problem, cost metric may be a number of comparisons or number of swaps. For matrix multiplication, a cost metric is a number of multiplications. For knapsack problem, cost metric is total profit earned.

4.1.1 Introduction

Q. What is Dynamic programming? Is this the optimization technique? Give reasons. What are the drawbacks of dynamic programming? **(5 Marks)**

- Dynamic programming is powerful design technique for optimization problems. Here word "programming" refers to planning or construction of a solution, it does not have any resemblance to computer programming.
- Divide and conquer divides the problem into small subproblems.
- Subproblems are solved recursively. Unlike divide and conquer, subproblems in dynamic programming are not independent. Subproblems in dynamic programming overlap with each other. Solutions of subproblems are merged to get the solution of the original large problem.
- In divide and conquer, subproblems are independent and hence repeated problems are solved multiple times. Dynamic programming saves the solution in the table, so when the same problem encounters again, the solution is retrieved from the table. Dynamic programming is bottom-up approach. It starts solving the smallest possible problem and uses a solution of the smaller problem to build solution of the larger problem.

☞ **Limitations**

- The method is applicable to only those problems which possess the property of principle of optimality.
- We must keep track of partial solutions.
- Dynamic programming is more complex and time-consuming.

4.1.2 Control Abstraction

- Dynamic Programming (DP) splits the large problem at every possible point. When the problem becomes sufficiently small, DP solves it.
- Dynamic programming is bottom-up approach, it finds the solution of the smallest problem and constructs the solution of the larger problem from already solved smaller problems.
- To avoid recomputation of the same problem, DP saves the result of subproblems into the table. When next time same problem encounters, the answer is retrieved from the table by lookup procedure.

– Control abstraction for dynamic programming is shown below :

```
Algorithm DYNAMIC_PROGRAMMING (P)
if solved(P) then
    return lookup(P)          // If P is already solved, then retrieve stored answer
else
    Ans ←SOLVE(P)
    store (P, Ans)
end

Function SOLVE(P)
if sufficiently small(P) then
    solution(P)
// Find solution for sufficiently small problem
else
    Divide P into smaller subproblems P1, P2, ...., Pn
    Ans1 ← DYNAMIC_PROGRAMMIN(P1)
    Ans2 ← DYNAMIC_PROGRAMMIN(P2)
    ...
    ...
    ...
    Ansn ← DYNAMIC_PROGRAMMIN(Pn)

                              // Combine solutions of smaller sub problems in order to achieve the solution to larger problem
return(combine(Ans1, Ans2, ... , Ansn))
end
```

4.1.3 Characteristics of Dynamic Programming

Q. What are the characteristics of Dynamic Programming? (4 Marks)

Dynamic programming works on following principles :

- Characterize structure of optimal solution, i.e. build a mathematical model of the solution.
- Recursively define the value of the optimal solution.
- Using bottom-up approach, compute the value of the optimal solution for each possible subproblems.
- Construct optimal solution for the original problem using information computed in the previous step.

4.1.4 Applications

Q. State applications of Dynamic Programming. (3 Marks)

Dynamic programming is used to solve optimization problems. It is used to solve many real-life problems such as,

(i) Make a change problem
(ii) Knapsack problem
(iii) Optimal binary search tree
(iv) Travelling salesman problem
(v) All pair shortest path problem
(vi) Assembly line scheduling
(vii) Multi-stage graph problem

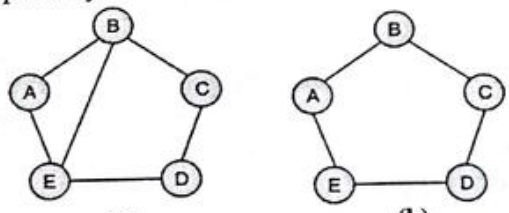
4.2 Principle of Optimality

Q. State "Principle of Optimality". (2 Marks)
Q. What is dynamic programming approach to solve the problem? (4 Marks)
Q. What is the principle of optimality in dynamic programming ? Give a suitable example for this property of optimality in dynamic programming. (7 Marks)

Definition

Principle of optimality : "In an optimal sequence of decisions or choices, each subsequence must also be optimal".

- The principle of optimality is the heart of dynamic programming. It states that to find the optimal solution of the original problem, a solution of each subproblem also must be optimal. It is not possible to derive optimal solution using dynamic programming if the problem does not possess the principle of optimality.
- Shortest path problem satisfies the principle of optimality. If A - X_1 - X_2 - . . . X_n - B is the shortest path between nodes A and B, then any subpath X_i to X_j must be shortest. If there exist multiple paths from X_i to X_j, and the selected path is not minimum, then obviously the path from A to B cannot be shortest.
- **For example :** In Fig. 4.2.1(a) shortest path from A to C is A – B – C. There exist two paths from B to C. One is B – C and other is B – E – D – C. But B – C is the shortest path so we should add that one in the final solution.
- On the other hand, longest path problem does not satisfy the principle of optimality. In Fig. 4.2.1(b), the longest non-cyclic path from A to D is A – B – C – D. In this path, sub-path from B to C is just an edge joining B and C. However, BC itself is not the longest path because longest non-cyclic path from B to C is B – A – E – D – C. Thus the sub path of the longest path may not be longest always, so violets principle of optimality.



(a) (b)

Fig. 4.2.1 : Illustration of shortest and longest path problem

4.3 Elements of Dynamic Programming

Q. Discuss the elements of dynamic programming. **(4 Marks)**

Q. What are the common steps in the dynamic programming to solve any problem? **(6 Marks)**

- **Optimal substructure :** "For the optimal solution of the problem, a solution of subproblem also must be optimal". Dynamic programming builds the optimal solution of the bigger problem using the solution of smaller subproblems. Hence we should consider only those subproblems which have an optimal solution.
- **Overlapping subproblems :** When the big problem is divided into small problems, it may create exponential subproblems. Only polynomial numbers of them are distinct. Fig. 4.3.1 shows the overlapping problems of the binomial coefficient. Dynamic programming saves solution in the table, so no rework is done. When $C(n-3, r-2)$ problem encounters again, its solution is retrieved from the table. Divide and conquer solves $C(n-3, r-2)$ four times, whereas dynamic programming solves only once. There may exist many subproblems with multiplicity greater than one.

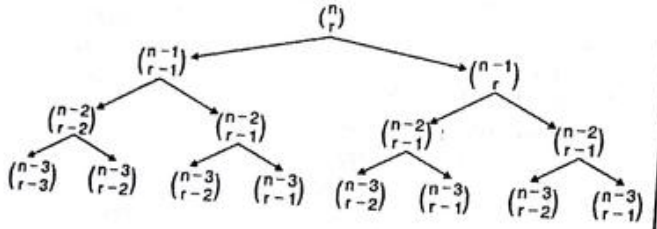


Fig. 4.3.1 : Divide and conquer tree for ${}_nC_r$

4.4 Divide and Conquer Vs Dynamic Programming

Q. Distinguish between dynamic programming and divide and conquer technique. **(6 Marks)**

Sr. No.	Divide and Conquer	Dynamic Programming
1.	Divide and conquer is the top-down approach.	Dynamic programming is bottom-up approach.
2.	Divide and conquer prefers recursion.	Dynamic programming prefers iteration.
3.	In divide and conquer, sub problems are independent.	Subproblems of dynamic programming are dependent and overlapping.
4.	Solutions of subproblems are not stored.	Solutions of subproblems are stored in the table.
5.	Lots of repetition of work.	No repetition at work at all.
6.	It splits input at a specific point.	It splits input at each and every possible point.
7.	Less efficient due to rework.	More efficient due to the saving of solution.
8.	Solution using divide and conquer is simple.	Sometimes, a solution using dynamic programming is complex and tricky.
9.	Only one decision sequence is generated.	Many decision sequences may be generated.

Syllabus Topic : Multistage Graph

4.5 Multistage Graph

→ **(Dec. 16, May 17)**

Q. Write a short note on Multistage graph. **MU - Dec. 2016, May 2017, 5 Marks**

Q. What is multistage graph? How to solve it using dynamic programming? **(6 Marks)**

- Multistage graph G = (V, E, W) is a weighted directed graph in which vertices are partitioned into $k \geq 2$ disjoint subsets $V = \{V_1, V_2, \ldots, V_k\}$ such that if the edge (u, v) is present in E then $u \in V_i$ and $v \in V_{i+1}$, $1 \leq i \leq k$. The goal of multistage graph problem is to find minimum cost path from source to destination vertex.
- The input to the algorithm is a k-stage graph, n vertices are indexed in increasing order of stages. The algorithm operates in the backward direction, i.e. it starts from the last vertex of the graph and proceeds in a backward direction to find minimum cost path.
- Minimum cost of vertex $j \in V_i$ from vertex $r \in V_{i+1}$ is defined as,

  $Cost[j] = \min\{c[j, r] + cost[r]\}$

  where, c[j, r] is the weight of edge <j, r> and cost[r] is the cost of moving from end vertex to vertex r.
- Algorithm for the multistage graph is described below :

```
Algorithm MULTI_STAGE(G, k, n, p)
// Description : Solve multi-stage problem using dynamic
    programming
// Input :  k: Number of stages in graph G = (V, E)
            c[i, j]: Cost of edge (i, j)
// Output : p[1:k]: Minimum cost path

cost[n] ← 0
for j ← n – 1 to 1 do        Compute cost of j^th vertex
```

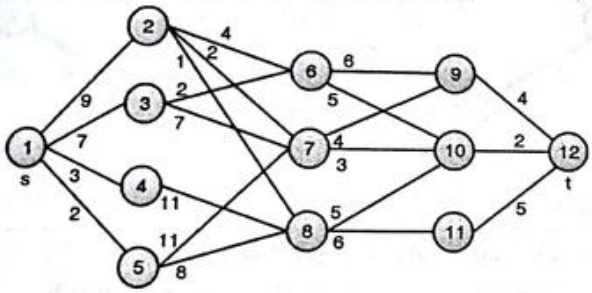
```
    // Let r be a vertex such that (j, r) ∈ E and c[j, r] +
       cost[r] is minimum
    cost[j] ← c[j, r] + cost[r]
    π[j] ← r                          Set successor of vertex j
end
//Find minimum cost path
p[1] ← 1
p[k] ← n                              Trace minimum cost path
for j ← 2 to k - 1 do
    p[j] ←π[p[j - 1]]
end
```

☞ **Complexity analysis**

If graph G has |E| edges, then cost computation time would be O(n + |E|). The complexity of tracing the minimum cost path would be O(k), k < n. Thus total time complexity of multistage graph using dynamic programming would be O(n + |E|).

Ex. 4.5.1

Find minimum path cost between vertex s and t for following multistage graph using dynamic programming.



Soln. :

Solution to multistage graph using dynamic programming is constructed as,

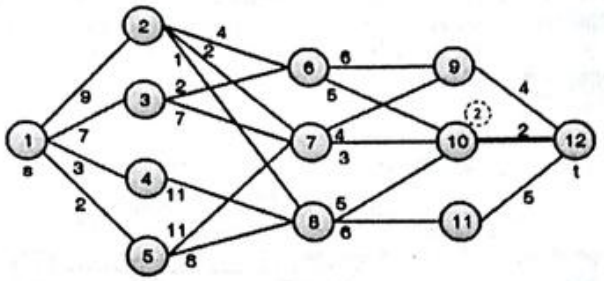
Cost[j] = min{c[j, r] + cost[r]}
Here, number of stages k = 5, number of vertices n = 12, source s = 1 and target t = 12
Initialization:
Cost[n] = 0 ⇒Cost[12] = 0.
p[1] = s ⇒ p[1] = 1
p[k] = t ⇒p[5] = 12.
r = t = 12.

Stage 4

Cost[9] = c[9, 12] + cost[12] = 4 + 0 = 4
π[9] = 12 // Successor of vertex 9 is 12

Cost[10] = c[10, 12] + cost[12] = 2 + 0 = 2
π[10] = 12 // Successor of vertex 10 is 12

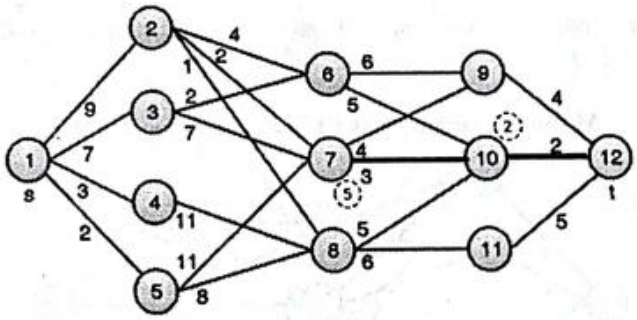
Cost[11] = c[11, 12] + cost[12] = 5 + 0 = 5
π[11] = 12 // Successor of vertex 11 is 12



Stage 3

Vertex 6 is connected to vertices 9 and 10.
Cost[6] = min{ c[6, 10] + Cost[10], c[6, 9] + Cost[9] }
= min{5 + 2, 6 + 4} = min{7, 10} = 7
π[6] = 10
Vertex 7 is connected to vertices 9 and 10.
Cost[7] = min{ c[7, 10] + Cost[10], c[7, 9] + Cost[9] }
= min{3 + 2, 4 + 4} = min{5, 8} = 5
π[7] = 10

Vertex 8 is connected to vertex 10 and 11.
Cost[8] = min{ c[8, 11] + Cost[11], c[8, 10] + Cost[10] }
= min{6 + 5, 5 + 2} = min{11, 7} = 7
π[8] = 10



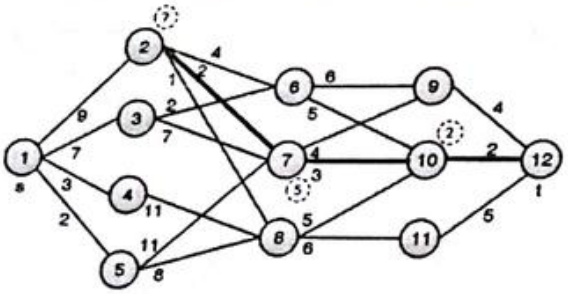
Stage 2

Vertex 2 is connected to vertices 6, 7 and 8.
Cost[2] = min{ c[2, 6] + Cost[6], c[2, 7] + Cost[7], c[2, 8] + Cost[8] }
= min{4 + 7, 2 + 5, 1 + 7} = min{11, 7, 8} = 7
π[2] = 7

Vertex 3 is connected to vertices 6 and 7.
Cost[3] = min{ c[3, 6] + Cost[6], c[3, 7] + Cost[7] }
= min{2 + 7, 7 + 5} = min{9, 12} = 9
π[3] = 6

Vertex 4 is connected to vertex 8.
Cost[4] = c[4, 8] + Cost[8] = 11 + 7 = 18
π[4] = 8

Vertex 5 is connected to vertices 7 and 8.
Cost[5] = min{ c[5, 7] + Cost[7], c[5, 8] + Cost[8] }
= min{11 + 5, 8 + 7} = min{16, 15} = 15
π[5] = 8



Stage 1

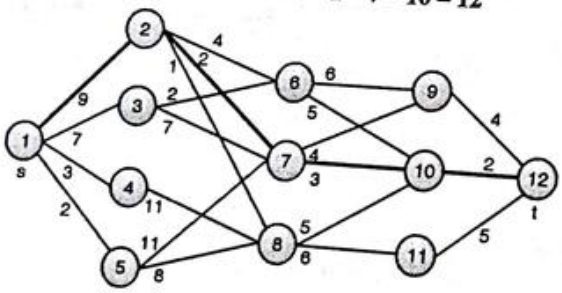
Vertex 1 is connected to vertices 2, 3, 4 and 5.
Cost[1] = min{ c[1, 2] + Cost[2], c[1, 3] + Cost[3], c[1, 4] + Cost[4], c[1, 5] + Cost[5]}
= min{ 9 + 7, 7 + 9, 3 + 18, 2 + 15 }
= min { 16, 16, 21, 17 } = 16
π[1] = 2

Trace the solution

π[1] = 2 π[2] = 7
π[7] = 10 π[10] = 12

d[1]	d[2]	d[3]	d[4]	d[5]	d[6]	d[7]	d[8]	d[9]	d[10]	d[11]	d[12]
2	7	6	8	8	10	10	10	12	12	12	12

Minimum cost path is : 1 – 2 – 7 – 10 – 12



Cost of the path is : 9 + 2 + 3 + 2 = 16

Ex. 4.5.2 MU - Dec. 2014, 10 Marks

Find a minimum cost path from 1 to 9 in the given graph using dynamic programming.

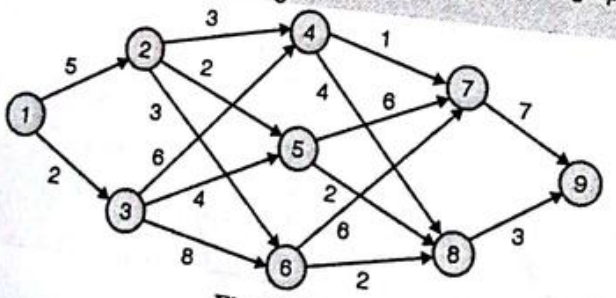


Fig. P. 4.5.2

Soln. :

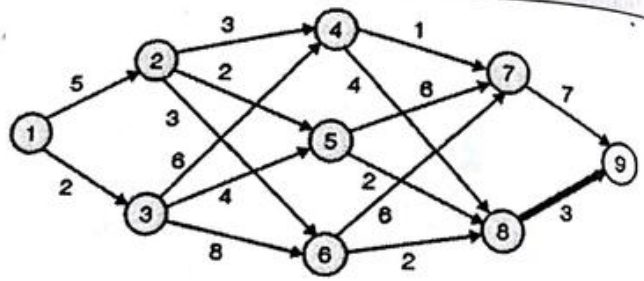
Solution to multistage graph using dynamic programming is constructed as,

Cost[j] = min{c[j, r] + cost[r]}
Here, number of stages k = 5, number of vertices n = 9, source s = 1 and target t = 9
Initialization:
Cost[n] = 0 ⇒ Cost[9] = 0.
p[1] = s ⇒ p[1] = 1
p[k] = t ⇒ p[5] = 9.
r = t = 9.

Stage 4

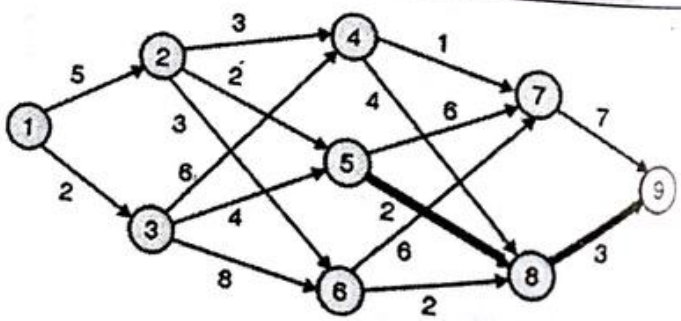
Cost[7] = c[7, 9] + cost[9] = 7 + 0 = 7
π[7] = 9 // Successor of vertex 7 is 9

Cost[8] = c[8, 9] + cost[9] = 3 + 0 = 3
π[8] = 9 // Successor of vertex 8 is 9



Stage 3

Vertex 4 is connected to vertices 7 and 8.
Cost[4] = min{ c[4, 7] + Cost[7], c[4, 8] + Cost[8] }
= min{1 + 7, 4 + 3} = min{8, 7} = 7
π[4] = 8
Vertex 5 is connected to vertices 7 and 8.
Cost[5] = min{ c[5, 7] + Cost[7], c[5, 8] + Cost[8] }
= min{6 + 7, 2 + 3} = min{13, 5} = 5
π[5] = 8
Vertex 6 is connected to vertex 7 and 8.
Cost[6] = min{ c[6, 7] + Cost[7], c[6, 8] + Cost[8] }
= min{6 + 7, 2 + 3} = min{13, 5} = 5
π[6] = 8



Stage 2

Vertex 2 is connected to vertices 4, 5 and 6.

$Cost[2] = \min\{ c[2, 4] + Cost[4], c[2, 5] + Cost[5], c[2, 6] + cost[6] \}$

$= \min\{3 + 7, 2 + 5, 3 + 5\} = \min\{9, 7, 8\} = 7$

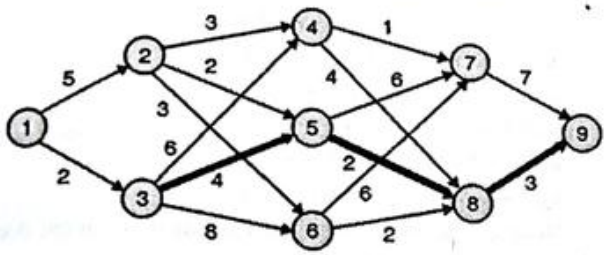
$\pi[2] = 5$

Vertex 3 is connected to vertices 4, 5 and 6.

$Cost[3] = \min\{ c[3, 4] + Cost[4], c[3, 5] + Cost[5], c[3, 6] + cost[6] \}$

$= \min\{6 + 7, 4 + 5, 8 + 5\} = \min\{13, 9, 13\} = 9$

$\pi[3] = 5$



Stage 1

Vertex 1 is connected to vertices 2 and 3.

$Cost[1] = \min\{ c[1, 2] + Cost[2], c[1, 3] + Cost[3] \}$

$= \min\{5 + 7, 2 + 9\} = \min\{12, 11\} = 11$

$\pi[1] = 3$

Trace the solution

$\pi[1] = 3 \quad \pi[3] = 5$

$\pi[5] = 8 \quad \pi[8] = 9$

Minimum cost path is : 1 – 3 – 5 – 8 – 9

Syllabus Topic : Single Source Shortest Path

4.6 Single Source Shortest Path

Q. How to find single Bellman-Ford algorithm? **(4 Marks)**

Q. How to solve single source shortest path problem using a Bellman-Ford algorithm? **(4 Marks)**

- Bellman-Ford algorithm is used to find the shortest path from the source vertex to remaining all other vertices in the weighted graph.
- It is slower compared to Dijkstra's algorithm but it can handle ***negative weights*** also.
- If the graph contains negative-weight cycle (edge sum of the cycle is negative), it is not possible to find the minimum path, because on every iteration of cycle gives a better result.
- Bellman-Ford algorithm can detect a negative cycle in the graph but it cannot find the solution for such graphs.
- Like Dijkstra, this algorithm is also based on the principle of relaxation. The path is updated with better values until it reaches the optimal value.
- Dijkstra uses a priority queue to greedily select the closest vertex and perform relaxation on all its adjacent outgoing edges. By contrast, Bellman-Ford relaxes all the edges |V| - 1 time. Bellman-Ford runs in O(|V|.|E|) time.

```
Algorithm BELLMAN-FORD (G)
// Description : Find the shortest path from source vertex to all
     other vertices using Bellman-Ford algorithm
// Input : Weighted graph G = (V, E, W)
          w(u, v) : Weight of edge (u, v)
          d[u] : distance of vertex u from source
          π[u] : Successor of vertex u
// Output : Shortest distance of each vertex from given source
     vertex.

// Initialization
for each v ∈ V do
     d[v] ←∞
     π[v] ← NULL
end
d[s] ← 0

// Relaxation
for i ← 1 to |V| - 1 do
     for each edge (u, v) ∈ E do
          if d[u] + w (u, v) < d[v] then
               d[v] ← d[u] + w(u, v)
               π[v] ← u
          end
     end
end

// Check for negative cycle
for each edge (u, v) ∈ E do
     if d[u] + w(u, v) < d[v] then
     error "Graph contains negative cycle"
     end
end
return d, π
```

Ex. 4.6.1

Find the shortest path from source vertex 5 for given graph.

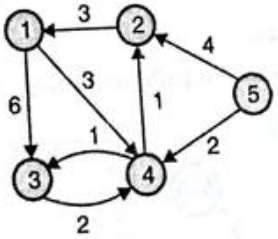
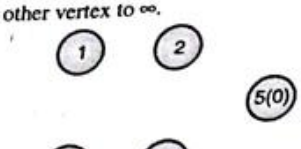


Fig. P. 4.6.1

Soln. :

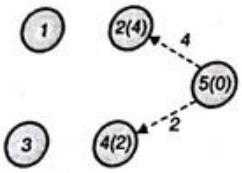
Initialize the distance: Set distance of source vertex to 0 and every other vertex to ∞.



V	1	2	3	4	5
d[v]	∞	∞	∞	∞	0
π[v]	/	/	/	/	-

Iteration 1

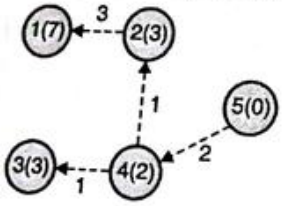
Edges <5, 2> and <5, 4> will be relaxed as their distance will be updated to 4 and 2 respectively.



V	1	2	3	4	5
d[v]	∞	4	∞	2	0
π[v]	/	5	/	5	-

Iteration 2

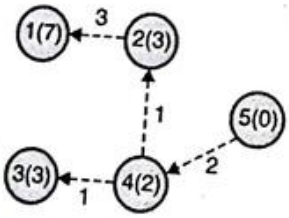
Edges <2, 1>, <4, 3> and <2, 4> will be relaxed as their distance will be updated to 7, 3 and 3 respectively.



V	1	2	3	4	5
d[v]	7	3	3	2	0
π[v]	2	4	4	5	-

Iteration 3

Edge <2, 1> will be relaxed as its distance will be updated to 6.

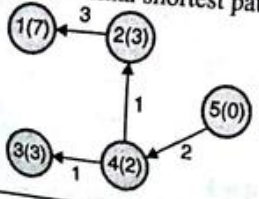


V	1	2	3	4	5
d[v]	6	3	3	2	0
π[v]	2	4	4	5	-

Iteration 4

No edge relaxed

So the final shortest path from source vertex 5 is:



V	1	2	3	4	5
d[v]	6	3	3	2	0
π[v]	2	4	4	5	-

Shortest path from any vertex can be found by starting from the destination vertex and following its predecessor π's back to the source. For example, starting at vertex 1, $u1.\pi = 2$, $u2.\pi = 4$, $u4.\pi = 5 \Rightarrow$ the shortest path to vertex 1 is {5,4,2,1}.

Negative cycle check

We have to test following condition for each edge.

if d[u] + w(u, v) < d[v] **then**

error "Graph contains negative cycle"

end

$u1.d + w(1,3) < v3.d \Rightarrow 6 + 6 \geq 4$

$u_1.d + w(1,4) < v_4.d \Rightarrow 6 + 3 \geq 2$

$u_2.d + w(2,1) < v_1.d \Rightarrow 3 + 3 \geq 6$

$u_3.d + w(3,4) < v_4.d \Rightarrow 3 + 2 \geq 2$

$u_4.d + w(4,2) < v_2.d \Rightarrow 2 + 1 \geq 3$

$u_4.d + w(4,3) < v_3.d \Rightarrow 2 + 1 \geq 3$

$u_5.d + w(5,2) < v_2.d \Rightarrow 0 + 4 \geq 3$

$u_5.d + w(5,4) < v_4.d \Rightarrow 0 + 2 \geq 2$

None of the above conditions satisfied, so graph does not contain any negative cycle.

Ex. 4.6.2 **MU - Dec. 2015, 10 Marks**

Find a minimum cost path from 3 to 2 in the given graph using dynamic programming.

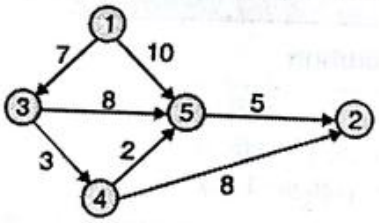
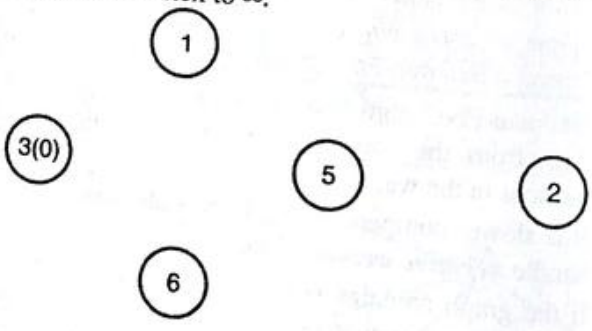


Fig. P. 4.6.2

Soln. :

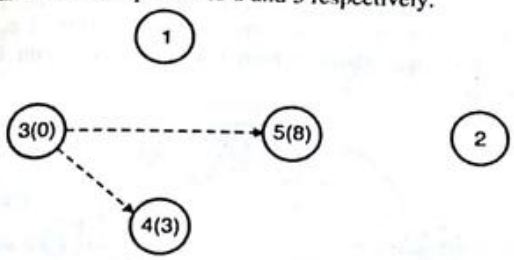
Initialize the distance: Set distance of source vertex to 0 and every other vertex to ∞.



V	1	2	3	4	5
d[v]	∞	∞	0	∞	∞
π[v]	/	/	-	/	/

Iteration 1

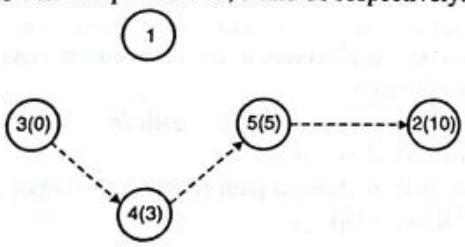
Edges <3, 5> and <3, 4> will be relaxed as their distance will be updated to 8 and 3 respectively.



V	1	2	3	4	5
d[v]	∞	∞	0	3	8
π[v]	/	/	-	3	3

Iteration 2

Edges <3, 5>, <4, 5> and <5, 2> will be relaxed as their distance will be updated to 5, 3 and 10 respectively.



V	1	2	3	4	5
d[v]	∞	10	0	3	5
π[v]	/	5	-	3	4

Iteration 3

No updates in weight

Shortest path from any vertex can be found by starting from the destination vertex and following its predecessor π's back to the source. For example, starting at vertex 1, u2.π = 5, u5.π = 4, u4.π = 3 ⇒ the shortest path from vertex 3 to vertex 2 is {3,4,5,2}.

Ex. 4.6.3 **MU - May 2013, 10 Marks**

Solve the shortest path from source 1 for following graph using dynamic programming.

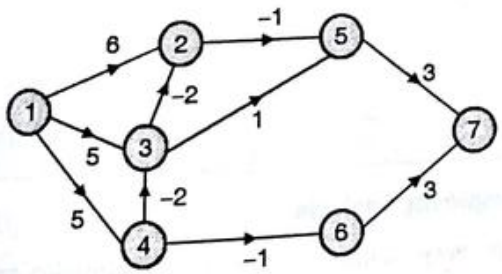


Fig. P. 4.6.3

Soln. :

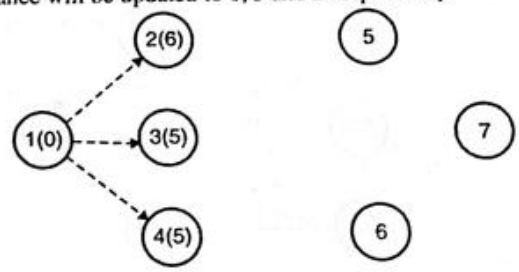
Graph contains negative weight edges, but it does not contain any negative cycle.

Initialize the distance : Set distance of source vertex to 0 and every other vertex to ∞.

V	1	2	3	4	5	6	7
d[v]	0	∞	∞	∞	∞	∞	∞
π[v]	-	/	/	/	/	/	/

Iteration 1

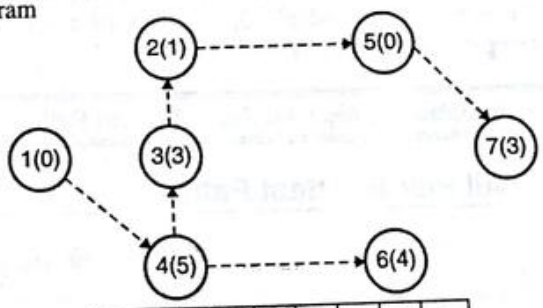
Edges <1, 2>, <1, 3> and <1, 4> will be relaxed as their distance will be updated to 6, 5 and 5 respectively.



V	1	2	3	4	5	6	7
d[v]	0	6	5	5	∞	∞	∞
π[v]	-	1	1	1	/	/	/

Iteration 2

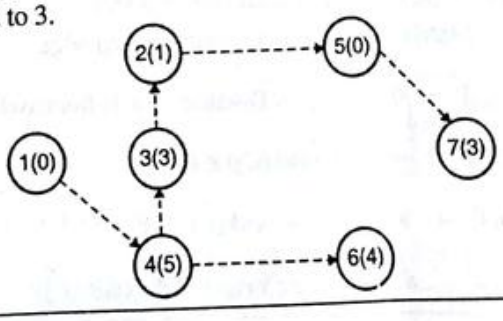
Path cost after second iteration is shown in following diagram



V	1	2	3	4	5	6	7
d[v]	0	1	3	5	0	4	∞
π[v]	-	3	4	1	2	4	/

Iteration 3

Edge <5, 7> will be relaxed as its distance will be updated to 3.



V	1	2	3	4	5	6	7
d[v]	0	1	3	5	0	4	3
π[v]	-	3	4	1	2	4	5

Iteration 4

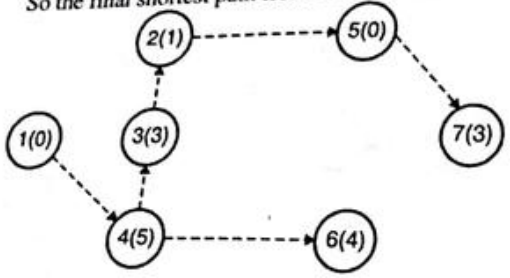
Iteration 5

Iteration 6

There won't be any change in weights in iteration 4, 5 and 6.

No edge relaxed

So the final shortest path from source vertex 5 is:



V	1	2	3	4	5	6	7
d[v]	0	1	3	5	0	4	3
π[v]	-	3	4	1	2	4	5

Shortest path from any vertex can be found by starting from the destination vertex and following its predecessor π's back to the source. For example, starting at vertex 7, $u7.\pi = 5$, $u5.\pi = 2$, $u2.\pi = 3$, $u3.\pi = 4$, $u4.\pi = 1 \Rightarrow$ the shortest path to vertex 1 is $1 - 4 - 3 - 2 - 5 - 7$

Syllabus Topic : All Pair Shortest Path

4.7 All Pair Shortest Path

→ **(May 14)**

Q. Explain all pair shortest path algorithm with a suitable example. **MU - May 2014, 10 Marks**

Q. Write Floyd's algorithm for all pairs shortest path and find time complexity. **(7 Marks)**

☞ **Problem**

Let G = <V, E> be a directed graph, where V is set of vertices and E is set of edges with nonnegative length. Find the shortest path between each pair of nodes.

L = Matrix, which gives length of each edge

L[i, j] = 0, if i = j // Distance of node from itself is zero

L[i, j] = ∞, if i ≠ j and (i, j) ∉ E

L[i, j] = w (i, j), if i ≠ j and (i, j) ∈ E

// w(i, j) is the weight of the edge (i, j)

☞ **Principle of optimality**

If k is the node on shortest path from i to j, then path from i to k and k to j, must also be shortest. In Fig. 4.7.1, optimal path from i to j is either p or summation of p_1 and p_2. In first step, solution matrix is initialized with input graph matrix.

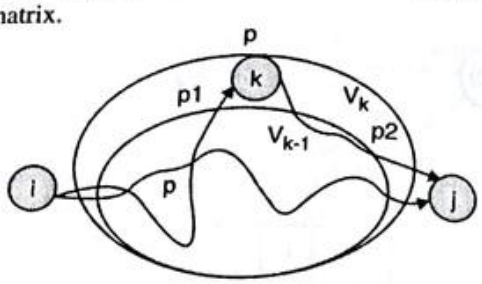


Fig. 4.7.1

We update the distance matrix by considering one by one all vertices as intermediate vertices. In iteration i, the i^{th} vertex is used as an intermediate vertex in the shortest path.

When a k^{th} vertex is under consideration, we already have explored 1 to k – 1 vertex as intermediate vertices. While considering k^{th} vertex as an intermediate vertex, there are two possibilities :

If k is not part of the shortest path from i to j, we keep the distance D[i, j] as it is.

If k is part of shortest path from i to j, update distance D[i, j] as D[i, k] + D[k, j].

Optimal sub structure of the problem is given as :

$D^k[i, j] = \min\{ D^{k-1}[i, j], D^{k-1}[i, k] + D^{k-1}[k, j] \}$

D^k = Distance matrix after k^{th} iteration

This approach is also known as **Floyd-warshall** shortest path algorithm. Algorithm for all pair shortest path (APSP) problem is described below:

```
Algorithm FLOYD_APSP ( L)
// L is the matrix of size n × n representing original graph
// D is the distance matrix
D ← L
for k ← 1 to n do
    for i ←1 to n do
        for j ← 1 to n do
            D[i, j]^k ← min ( D[i, j]^k-1, D[i, k]^k-1 + D[k, j]^k-1 )
        end
    end
end
return D
```

☞ **Complexity analysis**

It is very simple to derive the complexity of this approach from above algorithm. It uses three nested loops

Innermost loop has only one statement. The complexity of that statement is $\Theta(1)$.

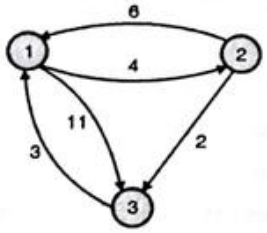
Running time of the algorithm is computed as :

$$T(n) = \sum_{k=1}^{n}\sum_{i=1}^{n}\sum_{j=1}^{n}\Theta(1) = \sum_{k=1}^{n}\sum_{i=1}^{n} n = \sum_{k=1}^{n} n^2$$
$$= \Theta(n^3)$$

Thus, floyd's algorithm runs in cubic time.

Ex. 4.7.1

Solve the all pairs shortest path problem for the given graph.



Soln. :

Optimal substructure formula for Floyd's algorithm,

$$D^k[i,j] = \min\{ D^{k-1}[i,j], D^{k-1}[i,k] + D^{k-1}[k,j] \}$$

$$D^0 = \begin{array}{|c|c|c|}\hline 0 & 4 & 11 \\ \hline 6 & 0 & 2 \\ \hline 3 & \infty & 0 \\ \hline \end{array}$$

Iteration 1 (k = 1)

$D^1[1,2] = \min\{D^0[1,2], D^0[1,1] + D^0[1,2]\}$
$= \min\{4, 0+4\} = 4$

$D^1[1,3] = \min\{D^0[1,3], D^0[1,1] + D^0[1,3]\}$
$= \min\{11, 0+11\} = 11$

$D^1[2,1] = \min\{D^0[2,1], D^0[2,1] + D^0[1,1]\}$
$= \min\{6, 6+0\} = 6$

$D^1[2,3] = \min\{D^0[2,3], D^0[2,1] + D^0[1,3]\}$
$= \min\{2, 6+11\} = 2$

$D^1[3,1] = \min\{D^0[3,1], D^0[3,1] + D^0[1,1]\}$
$= \min\{3, 3+0\} = 3$

$D^1[3,2] = \min\{D^0[3,2], D^0[3,1] + D^0[1,2]\}$
$= \min\{\infty, 3+4\} = 7$

$$D^1 = \begin{array}{|c|c|c|}\hline 0 & 4 & 11 \\ \hline 6 & 0 & 2 \\ \hline 3 & 7 & 0 \\ \hline \end{array}$$

Iteration 2 (k = 2)

$D^2[1,2] = \min\{D^1[1,2], D^1[1,2] + D^1[2,2]\}$
$= \min\{4, 4+0\} = 4$

$D^2[1,3] = \min\{D^1[1,3], D^1[1,2] + D^1[2,3]\}$
$= \min\{11, 4+2\} = 6$

$D^2[2,1] = \min\{D^1[2,1], D^1[2,2] + D^1[2,1]\}$
$= \min\{6, 0+6\} = 6$

$D^2[2,3] = \min\{D^1[2,3], D^1[2,2] + D^1[2,3]\}$
$= \min\{2, 0+2\} = 2$

$D^2[3,1] = \min\{D^1[3,1], D^1[3,2] + D^1[2,1]\}$
$= \min\{3, 7+6\} = 3$

$D^2[3,2] = \min\{D^1[3,2], D^1[3,2] + D^1[2,2]\}$
$= \min\{7, 7+0\} = 7$

$$D^2 = \begin{array}{|c|c|c|}\hline 0 & 4 & 6 \\ \hline 6 & 0 & 2 \\ \hline 3 & 7 & 0 \\ \hline \end{array}$$

Iteration 3 (k = 3)

$D^3[1,2] = \min\{D^2[1,2], D^2[1,3] + D^2[3,2]\}$
$= \min\{4, 6+7\} = 4$

$D^3[1,3] = \min\{D^2[1,3], D^2[1,3] + D^2[3,3]\}$
$= \min\{6, 6+0\} = 6$

$D^3[2,1] = \min\{D^2[2,1], D^2[2,3] + D^2[3,1]\}$
$= \min\{6, 2+3\} = 5$

$D^3[2,3] = \min\{D^2[2,3], D^2[2,3] + D^2[3,3]\}$
$= \min\{2, 2+0\} = 2$

$D^3[3,1] = \min\{D^2[3,1], D^2[3,3] + D^2[3,1]\}$
$= \min\{3, 0+3\} = 3$

$D^3[3,2] = \min\{D^2[3,2], D^2[3,3] + D^2[3,2]\}$
$= \min\{7, 0+7\} = 7$

$$D^3 = \begin{array}{|c|c|c|}\hline 0 & 4 & 6 \\ \hline 5 & 0 & 2 \\ \hline 3 & 7 & 0 \\ \hline \end{array}$$

Ex. 4.7.2

Apply Floyd's method to find the shortest path for below mentioned all pairs.

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & \infty & 3 & \infty \\ 2 & 2 & 0 & \infty & \infty \\ 3 & \infty & 7 & 0 & 1 \\ 4 & 6 & \infty & \infty & 0 \end{array}$$

Soln. :

Optimal substructure formula for Floyd's algorithm,

$$Dk[i,j] = \min\{ D^{k-1}[i,j], D^{k-1}[i,k] + D^{k-1}[k,j] \}$$

$$D^0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & \infty & 3 & \infty \\ 2 & 2 & 0 & \infty & \infty \\ 3 & \infty & 7 & 0 & 1 \\ 4 & 6 & \infty & \infty & 0 \end{array}$$

Iteration 1 : k = 1

$D^1[1,2] = \min\{ D^0[1,2], D^0[1,1] + D^0[1,2] \}$

$= \min \{\infty, 0+\infty\} = \infty$

$D^1[1,3] = \min \{ D^0[1,3], D^0[1,1] + D^0[1,3] \}$
$= \min \{3, 0+3\} = 3$

$D^1[1,4] = \min \{ D^0[1,4], D^0[1,1] + D^0[1,4] \}$
$= \min \{\infty, 0+\infty\} = \infty$

$D^1[2,1] = \min \{ D^0[2,1], D^0[2,1] + D^0[1,1] \}$
$= \min \{2, 2+0\} = 2$

$D^1[2,3] = \min \{ D^0[2,3], D^0[2,1] + D^0[1,3] \}$
$= \min \{\infty, 2+3\} = 5$

$D^1[2,4] = \min \{ D^0[2,4], D^0[2,1] + D^0[1,4] \}$
$= \min \{\infty, 2+\infty\} = \infty$

$D^1[3,1] = \min \{ D^0[3,1], D^0[3,1] + D^0[1,1] \}$
$= \min \{\infty, 0+\infty\} = \infty$

$D^1[3,2] = \min \{ D^0[3,2], D^0[3,1] + D^0[1,2] \}$
$= \min \{7, \infty+\infty\} = 7$

$D^1[3,4] = \min \{ D^0[3,4], D^0[3,1] + D^0[1,4] \}$
$= \min \{1, \infty+\infty\} = 1$

$D^1[4,1] = \min \{ D^0[4,1], D^0[4,1] + D^0[1,1] \}$
$= \min \{6, 6+0\} = 6$

$D^1[4,2] = \min \{ D^0[4,2], D^0[4,1] + D^0[1,2] \}$
$= \min \{\infty, 6+\infty\} = \infty$

$D^1[4,3] = \min \{ D^0[4,3], D^0[4,1] + D^0[1,3] \}$
$= \min \{\infty, 6+3\} = 9$

$D^1 =$

	1	2	3	4
1	0	∞	3	∞
2	2	0	5	∞
3	∞	7	0	1
4	6	∞	9	0

Note : Path distance for the highlighted cell is an improvement over the original matrix.

Iteration 2 (k = 2) :

$D^2[1,2] = D^1[1,2] = \infty$

$D^2[1,3] = \min \{ D^1[1,3], D^1[1,2] + D^1[2,3] \}$
$= \min \{3, \infty+5\} = 3$

$D^2[1,4] = \min \{ D^1[1,4], D^1[1,2] + D^1[2,4] \}$
$= \min \{\infty, \infty+\infty\} = \infty$

$D^2[2,1] = D^1[2,1] = 2$

$D^2[2,3] = D^1[2,3] = 5$

$D^2[2,4] = D^1[2,4] = \infty$

$D^2[3,1] = \min \{ D^1[3,1], D^1[3,2] + D^1[2,1] \}$
$= \min \{\infty, 7+2\} = 9$

$D^2[3,2] = D^1[3,2] = 7$

$D^2[3,4] = \min \{ D^1[3,4], D^1[3,2] + D^1[2,4] \}$
$= \min \{1, 7+\infty\} = 1$

$D^2[4,1] = \min \{ D^1[4,1], D^1[4,2] + D^1[2,1] \}$
$= \min \{6, \infty+2\} = 6$

$D^2[4,2] = D^1[4,2] = \infty$

$D^2[4,3] = \min \{ D^1[4,3], D^1[4,2] + D^1[2,3] \}$
$= \min \{9, \infty+5\} = 9$

$D^2 =$

	1	2	3	4
1	0	∞	3	∞
2	2	0	5	∞
3	9	7	0	1
4	6	∞	9	0

Iteration 3 (k = 3) :

$D^3[1,2] = \min \{ D^2[1,2], D^2[1,3] + D^2[3,2] \}$
$= \min \{\infty, 3+7\} = 10$

$D^3[1,3] = D^2[1,3] = 3$

$D^3[1,4] = \min \{ D^2[1,4], D^2[1,3] + D^2[3,4] \}$
$= \min \{\infty, 3+1\} = 4$

$D^3[2,1] = \min \{ D^2[2,1], D^2[2,3] + D^2[3,1] \}$
$= \min \{2, 5+9\} = 2$

$D^3[2,3] = D^2[2,3] = 5$

$D^3[2,4] = \min \{ D^2[2,4], D^2[2,3] + D^2[3,4] \}$
$= \min \{\infty, 5+1\} = 6$

$D^3[3,1] = D^2[3,1] = 9$

$D^3[3,2] = D^2[3,2] = 7$

$D^3[3,4] = D^2[3,4] = 1$

$D^3[4,1] = \min \{ D^2[4,1], D^2[4,3] + D^2[3,1] \}$
$= \min \{6, 9+9\} = 6$

$D^3[4,2] = \min \{ D^2[4,2], D^2[4,3] + D^2[3,2] \}$
$= \min \{\infty, 9+7\} = 16$

$D^3[4,3] = D^2[4,3] = 9$

$D^3 =$

	1	2	3	4
1	0	10	3	4
2	2	0	5	6
3	9	7	0	1
4	6	16	9	0

Iteration 4 (k = 4) :

$D^4[1,2] = \min \{ D^3[1,2], D^3[1,4] + D^3[4,2] \}$
$= \min \{10, 4+16\} = 10$

$D^4[1,3] = \min \{ D^3[1,3], D^3[1,4] + D^3[4,1] \}$
$= \min \{3, 4+9\} = 3$

$D^4[1,4] = D^3[1,4] = 4$

$D^4[2,1] = \min \{ D^3[2,1], D^3[2,4] + D^3[4,1] \}$
$= \min \{2, 6+6\} = 2$

$D^4[2,3] = \min \{ D^3[2,3], D^3[2,4] + D^3[4,3] \}$
$= \min \{5, 6+9\} = 5$
$D^4[2,4] = D^3[2,4] = 6$
$D^4[3,1] = \min \{ D^3[3,1], D^3[3,4] + D^3[4,1] \}$
$= \min \{9, 1+6\} = 7$
$D^4[3,2] = \min \{ D^3[3,2], D^3[3,4] + D^3[4,2] \}$
$= \min \{7, 1+16\} = 7$
$D^4[3,4] = D^3[3,4] = 1$
$D^4[4,1] = D^3[4,1] = 6$
$D^4[4,2] = D^3[4,2] = 16$
$D^4[4,3] = D^3[4,3] = 9$

Final distance matrix is,

$D^4 =$

0	10	3	4
2	0	5	6
7	7	0	1
6	16	9	0

Ex. 4.7.3

Suppose Floyd-Warshall's algorithm is run on the weighted, directed graph shown below, show the values of the matrices that result from each iteration in the algorithm.

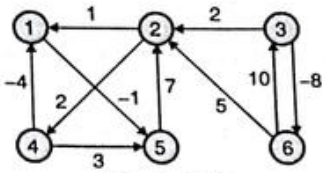


Fig. P. 4.7.3

Soln. :

Initial matrix D^0 for given graph is shown below :

$D^3 =$

	1	2	3	4	5	6
1	0	∞	∞	∞	−1	∞
2	1	0	∞	2	∞	∞∞∞
3	∞	2	0	∞	∞	−8
4	−4	∞	∞	0	3	∞
5	∞	7	∞	∞	0	∞
6	∞	5	10	∞	∞	0

Optimal substructure formula for Floyd's algorithm,

$D^k[i,j] = \min \{ D^{k-1}[i,j], D^{k-1}[i,k] + D^{k-1}[k,j] \}$

Values of matrices are shown below.

$D^1 =$

	1	2	3	4	5	6
1	0	∞	∞	∞	-1	∞
2	1	0	∞	2	0	∞
3	∞	2	0	∞	∞	-8
4	-4	∞	∞	0	-5	∞
5	∞	7	∞	∞	0	∞
6	∞	5	10	∞	∞	0

Iteration 1 : k = 1

$D^2 =$

	1	2	3	4	5	6
1	0	∞	∞	∞	-1	∞
2	1	0	∞	2	0	∞
3	3	2	0	4	2	-8
4	-4	∞	∞	0	-5	∞
5	8	7	∞	9	0	∞
6	6	5	10	7	5	0

Iteration 2 : k = 2

$D^3 =$

	1	2	3	4	5	6
1	0	∞	∞	∞	-1	∞
2	1	0	∞	2	0	∞
3	3	2	0	4	2	-8
4	-4	∞	∞	0	-5	∞
5	8	7	∞	9	0	∞
6	6	5	10	7	5	0

Iteration 3 : k = 3

$D^4 =$

	1	2	3	4	5	6
1	0	∞	∞	∞	-1	∞
2	1	0	∞	2	0	∞
3	0	2	0	4	2	-8
4	-4	∞	∞	0	-5	∞
5	5	7	∞	9	0	∞
6	3	5	10	7	2	0

Iteration 4 : k = 4

$D^5 =$

	1	2	3	4	5	6
1	0	6	∞	8	-1	∞
2	1	0	∞	2	0	∞
3	0	2	0	4	2	-8
4	-4	2	∞	0	-5	∞
5	5	7	∞	9	0	∞
6	3	5	10	7	2	0

Iteration 5 : k = 5

$D^6 =$

	1	2	3	4	5	6
1	0	6	∞	8	-1	∞
2	1	0	∞	2	0	∞
3	-5	-3	0	-1	-6	-8
4	-4	2	∞	0	-5	∞
5	5	7	∞	9	0	∞
6	3	5	10	7	2	0

Iteration 6 : k = 6

Ex. 4.7.4

Obtain all pair shortest path using Floyd's Algorithm for given weighted graph.

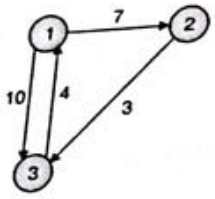


Fig. P. 4.7.4

Soln. :

Optimal substructure formula for Floyd's algorithm,

$D^k[i,j] = \min\{ D^{k-1}[i,j], D^{k-1}[i,k] + D^{k-1}[k,j] \}$

$$D^0 = \begin{array}{|c|c|c|}\hline 0 & 8 & 5 \\ \hline 2 & 0 & \infty \\ \hline \infty & 1 & 0 \\ \hline \end{array}$$

Iteration 1 (k = 1)

$D^1[1,2] = \min\{D^0[1,2], D^0[1,1] + D^0[1,2]\}$
$= \min\{8, 0+8\} = 8$

$D^1[1,3] = \min\{D^0[1,3], D^0[1,1] + D^0[1,3]\}$
$= \min\{5, 0+5\} = 5$

$D^1[2,1] = \min\{D^0[2,1], D^0[2,1] + D^0[1,1]\}$
$= \min\{2, 2+0\} = 2$

$D^1[2,3] = \min\{D^0[2,3], D^0[2,1] + D^0[1,3]\}$
$= \min\{\infty, 2+5\} = 7$

$D^1[3,1] = \min\{D^0[3,1], D^0[3,1] + D^0[1,1]\}$
$= \min\{\infty, \infty+0\} = \infty$

$D^1[3,2] = \min\{D^0[3,2], D^0[3,1] + D^0[1,2]\}$
$= \min\{1, 1+8\} = 1$

$$D^1 = \begin{array}{|c|c|c|}\hline 0 & 8 & 5 \\ \hline 2 & 0 & 7 \\ \hline \infty & 1 & 0 \\ \hline \end{array}$$

Iteration 2 (k = 2)

$D^2[1,2] = \min\{D^1[1,2], D^1[1,2] + D^1[2,2]\}$
$= \min\{8, 8+0\} = 8$

$D^2[1,3] = \min\{D^1[1,3], D^1[1,2] + D^1[2,3]\}$
$= \min\{5, 8+7\} = 5$

$D^2[2,1] = \min\{D^1[2,1], D^1[2,2] + D^1[2,1]\}$
$= \min\{2, 0+2\} = 2$

$D^2[2,3] = \min\{D^1[2,3], D^1[2,2] + D^1[2,3]\}$
$= \min\{7, 0+7\} = 7$

$D^2[3,1] = \min\{D^1[3,1], D^1[3,2] + D^1[2,1]\}$
$= \min\{\infty, 1+2\} = 3$

$D^2[3,2] = \min\{D^1[3,2], D^1[3,2] + D^1[2,2]\}$
$= \min\{1, 1+0\} = 1$

$$D^2 = \begin{array}{|c|c|c|}\hline 0 & 8 & 5 \\ \hline 2 & 0 & 7 \\ \hline 3 & 1 & 0 \\ \hline \end{array}$$

Iteration 3 (k = 3)

$D^3[1,2] = \min\{D^2[1,2], D^2[1,3] + D^2[3,2]\}$
$= \min\{8, 5+1\} = 6$

$D^3[1,3] = \min\{D^2[1,3], D^2[1,3] + D^2[3,3]\}$
$= \min\{5, 5+0\} = 5$

$D^3[2,1] = \min\{D^2[2,1], D^2[2,3] + D^2[3,1]\}$
$= \min\{2, 7+3\} = 2$

$D^3[2,3] = \min\{D^2[2,3], D^2[2,3] + D^2[3,3]\}$
$= \min\{7, 7+0\} = 7$

$D^3[3,1] = \min\{D^2[3,1], D^2[3,3] + D^2[3,1]\}$
$= \min\{3, 0+3\} = 3$

$D^3[3,2] = \min\{D^2[3,2], D^2[3,3] + D^2[3,2]\}$
$= \min\{1, 0+1\} = 1$

$$D^3 = \begin{array}{|c|c|c|}\hline 0 & 6 & 5 \\ \hline 2 & 0 & 7 \\ \hline 3 & 1 & 0 \\ \hline \end{array}$$

Syllabus Topic : Assembly-Line Scheduling

4.8 Assembly Line Scheduling

Q. Explain Assembly line scheduling. (7 Marks)

- Assembly line scheduling is a manufacturing problem. In automobile industries, assembly lines are used to transfer parts from one station to another.
- Manufacturing of large items like car, trucks etc generally undergoes through multiple stations, where each station is responsible for assembling particular part only. Entire product will be ready after it goes through predefined n stations in sequence.
- For example, manufacturing of car may be done in several stages like engine fitting, colouring, light fitting, fixing of controlling system, gates, seats and many other things.
- The particular task is carried out at the station dedicated to that task only. Based on requirement, there may be more than one assembly line.
- In case of two assembly lines, if the load at station j of assembly line 1 is very high, then components are transferred to station j of assembly line 2, the converse is also true. This helps to speed up the manufacturing process.
- The time to transfer partial product from one station to next station on the same assembly line is negligible. During rush, factory manager may transfer partially completed auto from one assembly line to another, to

complete the manufacturing as quickly as possible. Some penalty of time t occurs when the product is transferred from assembly 1 to 2 or 2 to 1.

☞ Problem

- Determine which station should be selected from assembly line 1 and which to choose from assembly line 2 in order to minimize the total time to build the entire product.
- Selecting stations by brute force attack is quite infeasible. If there are n stations, unfortunately, there are 2^n possible ways to choose stations. Brute force attack takes $\Omega(2^n)$ time, it is not acceptable when n is large.
- General architecture of two assembly lines with n stations is shown in Fig. 4.8.1. Terminologies are explained here :
  - S_{ij} = Station j on assembly line i.
  - a_{ij} = Time needed to assemble the partial component at station j on assembly line i.

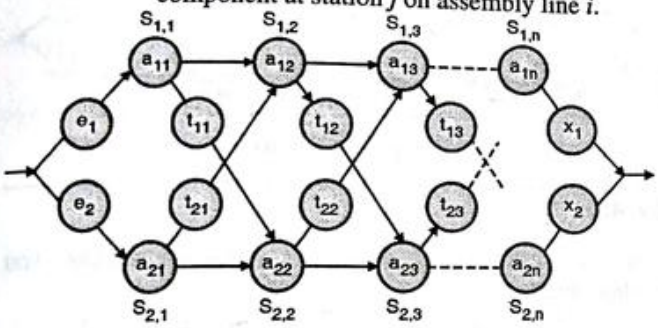


Fig. 4.8.1 : Architecture of assembly line

  - t_{ij} = Time required to transfer component from one assembly to other from station j to (j + 1).
  - e_i = Entry time on assembly line i.
  - x_i = Exit time from assembly line i.

☞ Mathematical formulation

The time required to build component on the first station of both lines is a summation of entry time for the particular assembly line and assembly time on the first station of a particular line.

$f_i[j] = e_i + a_{i1}$; Initial condition (if j == 1), i = 1, 2

Minimum time required to put the partially assembled product out of any station j on line 1 is minimum of :

1. (Minimum time taken up to station j – 1 on same assembly line 1) + (assembly time on station j of same assembly line)
2. (Minimum time taken up to station j – 1 on assembly line 2) + (assembly line transfer time) + (assembly time on station j of assembly line 2)

We can derive the conclusion for assembly line 2 in similar way. So mathematically we can formulate the problem as :

For j > 1 :

$f_{1[j]} = \min (f_1[j-1] + a_1, f_2[j-1] + t_{2j-1} + a_{1j})$

$f_{2[j]} = \min (f_1[j-1] + t_{1j-1} + a_{2j}, f_2[j-1] + a_{2j})$

Exit time from assembly line 1 and 2 are x_1 and x_2 respectively. Thus, minimum time to take away the product from assembly line is :

$f^* = \min (f_1[n] + x_1, f_2[n] + x_2)$

Algorithm for assembly line scheduling using dynamic programming is described below :

```
Algorithm ASSEMBLY_LINE_SCHEDULING(n, e, a, t, x)
// n is number of stations on both assembly
// e is array of entry time on assembly
// a is array of assembly time on given station
// t is array of the time required to change assembly line
//x is array of exit time from assembly

f1 [1]← e1 + a1,1
f2 [1]← e2 + a2,1
for j ← 2 to n do
    if f1 [j – 1] + a1,j ≤ f2 [j – 1] + t2,j–1 + a2,j then
        f1 [j] ← f1 [j – 1] + a1,j
        L1 [j] ← 1
    else
        f1 [j] ← f2[j – 1] + t2,j–1 + a1,j
        L1 [j]← 2
    end

    if f2 [j – 1] + a2,j ≤ f1 [j – 1] + t1,j–1 + a2,j then
        f2 [j] ← f2 [j – 1] + a2,j
        l2 [j] ← 2
    else
        f2 [j] ← f1 [j – 1] + t1,j–1 + a2,j
        l2 [j] ← 1
    end
    if f1[n] + x1 ≤ f2 [n] + x2 then
        F* ← f1 [n] + x1
        L* ← 1
    else
        F* ← f2 [n] + x2
        L* ← 2
    end
end
```

- Above algorithm will just tell us the minimum time required to take away product from station n. It does not tell anything about the sequence of stations on assembly. We can find the station sequence by tracing the solution using the following algorithm.

```
Algorithm PRINT_STATIONS(l, n)
i ← l*
print "line " i ", station " n
```

```
for j ← n down to 2 do
    i ← l[j]
    print "line " i ",station " j – 1
end
```

Ex. 4.8.1

Find optimal assembly line schedule for given data.

