End Sem Exam 70 Marks (Unit III, IV, V, VI)

▶ Unit III : Greedy and Dynamic Programming Algorithmic Strate — 08 Hrs.

Greedy strategy : Principle, control abstraction, time analysis of control abstraction, knapsack problem, scheduling algorithms-Job scheduling and activity selection problem.

Dynamic Programming : Principle, control abstraction, time analysis of control abstraction, binomial coefficients, OBST, 0/1 knapsack, Chain Matrix multiplication. **(Refer chapter 3)**

▶ Unit IV : Backtracking and Branch-n-Bound — 08 Hrs.

Backtracking : Principle, control abstraction, time analysis of control abstraction, 8-queen problem, graph coloring problem, sum of subsets problem.

Branch-n-Bound : Principle, control abstraction, time analysis of control abstraction, strategies- FIFO, LIFO and LC approaches, TSP, knapsack problem. **(Refer chapters 4, 5 and 6)**

▶ Unit V : Amortized Analysis — 07 Hrs.

Amortized Analysis : Aggregate Analysis, Accounting Method, Potential Function method, Amortized analysis-binary counter, stack Time-Space tradeoff, Introduction to Tractable and Non- tractable Problems, Introduction to Randomized and Approximate algorithms, Embedded Algorithms: Embedded system scheduling (power optimized scheduling algorithm), sorting algorithm for embedded systems. **(Refer chapter 7)**

▶ Unit VI : Multithreaded And Distributed Algorithms — 07 Hrs.

Multithreaded Algorithms - Introduction, Performance measures, Analyzing multithreaded algorithms, Parallel loops, Race conditions.

Problem Solving using Multithreaded Algorithms - Multithreaded matrix multiplication, Multithreaded merge sort.

Distributed Algorithms - Introduction, Distributed breadth first search, Distributed Minimum Spanning Tree.

String Matching- Introduction, The Naive string matching algorithm, The Rabin-Karp algorithm.

(Refer chapter 8)

❑❑❑

INDEX

In Sem

UNIT I : ALGORITHMS AND PROBLEM SOLVING

- **Chapter 1 :** Algorithms and Problem Solving — 1-1 to 1-14

UNIT II : ANALYSIS OF ALGORITHMS AND COMPLEXITY THEORY

- **Chapter 2 :** Analysis of Algorithms and Complexity Theory — 2-1 to 2-29

End Sem

UNIT III : GREEDY AND DYNAMIC PROGRAMMING ALGORITHMIC STRATE

- **Chapter 3 :** Greedy Strategy — 3-1 to 3-13

UNIT IV : BACKTRACKING AND BRANCH-N-BOUND

- **Chapter 4 :** Dynamic Programming — 4-1 to 4-31
- **Chapter 5 :** Backtracking — 5-1 to 5-21
- **Chapter 6 :** Branch and Bound — 6-1 to 6-44

UNIT V : AMORTIZED ANALYSIS

- **Chapter 7 :** Amortized Analysis — 7-1 to 7-16

UNIT VI : MULTITHREADED AND DISTRIBUTED ALGORITHMS

- **Chapter 8 :** Multithreaded and Distributed Algorithms — 8-1 to 8-22

❑❑❑

UNIT III

CHAPTER 3

Greedy Strategy

Syllabus

Greedy Strategy : Principle, Control abstraction, time analysis of control abstraction, Knapsack problem, Scheduling algorithms-Job Scheduling and Activity selection problem.

3.1 Principle and General Method 3-2

3.1.1 Characteristic Components of Greedy Algorithm 3-2

3.1.2 Versions of Greedy Method 3-3

UQ. Write control abstraction for greedy strategy. SPPU - Q. 4(b), Aug. 17, 4 Marks. 3-3

3.2 Knapsack Problem 3-3

Solved University Examples

UEx. 3.2.1 (SPPU - Q. 2(a), Dec. 19, 6 Marks) 3-4

UEx. 3.2.2 (SPPU - Q. 1(b), May 18, Q. 3(a), Aug. 17, 6 Marks) 3-5

UEx. 3.2.3 (SPPU - Q. 3(a), March 19, 5 Marks) 3-5

UEx. 3.2.4 (SPPU - Q. 1(c), May 16, 8 Marks) 3-5

3.3 Job Scheduling Problem 3-7

Solved University Examples

UEx. 3.3.3 (SPPU - Q. 5(a), May 19, 5 Marks) 3-9

UEx. 3.3.4 (SPPU - Q. 1(a), Dec. 17, 6 Marks) 3-10

3.4 Activity Selection Problem 3-10

Chapter Ends **3-13**

- In cricket, when you bat you concentrate on the current ball and you hit it by playing the best possible shot at that moment. The strategy for any shot depends on the current ball and run time condition of the match. Without thinking for the overall inning and future strategy of batting, you play the best possible game at that moment.
- Once you hit the ball, the action is irreversible. You cannot undo your played shot in any circumstances in future. For every shot, you are greedy to grab more runs by hitting that ball. The strategy used here is the greedy strategy.
- It is suitable for solving the optimization problems where a multi-step progressive solution is constructed by applying some locally optimal criteria. In optimization problems, the desired extremum (minimum or maximum value) is achieved by satisfying specified constraints.

3.1 PRINCIPLE AND GENERAL METHOD

GQ. Describe the greedy method. **(4 Marks)**

- The greedy method builds a solution in stages. At every