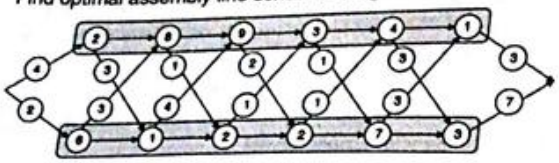


Fig. P. 4.8.1

Soln. :

$f_i[j] = e_i + a_{i,1}$; Initial condition (j = 1, i=1, 2)

$f_1[j] = \min(f_1[j-1] + a_{1j}, f_2[j-1] + t_{2j-1}) + a_{1j})$

$f_2[j] = \min(f_1[j-1] + t_{1j-1}) + a_{2j}, f_2[j-1] + a_{2j})$

$f^* = \min(f_1[n] + x_1, f_2[n] + x_2)$

$f_1[1] = e_1 + a_{11} = 4 + 2 = 6$

$f_2[1] = e_2 + a_{21} = 2 + 6 = 8$

$f_1[2] = \min(f_1[j-1] + a_{1j}, f_2[j-1] + t_{2j-1} + a_{1j})$

$= \min(f_1[1] + a_{12}, f_2[1] + t_{21} + a_{12})$

$= \min(6 + 8, 8 + 3 + 8)$

$= \min(14, 19) = 14$

$f_2[2] = \min(f_1[j-1] + t_{j-1} + a_{2j}, f_2[j-1] + a_{2j})$

$= \min(f_1[1] + t_{11} + a_{22}, f_2[1] + a_{22})$

$= \min(6 + 3 + 11, 8 + 11) = \min(20, 19) = 19$

$f_1[3] = \min(f_1[j-1] + a_{1j}, f_2[j-1] + t_{2j-1}) + a_{1j})$

$= \min(f_1[2] + a_{13}, f_2[2] + t_{22} + a_{13})$

$= \min(14 + 9, 19 + 4 + 9)$

$= \min(23, 32) = 23$

$f_2[3] = \min(f_1[j-1] + t_{j-1} + a_{2j}, f_2[j-1] + a_{2j})$

$= \min(f_1[2] + t_{12} + a_{23}, f_2[2] + a_{23})$

$= \min(14 + 1 + 2, 19 + 2)$

$= \min(17, 21) = 17$

$f_1[4] = \min(f_1[j-1] + a_{1j}, f_2[j-1] + t_{2j-1} + a_{1j})$

$= \min(f_1[3] + a_{14}, f_2[3] + t_{23} + a_{14})$

$= \min(23 + 3, 17 + 1 + 3) = \min(26, 21)$

$= 21$

$f_2[4] = \min(f_1[j-1] + t_{j-1} + a_{2j}, f_2[j-1] + a_{2j})$

$= \min(f_1[3] + t_{13} + a_{24}, f_2[3] + a_{24})$

$= \min(23 + 2 + 2, 17 + 2)$

$= \min(27, 19) = 19$

Hence we can summarize $f_{1(j)}$ and $f_{2(j)}$ as follows :

	j = 1	j = 2	j= 3	j = 4	j = 5	j = 6
$f_1[j]$	6	14	23	21	24	25
$f_2[j]$	8	17	19	26	29	29

$f^* = \min(f_1[n] + x_1, f_2[n] + x_2)$

$f^* = \min(25 + 3, 29 + 7) = \min(28, 36) = 28$

Compute fastest path

J	2	3	4	5	6	
$l_1[j]$	1	1	2	2	1	$l^* = 1$
$l_2[j]$	2	1	2	2	2	

Trace back the path

- $l^* = 1$, so we have to select last station $S_{1,6}$ on line 1. Now we will look at $l_1[6]$, which is 1, so we will select the previous station on line 1 that is station $S_{1,5}$.
- Now we will look at $l_{1,5}$ which is 2 so we will change the line to 2 and previous station that is $S_{2,4}$.
- Next, observe $l_2[4]$, it's 2 so we have to remain on the same line and select station $S_{2,3}$. Now observe $l_2[3]$, which is 1, so move on station $S_{1,2}$ on line 1. Now observe $l_1[3]$, which is 1, so remain on the same line.
- Final path is shown in Fig. 4.8.1(a).

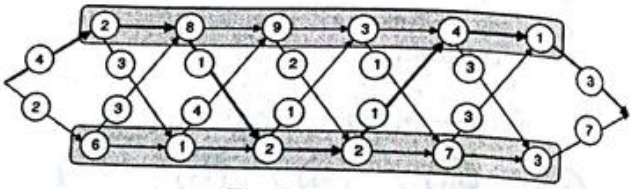


Fig. P. 4.8.1(a)

Ex. 4.8.2

Find an optimal solution for given assembly line configuration.

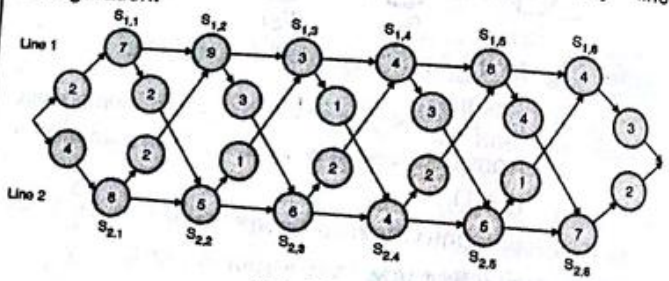


Fig. P. 4.8.2

Soln. :

$f_i[1] = e_i + a_{i1}$, for i = 1, 2

Here, i indicates assembly line, i.e. 1 or 2,

$f_1[1] = e_1 + a_{11} = 2 + 7 = 9$

$f_2[1] = e_2 + a_{21} = 4 + 8 = 12$

For j > 1, where j indicates station numbers.

$f_1[j] = \min(f_1[j-1] + a_{1j}, f_2[j-1] + t_{2,j-1} + a_{1j})$

$f_2[1] = \min(f_1[j-1] + t_{1,j-1} + a_{2j}, f_2[j-1] + a_{2j})$

For j = 2

$f_1[2] = \min(f_1[1] + a_{12}, f_2[1] + t_{21} + a_{12})$

$= \min(9 + 9, 12 + 2 + 9) = 18$

$f_2[2] = \min(f_1[1] + t_{11} + a_{22}, f_2[1] + a_{22})$

$= \min(9 + 2 + 5, 12 + 5) = 16$

For j = 3

$f_1[3] = \min(f_1[2] + a_{13}, f_2[2] + t_{22} + a_{13})$

$= \min(18 + 3, 16 + 1 + 3) = 20$

$f_2[3] = \min(f_1[2] + t_{12} + a_{23}, f_2[2] + a_{23})$

$= \min(18 + 3 + 6, 16 + 6) = 22$

For j = 4

$f_1(4) = \min(f_1[3] + a_{14}, f_2[3] + t_{23} + a_{14})$

$= \min(20 + 4, 22 + 2 + 4) = 24$

$f_2(4) = \min(f_1[3] + t_{13} + a_{24}, f_2[3] + a_{24})$

$= \min(20 + 1 + 4, 22 + 4) = 25$

For j = 5

$f_1(5) = \min(f_1[4] + a_{15}, f_2[4] + t_{24} + a_{15})$

$= \min(24 + 8, 25 + 2 + 8) = 32$

$f_2(5) = \min(f_1[4] + t_{14} + a_{25}, f_2[4] + a_{25})$

$= \min(24 + 3 + 5, 25 + 5) = 30$

For j = 6

$f_1(6) = \min(f_1[5] + a_{16}, f_2[5] + t_{25} + a_{16})$

$= \min(32 + 4, 30 + 1 + 4) = 35$

$f_2(6) = \min(f_1[5] + t_{15} + a_{26}, f_2[5] + a_{26})$

$= \min(32 + 4 + 7, 30 + 7) = 37$

$f^* = \min(f_1(n) + x_1, f_2(n) + x_2)$

where n = number of stations = 6

x_1 and x_2 are exit time from line 1 and line 2 respectively.

$x_1 = 3, \quad x_2 = 2$

$f^* = \min(f_1[6] + x_1, f_2[6] + x_2)$

$= \min(35 + 3, 37 + 2) = 38$

$f_i[j]$	9	18	20	24	32	35	
$f_2[j]$	12	16	22	25	30	37	$f^* = 38$
J	1	2	3	4	5	6	

$l_1[j]$	1	2	1	1	2
$l_2[j]$	1	2	1	2	2

$l^* = 1$

Syllabus Topic : 0/1 knapsack

4.9 0/1 Knapsack

➔ **(May 14, Dec. 15)**

Q. Explain 0/1 Knapsack Problem with an example. **MU - May 2014, 10 Marks**

Q. Explain 0/1 knapsack problem using dynamic programming. **MU - Dec. 2015, 10 Marks**

Q. Define the knapsack problem. How to solve it using dynamic programming? **(5 Marks)**

☞ **Problem**

- Given a set of items, each having different weight and value or profit associated with it. Find the set of items such that the total weight is less than or equal to a capacity of the knapsack and the total value earned is as large as possible.
- The knapsack problem is useful in solving resource allocation problem. Let $X = \langle x_1, x_2, x_3, \ldots, x_n \rangle$ be the set of n items. Sets $W = \langle w_1, w_2, w_3, \ldots, w_n \rangle$ and $V = \langle v_1, v_2, v_3, \ldots, v_n \rangle$ are weight and value associated with each item in X. Knapsack capacity is M unit. The knapsack problem is to find the set of items which maximizes the profit such that collective weight of selected items does not cross the knapsack capacity.
- Select items from X and fill the knapsack such that it would maximize the profit. Knapsack problem has two variations. 0/1 knapsack, that does not allow breaking of items. Either add an entire item or reject it. It is also known as a binary knapsack. Fractional knapsack allows breaking of items. Profit will be earned proportionally.

☞ **Mathematical formulation**

We can select any item only ones. We can put items x_i in knapsack if knapsack can accommodate it. If the item is added to a knapsack, associated profit is accumulated.

Objective is to maximize $\sum_{i=1}^{n} = x_i v_i$, Subjected to

$$\sum_{i=1}^{n} x_i w_i \leq M$$

Where,

v_i = Profit associated with i^{th} item

w_i = Weight of i^{th} item

M = Capacity of knapsack

x_i = Fraction of i^{th} item

= $\{0, 1\} \rightarrow 0 / 1$ knapsack

= $[0, 1] \rightarrow$ fractional knapsack.

We will discuss two approaches for solving knapsack using dynamic programming.

4.9.1 First Approach

Q. How to solve knapsack problem using dynamic programming? **(5 Marks)**

If the weight of the item is larger than remaining knapsack capacity, we skip the item and solution of the previous step remain as it is. Otherwise, we should add the item to solution set and problem size will be reduced by the weight of that item. Corresponding profit will be added to the selected item.

Dynamic programming divides the problem into small subproblems. Let V is an array of the solution of subproblems. V[i, j] represents the solution for problem size j with first i items.

Mathematical notion of knapsack problem is given as :

$$V[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ V[i-1, j] & \text{if } j < wi \\ \max\{V[i-1, j], vi + V[i-1, j-wi]\} & \text{if } j \geq w \end{cases}$$

V [1 .... n, 0 ... M] : Size of the table

V (n, M) = Solution

n = Number of items

Algorithm for binary knapsack is described below :

```
Algorithm DP_BINARY_KNAPSACK (V, W, M)
// Description : Solve binary knapsack problem using dynamic
    programming
// Input : Set of items X, set of weight W, profit of items V and
    knapsack capacity M

// Output : Array V, which holds the solution of problem

for i← 1 to n do
    V[i, 0] ← 0
end
for i← 1 to M do
    V[0, i] ← 0
end

for i← 1 to n do
    for j ← 0 to M do
        if w[i] ≤ j
            V[i, j] ← max{V[i-1, j], v[i]+V[i – 1, j – w[i]]}
        else
            V[i, j] ← V[i – 1, j]
        // w[i]> j
        end
    end
end
```

Initialization

Solution after adding i^{th} item

i^{th} item is not feasible, so select old solution

Above algorithm will just tell us the maximum value we can earn with the dynamic programming. It does not speak anything about which items should be selected. We can find the items that give optimum result using the following algorithm.

```
Algorithm TRACE_KNAPSACK(w, v, M)
// w is array of weight of n items
// v is array of value of n items
// M is the knapsack capacity
SW ←{ }
SP ←{ }
i ← n
j ← M
while( j > 0 ) do
    if (V[i, j] == V([i – 1, j] then
        i ← i – 1
    else
        V[i, j] ← V[i, j] – vi
        j ← j – w[i]
        SW ← SW +w[i]
        SP ← SP +v[i]
    end
end
```

If i^{th} item is not selected

If i^{th} item is selected

Complexity analysis

With n items, there exist 2^n subsets, brute force approach examines all sub sets to find optimal solution. Hence, running time of brute force approach is $O(2^n)$. This is unacceptable for large n.

Dynamic programming finds an optimal solution by constructing a table of size n × M, where n is a number of items and M is the capacity of the knapsack. This table can be filled up in O(nM) time, same is the space complexity.

- Running time of Brute force approach is $O(2^n)$.
- Running time using dynamic programming with memorization is O(n * M).

Ex. 4.9.1

Consider 0/1 knapsack problem number of items N = 3, w = (4, 6, 8), p = (10, 12, 15) using dynamic programming device the recurrence relations for the problem and solve the same. Determine the optimal profit for the knapsack of capacity M = 10.

Soln. :

Here, knapsack capacity is sufficiently small, so we can use the first approach to solve the problem.

Solution of knapsack problem is defined as,

$$V[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ V[i-1, j] & \text{if } j < wi \\ \max\{V[i-1, j], vi + V[i-1, j-wi]\} & \text{if } j \geq w \end{cases}$$

We have following stats about problem,

Table P. 4.9.1

Item	Weight (w_i)	Value (v_i)
I_1	4	10
I_2	6	12
I_3	8	15

Boundary conditions would be V [0, j] = V[i, 0] = 0.
Initial configuration of table looks like,

	Item detail		j 0	1	2	3	4	5	6	7	8	9	10
i=0			0	0	0	0	0	0	0	0	0	0	0
i=1	$w_1 = 4$	$v_1 = 10$	0										
i=2	$w_2 = 6$	$v_2 = 12$	0										
i=3	$w_3 = 8$	$v_3 = 15$	0										

Filling first column, j = 1

$V[1, 1] \Rightarrow i = 1,\ j = 1, w_i = w_1 = 4$

As, $j < w_i$, $V[i, j] = V[i - 1, j]$

$\therefore\ V[1, 1] = V[0, 1] = 0$

$V[2, 1] \Rightarrow i = 2,\ j = 1, w_i = w_2 = 6$

As, $j < w_i$, $V[i, j] = V[i - 1, j]$

$\therefore\ V[2, 1] = V[1, 1] = 0$

$V[3, 1] \Rightarrow i = 3,\ j = 1, w_i = w_3 = 8$

As, $j < w_i$, $V[i, j] = V[i - 1, j]$

$\therefore\ V[3, 1] = V[2, 1] = 0$

	Item detail		j 0	1	2	3	4	5	6	7	8	9	10
i=0			0	0	0	0	0	0	0	0	0	0	0
i=1	$w_1 = 4$	$v_1 = 10$	0	0									
i=2	$w_2 = 6$	$v_2 = 12$	0	0									
i=3	$w_3 = 8$	$v_3 = 15$	0	0									

Filling second column, j = 2

$V[1, 2] \Rightarrow i = 1,\ j = 2,\ w_i = w_1 = 4$

As, $j < w_i$, $V[i, j] = V[i - 1, j]$

$\therefore\ V[1, 2] = V[0, 2] = 0$

$V[2, 2] \Rightarrow i = 2,\ j = 2,\ w_i = w_2 = 6$

As, $j < w_i$, $V[i, j] = V[i - 1, j]$

$\therefore\ V[2, 2] = V[1, 2] = 0$

$V[3, 2] \Rightarrow i = 3,\ j = 2,\ w_i = w_3 = 8$

As, $j < w_i$, $V[i, j] = V[i - 1, j]$

$\therefore\ V[3, 2] = V[2, 2] = 0$

	Item detail		j 0	1	2	3	4	5	6	7	8	9	10
i=0			0	0	0	0	0	0	0	0	0	0	0
i=1	$w_1 = 4$	$v_1 = 10$	0	0	0								
i=2	$w_2 = 6$	$v_2 = 12$	0	0	0								
i=3	$w_3 = 8$	$v_3 = 15$	0	0	0								

Filling second column, j = 3

$V[1, 3] \Rightarrow i = 1,\ j = 3, w_i = w_1 = 4$

As, $j < w_i$, $V[i, j] = V[i - 1, j]$

$\therefore\ V[1, 3] = V[0, 3] = 0$

$V[2, 3] \Rightarrow i = 2,\ j = 3,\ w_i = w_2 = 6$

As, $j < w_i$, $V[i, j] = V[i - 1, j]$

$\therefore\ V[2, 3] = V[1, 3] = 0$

$V[3, 3] \Rightarrow i = 3,\ j = 3,\ w_i = w_3 = 8$

As, $j < w_i$, $V[i, j] = V[i - 1, j]$

$\therefore\ V[3, 3] = V[2, 3] = 0$

	Item detail		j 0	1	2	3	4	5	6	7	8	9	10
i=0			0	0	0	0	0	0	0	0	0	0	0
i=1	$w_1 = 4$	$v_1 = 10$	0	0	0	0							
i=2	$w_2 = 6$	$v_2 = 12$	0	0	0	0							
i=3	$w_3 = 8$	$v_3 = 15$	0	0	0	0							

Filling second column, j = 4

$V[1, 4] \Rightarrow i = 1,\ j = 4,\ w_i = w_1 = 4, v_1 = 10$

As, $j \geq w_i$, $V[i, j] = \max\{V[i - 1, j], v_i + V[i - 1, j - w_i]\}$

$= \max\{V[0, 4], 10 + V[0, 0]\}$

$V[1, 4] = \max(0, 10 + 0) = 10$

$V[2, 4] \Rightarrow i = 2,\ j = 4,\ w_i = w_2 = 6, v_2 = 12$

As, $j < w_i$, $V[i, j] = V[i - 1, j]$

$\therefore\ V[2, 4] = V[1, 4] = 10$

$V[3, 4] \Rightarrow i = 3,\ j = 4,\ w_i = w_3 = 8, v_3 = 15$

As, $j < w_i$, $V[i, j] = V[i - 1, j]$

$\therefore\ V[3, 4] = V[2, 4] = 10$

	Item detail		j 0	1	2	3	4	5	6	7	8	9	10
i=0			0	0	0	0	0	0	0	0	0	0	0
i=1	$w_1 = 4$	$v_1 = 10$	0	0	0	0	10						
i=2	$w_2 = 6$	$v_2 = 12$	0	0	0	0	10						
i=3	$w_3 = 8$	$v_3 = 15$	0	0	0	0	10						

Filling second column, j = 5

$V[1, 5] \Rightarrow i = 1,\ j = 5,\ w_i = w_1 = 4, v_1 = 10$

As, $j \geq w_i$, $V[i, j] = \max\{V[i - 1, j], v_i + V[i - 1, j - w_i]\}$

$= \max\{V[0, 5], 10 + V[0, 1]\}$

$V[1, 5] = \max(0, 10 + 0) = 10$

$V[2, 5] \Rightarrow i = 2,\ j = 5,\ w_i = w_2 = 6, v_2 = 12$

As, $j < w_i$, $V[i, j] = V[i - 1, j]$

$\therefore\ V[2, 5] = V[1, 5] = 10$

$V[3, 5] \Rightarrow i = 3,\ j = 5,\ w_i = w_3 = 8, v_3 = 15$

As, $j < w_i$, $V[i, j] = V[i - 1, j]$

$\therefore\ V[3, 5] = V[2, 5] = 10$

	Item detail		0	1	2	3	4	5	6	7	8	9	10
i=0			0	0	0	0	0	0	0	0	0	0	0
i=1	$w_1 = 4$	$v_1 = 10$	0	0	0	0	10	10					
i=2	$w_2 = 6$	$v_2 = 12$	0	0	0	0	10	10					
i=3	$w_3 = 8$	$v_3 = 15$	0	0	0	0	10	10					

Filling second column, j = 6

$V[1, 6] \Rightarrow i = 1,\ j = 6,\ w_i = w_1 = 4,\ v_1 = 10$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j],\ v_i + V[i-1, j-w_i]\}$

$= \max \{V[0, 6],\ 10 + V[0, 2]\}$

$V[1, 6] = \max (0,\ 10 + 0) = 10$

$V[2, 6] \Rightarrow i = 2,\ j = 6,\ w_i = w_2 = 6,\ v_2 = 12$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j],\ v_i + V[i-1, j-w_i]\}$

$= \max \{V[1, 6],\ 12 + V[1, 0]\}$

$V[2, 6] = \max (10,\ 12 + 0) = 12$

$V[3, 6] \Rightarrow i = 3,\ j = 6,\ w_i = w_3 = 8,\ v_3 = 15$

As, $j < w_i$, $V[i, j] = V[i-1, j]$

$\therefore\ V[3, 6] = V[2, 6] = 12$

	Item detail		0	1	2	3	4	5	6	7	8	9	10
i=0			0	0	0	0	0	0	0	0	0	0	0
i=1	$w_1 = 4$	$v_1 = 10$	0	0	0	0	10	10	10				
i=2	$w_2 = 6$	$v_2 = 12$	0	0	0	0	10	10	12				
i=3	$w_3 = 8$	$v_3 = 15$	0	0	0	0	10	10	12				

Filling second column, j = 7

$V[1, 7] \Rightarrow i = 1,\ j = 7,\ w_i = w_1 = 4,\ v_1 = 10$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j],\ v_i + V[i-1, j-w_i]\}$

$= \max \{V[0, 7],\ 10 + V[0, 3]\}$

$V[1, 7] = \max (0,\ 10 + 0) = 10$

$V[2, 7] \Rightarrow i = 2,\ j = 7,\ w_i = w_2 = 6,\ v_2 = 12$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j],\ v_i + V[i-1, j-w_i]\}$

$= \max \{V[1, 7],\ 12 + V[1, 1]\}$

$V[2, 7] = \max (10,\ 12 + 0) = 12$

$V[3, 7] \Rightarrow i = 3,\ j = 7,\ w_i = w_3 = 8,\ v_3 = 15$

As, $j < w_i$, $V[i, j] = V[i-1, j]$

$\therefore\ V[3, 7] = V[2, 7] = 12$

	Item detail		0	1	2	3	4	5	6	7	8	9	10
i=0			0	0	0	0	0	0	0	0	0	0	0
i=1	$w_1 = 4$	$v_1 = 10$	0	0	0	0	10	10	10	10			
i=2	$w_2 = 6$	$v_2 = 12$	0	0	0	0	10	10	12	12			
i=3	$w_3 = 8$	$v_3 = 15$	0	0	0	0	10	10	12	12			

Filling second column, j = 8

$V[1, 8] \Rightarrow i = 1,\ j = 8,\ w_i = w_1 = 4,\ v_1 = 10$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j],\ v_i + V[i-1, j-w_i]\}$

$= \max \{V[0, 8],\ 10 + V[0, 4]\}$

$V[1, 8] = \max (0,\ 10 + 0) = 10$

$V[2, 8] \Rightarrow i = 2,\ j = 8,\ w_i = w_2 = 6,\ v_2 = 12$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j],\ v_i + V[i-1, j-w_i]\}$

$= \max \{V[1, 8],\ 12 + V[1, 2]\}$

$V[2, 8] = \max (10,\ 12 + 0) = 12$

$V[3, 8] \Rightarrow i = 3,\ j = 8,\ w_i = w_3 = 8,\ v_3 = 15$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j],\ v_i + V[i-1, j-w_i]\}$

$= \max \{V[2, 8],\ 15 + V[2, 0]\}$

$V[3, 8] = \max (12,\ 15 + 0) = 15$

	Item detail		0	1	2	3	4	5	6	7	8	9	10
i=0			0	0	0	0	0	0	0	0	0	0	0
i=1	$w_1 = 4$	$v_1 = 10$	0	0	0	0	10	10	10	10	10		
i=2	$w_2 = 6$	$v_2 = 12$	0	0	0	0	10	10	12	12	12		
i=3	$w_3 = 8$	$v_3 = 15$	0	0	0	0	10	10	12	12	15		

Filling second column, j = 9

$V[1, 9] \Rightarrow i = 1,\ j = 9,\ w_i = w_1 = 4,\ v_1 = 10$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j],\ v_i + V[i-1, j-w_i]\}$

$= \max \{V[0, 9],\ 10 + V[0, 5]\}$

$V[1, 9] = \max (0,\ 10 + 0) = 10$

$V[2, 9] \Rightarrow i = 2,\ j = 9,\ w_i = w_2 = 6,\ v_2 = 12$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j],\ v_i + V[i-1, j-w_i]\}$

$= \max \{V[1, 9],\ 12 + V[1, 3]\}$

$V[2, 9] = \max (10,\ 12 + 0) = 12$

$V[3, 9] \Rightarrow i = 3,\ j = 9,\ w_i = w_3 = 8,\ v_3 = 15$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j],\ v_i + V[i-1, j-w_i]\}$

$= \max \{V[2, 9],\ 15 + V[2, 1]\}$

$V[3, 9] = \max (12,\ 15 + 0) = 15$

	Item detail		0	1	2	3	4	5	6	7	8	9	10
i=0			0	0	0	0	0	0	0	0	0	0	0
i=1	$w_1 = 4$	$v_1 = 10$	0	0	0	0	10	10	10	10	10	10	
i=2	$w_2 = 6$	$v_2 = 12$	0	0	0	0	10	10	12	12	12	12	
i=3	$w_3 = 8$	$v_3 = 15$	0	0	0	0	10	10	12	12	15	15	

Filling second column, j = 10

$V[1, 10] \Rightarrow i = 1,\ j = 10,\ w_i = w_1 = 4,\ v_1 = 10$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j],\ v_i + V[i-1, j-w_i]\}$

$= \max \{V[0, 10],\ 10 + V[0, 6]\}$

$V[1, 10] = \max (0,\ 10 + 0) = 10$

$V[2, 10] \Rightarrow i = 2,\ j = 10,\ w_i = w_2 = 6,\ v_2 = 12$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j],\ v_i + V[i-1, j-w_i]\}$

$= \max \{V[1, 10],\ 12 + V[1, 4]\}$

$V[2, 10] = \max (10,\ 12 + 10) = 22$

$V[3, 10] \Rightarrow i = 3,\ j = 10,\ w_i = w_3 = 8,\ v_3 = 15$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j], v_i + V[i-1, j-w_i]\}$

$= \max \{V[2, 10], 15 + V[2, 2]\}$

$V[3, 10] = \max (22, 15 + 0) = 22$

	Item detail		0	1	2	3	4	5	6	7	8	9	10
i=0			0	0	0	0	0	0	0	0	0	0	0
i=1	$w_1 = 4$	$v_1 = 10$	0	0	0	0	10	10	10	10	10	10	10
i=2	$w_2 = 6$	$v_2 = 12$	0	0	0	0	10	10	12	12	12	12	22
i=3	$w_3 = 8$	$v_3 = 15$	0	0	0	0	10	10	12	12	15	15	22

Find selected items for M = 10

Step 1 : Initially, $i = n = 3$, $j = M = 10$

	Item detail		0	1	2	3	4	5	6	7	8	9	10
i=0			0	0	0	0	0	0	0	0	0	0	0
i=1	$w_1 = 4$	$v_1 = 10$	0	0	0	0	10	10	10	10	10	10	10
i=2	$w_2 = 6$	$v_2 = 12$	0	0	0	0	10	10	12	12	12	12	22
i=3	$w_3 = 8$	$v_3 = 15$	0	0	0	0	10	10	12	12	15	15	22

$V[i, j] = V[3, 10] = 22$

$V[i-1, j] = V[2, 10] = 22$

Here, $V[i, j] = V[i-1, j]$, so don't select i^{th} item. Check for the previous item

So $i = i - 1 = 3 - 1 = 2$

Step 2 : $i = 2, j = 10$

	Item detail		0	1	2	3	4	5	6	7	8	9	10
i=0			0	0	0	0	0	0	0	0	0	0	0
i=1	$w_1 = 4$	$v_1 = 10$	0	0	0	0	10	10	10	10	10	10	10
i=2	$w_2 = 6$	$v_2 = 12$	0	0	0	0	10	10	12	12	12	12	22
i=3	$w_3 = 8$	$v_3 = 15$	0	0	0	0	10	10	12	12	15	15	22

$V[i, j] = V[2, 10] = 22$

$V[i-1, j] = V[1, 10] = 10$

$V[i, j] \neq V[i-1, j]$, so add item $Ii = I_2$ in solution set.

Reduce problem size j by w_i

$j = j - w_i = j - w_2 = 10 - 6 = 4$

$i = i - 1 = 2 - 1 = 1$

Solution Set $S = \{I_2\}$

Step 3 : $i = 1, j = 4$

	Item detail		0	1	2	3	4	5	6	7	8	9	10
i=0			0	0	0	0	0	0	0	0	0	0	0
i=1	$w_1 = 4$	$v_1 = 10$	0	0	0	0	10	10	10	10	10	10	10
i=2	$w_2 = 6$	$v_2 = 12$	0	0	0	0	10	10	12	12	12	12	22
i=3	$w_3 = 8$	$v_3 = 15$	0	0	0	0	10	10	12	12	15	15	22

$V[i, j] = V[1, 10] = 10$

$V[i-1, j] = V[0, 10] = 0$

$V[i, j] \neq V[i-1, j]$, so add item $Ii = I_1$ in solution set. Reduce problem size j by w_i

$j = j - w_i = j - w_1 = 4 - 4 = 0$

Solution Set

$S = \{I_1, I_2\}$

Problem size has reached to 0, hence the final solution is $S = \{I_1, I_2\}$

Earned profit = p1 + p2 = 10 + 12 = 22

Ex. 4.9.2

Find an optimal solution for following 0/1 Knapsack problem using dynamic programming : Number of objects n = 4, Knapsack Capacity M = 5, Weights $(W_1, W_2, W_3, W_4) = (2, 3, 4, 5)$ and profits $(P_1, P_2, P_3, P_4) = (3, 4, 5, 6)$.

Soln. :

Solution of knapsack problem is defined as,

$$V[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ V[i-1, j] & \text{if } j < wi \\ \max \{V[i-1, j], vi + V[i-1, j-wi]\} & \text{if } j \geq w \end{cases}$$

We have following stats about problem,

Table P. 4.9.2

Item	Weight (w_i)	Value (v_i)
I_1	2	3
I_2	3	4
I_3	4	5
I_4	5	6

– Boundary conditions would be V [0, i] = V[i, 0] = 0.

Initial configuration of table looks like.

			j →					
	Item Detail		0	1	2	3	4	5
i=0			0	0	0	0	0	0
i=1	$w_1 = 2$	$v_1 = 3$	0					
i=2	$w_2 = 3$	$v_2 = 4$	0					
i=3	$w_3 = 4$	$v_3 = 5$	0					
i=4	$w_4 = 5$	$v_4 = 6$	0					

Filling first column, j = 1

$V[1, 1] \Rightarrow i = 1,\ j = 1,\ w_i = w_1 = 2$

As, $j < w_i$, $V[i, j] = V[i-1, j]$

$\therefore V[1, 1] = V[0, 1] = 0$

$V[2, 1] \Rightarrow i = 2,\ j = 1,\ w_i = w_2 = 3$

As, $j < w_i$, $V[i, j] = V[i-1, j]$

$\therefore V[2, 1] = V[1, 1] = 0$

$V[3, 1] \Rightarrow i = 3,\ j = 1,\ w_i = w_3 = 4$

As, $j < w_i$, $V[i, j] = V[i-1, j]$

$\therefore V[3, 1] = V[2, 1] = 0$

$V[4, 1] \Rightarrow i = 4,\ j = 1,\ w_i = w_4 = 5$

As, $j < w_i$, $V[i, j] = V[i-1, j]$

$\therefore V[4, 1] = V[3, 1] = 0$

Filling first column, j = 2

$V[1, 2] \Rightarrow i = 1,\ j = 2, w_i = w_1 = 2, v_i = 3$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j], v_i + V[i-1, j-w_i]\}$

$= \max \{V[0, 2], 3 + V[0, 0]\}$

$V[1, 2] = \max (0, 3) = 3$

$V[2, 2] \Rightarrow i = 2,\ j = 2, w_i = w_2 = 3, v_i = 4$

As, $j < w_i$, $V[i, j] = V[i-1, j]$

$\therefore V[2, 2] = V[1, 2] = 3$

$V[3, 2] \Rightarrow i = 3,\ j = 2, w_i = w_3 = 4, v_i = 5$

As, $j < w_i$, $V[i, j] = V[i-1, j]$

$\therefore V[3, 2] = V[2, 2] = 3$

$V[4, 2] \Rightarrow i = 4,\ j = 2, w_i = w_4 = 5, v_i = 6$

As, $j < w_i$, $V[i, j] = V[i-1, j]$

$\therefore V[4, 2] = V[3, 2] = 3$

Filling first column, j = 3

$V[1, 3] \Rightarrow i = 1,\ j = 3, w_i = w_1 = 2, v_i = 3$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j], v_i + V[i-1, j-w_i]\}$

$= \max \{V[0, 3], 3 + V[0, 1]\}$

$V[1, 3] = \max (0, 3) = 3$

$V[2, 3] \Rightarrow i = 2,\ j = 3, w_i = w_2 = 3, v_i = 4$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j], v_i + V[i-1, j-w_i]\}$

$= \max \{V[1, 3], 4 + V[1, 0]\}$

$V[2, 3] = \max (3, 4) = 4$

$V[3, 3] \Rightarrow i = 3,\ j = 3, w_i = w_3 = 4, v_i = 5$

As, $j < w_i$, $V[i, j] = V[i-1, j]$

$\therefore V[3, 3] = V[2, 3] = 4$

$V[4, 3] \Rightarrow i = 4,\ j = 3, w_i = w_4 = 5, v_i = 6$

As, $j < w_i$, $V[i, j] = V[i-1, j]$

$\therefore V[4, 3] = V[3, 3] = 4$

Filling first column, j = 4

$V[1, 4] \Rightarrow i = 1,\ j = 4,\ w_i = w_1 = 2, v_i = 3$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j], v_i + V[i-1, j-w_i]\}$

$= \max \{V[0, 4], 3 + V[0, 2]\}$

$V[1, 4] = \max (0, 3) = 3$

$V[2, 4] \Rightarrow i = 2,\ j = 4,\ w_i = w_2 = 3, v_i = 4$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j], v_i + V[i-1, j-w_i]\}$

$= \max \{V[1, 4], 4 + V[1, 1]\}$

$V[2, 4] = \max (3, 4 + 0) = 4$

$V[3, 4] \Rightarrow i = 3,\ j = 4,\ w_i = w_3 = 4, v_i = 5$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j], v_i + V[i-1, j-w_i]\}$

$= \max \{V[2, 4], 5 + V[2, 0]\}$

$V[3, 4] = \max (4, 5 + 0) = 5$

$V[4, 4] \Rightarrow i = 4,\ j = 4, w_i = w_4 = 5, v_i = 6$

As, $j < w_i$, $V[i, j] = V[i-1, j]$

$\therefore V[4, 4] = V[3, 4] = 5$

Filling first column, j = 5

$V[1, 5] \Rightarrow i = 1,\ j = 5,\ w_i = w_1 = 2, v_i = 3$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j], v_i + V[i-1, j-w_i]\}$

$= \max \{V[0, 5], 3 + V[0, 3]\}$

$V[1, 5] = \max (0, 3) = 3$

$V[2, 5] \Rightarrow i = 2,\ j = 5,\ w_i = w_2 = 3, v_i = 4$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j], v_i + V[i-1, j-w_i]\}$

$= \max \{V[1, 5], 4 + V[1, 2]\}$

$V[2, 5] = \max (3, 4 + 3) = 7$

$V[3, 5] \Rightarrow i = 3,\ j = 5,\ w_i = w_3 = 4, v_i = 5$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j], v_i + V[i-1, j-w_i]\}$

$= \max \{V[2, 5], 5 + V[2, 1]\}$

$V[3, 5] = \max (7, 5 + 0) = 7$

$V[4, 5] \Rightarrow i = 4,\ j = 5,\ w_i = w_4 = 5, v_i = 6$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j], v_i + V[i-1, j-w_i]\}$

$= \max \{V[3, 5], 6 + V[3, 0]\}$

$V[4, 5] = \max (7, 6 + 0) = 7$

Final table would be,

	Item Detail		j →					
			0	1	2	3	4	5
i=0			0	0	0	0	0	0
i=1	$w_1 = 2$	$v_1 = 3$	0	0	3	3	3	3
i=2	$w_2 = 3$	$v_2 = 4$	0	0	3	4	4	7
i=3	$w_3 = 4$	$v_3 = 5$	0	0	3	4	5	7
i=4	$w_4 = 5$	$v_4 = 6$	0	0	3	4	5	7

Find selected items for M = 5

Step 1 : Initially, i = n = 4, j = M = 5

	Item Detail		j → 0	1	2	3	4	5
i=0			0	0	0	0	0	0
i=1	$w_1 = 2$	$v_1 = 3$	0	0	3	3	3	3
i=2	$w_2 = 3$	$v_2 = 4$	0	0	3	4	4	7
i=3	$w_3 = 4$	$v_3 = 5$	0	0	3	4	5	7
i=4	$w_4 = 5$	$v_4 = 6$	0	0	3	4	5	7

V[i, j] = V[4, 5] = 7

V[i - 1, j] = V[3, 5] = 7

V[i, j] = V[i – 1, j], so don't select i^{th} item and check for the previous item.

so i = i – 1 = 4 – 1 = 3

Solution Set S = { }

Step 2 : i = 3, j = 5

	Item Detail		j → 0	1	2	3	4	5
i=0			0	0	0	0	0	0
i=1	$w_1 = 2$	$v_1 = 3$	0	0	3	3	3	3
i=2	$w_2 = 3$	$v_2 = 4$	0	0	3	4	4	7
i=3	$w_3 = 4$	$v_3 = 5$	0	0	3	4	5	7
i=4	$w_4 = 5$	$v_4 = 6$	0	0	3	4	5	7

V[i, j] = V[3, 5] = 7

V[i - 1, j] = V[2, 5] = 7

V[i, j] = V[i – 1, j], so don't select i^{th} item and check for the previous item.

so i = i – 1 = 3 – 1 = 2

Solution Set S = { }

Step 3 : i = 2, j = 5

	Item Detail		j → 0	1	2	3	4	5
i=0			0	0	0	0	0	0
i=1	$w_1 = 2$	$v_1 = 3$	0	0	3	3	3	3
i=2	$w_2 = 3$	$v_2 = 4$	0	0	3	4	4	7
i=3	$w_3 = 4$	$v_3 = 5$	0	0	3	4	5	7
i=4	$w_4 = 5$	$v_4 = 6$	0	0	3	4	5	7

V[i, j] = V[2, 5] = 7

V[i - 1, j] = V[1, 5] = 3

[i, j] ≠ V[i – 1, j], so add item Ii = I_2 in solution set.

Reduce problem size j by w_i

j = j – w_i = j – w_2 = 5 – 3 = 2

i = i – 1 = 2 – 1 = 1

Solution Set S = {I_2}

Step 4 : i = 1, j = 2

	Item Detail		j → 0	1	2	3	4	5
i=0			0	0	0	0	0	0
i=1	$w_1 = 2$	$v_1 = 3$	0	0	3	3	3	3
i=2	$w_2 = 3$	$v_2 = 4$	0	0	3	4	4	7
i=3	$w_3 = 4$	$v_3 = 5$	0	0	3	4	5	7
i=4	$w_4 = 5$	$v_4 = 6$	0	0	3	4	5	7

V[1, j] = V[1, 2] = 3

V[i - 1, j] = V[0, 2] = 0

V[i, j] ≠ V[i – 1, j], so add item Ii = I_1 in solution set.

Reduce problem size j by w_i

j = j – w_i = j – w_1 = 2 – 2 = 0

Solution Set

S = {I_1, I_2}

Problem size has reached to 0, so final solution is S = {I_1, I_2}

Earned profit = $P_1 + P_2$ = 7

Ex. 4.9.3

Consider the knapsack problem : n = 3, W_1, W_2, W_3) = (2, 3, 4) (P_1, P_2, P_3) = (1, 2, 5) and m = 6. Solve the problem using dynamic programming approach.

Soln. :

Solution of knapsack problem is defined as,

$$V[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ V[i-1, j] & \text{if } j < wi \\ \max\{V[i-1, j], vi + V[i-1, j-wi]\} & \text{if } j \geq w \end{cases}$$

We have following stats about problem,

Table P. 4.9.3

Item	Weight (w_i)	Value (v_i)
I_1	2	1
I_2	3	2
I_3	4	5

– Boundary conditions would be V [0, i] = V[i, 0] = 0.

Initial configuration of table looks like.

	Item Detail		j → 0	1	2	3	4	5	6
i=0			0	0	0	0	0	0	0
i=1	$w_1 = 2$	$v_1 = 1$	0						
i=2	$w_1 = 3$	$v_1 = 2$	0						
i=3	$w_1 = 4$	$v_1 = 5$	0						

Filling first column, j = 1

V [1, 1] ⇒ i = 1, j = 1, $w_i = w_1 = 2$

As, j < w_i, V [i, j] = V [i – 1, j]

∴ V [1, 1] = V [0, 1] = 0

$V[2, 1] \Rightarrow i = 2,\ j = 1,\ w_i = w_2 = 3$

As, $j < w_i$, $V[i, j] = V[i-1, j]$

$\therefore V[2, 1] = V[1, 1] = 0$

$V[3, 1] \Rightarrow i = 3,\ j = 1,\ w_i = w_3 = 4$

As, $j < w_i$, $V[i, j] = V[i-1, j]$

$\therefore V[3, 1] = V[2, 1] = 0$

Filling first column, j = 2

$V[1, 2] \Rightarrow i = 1,\ j = 2, w_i = w_1 = 2, v_i = 1$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j], v_i + V[i-1, j-w_i]\}$

$= \max \{V[0, 2], 1 + V[0, 0]\}$

$V[1, 2] = \max (0, 1) = 1$

$V[2, 2] \Rightarrow i = 2,\ j = 2, w_i = w_2 = 3, v_i = 2$

As, $j < w_i$, $V[i, j] = V[i-1, j]$

$\therefore V[2, 2] = V[1, 2] = 1$

$V[3, 2] \Rightarrow i = 3,\ j = 2, w_i = w_3 = 4, v_i = 5$

As, $j < w_i$, $V[i, j] = V[i-1, j]$

$\therefore V[3, 2] = V[2, 2] = 1$

Filling first column, j = 3

$V[1, 3] \Rightarrow i = 1,\ j = 3, w_i = w_1 = 2, v_i = 1$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j], v_i + V[i-1, j-w_i]\}$

$= \max \{V[0, 3], 1 + V[0, 1]\}$

$V[1, 3] = \max (0, 1) = 1$

$V[2, 3] \Rightarrow i = 2,\ j = 3, w_i = w_2 = 3, v_i = 2$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j], v_i + V[i-1, j-w_i]\}$

$= \max \{V[1, 3], 2 + V[1, 0]\}$

$V[2, 3] = \max (1, 2) = 2$

$V[3, 3] \Rightarrow i = 3,\ j = 3, w_i = w_3 = 4, v_i = 5$

As, $j < w_i$, $V[i, j] = V[i-1, j]$

$\therefore V[3, 3] = V[2, 3] = 2$

Filling first column, j = 4

$V[1, 4] \Rightarrow i = 1,\ j = 4,\ w_i = w_1 = 2, v_i = 1$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j], v_i + V[i-1, j-w_i]\}$

$= \max \{V[0, 4], 1 + V[0, 2]\}$

$V[1, 4] = \max (0, 1) = 1$

$V[2, 4] \Rightarrow i = 2,\ j = 4,\ w_i = w_2 = 3, v_i = 2$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j], v_i + V[i-1, j-w_i]\}$

$= \max \{V[1, 4], 2 + V[1, 1]\}$

$V[2, 4] = \max (1, 2 + 0) = 2$

$V[3, 4] \Rightarrow i = 3,\ j = 4,\ w_i = w_3 = 4, v_i = 5$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j], v_i + V[i-1, j-w_i]\}$

$= \max \{V[2, 4], 5 + V[2, 0]\}$

$V[3, 4] = \max (2, 5 + 0) = 5$

Filling first column, j = 5

$V[1, 5] \Rightarrow i = 1,\ j = 5,\ w_i = w_1 = 2, v_i = 1$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j], v_i + V[i-1, j-w_i]\}$

$= \max \{V[0, 5], 1 + V[0, 3]\}$

$V[1, 5] = \max (0, 1) = 1$

$V[2, 5] \Rightarrow i = 2,\ j = 5,\ w_i = w_2 = 3, v_i = 2$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j], v_i + V[i-1, j-w_i]\}$

$= \max \{V[1, 5], 2 + V[1, 2]\}$

$V[2, 5] = \max (1, 2 + 1) = 3$

$V[3, 5] \Rightarrow i = 3,\ j = 5,\ w_i = w_3 = 4, v_i = 5$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j], v_i + V[i-1, j-w_i]\}$

$= \max \{V[2, 5], 5 + V[2, 1]\}$

$V[3, 5] = \max (3, 5 + 0) = 5$

Filling first column, j = 6

$V[1, 6] \Rightarrow i = 1,\ j = 6,\ w_i = w_1 = 2, v_i = 1$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j], v_i + V[i-1, j-w_i]\}$

$= \max \{V[0, 6], 1 + V[0, 4]\}$

$V[1, 6] = \max (0, 1) = 1$

$V[2, 6] \Rightarrow i = 2,\ j = 6,\ w_i = w_2 = 3, v_i = 2$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j], v_i + V[i-1, j-w_i]\}$

$= \max \{V[1, 6], 2 + V[1, 3]\}$

$V[2, 6] = \max (1, 2 + 1) = 3$

$V[3, 6] \Rightarrow i = 3,\ j = 6,\ w_i = w_3 = 4, v_i = 5$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j], v_i + V[i-1, j-w_i]\}$

$= \max \{V[2, 6], 5 + V[2, 2]\}$

$V[3, 6] = \max (3, 5 + 1) = 6$

j →

	Item Detail		0	1	2	3	4	5	6
i=0			0	0	0	0	0	0	0
i=1	$w_1 = 2$	$v_1 = 1$	0	0	1	1	1	1	1
i=2	$w_1 = 3$	$v_1 = 2$	0	0	1	2	2	3	3
i=3	$w_1 = 4$	$v_1 = 5$	0	0	1	2	5	5	6

Find selected items for M = 6

Step 1 : Initially, i = n = 3, j = M = 6

j →

	Item Detail		0	1	2	3	4	5	6
i=0			0	0	0	0	0	0	0
i=1	$w_1 = 2$	$v_1 = 1$	0	0	1	1	1	1	1
i=2	$w_1 = 3$	$v_1 = 2$	0	0	1	2	2	3	3
i=3	$w_1 = 4$	$v_1 = 5$	0	0	1	2	5	5	6

$V[i, j] = V[3, 6] = 6$

$V[i-1, j] = V[2, 6] = 3$

$V[i, j] \neq V[i-1, j]$, so add item $Ii = I_3$ in solution set.

Reduce problem size j by w_i

$j = j - w_i = j - w_3 = 6 - 4 = 2$

$i = i - 1 = 3 - 1 = 2$

Solution Set $S = \{I_3\}$

Step 2 : i = 2, j = 2

j ⟶

	Item Detail		0	1	2	3	4	5	6
i=0			0	0	0	0	0	0	0
i=1	$w_1 = 2$	$v_1 = 1$	0	0	1	1	1	1	1
i=2	$w_1 = 3$	$v_1 = 2$	0	0	1	2	2	3	3
i=3	$w_1 = 4$	$v_1 = 5$	0	0	1	2	5	5	6

$V[i, j] = V[2, 2] = 1$

$V[i - 1, j] = V[1, 2] = 1$

$V[i, j] = V[i - 1, j]$, so don't select i^{th} item and check for the previous item.

so $i = i - 1 = 2 - 1 = 1$

Solution Set $S = \{I_3\}$

Step 3 : i = 1, j = 2

j ⟶

	Item Detail		0	1	2	3	4	5	6
i=0			0	0	0	0	0	0	0
i=1	$w_1 = 2$	$v_1 = 1$	0	0	1	1	1	1	1
i=2	$w_1 = 3$	$v_1 = 2$	0	0	1	2	2	3	3
i=3	$w_1 = 4$	$v_1 = 5$	0	0	1	2	5	5	6

$V[i, j] = V[1, 2] = 1$

$V[i - 1, j] = V[0, 2] = 0$

$V[i, j] \neq V[i - 1, j]$, so add item $Ii = I_1$ in solution set.

Reduce problem size j by w_i

$j = j - w_i = j - w_1 = 2 - 2 = 0$

Solution Set

$S = \{I_1, I_3\}$

Problem size has reached to 0, so final solution is $S = \{I_1, I_3\}$

Earned profit = $p_1 + p_3 = 1 + 5 = 6$

Ex. 4.9.4

Solve following knapsack problem using dynamic programming algorithm with given capacity W = 5, Weight and Value are as follows : (2,12), (1,10), (3,20), (2,15).

Soln. :

Solution of knapsack problem is defined as,

$$V[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ V[i-1, j] & \text{if } j < wi \\ \max\{V[i-1, j], vi + V[i-1, j-wi]\} & \text{if } j \geq w \end{cases}$$

We have following stats about problem,

Table P. 4.9.4

Item	Weight (w_i)	Value (v_i)
I_1	2	12
I_2	1	10
I_3	3	20
I_4	2	15

Boundary conditions would be V [0, 0] = 0. Initial configuration of table looks like,

j ⟶

i ↓

Item detail		0	1	2	3	4	5
		0	0	0	0	0	0
$w_1 = 2$	$v_1 = 12$	0					
$w_2 = 1$	$v_2 = 10$	0					
$w_3 = 3$	$v_3 = 20$	0					
$w_4 = 2$	$v_4 = 15$	0					

Filling first column, j = 1

$V[1, 1] \Rightarrow i = 1,\ j = 1,\ w_i = w_1 = 2$

As, $j < w_i$, $V[i, j] = V[i - 1, j]$

$\therefore V[1, 1] = V[0, 1] = 0$

$V[2, 1] \Rightarrow i = 2,\ j = 1,\ w_i = w_2 = 1$

As, $j \geq w_i$, $V[i, j] = \max\{V[i - 1, j], v_i + V[i - 1, j - w_i]\}$

$= \max\{V[1, 1], 10 + V[1, 0]\}$

$= \max\{0, 10 + 0\} = 10$

$V[3, 1] \Rightarrow i = 3,\ j = 1,\ w_i = w_3 = 3$

As, $j < w_i$, $V[i, j] = V[i - 1, j]$

$\therefore V[3, 1] = V[2, 1] = 10$

$V[4, 1] \Rightarrow i = 4,\ j = 1,\ w_i = w_4 = 2$

As, $j < w_i$, $V[i, j] = V[i - 1, j]$

$\therefore V[4, 1] = V[3, 1] = 10$

Filling second column, j = 2

$V[1, 2] \Rightarrow i = 1,\ j = 2,\ w_i = w_1 = 2,\ v_i = 12$

As, $j \geq w_i$, $V[i, j] = \max\{V[i - 1, j], v_i + V[i - 1, j - w_i]\}$

$= \max\{V[0, 2], 12 + V[0, 0]\}$

$= \max(0, 12 + 0) = 12$

$V[2, 2] \Rightarrow i = 2,\ j = 2,\ w_i = w_2 = 1,\ v_i = 10$

As, $j \geq w_i$, $V[i, j] = \max\{V[i - 1, j], v_i + V[i - 1, j - w_i]\}$

$= \max\{V[1, 2], 10 + V[1, 1]\}$

$= \max(12, 10 + 0) = 12$

$V[3, 2] \Rightarrow i = 3,\ j = 2, w_i = w_3 = 3$

As, $j < w_i$, $V[i, j] = V[i - 1, j]$

$\therefore$ V [3, 2] = V [2, 2] = 12

V [4, 2] $\Rightarrow$ i = 4, j = 2, $w_i = w_4 = 2$, $v_i = 15$

As, $j \geq w_i$, V [i, j] = max {V [i − 1, j], v_i + V [i − 1, j − w_i] }

= max {V [3, 2], 15 + V [3, 0]}

= max (12, 15 + 0) = 15

Filling third column, j = 3

V [1, 3] $\Rightarrow$ i = 1, j = 3, $w_i = w_1 = 2$, $v_i = v_1 = 12$

As, $j \geq w_i$, V [i, j] = max {V [i − 1, j], v_i + V [i − 1, j − w_i] }

= max {V [0, 3], 12 + V [0, 1]}

= max (0, 12 + 0) = 12

V [2, 3] $\Rightarrow$ i = 2, j = 3, $w_i = w_2 = 1$, $v_i = v_2 = 10$

As, $j \geq w_i$, V [i, j] = max {V [i − 1, j], v_i + V [i − 1, j − w_i] }

= max {V [1, 3], 10 + V [1, 2]}

= max (12, 10 + 12) = 22

V [3, 3] $\Rightarrow$ i = 3, j = 3, $w_i = w_3 = 3$, $v_i = v_3 = 20$

As, $j \geq w_i$, V [i, j] = max {V [i − 1, j], v_i + V [i − 1, j − w_i] }

= max {V [2, 3], 20 + V [2, 0]}

= max (22, 20 + 0) = 22

V [4, 3] $\Rightarrow$ i = 4, j = 3, $w_i = w_4 = 2$, $v_i = v_4 = 15$

As, $j \geq w_i$, V [i, j] = max {V [i − 1, j], v_i + V [i − 1, j − w_i] }

= max {V [3, 3], 15 + V [3, 1]}

= max (22, 15 + 10) = 25

Filling fourth column, j = 4

V [1, 4] $\Rightarrow$ i = 1, j = 4, $w_i = w_1 = 2$, $v_i = v_1 = 12$

As, $j \geq w_i$, V [i, j] = max {V [i − 1, j], v_i + V [i − 1, j − w_i] }

= max {V [0, 4], 12 + V [0, 2]}

= max (0, 12 + 0) = 12

V [2, 4] $\Rightarrow$ i = 2, j = 4, $w_i = w_2 = 1$, $v_i = v_2 = 10$

As, $j \geq w_i$, V [i, j] = max {V [i − 1, j], v_i + V [i − 1, j − w_i] }

= max {V [1, 4], 10 + V [1, 3]}

= max (12, 10 + 12) = 22

V [3, 4] $\Rightarrow$ i = 3, j = 4, $w_i = w_3 = 3$, $v_i = v_3 = 20$

As, $j \geq w_i$, V [i, j] = max {V [i − 1, j], v_i + V [i − 1, j − w_i] }

= max {V [2, 4], 20 + V [2, 1]}

= max (22, 20 + 10) = 30

V [4, 4] $\Rightarrow$ i = 4, j = 4, $w_i = w_4 = 2$, $v_i = v_4 = 15$

As, $j \geq w_i$, V [i, j] = max {V [i − 1, j], v_i + V [i − 1, j − w_i]}

= max {V [3, 4], 15 + V [3, 2]}

= max (30, 15 + 12) = 30

Filling fifth column, j = 5

V [1, 5] $\Rightarrow$ i = 1, j = 5, $w_i = w_1 = 2$, $v_i = v_1 = 12$

As, $j \geq w_i$, V [i, j] = max {V [i − 1, j], v_i + V [i − 1, j − w_i] }

= max {V [0, 5], 12 + V [0, 3]}

= max (0, 12 + 0) = 12

V [2, 5] $\Rightarrow$ i = 2, j = 5, $w_i = w_2 = 1$, $v_i = v_2 = 10$

As, $j \geq w_i$, V [i, j] = max {V [i − 1, j], v_i + V [i − 1, j − w_i] }

= max {V [1, 5], 10 + V [1, 3]}

= max (12, 10 + 12) = 22

V [3, 5] $\Rightarrow$ i = 3, j = 5, $w_i = w_3 = 3$, $v_i = v_3 = 20$

As, $j \geq w_i$, V [i, j] = max {V [i − 1, j], v_i + V [i − 1, j − w_i] }

= max {V [2, 5], 20 + V [2, 2]}

= max (22, 20 + 12) = 32

V [4, 5] $\Rightarrow$ i = 4, j = 5, $w_i = w_4 = 2$, $v_i = v_4 = 15$

As, $j \geq w_i$, V [i, j] = max {V [i − 1, j], v_i + V [i − 1, j − w_i] }

= max {V [3, 5], 15 + V [3, 3]}

= max (32, 15 + 22) = 37

So finally, table looks like,

j →

i ↓	Item detail		0	1	2	3	4	5
			0	0	0	0	0	0
$w_1 = 2$	$v_1 = 12$		0	0	12	12	12	12
$w_2 = 1$	$v_2 = 10$		0	10	12	22	22	22
$w_3 = 3$	$v_3 = 20$		0	10	12	22	30	32
$w_4 = 2$	$v_4 = 15$		0	10	15	25	30	37

The maximum profit we can earn is 37 units. This table explicitly does not say anything about which items should be selected to earn this profit. Let us trace the table to find the items to be added in a knapsack to earn the profit of 37 units.

Tracing solution

For the last cell in the table, i and j would be 4 and 5 respectively.

V[n, M] = V[i, j] = V[4, 5] = 37

Step 1 : Initially, i = n = 4, j = M = 5

j →

	Item detail		0	1	2	3	4	5
i = 0			0	0	0	0	0	0
i = 1	$w_1 = 2$	$v_1 = 12$	0	0	12	12	12	12
i = 2	$w_2 = 1$	$v_2 = 10$	0	10	12	22	22	22
i = 3	$w_3 = 3$	$v_3 = 20$	0	10	12	22	30	32
i = 4	$w_4 = 2$	$v_4 = 15$	0	10	15	25	30	37

V[i, j] = V[4, 5] = 37

V[i - 1, j] = V[3, 5] = 32

Here, V[i, j] ≠ V[i − 1, j], so add Ii in solution set and reduce problem size j by w_i and reduce i because the same item cannot be added again.

Solution set

S = {I_4}, j = j − w_i = 5 − 2 = 3.

And i = i − 1 = 4 − 1 = 3

Now we have,

i = 3 and j = 3.

Step 2 : i = 3, j = 3

j ⟶

	Item detail		0	1	2	3	4	5
i = 0			0	0	0	0	0	0
i = 1	$w_1 = 2$	$v_1 = 12$	0	0	12	12	12	12
i = 2	$w_2 = 1$	$v_2 = 10$	0	10	12	22	22	22
i = 3	$w_3 = 3$	$v_3 = 20$	0	10	12	22	30	32
i = 4	$w_4 = 2$	$v_4 = 15$	0	10	15	25	30	37

V[i, j] = V[3, 3] = 22

V[i - 1, j] = V[2, 3] = 22

Here, V[i, j] = V[i – 1, j], reject item $I_i = I_3$

i = i – 1 = 3 – 1 = 2

Step 3 : i = 2, j = 3 :

j ⟶

	Item detail		0	1	2	3	4	5
i = 0			0	0	0	0	0	0
i = 1	$w_1 = 2$	$v_1 = 12$	0	0	12	12	12	12
i = 2	$w_2 = 1$	$v_2 = 10$	0	10	12	22	22	22
i = 3	$w_3 = 3$	$v_3 = 20$	0	10	12	22	30	32
i = 4	$w_4 = 2$	$v_4 = 15$	0	10	15	25	30	37

V[i, j] = V[2, 3] = 22

V[i - 1, j] = V[1, 3] = 12

Here, V[i, j] ≠ V[i – 1, j], so add I_2 in solution set and reduce problem size j by w_i and reduce i because the same item cannot be added again.

Solution set S = {I_2, I_4},

j = j – w_i = 3 – 1 = 2

And i = i – 1 = 2 – 1 = 1

Step 4 : i = 1, j = 2

j ⟶

	Item detail		0	1	2	3	4	5
i = 0			0	0	0	0	0	0
i = 1	$w_1 = 2$	$v_1 = 12$	0	0	12	12	12	12
i = 2	$w_2 = 1$	$v_2 = 10$	0	10	12	22	22	22
i = 3	$w_3 = 3$	$v_3 = 20$	0	10	12	22	30	32
i = 4	$w_4 = 2$	$v_4 = 15$	0	10	15	25	30	37

V[i, j] = V[1, 2] = 12

V[i - 1, j] = V[0, 2] = 0

Here, V[i, j] ≠ V[i – 1, j], so add I_1 in the solution set and reduce problem size j by w_i and reduce i because the same item cannot be added again.

Solution set

S = {I_1, I_2, I_4}, j = j – w_i = 2 – 2 = 0.

And i = i – 1 = 1 – 1 = 0

Problem size has reached to zero. Selected items are S = {I_1, I_2, I_4}.

Earned profit = $p_1 + p_2 + p_4$ = 12 + 10 + 15 = 37

Ex. 4.9.5

Solve the following instance using Dynamic programming, write the algorithm also. Knapsack Capacity = 10, P = < 1, 6, 18, 22, 28 > and w = < 1, 2, 5, 6, 7 >.

Soln. : Solution of knapsack problem is defined as,

$$V[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ V[i-1, j] & \text{if } j < wi \\ \max\{V[i-1, j], vi + V[i-1, j-wi]\} & \text{if } j \geq w \end{cases}$$

We have following stats about problem,

Table P. 4.9.5

Item	Weight (w_i)	Value (v_i)
I_1	1	1
I_2	2	6
I_3	5	18
I_4	6	22
I_5	7	28

Boundary conditions would be V [1, 0] = v_i and V [0, 1] = 0. Initial configuration of table looks like.

	Item detail		0	1	2	3	4	5	6	7	8	9	10	11
	$w_1 = 1$	$v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
	$w_2 = 2$	$v_2 = 6$	0											
i ⟶	$w_3 = 5$	$v_3 = 18$	0											
	$w_4 = 6$	$v_4 = 22$	0											
	$w_5 = 7$	$v_5 = 28$	0											

Filling first column, j = 1

V [2, 1] ⇒ i = 2, j = 1, $w_i = w_2 = 2$

As, j < w_i, V [i, j] = V [i – 1, j]

∴ V [2, 1] = V [1, 1] = 1

Similarly, we will get,

V [3, 1] = 1

V [4, 1] = 1

V [5, 1] = 1

Filling second column, j = 2

V [2, 2] ⇒ i = 2, j = 2, $w_i = w_2 = 2$, $v_i = 6$

As, $w_i \geq j$ (j ≤ w_i), V[i, j] = max{V [i – 1, j], v_i + V [i – 1, j – w_i]}

= max {V [1, 2], 6 + V [1, 0]}

= max (1, 6 + 0) = 6

Similarly, we will get,

$V[3, 2] = 6$

$V[4, 2] = 6$

$V[5, 2] = 6$

Filling third column, j = 3

$V[2, 3] \Rightarrow i = 2,\ j = 3,\ w_i = w_2 = 2,\ v_i = v_2 = 6$

As, $w_i \geq j$, $V[i, j] = \max \{V[i-1, j], v_i + V[i-1, j-w_i]\}$

$= \max \{V[1, 3], 6 + V[1, 1]\}$

$= \max (1, 6 + 1) = 7$

Similarly, we will get,

$V[3, 3] = 7$

$V[4, 3] = 7$

$V[5, 3] = 7$

Filling fourth column, j = 4

$V[2, 4] \Rightarrow i = 2,\ j = 4,\ w_i = w_2 = 2,\ v_i = v_2 = 6$

As, $w_i \geq j$, $V[i, j] = \max \{V[i-1, j], v_i + V[i-1, j-w_i]\}$

$= \max \{V[1, 4], 6 + V[1, 2]\}$

$= \max (1, 6 + 1) = 7$

Similarly, we will get,

$V[3, 4] = 7$

$V[4, 4] = 7$

$V[5, 4] = 7$

Filling fifth column, j = 5

$V[2, 5] \Rightarrow i = 2,\ j = 5,\ w_i = w_2 = 2,\ v_i = v_2 = 6$

As, $w_i \geq j$, $V[i, j] = \max \{V[i-1, j], v_i + V[i-1, j-w_i]\}$

$= \max \{V[1, 5], 6 + V[1, 3]\}$

$= \max (1, 6 + 1) = 7$

$V[3, 5] \Rightarrow i = 3,\ j = 5,\ w_i = w_3 = 5,\ v_i = v_3 = 18$

As, $w_i \geq j$, $V[i, j] = \max \{V[i-1, j], v_i + V[i-1, j-w_i]\}$

$= \max \{V[2, 5], 18 + V[2, 0]\}$

$= \max (7, 18 + 0) = 18$

Similarly, we will get,

$V[4, 5] = 18$

$V[5, 5] = 18$

Filling sixth column, j = 6

$V[2, 6] = 7$

$V[3, 6] \Rightarrow i = 3,\ j = 6,\ w_i = w_3 = 5,\ v_i = v_3 = 18$

As, $j \geq w_i$

$V[i, j] = \max \{V[i-1, j], v_i + V[i-1, j-w_i]\}$

$= \max \{V[2, 6], 18 + V[2, 1]\}$

$= \max (7, 18 + 1) = 19$

$V[4, 6] \Rightarrow i = 4,\ j = 6,\ w_i = w_4 = 6,\ v_i = v_4 = 22$

As, $j \geq w_i$, $V[i, j] = \max \{V[i-1, j], v_i + V[i-1, j-w_i]\}$

$= \max \{V[3, 6], 22 + V[3, 0]\}$

$= \max (19, 22 + 0) = 22$

Similarly, we will get, $V[5, 6] = 22$

If we continue in a similar way, final table would look like,

j →

i ↓	Item detail		0	1	2	3	4	5	6	7	8	9	10	11
	$w_1 = 1$	$v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
	$w_2 = 2$	$v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
	$w_3 = 5$	$v_3 = 18$	0	1	6	7	7	18	19	24	25	25	25	25
	$w_4 = 6$	$v_4 = 22$	0	1	6	7	7	18	22	24	28	29	29	40
	$w_5 = 7$	$v_5 = 28$	0	1	6	7	7	18	22	28	29	34	35	40

Trace for W = 11

Step 1 : Initially, i = 5, j = 11

j →

i = 0	Item detail		0	1	2	3	4	5	6	7	8	9	10	11
i = 1	$w_1 = 1$	$v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
i = 2	$w_2 = 2$	$v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
i = 3	$w_3 = 5$	$v_3 = 18$	0	1	6	7	7	18	19	24	25	25	25	25
i = 4	$w_4 = 6$	$v_4 = 22$	0	1	6	7	7	18	22	24	28	29	29	40
i = 5	$w_5 = 7$	$v_5 = 28$	0	1	6	7	7	18	22	28	29	34	35	40

$V[i, j] = V[i-1, j]$, so don't select i^{th} item and check for the previous item.

So, $i = i - 1 = 5 - 1 = 4$

Step 2 : $i = 4, j = 11$

j →

i = 0	Item detail		0	1	2	3	4	5	6	7	8	9	10	11
i = 1	$w_1 = 1$	$v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
i = 2	$w_2 = 2$	$v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
i = 3	$w_3 = 5$	$v_3 = 18$	0	1	6	7	7	18	19	24	25	25	25	25
i = 4	$w_4 = 6$	$v_4 = 22$	0	1	6	7	7	18	22	24	28	29	29	40
i = 5	$w_5 = 7$	$v_5 = 28$	0	1	6	7	7	18	22	28	29	34	35	40

$V[i, j] \neq V[i-1, j]$, so add item $I_i = I_4$ in solution set.

Reduce problem size j by $w_i = w_4$

$j = j - wi = j - w_4 = 11 - 6 = 5$

$i = i - 1 = 4 - 1 = 3$

Solution Set

$S = \{I_4\}$

Step 3 : i = 3, j = 5

i = 0	Item detail		0	1	2	3	4	5	6	7	8	9	10	11
i = 1	$w_1 = 1$	$v_1 = 1$	0	1	1	1	1	1	1	1	1	1	1	1
i = 2	$w_2 = 2$	$v_2 = 6$	0	1	6	7	7	7	7	7	7	7	7	7
i = 3	$w_3 = 5$	$v_3 = 18$	0	1	6	7	7	18	19	24	25	25	25	25
i = 4	$w_4 = 6$	$v_4 = 22$	0	1	6	7	7	18	22	24	28	29	29	40
i = 5	$w_5 = 7$	$v_5 = 28$	0	1	6	7	7	18	22	28	29	34	35	40

j →

$V[i, j] \neq V[i-1, j]$, so add item $I_i = I_3$ in solution set.

Reduce problem size j by $w_i = w_3$

$$j = j - w_i = j - w_3 = 5 - 5 = 0$$

Solution Set

$$S = \{I_3, I_4\}$$

Earned profit $= p_3 + p_4 = 18 + 22 = 40$

Syllabus Topic : Travelling Salesman Problem

4.10 Travelling Salesman Problem

→ (May 15)

Q. Write short on Travelling sales person problem. **MU - May 2015, 10 Marks**

Q. Define and solve Travelling Salesman Problem using dynamic programming. **(10 Marks)**

- Traveling Salesman Problem (TSP) is well studied and well-explored problem of computer science. Due to its application in diverse fields, TSP has been one of the most interesting problems for researchers and mathematicians.
- Traveling salesman problem is stated as, "Given a set of *n* cities and distance between each pair of cities, find the minimum length path such that it covers each city exactly once and terminates the tour at starting city."
- It is not difficult to show that this problem is NP-complete problem. There exists *n!* paths, a search of the optimal path becomes very slow when *n* is considerably large.
- Each edge (u, v) in TSP graph is assigned some non-negative weight, which represents the distance between city u and v. This problem can be solved by finding the Hamiltonian cycle of the graph.
- The distance between cities is best described by the weighted graph, where edge (u, v) indicates the path from city u to v and w(u, v) represents the distance between cities u and v.
- Let us formulate the solution of TSP using dynamic programming.

Step 1 : Let d[i, j] indicates the distance between cities i and j. Function C[x, V – { x }] is the cost of the path starting from city x. V is the set of cities/vertices in given graph. The aim of TSP is to minimize the cost function.

Step 2 : Assume that graph contains *n* vertices $V_1, V_2, ..., V_n$. TSP finds a path covering all vertices exactly once, and the same time it tries to minimize the overall travelling distance.

Step 3 : Mathematical formula to find minimum distance is stated below:

- $C(i, V) = \min \{ d[i, j] + C(j, V - \{ j \}) \}, j \in V$ and $i \notin V$.
- TSP problem possesses the principle of optimality, i.e. for $d[V_1, V_n]$ to be minimum, any intermediate path (V_i, V_j) must be minimum.
- From Fig. 4.10.1, d[i, j] = min(d[i, j], d[i, k] + d[k, j])
- Dynamic programming always selects the path which is minimum.

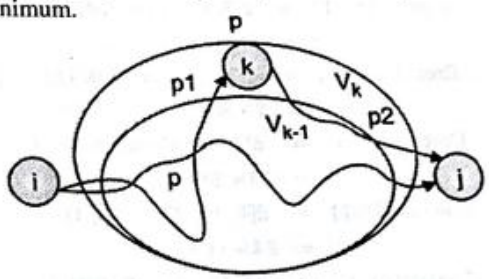


Fig. 4.10.1

☞ **Complexity analysis**

Dynamic programming creates $n.2^n$ subproblems for n cities. Each subproblem can be solved in linear time. Thus the time complexity of TSP using dynamic programming would be $O(n^2 2^n)$. It is much less than n! but still, it is an exponent. Space complexity is also exponential.

Let us explore the TSP problem with the help of an example.

Ex. 4.10.1

Solve the travelling salesman problem with the associated cost adjacency matrix using dynamic programming.

-	24	11	10	9
8	-	2	5	11
26	12	-	8	7
11	23	24	-	6
5	4	8	11	-

Soln. :

Let us start our tour from city 1.

Step 1 : Initially, we will find the distance between city 1 and city {2, 3, 4, 5} without visiting any intermediate city.

Cost(x, y, z) represents the distance from x to z and y as an intermediate city.

$$Cost(2, \phi, 1) = d[2, 1] = 24$$
$$Cost(3, \phi, 1) = d[3, 1] = 11$$
$$Cost(4, \phi, 1) = d[4, 1] = 10$$
$$Cost(5, \phi, 1) = d[5, 1] = 9$$

Step 2 :

In this step, we will find the minimum distance by visiting 1 city as intermediate city.

$$Cost\{2, \{3\}, 1\} = d[2, 3] + Cost(3, \phi, 1) = 2 + 11 = 13$$
$$Cost\{2, \{4\}, 1\} = d[2, 4] + Cost(4, \phi, 1) = 5 + 10 = 15$$
$$Cost\{2, \{5\}, 1\} = d[2, 5] + Cost(5, \phi, 1) = 11 + 9 = 20$$
$$Cost\{3, \{2\}, 1\} = d[3, 2] + Cost(2, \phi, 1) = 12 + 24 = 36$$
$$Cost\{3, \{4\}, 1\} = d[3, 4] + Cost(4, \phi, 1) = 8 + 10 = 18$$
$$Cost\{3, \{5\}, 1\} = d[3, 5] + Cost(5, \phi, 1) = 7 + 9 = 16$$
$$Cost\{4, \{2\}, 1\} = d[4, 2] + Cost(2, \phi, 1) = 23 + 24 = 47$$
$$Cost\{4, \{3\}, 1\} = d[4, 3] + Cost(3, \phi, 1) = 24 + 11 = 35$$
$$Cost\{4, \{5\}, 1\} = d[4, 5] + Cost(5, \phi, 1) = 6 + 9 = 15$$
$$Cost\{5, \{2\}, 1\} = d[5, 2] + Cost(2, \phi, 1) = 4 + 24 = 28$$
$$Cost\{5, \{3\}, 1\} = d[5, 3] + Cost(3, \phi, 1) = 8 + 11 = 19$$
$$Cost\{5, \{4\}, 1\} = d[5, 4] + Cost(4, \phi, 1) = 11 + 10 = 21$$

Step 3 : In this step, we will find the minimum distance by visiting 2 cities as an intermediate city.

$$Cost(2, \{3, 4\}, 1) = \min \{ d[2, 3] + Cost(3, \{4\}, 1), d[2, 4] + Cost(4, \{3\}, 1)]\} = \min \{ [2 + 18], [5 + 35] \} = \min\{20, 40\} = 20$$

$$Cost(2, \{4, 5\}, 1) = \min \{ d[2, 4] + Cost(4, \{5\}, 1), d[2, 5] + Cost(5, \{4\}, 1)]\} = \min \{ [5 + 15], [11 + 21] \} = \min\{20, 32\} = 20$$

$$Cost(2, \{3, 5\}, 1) = \min \{ d[2, 3] + Cost(3, \{4\}, 1), d[2, 4] + Cost(4, \{3\}, 1)]\} = \min \{ [2 + 18], [5 + 35] \} = \min\{20, 40\} = 20$$

$$Cost(3, \{2, 4\}, 1) = \min \{ d[3, 2] + Cost(2, \{4\}, 1), d[3, 4] + Cost(4, \{2\}, 1)]\} = \min \{ [12 + 15], [8 + 47] \} = \min\{27, 55\} = 27$$

$$Cost(3, \{4, 5\}, 1) = \min \{ d[3, 4] + Cost(4, \{5\}, 1), d[3, 5] + Cost(5, \{4\}, 1)]\} = \min \{ [8 + 15], [7 + 21] \} = \min\{23, 28\} = 23$$

$$Cost(3, \{2, 5\}, 1) = \min \{ d[3, 2] + Cost(2, \{5\}, 1), d[3, 5] + Cost(5, \{2\}, 1)]\} = \min \{ [12 + 20], [7 + 28] \} = \min\{32, 35\} = 32$$

$$Cost(4, \{2, 3\}, 1) = \min\{ d[4, 2] + Cost(2, \{3\}, 1), d[4, 3] + Cost(3, \{2\}, 1)]\} = \min \{ [23 + 13], [24 + 36] \} = \min\{36, 60\} = 36$$

$$Cost(4, \{3, 5\}, 1) = \min\{ d[4, 3] + Cost(3, \{5\}, 1), d[4, 5] + Cost(5, \{3\}, 1)]\} = \min \{ [24 + 16], [6 + 19] \} = \min\{40, 25\} = 25$$

$$Cost(4, \{2, 5\}, 1) = \min\{ d[4, 2] + Cost(2, \{5\}, 1), d[4, 5] + Cost(5, \{2\}, 1)]\} = \min \{ [23 + 20], [6 + 28] \} = \min\{43, 34\} = 34$$

$$Cost(5, \{2, 3\}, 1) = \min\{ d[5, 2] + Cost(2, \{3\}, 1), d[5, 3] + Cost(3, \{2\}, 1)]\} = \min \{ [4 + 13], [8 + 36] \} = \min\{17, 44\} = 17$$

$$Cost(5, \{3, 4\}, 1) = \min\{ d[5, 3] + Cost(3, \{4\}, 1), d[5, 4] + Cost(4, \{3\}, 1)]\} = \min \{ [8 + 18], [11 + 35] \} = \min\{26, 46\} = 26$$

$$Cost(5, \{2, 4\}, 1) = \min\{ d[5, 2] + Cost(2, \{4\}, 1), d[5, 4] + Cost(4, \{2\}, 1)]\} = \min \{ [4 + 15], [11 + 47] \} = \min\{19, 58\} = 19$$

Step 4 : In this step, we will find the minimum distance by visiting 3 cities as an intermediate city.

$$\begin{aligned}
Cost(2, \{3, 4, 5\}, 1) &= \min \{ d[2, 3] \\
&\quad + Cost(3, \{4, 5\}, 1), d[2, 4] \\
&\quad + Cost(4, \{3, 5\}, 1), d[2, 5] \\
&\quad + Cost(5, \{3, 4\}, 1)\} \\
&= \min \{ 2 + 23, 5 + 25, 11 + 36\} \\
&= \min\{25, 30, 47\} = 25 \\
Cost(3, \{2, 4, 5\}, 1) &= \min \{ d[3, 2] \\
&\quad + Cost(2, \{4, 5\}, 1), d[3, 4] \\
&\quad + Cost(4, \{2, 5\}, 1), d[3, 5] \\
&\quad + Cost(5, \{2, 4\}, 1)\} \\
&= \min \{ 12 + 20, 8 + 34, 7 + 19\} \\
&= \min\{32, 42, 26\} = 26 \\
Cost(4, \{2, 3, 5\}, 1) &= \min \{ d[4, 2] \\
&\quad + Cost(2, \{3, 5\}, 1), d[4, 3] \\
&\quad + Cost(3, \{2, 5\}, 1), d[4, 5] \\
&\quad + Cost(5, \{2, 3\}, 1)\} \\
&= \min \{23 + 30, 24 + 32, 6 + 17\} \\
&= \min\{53, 56, 23\} = 23 \\
Cost(5, \{2, 3, 4\}, 1) &= \min \{ d[5, 2] \\
&\quad + Cost(2, \{3, 4\}, 1), d[5, 3] \\
&\quad + Cost(3, \{2, 4\}, 1), d[5, 4] \\
&\quad + Cost(4, \{2, 3\}, 1)\} \\
&= \min \{4 + 30, 8 + 27, 11 + 36\} \\
&= \min\{34, 35, 47\} = 34
\end{aligned}$$

Step 5 : In this step, we will find the minimum distance by visiting 4 cities as an intermediate city.

$$\begin{aligned}
Cost(1, \{2, 3, 4, 5\}, 1) &= \min \{ d[1, 2] \\
&\quad + Cost(2, \{3, 4, 5\}, 1), d[1, 3] \\
&\quad + Cost(3, \{2, 4, 5\}, 1), d[1, 4] \\
&\quad + Cost(4, \{2, 3, 5\}, 1), d[1, 5] \\
&\quad + Cost(5, \{2, 3, 4\}, 1)\} \\
&= \min \{ 24 + 25, 11 + 26, 10 \\
&\quad + 23, 9 + 34 \} \\
&= \min\{49, 37, 33, 43\} = 33
\end{aligned}$$

Thus, minimum length tour would be of 33.

Trace the path

- Let us find the path that gives the distance of 33.
- Cost(1, {2, 3, 4, 5}, 1) is minimum due to d[1, 4], so move from 1 to 4. Path = {1, 4}.
- Cost(4, {2, 3, 5}, 1) is minimum due to d[4, 5], so move from 4 to 5. Path = {1, 4, 5}.
- Cost(5, {2, 3}, 1) is minimum due to d[5, 2], so move from 5 to 2. Path = {1, 4, 5, 2}.
- Cost(2, {3}, 1) is minimum due to d[2, 3], so move from 2 to 3. Path = {1, 4, 5, 2, 3}.
- All cities are visited so come back to 1. Hence the optimum tour would be 1 – 4 – 5 – 2 – 3 – 1.

Ex. 4.10.2

Explain the travelling salesman problem as dynamic programming algorithm strategy. Discuss the time and space complexities. Find out the solution for following examples.

	City 1	City 2	City 3	City 4
Pers 1	0	10	15	20
Pers 2	5	0	9	10
Pers 3	6	13	0	12
Pers 4	8	8	9	0

Soln. :

Let us start our tour from city 1.

Step 1 : Initially, we will find the distance between city 1 and city {2, 3, 4} without visiting any intermediate city.

Cost(x, y, z) represents the distance from x to z and y as an intermediate city.

$$\begin{aligned}
Cost(2, \phi, 1) &= d[2, 1] = 5 \\
Cost(3, \phi, 1) &= d[3, 1] = 6 \\
Cost(4, \phi, 1) &= d[4, 1] = 8
\end{aligned}$$

Step 2 : In this step, we will find the minimum distance by visiting 1 city as an intermediate city.

$$\begin{aligned}
Cost\{2, \{3\}, 1\} &= d[2, 3] + Cost(3, \phi, 1) \\
&= 9 + 6 = 15 \\
Cost\{2, \{4\}, 1\} &= d[2, 4] + Cost(4, \phi, 1) \\
&= 10 + 8 = 18 \\
Cost\{3, \{2\}, 1\} &= d[3, 2] + Cost(2, \phi, 1) \\
&= 13 + 5 = 18 \\
Cost\{3, \{4\}, 1\} &= d[3, 4] + Cost(4, \phi, 1) \\
&= 12 + 8 = 20 \\
Cost\{4, \{2\}, 1\} &= d[4, 2] + Cost(2, \phi, 1) \\
&= 8 + 5 = 13 \\
Cost\{4, \{3\}, 1\} &= d[4, 3] + Cost(3, \phi, 1) \\
&= 9 + 6 = 15
\end{aligned}$$

Step 3 : In this step, we will find the minimum distance by visiting 2 cities as an intermediate city.

$$\begin{aligned}
Cost(2, \{3, 4\}, 1) &= \min \{d[2, 3] + Cost(3, \{4\}, 1), \\
&\quad d[2, 4] + Cost(4, \{3\}, 1)]\} \\
&= \min \{ [9 + 20], [10 + 15] \} \\
&= \min\{29, 25\} = 25 \\
Cost(3, \{2, 4\}, 1) &= \min \{ d[3, 2] + Cost(2, \{4\}, 1), \\
&\quad d[3, 4] + Cost(4, \{2\}, 1)]\} \\
&= \min \{ [13 + 18], [12 + 13] \} \\
&= \min\{31, 25\} = 25 \\
Cost(4, \{2, 3\}, 1) &= \min\{ d[4, 2] + Cost(2, \{3\}, 1), \\
&\quad d[4, 3] + Cost(3, \{2\}, 1)]\} \\
&= \min \{ [8 + 15], [9 + 18] \} \\
&= \min\{23, 27\} = 23
\end{aligned}$$

Step 4 : In this step, we will find the minimum distance by visiting 3 cities as intermediate city.

$$\begin{aligned} Cost(1, \{2, 3, 4\}, 1) &= \min\{d[1, 2] + Cost(2, \{3, 4\}, 1), \\ &\quad d[1, 3] + Cost(3, \{2, 4\}, 1), \\ &\quad d[1, 4] + Cost(4, \{2, 3\}, 1)\} \\ &= \min\{10 + 25, 15 + 25, 20 + 23\} \\ &= \min\{35, 40, 43\} = 35 \end{aligned}$$

Thus, minimum length tour would be of 35.

Trace the path

- Let us find the path that gives the distance of 35. $Cost(1, \{2, 3, 4\}, 1)$ is minimum due to d[1, 2], so move from 1 to 2. Path = {1, 2}
- $Cost(2, \{3, 4\}, 1)$ is minimum due to d[2, 4], so move from 2 to 4. Path = {1, 2, 4}
- $Cost(4, \{3\}, 1)$ is minimum due to d[4, 3], so move from 4 to 3. Path = {1, 2, 4, 3}. All cities are visited so come back to 1. Hence the optimum tour would be 1 – 2 – 4 – 3 – 1.

Ex. 4.10.3

Define the Traveling Salesperson Problem. Solve the TSP problem using Dynamic programming where the edge lengths are given as :

0	9	8	8
12	0	13	6
10	9	0	5
20	15	10	0

Soln. :

Traveling salesman problem is stated as, "Given a set of n cities and distance between each pair of cities, find the minimum length path such that it covers each city exactly once and terminates the tour at starting city."

Let us start our tour from city 1.

Step 1 : Initially, we will find the distance between city 1 and city {2, 3, 4} without visiting any intermediate city.

Cost(x, y, z) represents the distance from x to z and y as an intermediate city.

$$\begin{aligned} Cost(2, \phi, 1) &= d[2, 1] = 12 \\ Cost(3, \phi, 1) &= d[3, 1] = 10 \\ Cost(4, \phi, 1) &= d[4, 1] = 20 \end{aligned}$$

Step 2 : In this step, we will find the minimum distance by visiting 1 city as intermediate city.

$$\begin{aligned} Cost\{2, \{3\}, 1\} &= d[2, 3] + Cost(3, \phi, 1) \\ &= 13 + 10 = 23 \\ Cost\{2, \{4\}, 1\} &= d[2, 4] + Cost(4, \phi, 1) \\ &= 6 + 20 = 26 \\ Cost\{3, \{2\}, 1\} &= d[3, 2] + Cost(2, \phi, 1) \\ &= 9 + 12 = 21 \\ Cost\{3, \{4\}, 1\} &= d[3, 4] + Cost(4, \phi, 1) \\ &= 5 + 20 = 25 \\ Cost\{4, \{2\}, 1\} &= d[4, 2] + Cost(2, \phi, 1) \\ &= 15 + 12 = 27 \\ Cost\{4, \{3\}, 1\} &= d[4, 3] + Cost(3, \phi, 1) \\ &= 10 + 10 = 20 \end{aligned}$$

Step 3 : In this step, we will find the minimum distance by visiting 2 cities as an intermediate city.

$$\begin{aligned} Cost(2, \{3, 4\}, 1) &= \min\{d[2, 3] + Cost(3, \{4\}, 1), \\ &\quad d[2, 4] + Cost(4, \{3\}, 1)\}\} \\ &= \min\{[13 + 25], [6 + 20]\} \\ &= \min\{38, 26\} = 26 \\ Cost(3, \{2, 4\}, 1) &= \min\{d[3, 2] + Cost(2, \{4\}, 1), \\ &\quad d[3, 4] + Cost(4, \{2\}, 1)\}\} \\ &= \min\{[9 + 26], [5 + 27]\} \\ &= \min\{35, 32\} = 32 \\ Cost(4, \{2, 3\}, 1) &= \min\{d[4, 2] + Cost(2, \{3\}, 1), \\ &\quad d[4, 3] + Cost(3, \{2\}, 1)\}\} \\ &= \min\{[15 + 23], [10 + 21]\} \\ &= \min\{38, 31\} = 31 \end{aligned}$$

Step 4 : In this step, we will find the minimum distance by visiting 3 cities as intermediate city.

$$\begin{aligned} Cost(1, \{2, 3, 4\}, 1) &= \min\{d[1, 2] + Cost(2, \{3, 4\}, 1), \\ &\quad [1, 3] + Cost(3, \{2, 4\}, 1), \\ &\quad d[1, 4] + Cost(4, \{2, 3\}, 1)\} \\ &= \min\{9 + 26, 8 + 32, 8 + 31\} \\ &= \min\{35, 40, 39\} = 35 \end{aligned}$$

Thus, minimum length tour would be of 35.

Trace the path

- Let us find the path that gives the distance of 35. $Cost(1, \{2, 3, 4\}, 1)$ is minimum due to d[1, 2], so move from 1 to 2. Path = {1, 2}
- $Cost(2, \{3, 4\}, 1)$ is minimum due to d[2, 4], so move from 2 to 4. Path = {1, 2, 4}
- $Cost(4, \{3\}, 1)$ is minimum due to d[4, 3], so move from 4 to 3. Path = {1, 2, 4, 3}. All cities are visited so come back to 1. Hence the optimum tour would be 1 – 2 – 4 – 3 – 1

Ex. 4.10.4 **MU - Dec. 2014, 10 Marks**

Find the path of travelling salesperson problem of given graph.

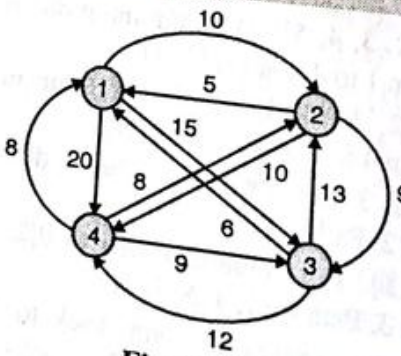


Fig. P. 4.10.4

Soln. :

Above graph can be represented as,

0	10	15	20
5	0	9	10
6	13	0	12
8	8	9	0

Let us start our tour from city 1.

Step 1 : Initially, we will find the distance between city 1 and city {2, 3, 4} without visiting any intermediate city.

Cost(x, y, z) represents the distance from x to z and y as an intermediate city.

$$\text{Cost}(2, \phi, 1) = d[2, 1] = 5$$
$$\text{Cost}(3, \phi, 1) = d[3, 1] = 6$$
$$\text{Cost}(4, \phi, 1) = d[4, 1] = 8$$

Step 2 : In this step, we will find the minimum distance by visiting 1 city as intermediate city.

$$\text{Cost}\{2, \{3\}, 1\} = d[2, 3] + \text{Cost}(3, \phi, 1) = 9 + 6 = 15$$
$$\text{Cost}\{2, \{4\}, 1\} = d[2, 4] + \text{Cost}(4, \phi, 1) = 10 + 8 = 18$$
$$\text{Cost}\{3, \{2\}, 1\} = d[3, 2] + \text{Cost}(2, \phi, 1) = 13 + 5 = 18$$
$$\text{Cost}\{3, \{4\}, 1\} = d[3, 4] + \text{Cost}(4, \phi, 1) = 12 + 8 = 20$$
$$\text{Cost}\{4, \{2\}, 1\} = d[4, 2] + \text{Cost}(2, \phi, 1) = 8 + 5 = 13$$
$$\text{Cost}\{4, \{3\}, 1\} = d[4, 3] + \text{Cost}(3, \phi, 1) = 9 + 6 = 15$$

Step 3 : In this step, we will find the minimum distance by visiting 2 cities as an intermediate city.

$$\text{Cost}(2, \{3, 4\}, 1) = \min\{d[2, 3] + \text{Cost}(3, \{4\}, 1), d[2, 4] + \text{Cost}(4, \{3\}, 1)]\} = \min\{[9 + 20], [10 + 15]\} = \min\{29, 25\} = 25$$

$$\text{Cost}(3, \{2, 4\}, 1) = \min\{d[3, 2] + \text{Cost}(2, \{4\}, 1), d[3, 4] + \text{Cost}(4, \{2\}, 1)]\} = \min\{[13 + 18], [12 + 13]\} = \min\{31, 25\} = 25$$

$$\text{Cost}(4, \{2, 3\}, 1) = \min\{d[4, 2] + \text{Cost}(2, \{3\}, 1), d[4, 3] + \text{Cost}(3, \{2\}, 1)]\} = \min\{[8 + 15], [9 + 18]\} = \min\{23, 27\} = 23$$

Step 4 : In this step, we will find the minimum distance by visiting 3 cities as intermediate city.

$$\text{Cost}(1, \{2, 3, 4\}, 1) = \min\{d[1, 2] + \text{Cost}(2, \{3, 4\}, 1), [1, 3] + \text{Cost}(3, \{2, 4\}, 1), d[1, 4] + \text{Cost}(4, \{2, 3\}, 1)\} = \min\{10 + 25, 15 + 25, 20 + 23\} = \min\{35, 40, 43\} = 35$$

Thus, minimum length tour would be of 35.

Trace the path

- Let us find the path that gives the distance of 35. Cost(1, {2, 3, 4}, 1) is minimum due to d[1, 2], so move from 1 to 2. Path = {1, 2}
- Cost(2, {3, 4}, 1) is minimum due to d[2, 4], so move from 2 to 4. Path = {1, 2, 4}
- Cost(4, {3}, 1) is minimum due to d[4, 3], so move from 4 to 3. Path = {1, 2, 4, 3}. All cities are visited so come back to 1. Hence the optimum tour would be 1 – 2 – 4 – 3 – 1.

Syllabus Topic : Longest Common Subsequence

4.11 Longest Common Subsequence

→ **(May 14, May 17)**

Q. Explain longest common subsequence with an example. **MU - May 2014, 10 Marks**

Q. What is a longest common subsequence problem? **MU - May 2017, 5 Marks**

Q. What is LCS problem? How to solve it using dynamic programming? Explain in detail. **(7 Marks)**

☞ **Problem**

The longest common sequence is the problem of finding maximum length common subsequence from given two strings A and B.

☞ **Explanation**

Let $A = < a_1, a_2, a_3 \ldots a_n >$ and $B = < b_1, b_2, b_3 \ldots b_m >$ be two strings over analphabets. Then B is a subsequence of A if B can be generated by striking out some elements from A.

By subsequence, we mean that the values must occur in the order of the sequence, but they need not be consecutive.

If A = < X, Y, X, X, Z, Y, X > and B = < X, X, Y, X > then by deleting A[2], A[4] and A[5] from A, we can derive B. So B is the subsequence of C.

Common sub sequence of A and B is the sub sequence which can be generated by striking some characters from A and B both. If A = {a, b, a, c, b, c, b} and B = {a, b, c, b, b, a, c}, then the sequence {a, c}, {a, b, c}, {a, c, c}, {a, b, c, b} etc are common sub sequences of A and B, but {a, b, c, b, a} is not. {a, b, c, b, a} is sub sequence of B but it is not sub sequence of A.

A string of length m has 2^m subsequences. So finding longest common subsequences using brute force approach takes exponential time. Such algorithms are practically not useful for long sequences.

Mathematical Formulation

Let us consider two strings A_m and B_n of length m and n respectively. If the last character of both strings is same i.e. $a_m = b_n$, then the length of LCS is incremented by one and length of both strings is reduced by one. The new problem is now to find LCS of strings A_{m-1} and B_{n-1}.

If $a_m \neq b_n$, we shall consider two subproblems.

- Apply LCS on strings A_{m-1} and B
- Apply LCS on strings A and B_{n-1}

Select the result which gives the longest subsequence.

Thus, optimal substructure of LCS problem is defined as,

$$LCS[i, j] = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0 \\ LCS[i-1, j-1] + 1, & \text{if } a_i = b_j \\ \max(LCS[i, j-1], LCS[i-1, 1]), & a_i \neq b_j \end{cases}$$

The algorithm to solve LCS problem is described below :

```
Algorithm LONGEST_COMMON_SUBSEQUENCE(X, Y)
// X is string of length n
// Y is string of length m

for i ← 1 to m do
    LCS [i, 0] ← 0
    end
for j ← 0 to n do
    LCS [0, j] ← 0
end
for i ← 1 to m do
    for j ← 1 to n do
    if  X_i = = Y_j  then
    LCS[i, j] ← LCS[i – 1, j – 1] + 1
    else
    if LCS[i – 1, j] ≥ LCS[i, j – 1]  then
        LCS[i, j] ← LCS[i – 1, j]
    else
        LCS[i, j]← LCS[i, j – 1]
    end
    end
    end
end
return LCS
```

Complexity analysis

In brute force attack, we need to perform check every subsequence of P[1…m] to see if it is also a subsequence of Q[1…n]. Checking membership of one subsequence of P[1…m] into Q[1…n] takes O(n) time. 2^m subsequences are possible for string P of length m. So worst case running time of brute force approach would be $O(n.2^m)$

In dynamic programming, the only table of size m × n is filled up using two nested for loops. So running time of dynamic programming approach would take O(mn), same is the space complexity.

Ex. 4.11.1 MU - May 2017, 10 Marks

Find LCS for following string X = ACBAED and Y = ABCABE

Soln. :

Given two strings are P = <MLNOM> and Q = <MNOM>

Optimal substructure of LCS problem using dynamic programming is given as :

$$LCS[i, j] = \begin{cases} 0 & \text{, if } i = 0 \text{ or } j = 0 \\ 1 + LCS[i-1, j-1] & \text{, if } P_i = Q_j \\ \max(LCS[i, j-1], LCS[i-1, j]) & \text{if } P_i \neq Q_j \end{cases}$$

Initially, table looks like,

Table P. 4.11.1

			Y_j						
			0	1	2	3	4	5	6
			0	A	B	C	A	B	E
X_i	0	0	0	0	0	0	0	0	0
	1	A	0						
	2	C	0						
	3	B	0						
	4	A	0						
	5	E	0						
	6	D	0						

Let's compute the remaining cell values row by row :

To be consistent with the previous examples, let P = X and Q = Y ⇒ P = ABCAED and Q = ABCABE

Computation for row 1

LCS [1, 1] ⇒ i = 1, j = 1, X_i = A, Y_j = A

$X_i = Y_j$ ⇒ LCS [i, j] = 1 + LCS[i – 1, j – 1]

LCS [1, 1] = 1 + LCS[0, 0]

= 1 + 0 = 1

LCS [1, 2] ⇒ i = 1, j = 2, X_i = A, Y_j = B

$X_i \neq Y_j \Rightarrow$ LCS [i, j] = max (LCS [i – 1 , j], LCS [i, j – 1])

LCS [1, 2] = max (LCS [0, 2], LCS [1, 1])
= max (0, 1) = 1

LCS [1, 3] $\Rightarrow$ i = 1, j = 3, $X_i = A$, $Y_j = C$
$X_i \neq Y_j \Rightarrow$ LCS [i, j] = max (LCS [i – 1 , j], LCS [i, j – 1])
LCS [1, 3] = max(LCS[0, 3], LCS[1, 2])
= max(0, 1) = 1

LCS [1, 4] $\Rightarrow$ i = 1, j = 4, $X_i = A$, $Y_j = A$
$X_i = Y_j \Rightarrow$ LCS [i, j] = 1 + LCS[i – 1, j – 1]
LCS [1, 4] = 1 + LCS[0, 3]
= 1 + 0 = 1

LCS [1, 5] $\Rightarrow$ i = 1, j = 5, $X_i = A$, $Y_j = B$
$X_i \neq Y_j \Rightarrow$ LCS [i, j] = max (LCS [i – 1 , j], LCS [i, j – 1])
LCS [1, 5] = max(LCS[0, 5], LCS[1, 4])
= max(0, 1) = 1

LCS [1, 6] $\Rightarrow$ i = 1, j = 6, $X_i = A$, $Y_j = E$
$X_i \neq Y_j \Rightarrow$ LCS [i, j] = max (LCS [i – 1 , j], LCS [i, j – 1])
LCS [1, 6] = max(LCS[0, 6], LCS[1, 5])
= max(0, 1) = 1

Computation for row 2

LCS [2, 1] $\Rightarrow$ i = 2, j = 1, $X_i = C$, $Y_j = A$
$X_i \neq Y_j \Rightarrow$ LCS [i, j] = max (LCS [i – 1 , j], LCS [i, j – 1])
LCS [2, 1] = max (LCS [1, 1], LCS [2, 0])
= max (1, 0) = 1

LCS [2, 2] $\Rightarrow$ i = 2, j = 2, $X_i = C$, $Y_j = B$
$X_i \neq Y_j \Rightarrow$ LCS [i, j] = max (LCS [i – 1 , j], LCS [i, j – 1])
LCS [2, 2] = max (LCS [1, 2], LCS [2, 1])
= max (1, 1) = 1

LCS [2, 3] $\Rightarrow$ i = 2, j = 3, $X_i = C$, $Y_j = C$
$X_i = Y_j \Rightarrow$ LCS [i, j] = 1 + LCS[i – 1, j – 1]
LCS [2, 3] = 1 + LCS[1, 2]
= 1 + 1 = 2

LCS [2, 4] $\Rightarrow$ i = 2, j = 4, $X_i = C$, $Y_j = A$
$X_i \neq Y_j \Rightarrow$ LCS [i, j] = max (LCS [i – 1 , j], LCS [i, j – 1])
LCS [2, 4] = max (LCS [1, 4], LCS [2, 3])
= max (1, 2) = 2

LCS [2, 5] $\Rightarrow$ i = 2, j = 5, $X_i = C$, $Y_j = B$
$X_i \neq Y_j \Rightarrow$ LCS [i, j] = max (LCS [i – 1 , j], LCS [i, j – 1])
LCS [2, 5] = max (LCS [1, 5], LCS [2, 4])
= max (1, 2) = 2

LCS [2, 6] $\Rightarrow$ i = 2, j = 6, $X_i = C$, $Y_j = E$
$X_i \neq Y_j \Rightarrow$ LCS [i, j] = max (LCS [i – 1 , j], LCS [i, j – 1])
LCS [2, 6] = max (LCS [1, 6], LCS [2, 5])
= max (1, 2) = 2

Computation for row 3 :

LCS [3, 1] $\Rightarrow$ i = 3, j = 1, $X_i = B$, $Y_j = A$
$X_i \neq Y_j \Rightarrow$ LCS [i, j] = max (LCS [i – 1 , j], LCS [i, j – 1])
LCS [3, 1] = max (LCS [2, 1], LCS [3, 0])
= max (1, 0) = 1

LCS [3, 2] $\Rightarrow$ i = 3, j = 2, $X_i = B$, $Y_j = B$
$X_i = Y_j \Rightarrow$ LCS [i, j] = 1 + LCS[i – 1, j – 1]
LCS [3, 2] = 1 + LCS[2, 1] = 1 + 1 = 2

LCS [3, 3] $\Rightarrow$ i = 3, j = 3, $X_i = B$, $Y_j = C$
$X_i \neq Y_j \Rightarrow$ LCS [i, j] = max (LCS [i – 1 , j], LCS [i, j – 1])
LCS [3, 3] = max(LCS[2, 3], LCS[3, 2])
= max(2, 2) = 2

LCS [3, 4] $\Rightarrow$ i = 3, j = 4, $X_i = B$, $Y_j = A$
$X_i \neq Y_j \Rightarrow$ LCS [i, j] = max (LCS [i – 1 , j], LCS [i, j – 1])
LCS [3, 4] = max (LCS [2, 4], LCS [3, 3]) = max (2, 2) = 2

LCS [3, 5] $\Rightarrow$ i = 3, j = 5, $X_i = B$, $Y_j = B$
$X_i = Y_j \Rightarrow$ LCS [i, j] = 1 + LCS[i – 1, j – 1]
LCS [3, 5] = 1 + LCS[2, 4] = 1 + 2 = 3

LCS [3, 6] $\Rightarrow$ i = 3, j = 6, $X_i = B$, $Y_j = E$
$X_i \neq Y_j \Rightarrow$ LCS [i, j] = max (LCS [i – 1 , j], LCS [i, j – 1])
LCS [3, 6] = max (LCS [2, 6], LCS [3, 5]) = max (2, 3) = 3

Computation for row 4

LCS [4, 1] $\Rightarrow$ i = 4, j = 1, $X_i = A$, $Y_j = A$
$X_i = Y_j \Rightarrow$ LCS [i, j] = 1 + LCS[i – 1, j – 1]
LCS [4, 1] = 1 + LCS[3, 0] = 1 + 0 = 1

LCS [4, 2] $\Rightarrow$ i = 4, j = 2, $X_i = A$, $Y_j = B$
$X_i \neq Y_j \Rightarrow$ LCS [i, j] = max (LCS [i – 1 , j], LCS [i, j – 1])
LCS [4, 2] = max (LCS [3, 2], LCS [4, 1]) = max (2, 1) = 2

LCS [4, 3] $\Rightarrow$ i = 4, j = 3, $X_i = A$, $Y_j = C$
$X_i \neq Y_j \Rightarrow$ LCS [i, j] = max (LCS [i – 1 , j], LCS [i, j – 1])
LCS [4, 3] = max (LCS [3, 3], LCS [4, 2])
= max (2, 2) = 2

LCS [4, 4] $\Rightarrow$ i = 4, j = 4, X_i = A, Y_j = A

$X_i = Y_j \Rightarrow$ LCS [i, j] = 1 + LCS[i – 1, j – 1]

LCS [4, 4] = 1 + LCS[3, 3] = 1 + 2 = 3

LCS [4, 5] $\Rightarrow$ i = 4, j = 5, X_i = A, Y_j = B

$X_i \neq Y_j \Rightarrow$ LCS [i, j] = max (LCS [i – 1 , j], LCS [i, j – 1])

LCS [4, 5] = max (LCS [3, 5], LCS [4, 4])

= max (3, 3) = 3

LCS [4, 6] $\Rightarrow$ i = 4, j = 6, X_i = A, Y_j = E

$X_i \neq Y_j \Rightarrow$ LCS [i, j]

= max (LCS [i – 1 , j], LCS [i, j – 1])

LCS [4, 6] = max (LCS [3, 6], LCS [4, 5])

= max (3, 3) = 3

Computation for row 5

LCS [5, 1] $\Rightarrow$ i = 5, j = 1, X_i = E, Y_j = A

$X_i \neq Y_j \Rightarrow$ LCS [i, j] = max (LCS [i – 1 , j], LCS [i, j – 1])

LCS [5, 1] = max (LCS [4, 1], LCS [5, 0])

= max (1, 0) = 1

LCS [5, 2] $\Rightarrow$ i = 5, j = 2, X_i = E, Y_j = B

$X_i \neq Y_j \Rightarrow$ LCS [i, j] = max (LCS [i – 1 , j], LCS [i, j – 1])

LCS [5, 2] = max (LCS [4, 2], LCS [5, 1]) = max (2, 1) = 2

LCS [5, 3] $\Rightarrow$ i = 5, j = 3, X_i = E, Y_j = C

$X_i \neq Y_j \Rightarrow$ LCS [i, j] = max (LCS [i – 1 , j], LCS [i, j – 1])

LCS [5, 3] = max (LCS [4, 3], LCS [5, 2]) = max (2, 2) = 2

LCS [5, 4] $\Rightarrow$ i = 5, j = 4, X_i = E, Y_j = A

$X_i \neq Y_j \Rightarrow$ LCS [i, j] = max (LCS [i – 1 , j], LCS [i, j – 1])

LCS [5, 4] = max (LCS [4, 4], LCS [5, 3]) = max (3, 2) = 3

LCS [5, 5] $\Rightarrow$ i = 5, j = 5, X_i = E, Y_j = B

$X_i \neq Y_j \Rightarrow$ LCS [i, j] = max (LCS [i – 1 , j], LCS [i, j – 1])

LCS [5, 5] = max (LCS [4, 5], LCS [5, 4]) = max (3, 3) = 3

LCS [5, 6] $\Rightarrow$ i = 5, j = 6, X_i = E, Y_j = E

$X_i = Y_j \Rightarrow$ LCS [i, j] = 1 + LCS[i – 1, j – 1]

LCS [5, 6] = 1 + LCS[4, 5] = 1 + 3 = 4

Computation for row 5

LCS [6, 1] $\Rightarrow$ i = 6, j = 1, X_i = D, Y_j = A

$X_i \neq Y_j \Rightarrow$ LCS [i, j] = max (LCS [i – 1 , j], LCS [i, j – 1])

LCS [6, 1] = max (LCS [5, 1], LCS [6, 0])

= max (1, 0) = 1

LCS [6, 2] $\Rightarrow$ i = 6, j = 2, X_i = D, Y_j = B

$X_i \neq Y_j \Rightarrow$ LCS [i, j] = max (LCS [i – 1 , j], LCS [i, j – 1])

LCS [6, 2] = max (LCS [5, 2], LCS [6, 1])

= max (2, 1) = 2

LCS [6, 3] $\Rightarrow$ i = 6, j = 3, X_i = D, Y_j = C

$X_i \neq Y_j \Rightarrow$ LCS [i, j] = max (LCS [i – 1 , j], LCS [i, j – 1])

LCS [6, 3] = max (LCS [5, 3], LCS [6, 2])

= max (2, 2) = 2

LCS [6, 4] $\Rightarrow$ i = 6, j = 4, X_i = D, Y_j = A

$X_i \neq Y_j \Rightarrow$ LCS [i, j] = max (LCS [i – 1 , j], LCS [i, j – 1])

LCS [6, 4] = max (LCS [5, 4], LCS [6, 3])

= max (3, 2) = 3

LCS [6, 5] $\Rightarrow$ i = 6, j = 5, X_i = D, Y_j = B

$X_i \neq Y_j \Rightarrow$ LCS [i, j] = max (LCS [i – 1 , j], LCS [i, j – 1])

LCS [6, 5] = max (LCS [5, 5], LCS [6, 4])

= max (3, 3) = 3

LCS [6, 6] $\Rightarrow$ i = 6, j = 6, X_i = D, Y_j = E

$X_i \neq Y_j \Rightarrow$ LCS [i, j] = max (LCS [i – 1 , j], LCS [i, j – 1])

LCS [6, 6] = max (LCS [5, 6], LCS [6, 5])

= max (4, 3) = 4

So finally, table would be,

Table P. 4.11.1(a) : LCS table for X = ACBAED and Y = ABCABE

Table P. 4.11.1(a)

			Y_j →						
			0	1	2	3	4	5	6
			0	A	B	C	A	B	E
0	0	0	0	0	0	0	0	0	0
1	A	0	1	1	1	1	1	1	1
2	C	0	1	1	2	2	2	2	2
3	B	0	1	2	2	2	3	3	3
4	A	0	1	2	2	3	3	3	3
5	E	0	1	2	2	3	3	4	4
6	D	0	1	2	2	3	3	4	4

X_i ↓

Find LCS

Longest common subsequence for given strings is of length 4. Let us find LCS of X and Y :

Step 1 : $i = 6, j = 6, X_i = D, Y_j = E$

$X_i \neq Y_j$, and LCS[i, j] is derived from LCS[i – 1, j]. So move in vertical direction.

Solution set S = { }

Step 2 : $i = 5, j = 6, X_i = E, Y_j = E$

$X_i = Y_i$, so add Xi to solution set. And move diagonally back.

Solution set S = { E }

Step 3 : $i = 4, j = 5, X_i = A, Y_j = B$

$X_i \neq Y_j$, and LCS[i, j] is derived from LCS[i – 1, j] or LCS[i, j - 1]. We can move in any direction. Let us choose vertical.

Solution set S = { E }

Step 4 : $i = 3, j = 5, X_i = L, Y_j = B$

$X_i = Y_i$, so add Xi to solution set. And move diagonally back.

Solution set S = { E, B }

Step 5 : $i = 2, j = 4, X_i = C, Y_j = A$

$X_i \neq Y_j$, and LCS[i, j] is derived from LCS[i, j - 1]. So move horizontally.

Solution set S = { E, B }

Step 6 : $i = 2, j = 3, X_i = C, Y_j = C$

$X_i = Y_i$, so add Xi to solution set. And move diagonally back.

Solution set S = { E, B, C }

Step 7 : $i = 1, j = 2, X_i = A, Y_j = B$

$X_i \neq Y_j$, and LCS[i, j] is derived from LCS[i, j - 1]. So move horizontally.

Solution set S = { E, B, C }

Step 8 : $i = 1, j = 2, X_i = A, Y_j = A$

$X_i = Y_i$, so add Xi to solution set. And move diagonally back.

Solution set S = { E, B, C, A }

Moving diagonally back i and j become zero. So stop. We have collected characters from the last position of the string. So reverse the solution set, which is the LCS of X and Y.

So, LCS = ACBE

Ex. 4.11.2

Using algorithm determine an longest common sequence of (A,B,C,D,B,A,C,D,F) and (C,B,A,F)

Soln. :

Let us consider P = (A, B, C, D, B, A, C, D, F) and Q = (C, B, A, F)

Optimal substructure of LCS problem using dynamic programming is given as :

$$LCS[i, j] = \begin{cases} 0 & , \text{if } i = 0 \text{ or } j = 0 \\ 1 + LCS[i-1, j-1] & , \text{if } P_i = Q_i \\ \max(LCS[i, j-1], LCS[i-1, j]) & \text{if } P_i \neq Q_i \end{cases}$$

Initially table looks like,

Table P. 4.11.2

				Q_j →			
		0	1	2	3	4
			C	B	A	F
		0	0	0	0	0
1	A	0				
2	B	0				
3	C	0				
4	D	0				
5	B	0				
6	A	0				
7	C	0				
8	D	0				
9	F	0				

(P_i ↓)

Let's compute the remaining cell values row by row :

Computation for row 1

$LCS[1, 1] \Rightarrow i = 1,\ j = 1,\ P_i = A,\ Q_j = C$

$P_i \neq Q_j \Rightarrow LCS[i, j] = \max(LCS[i, j-1], LCS[i-1, j])$

$\therefore LCS[1, 1] = \max(LCS[1, 0], LCS[0, 1])$

$= \max(0, 0) = 0$

$LCS[1, 2] \Rightarrow P_i = A,\ Q_j = B$

$P_i \neq Q_j \Rightarrow LCS[1, 2] = \max(LCS[1, 1], LCS[0, 2])$

$= \max(0, 0) = 0$

$LCS[1, 3] \Rightarrow P_i = A,\ Q_j = A$

$P_i = Q_j \Rightarrow LCS[1, 3] = 1 + LCS[i-1, j-1]$

$= 1 + LCS[0, 2]$

$= 1 + 0 = 1$

$LCS[1, 4] \Rightarrow P_i = X,\ Q_j = F$

$P_i \neq Q_j \Rightarrow LCS[1, 4] = \max(LCS[1, 3], LCS[0, 4])$

$= \max(1, 0) = 1$

Computation for row 2

$LCS[2, 1] \Rightarrow i = 2,\ j = 1,\ P_i = B,\ Q_j = C$

$P_i \neq Q_j \Rightarrow LCS[2, 1] = \max(LCS[1, 0], LCS[0, 1])$

$= \max(0, 0) = 0$

$LCS[2, 2] \Rightarrow P_i = B,\ Q_j = B$

$P_i = Q_j \Rightarrow LCS[2, 2] = 1 + LCS[1, 1]$

$= 1 + 0 = 1$

$LCS[2, 3] \Rightarrow P_i = B,\ Q_j = A$

$P_i \neq Q_j \Rightarrow LCS[2, 3]$

$= \max(LCS[1, 3], LCS[2, 2])$

$= \max(1, 1) = 1$

LCS [2, 4] ⇒ P_i = B, Q_j = F

$P_i \neq Q_j$ ⇒ LCS [2, 4] = max (LCS [1, 4], LCS [2, 3])
= max(1, 1) = 1

Computation for row 3

LCS [3, 1] ⇒ P_i = C, Q_j = C

$P_i = Q_j$ ⇒ LCS [1, 2] = 1 + LCS[2, 0]
= 1 + 0 = 1

LCS [3, 2] ⇒ P_i = C, Q_j = B

$P_i \neq Q_j$ ⇒ LCS [3, 2] = max (LCS [2, 2], LCS [3, 1])
= max(1, 1) = 1

LCS [3, 3] ⇒ P_i = C, Q_j = A

$P_i \neq Q_j$ ⇒ LCS [3, 3] = max (LCS [2, 3], LCS [3, 2])
= max (1, 1) = 1

LCS [3, 4] ⇒ P_i = C, Q_j = F

$P_i \neq Q_j$ ⇒ LCS [3, 4] = max (LCS [2, 4], LCS [3, 3])
= max(1, 1) = 1

Computation for row 4

LCS [4, 1] ⇒ P_i = D, Q_j = C

$P_i \neq Q_j$ ⇒ LCS [4, 1] = max (LCS [3, 1], LCS [4, 0])
= max(1, 0) = 1

LCS [4, 2] ⇒ P_i = D, Q_j = B

$P_i \neq Q_j$ ⇒ LCS [4, 2] = max (LCS [3, 2], LCS [4, 1])
= max(1, 1) = 1

LCS [4, 3] ⇒ P_i = D, Q_j = A

$P_i \neq Q_j$ ⇒ LCS [4, 3] = max (LCS [3, 3], LCS [4, 2])
= max (1, 1) = 1

LCS [4, 4] ⇒ P_i = D, Q_j = F

$P_i \neq Q_j$ ⇒ LCS [4, 4] = max (LCS [3, 4], LCS [4, 3])
= max (1, 1) = 1

Computation for row 5

LCS [5, 1] ⇒ P_i = B, Q_j = C

$P_i \neq Q_j$ ⇒ LCS [5, 1] = max (LCS [4, 1], LCS [5, 0]) = max (1, 0) = 1

LCS [5, 2] ⇒ P_i = B, Q_j = B

$P_i = Q_j$ ⇒ LCS [5, 2] = 1 + LCS[i − 1, j − 1]
= 1 + LCS[4, 1]= 1 + 1 = 2

LCS [5, 3] ⇒ P_i = B, Q_j = A

$P_i \neq Q_j$ ⇒ LCS [5, 3] = max (LCS [4, 3], LCS [5, 2])
= max (1, 2) = 2

LCS [5, 4] ⇒ P_i = B, Q_j = F

$P_i \neq Q_j$ ⇒ LCS [5, 4] = max (LCS [4, 4], LCS [5, 3])
= max (1, 2) = 2

Computation for row 6

LCS [6, 1] ⇒ P_i = A, Q_j = C

$P_i \neq Q_j$ ⇒ LCS [6, 1] = max (LCS [5, 1], LCS [6, 0])
= max (1, 0) = 1

LCS [6, 2] ⇒ P_i = A, Q_j = B

$P_i \neq Q_j$ ⇒ LCS [6, 2] = max (LCS [5, 2], LCS [6, 1])
= max (2, 1) = 2

LCS [6, 3] ⇒ P_i = A, Q_j = A

$P_i = Q_j$ ⇒ LCS [6, 3] = 1 + LCS[i − 1, j − 1]
= 1 + LCS[5, 2] = 1 + 2 = 3

LCS [6, 4] ⇒ P_i = A, Q_j = F

$P_i \neq Q_j$ ⇒ LCS [6, 4] = max (LCS [5, 4], LCS [6, 3])
= max (2, 3) = 3

Computation for row 7

LCS [7, 1] ⇒ P_i = C, Q_j = C

$P_i = Q_j$ ⇒ LCS [7, 1] = 1 + LCS[i − 1, j − 1]
= 1 + LCS[6, 0]= 1 + 0 = 1

LCS [7, 2] ⇒ P_i = C, Q_j = B

$P_i \neq Q_j$ ⇒ LCS [7, 2] = max (LCS [6, 2], LCS [7, 1])
= max(2, 1) = 2

LCS [7, 3] ⇒ P_i = C, Q_j = A

$P_i \neq Q_j$ ⇒ LCS [7, 3] = max (LCS [6, 3], LCS [7, 2])
= max(3, 2) = 3

LCS [7, 4] ⇒ P_i = C, Q_j = F

$P_i \neq Q_j$ ⇒ LCS [7, 4]
= max (LCS [6, 4], LCS [7, 3])
= max(3, 3) = 3

Computation for row 8

LCS [8, 1] ⇒ P_i = D, Q_j = C

$P_i \neq Q_j$ ⇒ LCS [8, 1] = max (LCS [7, 1], LCS [8, 0]) = max(1, 0) = 1

LCS [8, 2] ⇒ P_i = D, Q_j = B

$P_i \neq Q_j$ ⇒ LCS [8, 2] = max (LCS [7, 2], LCS [8, 1]) = max(2, 1) = 2

LCS [8, 3] ⇒ P_i = D, Q_j = A

$P_i \neq Q_j$ ⇒ LCS [8, 3] = max (LCS [7, 3], LCS [8, 2])
= max(3, 2) = 3

LCS [8, 4] ⇒ P_i = D, Q_j = F

$P_i \neq Q_j$ ⇒ LCS [8, 4]
=max (LCS [7, 4], LCS [8, 3]) = max(3, 3) = 3

Computation for row 9

LCS [9, 1] ⇒ P_i = F, Q_j = C

$P_i \neq Q_j$ ⇒ LCS [9, 1] = max (LCS [8, 1], LCS [9, 0]) = max(1, 0) = 1

LCS [9, 2] ⇒ P_i = F, Q_j = B

$P_i \neq Q_j$ ⇒ LCS [9, 2]
=max (LCS [8, 2], LCS [9, 1]) = max(2, 1) = 2

LCS [9, 3] ⇒ P_i = F, Q_j = A

$P_i \neq Q_j$ ⇒ LCS [9, 3]
=max (LCS [8, 3], LCS [9, 2]) = max(3, 2) = 3

LCS [9, 4] ⇒ P_i = F, Q_j = F

$P_i = Q_j$ ⇒ LCS [9, 4]

$= 1 + LCS[i-1, j-1] = 1 + LCS[8, 3] = 1 + 3 = 4$

So finally table would be

Table P. 4.11.2(a)

		0	1	2	3	4
			C	B	A	F
0		0	0	0	0	0
1	A	0	0	0	1	1
2	B	0	0	1	1	1
3	C	0	1	1	1	1
4	D	0	1	1	1	1
5	B	0	1	2	2	2
6	A	0	1	2	3	3
7	C	0	1	2	3	3
8	D	0	1	2	3	3
9	F	0	1	2	3	4

Longest common subsequence for given strings is of length 4. Let us find LCS of P and Q :

Step 1 : $i = 9, j = 4, P_i = F, Q_j = F$

$P_i = Q_i$, so add P_i to solution set. And move diagonally back.

Solution set S = { F, }

Step 2 : $i = 8, j = 3, P_i = D, Q_j = A$

$P_i \neq Q_j$, and LCS[i, j] is derived from LCS[i – 1, j]. So move in vertical direction.

Solution set S = { F }

Step 3 : $i = 7, j = 3, P_i = C, Q_j = A$

$P_i \neq Q_j$, and LCS[i, j] is derived from LCS[i – 1, j]. So move in vertical direction.

Solution set S = { F }

Step 4 : $i = 6, j = 3, P_i = A, Q_j = A$

$P_i = Q_i$, so add Pi to solution set. And move diagonally back.

Solution set S = { F, A }

Step 5 : $i = 5, j = 2, P_i = B, Q_j = B$

$P_i = Q_i$, so add Pi to solution set. And move diagonally back.

Solution set S = { F, A, B }

Step 6 : $i = 4, j = 1, P_i = D, Q_j = C$

$P_i \neq Q_j$, and LCS[i, j] is derived from LCS[i, j – 1]. So move in horizontal direction.

Solution set S = { F, A, B }

Step 7 : $i = 3, j = 1, P_i = C, Q_j = C$

$P_i = Q_i$, so add Pi to the solution set. And move diagonally back.

Solution set S = { F, A, B, C }

Moving diagonally back j becomes zero. So stop. We have collected characters from the last position of the string. So reverse the solution set, which is the LCS of P and Q.

So, LCS = CBAF

Ex. 4.11.3

Given two sequences of characters, P = <MLNOM> Q=<MNOM>. Obtain the longest common subsequence.

Soln. :

Given two strings are P = <MLNOM> and Q = <MNOM>

Optimal substructure of LCS problem using dynamic programming is given as :

$$LCS[i, j] = \begin{cases} 0 & \text{, if } i = 0 \text{ or } j = 0 \\ 1 + LCS[i-1, j-1] & \text{, if } P_i = Q_j \\ \max(LCS[i, j-1], LCS[i-1, j]) & \text{if } P_i \neq Q_j \end{cases}$$

Initially, table looks like,

Table P. 4.11.3

			Q_j →				
		0	1	2	3	4
			M	N	O	M
		0	0	0	0	0
1	M	0				
2	L	0				
3	N	0				
4	O	0				
5	M	0				

P_i ↓

Let's compute the remaining cell values row by row :

Computation for row 1 :

$LCS[1, 1] \Rightarrow i = 1, \ j = 1, \ P_i = M, \ Q_j = M$

$P_i = Q_j \Rightarrow LCS[i, j] = 1 + LCS[i-1, j-1]$

$LCS[1, 1] = 1 + LCS[0, 0]$

$= 1 + 0 = 1$

$LCS[1, 2] \Rightarrow i = 1, \ j = 2, \ P_i = M, \ Q_j = N$

$P_i \neq Q_j \Rightarrow LCS[i, j] = \max(LCS[i-1, j], LCS[i, j-1])$

$LCS[1, 2] = \max(LCS[0, 2], LCS[1, 1])$

$= \max(0, 1) = 1$

$LCS[1, 3] \Rightarrow i = 1, \ j = 3, \ P_i = M, \ Q_j = O$

$P_i \neq Q_j \Rightarrow LCS[i, j] = \max(LCS[i-1, j], LCS[i, j-1])$

$LCS[1, 3] = \max(LCS[0, 3],$

LCS[1, 2])
= max(0, 1) = 1

LCS [1, 4] ⇒ i = 1, j = 4, P_i = M, Q_j = M
$P_i = Q_j$ ⇒ LCS [i, j] = 1 + LCS[i − 1, j − 1]
LCS [1, 4] = 1 + LCS[0, 3]
= 1 + 0 = 1

Computation for row 2 :

LCS [2, 1] ⇒ i = 2, j = 1, P_i = L, Q_j = M
$P_i \neq Q_j$ ⇒ LCS [i, j] = max (LCS [i − 1 , j], LCS [i, j − 1])
LCS [2, 1] = max (LCS [1, 1], LCS [2, 0])
= max (1, 0) = 1

LCS [2, 2] ⇒ i = 2, j = 2, P_i = L, Q_j = N
$P_i \neq Q_j$ ⇒ LCS [i, j] = max (LCS [i − 1 , j], LCS [i, j − 1])
LCS [2, 2] = max (LCS [1, 2], LCS [2, 1])
= max (1, 1) = 1

LCS [2, 3] ⇒ i = 2, j = 3, P_i = L, Q_j = O
$P_i \neq Q_j$ ⇒ LCS [i, j] = max (LCS [i − 1 , j], LCS [i, j − 1])
LCS [2, 3] = max(LCS[1, 3], LCS[2, 2])
= max(1, 1) = 1

LCS [2, 4] ⇒ i = 2, j = 4, P_i = L, Q_j = M
$P_i \neq Q_j$ ⇒ LCS [i, j] = max (LCS [i − 1 , j], LCS [i, j − 1])
LCS [2, 4] = max (LCS [1, 4], LCS [2, 3])
= max (1, 1) = 1

Computation for row 3 :

LCS [3, 1] ⇒ i = 3, j = 1, P_i = N, Q_j = M
$P_i \neq Q_j$ ⇒ LCS [i, j] = max (LCS [i − 1 , j], LCS [i, j − 1])
LCS [3, 1] = max (LCS [2, 1], LCS [3, 0])
= max (1, 0) = 1

LCS [3, 2] ⇒ i = 3, j = 2, P_i = N, Q_j = N
$P_i = Q_j$ ⇒ LCS [i, j] = 1 + LCS[i − 1, j − 1]
LCS [3, 2] = 1 + LCS[2, 1] = 1 + 1 = 2

LCS [3, 3] ⇒ i = 3, j = 3, P_i = N, Q_j = O
$P_i \neq Q_j$ ⇒ LCS [i, j] = max (LCS [i − 1 , j], LCS [i, j − 1])
LCS [3, 3] = max(LCS[2, 3], LCS[3, 2]) = max(1, 2) = 2

LCS [3, 4] ⇒ i = 3, j = 4, P_i = N, Q_j = M
$P_i \neq Q_j$ ⇒ LCS [i, j] = max (LCS [i − 1 , j], LCS [i, j − 1])
LCS [3, 4] = max (LCS [2, 4], LCS [3, 3]) = max (1, 2) = 2

Computation for row 4 :

LCS [4, 1] ⇒ i = 4, j = 1, P_i = O, Q_j = M
$P_i \neq Q_j$ ⇒ LCS [i, j] = max (LCS [i − 1 , j], LCS [i, j − 1])
LCS [4, 1] = max (LCS [3, 1], LCS [4, 0]) = max (1, 0)
= 1

LCS [4, 2] ⇒ i = 4, j = 2, P_i = O, Q_j = N
$P_i \neq Q_j$ ⇒ LCS [i, j] = x (LCS [i − 1 , j], LCS [i, j − 1])
LCS [4, 2] = max (LCS [3, 2], LCS [4, 1]) = max (2, 1)
= 2

LCS [4, 3] ⇒ i = 4, j = 3, P_i = O, Q_j = O
$P_i = Q_j$ ⇒ LCS [i, j] 1 + LCS[i − 1, j − 1]
LCS [4, 3] = 1 + LCS[3, 2] = 1 + 2 = 3

LCS [4, 4] ⇒ i = 4, j = 4, P_i = O, Q_j = M
$P_i \neq Q_j$ ⇒ LCS [i, j] = max (LCS [i − 1 , j], LCS [i, j − 1])
LCS [4, 4] = max (LCS [3, 4], LCS [4, 3]) = max (2, 3)
= 3Computation for row 5 :

LCS [5, 1] ⇒ i = 5, j = 1, P_i = M, Q_j = M
$P_i = Q_j$ ⇒ LCS [i, j] = 1 + LCS[i − 1, j − 1]
LCS [5, 1] = 1 + LCS[4, 0] = 1 + 0 = 1

LCS [5, 2] ⇒ i = 5, j = 2, P_i = M, Q_j = N
$P_i \neq Q_j$ ⇒ LCS [i, j] = max (LCS [i − 1 , j], LCS [i, j − 1])
LCS [5, 2] = max (LCS [4, 2], LCS [5, 1])
= max (2, 1) = 2

LCS [5, 3] ⇒ i = 5, j = 3, P_i = M, Q_j = O
$P_i \neq Q_j$ ⇒ LCS [i, j] = max (LCS [i − 1 , j], LCS [i, j − 1])
LCS [5, 3] = max (LCS [4, 3], LCS [5, 2]) = max (3, 2)
= 3

LCS [5, 4] ⇒ i = 5, j = 4, P_i = M, Q_j = M
LCS [5, 4] ⇒ i = 5, j = 4, P_i = M, Q_j = M
$P_i = Q_j$ ⇒ LCS [i, j] = 1 + LCS[i − 1, j − 1]
LCS [5, 4] = 1 + LCS[5, 3] = 1 + 3 = 4

Longest common subsequence for given strings is of length 4. Let us find LCS of P and Q :

Step 1 : i = 5, j = 4, P_i = M, Q_j = M

$P_i = Q_i$, so add Pi to solution set. And move diagonally back.

Solution set S = { M }

Step 2 : i = 4, j = 3, P_i = O, Q_j = O

$P_i = Q_i$, so add Pi to solution set. And move diagonally back.

Solution set S = { M, O }

Step 3 : $i = 3, j = 2, P_i = N, Q_j = N$

$P_i = Q_j$, so add Pi to solution set. And move diagonally back.

Solution set S = { M, O, N }

Step 4 : $i = 2, j = 1, P_i = L, Q_j = M$

$P_i \neq Q_j$, and LCS[i, j] is derived from LCS[i – 1, j]. So move in vertical direction.

Solution set S = { M, O, N }

So finally, table would be,

Table P. 4.11.3(a)

$Q_j \longrightarrow$

			0	1	2	3	4
				M	N	O	M
0	0	0	0	0	0	0	0
1	M	0	1	1	1	1	1
2	L	0	1	1	1	1	1
3	N	0	1	2	2	2	2
4	O	0	1	2	3	3	3
5	M	0	1	2	3	4	4

P_i ↓

Step 5 : $i = 1, j = 1, P_i = M, Q_j = M$

$P_i = Q_j$, so add Pi to solution set. And move diagonally back.

Solution set S = { M, O, N, M }

Moving diagonally back i and j become zero. So stop. We have collected characters from the last position of the string. So reverse the solution set, which is the LCS of P and Q.

So, LCS = MNOM

4.12 Exam Pack

(University and Review Questions)

☞ Syllabus Topic : General Method

Q. What is Dynamic programming? Is this the optimization technique? Give reasons. What are the drawbacks of dynamic programming? *(Ans. : Refer section 4.1.1)* (5 Marks)

Q. What are the characteristics of Dynamic Programming? *(Ans. : Refer section 4.1.3)* (4 Marks)

Q. State applications of Dynamic Programming. *(Ans. : Refer section 4.1.4)* (3 Marks)

Q. State "Principle of Optimality". *(Ans. : Refer section 4.2)* (2 Marks)

Q. What is dynamic programming approach to solve the problem? *(Ans. : Refer section 4.2)* (4 Marks)

Q. What is the principle of optimality in dynamic programming ? Give a suitable example for this property of optimality in dynamic programming. *(Ans. : Refer section 4.2)* (7 Marks)

Q. Discuss the elements of dynamic programming. *(Ans. : Refer section 4.3)* (4 Marks)

Q. What are the common steps in the dynamic programming to solve any problem? *(Ans. : Refer section 4.3)*(6 Marks)

Q. Distinguish between dynamic programming and divide and conquer technique. *(Ans. : Refer section 4.4)* (6 Marks)

☞ Syllabus Topic : Multistage Graph

Q. Write a short note on Multistage graph. *(Ans. : Refer section 4.5)* (5 Marks) **(Dec. 2016, May 2017)**

Q. What is multistage graph? How to solve it using dynamic programming? *(Ans. : Refer section 4.5)*(6 Marks)

Ex. 4.5.2 (10 Marks) **(Dec. 2014)**

☞ Syllabus Topic : Single Source Shortest Path

Q. How to find single Bellman-Ford algorithm? *(Ans. : Refer section 4.6)*(4 Marks)

Q. How to solve single source shortest path problem using a Bellman-Ford algorithm? *(Ans. : Refer section 4.6)*(4 Marks)

Ex. 4.6.2 (10 Marks) **(Dec. 2015)**

Ex. 4.6.3 (10 Marks) **(May 2013)**

☞ Syllabus Topic : All Pair Shortest Path

Q. Explain all pair shortest path algorithm with a suitable example. *(Ans. : Refer section 4.7)*(10 Marks) **(May 2014)**

Q. Write Floyd's algorithm for all pairs shortest path and find time complexity. *(Ans. : Refer section 4.7)* (7 Marks)

☞ Syllabus Topic : Assembly-Line Scheduling

Q. Explain Assembly line scheduling. *(Ans. : Refer section 4.8)* (7 Marks)

☞ **Syllabus Topic : 0/1 knapsack**

Q. Explain 0/1 Knapsack Problem with an example. *(Ans. : Refer section 4.9)* (10 Marks) **(May 2014)**

Q. Explain 0/1 knapsack problem using dynamic programming. *(Ans. : Refer section 4.9)* (10 Marks) **(Dec. 2015)**

Q. Define the knapsack problem. How to solve it using dynamic programming? *(Ans. : Refer section 4.9)*(5 Marks)

Q. How to solve knapsack problem using dynamic programming? *(Ans. : Refer section 4.9.1)* (5 Marks)

☞ **Syllabus Topic : Travelling Salesman Problem**

Q. Write short on Travelling sales person problem. *(Ans. : Refer section 4.10)* (10 Marks) **(May 2015)**

Q. Define and solve Travelling Salesman Problem using dynamic programming. *(Ans. : Refer section 4.10)* (10 Marks)

Ex. 4.10.4 (Dec. 2014, 10 Marks)

☞ **Syllabus Topic : Longest Common Subsequence**

Q. Explain longest common subsequence with an example. *(Ans. : Refer section 4.11)* (10 Marks) **(May 2014)**

Q. What is a longest common subsequence problem? *(Ans. : Refer section 4.11)* (5 Marks)

Q. What is LCS problem? How to solve it using dynamic programming? Explain in detail. *(Ans. : Refer section 4.11)* (7 Marks)

Ex. 4.11.1 (10 Marks) **(May 2017)**

❑❑❑

CHAPTER 5

Greedy Algorithms

Module 3

Syllabus Topics

General Method, Single source shortest path, Knapsack problem, Job sequencing with deadlines, Minimum cost spanning trees-Kruskal and Prim's algorithm, Optimal storage on tapes.

Syllabus Topic : General Method

5.1 General Method

5.1.1 Introduction

Q. What is the greedy algorithmic approach? (5 Marks)

- When the problem has many feasible solutions with different cost or benefit, finding the best solution is known as an **optimization problem** and the best solution is known as the **optimal solution**.
- In real life scenario, there exist uncountable optimization problems such as make a change, knapsack, shortest path, job sequencing and so on.
- Like dynamic programming and many other techniques, greedy algorithms are also used to solve optimization problems.
- The beauty of greedy algorithms is that they are very simple in nature. They are easy to understand and quick to build. Perhaps greedy algorithms are the least complex among all optimization techniques.
- Greedy algorithm derives solution step by step, by looking at the information available at the current moment. It does not look at future prospects. Decisions are completely locally optimal. This method constructs the solution simply by looking at current benefit without exploring future possibilities and hence they are known as greedy.
- The choice made under greedy solution procedure are irrevocable, means once we have selected the local best solution, it cannot be backtracked.
- Thus, a choice made at each step in the greedy method should be:
- **Feasible** : Choice should satisfy problem constraints.
- **Locally optimal** : Best solution from all feasible solution at current stage should be selected.
- **Irrevocable** : Once the choice is made, it cannot be altered, i.e. if a feasible solution is selected (or rejected) in step i, it cannot be rejected (or selected) in subsequent stages.

5.1.2 Control Abstraction

→ (May 15)

Q. Write an abstract algorithm for greedy design method. MU - May 2015, 5 Marks

Q. Explain the control abstraction of the greedy method. (3 Marks)

Greedy algorithm derives solution step by step, by looking at the information available at the current moment. It does not look at future prospects. Decisions are completely locally optimal. The choice made under greedy solution procedure are irrevocable, means once we have selected the local best solution, it cannot be backtracked.

Control abstraction for the greedy approach is described below :

```
Algorithm GREEDY_APPROACH(L, n)
// Description : Solve the given problem using greedy approach
// Input : L: List of possible choices, n: size of solution
// Output : Set Solution containing solution of given problem

Solution ← ∅
for i ← 1 to n do
    Choice ← Select(L)

if (feasible(Choice ∪ Solution)) then
    Solution ← Choice ∪ Solution
    end
end
return Solution
```

Select locally optimal, irrevocable choice from L

Check if selected choice is feasible or not

Add feasible choice to partial solution

The greedy approach does not ensure the optimal solution, but in most of the cases, it provides a good approximation. For certain problems, greedy always produces an optimal solution (like Prim's and Kruskal's algorithm). If greedy provides the optimal solution, then it is the best among all other techniques.

5.1.3 Characteristics

→ (May 14)

Q. Comment on the module of computation : Greedy Method. **MU - May 2014, 5 Marks**

Q. Enlist and explain the characteristics of greedy algorithms. **(4 Marks)**

Problems which can be solved using the greedy method generally possesses following two interesting properties :

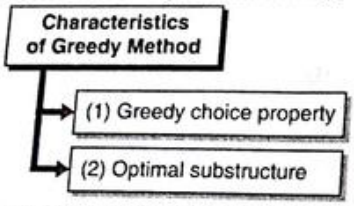


Fig. C5.1 : Characteristics of Greedy Method

→ **(1) Greedy choice property**

- The global optimal solution is derived by making locally optimal choices, i.e. the choice which looks best at moment.
- If the choice is feasible, then add it to the solution set and reduce the problem size by the same amount.
- The current choice may depend on previously made choices but it does not depend on future choice.
- The greedy choice is irrevocable, so it does not reconsider any choice.
- Greedy choice property helps to derive optimal solution by reducing the problem size by solving local optimal subproblems.

→ **(2) Optimal substructure**

- We say given problem exhibits optimal substructure if the optimal solution to given problem contains the optimal solution to its subproblems too. In the problem which possesses the optimal substructure, best next choice always leads to an optimal solution.

☞ **Cases of failure**

- In many cases, the greedy algorithm fails to generate the best solution. Sometimes it may even create the worst solution.

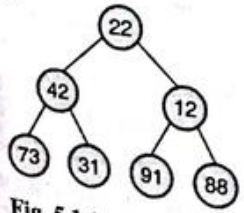


Fig. 5.1.1 : Binary tree

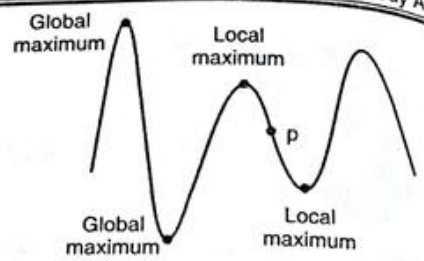


Fig. 5.1.2 : Cost curve

- Suppose we want to find maximum number from a binary tree in Fig. 5.1.1. We start from the root, as we are greedy to find the maximum element, at each level we compare local maximum element with its two children and will take a decision based on that level only, without exploring future branches.
- At level 1, the algorithm selects the branch having a maximum value, so it selects the left branch as it has value 42 which is higher than right branch value 12.
- Hence, the algorithm terminates with 73, instead of finding 91 as best solutions. Greedy approach fails to find a maximum element of this case.
- Same way, in some search space, suppose we are at some point p as shown in Fig. 5.1.2, if the problem is of maximization or minimization, in no case greedy can generate an optimal solution.
- Greedy can achieve only local maximum or local minimum solution in discussed case.

5.1.4 Applications of Greedy Approach

Greedy algorithms are used to find an optimal or near optimal solution to many real-life problems. Few of them are listed below:

(1) Make a change problem
(2) Knapsack problem
(3) Minimum spanning tree
(4) Single source shortest path
(5) Activity selection problem
(6) Job sequencing problem
(7) Huffman code generation.

5.1.5 Divide and Conquer vs. Greedy Algorithms

Sr. No.	Divide and conquer	Greedy Algorithm
1.	Divide and conquer is used to find the solution, it does not aim for the optimal solution.	A greedy algorithm is optimization technique. It tries to find an optimal solution from the set of feasible solutions.
2.	DC approach divides the problem into small subproblems, each	In greedy approach, the optimal solution is obtained from a set of

Sr. No.	Divide and conquer	Greedy Algorithm
	subproblem is solved independently and solutions of the smaller problems are combined to find the solution to the large problem.	feasible solutions.
3.	Subproblems are independent, so DC might solve same subproblem multiple time.	Greedy algorithm does not consider the previously solved instance again, thus it avoids the re-computation.
4.	DC approach is recursive in nature, so it is slower and inefficient.	Greedy algorithms are iterative in nature and hence faster.
5.	Example : Merge sort, Quick sort, Binary search.	Example : Knapsack problem, Huffman codes, Minimum spanning tree.

5.1.6 Dynamic Programming Vs Greedy Approach

→ (May 13)

Q. Write a short note : Differentiate between greedy approach and dynamic programming.
MU - May 2013, 5 Marks

Q. Compare dynamic programming with the greedy approach. **(5 Marks)**

Sr. No.	Dynamic Programming	Greedy Approach
1.	It guarantees an optimal solution.	It does not guarantee an optimal solution.
2.	Subproblems overlap.	Subproblems do not overlap.
3.	It does more work.	It does little work.
4.	Considers the future choices.	Only considers the current choices.
5.	There is no specialized set of feasible choices.	Construct the solution from the set of feasible solutions.
6.	Select choice which is globally optimum.	Select choice which is locally optimum.
7.	Employ memorization.	There is no concept of memorization.

Syllabus Topic : Single Source Shortest Path

5.2 Single Source Shortest Path

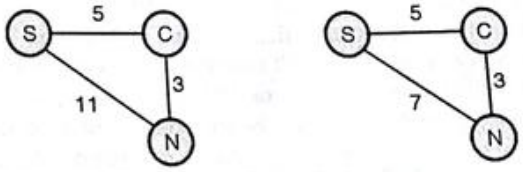
Q. Write an algorithm to compute the shortest distance between the source and destination vertices of a connected graph. Will your algorithm work for negative weights? **(6 Marks)**

The graph is widely accepted data structure to represent distance map. The distance between cities effectively represented using graph.

- Dijkstra proposed an efficient way to find single source shortest path from the weighted graph. For given source vertex *s*, the algorithm finds the shortest path to every other vertex *v* in the graph.
- **Assumption** : Weight of all edges is non-negative
- Steps of the Dijkstra's algorithm are explained here:

1. Initializes the distance of source vertex to zero and remaining all other vertices to infinity.
2. Set source node to current node and put remaining all nodes in the list of unvisited vertex list. Compute the tentative distance of all immediate neighbour vertex of the current node.
3. If the newly computed value is smaller than the old value, then update it.

- For example, C is the current node, whose distance from source S is dist (S, C) = 5. Consider N is the neighbour of C and weight of edge (C, N) is 3. So the distance of N from source via C would be 8. If the distance of N from source was already computed and if it is greater than 8 then relax edge (S, N) and update it to 8, otherwise don't update it.



d(S, N) = 11 d(S, N) = 7

d(S, C) + d(C, N) < d(S, N) d(S, C) + d(C, N) > d(S, N)

⇒ Relax edge (S, N) ⇒ Don't update d(S, N)

Update d(S, N) = 8

Fig. 5.2.1 : Weight updating in Dijkstra's algorithm

1. When all the neighbours of a current node are explored, mark it as visited. Remove it from unvisited vertex list. Mark the vertex from unvisited vertex list with minimum distance and repeat the procedure.
2. Stop when the destination node is tested or when unvisited vertex list becomes empty.

Dijkstra's shortest path algorithm is described below :

```
Algorithm DIJAKSTRA_SHORTEST_PATH(G, s, t)
// s is the source vertex
// t is the target vertex
// π[u] stores the parent / previous node of u
// V is the set of vertices in graph G
dist[s] ← 0                          Initialize distance of source node
π[s] ← NIL.                          Stores parent of node s
for each vertex v ∈ V do             Stores parent of node s
    if v ≠ s then
    dist[v] ←∞                       Initialize distance and
    π [v] ← undefined                parent of remaining nodes
    end
    ENQUEUE(v, Q)                    // insert v to queue Q
end
while Q is not empty do
    u ← vertex in Q having minimum dist[u]
    if u == t then                   Path from source s to target t
    break                            is constructed
    end
    DEQUEUE(u, Q)                    // Remove u from queue Q
    for each adjacent node v of u do
    val ← dist[u] + weight(u, v)
    if val < dist[v] then            Update node distance
    dist[v] ← val
    π[v] ← u
    end
    end
end
```

Complexity analysis

First for loop does initialization in O(|V|) time. As there are |V| nodes in the graph, size of queue Q would be V, and hence while loop iterates |V| times in worst case. For loop inside while loop run maximum |V| time, because a node can have maximum |V| - 1 neighbours. The worst case upper bound running time of this algorithm is described as $O(|V^2|)$.

Note : Dijkstra's algorithm cannot handle negative weight

Ex. 5.2.1

Suppose Dijkstra's algorithm is run on the following graph, starting at node A,

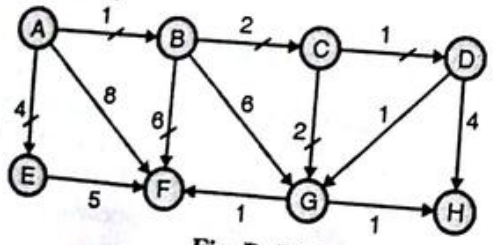


Fig. P. 5.2.1

(i) Draw a table showing the intermediate distance values of all the nodes at each iteration of the algorithm.

(ii) Show the final shortest path tree.

Soln. :

Here, source vertex is A.

dist[u] indicates distance of vertex u from source

π[u] indicates parent / previous node of u

Initialization

dist[source] = 0 ⇒ dist[A] = 0

π[source] = undefined ⇒ π [A] = NIL

dist[B] = dist[C] = dist[D] = dist[E] = dist [F] = dist[G]= dist[H] = ∞

π[B] = π[C] = π[D] = π[E] = π[F] = π[G] = π[H]= NIL

Q =

Vertex u	A	B	C	D	E	F	G	H
dist[u]	0	∞	∞	∞	∞	∞	∞	∞
π[u]	NIL	NIL	NIL	NIL	NIL	NIL	NIL	NIL

Iteration 1

u = unprocessed vertex in Q having minimum dist[u] = A

Adjacent[A] = {B, E, F}

val[B] = dist[A] + weight(A, B)
= 0 + 1
= 1

Here, val[B] < dist[B], so update dist[B]

∴dist[B] = 1, and π[B] = A

val[E] = dist[A] + weight(A, E)
= 0 + 4
= 4

Here, val[E] < dist[E], so update dist[E]

∴dist[E] = 4 and π[6] = A

val[F] = dist[A] + weight(A, F)
= 0 + 8
= 8

Here, val[F] < dist[F], so update dist[F]

∴dist[F] = 8 and π[F] = A

Q =

Vertex u	A	B	C	D	E	F	G	H
dist[u]	0	1	∞	∞	4	8	∞	∞
π[u]	NIL	A	NIL	NIL	A	A	NIL	NIL

Shaded cells are processed vertexIteration 2

u = unprocessed vertex in Q having minimum dist[u] = B

Adjacent[B] = {C, F, G}

val[C] = dist[B] + weight(B, C)
= 1 + 2
= 3

Here, val[C] < dist[C], so update dist[C]

∴dist[C] = 3 and π[C] = B

val[F] = dist[B] + weight(B, F)
= 1 + 6 = 7

Here, val[F] < dist[F], so update dist[F]
∴dist[F] = 7 and π[F] = B
val[G] = dist[B] + weight(B, G)
= 1 + 6
= 7

Here, val[G] < dist[G], so update dist[G]
∴dist[G] = 7 and π[G] = B

Q =

Vertex u	A	B	C	D	E	F	G	H
dist[u]	0	1	3	∞	4	7	7	∞
π[u]	NIL	A	B	NIL	A	B	B	NIL

Iteration 3

u = unprocessed vertex in Q having minimum dist[u] = C
Adjacent[C] = {D, G}
val[D] = dist[C] + weight(C, D)
= 3 + 1 = 4

Here, val[D] < dist[D], so update dist[D]
∴dist[D] = 4 and π[D] = C
val[G] = dist[C] + weight(C, G)
= 3 + 2 = 5

Here, val[G] < dist[G], so update dist[G]
∴dist[G] = 5 and π[G] = C

Q =

Vertex u	A	B	C	D	E	F	G	H
dist[u]	0	1	3	4	4	7	5	∞
π[u]	NIL	A	B	C	A	B	C	NIL

Iteration 4

u = unprocessed vertex in Q having minimum dist[u] = E
Adjacent[E] = {F}
val[F] = dist[E] + weight(E, F)
= 4 + 5
= 9

Here, val[F] > dist[F], so no change in table

Q =

Vertex u	A	B	C	D	E	F	G	H
dist[u]	0	1	3	4	4	7	5	∞
π[u]	NIL	A	B	C	A	B	C	NIL

Iteration 5

u = unprocessed vertex in Q having minimum dist[u] = D
Adjacent[D] = {G, H}
val[G] = dist[D] + weight(D, G)
= 4 + 1
= 5

Here, val[G] = dist[G], so don't update dist[G]
val[H] = dist[D] + weight(D, H)
= 4 + 4
= 8

Here, val[H] < dist[H], so update dist[H]
∴dist[H] = 8 and π[H] = D

Q =

Vertex u	A	B	C	D	E	F	G	H
dist[u]	0	1	3	4	4	7	5	8
π[u]	NIL	A	B	C	A	B	D	D

Iteration 6

u = unprocessed vertex in Q having minimum dist[u] = G
Adjacent[G] = {F, H}
val[F] = dist[G] + weight(G, F)
= 5 + 1
= 6

Here, val[F] < dist[F], so update dist[F]
∴dist[F] = 6 and π[F] = G
val[H] = dist[G] + weight(G, H)
= 5 + 1
= 6

Here, val[H] < dist[H], so update dist[H]
∴dist[H] = 6 and π[H] = G

Q =

Vertex u	A	B	C	D	E	F	G	H
dist[u]	0	1	3	4	4	6	5	6
π[u]	NIL	A	B	C	A	G	C	G

Iteration 7

u = unprocessed vertex in Q having minimum dist[u] = F
Adjacent[F] = { }
So, no change in table

Q =

Vertex u	A	B	C	D	E	F	G	H
dist[u]	0	1	3	4	4	6	5	6
π[u]	NIL	A	B	C	A	G	C	G

Iteration 8

u = unprocessed vertex in Q having minimum dist[u] = H
Adjacent[H] = { }
So, no change in table

Q =

Vertex u	A	B	C	D	E	F	G	H
dist[u]	0	1	3	4	4	6	5	6
π[u]	NIL	A	B	C	A	G	C	G

We can easily derive the shortest path tree for given graph from above table. In the table, π[u] indicates the parent node of vertex u. The shortest path tree is shown in Fig. P. 5.2.1(a).

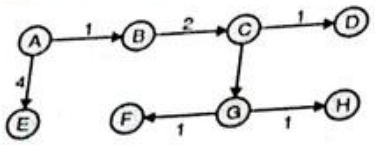


Fig. P. 5.2.1(a)

Ex. 5.2.2

Find minimum distance path from vertex 1 to 7.

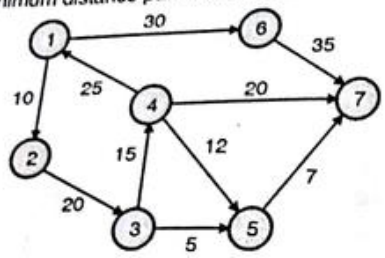


Fig. P. 5.2.2

Soln. : Here, source vertex is 1 and destination vertex is 7.

Initialization

$dist[source] = 0 \Rightarrow dist[1] = 0$

$prev[source] = undefined \Rightarrow prev[1] = NIL$

$dist[2] = dist[3] = dist[4] = dist[5] = dist[6] = dist[7] = \infty$

$prev[2] = prev[3] = prev[4] = prev[5] = prev[6] = prev[7] = NIL$

Q =

Vertex u	1	2	3	4	5	6	7
dist[u]	0	∞	∞	∞	∞	∞	∞
prev[u]	NIL	NIL	NIL	NIL	NIL	NIL	NIL

Iteration 1

u = unprocessed vertex in Q having minimum dist[u] = 1

Adjacent[u] = {2, 6}

val[2] = dist[1] + weight(1, 2)
= 0 + 10
= 10

Here, val[2] < dist[2], so update dist[2]

∴dist[2] = 10, and prev[2] = 1

val[6] = dist[1] + weight(1, 6)
= 0 + 30
= 30

Here, val[6] < dist[6], so update dist[6]

∴dist[6] = 30 and prev[6] = 1

Q =

Vertex u	1	2	6	3	4	5	7
dist[u]	0	10	30	∞	∞	∞	∞
prev[u]	NIL	1	1	NIL	NIL	NIL	NIL

Shaded cells are processed vertex

Iteration 2

u = unprocessed vertex in Q having minimum dist[u] = 2

Adjacent[u] = {3}

val[3] = dist[2] + weight(2, 3)
= 10 + 20
= 30

Here, val[3] < dist[3], so update dist[3]

∴dist[3] = 30 and prev[3] = 2

Q =

Vertex u	1	2	3	6	4	5	7
dist[u]	0	10	30	30	∞	∞	∞
prev[u]	NIL	1	2	1	NIL	NIL	NIL

Iteration 3

u = unprocessed vertex in Q having minimum dist[u] = 3

Adjacent[u] = {4, 5}

val[4] = dist[3] + weight(3, 4)
= 30 + 15 = 45

Here, val[4] < dist[4], so update dist[4]

∴dist[4] = 45 and prev[4] = 3

val[5] = dist[3] + weight(3, 5)
= 30 + 5 = 35

Here, val[5] < dist[5], so update dist[5]

∴dist[5] = 35 and prev[5] = 3

Q =

Vertex u	1	2	3	6	5	4	7
dist[u]	0	10	30	30	35	45	∞
prev[u]	NIL	1	2	1	3	3	NIL

Iteration 4

u = unprocessed vertex in Q having minimum dist[u] = 6

Adjacent[u] = {7}

val[7] = dist[6] + weight(6, 7)
= 30 + 35 = 65

Here, val[7] < dist[7], so update dist[7]

∴dist[7] = 65 and prev[7] = 6

Q =

Vertex u	1	2	3	6	5	4	7
dist[u]	0	10	30	30	35	45	65
prev[u]	NIL	1	2	1	3	3	6

Iteration 5

u = unprocessed vertex in Q having minimum dist[u] = 5

Adjacent[u] = {7}

val[7] = dist[5] + weight(5, 7)
= 35 + 7
= 42

Here, val[7] < dist[7], so update dist[7]
∴ dist[7] = 42 and prev[7] = 5

Q =

Vertex u	1	2	3	6	5	7	4
dist[u]	0	10	30	30	35	42	45
prev[u]	NIL	1	2	1	3	5	3

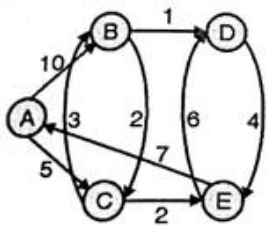
Iteration 6

u = unprocessed vertex in Q having a minimum dist[u] = 7
Vertex 7 is the target vertex, so stop. Let us trace the path.
Vertex 1 is the source vertex. From above table,
Vertex 1 is predecessor of vertex 2, path: 1 – 2
Vertex 2 is predecessor of vertex 3, path: 1 – 2 – 3
Vertex 3 is predecessor of vertex 5, path: 1 – 2 – 3 – 5
Vertex 5 is predecessor of vertex 7, path: 1 – 2 – 3 – 5 – 7

Hence, the shortest path is 1 – 2 – 3 – 5 – 7 and the path has cost 42.

Ex. 5.2.3

Find the shortest path from the source vertex A using Dijkstra's algorithm.



Soln. :

Visited vertices ↓	dist[u] B	C	D	E	Comment
A	10	5	∞	∞	List distance of vertex from A. select the vertex having minimum distance, i.e. C
A – C	min{10, 5 + 3} = 8	5	∞	min {∞, 5 + 2} = 7	Visit remaining vertices from A via C, update distance and go to vertex having minimum distance from C, i.e. E
A – C – E	min{8, 5 + 2 + ∞} = 8	5	Min{∞, 5+2+6} = 13	7	Visit remaining vertices from A via C and E, update distance and go to vertex having minimum distance from E, i.e. B
A – C – E – B	8	5	min{13, 5 + 3 + 1} = 9	7	Visit remaining vertices from A via C, E, B, update distance and go to vertex having minimum distance from B, i.e. D

Ex. 5.2.4 **MU - May 2014, 10 Marks**

To find Dijkstra's shortest path from vertex 1 to vertex 4 for the following graph.

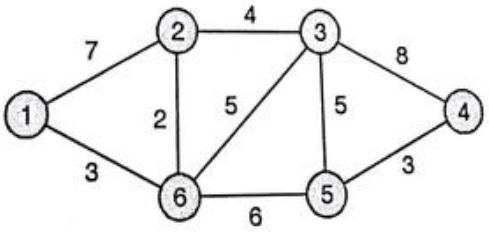


Fig. P. 5.2.4

Soln. :

Visited vertices ↓	dist[u] 2	3	4	5	6	Comment
1	7	∞	∞	∞	3	List distance of vertices from 1. select the vertex having minimum distance, i.e. 3

Visited vertices ↓	dist[u]					Comment
	2	3	4	5	6	
1 – 6	min{7, 3 + 2} = 5	min{∞, 3 + 5} = 8	∞	min {∞, 3 + 6 } = 9	3	Visit remaining vertices from 1 via 6, update distance and go to vertex having minimum distance from 6, i.e. 2
1 – 6 – 2	5	min{8, 5 + 4} = 8	∞	min {9, 5 + ∞ } = 9	3	Visit remaining vertices from 1 via 6 and 2, update distance and go to vertex having minimum distance from E, i.e. 3
1 – 6 – 2 – 3	5	8	min{∞, 8 + 8} = 16	min {9, 8 + 5} = 9	3	Visit remaining vertices from 1 via 6, 2, 3, update distance and go to vertex having minimum distance from 3, i.e. 5
1 – 6 – 2 – 3 – 5	5	8	= min {16, 9 + 3} = 12	9	3	Visit remaining vertices from 1 via 6, 2, 3, 4 update distance and go to vertex having minimum distance from 5, i.e. 4

Shortest path for the given graph is : 1 – 6 – 2 – 3 – 5

Ex. 5.2.5 **MU - May 2016, 10 Marks**

Find the shortest path from source vertex A using Dijkstra's algorithm

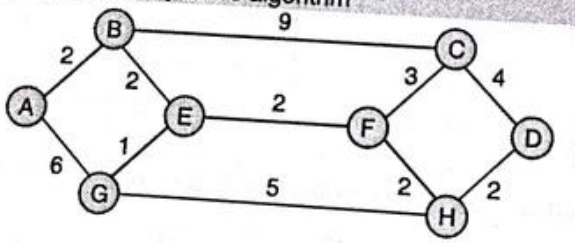


Fig. P. 5.2.5

Soln. :

Visited vertices ↓	dist[u]							Comment
	B	C	D	E	F	G	H	
A	2	∞	∞	∞	∞	6	∞	List distance of vertices from A. select the vertex having minimum distance, i.e. B
B	2	min{∞, 2 + 9} = 11	∞	min {∞, 2 + 2 } = 4	∞	min {6, 2 + ∞ } = 6	∞	Visit remaining vertices from A via B, update distance and go to vertex having minimum distance from B, i.e. E

Visited		dist[u]						Comment
A – B – E	2	min{11, 4 + ∞} = 11	∞	4	min{∞, 4 + 2} = 6	min {6, 4 + 1 } = 5	∞	Visit remaining vertices from A via B and E, update distance and go to vertex having minimum distance from E, i.e. G
A – B – E – G	2	min{11, ∞} = 11	∞	4	Min {6, 5 + ∞} = 6	5	Min {∞, 5 + 5} = 10	Visit remaining vertices from A via B, E and G, update distance and go to vertex having minimum distance from E, i.e. F
A – B – E – G – F	2	min{11, 6 + 3} = 9	∞	4	6	5	min{10, 6 + 2} = 8	Visit remaining vertices from A via B, E, G and F, update distance and go to vertex having minimum distance from F, i.e. H
A – B – E – G – F – H	2	min{9, 8 + ∞} = 9	min{∞, 8 + 2} = 10	4	6	5	8	Visit remaining vertices from A via B, E, G, F and H, update distance and go to vertex having minimum distance from H, i.e. C
A – B – E – G – F – H – C	2	9	min{10, 9 + 4} = 13	4	6	5	8	Visit remaining vertices from A via B, E, G, F, H and C, update distance and go to vertex having minimum distance from C i.e. D

- Bold edges indicate the minimum cost path in the following figure.

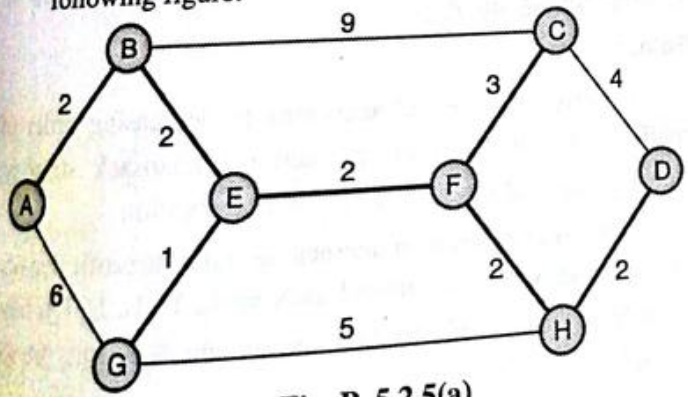


Fig. P. 5.2.5(a)

Syllabus Topic : Knapsack Problem

5.3 Knapsack Problem

→ (May 17)

Q. What is 0/1 Knapsack and fractional knapsack problem. MU - May 2017, 3 Marks

Q. State and solve knapsack problem using the greedy method. (6 Marks)

Problem

Given a set of items having some weight and value/profit associated with it. The knapsack problem is to find the set of items such that the total weight is less than or equal to a given limit (size of knapsack) and the total value/profit earned is as large as possible.

Knapsack problem has two variants.

- **Binary or 0/1 knapsack :** Item cannot be broken down into parts.
- **Fractional knapsack :** Item can be divided into parts.

5.3.1 Fractional Knapsack Problem

Q. Explain and write the algorithm for 0/1 Knapsack using greedy approach. (6 Marks)

Problem

Let $X = \langle x_1, x_2, x_3, \ldots, x_n \rangle$ be the set of n items, $W = \langle w_1, w_2, w_3, \ldots, w_n \rangle$ and $V = \langle v_1, v_2, v_3, \ldots, v_n \rangle$ be the set of weight and value associated with each item in X, respectively. Let M is the capacity of the knapsack, i.e. knapsack cannot hold items having collective weight more than M. Knapsack problem is the problem of filling the knapsack using items in X such that it maximizes the profit and collective weight of added items should not cross the knapsack capacity.

In fractional knapsack, breaking of the item is allowed. If items are selected in decreasing order of value to weight ratio, fractional knapsack guarantees the optimal solution. With fractional knapsack, knapsack would be full at the end of the algorithm.

Binary knapsack does not guarantee an optimal solution, and knapsack may not be full when algorithm halts.

Knapsack problem can be formulated as,

$$\text{Maximize} \sum_{i=1}^{n} v_i x_i \text{ subjected to } \sum_{i=1}^{n} w_i x_i \leq M$$

$x_i \in (0, 1)$ for binary knapsack

$x_i \in [0, 1]$ for fractional knapsack

Algorithm for fractional knapsack is described below :

```
Algorithm GREEDY_FRACTIONAL_KNAPSACK(X, V, W, M)
// Description : Solve the knapsack problem using greedy approach
// Input : X: An array of n items
           V: An array of profit associated with each item
           W: An array of weight associated with each item
           M: Capacity of knapsack
// Output: SW: Weight of selected items
           SP: Profit of selected items
// Items are presorted in decreasing order of pi = vi / wi ratio

S ← ∅        // Set of selected items, initially empty
SW ← 0       // weight of selected items
SP ← 0       // profit of selected items
i ← 1
while i ≤ n do
if (SW + w[i]) ≤ M then
    S ← S ∪ X[i]                    Addition of item does not cross the knapsack capacity, so add whole item
    SW ← SW + W[i]
    SP ← SP + V[i]                  Whole item is added to solution set
else                                Compute the fraction to be added
    frac ← (M - SW) / W[i]
    S ← S ∪ X[i] * frac// Add fraction of item X[i]
    SP ← SP + V[i] * frac    // Add fraction of profit
    SW ← SW + W[i] * frac  // Add fraction of weight
end
    i ← i + 1                       Select next item
end
```

Complexity analysis

- For one item there are two choices, either to select or reject. For 2 items we have four choices:
  - Select both items
  - Reject both items
  - Select first and reject second
  - Reject first and select second
- In general, for n items, knapsack has 2^n choices. So brute force approach runs in $O(2^n)$ time.
- We can improve performance by sorting items in advance. Using merge sort or heap sort, n items can be sorted in $O(n\log_2 n)$ time. Merge sort and heap sort are non-adaptive and their running time is same in best, average and worst case.
- To select the items, we need one scan to this sorted list, which will take O(n) time.
- So total time required is $T(n) = O(n\log_2 n) + O(n)$
  $= O(n\log_2 n)$.

Ex. 5.3.1

Find an optimal solution for the following knapsack instance using the greedy method. Number of objects n = 5, capacity of knapsack M = 100, profit V = (10, 20, 30, 40, 50), weights W = (20, 30, 66, 40, 60).

Soln. :

If items are selected according to decreasing order of profit to weight ratio using fractional knapsack strategy, then algorithm always finds the optimal solution.

First arrange items in decreasing order of profit density. Assume that items are labeled as $X = (I_1, I_2, I_3, I_4, I_5)$, have profit V = {10, 20, 30, 40, 50} and weight W = {20, 30, 66, 40, 60}.

Item	Value (v_i)	Weight (w_i)	$p_i = v_i / w_i$
I_4	40	40	1.00
I_5	50	60	0.83
I_2	20	30	0.67
I_1	10	20	0.50
I_3	30	66	0.46

We shall select one by one item from above table. If the inclusion of an item does not cross the knapsack capacity, then add it. Otherwise, break the current item and select only the portion of item equivalent to remaining knapsack capacity. Select the profit accordingly. We should stop when knapsack is full or all items are scanned.

Initialize, Weight of selected items, SW = 0,

Profit of selected items, SP = 0,

Set of selected items, S = { },

Here, Knapsack capacity M = 100.

Iteration 1 : SW= (SW + w_4) = 0 + 40 = 40

SW ≤ M, so select I_4

S = { I_4 }, SW = 40, SP = 0 + 40 = 40

Iteration 2 : SW = (SW + w_1) = 40 + 60 = 100

SW ≤ M, so select I_1

S = { I_4, I_1 }, SW = 100, SP = 40 + 50 = 90

Knapsack is full, no more items can be added to knapsack. Fractional Greedy algorithm selects items { I_1, I_4}, and it gives profit of 90 units, which is optimal.

Ex. 5.3.2

Consider the following instances of the fractional knapsack problem: n = 3, M = 20, V = (24, 25, 15) and W = (18, 15, 20) find the feasible solutions.

Soln. :

Let us arrange items by decreasing order of profit density. Assume that items are labeled as X = (I_1, I_2, I_3), have profit V = {24, 25, 15} and weight W = {18, 15, 20}.

Item (x_i)	Value (v_i)	Weight (w_i)	$p_i = v_i / w_i$
I_2	25	15	1.67
I_1	24	18	1.33
I_3	15	20	0.75

We shall select one by one item from above table. If the inclusion of an item does not cross the knapsack capacity, then add it. Otherwise, break the current item and select only the portion of item equivalent to remaining knapsack capacity. Select the profit accordingly. We should stop when knapsack is full or all items are scanned.

Initialize, Weight of selected items, SW = 0,

Profit of selected items, SP = 0,

Set of selected items, S = { },

Here, Knapsack capacity M = 20.

Iteration 1 : SW= (SW + w_2) = 0 + 15 = 15

SW ≤ M, so select I_2

S = { I_2 }, SW = 15, SP = 0 + 25 = 25

Iteration 2 : SW + w_1 > M, so break down item I_1.

The remaining capacity of the knapsack is 5 unit, so select only 5 units of item I_1.

frac = (M - SW) / W[i] = (20 - 15) / 18 = 5 / 18

S = { I_2, I_1 * 5/18 }

SP = SP + v_1 * *frac* = 25 + (24 * (5/18))

= 25 + 6.67 = 31.67

SW = SW + w_1 * *frac* = 15 + (18 * (5/18))

= 15 + 5 = 20

Knapsack is full. Fractional Greedy algorithm selects items { I_2, I_1 * 5/18 }, and it gives profit of 31.67 units.

Ex. 5.3.3 **MU - May 2017, 7 Marks**

Solve following using 0/1 knapsack method.

Item (i)	Value (vi)	Weight (wi)
1	18	3
2	25	5
3	27	4
4	10	3
5	15	6

Knapsack capacity M = 12.

Soln. :

If items are selected according to decreasing order of profit to weight ratio using fractional knapsack strategy, then algorithm always finds the optimal solution. First arrange items in decreasing order of profit density. Items are labeled as X = (3, 1, 2, 4, 5), have profit V = {27, 18, 25, 10, 15} and weight W = {4, 3, 5, 3, 6}.

Item	Value (v_i)	Weight (w_i)	$p_i = v_i / w_i$
3	27	4	6.75
1	18	3	6
2	25	5	5
4	10	3	3.33
5	15	6	2.5

We shall select one by one item from above table. If the inclusion of an item does not cross the knapsack capacity, then add it. Otherwise, break the current item and select only the portion of item equivalent to remaining knapsack capacity. Select the profit accordingly. We should stop when knapsack is full or all items are scanned.

Initialize, Weight of selected items, SW = 0,

Profit of selected items, SP = 0,

Set of selected items, S = { }.

Here, Knapsack capacity M = 12.

Iteration 1 : *SW= (SW + w_3) = 0 + 4 = 4*

SW ≤ M, so select item 3

S = { 3 }, SW = 4, SP = 0 + 27 = 27

Iteration 2 : *SW = (SW + w_1) = 4 + 3 = 7*

SW ≤ M, so select item 1

S = { 3, 1 }, SW = 7, SP = 27 + 18 = 45

Iteration 3: *SW = (SW + w_2) = 7 + 5 = 12*

SW ≤ M, so select item 2

S = { 3, 1, 2 }, SW = 12, SP = 45 + 25 = 70

- Knapsack is full, no more items can be added to knapsack. Selected items are {3, 1, 2}, which gives the profit of 70.

Syllabus Topic : Job Sequencing with Deadlines

5.4 Job Sequencing

→ **(May 14, Dec. 16)**

Q. *Write a note on : Job sequencing with deadlines.*

MU - May 2014, Dec. 2016, 10 Marks

Q. *State and solve job sequencing problem using greedy approach.* **(6 Marks)**

☞ **Problem**

Schedule jobs out of a set of N jobs on a single processor which maximizes profit as much as possible.

☞ **Explanation**

- Consider N jobs, each taking unit time for execution. Each job is having some profit and deadline associated with it. Profit earned only if the job is completed on or before its deadline. Otherwise, we have to pay profit as a penalty. Each job has deadline $d_i \geq 1$ and profit $p_i \geq 0$. At a time, only one job can be active on the processor.
- The job is feasible only if it can be finished on or before its deadline. A feasible solution is a subset of N jobs such that each job can be completed on or before its deadline. An optimal solution is a solution with maximum profit.
- The simple and inefficient solution is to generate all subsets of given set of jobs and find the feasible set that maximizes the profit. For N jobs, there exist 2^N schedules, so this brute force approach runs in $O(2^N)$ time.
- However, greedy approach produces an optimal result in fairly less time. As each job takes the same amount of time, we can think of the schedule S consisting of a sequence of job slots 1, 2, 3, …, N, where S(t) indicates job scheduled in slot t. Slot t has a span of (t – 1) to t. S(t) = 0 implies no job is scheduled in slot t.
- Schedule S is an array of slots S(t), S(t) ∈ {1, 2, 3, …, N} for each t ∈ {1, 2, 3, …, N}
- Schedule S is *feasible* if S(t) = i, then t ≤ d_i (Scheduled job must meet its deadline)
- Our goal is to find feasible schedule S which maximizes the profit of scheduled job.
- The goal can be achieved as follow: Sort all jobs in decreasing order of profit. Start with the empty schedule, select one job at a time and if it is feasible then schedule it in the *latest possible slot*.
- Algorithm for job scheduling is described below:

```
Algorithm JOB_SCHEDULING( J, D, P )
// Description: Schedule the jobs using greedy approach which
      maximizes the profit
// Input:  J: Array of N jobs
      D: Array of deadline for each job
      P: Array of profit associated with each job
Sort all jobs in J in decreasing order of profit
                                        If job does not miss
                                        its deadline
S ← ∅   // S is set of scheduled jobs, initially it is empty
SP ← 0  // Sum is the profit earned
for i ← 1 to N do          Add job to solution set
    if Job J[i] is feasible then
    Schedule the job in latest possible free slot meeting its
    deadline.
    S ← S ∪ J[i]
    SP ← SP + P[i]
    end
end                                  Add respective profit
```

☞ **Complexity analysis**

Simple greedy algorithm spends most of the time looking for latest slot a job can use. On average, N jobs search N/2 slots. This would take $O(N^2)$ time.

However, with the use of set data structure (find and union), the algorithm runs nearly in O(N) time.

Ex. 5.4.1 **MU - Dec. 2015, May 2017, 10 Marks**

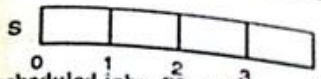
Let n = 4, (p_1, p_2, p_3, p_4) = (100, 10, 15, 27) and (d_1, d_2, d_3, d_4) = (2, 1, 2, 1). Find feasible solutions, using job sequencing with deadlines.

Soln. :

Sort all jobs in descending order of profit.

So, P = (100, 27, 15, 10), J = (J_1, J_4, J_3, J_2) and D = (2, 1, 2, 1). We shall select one by one job from the list of sorted jobs, and check if it satisfies the deadline. If so, schedule the job in the latest free slot. If no such slot is found, skip the current job and process the next one.

Initially,

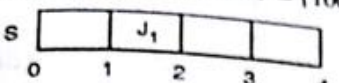


Profit of scheduled jobs, SP = 0

Iteration 1

Deadline for job J_1 is 2. Slot 2 (t = 1 to t = 2) is free, so schedule it in slot 2.

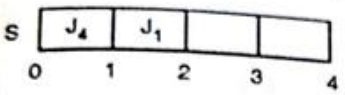
Solution set S = {J_1}, and Profit SP = {100}



Iteration 2

Deadline for job J_4 is 1. Slot 1 (t = 0 to t = 1) is free, so schedule it in slot 1.

Solution set S = {J_1, J_4}, and Profit SP = {100, 27}



Iteration 3

Job J_3 is not feasible because first two slots are already occupied and if we schedule J_3 any time later t = 2, it cannot be finished before its deadline 2. So job J_3 is discarded,

Solution set S = {J_1, J_4}, and Profit SP = {100, 27}

Iteration 4

Job J_2 is not feasible because first two slots are already occupied and if we schedule J_2 any time later t = 2, it cannot be finished before its deadline 1. So job J_2 is discarded,

Solution set S = {J_1, J_4}, and Profit SP = {100, 27}

With the greedy approach, we will be able to schedule two jobs {J_1, J_4}, which gives a profit of 100 + 27 = 127 units..

Ex. 5.4.2 **MU - May 2016, 10 Marks**

Solve the following instance of "job sequencing with deadlines" problem :

n = 7, profits (p_1, p_2, p_3, p_4, p_5, p_6, p_7) = (3, 5, 20, 18, 1, 6, 30) and deadlines (d_1, d_2, d_3, d_4, d_5, d_6, d_7) = (1, 3, 4, 3, 2, 1, 2).

Schedule the jobs in such way so as to get maximum profit.

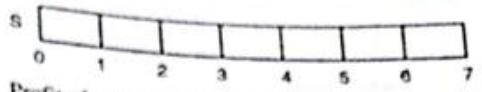
Jobs	j_1	j_2	j_3	j_4	j_5	j_6	j_7
Profit	3	5	20	18	1	6	30
Deadline	1	3	4	3	2	1	2

Soln. :

Sort all jobs in descending order of profit.

So, P = (30, 20, 18, 6, 5, 3, 1), J = (J_7, J_3, J_4, J_6, J_2, J_1, J_5) and D = (2, 4, 3, 1, 3, 1, 2). We shall select one by one job from the list of sorted jobs J, and check if it satisfies the deadline. If so, schedule the job in the latest free slot. If no such slot is found, skip the current job and process the next one.

Initially,

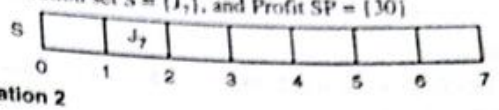


Profit of scheduled jobs, SP = 0

Iteration 1

Deadline for job J_7 is 2. Slot 2 (t = 1 to t = 2) is free, so schedule it in slot 2.

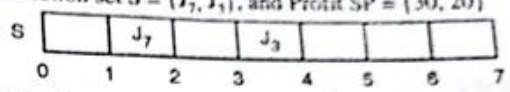
Solution set S = {J_7}, and Profit SP = {30}



Iteration 2

Deadline for job J_3 is 4. Slot 4 (t = 3 to t = 4) is free, so schedule it in slot 4.

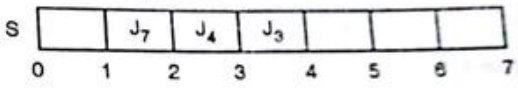
Solution set S = {J_7, J_3}, and Profit SP = {30, 20}



Iteration 3

Deadline for job J_4 is 3. Slot 3 (t = 2 to t = 3) is free, so schedule it in slot 3.

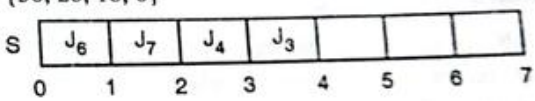
Solution set S = {J_7, J_3, J_4}, and Profit SP = {30, 20, 18}



Iteration 4

Deadline for job J_6 is 1. Slot 1 (t = 0 to t = 1) is free, so schedule it in slot 1.

Solution set S = {J_7, J_3, J_4, J_6}, and Profit SP = {30, 20, 18, 6}



First, all four slots are occupied and none of the remaining jobs has deadline lesser than 4. So none of the remaining jobs can be scheduled.

Thus, with the greedy approach, we will be able to schedule four jobs {J_7, J_3, J_4, J_6}, which give a profit of 30 + 20 + 18 + 6 = 74 units.

Ex. 5.4.3

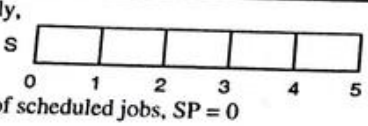
Obtain the optimal job sequence for following data of jobs.

N = 5, (P_1, P_2, P_3, P_4, P_5) = (25,15,12,5,1) and (D_1, D_2, D_3, D_4, D_5) = (3,2,1,3,3).

Soln. :

Jobs are already sorted in descending order of profit. We shall select one by one job from the list of sorted jobs checks if it satisfies the deadline. If so, schedule the job in the latest free slot. If no such slot is found, then skip the current job and process the next one.

Initially,

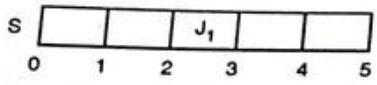


Profit of scheduled jobs, SP = 0

Iteration 1

Deadline for job J_1 is 3. Slot 3 (t = 2 to t = 3) is free, so schedule it in slot 3.

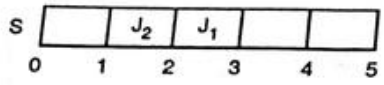
Solution set S = {J_1}, and Profit SP = {25}



Iteration 2

Deadline for job J_2 is 2. Slot 2 (t = 1 to t = 2) is free, so schedule it in slot 2.

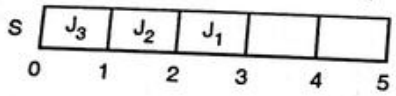
Solution set S = {J_1, J_2}, and Profit SP = {25, 15}



Iteration 3

Deadline for job J_3 is 1. Slot 1 (t = 0 to t = 1) is free, so schedule it in slot 1.

Solution set S = {J_1, J_2, J_3}, and Profit SP = {25, 15, 12}



Iteration 4

Job J_4 is not feasible because first three slots are already occupied and if we schedule J_4 any time after t = 3, it cannot be finished before its deadline 3. So job J_4 is discarded,

Solution set S = {J_1, J_2, J_3}, and Profit SP = {25, 15, 12}

Iteration 5

Job J_5 is not feasible because first three slots are already occupied and if we schedule J_5 any time after t = 3, it cannot be finished before its deadline 3. So job J_5 is discarded,

Solution set S = {J_1, J_2, J_3}, and Profit SP = {25, 15, 12}

With the greedy approach, we will be able to schedule three jobs {J_1, J_2, J_3 }, which gives a profit of 25 + 15 + 12 = 52 units.

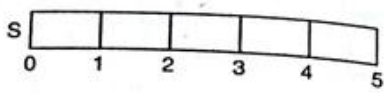
Ex. 5.4.4 **MU - May 2015, 10 Marks**

Explain Job sequencing with deadliner for the given instance n = 5, {P_1, P_2, P_3, P_4, P_5} = {20, 15, 10, 5, 3} and {d_1, d_2, d_3, d_4, d_5} = {2, 2, 1, 3, 3}

Soln. :

Jobs are already sorted in descending order of profit. We shall select one by one job from the list of sorted jobs, and check if it satisfies the deadline. If so, schedule the job in the latest free slot. If no such slot is found, skip the current job and process the next one.

Initially,

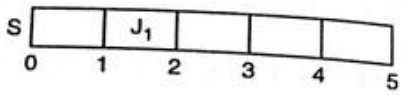


Profit of scheduled jobs, SP = 0

Iteration 1

Deadline for job J_1 is 2. Slot 2 (t = 1 to t = 2) is free, so schedule it in slot 2.

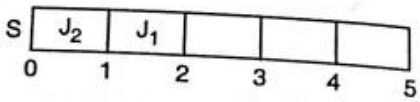
Solution set S = {J_1}, and Profit SP = {20}



Iteration 2

Deadline for job J_2 is 2. Slot 2 (t = 1 to t = 2) is occupied, so look for the previous possible slot. The slot 1 (t = 0 to t = 1) is free, so schedule it in slot 1.

Solution set S = {J_1, J_2}, and Profit SP = {20, 15}



Iteration 3

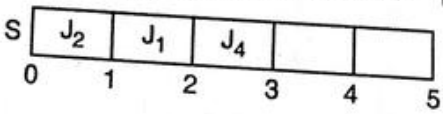
Job J_3 is not feasible because first two slots are already occupied and if we schedule J_3 any time later t = 2, it cannot be finished before its deadline 1. So job J_3 is discarded,

Solution set S = {J_1, J_2}, and Profit SP = {20, 15}

Iteration 4

Deadline for job J_4 is 3. Slot 3 (t = 2 to t = 3) is free, so schedule it in slot 3.

Solution set S = {J_1, J_2, J_4}, and Profit SP = {20, 15, 5}



Iteration 5

Job J_5 is not feasible because first three slots are already occupied and if we schedule J_5 any time later t = 3, it cannot be finished before its deadline 3. So job J_5 is discarded,

Solution set S = {J_1, J_2, J_4}, and Profit SP = {20, 15, 5}

– With greedy approach, we will be able to schedule three jobs {J_1, J_2, J_4}, which gives profit of 20 + 15 + 5 = 40 units.

Syllabus Topic : Minimum Cost Spanning Trees - Kruskal and Prim's Algorithm

5.5 Minimum Spanning Tree

5.5.1 Basics of Graph

Q. Define the terms : Graph, weighted graph, tree, spanning tree, minimum spanning tree. (5 Marks)

1. Graph

Definition

Graph G = (V, E) is defined by a set of vertices (V) and set of edges (E) joining those vertices.

For the graph shown in Fig. 5.5.1,

V = {A, B, C, D, E}

E = {AB, AD, AE, BD, BE, DC}

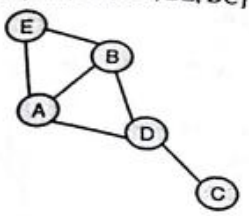


Fig. 5.5.1 : Graph

2. Weighted Graph

Definition

Graph G = (V, E, W) is called **weighted graph** if some weight or cost is associated with each of its edges. W represents the set of weight associated with each edge.

A graph representing the distance between cities is a weighted graph, where cost / weight is the distance between two corresponding cities. For the weighted graph is illustrated in Fig. 5.5.2,

V = {V1, V2, V3, V4}

E	V_1V_2	V_1V_3	V_1V_4	V_2V_3	V_2V_4	V_3V_4
W	10	20	12	15	22	25

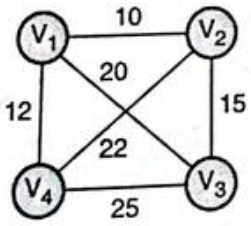


Fig. 5.5.2 : Weighted Graph

3. Tree

Definition

Tree T = (V', E') is a subset of Graph G = (V, E), where V' is a subset of V and E' is a subset of E. Tree does not contain a cycle, while Graph or subgraph can have a cycle.

Fig. 5.5.3 shows the graph, its tree and subgraph. Multiple trees and subgraphs are possible for given graph.

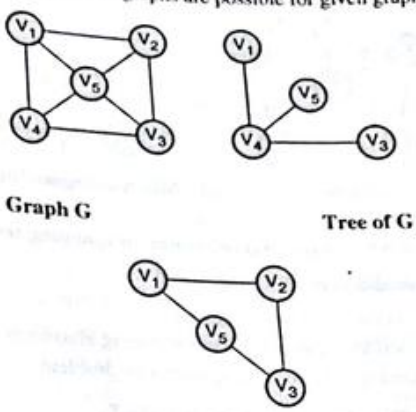


Fig. 5.5.3 : Graph, its tree and subgraph

4. Spanning Tree

Definition

Spanning tree T = (V', E') is a tree of connected, undirected, weighted graph G = (V, E, W), which contains all the vertices of G and some or all edges of G. So V' = V and E' ⊂ E.

Fig. 5.5.4 illustrates the spanning tree of graph G.

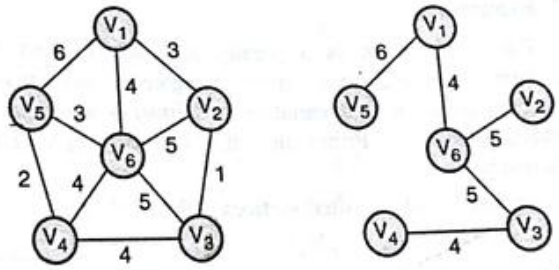


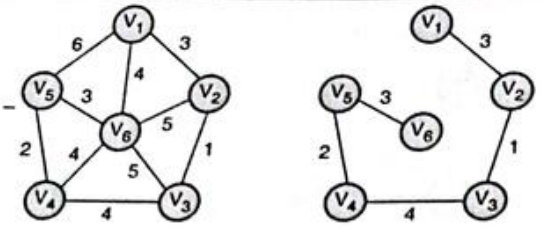
Fig. 5.5.4 : Graph and its spanning tree

5. Minimum Spanning Tree

Q. What is MST ? (2 Marks)

Definition : Spanning tree T = (V', E') is a tree of connected, undirected, weighted graph G = (V, E, W), which contains all the vertices of G and some or all edges of G. So V' = V and E' ⊂ E. Grapi, G can have many spanning trees with a different cost. **Minimum spanning tree** is a spanning tree with minimum cost.

Graph and its minimum spanning tree are shown in Fig. 5.5.5.



Graph G — Minimum spanning tree of G

Fig. 5.5.5 : Graph and its minimum spanning tree

☞ **Applications of MST**

(1) MST is used in network design.

(2) It is used to implement efficient routing algorithm.

(3) It is used to solve travelling salesman problem.

5.5.2 Prim's Algorithm for MST

Q. Write PRISM's algorithm of MST. Mention its time complexity. **(7 Marks)**

Q. Comment on the complexity of Prim's algorithm. Analysis complexity of Prim's algorithm using Greedy approach. **(7 Marks)**

Q. Explain prim's algorithm. **(6 Marks)**

☞ **Problem**

Find minimum spanning tree from given weighted, undirected graph G = <V, E>

☞ **Explanation**

Prim's algorithm is a greedy approach to find a minimum spanning tree from weighted, connected, undirected graph. It is a variation of Dijkstra's algorithm. Working principle of Prim's algorithm is very simple and explained here:

Initialize list of unvisited vertices with

$U_V = \{V_1, V_2, V_3, \ldots, V_n\}$

1. Select any arbitrary vertex from input graph G, call that subgraph as partial MST. Remove a selected vertex from U_V.
2. Form a set N_E –a set of unvisited neighbour edges of all vertices present in partial MST.
3. Select an edge $e = (u, v)$ from N_E with minimum weight, and add it to partial MST if it does not form a cycle and it is not already added.

   If the addition of edge e forms a cycle, then skip that edge and select next minimum weight edge from N_E. Continue this procedure until we get an edge e which does not form a cycle. Remove corresponding added vertices u and v from U_V.
4. Go to step 2 and repeat the procedure until U_V is empty.

Prim's algorithm is preferred when the graph is dense with V << E. Kruskal performs better in the case of the sparse graph with V ≡ E.

☞ **Algorithm for Prim's approach**

Algorithm for Prim's approach is described below :

```
Algorithm PRIM_MST(G, n)
// Description: Find MST of graph G using Prim's algorithm
// Input : Weighted connected graph G and number of vertices n
// Output : Minimum spanning tree MST and weight of it i.e. cost
cost ← 0
// Initialize tree nodes
for i ← 0 to n – 1 do
    MST[i] ← 0
end
MST[0] ← 1                    // 1 is the initial vertex
for k ← 1 to n do
    min_dist ← ∞
    for i ← 1 to n – 1 do
    for j ← 1 to n – 1 do
    if G[i, j] AND ((MST[i] AND ¬MST[j]) OR (¬MST[i] AND
    MST[j])) then
        if G[i, j] < min_dist then
            min_dist ← G[i, j]
            u ← i
            v ← j
        end
    end
    end
    end
    print (u, v, min_dist)
    MST[u] ← MST[v] ← 1
    cost ← cost + min_dist
end
print ("Total cost = ", cost)
```

☞ **Complexity analysis**

- In every phase, algorithm search for minimum weight edge. Running time of algorithm heavily depends on how efficiently it performs search operation.
- The time taken by the algorithm can be computed as,

$$T(n) = \sum_{k=1}^{n} \left(\sum_{i=0}^{n-1} 1 + \sum_{j=0}^{n-1} 1 \right)$$

Time taken by the outer most loop of k — Time taken by the middle loop of i — Time taken by the inner most loop of j

Rewriting the equation,

$$T(n) = \sum_{k=1}^{n} \left(\sum_{i=1}^{n} 1 + \sum_{j=1}^{n} 1 \right) = \sum_{k=1}^{n} (n+n)$$

$$= \sum_{k=1}^{n} 2n = 2n \sum_{k=1}^{n} 1 = 2n^2$$

$T(n) = O(n^2)$

- Straight forward approach to this is search entire adjacency matrix for each vertex. It takes $O(|V|^2)$ time for |V| vertices and |E| edges, where |V| represents a number of vertices i.e. n.
- However, with the help of some other data structures, we can improve the performance. Using binary heap and adjacency list, time can be reduced to $O(|E| \log |V|)$
- Using Fibonacci heap and adjacency matrix, performance can be improved to $O(|E| + |V| \log |V|)$.

Ex. 5.5.1

Find the cost of Minimal Spanning Tree of the given graph by using Prim's Algorithm.

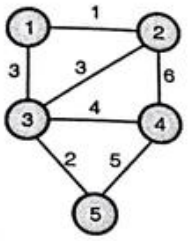
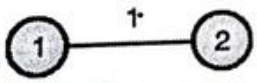
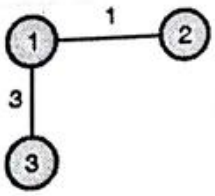
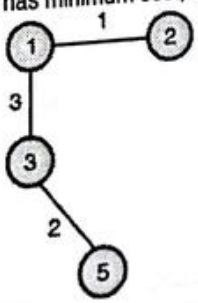
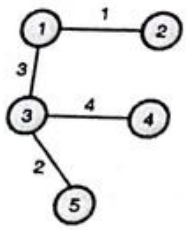


Fig. P. 5.5.1

Soln. :

- Initialize U_V – list of unvisited vertices with all vertices in given graph.
- Prim's algorithm adds some arbitrary vertex V_s from U_V to the partial solution and removes V_s from U_V.
- N_E - set of all neighbour edges of vertices in partial solution is explored.
- The neighbour edge with minimum cost is added in partial solution if it does not form cycle and not already added to the partial solution.
- Otherwise, select next minimum cost edge and add it to the partial solution and remove it from the list of unvisited vertices U_V.
- This process is repeated until all vertices are visited, i.e. U_V becomes empty.
- For given graph, initially the set of unvisited vertices $U_V = \{1, 2, 3, 4, 5\}$.

Partial solution	Set of neighbour edges N_E	Cost of neighbour edges	Updated U_V
Step 1 : $U_V = \{1, 2, 3, 4, 5\}$. Let us start with arbitrary vertex 1. Initial partial solution 	<1, 2> <1, 3>	1 3	{2, 3, 4, 5}
Step 2 : Edge <1, 2> has minimum cost, so add it 	<1, 3> <2, 3> <2, 4>	3 3 4	{3, 4, 5}
Step 3 : Edge <1, 3> has minimum cost, so add it. 	<2, 3> <2, 4> <3, 4> <3, 5>	3 4 4 2	{4, 5}
Step 4 : Edge <3, 5> has minimum cost, so add it. 	<2, 4> <3, 4> <5, 4>	6 4 5	{4}

Partial solution	Set of neighbour edges N_E	Cost of neighbour edges	Updated U_V
Step 5 : Edge <3, 4> has a minimum cost, so add it. 	U_V is empty so the tree generated in step 5 is the minimum spanning tree of given graph. Let w(u, v) represent the weight of edge (u, v) Cost of solution: $w(1, 2) + w(1, 3) + w(3, 4) + w(3, 5)$ $= 1 + 3 + 4 + 2$ $= 10$		

Ex. 5.5.2

Find the cost of minimum spanning tree of the given graph by using Prim's algorithm.

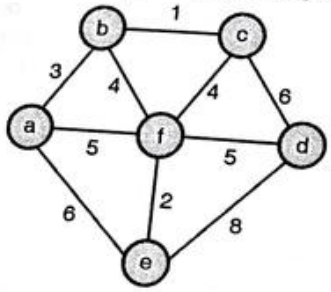
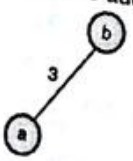
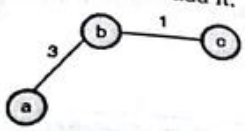
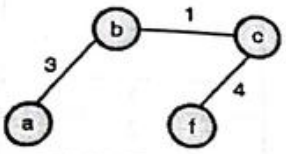
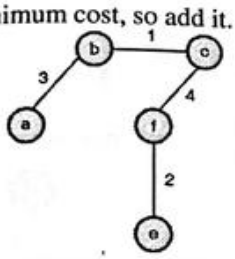
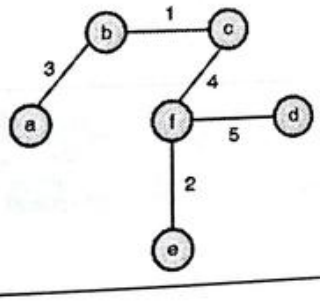


Fig. P. 5.5.2

Soln. :

- Initialize U_V – list of unvisited vertices - with all vertices in given graph.
- Prim's algorithm adds some arbitrary vertex V_x from U_V to the partial solution and removes V_x from U_V.
- N_E - set of all neighbour edges of vertices in partial solution is explored.
- The neighbour edge with minimum cost is added in partial solution if it does not form cycle and not already added to the partial solution.
- Otherwise, select next minimum cost edge and add it to the partial solution and remove it from the list of unvisited vertices U_V. This process is repeated until all vertices are visited, i.e. U_V becomes empty.
- For given graph, initially the set of unvisited vertices U_V = { a, b, c, d, e, f }.

Partial solution	Set of neighbour edges N_E	Cost of neighbour edges	Updated U_V
Step 1 : U_V = {a, b, c, d, e, f}. Let us start with arbitrary vertex a. Initial partial solution 	<a, b> <a, f> <a, e>	3 5 6	{ b, c, d, e, f}
Step 2 : Edge <a, b> has minimum cost, so add it. 	<a, f> <a, e> <b, c> <b, f>	5 6 1 4	{ c, d, e, f}

Partial solution	Set of neighbour edges N_E	Cost of neighbour edges	Updated U_V
Step 3 : Edge <b, c> has minimum cost, so add it. 	<a, f> <a, e> <b, f> <c, f> <c, d>	5 6 4 4 6	{ d, e, f}
Step 4 : Edge <b, f> and <c, f> have same minimum cost, so we can add any of it. Let us add <c, f> 	<a, f> <a, e> <b, f> <c, d> <f, d> <f, e>	5 6 4 6 5 2	{ d, e }
Step 5 : Edge <f, e> has minimum cost, so add it. 	<a, f> <a, e> <b, f> <c, d> <f, d>	5 6 4 6 5	{ d }
Step 6 : Edge <b, f> has minimum cost, but its inclusion in above partial solution creates cycle, so skip edge <b, f> and check for next minimum cost edge i.e. <a, f> Inclusion of <a, f> also creates cycle so skip it and check for next minimum cost edge, i.e. <f, d> The inclusion of <f, d> does not create a cycle, so it is a feasible edge, add it. 			U_V is empty so the tree generated in step 6 is the minimum spanning tree of given graph. Let w(u, v) represent the weight of edge (u, v) Cost of solution: w(a, b) + w(b, c) + w(c, f) + w(f, d) + w(f, e) = 3 + 1 + 4 + 5 + 2 = 15

Ex. 5.5.3

Find MST using Prim's algorithm.

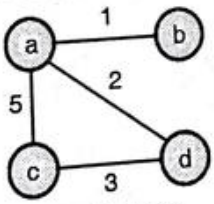
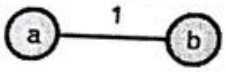
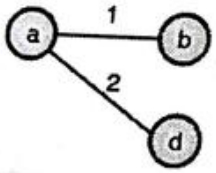
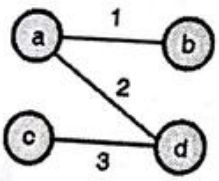


Fig. P. 5.5.3

Soln. :

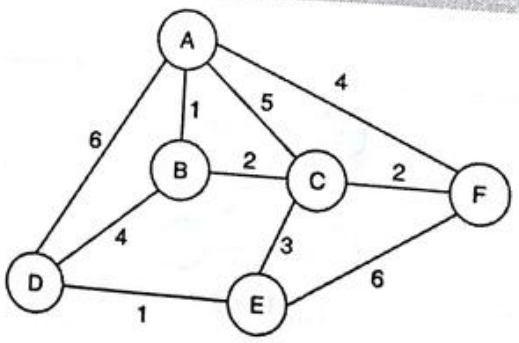
- Initialize U_V – list of unvisited vertices with all vertices in given graph.

- Prim's algorithm adds some arbitrary vertex V_x from U_V to the partial solution and removes V_x from U_V.
- N_E - set of all neighbour edges of vertices in partial solution is explored.
- The neighbour edge with minimum cost is added in partial solution if it does not form cycle and not already added to the partial solution.
- Otherwise, select next minimum cost edge and add it to the partial solution and remove it from the list of unvisited vertices U_V.
- This process is repeated until all vertices are visited, i.e. U_V becomes empty.
- For given graph, initially the set of unvisited vertices U_V = { a, b, c, d, e, f }.

Partial solution	Set of neighbour edges N_E	Cost of neighbour edges	Updated U_V
Step 1 : U_V = {a, b, c, d }. Let us start with arbitrary vertex a. Initial partial solution 	<a, b> <a, c> <a, d>	1 5 2	{ b, c, d }
Step 2 : Edge <a, b> has minimum cost, so add it. 	<a, c> <a, d>	5 2	{ c, d }
Step 3 : Edge <a, d> has minimum cost, so add it. 	<a, c> <d, c>	5 3	{ c }
Step 4 : Edge <d, c> has a minimum cost, so add it. 	U_V is empty so the tree generated in step 4 is the minimum spanning tree of given graph. Let w(u, v) represent the weight of edge (u, v) Cost of solution: w(a, b) + w(a, d) + w(c, d) = 1 + 2 + 3 = 6		

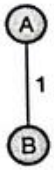
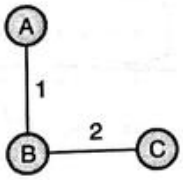
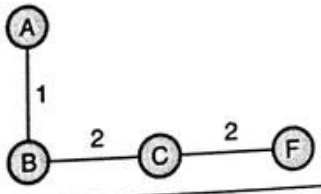
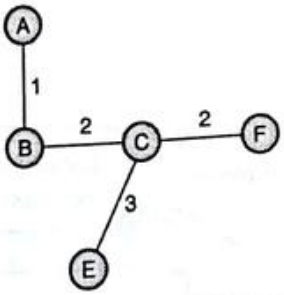
Ex. 5.5.4 **MU - May 2017. 5 Marks**

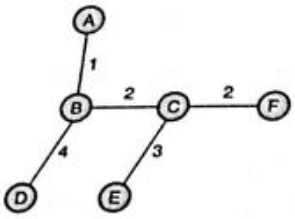
Find minimum spanning tree for the following graph.



Soln. :

- Initialize U_V – list of unvisited vertices with all vertices in given graph.
- Prim's algorithm adds some arbitrary vertex V_x from U_V to the partial solution and removes V_x from U_V.
- N_E - set of all neighbour edges of vertices in partial solution is explored.
- The neighbour edge with minimum cost is added in partial solution if it does not form cycle and not already added to the partial solution.
- Otherwise, select next minimum cost edge and add it to the partial solution and remove it from the list of unvisited vertices U_V.
- This process is repeated until all vertices are visited, i.e. U_V becomes empty.
- For given graph, initially the set of unvisited vertices U_V = { a, b, c, d, e, f }.

Partial solution	Set of neighbour edges N_E	Cost of neighbour edges	Updated U_V
Step 1 : U_V = {A, B, C, D, E, F }. Let us start with arbitrary vertex A. Initial partial solution 	<A, B> <A, C> <A, D> <A, F>	1 5 6 4	{B, C, D, E, F }
Step 2 : Edge <A, B> has minimum cost, so add it 	<A, C> <A, D> <A, F> <B, D> <B, C>	5 6 4 4 2	{ C, D, E, F }
Step 3 : Edge <B, C> has minimum cost, so add it. 	<A, C> <A, D> <A, F> <B, D> <C, E> <C, F>	5 6 4 4 3 2	{ D, E, F }
Step 4 : Edge <C, F> has minimum cost, so add it. 	<A, C> <A, D> <A, F> <B, D> <C, E>	5 6 4 4 3	{ D, E }
Step 5 : Edge <C, E> has minimum cost, so add it. 	<A, C> <A, D> <A, F> <B, D>	5 6 4 4	{ D }
Step 6 : Edge <A, F> and <B, D> have minimum cost. But inclusion of <A, F> forms cycle, so reject it. Whereas inclusion of <B, D> does not form cycle so	<A, C> <A, D>	5 6	{ }

Partial solution	Set of neighbour edges N_E	Cost of neighbour edges	Updated U_V
add it to the solution. 			
Step 7 : All vertices are added to the solution, and they all are connected	– U_V is empty so the tree generated in step 6 is the minimum spanning tree of given graph. – Let w(u, v) represent the weight of edge (u, v) – Cost of solution: w(A, B) + w(B, C) + w(B, D) + w(C, E) + w(C, F) = 1 + 2 + 4 + 3 + 2 = 12		

5.5.3 Kruskal's Algorithm

Q. Write Kruskal's algorithm for finding a minimum spanning tree. Compute the time complexity of the same. **(7 Marks)**

Q. Write Kruskal's algorithm to find a minimum spanning tree. **(7 Marks)**

Problem

Find minimum spanning tree from given weighted, undirected graph G = <V, E, W>

Explanation

- Joseph Kruskal has suggested a greedy approach to find MST from given weighted, undirected graph.
- The algorithm first sorts all the edges in nondecreasing order of their weight.
- Edge with minimum weight is selected and its feasibility is tested.
- If inclusion of the edge to a partial solution does not form the cycle, then the edge is feasible and added to the partial solution.
- If is not feasible then skip it and check for the next edge. The process is repeated until all edges are scanned.
- Let, list unvisited edges $U_E = \{e_1, e_2, e_3, \ldots, e_m\}$, be the edges sorted by increasing order of their weight.

1. Select minimum weight edge e_{min} from input graph G, which is not already added.
2. If the addition of edge e_{min} to a partial solution does not form a cycle, add it. Otherwise look for next minimum weight edge until we get the feasible edge. Remove checked edges from E.
3. Go to step 1 and repeat the procedure until all edges in E are scanned.

Algorithm for Kruskal's approach

Algorithm for Kruskal's approach is shown below :

```
Algorithm KUSKAL_MST(G)
// Description : Find minimum spanning tree of graph G of n vertices
// Input : Weighted undirected graph G
// Output : Minimum spanning tree of G

MST ← ∅
for each v ∈ V do                          Create disjoint-set tree
    MAKE-SET(v)
end

                                           Add x to disjoint-set tree
for each (u, v) ∈ E ordered by weight in increasing order do
    if FIND-SET(u) ≠ FIND-SET(v) then
        MST ← MST ∪ (u, v)
        UNION(u, v)
    end
end

Function MAKE-SET(x)
if x is not already present then
    x.parent ← x
    x.rank ← 0
end

Function FIND(x)
if x.parent ≠ x then
    x.parent ← FIND(x.parent)              Determine the roots of
end                                        tree x and y belongs to
return x.parent
Function UNITON(x, y)
xRoot ← FIND(x)
yRoot ← FIND(y)                            If x and y are already
if xRoot == yRoot then                     in same set
```

```
    return
end
if xRoot.rank < yRoot.rank then
    xRoot.parent ← yRoot
else if yRoot.rank < xRoot.rank then
    yRoot.parent ← xRoot
else
    xRoot.parent ← yRoot
    yRoot.rank ← yRoot.rank + 1
end
```

x and y are not in same set so merge

Arbitrary make one root the new parent

Time complexity

- Using Disjoint set data structures, with |E| edges and |V| vertices in the graph, Kruskal's algorithm runs in O(|E| $\log_2$ |E|) or O(|E| $\log_2$ |V|) time, both are identical. For the dense graph, |E| is at most $|V^2|$. So, $\log_2|E| = \log_2|V^2| = 2\log_2|V| \approx \log_2|V|$
- Sort edges using comparison based algorithm, which could sort it in O(|E|$\log_2$|E|) time. Selection and feasibility checking of the edge are done in O(|E|) time. Thus total running time is given as O(|E| + |E| log |E|) = O(|E| log |E|).

Ex. 5.5.5

Find the cost of Minimal Spanning Tree of the given graph by using Kruskal's Algorithm.

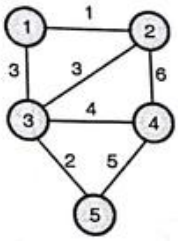


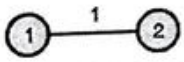
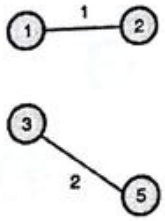
Fig. P. 5.5.5

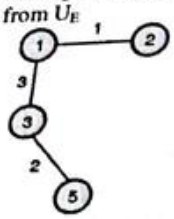
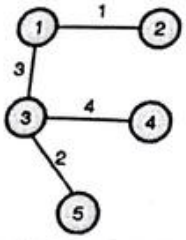
Soln. :

- Kruskal's algorithm sorts the edges according to decreasing order of their weight. Sorted edges are listed in following table:

Edge	<1, 2>	<3, 5>	<1, 3>	<2, 3>	<3, 4>	<4, 5>	<2, 4>
Cost	1	2	3	3	4	5	6

- One by one edge is added to a partial solution if it is not forming the cycle.
- Initially, set of unvisited edges U_E = {<1, 2>, <3, 5>, <1, 3>, <2, 3>, <3, 4>, <4, 5>, <2, 4>}

Partial solution	Updated U_E
Step 1 : Minimum cost edge is <1, 2>, so add it to MST and remove from U_E 	U_E = { <3, 5>, <1, 3>, <2, 3>, <3, 4>, <4, 5>, <2, 4> }
Step 2 : Minimum cost edge is <3, 5>, so add it to MST and remove from U_E 	U_E = { <1, 3>, <2, 3>, <3, 4>, <4, 5>, <2, 4> }

Partial solution	Updated U_E
Step 3 : Minimum cost edge is <1, 3>, so add it to MST and remove from U_E 	U_E = { <2, 3>, <3, 4>, <4, 5>, <2, 4> }
Step 4 : Minimum cost edge is <2, 3>, but its inclusion creates cycle so remove it from U_E So, U_E={<3, 4>, <4, 5>, <2, 4> } Next minimum cost edge is <3, 4> and its inclusion does not form cycle, so add it to MST and remove from U_E 	U_E = { <4, 5>, <2, 4> }
Step 5 : Minimum cost edge is <4, 5>, but its inclusion creates cycle so remove it from U_E So, U_E = { <2, 4> } Minimum cost edge is <2, 4>, but its inclusion creates cycle so remove it from U_E So, U_E = { }	U_E is empty so the tree generated in step 4 is the minimum spanning tree of given graph. Let w(u, v) represent the weight of edge (u, v) Cost of solution: w(1, 2) + w(1, 3) + w(3, 4) + w(3, 5) = 1 + 3 + 4 + 2 = 10

Ex. 5.5.6

Find out minimum cost spanning tree using Kruskal algorithm.

Edge	Cost	Edge	Cost
(V_1,V_7)	1	(V_4,V_5)	7
(V_3,V_4)	3	(V_1,V_2)	20
(V_2,V_7)	4	(V_1,V_6)	23
(V_3,V_7)	9	(V_5,V_7)	25
(V_2,V_3)	15	(V_5,V_6)	28
(V_4,V_7)	16	(V_6,V_7)	36

Soln. : Graph representation of given data is shown in Fig. P. 5.5.6.

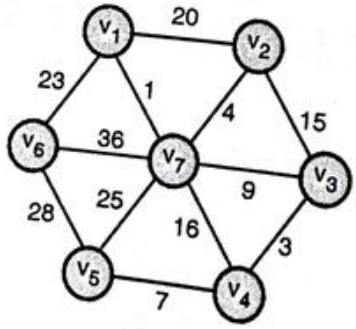
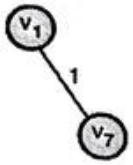
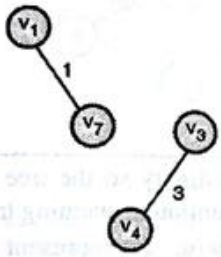
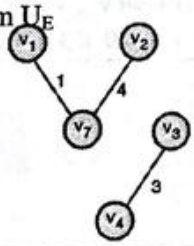
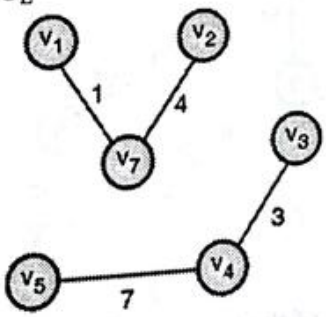


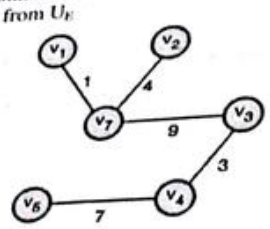
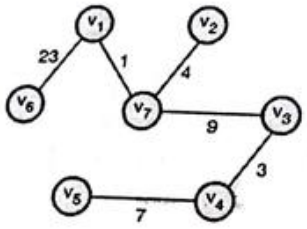
Fig. P. 5.5.6

- Kruskal's algorithm sorts the edges according to decreasing order of their weight. Sorted edges are listed in the following table:

Edge	$<V_1, V_7>$	$<V_3, V_4>$	$<V_2, V_7>$	$<V_4, V_5>$	$<V_3, V_7>$	$<V_2, V_3>$
Cost	1	3	4	7	9	15
Edge	$<V_4, V_7>$	$<V_1, V_2>$	$<V_1, V_6>$	$<V_5, V_7>$	$<V_5, V_6>$	$<V_6, V_7>$
Cost	16	20	23	25	28	36

- One by one edge is added to a partial solution if it is not forming the cycle.
- Initially, set of unvisited edges U_E = {$<V_1, V_7>$, $<V_3, V_4>$, $<V_2, V_7>$, $<V_4, V_5>$, $<V_3, V_7>$, $<V_2, V_3>$, $<V_4, V_7>$, $<V_1, V_2>$, $<V_1, V_6>$, $<V_5, V_7>$, $<V_5, V_6>$, $<V_6, V_7>$}

Partial solution	Updated U_E
Step 1 : Minimum cost edge is $<V_1, V_7>$, so add it to MST and remove from U_E. 	U_E = {$<V_3, V_4>$, $<V_2, V_7>$, $<V_4, V_5>$, $<V_3, V_7>$, $<V_2, V_3>$, $<V_4, V_7>$, $<V_1, V_2>$, $<V_1, V_6>$, $<V_5, V_7>$, $<V_5, V_6>$, $<V_6, V_7>$}
Step 2 : Next minimum cost edge is $<V_3, V_4>$, so add it to MST and remove from U_E 	U_E = {$<V_2, V_7>$, $<V_4, V_5>$, $<V_3, V_7>$, $<V_2, V_3>$, $<V_4, V_7>$, $<V_1, V_2>$, $<V_1, V_6>$, $<V_5, V_7>$, $<V_5, V_6>$, $<V_6, V_7>$}
Step 3 : Next minimum cost edge is $<V_2, V_7>$, so add it to MST and remove from U_E 	U_E = {$<V_4, V_5>$, $<V_3, V_7>$, $<V_2, V_3>$, $<V_4, V_7>$, $<V_1, V_2>$, $<V_1, V_6>$, $<V_5, V_7>$, $<V_5, V_6>$, $<V_6, V_7>$}
Step 4 : Next minimum cost edge is $<V_4, V_5>$, so add it to MST and remove from U_E 	U_E = {$<V_3, V_7>$, $<V_2, V_3>$, $<V_4, V_7>$, $<V_1, V_2>$, $<V_1, V_6>$, $<V_5, V_7>$, $<V_5, V_6>$, $<V_6, V_7>$}

Partial solution	Updated U_E
Step 5 : Next minimum cost edge is $<V_3, V_7>$, so add it to MST and remove from U_E 	$U_E = \{<V_2, V_3>, <V_4, V_7>, <V_1, V_2>, <V_1, V_6>, <V_5, V_7>, <V_5, V_6>, <V_6, V_7>\}$
Step 6 : Next minimum cost edge is $<V_2, V_3>$, but its inclusion creates cycle so remove it from U_E. So, $U_E = \{<V_4, V_7>, <V_1, V_2>, <V_1, V_6>, <V_5, V_7>, <V_5, V_6>, <V_6, V_7>\}$ Similarly, next minimum cost edges are $<V_4, V_7>$ and $<V_1, V_2>$ form cycle so remove them from U_E. So, $U_E = \{ <V_1, V_6>, <V_5, V_7>, <V_5, V_6>, <V_6, V_7> \}$  so add it to MST and remove from U_E	$U_E = \{ <V_5, V_7>, <V_5, V_6>, <V_6, V_7> \}$
Step 7 : Next minimum cost edge is $<V_5, V_7>$, but its inclusion creates cycle so remove it from U_E. So, $U_E = \{ <V_5, V_6>, <V_6, V_7> \}$ Similarly, next minimum cost edges are $<V_5, V_6>$ and $<V_6, V_7>$ form cycle so remove them from U_E. So, $U_E = \{ \}$	U_E is empty so the tree generated in step 4 is the minimum spanning tree of given graph. Let w(u, v) represent the weight of edge (u, v) Cost of solution: $w(V_1, V_6) + w(V_1, V_7) + w(V_2, V_7) + w(V_7, V_3) + w(V_3, V_4) + w(V_4, V_5)$ $= 23 + 1 + 4 + 9 + 3 + 7 = 47$

Ex. 5.5.7 **MU - Dec. 2014, 10 Marks**

To find MST of following graph using prim's and kruskal's Algorithm.

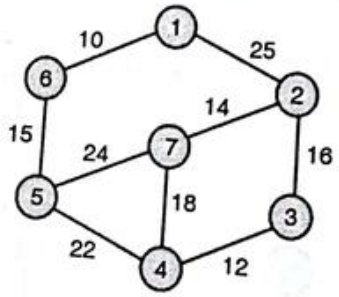


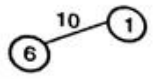
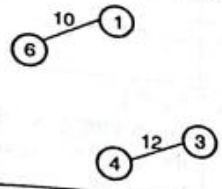
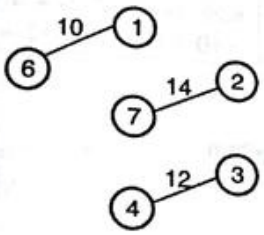
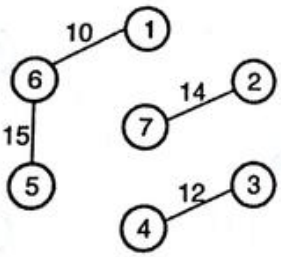
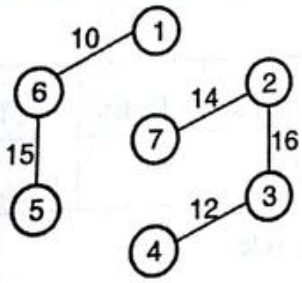
Fig. P. 5.5.7

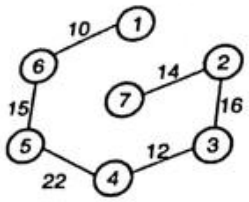
Soln. :

– Kruskal's algorithm sorts the edges according to decreasing order of their weight. Sorted edges are listed in following table.

Edge	<1, 6>	<3, 4>	<2, 7>	<5, 6>	<2, 3>	<4, 7>	<4, 5>	<5, 7>	<1, 2>
Cost	10	12	14	15	16	18	22	24	25

- One by one edge is added to a partial solution if it is not forming the cycle.
- Initially, set of unvisited edges U_E = {<1, 6>, <3, 4>, <2, 7>, <5, 6>, <2, 3>, <4, 7>, <4, 5>, <5, 7>, <1, 2>}

Partial solution	Updated U_E
Step 1 : Minimum cost edge is <1, 6>, so add it to MST and remove from U_E. 	U_E = { <3, 4>, <2, 7>, <5, 6>, <2, 3>, <4, 7>, <4, 5>, <5, 7>, <1, 2>}
Step 2 : Next minimum cost edge is <3, 4>, so add it to MST and remove from U_E 	U_E = { <2, 7>, <5, 6>, <2, 3>, <4, 7>, <4, 5>, <5, 7>, <1, 2>}
Step 3 : Next minimum cost edge is <2, 7>, so add it to MST and remove from U_E 	U_E = { <5, 6>, <2, 3>, <4, 7>, <4, 5>, <5, 7>, <1, 2>}
Step 4 : Next minimum cost edge is <5, 6>, so add it to MST and remove from U_E 	U_E = {<2, 3>, <4, 7>, <4, 5>, <5, 7>, <1, 2>}
Step 5 : Next minimum cost edge is <2, 3>, so add it to MST and remove from U_E 	U_E = {<4, 7>, <4, 5>, <5, 7>, <1, 2>}

Partial solution	Updated U_E
Step 6 : Next minimum cost edge is <4, 7>, but its inclusion creates cycle so remove it from U_E.	U_E = {<4, 5>, <5, 7>, <1, 2>}
Step 7 : Next minimum cost edge is <4, 5>, so add it to MST and remove from U_E 	U_E = {<5, 7>, <1, 2>}
Step 8 : Next minimum cost edge is <V_5, V_7>, but its inclusion creates cycle so remove it from U_E.	U_E = { <1, 2>}
Step 9 : Next minimum cost edge is <1, 2>, but its inclusion creates cycle so remove it from U_E.	U_E = { } U_E is empty so the tree generated in step 7 is the minimum spanning tree of given graph. Let w(u, v) represent the weight of edge (u, v) Cost of solution: w(1, 6) + w(5, 6) + w(4, 5) + w(3, 4) + w(2, 3) + w(2, 7) = 10 + 15 + 22 + 12 + 16 + 14= 89

Ex. 5.5.8 **MU - May 2016, 10 Marks**

Find the minimum spanning tree of the following graph using Kruskal's algorithm

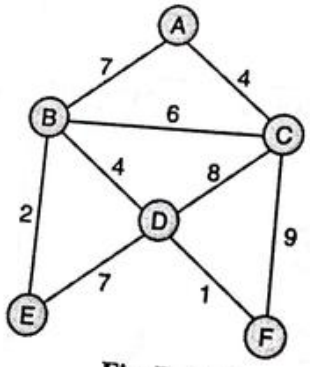


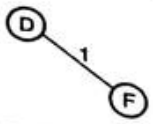
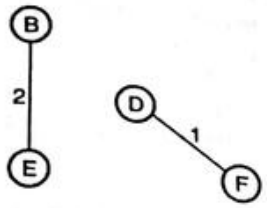
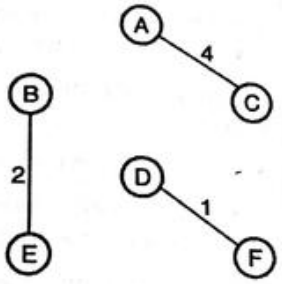
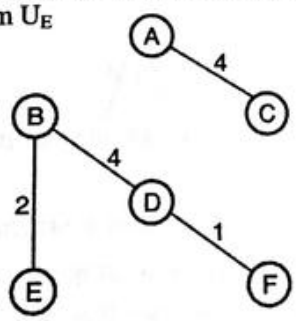
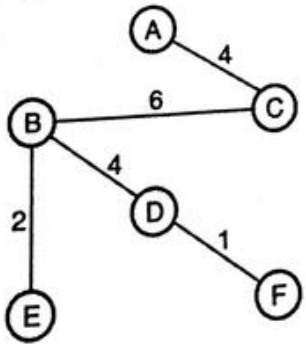
Fig. P. 5.5.8

Soln. :

- Kruskal's algorithm sorts the edges according to decreasing order of their weight. Sorted edges are listed in following table:

Edge	<D, F>	<B, E>	<A, C>	<B, D>	<B, C>	<A, B>	<D, E>	<C, D>	<C, F>
Cost	1	2	4	4	6	7	7	8	9

- One by one edge is added to a partial solution if it is not forming the cycle.
- Initially, set of unvisited edges U_E = {<D, F>, <B, E>, <A, C>, <B, D>, <B, C>, <A, B>, <D, E>, <C, D>, <C, F>}

Partial solution	Updated U_E
Step 1 : Minimum cost edge is <D, F>, so add it to MST and remove from U_E 	U_E = {<B, E>, <A, C>, <B, D>, <B, C>, <A, B>, <D, E>, <C, D>, <C, F>}
Step 2 : Next minimum cost edge is <B, E>, so add it to MST and remove from U_E 	U_E = {<A, C>, <B, D>, <B, C>, <A, B>, <D, E>, <C, D>, <C, F>}
Step 3 : Next minimum cost edge is <A, C>, so add it to MST and remove from UE 	U_E = {<B, D>, <B, C>, <A, B>, <D, E>, <C, D>, <C, F>}
Step 4 : Next minimum cost edge is <B, D>, so add it to MST and remove from U_E 	U_E = {<B, C>, <A, B>, <D, E>, <C, D>, <C, F>}
Step 5 : Next minimum cost edge is <B, C>, so add it to MST and remove from U_E 	U_E = {<A, B>, <D, E>, <C, D>, <C, F>}

Partial solution	Updated U_E
Step 6 : Next minimum cost edge is <A, B>, but its inclusion creates cycle so remove it from U_E.	U_E = {<D, E>, <C, D>, <C, F>}
Step 7 : Next minimum cost edge is <D, E>, but its inclusion creates cycle so remove it from U_E.	U_E = {<C, D>, <C, F>}
Step 8 : Next minimum cost edge is <C, D>, but its inclusion creates cycle so remove it from U_E.	U_E = {<C, F>}
Step 9 : Next minimum cost edge is <C, F>, but its inclusion creates cycle so remove it from U_E.	U_E = { } U_E is empty so the tree generated in step 5 is the minimum spanning tree of given graph. Let w(u, v) represent the weight of edge (u, v) Cost of solution: w(A, C) + w(B, C) + w(B, D) + w(B, E) + w(D, F) = 4 + 6 + 4 + 2 + 1= 17

5.5.4 Differentiate : Prim's vs. and Kruskal's algorithm

Q. Compare Prim's algorithm and Kruskal's algorithm for finding the minimum spanning tree. **(6 Marks)**

Sr.	Prim's Algorithm	Kruskal's Algorithm
1.	Prim's algorithm always chooses the next edge which is a neighbour of vertices in partially generated solution.	The Kruskal algorithm always chooses the edge having a minimum weight.
2.	Prim's algorithm ensures that partial solution is always a tree.	In Kruskal's algorithm, a partial solution can be a forest.
3.	Sorting of edges is not required.	Sorting of edges is compulsory.
4.	Picking up next edge requires finding an edge with minimum cost from a set of adjacent edges.	As edges are always sorted, minimum cost edge can be chosen in O(1) time.
5.	Prim's algorithm is a better choice for the dense graph.	Kruskal's algorithm is a better choice for the sparse graph.

Syllabus Topic : Optimal Storage on Tapes

5.6 Optimal Storage on Tapes

→ **(May 13, Dec. 14)**

Q. Write short note on optimal storage on tapes. **MU - May 2013, 10 Marks**

Q. Explain optimal storage on tape with example. **MU - Dec. 2014, 10 Marks**

Q. How to solve optimal storage problems using greedy approach? **(6 Marks)**

Problem

Given n programs $P_1, P_2, \ldots, P_n$ of length $L_1, L_2, \ldots, L_n$ respectively, store them on tap of length L such that Mean Retrieval Time (MRT) be minimum.

Retrieval time of the j^{th} program is a summation of the length of first j programs on tap. Let T_j be the time to retrieve program P_j. Retrieval time of P_j is computed as,

$$T_j = \sum_{k=1}^{j} L_k$$

Length of k^{th} program

Mean retrieval time of n programs is the average time required to retrieve any program. It is required to store programs in an order such that their Mean Retrieval Time is minimum. MRT is computed as,

$$MRT = \frac{1}{n}\sum_{i=1}^{N} T_j = \frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{j} L_k$$

Average retrieval time over n programs

Time to retrieve j^{th} program P_j

Length of k^{th} program

Optimal storage on tape is minimization problem which,

$$\text{Minimize} \sum_{i=1}^{n} \sum_{k=1}^{i} L_k$$

$$\text{Subjected to} \sum_{i=1}^{n} L_i \leq L$$

L : Length of tape

L_i : Length of i^{th} program

5.6.1 Storage on Single Tape

Q. Explain an algorithm to derive optimal storage schedule using greedy approach for a single tape. (10 Marks)

- In this case, we have to find the permutation of the program order which minimizes the MRT after storing all programs on *single tape* only.
- There exist many permutations of programs. Each gives possibly different MRT. Consider three programs (P_1, P_2, P_3) with length $(L_1, L_2, L_3) = (5, 10, 2)$.
- Let's find the MRT for different permutations. 6 permutations are possible for 3 items. Mean Retrieval Time for each permutation is listed in the following table.

Ordering	MRT
P_1, P_2, P_3	((5) + (5 + 10) + (5 + 10 + 2)) / 3 = 37 / 3
P_1, P_3, P_2	((5) + (5 + 2) + (5 + 2 + 10)) = 29 / 3
P_2, P_1, P_3	((10) + (10 + 5) + (10 + 5 + 2)) = 42 / 3
P_2, P_3, P_1	((10) + (10 + 2) + (10 + 2 + 5)) = 39 / 3
P_3, P_1, P_2	((2) + (2 + 5) + (2 + 5 + 10)) = 26 / 3
P_3, P_2, P_1	((2) + (2 + 10) + (2 + 10 + 5)) = 31 / 3

- It should be observed from above table that the MRT is 26/3, which is achieved by storing the programs in **ascending order of their length**.
- Thus, greedy algorithm stores the programs on tape in nondecreasing order of their length, which ensures the minimum MRT.
- Let L be the array of program length in ascending order. The greedy algorithm finds the MRT as following:

```
Algorithm MRT_SINGLE_TAPE(L)
// Description : Find storage order of n programs to such that mean
      retrieval time is minimum
// Input : L is array of program length sorted in ascending order
// Output : Minimum Mean Retrieval Time

Tj ← 0
for i ← 1 to n do
      for j ← 1 to i do
      Tj ←Tj + L[j]
      end
end
MRT ← sum(T) / n
```

Complexity analysis

Primitive operation in above algorithm is the addition of program length, which is enclosed within two loops. The running time of algorithm is given by,

$$T(n) = \sum_{i=1}^{n} \sum_{j=1}^{i} \theta(1)$$

Cost of addition

Loop over n programs

Retrieval time for j^{th} program

$$= \sum_{i=1}^{n} \sum_{j=1}^{i} 1$$

$$= \sum_{i=1}^{n} i = 1 + 2 + 3 + \ldots + n$$

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

$$T(n) = O(n^2)$$

This algorithm runs in $O(n^2)$ time.

Ex. 5.6.1 MU - Dec. 2015, 10 Marks

Explain optimal storage on tapes and find the optimal order for given instance. n = 3, and $(i_1, i_2, i_3) = (5, 10, 3)$.

Soln. :

First, arrange the programs in non-decreasing order of length.

List L = {i3, i1, i2} = {3, 5, 10}

Mean retrieval time for the given programs is computed as follow:

```
Tj ← 0
for i ← 1 to n do
      for j ← 1 to i do
      Tj ←Tj + L[j]
      end
end
MRT ← sum(T)/ n
```

n = number of programs = 3

For i = 1:

$T_1 = T_1 + L[1] = 0 + 3 = 3$

For i = 2:

$T_1 = T_1 + L[1] = 3 + 3 = 6$

$T_2 = T_2 + L[2] = 0 + 5 = 5$

For i = 3:

$T_1 = T_1 + L[1] = 6 + 3 = 9$

$T_2 = T_2 + L[2] = 5 + 5 = 10$

$T_3 = T_3 + L[3] = 0 + 10 = 10$

MRT = sum(T) / n
= (9 + 10 + 10) / 3
= 9.67

5.6.2 Storage on Multiple Tapes

Q. Explain greedy method to solve the problem of storage on multiple tapes. **(3 Marks)**

Q. With a suitable example, discuss the storage on multi-tape problem using greedy approach. **(3 Marks)**

- This is the problem of minimizing MRT on retrieval of the program from multiple tapes.
- Instead of single tape, programs are to be stored on multiple tapes. Greedy solves this problem in a similar way. It sorts the programs according to increasing length of program and stores the program in one by one in each tape.
- Working of this approach is explained in the following example.

Ex. 5.6.2

Given the program lengths L = {12, 34, 56, 73, 24, 11, 34, 56, 78, 91, 34, 91, 45}. Store them on three taps and minimize MRT.

Soln. :

Given data :

P_i	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}	P_{11}	P_{12}	P_{13}
L_i	12	34	56	73	24	11	34	56	78	91	34	91	45

First sort the programs in increasing order of their size.

Sorted data:

P_i	P_6	P_1	P_5	P_2	P_7	P_{11}	P_{13}	P_3	P_8	P_4	P_9	P_{10}	P_{12}
L_i	11	12	24	34	34	34	45	56	56	73	78	91	91

Now distribute the files among all three tapes.

Tape 1	P_6	P_2	P_{13}	P_4	P_{12}
Tape 2	P_1	P_7	P_3	P_9	
Tape 3	P_5	P_{11}	P_8	P_{10}	

$MRT_{Tape1} = ((11) + (11 + 34) + (11 + 34 + 45) + ((11 + 34 + 45 + 73)) + (11 + 34 + 45 + 73 + 91)) / 4$
$= 563 / 5 = 112.6$

$MRT_{Tape2} = ((12) + (12 + 34) + (12 + 34 + 56) + (12 + 34 + 56 + 78)) / 4$
$= 340 / 4 = 85$

$MRT_{Tape3} = ((24) + (24 + 34) + (24 + 34 + 56) + (24 + 34 + 56 + 91)) / 4$
$= 353 / 4 = 88.25$

$MRT = (MRT_{Tape1} + MRT_{Tape2} + MRT_{Tape3}) / 3 = (112.6 + 85 + 88.25) / 3 = 95.28$

5.7 Exam Pack (University and Review Questions)

☞ **Syllabus Topic : General Method**

Q. What is the greedy algorithmic approach? *(Ans. : Refer section 5.1.1)* (5 Marks)

Q. Write an abstract algorithm for greedy design method. *(Ans. : Refer section 5.1.2)* (5 Marks) **(May 2015)**

Q. Explain the control abstraction of the greedy method. *(Ans. : Refer section 5.1.2)* (3 Marks)

Q. Comment on the module of computation : Greedy Method. *(Ans. : Refer section 5.1.3)*(5 Marks)

Q. Enlist and explain the characteristics of greedy algorithms. *(Ans. : Refer section 5.1.3)*(4 Marks)

Q. Write a short note : Differentiate between greedy approach and dynamic programming. *(Ans. : Refer section 5.1.6)* (5 Marks) **(May 2013)**

Q. Compare dynamic programming with the greedy approach. *(Ans. : Refer section 5.1.6)* (5 Marks)

☞ **Syllabus Topic : Single Source Shortest Path**

Q. Write an algorithm to compute the shortest distance between the source and destination vertices of a connected graph. Will your algorithm work for negative weights? *(Ans. : Refer section 5.2)* (6 Marks)

Ex. 5.2.4 (10 Marks) **(May 2014)**

Ex. 5.2.5 (10 Marks) **(May 2016)**

☞ **Syllabus Topic : Knapsack Problem**

Q. What is 0/1 Knapsack and fractional knapsack problem. *(Ans. : Refer section 5.3)* (3 Marks)

Q. State and solve knapsack problem using the greedy method. *(Ans. : Refer section 5.3)*

Q. Explain and write the algorithm for 0/1 Knapsack using greedy approach. *(Ans. : Refer section 5.3.1)* (6 Marks)

Ex. 5.3.3 (7 Marks) **(May 2017)**

☞ **Syllabus Topic : Job Sequencing with Deadlines**

Q. Write a note on : Job sequencing with deadlines. *(Ans. : Refer section 5.4)* (10 Marks)

(May 2014, Dec. 2016)

Q. State and solve job sequencing problem using greedy approach.
(Ans. : Refer section 5.4)(6 Marks)

Ex. 5.4.1 (10 Marks) **(Dec. 2015, May 2017)**

Ex. 5.4.2 (10 Marks) **(May 2016)**

Ex. 5.4.4 (10 Marks) **(May 2015)**

☞ **Syllabus Topic : Minimum Cost Spanning Trees - Kruskal and Prim's Algorithm**

Q. Define the terms : Graph, weighted graph, tree, spanning tree, minimum spanning tree.
(Ans. : Refer section 5.5.1) (5 Marks)

Q. What is MST ?
(Ans. : Refer section 5.5.1(5))(2 Marks)

Q. Write PRISM's algorithm of MST. Mention its time complexity. *(Ans. : Refer section 5.5.2)*(7 Marks)

Q. Comment on the complexity of Prim's algorithm. Analysis complexity of Prim's algorithm using Greedy approach.
(Ans. : Refer section 5.5.12) (7 Marks)

Q. Explain prim's algorithm.
(Ans. : Refer section 5.5.2) (6 Marks)

Ex. 5.5.4 (5 Marks) **(May 2017)**

Q. Write Kruskal's algorithm for finding a minimum spanning tree. Compute the time complexity of the same. *(Ans. : Refer section 5.5.3)*(7 Marks)

Q. Write Kruskal's algorithm to find a minimum spanning tree.
(Ans. : Refer section 5.5.3) (7 Marks)

Ex. 5.5.7 (10 Marks) **(Dec. 2014)**

Ex. 5.5.8 (10 Marks) **(May 2016)**

Q. Compare Prim's algorithm and Kruskal's algorithm for finding the minimum spanning tree.
(Ans. : Refer section 5.5.4) (6 Marks)

☞ **Syllabus Topic : Optimal Storage on Tapes**

Q. Write short note on optimal storage on tapes.
(Ans. : Refer section 5.6) (10 Marks) **(May 2013)**

Q. Explain optimal storage on tape with example.
(Ans. : Refer section 5.6)(10 Marks) **(Dec. 2014)**

Q. How to solve optimal storage problems using greedy approach? *(Ans. : Refer section* 5.6) (6 Marks)

Q. Explain an algorithm to derive optimal storage schedule using greedy approach for a single tape.
(Ans. : Refer section 5.6.1) (10 Marks)

Ex. 5.6.1 (10 Marks) **(Dec. 2015)**

Q. Explain greedy method to solve the problem of storage on multiple tapes.
(Ans. : Refer section 5.6.2) (3 Marks)

Q. With a suitable example, discuss the storage on multi-tape problem using greedy approach.
(Ans. : Refer section 5.6.2) (3 Marks)

❑❑❑

Module 4

CHAPTER 6

Backtracking & Branch and Bound

Syllabus Topics

General Method, 8 queen problem (N-queen problem), Sum of subsets, Graph coloring, 15 puzzle problem, Travelling salesman problem.

6.1 Backtracking

Syllabus Topic : General Method

6.1.1 General Method

6.1.1(A) Introduction

→ (May 14)

Q. Comment on model of computation : Backtracking.
MU - May 2014, 5 Marks

- Solution to many problems can be viewed as a making sequence of decisions. For example, TSP can be solved by making the sequence of the decision of which city should be visited next.
- Similarly, knapsack is also viewed as a sequence of the decision. At each step, the decision is made to select or reject the given item.
- Backtracking is an intelligent way of gradually building the solution. Typically, it is applied to constraint satisfaction problems like Sudoku, crossword, 8-queen puzzles, chess and many other games.
- Backtracking builds the solution incrementally. Partial solutions which do not satisfy the constraints are abandoned.

Backtracking is recursive approach and it is a refinement of brute force method. Brute force approach evaluates all possible candidates, whereas backtracking limits the search by eliminating the candidates that do not satisfy certain constraints. Hence, backtracking algorithms are often faster than brute force approach.

- Backtracking algorithms are used when we have set of choices, and we don't know which choice will lead to a correct solution. The algorithm generates all partial candidates that may generate a complete solution.
- The solution in backtracking is expressed as n-tuple $(x_1, x_2, \ldots, x_n)$, where x_i is chosen from the finite set of choices, S_i. Elements in solution tuple are chosen such that it maximize or minimize given criterion function $C(x_1, x_2, \ldots, x_n)$.
- Let C be the set of constraints on problem P, and let D is the set of all the solutions which satisfy the constraints C, then
  - Finding if given solution is feasible or not? – is the *decision problem.*
  - Finding best solution – is the *optimization problem.*
  - Listing all feasible solution – is *enumeration problem.*
- Backtracking systematically searches set of all feasible solutions called **solution space** to solve the given problem.
- Each choice leads to a new set of partial solutions. Partial solutions are explored in DFS (Depth First Search) order.
- If partial solution satisfies certain **bounding function** than the partial solution is explored in depth-first order.
- If the partial solution does not satisfy the constraint, it will not be explored further. Algorithm backtracks from that point and explores the next possible candidate.
- Such processing is convenient to represent using state space tree. In state space tree, root represents the initial state before the search begins.
- Fig. 6.1.1 shows the working of backtracking algorithms.
- It keeps exploring the partial solution by adding next possible choice in DFS order and builds the new partial solution. This process continues till partial solution satisfies the given constraints or solution is not found. This is an incremental approach.

A node in state space tree is called **promising** if it represents partially constructed solution which may lead to a complete solution. **Non-promising** node violates constraints and hence cut down from the further computation.

A leaf node is either a non-promising node or it represents a complete solution.

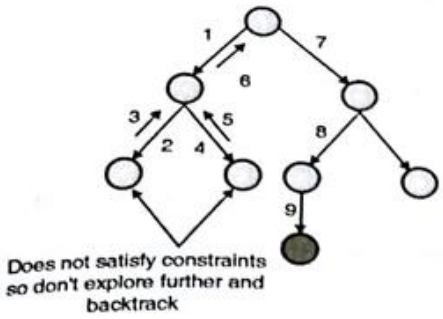


Fig. 6.1.1 : Process of backtracking

We can consider the backtracking process as finding a particular leaf in the tree. It works as follows :

- If node N is goal node, then return success and exit.
- If node N is leaf node but node goal node, then returns failure.
- Otherwise, for each child C of node N
  - Explore child node C
  - If C is goal node, return "success"
  - Return failure
- In backtracking, the solution is expressed in form of tuple $(x_1, x_2, \ldots, x_n)$, each $x_i \in S_i$, where S_i is some finite set of choices.
- Backtracking algorithm tries to maximize or minimize certain criterion function f(.) by selecting or not selecting item x_i.
- While solving the problem, backtracking method imposes two constraints :
  - **Explicit constraints :** It restricts the section of x_i from S_i. Explicit constraints are problem dependent. All tuples from solution space must satisfy explicit constraints.
  - **Implicit constraints :** Such constraints determine which tuple in solution space satisfies the criterion function.

6.1.1(B) Terminology

Q. Write a short note on state space tree. **(5 Marks)**

Q. What is a state space tree and with respect to state space tree explain the following terms:

(i) solution state
(ii) state space
(iii) answer states
(iv) static trees
(v) dynamic trees
(vi) live node
(vii) bounding function
(viii) state space tree
(ix) E-node. **(10 Marks)**

- The **state space** of the system is the set of states in which the system can exist.
- Solution to the problem which is derived by making sequence of decisions can be represented using **state space tree** T.
- The root of the tree is the state before making any decision. Nodes at the first level in the tree corresponding to all possible choices for the first decision.
- Nodes at second level corresponding to all possible choices for the second decision and so forth.
- Usually, number of decision sequence is in exponential order of input size n.
- State space for the 8-puzzle problem is shown in the Fig. 6.1.2.

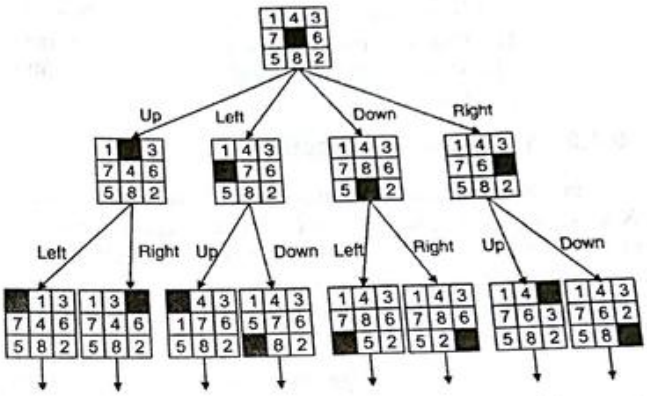


Fig. 6.1.2 : State space tree of 8-puzzle problem

- Backtracking guaranteed to find the solution to any problem modeled by a state space tree.
- Based on traversal order of the tree, two strategies are defined to solve the problem.
- Backtracking traverse the tree in depth-first order.
- Branch and bound traverse the tree in breadth-first order. As backtracking always explores the current node first, there is no need of explicitly implementing the state space tree.
- Some terminologies related to state space tree are discussed here :
  - (1) **Solution space :** Collection of all feasible solution is called solution space.
  - (2) **State space :** All the paths in the tree from root to other nodes form the state space of given problem.
  - (3) **Problem state :** Each node in the state space tree represents problem state.
  - (4) **Answer state :** Answer state is the leaf in the state space tree which satisfies the implicit constraints.

(5) **Solution-state :** State S for which the path from the root to S represents a tuple in the solution space is called solution state.

(6) **Live node :** In state space tree, a node which is generated but its children are yet to be generated is called live node.

(7) **E-Node :** Live node whose children are currently being explored is called E (Expansion) node.

(8) **Dead node :** In state space tree, a node which is generated and either it's all children are generated or node will not be expanded further due to a violation of criterion function, is called dead node.

(9) **Bounding Functions :** Bounding function kills the live node without exploring its children if the bound value of live node crosses the bound limits.

(10) **Static tree :** If tree is independent of instance of problem being solved, it is called static tree

(11) **Dynamic tree :** If tree depends on an instance of the problem being solved, it is called dynamic tree.

6.1.2 Control Abstraction

In backtracking, solution is defined as n-tuple $X = (x_1, x_2, \ldots, x_n)$, where $x_i = 0$ or 1. x_i is chosen from set of finite components S_i. If component x_i is selected then set $x_i = 1$ else set it to 0. Backtracking approach tries to find the vector X such that it maximize or minimize certain criterion function $P(x_1, x_2, \ldots, x_n)$.

Control abstraction for backtracking approach is shown below :

6.1.2(A) Recursive Backtracking Method

Q. Write a formulation of the recursive backing algorithm. **(5 Marks)**

Q. Write recursive backtracking algorithm for the sum of subset problem. **(6 Marks)**

- Backtracking is rather a typical recursive algorithm. Problems solved using backtracking are often solved using recursion. General algorithm discussed below finds all answer nodes, not just one.
- Let $(x_1, x_2, \ldots, x_k)$ be the path from the root to the i^{th} node. Let $T(x_1, x_2, \ldots, x_k)$ be the set all of all possible values for next node x_{k+1} such that $(x_1, x_2, \ldots, x_{k+1})$ is also a path from the root to a problem state.

Recursive approach for backtracking is stated below.

```
Algorithm REC_BACKTRACK(k)
// X[1...k – 1] is the solution vector
// T(x[1], x[2],..., x[K - 1]) is the state space tree
// Bk( ) is the bounding function

for each x[k] ∈ T[x[1], x[2], ..., x[k – 1]] do
    if (Bk(x[1], x[2], ..., x[k – 1])) == TRUE then
                        // (Represents feasible solution)
    if (x[1], x[2], ..., x[k]) is path to answer node then
        print (x[1], x[2], ..., x[k])
    end
    if k < n then
        REC_BACKTRACK(k + 1)
    end
    end
end
```

6.1.2(B) Iterative Backtracking Method

Q. What is backtracking? Write a general iterative algorithm for backtracking. **(7 Marks)**

Iterative approach for back tracking method is described below:

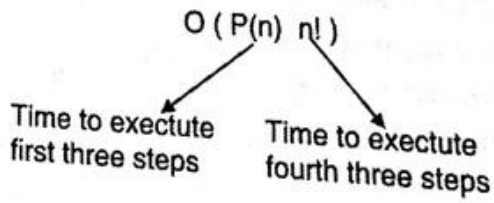
```
Algorithm ITERATIVE_BACKTRACK(n)
// X[1...k – 1] is the solution vector
// T(x[1], x[2],..., x[K - 1]) is the state space tree
// Bk( ) is the bounding function

k ← 1
while k ≠ 0 do
    if (untried x[k] ∈ T[x[1], x[2], ...x[k – 1]]) AND
        (Bk(x[1], x[2], ..., x[k]) is path to answer node) then
        print x[1], x[2],...x[k] // Solution
        k ← k + 1                    // Consider next candidate in set
    else
        k ← k – 1                    // Backtrack to recent node
    end
end
```

Performance analysis

The performance of backtracking algorithm depends on four parameters :

1. Time to compute the tuple x[k]
2. Number of x[k] which satisfy explicit constraint
3. Time taken by bounding function Bk to generate feasible sequence
4. A number of x[k] which satisfy the bounding function Bk for all k.

First three factors are independent and they can be computed in polynomial time. Whereas the last factor is problem dependent. In the worst case, checking of x[k] can be done in factorial time because the tree might have n! nodes. Time complexity of backtracking method is given as,



6.1.3 Applications of Backtracking

Backtracking is useful in solving following problems :

(i) N-Queen problem
(ii) Sum of subset problem
(iii) Graph coloring problem
(iv) Knapsack problem
(v) Hamiltonian cycle problem
(vi) Games like chess, tic-tac-toe, Sudoku etc.
(vii) Constraint satisfaction problems
(viii) Artificial intelligence
(ix) Network communication
(x) Robotics
(xi) Optimization problems
(xii) It is also a basis of a logic programming language called PROLOG.

Syllabus Topic : 8-Queen Problem

6.1.4 8-Queen Problem

➔ **(May 14, Dec. 14, Dec. 15, May 16, Dec. 16)**

Q Write note on : N-Queen Problem.
MU - May 2014, Dec. 2014, 10 Marks

Q. Explain 8 Queen problem.
MU - Dec. 2015, 10 Marks

Q. Write a short note on 8-Queen problem.
MU - May 2016, Dec. 2016, 10 Marks

- **Problem :** Given 8 × 8 chess board, arrange 8 queens in a way such that no two queens attack each other.
- Two queens are attacking each other if they are in the same row, column or diagonal. Cells attacked by the queen Q is shown in the Fig. 6.1.3.

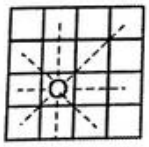


Fig. 6.1.3 : Attacked cells by queen Q

- 8 queen problem has $^{64}C_8$ = 4,42,61,65,368 different arrangements, out of these only 92 arrangements are valid solution. Out of which, only 12 are the fundamental solution, rest of 80 solutions can be generated using reflection and rotation.
- 2-Queen problem is not feasible. Minimum problem size for which solution can be found is 4. Let us understand the working of backtracking on the 4-queen problem.
- For simplicity, partial state space tree is shown in the Fig. 6.1.4. Queen 1 is placed in a 1st column in the 1st row. All the positions are crossed in which queen 1 is attacking. In next level, queen 2 is placed in a 3rd column in row 2 and all cells are crossed which are attacked by already placed queens 1 and 2. As can be seen from Fig. 6.1.4, no place is left to place next queen in row 3, so queen 2 backtracks to next possible position and process continue.
- In a similar way, if (1, 1) position is not feasible for queen 1, then algorithm backtracks and put the first queen in cell (1, 2), and repeats the procedure. For simplicity, only a few nodes are shown in the Fig. 6.1.4.

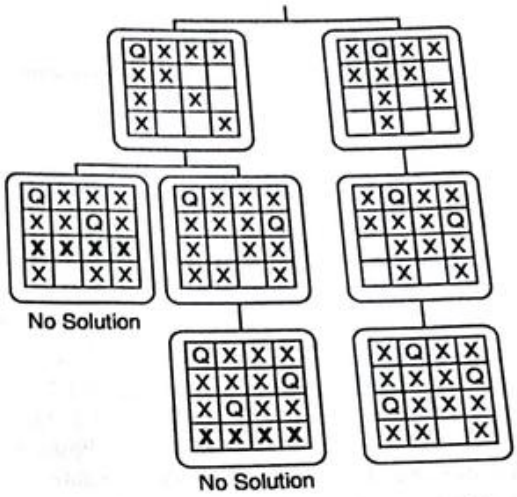


Fig. 6.1.4 : Snapshot of backtracking procedure of 4-Queen problem

- Complete state space tree for the 4-queen problem is shown in the Fig. 6.1.5.
- The number within the circle indicates the order in which the node gets explored. The height of node from root indicates row and label besides arc indicate that the Q is placed in an i^{th} column. Out of all possible states, few are the answer state.

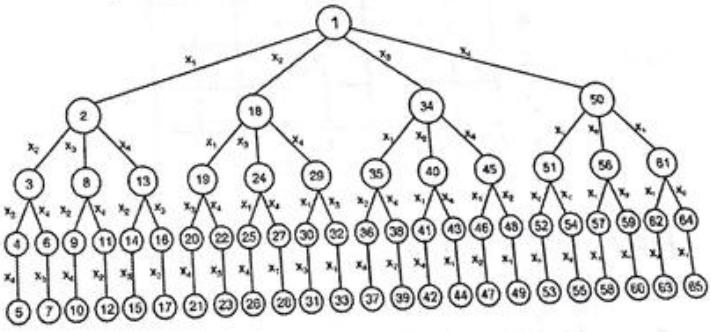


Fig. 6.1.5

- Algorithm halts when the first full solution is found in solution tree. The 4-queen problem can be easily generalized to 8 queen or N-queen problems. One instance of solution for the 8-queen problem is shown in Fig. 6.1.6.

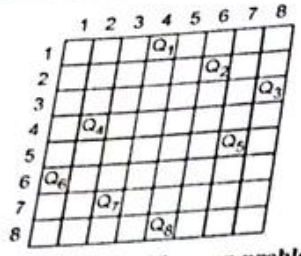


Fig. 6.1.6 : Solution of 8-queen problem

- *Solution tuple for the solution shown in Fig. 6.1.6 is defined as <4, 6, 8, 2, 7, 1, 3, 5>. From observations, two queens placed at (i, j) and (k, l) positions, can be in same diagonal only if,*

$(i - j) = (k - l)$ or

$(i + j) = (k + l)$

From first equality, $j - l = i - k$

From second equality, $j - l = k - i$

So queens can be in diagonal only if $|j - l| = |i - k|$.

The arrangement shown in the Fig. 6.1.7 leads to failure. As it can be seen from the Fig. 6.1.6. Queen Q6 cannot be placed anywhere in the 6th row. So Position of Q5 is backtracked and it is placed in another feasible cell. This process is repeated until the solution is found.

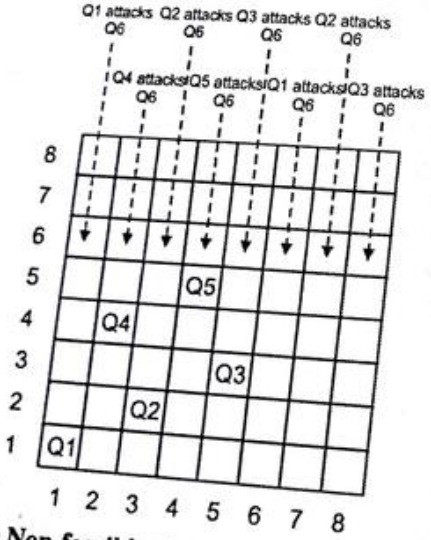


Fig. 6.1.7 : Non-feasible state of the 8-queen problem

Following algorithm arranges n queens on n × n board using a backtracking algorithm.

```
Algorithm N_QUEEN (k, n)
// Description : To find the solution of n x n queen problem using
    backtracking
// Input : n: Number of queen
    k: Number of the queen being processed currently, initially
    set to 1.
// Output : n x 1 Solution tuple
for i ← 1 to n do                          Process 1 to n queen
    if PLACE(k , i) then                   Returns true if kth queen
    x[k] ← i                               can be placed in ith row
    if k == n then
    print X[1...n]                         If all queens are
    else                                   processed, then
    N_QUEEN(k + 1, n)                      display solution tuple
    end
    end
end
```

- Function PLACE (k, i) returns true, if the k^{th} queen can be placed in i^{th} column. This function enumerates all the previously kept queen's positions to check if two queens are on same diagonal. It also checks that i is distinct from all previously arranged queens.

```
Function PLACE(k, i)
// k is the number of queen being processed
// i is the number of columns
                                   Loop through placed queens.
                                   (k – 1) queens are already placed
                                   and kth queen is being processed
for j ← 1 to k – 1 do
                                   ith and jth queen are in
                                   same column
if x[j] == i OR ((abs(x[j]) - i) == abs(j - k)) then
                                   ith and jth queen are in
return false                       same diagonal
    end
end
return true                        Return true if kth queen is not attacked
                                   by any of previous (k - 1) queens
```

- Function abs(a) returns absolute value of argument a. The array X is the solution tuple.
- Like all optimization problem, n queen problem also has some constraints, which it must satisfy. These constraints are divided into two categories : Implicit and explicit constraints

Explicit constraints

- Explicit constraints are the rules that allow/disallow selection of x_i to take value from the given set. For example, $x_i = 0$ or 1.

  $x_i = 1$ if $LB_i \le x_i \le UB_i$

  $x_i = 0$ otherwise

- **Solution space** is formed by the collection of all tuple which satisfies the constraint.

Implicit constraints

- The implicit constraint is to determine which of the tuple of solution space satisfies the given criterion functions. The implicit constraint for n queen problem is that two queens must not appear in the same row, column or diagonal.
- **Complexity analysis :** In backtracking, at each level branching factor decreases by 1 and it creates a new problem of size $(n - 1)$. With n choices, it creates n different problems of size $(n - 1)$ at level 1.
- PLACE function determines the position of the queen in O(n) time. This function is called n times.
- Thus, the recurrence of n-Queen problem is defined as, $T(n) = n*T(n - 1) + n^2$. Solution to recurrence would be O(n!).

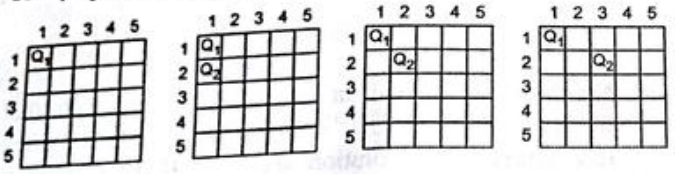
Ex. 6.1.1

Find all possible solutions for five queen problem using backtracking approach.

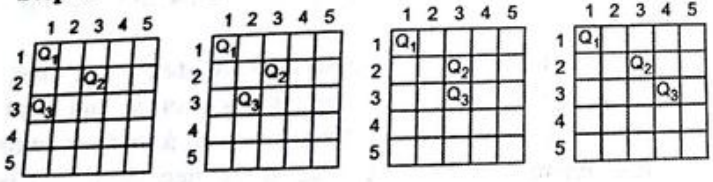
Soln. :

Solution of N queen problem is represented using n-tuple $X = [x_1, x_2, x_3, \ldots, x_n]$. Each $x_i = 1, 2, \ldots, n$.

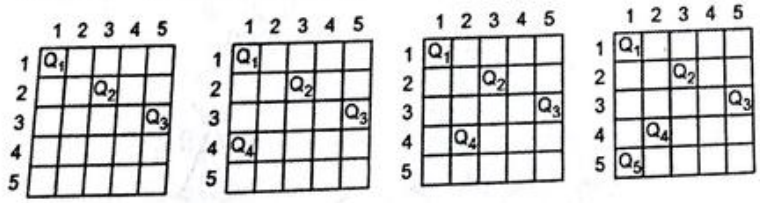
If queen Q_i can be placed successfully in column j, then set $x_i = j$. Q_i is always placed in row i.



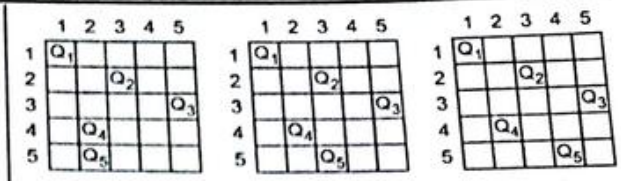
Step 1: Place Q_1 on (1, 1) → Successful
Step 2: Place Q_2 on (2, 1) → Fail → Backtrack
Step 3: Place Q_2 on (2, 2) → Fail → Backtrack
Step 4: Place Q_2 on (2, 3) → Successful



Step 5: Place Q_3 on (3, 1) → Fail → Backtrack
Step 6: Place Q_3 on (3, 2) → Fail → Backtrack
Step 7: Place Q_3 on (3, 3) → Fail → Backtrack
Step 8: Place Q_3 on (3, 4) → Fail → Backtrack



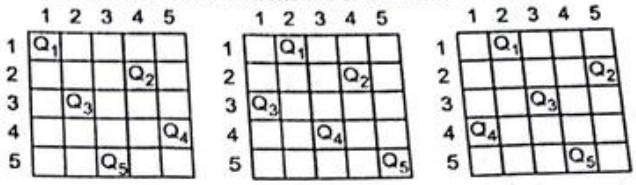
Step 9: Place Q_3 on (3, 5) → Successful
Step 10: Place Q_4 on (4, 1) → Fail → Backtrack
Step 11: Place Q_4 on (4, 2) → Successful
Step 12: Place Q_5 on (5, 1) → Fail → Backtrack



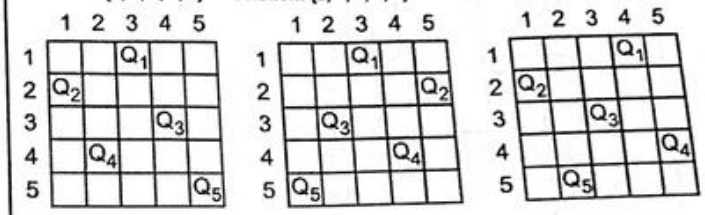
Step 13: Place Q_5 on (5, 2) → Fail → Backtrack
Step 14: Place Q_5 on (5, 3) → Fail → Backtrack
Step 15: Place Q_5 on (5, 4) → Successful

Thus, the solution of this instance is {1, 3, 5, 2, 4}.

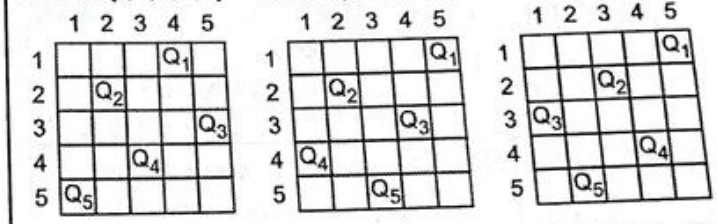
Few more combinations are shown below :



Solution: {1, 4, 2, 5, 3} **Solution: {2, 4, 1, 3, 5}** **Solution: {2, 5, 3, 1, 4}**



Solution: {3, 1, 4, 2, 5} Solution: {3, 5, 2, 4, 1} Solution: {4, 1, 3, 5, 2}



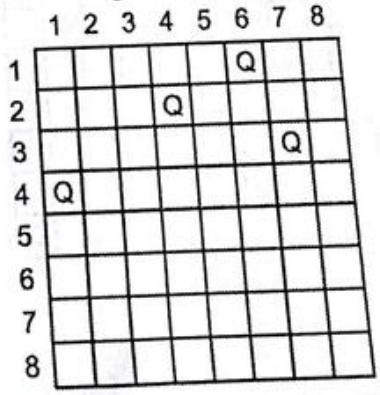
Solution :{4, 2, 5, 3, 1} Solution: {5, 2, 4, 1, 3} Solution: {5, 3, 1, 4, 2}

Ex. 6.1.2

Current configuration is (6, 4, 7, 1) for 8-queens problem. Find answer tuple.

Soln. :

- Tuple (6, 4, 7, 1) indicates the first queen is in the 6th column, the second queen is in the 4th column, the third queen is in the 7th column and forth queen is in the 1st column. Given arrangement of queen is,



- Assume queen Q_1 is placed on position (i, j) and Q_2 is placed on position (k, l). Q_1 and Q_2 would be on same diagonal if they satisfy following condition:

  $i + j = k + l$ OR $i - j = k - l$

- We cannot place next queen on position (5, 1) as one queen is already in the same column. If we place next queen on (5, 2) then,

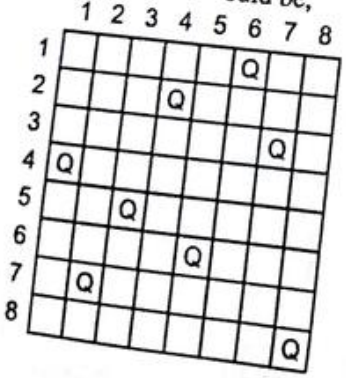
  Let Q_4 = (4, 1) → position of queen in row 4

  Q_5 = (5, 2) → position of queen in row 5

- For these two queens, $4 - 1 = 5 - 2$, so they are in same diagonal, so we cannot place next queen on position (5, 2). Try for next location (5, 3) and repeat the procedure for all possible positions.

	Queen position i (row) →								Action
	1	2	3	4	5	6	7	8	
	6	4	7	1					Initial configuration
	6	4	7	1	2				Column 1 is already occupied Let us test position (5, 2) 4 – 1 = 5 – 2, Conflict
	6	4	7	1	3				Let us test position (5, 3) No conflict, so place it on (5, 3)
	6	4	7	1	3	2			Column 1 is already occupied Let us test position (6, 2) 5 + 3 = 6 + 2, Conflict
j (col) ↓	6	4	7	1	3	5			Column 3 and 4 are already occupied Let us test position (6, 5) No conflict, so place it on (6, 5)
	6	4	7	1	3	5	2		Column 1 is already occupied Let us test position (6, 2) No conflict, so place it on (7, 2)
	6	4	7	1	3	5	2	8	Column 1 to 7 are already occupied Let us test position (8, 8) No conflict, so place it on (8, 8)

Thus, final board position would be,



Syllabus Topic : Sum of Subsets

6.1.5 Sum of Subset Problem

→ (Dec. 15)

Q. Explain sum of subset problem. **MU - Dec. 2015, 5 Marks**

Q. Write a recursive backtracking algorithm for sum of subset problem. **(5 Marks)**

Problem

Given a set of positive integers, find the combination of numbers that sum to given value M.

Sum of subset problem is analogs to knapsack problem. Knapsack problem tries to fill the knapsack using given set of items to maximize the profit. Items are selected in a way that total weight in knapsack does not cross the capacity of the knapsack. Inequality condition in knapsack problem is replaced by equality in the sum of subset problem.

Given the set of n positive integers $W = \{w_1, w_2, \ldots, w_n\}$,

and given a positive integer M, the sum of subset problem can be formulated as follows (where w_i and M correspond to item weights and knapsack capacity in knapsack problem) :

$$\sum_{i=1}^{n} w_i x_i = M \qquad \text{Where, } x_i \in \{0, 1\}.$$

- Numbers are sorted in ascending order, such that $w_1 < w_2 < w_3 < \ldots < w_n$. The solution is often represented using solution vector X. If the i^{th} item is included, set x_i to 1 else set it to 0. In each iteration, one item is tested. If inclusion of an item does not violet the constraint of the problem, add it.
- Otherwise, backtrack, remove the previously added item and continue the same procedure for all remaining items.
- The solution is easily described by state space tree. Each left edge denotes the inclusion of w_i and right edge denotes exclusion of w_i. Any path from the root to leaf forms a subset. A state space tree for n = 3 is demonstrated in the Fig. 6.1.8.

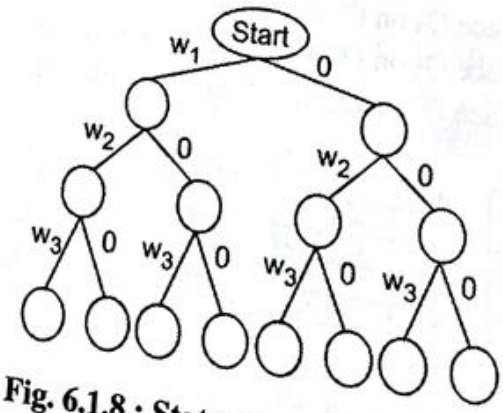


Fig. 6.1.8 : State space tree for n = 3

Algorithm for solving sum of subset problem using recursion is stated below.

Algorithm

→ **(May 14, May 15, May 16)**

Q. Write an algorithm of sum of subsets.
MU - May 2014, May 2015, May 2016, 5 Marks

Algorithm for subset sum problem :

```
Algorithm SUB_SET_PROBLEM(i, sum, W, remSum)
// Description : Solve sub of subset problem using backtracking
// Input : W: Number for which subset is to be computed
           i: Item index
           sum: Sum of integers selected so far
           remSum: Size of remaining problem i.e. (W – sum)
// Output : Solution tuple X

if FEASIBLE_SUB_SET(i) == 1 then        (If addition of number dose not cross W)
    if (sum == W) then
    print X[1...i]                      (Sum of selected integer is same as W, so problem is solved, print solution)
    end
else
    X[i + 1] ← 1
    SUB_SET_PROBLEM(i + 1, sum + w[i] + 1, W,
    remSum – w[i] + 1 )                 (Include i^th item to solution set and proceed)

    X[i + 1] ← 0        // Exclude the i^th item
    SUB_SET_PROBLEM(i + 1, sum, W, remSum – w[i] + 1 )
end                                     (Exclude i^th item to solution set and proceed)

Function FEASIBLE_SUB_SET(i)
if (sum + remSum ≥ W) AND (sum == W)
or (sum + w[i] + 1 ≤ W) then
    return 0
end
return 1
```

First recursive call represents the case when the current item is selected, and hence the problem size is reduced by *w[i]*.

Second recursive call represents the case when we do not select the current item.

Complexity Analysis : It is intuitive to derive the complexity of sum of subset problem. In state space tree, at level i, the tree has 2^i nodes. So given n items, total number of nodes in tree would be $1 + 2 + 2^2 + 2^3 + .. 2^n$.

$$T(n) = 1 + 2 + 2^2 + 2^3 + .. 2^n = 2^{n+1} - 1 = O(2^n)$$

Thus, sum of sub set problem runs in exponential order.

Let us try to understand the problem using an example.

Ex. 6.1.3

Consider the sum-of-subset problem, n = 4, Sum = 13, and $w_1 = 3$, $w_2 = 4$, $w_3 = 5$ and $w_4 = 6$. Find a solution to the problem using backtracking. Show the state-space tree leading to the solution. Also, number the nodes in the tree in the order of recursion calls.

Soln. :

The correct combination to get the sum of M = 13 for given W = {3, 4, 5, 6} is {3, 4, 6}. Solution vector for {3, 4, 6} would be X = {1, 1, 0, 1} because element 5 is not chosen so X[3] = 0. Let derive the solution using backtracking. Numbers in W are already sorted.

Set X = {0, 0, 0, 0}

Set Sum = 0. Sum indicates summation of selected numbers from W.

Step 1 : i = 1, Adding item w_i

Sum = Sum + w_i = Sum + w_1 = 0 + 3 = 3

Sum ≤ M, so add item i to solution set.

X[i] = X[1] = 1 ⇒ X = {1, 0, 0, 0}

Step 2 : i = 2, Adding item w_2

Sum = Sum + w_i = Sum + w_2 = 3 + 4 = 7

Sum ≤ M, so add item i to solution set.

X[i] = X[2] = 1 ⇒ X = {1, 1, 0, 0}

Step 3 : i = 3, Adding item w_3

Sum = Sum + w_i = Sum + w_3 = 7 + 5 = 12

Sum ≤ M, so add item i to solution set.

X[i] = X[3] = 1 ⇒ X = {1, 1, 1, 0}

Step 4 : i = 4, Adding item w_4

Sum = Sum + w_i = Sum + w_4 = 12 + 6 = 18

Sum > M, so backtrack and remove previously added item from solution set.

X[i] = X[3] = 0 ⇒ X = {1, 1, 0, 0}.

Update Sum accordingly. So, Sum = Sum – w_3

= 12 – 5 = 7

And don't increment i.

Step 5 : i = 4, Adding item w_4

Sum = Sum + w_i = Sum + w_4 = 7 + 6 = 13

Sum = M, so solution is found and add item i to solution set.

X[i] = X[4] = 1 ⇒ X = {1, 1, 0, 1}

Complete state space tree for given data is shown in Fig. P. 6.1.3.

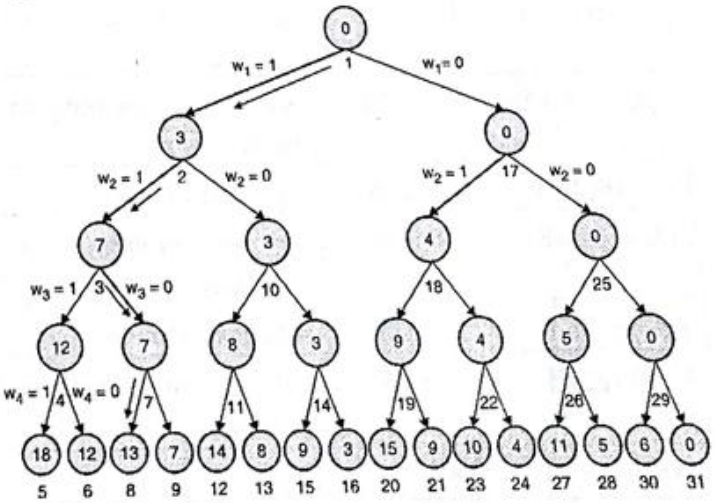


Fig. P .6.1.3

At level i, left branch corresponds to the inclusion of number w_i and right branch corresponds to exclusion of number w_i. Recursive call number for the node is stated below the node. Node 8 is the solution node. The bold solid line shows the path to the output node.

Ex. 6.1.4 **MU - May 2014, 5 Marks**

Analyze sum of subset algorithm on data :

M = 35 and

i) w = {5, 7, 10, 12, 15, 18, 20}

ii) w = {20, 18, 15, 12, 10, 7, 5}

iii) w = {15, 7, 20, 5, 18, 10, 12}

Are there any discernible differences in the computing time ?

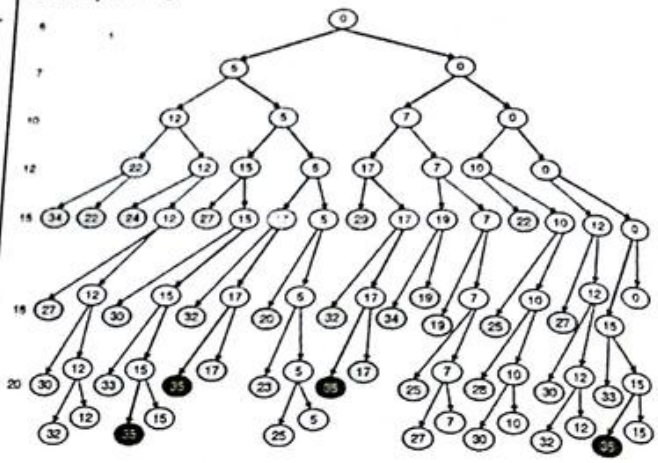
Soln. :

Let us run the algorithm on first instance

w = {5, 7, 10, 12, 15, 18, 20}.

Items in sub set	Condition	Comment
{ }	0	Initial condition
{ 5 }	5 < 35	Select 5 and Add next element
{ 5, 7 }	12 < 35	Select 7 and Add next element
{ 5, 7, 10 }	22 < 35	Select 20 and Add next element
{ 5, 7, 10, 1}	34 < 35	Select 12 and Add next element
{5,7, 10, 12,}	49 > 35	Sum exceeds M, so backtrack and remove 12
{5,7, 10, 15 }	37 > 35	Sum exceeds M, so backtrack and remove 15
{ 5,7, 12 }	24 < 35	Add next element
{ 5,7, 2, 15}	39 > 35	Sub set sum exceeds, so backtrack
{ 5, 10 }	15 < 35	Add next element
{ 5, 10, 12 }	27 < 35	Sub set sum exceeds, so backtrack
{5,10,12, 15}	42 > 35	Sub set sum exceeds, so backtrack
{ 5, 10, 15 }	30 < 35	Add next element
{5, 0, 15, 18}	48 > 35	Sub set sum exceeds, so backtrack
{ 5, 10, 18 }	33 < 35	Add next element
{5,10,18,20}	53 > 35	Sub set sum exceeds, so backtrack
{ 5, 10, 20 }	35	Solution Found

There may exist multiple solutions. State space tree for above sequence is shown here: Number on leftmost column shows the element under consideration. Left and right branch in tree indicate inclusion and exclusion of the corresponding element on that level respectively.



Numbers in leftmost column indicates element under consideration at that level. Gray circle indicates the node which cannot accommodate any of the next value, so we will cut sort them from further expansion. White leaves do not lead to a solution. An intermediate white node may or may not lead to a solution. The black circle is the solution states.

We get the four solutions:

1. {5, 10, 20} 2. {5, 12, 18} 3. {7, 10, 18} 4. {15, 20}

For efficient execution of subset sum problem, input element should be sorted in non-decreasing order. If elements are not in non-decreasing order, the algorithm does more backtracking. So second and third sequence would take more time for execution and may not find as many solutions as we get in the first sequence.

Ex. 6.1.5

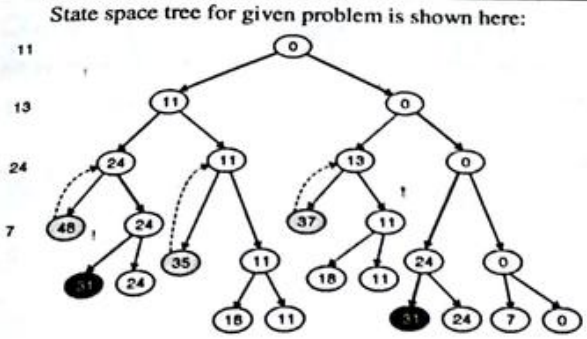
Solve the sum of subset problems using backtracking algorithmic strategy for the following data n = 4

$W = (w_1, w_2, w_3, w_4) = (11, 13, 24, 7)$ and M = 31.

Soln. :

Items in sub set	Condition	Comment
{ }	0	Initial condition
{ 11 }	11 < 31	Add next element
{ 11, 13 }	24 < 31	Add next element
{ 11, 13, 24 }	48 < 31	Sub set sum exceeds, so backtrack
{11,13,7}	31	Solution Found

State space tree for given problem is shown here:



In above graph, the black circle shows the correct result. Gray node shows from where algorithm backtracks. Numbers in leftmost column indicates element under consideration at that level. Left and right branch represent inclusion and exclusion of that element respectively.

We get two solutions :

1. {11, 13, 7} 2. {24, 7}

Ex. 6.1.6

Let W = {5, 10, 12, 13, 15, 18}, M = 30. Find all possible subsets of W that sum to M. Draw the portion of state space tree that is generated.

Soln. :

In state space tree, left branch follows inclusion of number and right branch follows exclusion of number. All numbers are sorted in increasing order, so if partial sum after adding i^{th} number exceeds M, then the addition of any subsequent numbers form (i + 1) to n definitely exceeds M. So the node where partial sum exceeds M is pruned immediately. No need to generate state space tree from that node onwards. The algorithm will backtrack from that point and explores next child of its parent. This process will continue until all numbers are checked.

State space tree for given problem is depicted in the Fig. P. 6.1.6. Adding next number to a gray node exceeds M, so the tree is pruned at that node. Black nodes correspond to answer node.

Fig. P. 6.1.6 : State space tree for W = {5, 10, 12, 13, 15, 18} and M = 30.

Possible subsets and their solution tuple are:

{5, 10, 15} → X = [1, 1, 0, 0, 1, 0]

{5, 12, 13} → X = [1, 0, 1, 1, 0, 0]

{12, 18} → X = [0, 0, 1, 0, 0, 1]

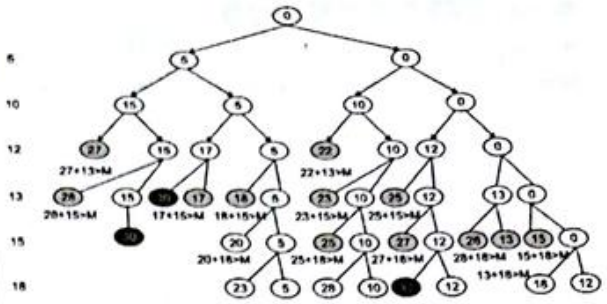


Fig. P. 6.1.6

Ex. 6.1.7 MU - May 2013, 10 Marks

Solve following sum of subset problem using backtracking:

w = { 1, 3, 4, 5 } ; m = 8

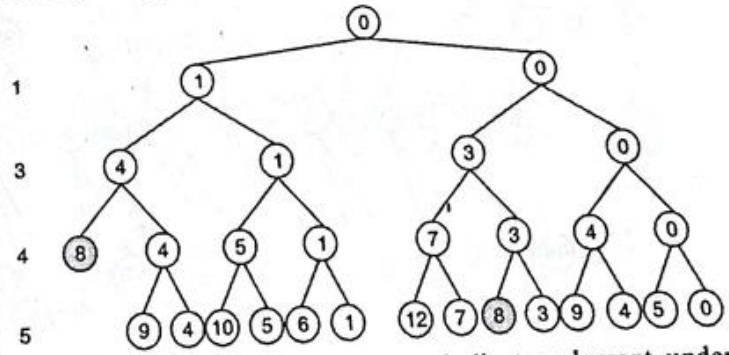
Find all possible subsets of 'w' that sum to 'm'

Soln. :

Let us run the algorithm on first instance w = {5, 7, 10, 12, 15, 18, 20}.

Items in sub set	Condition	Comment
{ }	0	Initial condition
{ 1 }	1 < 8	Select 1 and Add next element
{ 1, 3 }	4 < 8	Select 3 and Add next element
{ 1, 3, 4 }	8 = 8	Select 8 and Add next element

There may exist multiple solutions. State space tree for above sequence is shown here: Number on leftmost column shows the element under consideration. Left and right branch in tree indicate inclusion and exclusion of the corresponding element on that level respectively.



– Numbers in leftmost column indicates element under consideration at that level. The black circle is the solution states.

We get two solutions :

1. {1, 3, 4} 2. {3, 5}

Ex. 6.1.8 MU - May 2015, 10 Marks

Write and explain sum of subset algorithm for : n = 5, W = {2, 7, 8, 9, 15) M = 17

Soln. : State space tree for the given problem is shown in Fig. P. 6.1.8.

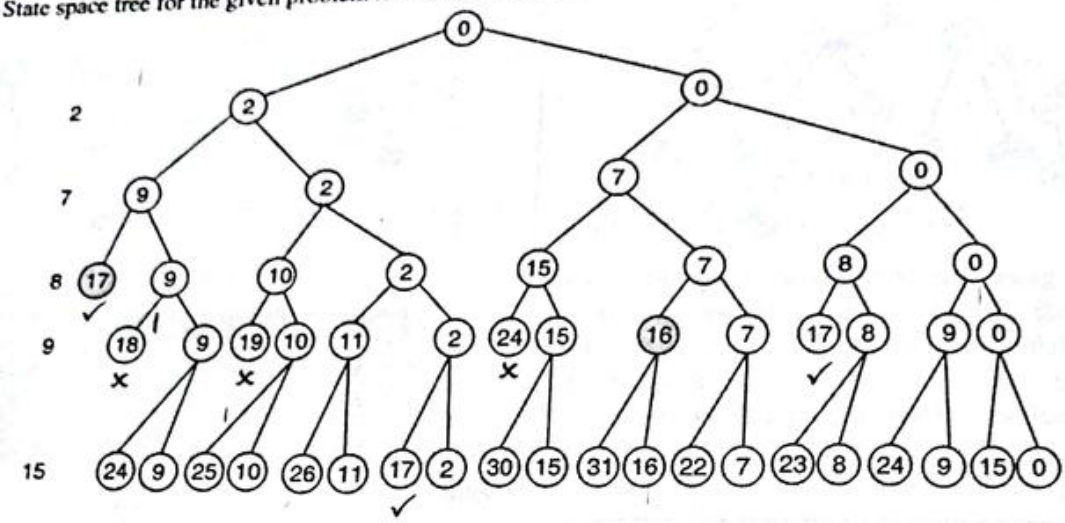


Fig. P. 6.1.8

The possible solutions are {2, 7, 8}, {2, 15} and {8, 9}

Ex. 6.1.9 MU - Dec. 2015, May 2016, 5 Marks

Find all possible subsets of weight that sum to m, let n = 6, m = 30, and w[1 : 6] = {5, 10, 12, 13, 15, 18}

Soln. :

- State space tree for the given problem is shown in Fig. P. 6.1.9.

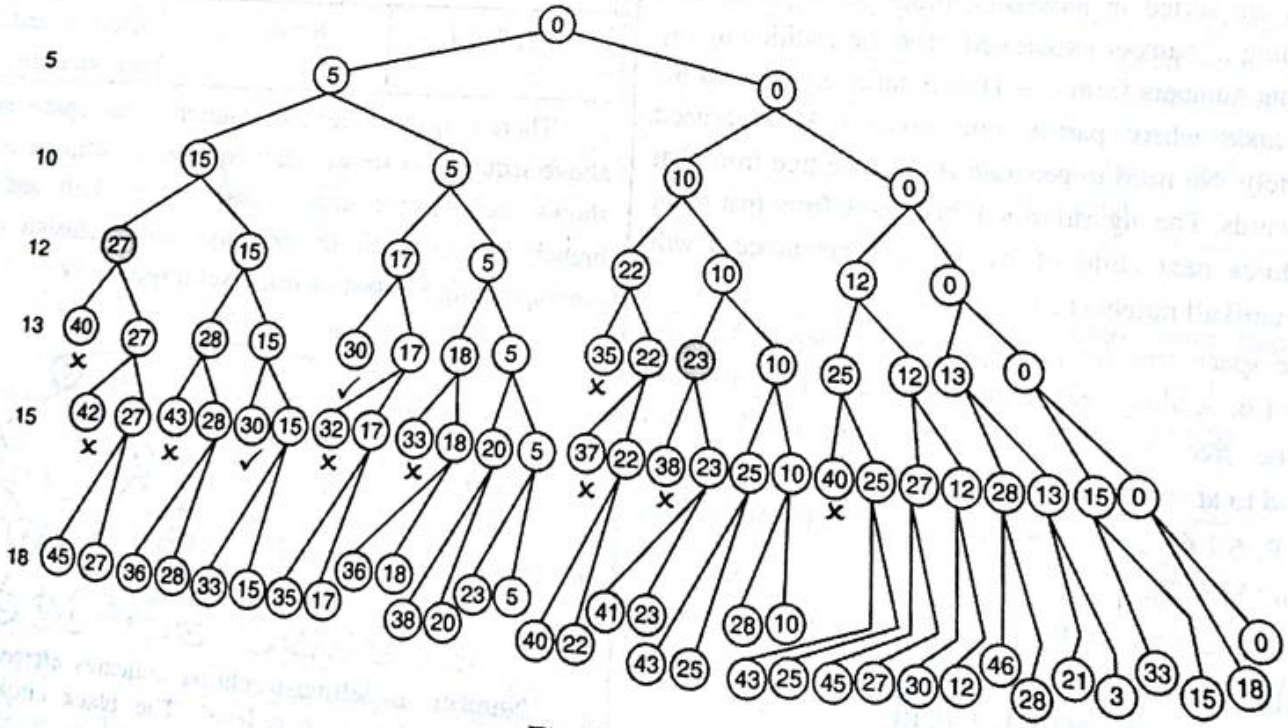


Fig. P. 6.1.9

- The possible solutions are {5, 12, 13}, {12, 18} and {5, 10, 15}

Syllabus Topic : Graph Coloring

6.1.6 Graph Coloring

→ **(May 13, Dec. 13, May 16)**

Q. Explain Graph coloring problem using backtracking. Write algorithm for same. **MU - May 2013, 10 Marks**

Q. State Graph coloring algorithm. Explain strategy used for solving it along with example. **MU - Dec. 2013, 10 Marks**

Q. Write a short note on Graph coloring. **MU - May 2016, 10 Marks**

Definition

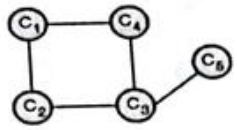
- **Graph coloring** is the problem of coloring vertices of a graph in such a way so that no two adjacent vertices have the same color. This is also called **vertex coloring** problem.
- If coloring is done using at most k colors, it is called **k-coloring.**
- The smallest number of colors required for coloring graph is called its **chromatic number.**

- The chromatic number is denoted by X(G). Finding the chromatic number for the graph is NP-complete problem.
- Graph coloring problem is both, decision problem as well as an optimization problem. A decision problem is stated as, "With given M colors and graph G, whether such color scheme is possible or not?".
- The optimization problem is stated as, "Given M colors and graph G, find the minimum number of colors required for graph coloring."
- Graph coloring problem is a very interesting problem of graph theory and it has many diverse applications. Few of them are listed below.

Applications

(i) Design a timetable
(ii) Sudoku
(iii) Register allocation in compiler
(iv) Map coloring
(v) Mobile radio frequency assignment

The input to the graph is adjacency matrix representation of the graph. Value M(i, j) = 1 in the matrix represents there exists an edge between vertex i and j. A graph and its adjacency matrix representation are shown in the Fig. 6.1.9.



	C_1	C_2	C_3	C_4	C_5
C_1	0	1	0	1	0
C_2	1	0	1	0	0
C_3	0	1	0	1	1
C_4	1	0	1	0	1
C_5	0	0	1	0	0

Fig. 6.1.9 : Graph and its adjacency matrix representation

- The problem can be solved simply by assigning a unique color to each vertex, but this solution is not optimal. It may be possible to color the graph with colors less than |V|. Fig. 6.1.10 and 6.1.11 demonstrate both such instances. Let Ci denotes the i^{th} color.

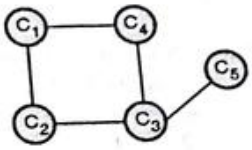


Fig. 6.1.10 : Nonoptimal solution (uses 5 colors)

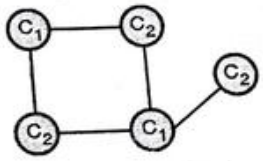


Fig. 6.1.11 : Optimal solution (uses 2 colors)

- This problem can be solved using backtracking algorithms as follows:
  - List down all the vertices and colors in two lists
  - Assign color 1 to vertex 1
  - If vertex 2 is not adjacent to vertex 1 then assign the same color, otherwise assign color 2.
  - Repeat the process until all vertices are colored.
- Algorithm backtracks whenever color i is not possible to assign to any vertex k and it selects next color i + 1 and test is repeated. Consider the graph shown in Fig. 6.1.12.

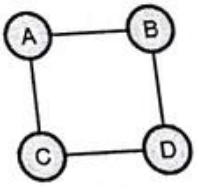


Fig. 6.1.12

- If we assign color 1 to vertex A, the same color cannot be assigned to vertex B or C. In next step, B is assigned some different color 2. Vertex A is already colored and vertex D is a neighbor of B, so D cannot be assigned color 2. The process goes on. State space tree is shown in Fig. 6.1.13.

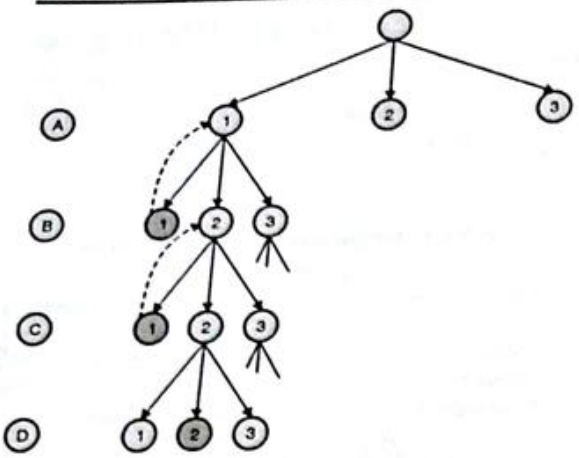


Fig. 6.1.13 : State space tree of the graph of Fig. 6.1.12

Thus, vertices A and C will be colored with color 1, and vertices B and D will be colored with color 2.

Algorithm for graph coloring is described here :

```
Algorithm GRAPH_COLORING(G, COLOR, i)
// Description : Solve the graph coloring problem using backtracking
// Input : Graph G with n vertices, list of colors, initial vertex i
// COLOR[1...n] is the array of n different colors
// Output : Colored graph with minimum color
if CHECK_VERTEX(i) == 1 then
    if i == N then
    print COLOR[1...n]
    else
    j ← 1
    while (j ≤ M) do
    COLOR(i + 1) ← j
    j ← j + 1
    end
    end
end
Function CHECK_VERTEX(i)
for j ← 1 to i – 1 do
    if Adjacent(i, j) then
        if COLOR(i) == COLOR(j) then
    return 0
    end
    end
end
return 1
```

If i is the last vertex in graph, print solution tuple

Assign color j to vertex (i + 1)

Loop through already colored (i - 1) vertices

Return true if vertices i and j are adjacent

If color assigned to vertex i and any of its adjacent vertex is same, then backtrack

Complexity Analysis

The number of anode increases exponentially at every level in state space tree. With M colors and n vertices, total number of nodes in state space tree would be $1 + M + M^2 + M^3 + ... + M^n$

Hence, $T(n) = 1 + M + M^2 + M^3 + ... + M^n = \frac{M^{n+1} - 1}{M - 1}$

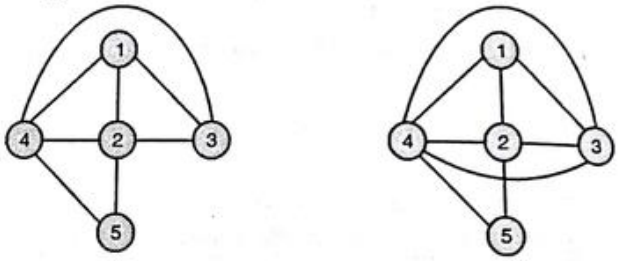
So, $T(n) = O(M^n)$.

Thus, graph coloring algorithm runs in exponential time.

1. Planar Graphs

Q. What are planar graphs? (2 Marks)

- A graph is called **planar** if it can be drawn on a 2D plane such that no two edges cross each other. Graph coloring problem is a well-known problem of a planar graph.
- Planar and non-planar graphs are illustrated in Fig. 6.1.14 :



Planar graph **Non planar graph**

Fig. 6.1.14 : Illustration of planar and non planar graph

2. Bipartite graph

Q. What is bipartite graph? How many colors are required to color bipartite graph? (5 Marks)

- Graph G is said to be **bipartite graph** if we can divide the vertices of graph G into two disjoint sets U and V such that each edge connects a vertex in U to one vertex in V. Disjoint independent sets U and V are called **partite sets.** If we cut the edges by vertical lines, all vertices become isolated vertex. Fig. 6.1.15 describes the bipartite graph.

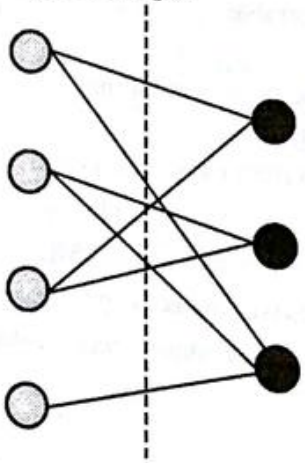


Fig. 6.1.15 : Bipartite Graph

- It is intuitive from the figure that we need maximum two colors for graph coloring problem if the graph is bipartite. None of the vertexes in U and V is connected to any other vertex of the set itself.
- All vertices in U can be colored using one color as there is no adjacency between vertices. Similarly, vertices in V can be colored using one color. Thus, bipartite graph needs only two colors for graph coloring problem.

Ex. 6.1.10

Construct planar graph for the following the map. Explain how to find m-coloring of this planar graph by using an m-coloring Backtracking algorithm.

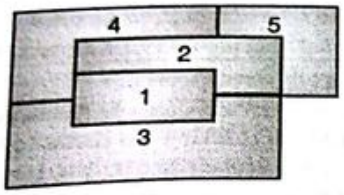


Fig. P. 6.1.10

Soln. :

In Fig. P. 6.1.10, area 1 is adjacent to area 2, 3 and 4. So in a planar graph, vertex 1 will be connected to vertices 2, 3 and 4. Area 2 is adjacent to area 1, 3, 4 and 5, so in a planar graph, vertex 2 will be connected to vertices 1, 3, 4 and 5. This process will be repeated for all areas. Hence, the planar graph would be,

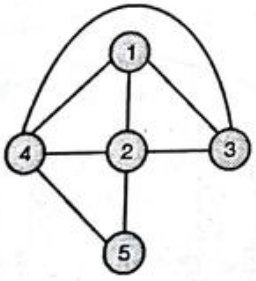


Fig. P. 6.1.10(a)

- This problem can be solved using backtracking algorithms as follows :
  - List down all the vertices and colors in two lists
  - Assign color 1 to vertex 1
  - If vertex 2 is not adjacent to vertex 1 then assign the same color, otherwise assign color 2.
  - Repeat the process until all vertices are colored.
- Algorithm backtracks whenever color i is not possible to assign to any vertex k and it selects next color i + 1 and test is repeated. This graph can be colored using 4 colors. Gray node indicates backtracking.

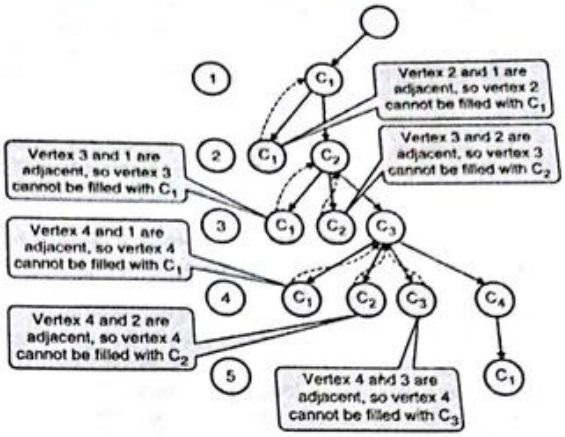


Fig. P. 6.1.10(b) : State space tree of given graph

Syllabus Topic : 15 Puzzle Problem

6.2 Branch and Bound

6.2.1 General Method

→ (May 14, Dec. 15)

Q. Comment on model of computation : Branch and Bound. **MU - May 2014, 5 Marks**

Q. Write a short note on Branch and bound strategy. **MU - Dec. 2015, 10 Marks**

Q. Describe the method with respect to Branch and Bound. **(7 Marks)**

Q. Explain the branch and bound algorithmic strategy for solving the problem. **(7 Marks)**

- Branch and bound builds the state space tree and find the optimal solution quickly by pruning few of the tree branches which does not satisfy the bound.
- Backtracking can be useful where some other optimization techniques like greedy or dynamic programming fail. Such algorithms are typically slower than their counterparts. In the worst case, it may run in exponential time, but careful selection of bounds and branches makes an algorithm to run reasonably faster.
- Most of the terminologies of backtracking are used in this chapter too. In branch and bound, all the children of E nodes are generated before any other live node becomes E node.
- Branch and bound technique in which E-node puts its children in the queue is called FIFO branch and bound approach.
- And if E-node puts its children in the stack, then it is called LIFO branch and bound approach.
- Bounding functions are a heuristic function. **Heuristic function** computes the node which maximizes the

probability of better search or minimizes the probability of worst search. According to maximization or minimization problem, highest or lowest heuristic valued node is selected for further expansion from a set of nodes.

- **Example :** FIFO Branch and bound approach.
- Let us try to understand the FIFO branch and bound with the help of 4-queen problem. State space tree generated by FIFO branch and bound method is shown in Fig. 6.2.1.
- Solution vector for the 4-queen problem is defined by 4-tuple $X = (x_1, x_2, x_3, x_4)$, each x_i indicates column number of the i^{th} queen in i^{th} row.
- Initially, when chess board is empty, only node 1 would be a live node. It becomes E node and generates node 2, 18, 34 and 50, and they are kept on live node list. These four nodes at level one represent that queen 1 is placed in column 1, 2, 3 and 4 respectively.
- The number inside the circle indicates the order in which node *becomes E node*. The number beside circle indicates the order in which node is *generated*.
- When all the children of node 1 are generated, node 2 is selected as E – node. It generates node 3, 8 and 13. Queen Q_2 cannot be placed in column 2, so node 3 is killed using bounding function. Node 8 and 13 are added to the list of live nodes and node 18 becomes the next E node. It generates node 19, 24 and 29. If Q_1 is in the 2^{nd} column, we cannot place Q_2 in column 1 or 3, so kill node 19 and 24. Node 29 is inserted in the list of live nodes.
- In FIFO approach, nodes become E-node in the order they are generated.
- Node 34 will be the next E node. It generates child 35, 40, and 45. If Q_1 is in column 3, Q_2 cannot be in column 2 and 4, so kill node 40 and 45. Insert node 35 in the list of the live node. This is how this search is proceeding.

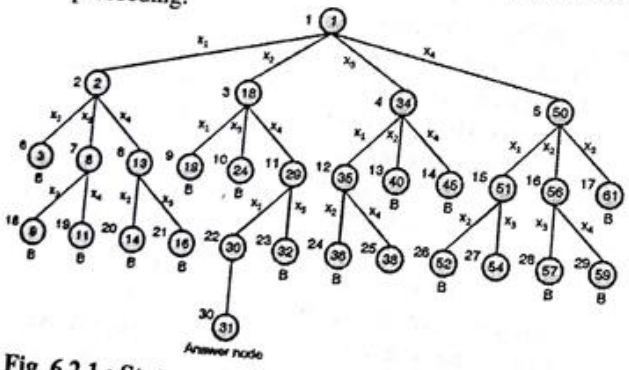


Fig. 6.2.1 : State space tree for 4-queen problem using branch and bound

6.2.2 Backtracking Vs Branch and Bound

Q. Differentiate Backtracking and Branch and Bound Method. Illustrate with an example of 4-Queen's Problem. **(5 Marks)**

Q. What is the difference between backtracking approach and branch and bound approach. **(5 Marks)**

Sr. No.	Backtracking	Branch and Bound
1.	Typically used to solve decision problems.	Used to solve optimization problems.
2.	Nodes in state space tree are explored in depth-first order.	Nodes in tree may be explored in depth-first or breadth-first order.
3.	Next move from current state can lead to bad choice.	Next move is always towards better solution.
4.	On successful search of solution in state space tree, search stops.	Entire state space tree is searched in order to find optimal solution.
5.	Applications Hamiltonian cycle problem Graph coloring problem	Applications Travelling salesman problem Knapsack problem

- **Example :** 4 Queen Problem.
- State space tree for 4-queen problem using branch and bound and backtracking are shown in Fig. 6.2.1 and Fig. 6.2.2, respectively
- Backtracking method explores the node in depth-first order. It keeps visiting nodes until it fails. Then the method backtracks. Nodes are explored in LIFO order. Refer Fig. 6.2.1, at level 0, the algorithm generates node 1, then instead of generating all child of node 1, backtracking approach expands node 2. After node 2, node 6 becomes the E-node. Node 6 fails to find the solution, so algorithm backtracks and generates another child of node 2, i.e. 8. Then 8 becomes E-node and generates node 9. Thus backtracking performs a depth-first search.

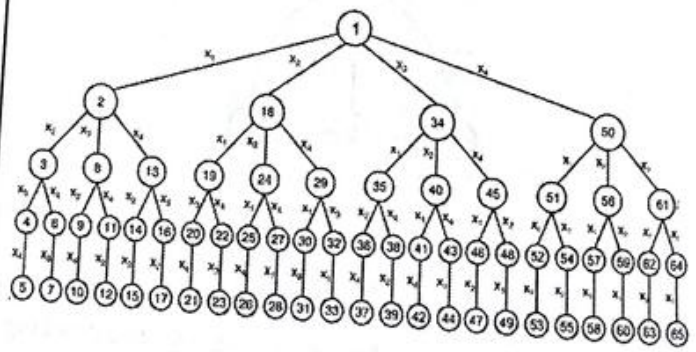


Fig. 6.2.2 : State space tree for four queen problem using backtracking

- Branch and bound cut down the node from further exploration by applying a certain bounding function on each node. If the node does not satisfy the bounding function then it rejects that node. Node selection in 4 queen problem using the branch and bound technique is discussed in the previous section.

6.2.3 Applications of Backtracking

Branch and bound technique is useful to solve problems like

(1) Knapsack problem
(2) Travelling salesman problem
(3) Assignment problem
(4) Job scheduling problem

6.2.4 Control Abstraction

6.2.4(A) LC Search

Q. What is LC search? (2 Marks)

- FIFO or LIFO is a very crude way of searching. It does not check the goodness of node. They blindly select the E node strictly in FIFO or LIFO order, without giving any preference to the node having better chances of getting answers quickly.
- In Fig. 6.2.1, node 31 is the answer node. When node 30 is generated, it should become the E-node, so that on every next step we get the solution. But the FIFO strategy forces the search to expand all the live nodes generated before 30 i.e. node {35, 40, 45, 51, 56, 61, 9, 11, 14, 16}.
- This search can be speeded up using the ranking function $\hat{c}(.)$for live nodes. Consider the Fig. 6.2.1. If ranking function assigns a better value to node 30 than remaining all live nodes, then node 30 will become the E- node as soon as it is generated and we will get the solution on very next step, i.e. node 31.
- Ranking basically depends on the additional effort (cost) required to reach to answer node from the live node.
- For any node x, the cost is determined by (1) number of nodes in subtree x needs to be generated before answer node is generated, or (2) number of levels from x to answer node.
- Using the second measure, it can be seen from the Fig. 6.2.1 that the cost of answer node is 4 (node 31 is four level away from the root node).
- Node 39 (child of 38) is also answer node. So the cost of nodes {18, 34}, {29, 35}, and {30, 38} are respectively 3, 2 and 1. And remaining all nodes on level 2, 3 and 4 have a cost greater than 3, 2 and 1, respectively.
- With this approach, only 1, 18, 29 and 30 becomes the E-node.
- And the other nodes which are generated but not converted to E node would be 2, 34, 50, 19, 24, 32 and 31.
- Cost measure 1 generates minimum number of nodes.
- If cost measure 2 is used, then only nodes to become E node would be on the path from answer node to root node.
- For any measure, searching the state space tree incurs major search time. And hence, nodes are ranked using estimated cost $\hat{g}(.)$.
- Let $\hat{g}(.)$ be the additional effort needed to reach an answer node from x. Function $\hat{c}(.)$ assigns ranking to node x such that,

$$\hat{c}(x) = f(h(x)) + g(x)$$

- $h(x)$ is the cost of reaching from x to root.
- In LC search, the cost function c(.) is defined as follows:
  - If x is answer node, then c(x) is the cost of reaching x from root.
  - If x is not answer node and subtree of x does not contain answer node than $c(x) = \infty$
  - Otherwise, c(x) is the cost of minimum cost answer node in subtree x.

6.2.4(B) Control Abstraction for Least Cost Search

Q. Describe the control abstraction for LC search with respect to Branch and Bound. (6 Marks)
Q. How does it help in finding a solution for branch and bound algorithm? (6 Marks)
Q. Explain in detail Control abstraction for LC search. (6 Marks)

- Let T and c(.) represent state space tree and cost function for the node in T.
- For node $x \in T$, c(x) represents the minimum cost of answer node in the subtree rooted at x. Thus, C(T) represents the cost of the minimum-cost answer in state space T.
- Finding exact cost function c(.) is challenging, so in practice, the heuristic function $\hat{c}(.)$ is used to estimate c(.).
- Heuristic functions are often simple to compute and provide close to optimal performance.
- Algorithm for LCSearch is described below. Function Least() finds the live node with the minimum $\hat{c}(.)$. This node is deleted from the list of live nodes and returned.
- Function Add(x) adds the new live node to the list of live nodes. Min-heap data structure is used to implement a list of live nodes so that the node with minimum cost always be at the front. So that Least() function can retrieve the minimum cost node in constant time.
- The LCSearch algorithm returns the path from answer node to the root node. To trace the path to the root, parent index is stored with each live node x.

```
Algorithm LCSearch(T)
// Description : Find answer node from state space tree and print
                 path
// Input : State space tree T
// Output : Success / Failure
if T is an answer node then
    output T
    return
end
E ← T
Initialize list of live node to empty
While (true) do
    for each child x of E do
        if x is an answer node then
            output path from x to T
            return
        end
        Add(x)
        x.parent ← E
    end
    if there are no more live nodes then
        print "No answer node"
        return
    end
    E = Least()
end
```

Root node is the answer node

Root be the first live node

x is new live node and added to list of live nodes

x was child of E, so E is set as a parent of x

List of live node is empty and answer node is not found

Current node was not answer node, so update E node with minimum cost node from list of live nodes

6.2.4(C) Bounding

Q. Describe bounding with respect to Branch and Bound. (7 Marks)

- Branch and bound technique generates all the child nodes of E-node before another node becomes E node.
- Let c(x) represent the cost of answer node x. The aim is to find minimum cost answer node.
- Search strategies like FIFO, LIFO and LC search differs in terms of the sequence in which they explore the nodes in state space tree.
- Let ranking function $\hat{c}(x) \leq c(x)$ be a lower bound of the solution. If *upper* represents the upper bound of minimum cost answer node, then all live node x with $\hat{c}(x) > upper$ may be killed because all the answer node reachable from x have cost $c(x) \geq \hat{c}(x) > upper$.
- The initial value of upper bound may be taken as ∞, or it may be computed using some heuristic function.
- Value of *upper* is updated for every new answer node.
- Maximization problems can be directly converted to minimization problem by changing the sign of objective function.

6.2.4(D) Control Abstraction for FIFO Branch and Bound

Q. Explain the FIFO branch and bound with respect to branch and bound technique with a suitable example. (10 Marks)

Q. Discuss the control abstraction for FIFO branch and bound. (7 Marks)

- Branch and bound method employ either BFS or DFS search. During BFS, expanded nodes are kept in a queue, whereas in DFS, nodes are kept on the stack. BFS approach is first in first out (FIFO) method, while DFS is last in first out (LIFO). In FIFO branch and bound, expanded nodes are inserted into the queue and the first generated node becomes next live node. Control abstraction for FIFO branch and bound technique is shown here.

```
Algorithm BB_FIFO()
// T is the state space tree and n is the node in tree T

if T is answer node then
    u ← min(cost[T], upperBound(T) + E)
    print (T)
    return
end
E ← T
while (true) do
for each child x of T do
    if x is answer node then
        print "Display path from x to root"
        return
    end
    ENQUEUE(x) // Insert new node at the end of QUEUE
    x.parent ← E // π[x] indicates parent of x
    if x is answer node AND cost(x) < upperBound
    then
        u ← min(cost[T], ub(T) + E)
    end
end
if no more live nodes then
    print "No live node exists"
    print "Minimum cost :", ub
    return n
end
E ← DEQUEUE()  // Remove front node from the QUEUE
end
```

6.2.5 15 Puzzle Problem

➔ **(May 13, May 14, Dec. 14, May 15, May 16, May 17)**

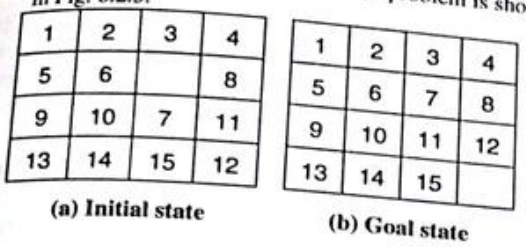
Q. Explain 15-puzzle problem using branch and bound. MU - May 2013, 10 Marks

Q. Write note on : 15-puzzle problem. MU - May 2014, Dec. 2014, May 2015, May 2016, 10 Marks

Q. Explain how branch and bound strategy can be used in 15 puzzle problem. MU - May 2017, 10 Marks

- 15-puzzle problem is the problem of arranging 15 titles in 4 x 4 board such that titles are ordered from left to right and bottom to top such that the bottom right title contains empty space.

- Given the initial state with random distribution of numbers between 1 to 15, aim is to achieve the goal state by moving the empty tile. This is NP complete problem. Initial and goal state of the problem is shown in Fig. 6.2.3.



(a) Initial state (b) Goal state

Fig. 6.2.3 : States of 15-puzzle problem.

- **Cost function :** Each node in the state space tree is associated with some cost. Cost function is applied to each node and the function helps to select the next E node. E node is the node being evaluated. The node with minimum cost should be selected for the further expansion.
- The cost function is defined as, $C(x) = g(x) + h(x)$, where $g(x)$ is the cost of reaching to current state from the initial state and $h(x)$ is the cost of reaching from current state to the answer state.
- Cost function for 15-puzzle problem is defined as the number of tiles on wring location.
- $C(x) = f(x) + h(x)$, where $f(x)$ is the length from the root node in state space tree, i.e. number of moves so far, and $h(x)$ is the non-blank tiles which are not on the correct location.
- For the initial state shown in Fig. 6.2.3, there are four possible moves (left, right, up, down), the cost for all four configurations is shown in Fig. 6.2.4. Tiles on wrong position are highlighted in red color.

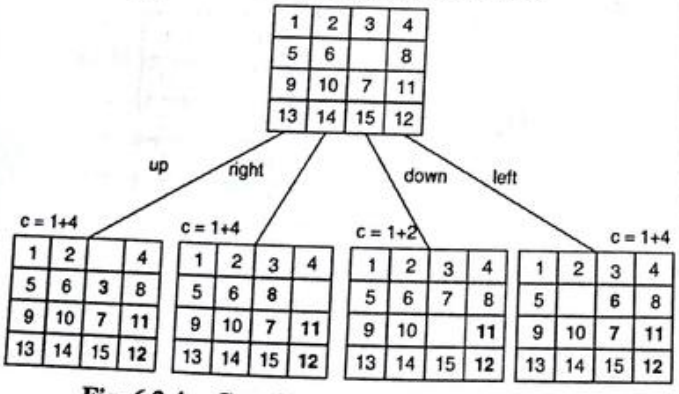


Fig. 6.2.4 : Cost f(x) + h(x) after first move

- Third configuration has the minimum cost, so remaining three states will be cut down and will not be explored further. The possibilities on next move is shown in Fig. 6.2.5.

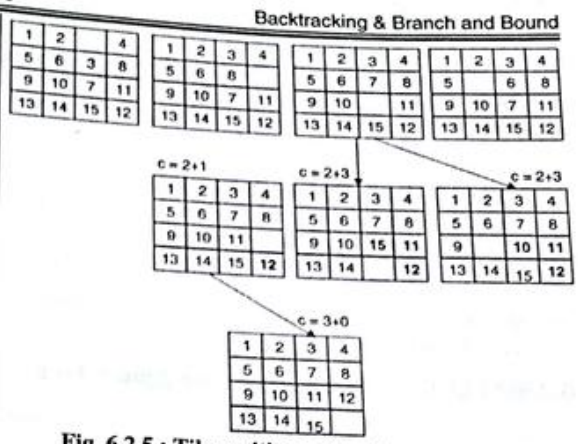


Fig. 6.2.5 : Tile positions after sbsequent moves

Algorithm for 15-puzzle problem is stated below:

```
Struct Node
{
    Node *next;
    Node *parent;
    Float cost;
}
Algorithm LC_SEARCH(Node * t)
if *t is an answer node then
    print *t
    return
end
E ← t;
Initialize the live nodes to be empty
while (true) do
    for each child x of E do
    if x is goal state then
    print path from root to x
    return
    end
    Add(x)          // Add x to list of live nodes
    (x → Parent) ← E  // Set E to the parent of x
    if there are no more live nodes then
    print "Answer not found"
    end
    E ← least()     // Set minimum cost node to E node
    end
end
```

Syllabus Topic : Travelling Salesman Problem

6.2.6 Travelling Salesman Problem

→ **(May 13, May 15)**

Q. Explain travelling salesperson problem using branch and bound method. **MU - May 2013, 10 Marks**

Q. Write note on : Travelling sales person problem **MU - May 2015, 10 Marks**

Q. What is travelling salesman problem? **(6 Marks)**

- Travelling Salesman Problem (TSP) is interesting problem. Problem is defined as "given n cities and distance between each pair of cities, find out the path which visits each city exactly once and come back to starting city, with constraint of minimizing the travelling distance."
- TSP has many practical applications. It is used in network design, transportation route design. Objective is to minimize the distance. We can start tour from any random city and visit other cities in any order. With n cities, n! different permutations are possible. Exploring all paths using brute force attack may not be useful in real life applications.

6.2.6(A) LCBB using Static State Space Tree

- Branch and bound is effective way to find better, if not best, solution in quick time by pruning some of the unnecessary branches of search tree.

It works as follow :

Consider directed weighted graph G = (V, E), where node represents cities and weighted directed edges represents direction and distance between two cities.

1. Initially, graph is represented by cost matrix C, where
   C_{ij} = cost of edge, if there is a direct path from city i to city j
   C_{ij} = ∞, if there is no direct path from city i to city j.
2. Convert cost matrix to reduced matrix by subtracting minimum values from appropriate rows and columns, such that each row and column contains at least one zero entry.
3. Find cost of reduced matrix. Cost is given by summation of subtracted amount from the cost matrix to convert it in to reduce matrix.
4. Prepare state space tree for the reduce matrix
5. Find least cost valued node A by computing reduced cost node matrix with every remaining node. This is required to find next best move from current city.
6. If <i, j> edge is to be included, then do following :
   (a) Set all values in row i and all values in column j of A to ∞
   (b) Set A[j, i] = ∞
   (c) Reduce A again, except rows and columns having all ∞ entries.
7. Compute the cost of newly created reduced matrix as,
   Cost = L + Cost(i, j) + r
   Where, L is cost of reduced cost matrix and r is A[i, j].
8. If all nodes are not visited then go to step 4.

Reduction procedure is described below:

Row reduction

Matrix M is called reduced matrix if each of its row and column has atleast one zero entry or entire row or entire column has ∞ value. Let M represents the distance matrix of 5 cities. M can be reduced as follow :

$M_{RowRed} = \{M_{ij} - \min \{M_{ij} \mid 1 \le j \le n, \text{ and } M_{ij} < \infty\}\}$

Consider the following distance matrix:

M =

∞	20	30	10	11
15	∞	16	4	2
3	5	∞	2	4
19	6	18	∞	3
16	4	7	16	∞

Find the minimum element from each row and subtract it from each cell of matrix.

M =

∞	20	30	10	11	→ 10
15	∞	16	4	2	→ 2
3	5	∞	2	4	→ 2
19	6	18	∞	3	→ 3
16	4	7	16	∞	→ 4

Reduced matrix would be:

M_{RowRed} =

∞	10	20	0	1
13	∞	14	2	0
1	3	∞	0	2
16	3	15	∞	0
12	0	3	12	∞

Row reduction cost is the summation of all the values subtracted from each rows:

Row reduction cost (M) = 10 + 2 + 2 + 3 + 4 = 21

Column reduction

Matrix M_{RowRed} is row reduced but not the column reduced. Matrix is called column reduced if each of its column has at least one zero entry or all ∞ entries.

$M_{ColRed} = \{M_{ji} - \min \{M_{ji} \mid 1 \le j \le n, \text{ and } M_{ji} < \infty\}\}$

To reduced above matrix, we will find the minimum element from each column and subtract it from each cell of matrix.

M_{RowRed} =

∞	10	20	0	1
13	∞	14	2	0
1	3	∞	0	2
16	3	15	∞	0
12	0	3	12	∞
↓	↓	↓	↓	↓
1	0	3	0	0

Column reduced matrix M_{ColRed} would be:

M_{ColRed} =

∞	10	17	0	1
12	∞	11	2	0
0	3	∞	0	2
15	3	12	∞	0
11	0	0	12	∞

Each row and column of M_{ColRed} has at least one zero entry, so this matrix is reduced matrix.

Column reduction cost (M) = 1 + 0 + 3 + 0 + 0 = 4

State space tree for 5 city problem is depicted in Fig. 6.2.6. Number within circle indicates the order in which the node is generated, and number of edge indicates the city being visited.

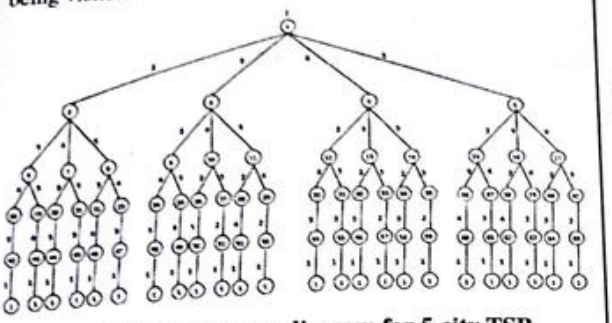


Fig. 6.2.6 : State space diagram for 5-city TSP

Ex. 6.2.1

What is travelling salesman problem? Find the solution of following travelling salesman problem using branch and bound method.

Cost Matrix =

∞	20	30	10	11
15	∞	16	4	2
3	5	∞	2	4
19	6	18	∞	3
16	4	7	16	∞

Soln. :

- The procedure for dynamic reduction is as follow:
- Draw state space tree with optimal reduction cost at root node
- Derive cost of path from node i to j by setting all entries in i^{th} row and j^{th} column as ∞.

  Set M[j][i] = ∞
- Cost of corresponding node N for path i to j is summation of optimal cost + reduction cost + M[j][i]
- After exploring all nodes at level i, set node with minimum cost as E node and repeat the procedure until all nodes are visited.
- Given matrix is not reduced. In order to find reduced matrix of it, we will first find the row reduced matrix followed by column reduced matrix if needed. We can find row reduced matrix by subtracting minimum element of each row from each element of corresponding row. Procedure is described below:
- Reduce above cost matrix by subtracting minimum value from each row and column.

∞	20	30	10	11	→ 10
15	∞	16	4	2	→ 2
3	5	∞	2	4	→ 2
19	6	18	∞	3	→ 3
16	4	7	16	∞	→ 4

⇒

∞	10	20	0	1	
13	∞	14	2	0	
1	3	∞	0	2	$= M_1'$
16	3	15	∞	0	
12	0	3	12	∞	
↓	↓	↓	↓	↓	
1	0	3	0	0	

M_1' is not reduced matrix. Reduce it subtracting minimum value from corresponding column. Doing this we get,

∞	10	17	0	1	
12	∞	11	2	0	
0	3	∞	0	2	$= M_1$
15	3	12	∞	0	
11	0	0	12	∞	

Cost of M_1 = C(1)

= Row reduction cost + Column reduction cost

= (10 + 2 + 2 + 3 + 4) + (1 + 3)=25

This means all tours in graph has length at least 25. This is the optimal cost of the path.

State space tree

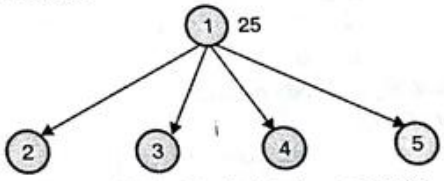


Fig. P. 6.2.1 : Partial state space tree

Let us find cost of edge from node 1 to 2, 3, 4, 5.

Select edge 1-2

Set M_1 [1] [] = M_1 [] [2] = ∞

Set M_1 [2] [1] = ∞

Reduce the resultant matrix if required.

∞	∞	∞	∞	∞	→ x
∞	∞	11	2	0	→ 0
0	∞	∞	0	2	→ 0 $= M_2$
15	∞	12	∞	0	→ 0
11	∞	0	12	∞	→ 0
↓	↓	↓	↓	↓	
0	x	0	0	0	

M_2 is already reduced.

∴ Cost of node 2 :

C(2) = C(1) + Reduction cost + M_1 [1] [2]

= 25 + 0 + 10 = 35

Select edge 1-3

Set M_1 [1] [] = M_1 [] [3] = ∞

Set M_1 [3] [1] = ∞

Reduce the resultant matrix if required.

$M_1 \Rightarrow$

∞	∞	∞	∞	∞	→ x
12	∞	11	2	0	→ 0
0	∞	∞	0	2	→ 0
15	∞	12	∞	0	→ 0
11	∞	0	12	∞	→ 0
↓	↓	↓	↓	↓	
11	0	x	0	0	

$\Rightarrow$

∞	∞	∞	∞	∞	
1	∞	∞	2	0	
∞	3	∞	0	2	$= M_3$
4	3	∞	∞	0	
0	0	∞	12	∞	

Cost of node 3

$C(3) = C(1) + \text{Reduction cost} + M_1[1][3]$

$= 25 + 11 + 17 = 53$

Select edge 1-4

Set $M_1[1][] = M_1[][4] = \infty$

Set $M_1[4][1] = \infty$

Reduce resultant matrix if required.

$M_1 \Rightarrow$

∞	∞	∞	∞	∞	→ x
12	∞	11	∞	0	→ 0
0	3	∞	∞	2	→ 0
∞	3	12	∞	0	→ 0
11	0	0	∞	∞	→ 0
↓	↓	↓	↓	↓	
0	0	0	x	0	

$= M_4$

Matrix M_4 is already reduced.

Cost of node 4

$C(4) = C(1) + \text{Reduction cost} + M_1[1][4]$

$= 25 + 0 + 0 = 25$

Select edge 1-5

Set $M_1[1][] = M_1[][5] = \infty$

Set $M_1[5][1] = \infty$

Reduce the resultant matrix if required.

$M_1 \Rightarrow$

∞	∞	∞	∞	∞	→ x
12	∞	11	2	∞	→ 2
0	3	∞	0	∞	→ 0
15	3	12	∞	∞	→ 3
∞	0	0	12	∞	→ 0

$\Rightarrow$

∞	∞	∞	∞	∞	
10	∞	9	0	∞	
0	3	∞	0	∞	$= M_5$
12	0	9	∞	∞	
∞	0	0	12	∞	
↓	↓	↓	↓	↓	
0	0	0	0	x	

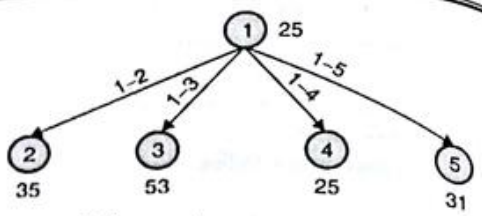
Cost of node 5

$C(5) = C(1) + \text{reduction cost} + M_1[1][5]$

$= 25 + 5 + 1 = 31$

State space diagram

Node 4 has minimum cost for path 1-4. We can go to vertex 2, 3 or 5.



Let's explore all three nodes.

Select path 1-4-2 : (Add edge 4-2)

Set $M_4[1][] = M_4[4][]$

$= M_4[][2] = \infty$

Set $M_4[2][1] = \infty$

Reduce resultant matrix if required.

$M_4 \Rightarrow$

∞	∞	∞	∞	∞	→ x
∞	∞	11	∞	0	→ 0
0	∞	∞	∞	2	→ 0
∞	∞	∞	∞	∞	→ x
11	∞	0	∞	∞	→ 0
↓	↓	↓	↓	↓	
0	0	0	x	0	

$= M_6$

Matrix M_6 is already reduced.

Cost of node 6

$C(6) = C(4) + \text{Reduction cost} + M_4[4][2]$

$= 25 + 0 + 3 = 28$

Select edge 4-3 (Path 1-4-3)

Set $M_4[1][] = M_4[4][] = M_4[][3] = \infty$

Set $M[3][1] = \infty$

Reduce the resultant matrix if required.

$M_4 \Rightarrow$

∞	∞	∞	∞	∞	→ x
12	∞	∞	∞	0	→ 0
∞	3	∞	∞	2	→ 2
∞	∞	∞	∞	∞	→∞
11	0	∞	∞	∞	→ 0

$\Rightarrow$

∞	∞	∞	∞	∞	
12	∞	∞	∞	0	
∞	1	∞	∞	0	$= M_7'$
∞	∞	∞	∞	∞	
11	0	∞	∞	∞	
↓	↓	↓	↓	↓	
11	0	x	x	0	

M_7' is not reduced. Reduce it by subtracting 11 from column 1.

$\therefore M_7' \Rightarrow$

∞	∞	∞	∞	∞
1	∞	∞	∞	0
∞	1	∞	∞	2
∞	∞	∞	∞	∞
0	0	∞	∞	∞

$= M_7$

Cost of node 7

$C(7) = C(4) + \text{Reduction cost} + M_4[4][3]$

$= 25 + 2 + 11 + 12 = 50$

Select edge 4-5 (Path 1-4-5)

$M_4 \Rightarrow$

∞	∞	∞	∞	∞	→ x
12	∞	11	∞	∞	→11
0	3	∞	∞	∞	→0
∞	∞	∞	∞	∞	→x
∞	0	0	∞	∞	→ 0

$\Rightarrow$

∞	∞	∞	∞	∞	
1	∞	0	∞	∞	
0	3	∞	∞	∞	$= M_8$
∞	∞	∞	∞	∞	
∞	0	0	∞	∞	
↓	↓	↓	↓	↓	
0	0	0	x	x	

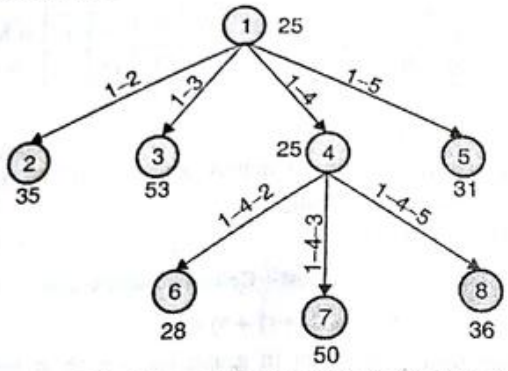
Matrix M_8 is reduced.

Cost of node 8

$C(8) = C(4) + \text{Reduction cost} + M_4[4][5]$

$= 25 + 11 + 0 = 36$

State space tree



Path 1-4-2 leads to minimum cost. Let's find the cost for two possible paths.

Add edge 2-3 (Path 1-4-2-3)

Set $M_6[1][] = M_6[4][]$

$= M_6[2][]$

$= M_6[][3] = \infty$

Set $M_6[3][1] = \infty$

Reduce resultant matrix if required.

$M_6 \Rightarrow$

∞	∞	∞	∞	∞	→ x
∞	∞	∞	∞	∞	→x
0	∞	∞	∞	2	→0
∞	∞	∞	∞	∞	→x
11	∞	∞	∞	∞	→11

$\Rightarrow$

∞	∞	∞	∞	∞	
∞	∞	∞	∞	∞	
0	∞	∞	∞	2	$= M_9'$
∞	∞	∞	∞	∞	
0	∞	∞	∞	∞	
↓	↓	↓	↓	↓	
0	x	x	x	2	

$\therefore M_9' \Rightarrow$

∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
0	∞	∞	∞	0
∞	∞	∞	∞	∞
0	∞	∞	∞	∞

$= M_9$

Cost of node 9

$C(9) = C(6) + \text{Reduction cost} + M_6[2][3]$

$= 28 + 11 + 2 + 11 = 52$

Add edge 2-5 (Path 1-4-2-5)

Set $M_6[1][] = M_6[4][] = M_6[2][] = M_6[][5] = \infty$

Set $M_6[5][1] = \infty$

Reduce resultant matrix if required.

$\therefore M_6 \Rightarrow$

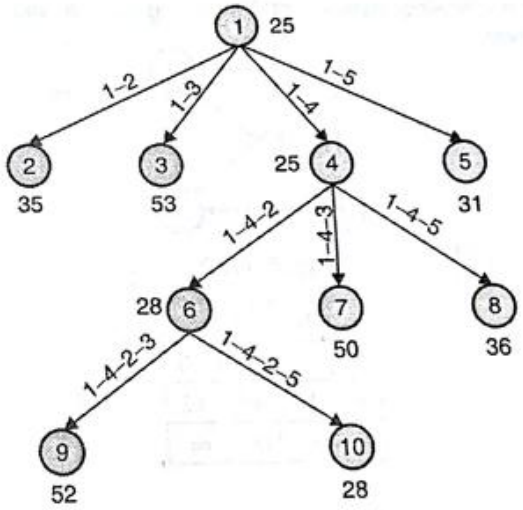
∞	∞	∞	∞	∞	
∞	∞	∞	∞	∞	
0	∞	∞	∞	∞	$= M_{10}$
∞	∞	∞	∞	∞	
∞	∞	0	∞	∞	

Cost of node 10

$C(10) = C(6) + \text{Reduction cost} + M_6[2][5]$

$= 28 + 0 + 0 = 28$

State space tree



Add edge 5-3 (Path 1-4-2-5-3)

$\therefore M_{10} \Rightarrow$

∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞

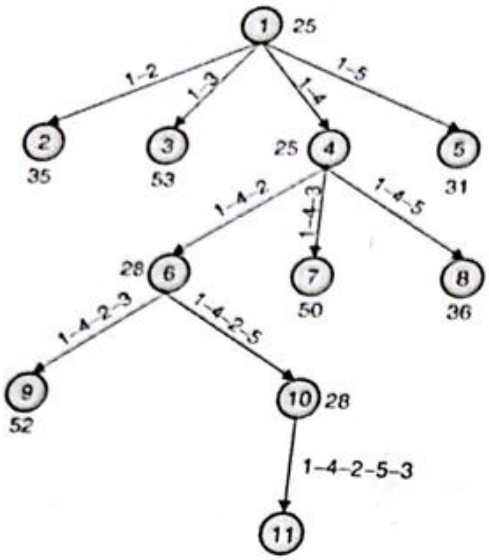
$= M_{11}$

Cost of node 11

$C(11) = C(10) + \text{Reduction cost} + M_{10}[5][3]$

$= 28 + 0 + 0 = 28$

State space tree



Ex. 6.2.2

Solve following instance of TSP using branch and bound method.

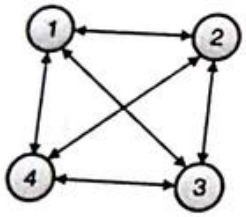


Fig. P. 6.2.2

∞	10	15	20
5	∞	9	10
6	13	∞	12
8	8	9	∞

Soln. :

- The procedure for dynamic reduction is as follow:
- Draw state space tree with optimal reduction cost at root node
- Derive cost of path from node i to j by setting all entries in i^{th} row and j^{th} column as ∞.
  Set M[j][i] = ∞
- Cost of corresponding node N for path i to j is summation of optimal cost + reduction cost + M[j][i]
- After exploring all nodes at level i, set node with minimum cost as E node and repeat the procedure until all nodes are visited.
- Given matrix is not reduced. In order to find reduced matrix of it, we will first find the row reduced matrix followed by column reduced matrix if needed. We can find row reduced matrix by subtracting minimum element of each row from each element of corresponding row. Procedure is described below:
- Reduce given by reducing minimum value from corresponding rows.

$$M = \begin{bmatrix} \infty & 10 & 15 & 20 \\ 5 & \infty & 9 & 10 \\ 6 & 13 & \infty & 12 \\ 8 & 8 & 9 & \infty \end{bmatrix} \begin{matrix} \to 10 \\ \to 5 \\ \to 6 \\ \to 8 \end{matrix} \Rightarrow \begin{bmatrix} \infty & 0 & 5 & 10 \\ 0 & \infty & 4 & 5 \\ 0 & 7 & \infty & 6 \\ 0 & 0 & 1 & \infty \end{bmatrix} = M_1'$$

- Column 3 and 4 in matrix M_1' does not contain 0 entry. So to reduce the matrix M_2' subtract minimum value from respective columns.

$$M_1' = \begin{bmatrix} \infty & 0 & 5 & 10 \\ 0 & \infty & 4 & 5 \\ 0 & 7 & \infty & 6 \\ 0 & 0 & 1 & \infty \end{bmatrix} \Rightarrow \begin{bmatrix} \infty & 0 & 4 & 5 \\ 0 & \infty & 3 & 0 \\ 0 & 7 & \infty & 1 \\ 0 & 0 & 0 & \infty \end{bmatrix} = M_1$$

$$\downarrow \quad \downarrow \quad \downarrow \quad \downarrow$$
$$0 \quad 0 \quad 1 \quad 5$$

- Matrix M_1 is reduced matrix because each of its row and column has 0 entry

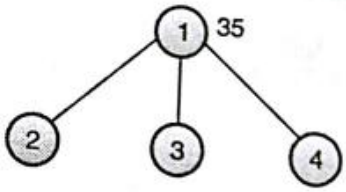
Cost (M_1) = C(1)

= Row reduction cost + Column reduction cost

= (10 + 5 + 6 + 8) + (1 + 5) = 35

- This means all tours in graph has length at least 35. This is the optimal cost of the path.

State space tree



Vertex 1 is connected with remaining all vertices let us final lower bound for each node in state space tree.

Add edge 1-2

Set $M_1[1][] = M_1[][2] = \infty$

Set M [2][] = ∞

Reduce resultant matrix if required.

$$\begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & 3 & 0 \\ 0 & \infty & \infty & 1 \\ 0 & \infty & 0 & \infty \end{bmatrix} \begin{matrix} \to x \\ \to 0 \\ \to 0 \\ \to 0 \end{matrix} = M_2$$

$$\downarrow \quad \downarrow \quad \downarrow \quad \downarrow$$
$$0 \quad x \quad 0 \quad 0$$

Each row and column in M_2 either contains at least one 0 entry or all ∞ entries. So it is already reached matrix.

Cost (M_2) = C(2) = C(1) + Reduction cost + M_1 [1][2]
= 35 + 0 + 0 = 35

Add edge 1-3

Set M_1 [1][] = M_1 [][3] = ∞

Set M_1 [3][1] = ∞

Reduce resultant matrix if required.

∞	∞	∞	∞	→	x
0	∞	∞	0	→	0
∞	7	∞	1	→	1
0	0	∞	∞	→	0

⇨

∞	∞	∞	∞	
0	∞	∞	0	
∞	6	∞	0	= M_3
0	0	∞	∞	
↓	↓	↓	↓	
0	0	x	0	

Matrix M_3 is reduced matrix

Cost (M_3) = C (3) = C (1) + Reduction cost + M_1 [1] [3]
= 35 + 1 + 4 = 40

Add edge 1 - 4

Set M_1 [1] [ ] = M_1 [ ] [4] = ∞

Set M_1 [4] [1] = ∞

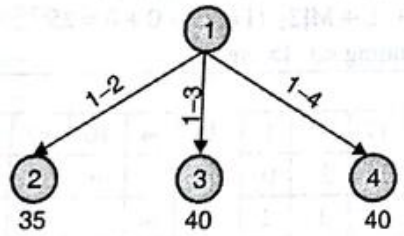
Reduce resultant matrix if required.

∞	∞	∞	∞	→	x
0	∞	3	∞	→	0
0	7	∞	∞	→	0 = M_4
∞	0	0	∞	→	0
↓	↓	↓	↓		
0	0	0	x		

M_4 is reduced matrix

Cost (M_4) = C (4) = C (1) + Reduction cost + M_1 [1] [4]
= 35 + 0 + 5 = 40

State space tree



Node 2 is the least cost node so select that path from 1 – 2, we have two choices, select 3 or 4

Add edge 1 – 2 – 3

Set M_2 [2] [ ] = M_2 [ ] [3]
= M_2 [1] [ ] = ∞

Set M_2 [3] [1] = ∞

Reduce resultant matrix if required

∞	∞	∞	∞	→	x
∞	∞	∞	∞	→	x
∞	∞	∞	1	→	1
0	∞	∞	∞	→	0

⇨

∞	∞	∞	∞	
∞	∞	∞	∞	
∞	∞	∞	0	= M_6
0	∞	∞	∞	
↓	↓	↓	↓	
0	x	x	0	

Cost of node 5

C (5) = C (2) + Reduction cost + M_2 [2] [3]
= 35 + 1 + 3 = 39

Add edge 2 – 4 (Path 1 – 2 – 4)

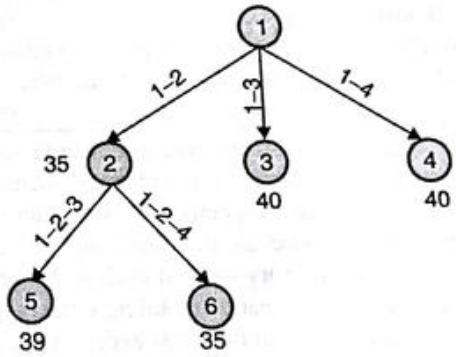
∞	∞	∞	∞	→	x
∞	∞	∞	∞	→	x
0	∞	∞	∞	→	0 = M_6
∞	∞	0	∞	→	0
↓	↓	↓	↓		
0	x	0	x		

M_7 is already reduced matrix

Cost of node 6

C (6) = C (2) + Reduction cost + M_2 [2] [4]
= 35 + 0 + 0 = 35

State space tree



Add edge 4 – 3 (Path 1 – 2 – 4 – 3)

Set M_7 [1] [ ] = M_7[2] [ ] = M_7 [4] [ ] = M_7 [ ] [3] = ∞

Set M_7 [3] [1] = ∞

Reduce resultant matrix if required.

∞	∞	∞	∞	→	x
∞	∞	∞	∞	→	x
∞	0	∞	∞	→	0 = M_7
∞	∞	∞	∞	→	x
↓	↓	↓	↓		
x	0	x	x		

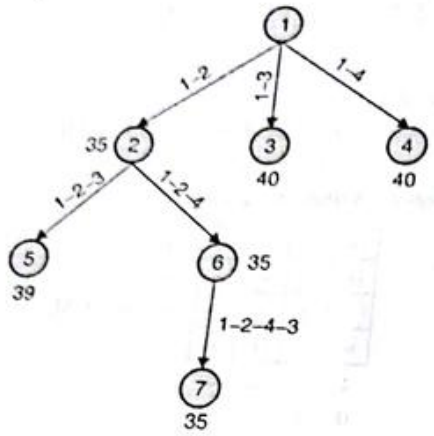
Matrix M_7 is already reduced.

Cost of node 7

$C(7) = C(6) + \text{Reduction cost} + M_6[4][3]$

$= 35 + 0 + 0 = 35$

State space tree



Optimal path = 1 – 2 – 4 – 3 – 1 and cost of path = 35.

6.2.6(B) LCBB using Dynamic State Space Tree

Q. Describe least cost search (LC Search) with respect to Branch and Bound. **(7 Marks)**

Q. Explain the Least cost search with respect to branch and bound technique with suitable example. **(7 Marks)**

- Static space tree is m-ary tree. In dynamic state space tree, left and right branch indicates inclusion and exclusion of edge respectively. Bounding function computation is same as previous method. Dynamic state space tree is binary tree. At each node, left branch (i, j) indicates all the paths including edge (i, j). Right branch (i, j) indicates all the paths excluding (i, j).
- At each level, we select the edge which gives highest probability of minimum tour cost. This can be done by selecting the right edge which gives highest value. We can find such edge from reduced cost matrix. Consider the following instance of TSP.

∞	20	30	10	11
15	∞	16	4	2
3	5	∞	2	4
19	6	18	∞	3
16	4	7	16	∞

∞	10	17	0	1
12	∞	11	2	0
0	3	∞	0	2
15	3	12	∞	0
11	0	0	12	∞

(a) Cost matrix C (b) Reduced matrix M with L = 25

Fig. 6.2.7 : Cost matrix and its reduced matrix

- Reduced matrix is computed in same way, as we did in previous case. Reduction cost of the matrix is 25. This implies, any route of the graph costs at least 25. Thus root of the tree has cost 25. We shall select the next edge such that exclusion of that leads to maximum. In other words, select the edge which gives maximum probability of minimum cost of path on inclusion of that edge.
- We can achieve this by considering one of the edge with reduced cost zero in reduced matrix. In Fig. 6.2.7(b) edge <1, 4>, <2, 5>, <3, 1>, <3, 4>, <4, 5>, <5, 2> and <5, 3> has 0 reduction cost. If we select any of the edge <a, b> from this list, then resultant cost matrix M will have entry ∞ on position M[a][b].
- If we include edge <1, 4>, set M[1][4] = ∞, and reduce M. This is done by subtracting 1 from row 1. So cost of right child will increase by 1.
- If we include edge <2, 5>, set M[2][5] = ∞, and reduce M. This is done by subtracting 2 from row 2. So cost of right child will increase by 1.
- If we include edge <3, 1>, set M[3][1] = ∞, and reduce M. This is done by subtracting 11 from column 1. So cost of right child will increase by 11.

Thus we will have,

Edge	<1, 4>	<2, 5>	<3, 1>	<3, 4>	<4, 5>	<5, 2>	<5, 3>
Increment in cost of right child	1	2	11	0	3	3	11

Here, edge <3, 1> and <5, 3> maximizes the cost of right child, so we can select any one of them. Let us select edge <3, 1>.

Set M[3][ ] = ∞ Set M[ ][1] = ∞ Set M[1][3] = ∞ Reduce the matrix if required	∞ 10 ∞ 0 1 ∞ ∞ 11 2 0 ∞ ∞ ∞ ∞ ∞ ∞ 3 12 ∞ 0 ∞ 0 0 12 ∞
Inclusion of <3, 1> in M	M_2 = Matrix is already reduced, L = 0.

Cost of matrix M_2= Cost of node 2,

$C(2) = C(1) + L + M[3][1] = 25 + 0 + 0 = 25$

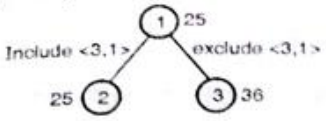
By excluding <3, 1> we get,

∞ 10 17 0 1 12 ∞ 11 2 0 ∞ 3 ∞ 0 2 15 3 12 ∞ 0 11 0 0 12 ∞	∞ 10 17 0 1 1 ∞ 11 2 0 ∞ 3 ∞ 0 2 4 3 12 ∞ 0 0 0 0 12 ∞
Set M[3, 1] = ∞ and reduce matrix if required	M_3 = Reduce matrix with L = 11

Cost of matrix M_1 = Cost of node 3.

$C(3) = C(1) + L + M[3][1] = 25 + 11 + 0 = 36$

At this point, dynamic state space tree would look like,



On inclusion of <3, 1>, we get matrix M_2. In M_2, we have <1, 4>, <2, 5>, <4, 5>, <5, 2> and <5, 3> edges with 0 value. So selecting any of the edge <a, b> from this list will make $M_2[a][b] = \infty$. Repeating previous step, we get

Edge	<1, 4>	<2, 5>	<4, 5>	<5, 2>	<5, 3>
Increment in $\hat{c}$ of right child	3	2	3	3	11

Here, edge <5, 3> maximizes of right child, so let us select edge <5, 3>.

Set $M_2[5][\ ] = \infty$ Set $M_2[\ ][3] = \infty$ Set $M_2[1][5] = \infty$ to avoid cycle Reduce the matrix if required	∞ 10 ∞ 0 ∞ ∞ ∞ ∞ 2 0 ∞ ∞ ∞ ∞ ∞ ∞ 3 ∞ ∞ 0 ∞ ∞ ∞ ∞ ∞	∞ 7 ∞ 0 ∞ ∞ ∞ ∞ 2 0 ∞ ∞ ∞ ∞ ∞ ∞ 0 ∞ ∞ 0 ∞ ∞ ∞ ∞ ∞
Inclusion of <5, 3> in M_2	This is not reduce matrix, reduce it by subtracting 3 from 2nd column	M_4 = Reduced matrix, L = 3

Cost of matrix M_4= Cost of node 4, $C(4) = C(2) + L + M_2[5][3] = 25 + 3 + 0 = 28$

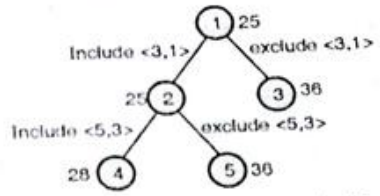
By excluding <5, 3> we get,

∞ 10 ∞ 0 1 ∞ ∞ 11 2 0 ∞ ∞ ∞ ∞ ∞ ∞ 3 12 ∞ 0 ∞ ∞ ∞ 12 ∞	∞ 10 ∞ 0 1 ∞ ∞ 0 2 0 ∞ ∞ ∞ ∞ ∞ ∞ 3 1 ∞ 0 ∞ 0 ∞ 12 ∞
Set $M_2[5, 3] = \infty$ and reduce matrix if required	M_5 = Reduce matrix with L = 11

Cost of matrix M_5= Cost of node 5,

$C(5) = C(2) + L + M_2[5][3] = 25 + 11 + 0 = 36$

At this point, dynamic state space tree would look like,



On inclusion of <5, 3>, we get matrix M_4. In M_4, we have <1, 4>, <2, 5>, <4, 2> and <4, 5> edges with 0 value. So selecting any of the edge <a, b> from this list will make $M_4[a][b] = \infty$. Repeating previous step, we get

Edge	<1, 4>	<2, 5>	<4, 2>	<4, 5>
Increment in $\hat{c}$ of right child	9	2	7	0

Here, edge <1, 4> maximizes $\hat{c}$ of right child, so let us select edge <1, 4>.

Set $M_4[1][\ ] = \infty$ Set $M_4[\ ][4] = \infty$ Set $M_2[4][5] = \infty$ to avoid cycle Reduce the matrix if required	∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞ 0 ∞ ∞ ∞ ∞ ∞ ∞ 0 ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞
Inclusion of <1, 4> in M_2	M_6 = Matrix M_6 is already reduced, L = 0

Cost of matrix M_6 = Cost of node 6, $C(6) = C(4) + L + M_4[1][4] = 28 + 0 + 0 = 28$

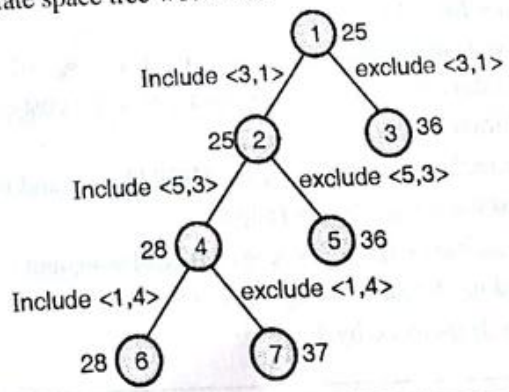
By excluding <1, 4> we get,

∞ 7 ∞ ∞ ∞ ∞ ∞ ∞ 2 0 ∞ ∞ ∞ ∞ ∞ ∞ 0 ∞ ∞ 0 ∞ ∞ ∞ ∞ ∞	∞ 0 ∞ ∞ ∞ ∞ ∞ ∞ 0 0 ∞ ∞ ∞ ∞ ∞ ∞ 0 ∞ ∞ 0 ∞ ∞ ∞ ∞ ∞
Set $M_4[1, 4] = \infty$ and reduce matrix if required	M_7 = Reduce matrix with L = 9

Cost of matrix M_7= Cost of node 7,

$C(7) = C(4) + L + M[1][4] = 28 + 9 + 0 = 37$

State space tree would be :



Node 6 will be the next E-node. On inclusion of <1, 4>, we get matrix M_6. In M_4, we have <2, 5> and <4, 2> edges with 0 value. So selecting any of the edge <a, b> from this list will make $M_6[a][b] = \infty$. Repeating previous step, we get

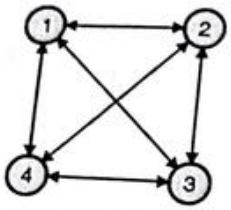
Edge	<2, 5>	<4, 2>
Increment in $\hat{c}$of right child	0	0

So we can select any of the edge. Thus the final path includes the edges <3, 1>, <5, 3>, <1, 4>, <4, 2>, <2, 5>, that forms the path 1 – 4 – 2 – 5 – 3 – 1. This path has cost of 28.

Ex. 6.2.3

Find the solution of the following travelling salesperson problem using Dynamic approach and Branch and Bound approach.

$$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$



Soln. :

Reduced matrix of given cost matrix is shown below :

Original cost matrix	M' = Row reduced matrix	M = Reduced matrix
∞ 10 15 20 → 10 5 ∞ 9 10 → 5 6 13 ∞ 12 → 6 8 8 9 ∞ → 8	∞ 0 5 10 0 ∞ 4 5 0 7 ∞ 6 0 0 1 0 ↓ ↓ ↓ ↓ 0 0 1 5	∞ 0 4 5 0 ∞ 3 0 0 7 ∞ 1 0 0 0 ∞

Reduction cost of the matrix is 35. This implies any route of the graph costs at least 35. Thus root of the tree has cost 35. We shall select the next edge such that exclusion of that leads to maximum ∞. In other words, select the edge which gives maximum probability of minimum cost of path on inclusion of that edge.

We can achieve this by considering one of the edge with reduced cost zero in reduced matrix M. In matrix M, edge <1, 2>, <2, 1>, <2, 4>, <3, 1>, <4, 1>, <4, 2> and <4, 3> has 0 reduction cost. If we select any of the edge <a, b> from this list, then resultant cost matrix M will have entry ∞ on position M[a][b].

If we include edge <1, 2>, set M[1][2] = ∞, and reduce M. This is done by subtracting 4 from row 1. So cost of right child will increase by 4.

If we include edge <2, 1>, set M[2][1] = ∞, and reduce M. Updated matrix is already reduced.

If we include edge <2, 4>, set M[2][4] = ∞, and reduce M. This is done by subtracting 1 from column 4. So cost of right child will increase by 1.

Thus we will have,

Edge	<1, 2>	<2, 1>	<2, 4>	<3, 1>	<4, 1>	<4, 2>	<4, 3>
Increment in $\hat{c}$of right child	4	0	1	1	0	0	3

Here, edge <1, 2> maximizes $\hat{c}$of right child, so let us select edge <1, 2>.

Set M[1][ ] = ∞ Set M[ ][2] = ∞ Set M[2][1] = ∞ Reduce the matrix if required x	∞ ∞ ∞ ∞ ∞ ∞ 3 0 0 ∞ ∞ 1 0 ∞ 0 ∞
Inclusion of <1, 2> in M	M_2 = Matrix is already reduced, L = 0

Cost of matrix M_2= Cost of node 2,

$C(2) = C(1) + L + M[2][1] = 35 + 0 + 0 = 35$

By excluding <1, 2> we get,

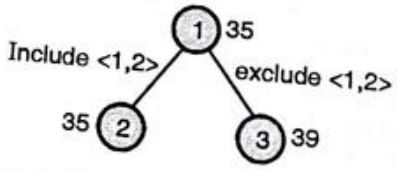
∞ ∞ 4 5 0 ∞ 3 0 0 7 ∞ 1 0 0 0 ∞	∞ ∞ 0 1 0 ∞ 3 0 0 7 ∞ 1 0 0 0 ∞
Set M[3, 1] = ∞ and reduce matrix if required	M_3 = Reduce matrix with L = 4

Cost of matrix M_3 = Cost of node 3,

$C(3) = C(1) + L + M[1][2]$

$= 35 + 4 + 0 = 39$

At this point, dynamic state space tree would look like,



On inclusion of <1, 2>, we get matrix M_2. In M_2, we have <2, 4>, <3, 1>, <4, 1> and <4, 3> edges with 0 value. So selecting any of the edge <a, b> from this list will make $M_2[a][b] = \infty$. Repeating previous step, we get

Edge	<2, 4>	<3, 1>	<4, 1>	<4, 3>
Increment in$\hat{c}$of right child	4	1	0	3

Here, edge <2, 4> maximizes $\hat{c}$of right child, so let us select edge <2, 4>.

Set $M_2[2][\ ] = \infty$ Set $M_2[\ ][4] = \infty$ Set $M_2[4][1] = \infty$ to avoid cycle Reduce the matrix if required x	∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞ 0 ∞ ∞ ∞ ∞ ∞ 0 ∞
Inclusion of <2, 4> in M_2	M_4 = Reduced matrix, L = 0

Cost of matrix M_4= Cost of node 4, $C(4) = C(2) + L + M_2[2][4] = 35 + 0 + 0 = 35$

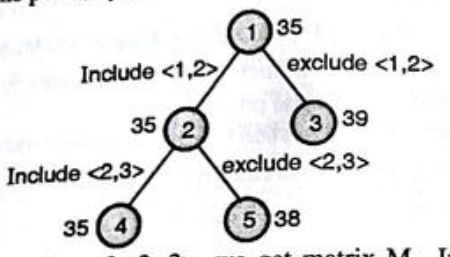
By excluding <2, 4> from M_2 we get,

∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞ 0 ∞ ∞ ∞ ∞ ∞ 0 ∞	∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞ 0 ∞ ∞ ∞ ∞ ∞ 0 ∞
Set $M_2[2][4] = \infty$ and reduce matrix if required	M_5 = Reduce matrix with L = 3

Cost of matrix M_5= Cost of node 5,

$C(5) = C(2) + L + M_2[2][4] = 35 + 3 + 0 = 38$

At this point, dynamic state space tree would look like,



On inclusion of <2, 3>, we get matrix M_4. In M_4, we have <3, 1> and <4, 3> edges with 0 value. So selecting any of the edge <a, b> from this list will make $M_4[a][b] = \infty$. Repeating previous step, we get

Edge	<3, 1>	<4, 3>
Increment in $\hat{c}$ of right child	0	0

So we can select any of the edge. Thus the final path includes the edges <1, 2>, <2, 4>, <4, 3>, <3, 1>, that forms the path 1 – 2 – 4 – 3 – 1. This path has cost of 35.

6.2.7 Comparison Divide and Conquer, Dynamic Programming and Backtracking

→ **(May 17)**

Q. Compare divide and conquer, dynamic programming and backtracking approaches used for algorithm design. **MU - May 2017, 5 Marks**

Divide and Conquer	Dynamic Programming	Backtracking
Breaks down large problems into independent small problems	Breaks down large problems into dependent small problems	Solves the problem in top down approach by pruning the non-promising nodes in state space tree
Does not construct state space tree	Does not construct state space tree	Solution is represented using state space tree
Does not employ and constraints or heuristic cost function	Does not employ and constraints or heuristic cost function	Objective function or cost function is used to determine the next E node
Recursive in nature	Iterative in nature	Recursive in nature
Ex: Multiplying large integers, Merge sort Quick sort	Ex: Make a change Knapsack problem Assembly line scheduling	Ex: Graph coloring N-Queen Problem Sudoku

6.3 Exam Pack (University and Review Questions)

☞ **Syllabus Topic : General Method**

Q. Comment on model of computation : Backtracking. *(Ans. : Refer section 6.1.1(A))* (5 Marks) **(May 2014)**

Q. Write a short note on state space tree. *(Ans. : Refer section 6.1.1(B))* (5 Marks)

Q. What is a state space tree and with respect to state space tree explain the following terms:

(i) solution state (ii) state space

(iii) answer states (iv) static trees

(v) dynamic trees (vi) live node

(vii) bounding function (viii) state space tree

(ix) E-node.

(Ans. : Refer section 6.1.1(B))(10 Marks)

Q. Write a formulation of the recursive backing algorithm. *(Ans. : Refer section 6.1.2(A))* (5 Marks)

Q. Write recursive backtracking algorithm for the sum of subset problem. *(Ans. : Refer section 6.1.2(A))* (6 Marks)

Q. What is backtracking? Write a general iterative algorithm for backtracking. *(Ans. : Refer section 6.1.2(B))* (7 Marks)

☞ **Syllabus Topic : 8-Queen Problem**

Q. Write note on : N-Queen Problem. *(Ans. : Refer section 6.1.4)* (10 Marks) **(May 2014, Dec. 2014)**

Q. Explain 8 Queen problem. *(Ans. : Refer section 6.1.4)* (10 Marks) **(Dec. 2015)**

Q. Write a short note on 8-Queen problem.
(Ans. : Refer section 6.1.4) (10 Marks)
(May 2016, Dec. 2016)

☞ **Syllabus Topic : Sum of Subsets**

Q. Explain sum of subset problem.
(Ans. : Refer section 6.1.5) (5 Marks) **(Dec. 2015)**

Q. Write a recursive backtracking algorithm for sum of subset problem.
(Ans. : Refer section 6.1.5) (5 Marks)

Q. Write an algorithm of sum of subsets.
(Ans. : Refer section 6.1.5) (5 Marks)
(May 2014, May 2015, May 2016)

Ex. 6.1.4 (5 Marks) **(May 2014)**
Ex. 6.1.7 (10 Marks) **(May 2013)**
Ex. 6.1.8 (10 Marks) **(May 2015)**
Ex. 6.1.9 5 Marks) **(Dec. 2015, May 2016)**

☞ **Syllabus Topic : Graph Coloring**

Q. Explain Graph coloring problem using backtracking. Write algorithm for same.
(Ans. : Refer section 6.1.6)(10 Marks) **(May 2013)**

Q. State Graph coloring algorithm. Explain strategy used for solving it along with example.
(Ans. : Refer section 6.1.6) (10 Marks) **(Dec. 2013)**

Q. Write a short note on Graph coloring.
(Ans. : Refer section 6.1.6) (10 Marks) **(May 2016)**

Q. What are planar graphs?
(Ans. : Refer section 6.1.6(1)) (2 Marks)

Q. What is bipartite graph? How many colors are required to color bipartite graph?
(Ans. : Refer section 6.1.6(2)) (5 Marks)

☞ **Syllabus Topic : 15 Puzzle Problem**

Q. Comment on model of computation : Branch and Bound. *(Ans. : Refer section 6.2.1)* (5 Marks)
(May 2014)

Q. Write a short note on Branch and bound strategy.
(Ans. : Refer section 6.2.1) (10 Marks) **(Dec. 2015)**

Q. Describe the method with respect to Branch and Bound. *(Ans. : Refer section 6.2.1)* (7 Marks)

Q. Explain the branch and bound algorithmic strategy for solving the problem.
(Ans. : Refer section 6.2.1) (7 Marks)

Q. Differentiate Backtracking and Branch and Bound Method. Illustrate with an example of 4-Queen's Problem. *(Ans. : Refer section 6.2.2)* (5 Marks)

Q. What is the difference between backtracking approach and branch and bound approach.
(Ans. : Refer section 6.2.2)(5 Marks)

Q. What is LC search?
(Ans. : Refer section 6.2.4(A)) (2 Marks)

Q. Describe the control abstraction for LC search with respect to Branch and Bound.
(Ans. : Refer section 6.2.4(B)) (6 Marks)

Q. How does it help in finding a solution for branch and bound algorithm?
(Ans. : Refer section 6.2.4(B)) (6 Marks)

Q. Explain in detail Control abstraction for LC search.
(Ans. : Refer section 6.2.4(B)) (6 Marks)

Q. Describe bounding with respect to Branch and Bound.
(Ans. : Refer section 6.2.4(C)) (7 Marks)

Q. Explain the FIFO branch and bound with respect to branch and bound technique with a suitable example. *(Ans. : Refer section 6.2.4(D))*(10 Marks)

Q. Discuss the control abstraction for FIFO branch and bound. *(Ans. : Refer section 6.2.4(D))* (7 Marks)

Q. Explain 15-puzzle problem using branch and bound.
(Ans. : Refer section 6.2.5) (10 Marks) **(May 2013)**

Q. Write note on : 15-puzzle problem. (10 Marks)
(May 2014, Dec. 2014, May 2015, May 2016)

Q. Explain how branch and bound strategy can be used in 15 puzzle problem.
(Ans. : Refer section 6.2.5) (10 Marks) **(May 2017)**

☞ **Syllabus Topic : Travelling Salesman Problem**

Q. Explain travelling salesperson problem using branch and bound method.
(Ans. : Refer section 6.2.6) (10 Marks) **(May 2013)**

Q. Write note on : Travelling sales person problem.
(Ans. : Refer section 6.2.6)(10 Marks) **(May 2015)**

Q. What is travelling salesman problem?
(Ans. : Refer section 6.2.6) (6 Marks)

Q. Describe least cost search (LC Search) with respect to Branch and Bound.
(Ans. : Refer section 6.2.6(B)) (7 Marks)

Q. Explain the Least cost search with respect to branch and bound technique with suitable example.
(Ans. : Refer section 6.2.6(B)) (7 Marks)

Q. Compare divide and conquer, dynamic programming and backtracking approaches used for algorithm design.
(Ans. : Refer section 6.2.7) (5 Marks) **(May 2017)**

❑❑❑

Module 5

CHAPTER 7

String Matching

Syllabus Topics

The naïve string matching Algorithms, The Rabin Karp algorithm, String matching with finite automata, The knuth-Morris-Pratt algorithm.

7.1 Introduction

- String matching operation is core part in many text processing applications. Objective of string matching algorithm is to find pattern P from given text T. Typically |P| << |T|. In design of compilers and text editors, string matching operation is crucial. So locating P in T efficiently is very important.
- String matching problem is defined as follow: "Given some text string T[1....n] of size n, find all occurrences of pattern P[1...m] of size m in T."
- We say that P occurs in text T with number of shifts s, if $0 \le s \le n - m$ and $T[(s + 1) \ldots (s + m)] = P[1 \ldots m]$.

Consider the following example :

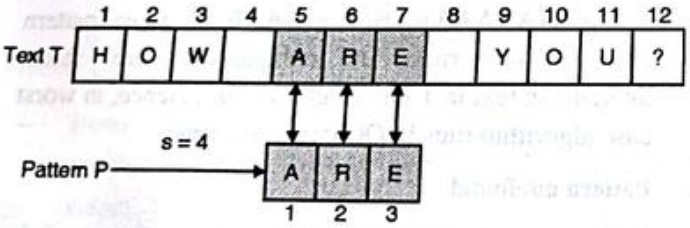


Fig. 7.1.1 : Pattern matching in text T

- In this example, pattern P = ARE is found in text T after four shifts.
- Classical application of string matching algorithm is to find particular protein pattern in DNA sequence.
- String may be encoded using set of character alphabets {a, b, ..., z}, binary alphabets {0, 1}, decimal alphabets {0, 1, 2, ..., 9}, DNA alphabets {A, C, G, T}. Encoding of string directly affects the efficiency of searching.
- In next sections, we will discuss and analyze few string matching algorithms.

Syllabus Topic : The Naïve String Matching Algorithms

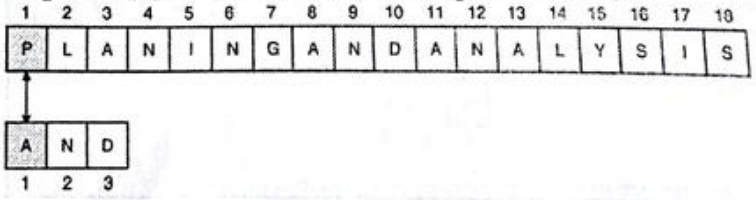
7.2 The Naïve String Matching Algorithms

→ **(May 14, Dec. 14, May 15, May 16)**

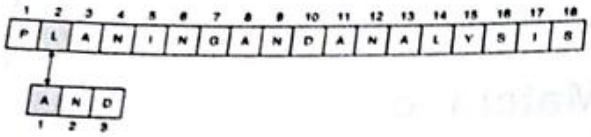
Q. Explain naïve string matching algorithm with example. **MU - May 2014, Dec. 2014, May 2015, May 2016, 10 Marks**

- This is simple and inefficient brute force approach. It compares first character of pattern with searchable text. If match is found, pointers in both strings are advanced. If match is not found, pointer of text is incremented and pointer of pattern is reset. This process is repeated till the end of text.
- Naive approach does not require any pre-processing. Given text T and pattern P, it directly starts comparing both strings character by character.
- After each comparison, it shifts pattern string *one position* to the right.
- Following example illustrates the working of naive string matching algorithm. Here,
  T = PLANINGANDANALYASIS and P = AND
- Here, t_i and p_j are indices of text and pattern respectively.

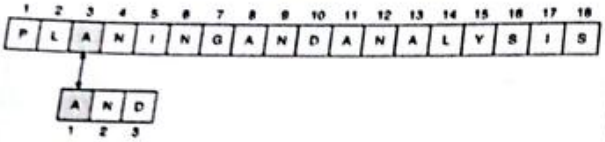
Step 1 : T[1] ≠ P[1], so advance text pointer, i.e. t_i++.



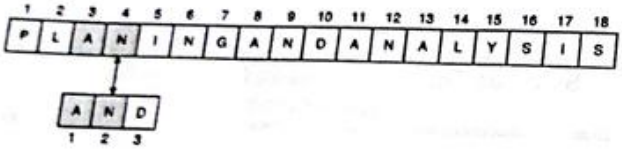
Step 2 : T[2] ≠ P[1], so advance text pointer i.e. t_i++



Step 3 : T[3] = P[1], so advance both pointers i.e. t_i++, p_j++

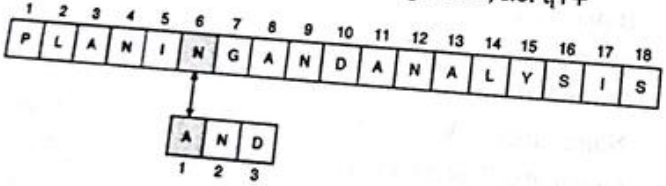


Step 4 : T[4] = P[2], so advance both pointers, i.e. t_i++, p_j++

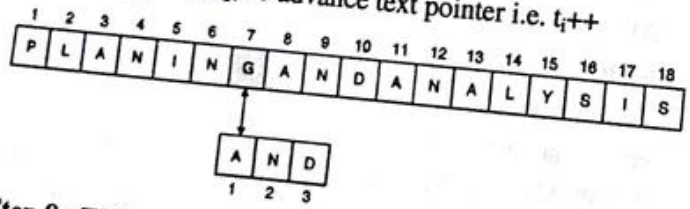


Step 5 : T[5] ≠ P[3], so advance text pointer and reset pattern pointer, i.e. t_i++, $p_j = 1$

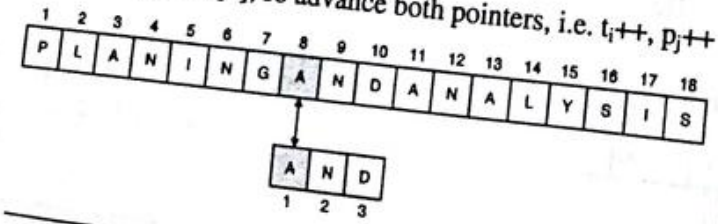
Step 6 : T[6] ≠ P[1], so advance text pointer, i.e. t_i++



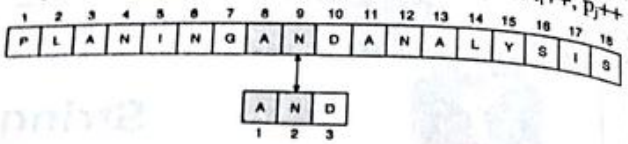
Step 7 : T[7] ≠ P[1], so advance text pointer i.e. t_i++



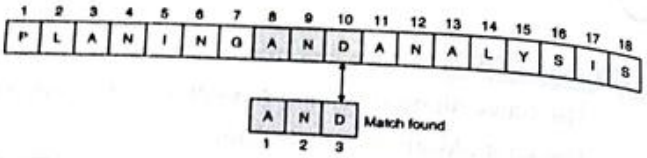
Step 8 : T[8] = P[1], so advance both pointers, i.e. t_i++, p_j++



Step 9 : T[9] = P[2], so advance both pointers, i.e. t_i++, p_j++



Step 10 : T[10] = P[3], so advance both pointers, i.e. t_i++, p_j++



This process continues till the end of text string.

Algorithm for naive string matching approach is described below :

```
Algorithm NAÏVE_STRING_MATCHING(T, P)
// T is the text string of length n
// P is the pattern of length m
for i ← 0 to n – m do
    if P[1... m] == T[i+1...i+m] then
        print "Match Found"
    end
end
```

Complexity analysis

There are two cases of consideration

1. **Pattern found**

- Worst case occurs when pattern is at last position and there are spurious hits all the way.
- Example,
  T = AAAAAAAAAAB, P = AAAB. To move pattern one position right, m comparisons are made. Searchable text in T has length (n – m). Hence, in worst case algorithm runs in O(m*(n – m)) time.

2. **Pattern not found**

- In best case, searchable text does not contain any of the prefix of pattern. Only one comparison requires moving pattern one position right.
  Example,
  T = ABABCDBCAC, P = XYXZ. Algorithm does O(n – m) comparisons.
- In worst case, first (m – 1) characters of pattern and text are matched and only last character does not match.
  Example,
  T = AAAAAAAAAAAAC,
  P = AAAAB. Algorithm takes O(m*(n – m)) time.

Syllabus Topic : The Rabin Karp Algorithm

7.3 The Rabin Karp Algorithm

→ **(May 14, May 15, Dec. 15, May 16)**

Q. Explain different string matching algorithms.
MU - May 2014, May 2016, 10 Marks

Q. Write a short note on Rabinkarp algorithm.
MU - May 2015, Dec. 2015, 10 Marks

- Comparing numbers is easier and cheaper than comparing strings. Rabin Karp algorithm represents strings in numbers.
- Suppose p represents values corresponding to pattern P[1...m] of length m. And t_s represents values of m-length substrings T[(s + 1) ... (s + m)] for s = 0, 1, 2, ..., n – m.
- We can compute p in O(m) time and all t_s can be computed in O(n – m + 1) time.
- Rabin Karp algorithm is based on hashing technique. It first computes the hash value of p and t_s.
- If hash values are same, i.e if hash(p) = hash(t_s), we check the equality of inverse hash similar to naïve method. If hash values are not same, no need to compare actual string.
- On hash match, actual characters of both strings are compared using brute force approach. If pattern is found then it is called *hit*. Otherwise it is called *spurious hit*.
- For example, let us consider the hash value of string T = ABCDE is 38 and hash of string P = ABCDX is 71. Clearly, hash values are not same, so strings cannot be same. Brute force approach does five comparisons where as Rabin Karp dose only one comparison.
- However, same hash value does not ensure the string match. Two different strings can have same hash values. That is why we need to compare them character by character on hash hit.
- Fig. 7.3.1 shows the difference between Brute Force and Rabin Karp approach.

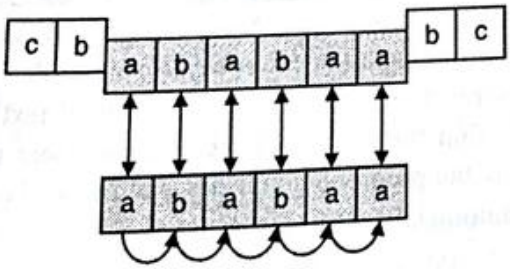


Fig. 7.3.1(a) : Brute force approach

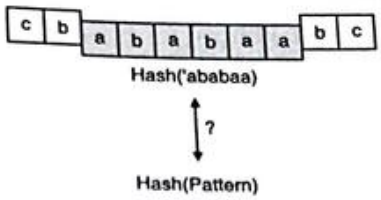


Fig. 7.3.1(b) : Rabin Karp Approach

- Given pattern P[1...m], we can derive its numeric value p in base d in O(m) time as follow: p = P[m] + d (P[m – 1] + d(P[m – 2] + ... + d(P[2] + dP[1]) ...))
- Similarly, we can derive numeric value of first substring t_0 of length m from text T[1...n] in O(m) time. Remaining all t_i, i = 1, 2, 3, ..., n – m, can be derived in constant time.
- Given t_s, we can compute t_s+1 as,

  $t_{s+1} = d(t_s - d^{m-1} T[s + 1]) + T[s + m + 1]$
- Assume that T = [4, 3, 1, 5, 6, 7, 5, 9, 3] and P = [1, 5, 6]. Here length of P is 3, so m = 3. Consider that, for given pattern P, its value p = 156, and

  $t_0 = 431$.

  $t_1 = 10(431 - 10^2 T[1]) + T[4] = 10(431 - 400) + 5$
  $= 315$
- Values of p and t_s may be too large to process. We can reduce these values by taking it's modulo with suitable number q. typically, q is prime number.
- Mod function has some nice mathematical property.
  - [(a mod k) + (b mod k)] mod k = (a + b) mod k
  - (a mod k) mod k = a mod k
- Computing hash value of every subsequence of m character of text may turn out to be time consuming. However, bit of mathematics makes it easier.
- Suppose t_s represents decimal value of substring T[(s + 1) ... (s + m)]. If hash of t_s is known, hash value can directly be derived for t_{s+1} as follow :

  Hash for t_{s+1} = (d*(t_s – T[s + 1]h) + T[s + m + 1]) mod q, Where $h = d^{m-1} \bmod q$

Two facts

$t_s = p \bmod q$, does not mean $t_s = p$

$t_s \neq p \bmod q$, means $t_s \neq p$

If p = 45365, t_s = 64371 and q = 11,

45365 mod 13 = 8

64371 mod 13 = 8

- From above example, it is clear that two *different* strings might have same mod. We can reject all negative tests to rule out all possible invalid shifts. And all positive tests must be validated to overcome spurious hits.
- Spurious hits reduce with larger value of q, and it increases with smaller q.

Algorithm

```
Algorithm RABIN_KARP(T, P)
// T is text of length n
// P is pattern of length m

h ← power(d, m − 1) mod q
p ← 0
t0 ← 0
for i ← 1 to m do
    p ← (d*p + P[i]) mod q
    t0 ← ((d * t0) + T[i]) mod q
end

for s ← 0 to n − m do
    if p == ts then
        if P[1...m] == T[s+1 ... s + m] then
            print "Match found at shift", s
        end
    end
    if s < (n − m) then
        ts + 1 ← (d*(ts − T[s + 1]) * h ) + T[s + m + 1]
    end
end
```

Complexity analysis

- Rabin karp algorithm is randomized algorithm. In most of the cases, it runs in linear time, i.e. in O(n). However, worst case of rabin karp algorithm is as bad as naïve algorithm, i.e. O(mn), but it's rare.
- It can happen only when prime number used for hashing is very small.

Ex. 7.3.1

Explain spurious hits in Rabin-Karp string matching algorithm with example. Working modulo q = 13, how many spurious hits does the Rabin-Karp matcher encounter in the text T = 2359023141526739921 when looking for the pattern P = 31415 ?

Soln. :

Given pattern P = 31415, and prime number q = 13

P mod q = 31415 mod 13 = 7

Let us find hash value for given text :

T[1...5] = 23590 mod 13 = 8

T[2...6] = 35902 mod 13 = 9

T[(m − 4) ... m] = 39921 mod 13 = 11

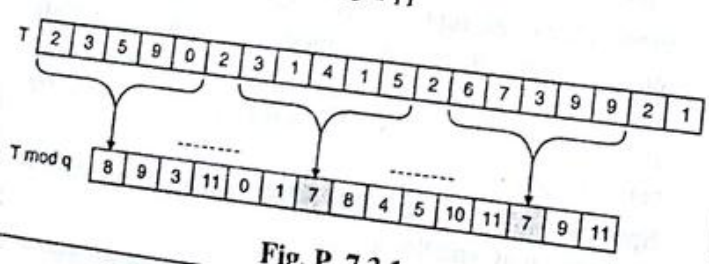


Fig. P. 7.3.1

- Hash value of pattern P is 7. We have two such values in hash(T). So there may be a spurious hit or actual string. In given text T, one hit is actual match and one is spurious hit.
- If hash of pattern and any substring in text is same, we have two possibilities:
  - **Hit :** Pattern and text are same.
  - **Spurious hit :** Hash value is same but pattern and corresponding text is not same.

Here, m = length of pattern = 5.

Consider t_s = 23590. Next value t_{s+1} is derived as,

$t_{s+1} = 10(t_s - 10^m * T[s+1] + T[s + m + 1])$

$= (10*23590) - 10^5 * 2 + 2$

$= 235900 - 200000 + 2$

$= 35902$

$t_{s+2} = 10(t_{s+1} - 10^m * T[s+2] + T[s + m + 2])$

$= (10*35902) - 10^5 * 3 + 3$

$= 359020 - 300000 + 3$

$= 59023$

- In same way, we can compute the next t_{s+i} using incremental approach.
- Rabin Karp algorithm matches hash value, rather than directly comparing actual string value. If hash value of pattern P and hash value of subsequence in string T are same, actual value of strings are compared using brute force approach. Like t_{s+1}, we can also derive hash value incrementally as shown below.

Calculation for hash of 14152

If t_s is known, hash value can directly be derived for t_{s+1} as follow.

Hash for $t_{s+1} = (d*(t_s - T[s+1]h) + T[s + m + 1]) \bmod q$

$= 10*(31415 - (3*10^4 \bmod 13)) + 2 \bmod 13$

$= 10(31415 - 9) + 2 \bmod 13$

$= 314062 \bmod 13$

$= 8$

Syllabus Topic : String Matching with Finite Automata

7.4 String Matching with Finite Automata

→ **(May 17)**

Q. Write and explain string matching with finite automata with an example. **MU - May 2017, 10 Marks**

- Idea of this approach is to build finite automata to scan text T for finding all occurrences of pattern P.
- This approach examines each character of text exactly once to find the pattern. Thus it takes linear time for matching but preprocessing time may be large.
- Finite automata is defined by tuple

$M = \{Q, \Sigma, q_0, F, \delta\}$

Where,

Q = Set of states in finite automata

Σ = Set of input symbols

q_0 = Initial state

F = Set of final states

δ = Transition function define

d as $\delta : Q \times \Sigma \rightarrow Q$

For example

Q = {0, 1, 2, 3, 4, 5},

q_0 = 0, F = {2, 3},

Σ = {a, b}, And transition function :

δ	a	b
0	1	3
1	2	3
2	2	4
3	1	5
4	2	5
5	5	5

- Graphically we can represent this finite automaton as shown in Fig. 7.4.1.
- Finite automata are widely used in compilers and text processors for string matching. Given the string S over alphabet set Σ. Finite automata, Starts with input state q_0
- Reads the input string character by character and changes the state according to transition function.

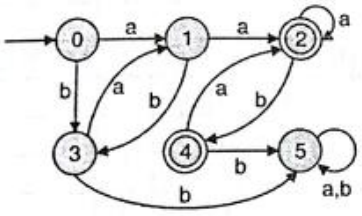


Fig. 7.4.1

- It accepts the string if a finite automaton ends up in one of the final / accepting states.
- It rejects the string if a finite automaton does not end up in final state.

Let us trace some string for above given finite automata. Consider the string S = abababaaabab. Following table shows the transitions on every input symbol.

Input		a	b	a	b	a	b	a	a	a	b	a	b
State	0	1	3	1	3	1	3	1	2	2	4	2	4

After scanning entire string, finite automata is in final state, so string is accepted by the automata. Consider the string S = bababbb.

Input		b	a	b	a	b	a	b	b
State	0	3	1	3	1	3	1	3	5

The state 5 is not accepting state, so string is not accepted by given finite automata.

Let us extend this concept of finite automata for pattern matching.

Let us consider the text $T = t_1, t_2, t_3 \ldots t_n$ and pattern $P = p_1, p_2, p_3 \ldots p_m$.

For the string of length m. FA will the have m + 1 states, numbered as 0, 1, 2, m.

State 0 will be initial / start state and state m will be the only accepting / final state.

If first k characters of pattern match with the text, FA will be in k^{th} state.

$T : t_1\ t_2\ t_3 \ldots$ | t_j | t_{j+1} | t_{j+2} | t_{j+k-1} | t_{j+k} …

$P :$ | p_1 | p_2 | p_3 | p_k | p_{k+1}

Next character t_{j+k} matches with p_{k+1}, implies first k + 1 characters are matched and FA goes in $(k+1)^{st}$ state.

$\therefore\ \delta(k, p_{k+1}) = k + 1$

If next character t_{j+k} does not match with p_{k+1}, then FA enters in one of the 0 to k state

Keep shifting pattern right till there is a match or p is exhausted.

$T : t_1\ t_2\ t_3 \ldots\ t_j$ | t_{j+1} | t_{j+2} | $t_{j+k\pm2}$ | $t_{j+k\pm2}$ | t_{j+k}

$P :$ | p_1 | p_2 | p_{k-2} | p_{k-1} | p_k

If match found, enter in state k, else continue

$T : t_1\ t_2\ t_3 \ldots\ t_j\ t_{j+1}$ | t_{j+2} | t_{j+k-2} | t_{j+k-1} | t_{j+k} …

$P :$ | p_1 | p_2 | p_{k-2} | p_{k-1} …

If match found, enter in state k – 1, else continue

⋮

$T : t_1\ t_2\ t_3 \ldots\ t_{j+1}\ \ t_{j+2} \cdots\ \ t_{j+k-2}\ \ t_{j+k-1}$ | t_{j+k} ….

$P :$ | p_1

If match found, enter in state 1, else state 0

If FA reaches to state m, pattern is found and computation stops.

- Search time for this approach is linear i.e. O(n). Each character in text is examined exactly once.
- Algorithm for pattern matching using finite automata is shown below :

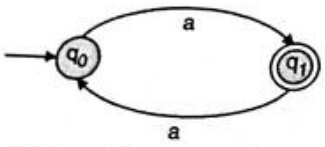
☞ **Algorithm**

```
Algorithm FINTE_AUTOMATA(T, P)
// T is text of length n
// P is pattern of length m

state ← 0                    // Initial state is 0
```

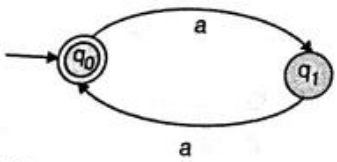
```
for i ← 1 to n do
    state ← δ(state, t_i)        // Return the new state after reading character t_i
    if state == m then
        print "Match found on position", i - m + 1
    end
end
```

Complexity analysis

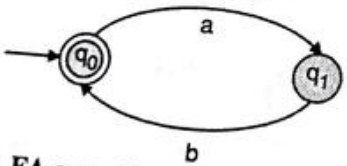
- Construction of finite automata takes $O(m^3|\Sigma|)$ time in worst case, where m is the length of pattern and $|\Sigma|$ is number of input symbols.
- This approach examines each character of text exactly once to find the pattern. So it takes linear time for matching. But pre-processing time may be large. Constructing automata is the difficult part for pattern matching.
- It runs fairly in O(n) time without considering time spend to build transition function.
- Finite automata for few patterns are shown in Fig. 7.4.2



FA accepting string a^{2n-1}



FA accepting string a^{2n}



FA accepting string $(ab)^n$

Fig. 7.4.2

Ex. 7.4.1

Construct and show simulation of finite automata for matching the pattern P = abb for text T = ababbaababba

Soln. : Finite automata that can accept the string abb is shown in Fig. P. 7.4.1.

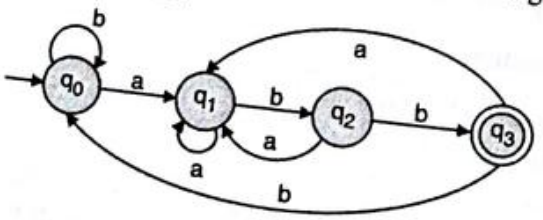


Fig. P. 7.4.1

Moves made by the FA to accept the string abb are shown in Fig. P. 7.4.1(a) :

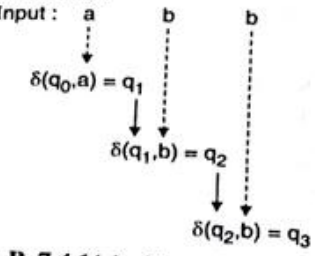


Fig. P. 7.4.1(a) : Transition steps

Transition table for above finite automata is given as,

Table P. 7.4.1

Input \ State	a	b
q_0	q_1	q_0
q_1	q_1	q_2
q_2	q_1	q_3
q_3	q_1	q_0

Simulation of given data on this FA is shown in Table P. 7.4.1(a).

Table P. 7.4.1(a)

a	b	a	b	b	a	a	b	a	b	b	a	Comment
a	b	b										
	a	b	b									
		a	b	b								Match found
			a	b	b							
				a	b	b						
					a	b	b					
						a	b	b				
							a	b	b			
								a	b	b		Match found
									a	b	b	

Syllabus Topic : The Knuth-Morris-Pratt Algorithm

7.5 The Knuth-Morris-Pratt Algorithm

(May 13, Dec. 14, Dec. 15, May 17)

Q. Explain and write Knuth-Morris Pratt algorithm. Explain with an example.
MU - May 2013, Dec. 2015, 10 Marks

Q. To implement the Knuth-Morris-Pratt, string matching algorithm.
MU - Dec. 2014, 10 Marks

Q. Write a short note on Knuth-Morris-Pratt's Pattern Matching.
MU - May 2017, 7 Marks

- This algorithm is also known as KMP (Knuth-Morris-Pratt) algorithm. This is the first linear time algorithm for string matching. It utilizes the concept of naïve approach in some different way. This approach keeps track of matched part of pattern.
- Main idea of the algorithm is to avoid computation of transition function δ and reducing useless shifts performed in naïve approach.
- By maintaining the information of already processed string, this approach reduce the time to O(m + n). That is, in worst case, KMP algorithm examines all characters of input pattern and text exactly once. This improvement is achieved by using auxiliary function π. Naïve approach was not storing anything it has already matched, and that's why it was shifting the pattern right side by just one. Computation of auxiliary function π take O(m) time.
- Function π is very useful, it computes the number of shifts of pattern by finding pattern matches within itself.

Sliding rule

- Naïve approach slides the pattern by one right side. KMP approach slides the string multiple steps and reduces comparisons. Let us consider pattern $P = p_1p_2p_3 \cdots p_{m-1}p_m$.
- For any string P, sub string P' of *first* i characters is called prefix of P, where i = 1 to m. Null string also belongs prefix of P. String P' is proper prefix of P if and only if $P' \neq P$.
- Similarly, for any string P, sub string P' of *last* i characters is called suffix of P, where i = 1 to m. Null string also belongs suffix of P. String P' is proper suffix of P if and only if $P' \neq P$.
- Consider the pattern P = abcbdabc ; Prefix of P are : {ϕ, a, ab, abc, abcd, abcda, abcdab, abcdabc}, where abcdabc is not proper prefix of P.
  Suffix of P are : {ϕ, c, bc, abc, dabc, cdabc, bcdabc, abcdabc}, where abcdabc is not proper suffix of P.
- Suppose the pattern P[1….q] is matched with the text at T[(i – q + 1)… i]. First mismatch occurs at position P[q + 1] and T[i + 1]. Slide the pattern right by x, where x is the length of longest proper prefix of P[1…q] and also longest suffix of P[1…q]

Working principle

- If P[1…q] matches with T[(i – q + 1) …i], there are two possibilities :
- If P[q + 1] = T[i + 1], matched string length is increased by one. If q = m, complete match is found and algorithm terminates.
- If P[q + 1] ≠ T[i + 1], then the slide pattern P to right by longest common suffix and prefix of P[1…q].

Algorithm PREFIX_FUNCTION(P)
// P is the pattern of length m

```
π[1] ← 0
k ← 0
for q ← 2 to m do
    while k > 0 and P[k+1] ≠ P[q] do
     k ← π[k]
    end
    if P[k+1] == P[q] then
     k ← k + 1
    end

    π[q] ← k
end
return π
```

Following algorithm finds the pattern within text :

```
KMP_STRING_MATCHING(T, P)
// T is text of length n
// P  is pattern of length m

π ← PREFIX_FUNCTION(P)

q ← 0
for i ← 1 to n do
    while q > 0 and P[q + 1] ≠ T[i] do
     q ← π[q]
    end
    if P[q + 1] == T[i] then
     q ← q + 1
    end
end
if q == m then
    print "Pattern found with shift", i – m
end
q ← π[q]
```

Complexity analysis

- Function π is computed in O(m) time where as actual string matching routine scans n characters. So worst case running time of algorithm is given by O(m + n).

Ex. 7.5.1

Compute the prefix function for pattern ababaca.

Soln. :

Initially, k = 0, q = 2, π[1] = 0 and m = 7

i	1	2	3	4	5	6	7
P[i]	a	b	a	b	a	c	a
π[i]	0						

Iteration 1 : $q = 2, k = 0$

k is not greater than 0, so skip while loop

$P[k+1] = P[1] = a$ and $P[q] = P[2] = b$

$P[k+1] \neq P[q]$, so $\pi[q] = k \rightarrow \pi[2] = 0$

i	1	2	3	4	5	6	7
P[i]	a	b	a	b	a	c	a
π[i]	0	0					

Iteration 2 : $q = 3, k = 0$

k is not greater than 0, so skip while loop

$P[k+1] = P[1] = a$

and $P[q] = P[3] = a$

$P[k+1] = P[q]$.

so $k = k + 1 \rightarrow k = 1$

$\pi[q] = k$

$\rightarrow \pi[3] = 1$

i	1	2	3	4	5	6	7
P[i]	a	b	a	b	a	c	a
π[i]	0	0	1				

Iteration 3 : $q = 4, k = 1$

k is greater than 0, so enter in to while loop

$P[k+1] = P[2] = b$ and $P[q] = P[4] = b$.

$k > 0$ but $P[k+1] = P[q]$ *// break while loop*

$P[k+1] = P[q]$. so $k = k + 1 \rightarrow k = 2$

$\pi[q] = k \rightarrow \pi[4] = 2$

i	1	2	3	4	5	6	7
P[i]	a	b	a	b	a	c	a
π[i]	0	0	1	2			

Iteration 4 : $q = 5, k = 2$

k is greater than 0, so enter in to while loop

$P[k+1] = P[3] = a$ and $P[q] = P[5] = a$,

$k > 0$ but $P[k+1] = P[q]$ *// break while loop*

$P[k+1] = P[q]$.

so $k = k + 1 \rightarrow k = 3$

$\pi[q] = k \rightarrow \pi[5] = 3$

i	1	2	3	4	5	6	7
P[i]	a	b	a	b	a	c	a
π[i]	0	0	1	2	3		

Iteration 5 : $q = 6, k = 3$

k is greater than 0, so enter in to while loop

$P[k+1] = P[4] = b$ and $P[q] = P[6] = c$,

$k > 0$

and $P[k+1] \neq P[q]$ *// Enter in while loop*

$k = \pi[k] \rightarrow k = \pi[3] = 1$

Now, $P[k+1] = P[2] = b$

and $P[q] = c$

Still $k > 0$ and $P[k+1] \neq P[q]$

// Enter in while loop again

$k = \pi[k] \rightarrow k = \pi[1] = 0$

As $k = 0$,

While loop breaks now

$P[k+1] = P[1] = a$

and $P[q] = P[6] = c$

$P[k+1] \neq P[q]$.

So, $\pi[q] = k \rightarrow \pi[6] = 0$

i	1	2	3	4	5	6	7
P[i]	a	b	a	b	a	c	a
π[i]	0	0	1	2	3	0	

Iteration 6 : $q = 7, k = 0$

k is not greater than 0, so skip while loop

$P[k+1] = P[1] = a$ and $P[q] = P[7] = a$

$P[k+1] = P[q]$, so $k = k + 1 = 1$

$\pi[q] = k \rightarrow \pi[7] = 1$

i	1	2	3	4	5	6	7
P[i]	a	b	a	b	a	c	a
π[i]	0	0	1	2	3	0	1

Ex. 7.5.2

Check if pattern P = abababca exits in text T = bacbababaabcbab.

Soln. :

The π function for given pattern would be,

i	1	2	3	4	5	6	7	8
P[i]	a	b	a	b	a	b	c	a
π[i]	0	0	1	2	3	4	0	1

- The first partial match is found on second position. Length of this partial match is 1. $\pi[1] = 0$, so we won't be able to skip ahead. We will continue matching by moving pattern right by one.

b	a	c	b	a	b	a	b	a	a	b	c	b	a	b

a	b	a	b	a	b	c	a

- Next partial match is found on position 5. Length of this match is 5. Next character of P and T does not match. . $\pi[5] = 3$, means we get skip ahead.

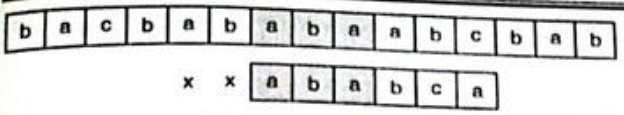
b	a	c	b	a	b	a	b	a	a	b	c	b	a	b

a	b	a	b	a	b	c	a

Skip = partial length - π[partial length]

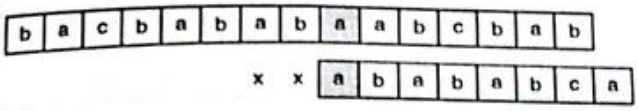
= $5 - \pi[5] = 5 - 3 = 2$

- So we can skip two characters ahead.
- Next partial match is found on position 7. Length of this match is 3. Next character of P and T does not match. $\pi[3] = 1$, means we get skip ahead. Cross indicates skipped characters.



Skip = partial length - π[partial length]
= 3 - π[3] = 3 – 1 = 2

– So we can skip two characters ahead.



– Now, pattern is longer than the remaining text, so match does not found.

7.6 Exam Pack (University Questions)

☞ **Syllabus Topic : The Naïve String Matching Algorithms**

Q. Explain naïve string matching algorithm with example. *(Ans. : Refer section 7.2)* (10 Marks)

(May 2014, Dec. 2014, May 2015, May 2016)

☞ **Syllabus Topic : The Rabin Karp Algorithm**

Q. Explain different string matching algorithms.

(Ans. : Refer section 7.3) (10 Marks)

(May 2014, May 2016)

Q. Write a short note on Rabinkarp algorithm.

(Ans. : Refer section 7.3) (10 Marks)

(May 2015, Dec. 2015)

☞ **Syllabus Topic : String Matching with Finite Automata**

Q. Write and explain string matching with finite automata with an example.

(Ans. : Refer section 7.4) (10 Marks) **(May 2017)**

☞ **Syllabus Topic : The Knuth-Morris-Pratt Algorithm**

Q. Explain and write Knuth-Morris Pratt algorithm. Explain with an example.

(Ans. : Refer section 7.5) (10 Marks)

(May 2013, Dec. 2015)

Q. To implement the Knuth-Morris-Pratt, string matching algorithm.

(Ans. : Refer section 7.5) (10 Marks) **(Dec. 2014)**

Q. Write a short note on Knuth-Morris-Pratt's Pattern Matching. *(Ans. : Refer section 7.5)* (7 Marks)

(May 2017)

❑❑❑

CHAPTER 8

Non Deterministic Polynomial Algorithms

Syllabus Topics

Polynomial time, Polynomial time verification NP Completeness and reducibility NP Completeness Proofs Vertex Cover Problems Clique Problems.

Basic Definitions

Q. Explain the following:
(i) Computational complexity
(ii) Decision problems
(iii) Deterministic and non-deterministic algorithms
(iv) Complexity classes
(v) Intractability. **(5 Marks)**

We can classify the problem in one of the following categories :

1. The problem which cannot be even defined in a proper way.
2. The problem which can be defined but cannot be solved.
3. The problem which can be solved theoretically, but computationally they are not feasible. The algorithm takes very long time to solve such problems, and the time is practically not acceptable. For example, cracking the password of 256 characters by brute force method may take years.

Definition

Problem is called **intractable** or **infeasible** if it takes too long time to be solved. A solution of such problems has no practical application.

4. Problems which can be solved theoretically and practically in a reasonable amount of time. For such problems, there exists a deterministic Turing machine that can solve the problem in O (p(n)) time, where p(n) is polynomial in n, and n is a problem size.

Definition

If the problem is solvable in polynomial time, it is called **tractable**. Such problems are denoted by P problems.

5. The problem which is not known whether it is in P or not in P. Such problems falls somewhere in between class 3 and 4.

Definition

- The problem is said to be **decision problem** if they produce output "Yes" or "No" for given input. An algorithm which solves the decision problem is called **decision algorithm**.
- An optimization **problem** aims for the best solution from the set of all feasible solutions. An algorithm which solves optimization problem is called **optimization algorithms**.
- Optimization algorithm seeks to find best profit or least cost solution. Decision problems are simpler than optimization problem.

- **Computational complexity :** Computational problems have infinite instances. Each instance in the set contains the solution. Typically, the solution to the problem in computational complexity is Boolean i.e. 'Yes' or 'No'.
- The computational problem can be **function problem** too. Solution to such problem differs on each execution even for the same input. So such problems are more complex than decision problems.

Definitions

- Complexity **classes** are set of problems of related complexity, like P problems, NP problems, Decision problems, Optimization problems etc. The complexity of any problem in given class falls within certain range.
- Problems which takes practically unacceptable time i.e. very long time to be solved are called **intractable problems**.
- If the running time of the algorithm is bounded to O(p(n)), where p(n) is some polynomial in n, where n represents the size of the problem, we say that the problem has **polynomial time complexity**.
- **Satisfiability** Problem is to check the correctness of

assignment. Satisfiability problem finds out whether for given input an expression is true for that assignment.

- **Reducibility :** Let P_1 and P_2 are two problems and there exists some deterministic algorithm A for P_1 such that P_1 can be solved in polynomial time using A. If the same algorithm can solve the problem P_2 then we can say that P_2 is reducible to P_1.

Syllabus Topic : Polynomial Time

8.1 Polynomial Time

Q. What do you mean by P problems? Give an example. (5 Marks)

Q. Write a short note on P problems. (5 Marks)

Definition

- **P problems** are set of problems that can be solved in polynomial time by deterministic algorithms.
- P is also known as **PTIME** or **DTIME** complexity class.

- They are simple to solve, easy to verify and take computationally acceptable time for solving any instance of the problem. Such problems are also known as *"tractable"*.
- In the worst case, searching an element from the list of size n takes n comparisons. The number of comparisons increases linearly with respect to input size. So linear search is P problem.
- In practice, most of the problems are P problems. Searching an element in the array (O(n)), inserting an element at the end of linked list (O(n)), sorting data using selection sort($O(n^2)$), finding height of tree ($O(\log_2 n)$), sort data using merge sort($O(n\log_2 n)$), matrix multiplication $O(n^3)$ are few of the examples of P problems.
- An algorithm with $O(2^n)$ complexity takes double time if it is tested on a problem of size (n + 1). Such problems do not belong to class P.
- It excludes all the problems which cannot be solved in polynomial time. Knapsack problem using brute force approach cannot be solved in polynomial time. Hence, it is not P problem.
- There exist many important problems whose solution is not found in polynomial time so far, nor it has been proved that such solution does not exist. TSP, Graph colouring, partition problem, knapsack etc. are examples of such class.

Examples of P problem

1. Insertion sort
2. Merge sort
3. Linear search
4. Matrix multiplication
5. Finding minimum and maximum element from array

Syllabus Topic : Polynomial Time Verification

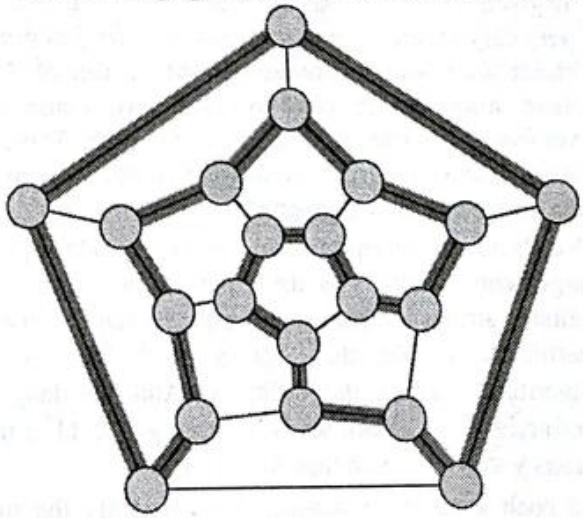
8.2 Polynomial Time Verification

Q. Write short note on Hamiltonian cycle. (5 Marks)

- Polynomial time verification checks the correctness of given solution in polynomial time. NP problems cannot derive the solution in polynomial time but given the solution, it can be verified in polynomial time.
- For problem instance <G, u, v, k>, we can check whether there exists a path p from vertex u to v of length at most k. And if so, p is called certificate for the given instance.
- Finding a path from u to v can be done in linear time and verifying the certificate also takes the same time. For this particular problem, verifying certificate takes as long as solving the problem.
- However, in practice there exist many problems for whom polynomial time solution is not known but still given the solution, it can be verified in polynomial time. Consider the following problem:

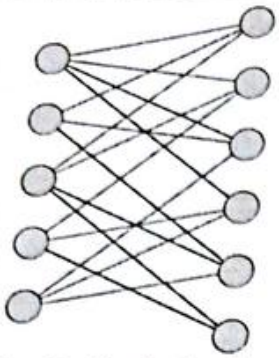
Hamiltonian cycle

- The hamiltonian cycle of undirected graph G = <V, E> is the cycle containing each vertex in V.
- If graph contains a Hamiltonian cycle, it is called Hamiltonian graph otherwise it is non-Hamiltonian.



(a) Hamiltonian Graph

Fig. 8.2.1 Cont...



(b) Non Hamiltonian Graph

Fig. 8.2.1 : Hamiltonian and Non-Hamiltonian graph

The dodecahedron shown in Fig. 8.2.1(a) is Hamiltonian, and the Hamiltonian cycle is shown with thick grey lines. Whereas the bipartite graph with an odd number of vertices as shown in Fig. 8.2.1(b) is a Non-Hamiltonian graph.

- One way of checking if the graph is Hamiltonian or not is to list out all possible permutations of vertices and check them one by one. There are m! different permutations of m vertices, and hence the running time of algorithm would be $\Omega(m!)$ time. By encoding the graph using its adjacency matrix representation, we can reduce the time to $\Omega(\sqrt{N}) = \Omega(2^{\sqrt{n}})$ which is not polynomial.
- Here n represents the length of encoding of graph G. Thus, the given problem cannot be solved in polynomial time.

Verification

- If given the solution string of Hamiltonian graph, it is very easy to prove the correctness of it. We just need to check if the solution contains all the vertices of V and there must be an edge between two consecutive vertices. This can be done in $O(n^2)$ time. Thus, the solution can be verified in polynomial time, where n is the length of encoded graph G.

  Verification algorithm A(x, y) is defined by two arguments, where x is the input string and y is the binary string, called *certificate*. If there exists a certificate y such that A(x, y) = 1, we say that algorithm verifies the string x. And the language defined by the algorithm is, $L = \{x \in \{0, 1\}^* : \text{there exists } y \in \{0, 1\} \text{ such that } A(x, y) = 1\}$
- For each legal string $x \in L$, A must verify the string and produce the certificate y. And for any string x which is not in L, A must not verify the string, and no certificate can prove that $x \in L$. For example if the graph is Hamiltonian, the proof can be checked in polynomial time as discussed earlier. But no vertex sequence should exist which can fool the algorithm to prove the non-hamiltonian graph as a Hamiltonian.

Syllabus Topic : NP-Completeness and Reducibility

8.3 NP-Completeness and Reducibility

Q. What is Reduction in NP-completeness proofs? What are types of reductions? **(7 Marks)**

Q. Explain Polynomial Time Algorithm. **(5 Marks)**

Definition

- The **polynomial time reduction** is a way of solving problem A by the hypothetical routine for solving different problem B, which runs in polynomial time.
- Basically, the polynomial reduction is a way of showing that the problem A is not harder than the problem B.

- For example, we have some hypothetical algorithm, which can sort numerical data in some polynomial time. The input to the algorithm can only be in numeric form. Suppose we have a new problem to sort names of the cities across the country. What can be done?
- Suppose we don't have an efficient algorithm to handle string data. We can apply some hashing function on city names to map them to numeric values. Now, this is identical to the first approach.
- Thus, the polynomial reduction is the way of turning one problem into another problem whose solution can be found in polynomial time.
- Reduction takes one of the three forms :
  1. Restriction
  2. Local Replacement
  3. Component design
- Consider two decision problems A and B. Reduction from A to B transforms input x of A to equivalent input f(x) of B by applying some transformation function on x.
- So given an input x to A, reduction algorithm produces intermediate result f(x) to problem B such that f(x) to B returns "yes" only if input x to A returns "Yes". Fig. 8.3.1 shows the scenario.

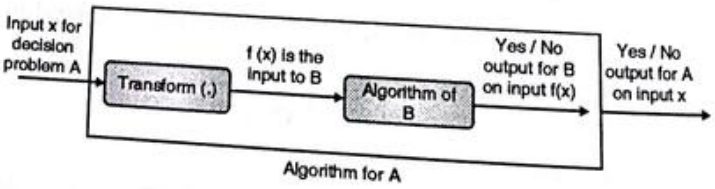


Fig. 8.3.1 : Reduction

- Thus, we say A is polynomial time reducible to B if there exists some transformation function f(.) such that,

- Transformation function f(.) maps input x of problem A to f(x) such that, input f(x) to B produce the same answer as x would have produced for A.
- f(x) should be computable in polynomial time of x. If such function exists, we say A is polynomial time reducible to B, denoted as $A \leq_p B$.

– $A \leq_p B$ implies if $B \in P$ then $A \in P$. However this does not imply if $A \in P$ then $B \in P$.

– Fig. 8.3.2 shows the mode of reduction from one problem to other.

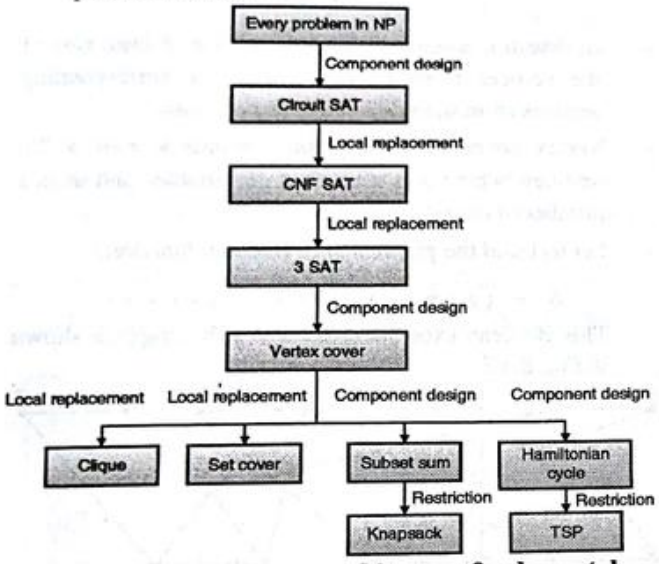


Fig. 8.3.2 : Reductions used in some fundamental NP-completeness proof

Syllabus Topic : NP-Completeness Proofs

8.4 NP-Completeness Proofs

Q. What do you mean by NP-Complete Problems? Give an example. **(2 Marks)**

Q. What are the conditions to prove that a problem P is NP-Complete? **(8 Marks)**

– If $B \leq_p A$, implies B is reducible to A and B is not harder than A by some polynomial factor.

Definition

Decision problem C is called **NP-complete** if it has following two properties :

1. C is in NP, and
2. Every problem X in NP is reducible to C in polynomial time, i.e. For every $X \in NP$, $X \leq_p C$.

– These two facts prove that NP-complete problems are the harder problems in class NP. They are often referred as **NPC**.

– Problem satisfying condition 2 is said to be **NP-hard**, whether or not it satisfies condition 1.

– If any NP-complete problem belongs to class P, then P = NP. However, a solution of any NP-complete problem can be verified in polynomial time, it cannot be obtained in polynomial time.

– Method for solving NP-complete problems in reasonable time remains undiscovered. NP-complete problems are often solved using randomization algorithms, heuristic approach or approximation algorithms.

☞ **Examples of NP-Complete problems**

(1) Boolean satisfiability problem.
(2) Knapsack problem.
(3) Hamiltonian path problem.
(4) Traveling salesman problem.
(5) Subset sum problem.
(6) Vertex cover problem.
(7) Graph colouring problem.
(8) Clique problem.

8.4.1 Vertex Cover Problem

Q. Specify one example of the NP-complete problem. Also, justify that why it is NP-complete. **(10 Marks)**

Q. Prove that vertex cover problem is Np complete. **(7 Marks)**

Definition

– **Vertex cover** of Graph G = (V, E) is set of vertices such that any edge (u, v) ∈ E, incident to *at least one* vertex in the cover. In other words, vertex cover is a subset of vertices $V' \subseteq V$ such that if the edge (u, v) ∈ E then $u \in V'$ *or* $v \in V'$.

– The size of the cover is a number of vertices in V'. **Vertex cover problem** is to find out such minimum size cover. A decision problem is to check if given graph has vertex cover of size k.

– The simplest way of finding vertex cover of graph G = (V, E) is to randomly select the edge (u, v) ∈ E and delete adjacent edges of u and v. Repeat the procedure until the cover is found. For example, consider Fig. 8.4.1.

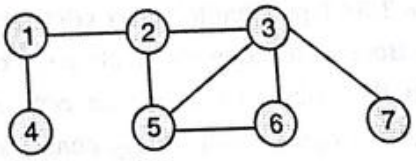


Fig. 8.4.1

– Let S represents the solution set. Initially, S = { }

Step 1 : Select any random edge, let us select edge <1, 2> as shown in the Fig. 8.4.2. Remove the incident edges of vertex 1 and 2. So, S = {1, 2}

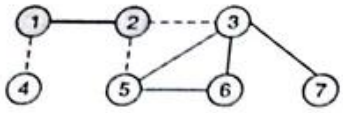


Fig. 8.4.2 : After selecting edge <1, 2>

Still few edges are not adjacent to vertices in S, so go on.

Step 2 : Let us now select edge <5, 6>. Remove adjacent edges to vertex 5 and 6.

So, S = {1, 2, 5, 6}

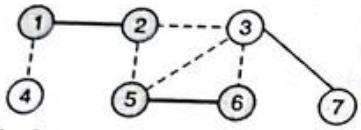


Fig. 8.4.3 : After selecting edge <5, 6>

- Still few edges are not adjacent to vertices in S, so go on.

Step 3 : Let us now select the edge <3, 7>.

So S = {1, 2, 3, 5, 6} and all the edges are adjacent to at least one vertex in S. So S is the cover of graph G.

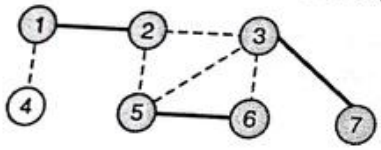


Fig. 8.4.4 : After selecting edge <3, 6>

- However, S is the cover of the graph but it may not be minimum. Instead of selecting edge <5, 6> in step 2, if we would have selected edge <3, 6>, it would have resulted in a minimum number of vertices.

Theorem : Vertex cover is NP-complete

Proof

- To prove that vertex cover is NP-complete, we will reduce 3-SAT problem to vertex cover problem. Let ϕ be the Boolean function with k clauses. For each literal 'an' in the clause, we create an edge as shown in Fig. 8.4.5. Edge is truth setting component, a vertex cover must include at least one of a or $\bar{a}$.

Fig. 8.4.5

- Addition to this, we add following : for each clause $C_i = (a + b + c)$ form a triangle with vertices a, b and c.

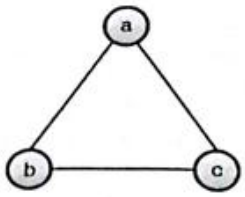


Fig. 8.4.6

- Any vertex cover will have to include at least two of the vertices from {a, b, c}. Join the corresponding vertices from triangle to edge as per clause.
- Vertex cover of such graph contains $k = n + 2m$ vertices, where n is a number of variables and m is a number of clauses.
- Let us build the graph 3-SAT Boolean function.

$$\phi = (a + b + c)(a + b + \bar{c})(\bar{a} + c + \bar{d})$$

- This Boolean expression generates the graph as shown in Fig. 8.4.7.

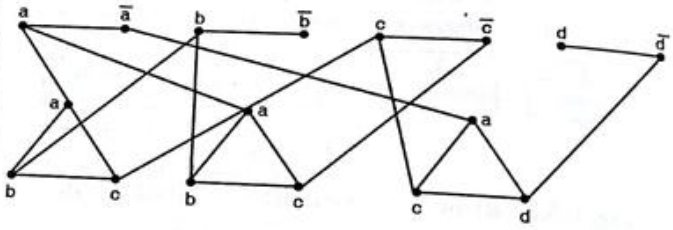


Fig. 8.4.7

- For given Boolean function,
  n = number of variables = 4
  m = number of clauses = 3
  k = n + 2m = 10
- So vertex cover of this graph must contain 10 vertices. One vertex of each triangle and one vertex from each edge.
- Vertices, which are part of vertex cover are shown in Fig. 8.4.8.
- This is how 3-SAT problem is reduced to vertex cover problem, so vertex cover is NP-complete problem.

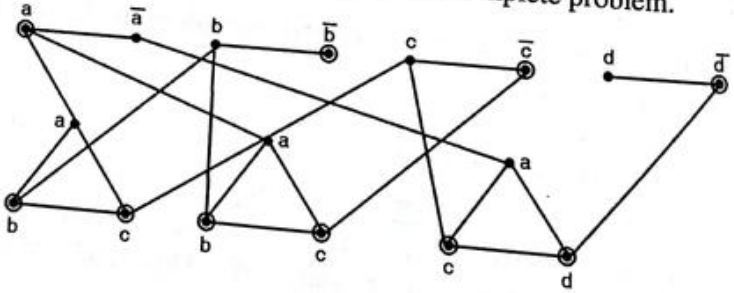


Fig. 8.4.8

Syllabus Topic : Clique Problem

8.4.2 Clique Problem

Q. Prove that Clique Decision Problem is NP-Hard. **(7 Marks)**

Q. Prove that a clique problem is NP-complete. **(7 Marks)**

☞ **Problem**

Clique is the complete subgraph of graph G. In complete subgraph, there exists an edge between every pair of vertices.

- The size of a clique is given by a number of vertices in it. Max clique is the clique of maximum size.
- Finding max clique is obviously optimization problem. Checking if graph G has a clique of size k is decision problem.
- The optimization problem is to find a clique of maximum size for given graph. A decision problem is to check whether a clique of size k exists for graph G.

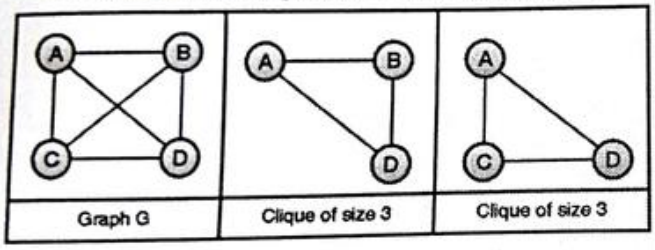
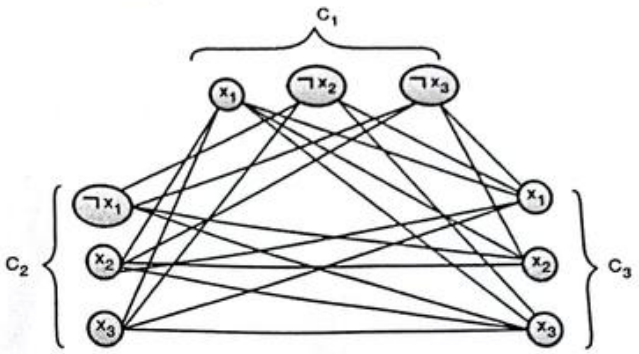


Fig. 8.4.9 : Graph G and its cliques

- Let algorithm CLIQUE(G, k) returns true if graph G has a clique of size k. We shall start with k = n, n – 1, n – 2, … until CLIQUE(G, k) returns true.

☞ **Theorem : Clique Decision Problem is NP-complete**

- To prove that clique belongs to NP-complete, we use V' as a certificate for graph G. Checking if V' is a clique, can be done in polynomial time by checking the presence of edge for each u, $v \in V'$.
- To show that clique is NP-complete, we will show that 2-CNF-SAT $\leq_p$ CLIQUE.
- Let $\phi = C_1 \wedge C_2 \wedge C_3 \wedge \ldots \wedge C_k$ be a Boolean function of k clause, where each clause C_i is in 3-CNF, i.e. each clause has exactly three literals. We shall construct a graph such that Boolean function ϕ be satisfiable if and only if G has a clique of size k. The graph can be constructed as follows:
- Each vertex corresponds to a literal.
- Connect each vertex to remaining all vertices in remaining clause except for $\bar{x}$ and $\neg x$
- **Example :** $\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$
- We shall show that the transformation ϕ to G is polynomial reduction. Let us consider that ϕ has satisfying assignment. All literals x_i in each clause are ORed with each other.
- So at least one literal in each clause is assigned value 1. If we pick up one such literal from each clause, it forms a set V' of k vertices as we have k clauses in ϕ. There exist an edge for each u, $v \in V'$.



- CNF satisfiability is NP-complete and it is reducible to clique, so clique is also NP-complete.

8.5 Exam Pack (Review Questions)

☞ **Syllabus Topic : Polynomial Time**

Q. What do you mean by P problems? Give an example. *(Ans. : Refer section 8.1)* (5 Marks)

Q. Write a short note on P problems.

(Ans. : Refer section 8.1) (5 Marks)

☞ **Syllabus Topic : Polynomial time verification**

Q. Write short note on Hamiltonian cycle.

(Ans. : Refer section 8.2) (5 Marks)

☞ **Syllabus Topic : NP-Completeness and Reducibility**

Q. What is Reduction in NP-completeness proofs? What are types of reductions?

(Ans. : Refer section 8.3) (7 Marks)

Q. Explain Polynomial Time Algorithm.

(Ans. : Refer section 8.3) (5 Marks)

☞ **Syllabus Topic : NP-Completeness Proofs**

Q. What do you mean by NP-Complete Problems? Give an example.

(Ans. : Refer section 8.4) (2 Marks)

Q. What are the conditions to prove that a problem P is NP-Complete? *(Ans. : Refer section 8.4)* (8 Marks)

Q. Specify one example of the NP-complete problem. Also, justify that why it is NP-complete.

(Ans. : Refer section 8.4.1) (10 Marks)

Q. Prove that vertex cover problem is Np complete. *(Ans. : Refer section 8.4.1)* (7 Marks)

☞ **Syllabus Topic : Clique Problem**

Q. Prove that Clique Decision Problem is NP-Hard. *(Ans. : Refer section 8.4.2)* (7 Marks)

Q. Prove that a clique problem is NP-complete.

(Ans. : Refer section 8.4.2) (7 Marks)

□□□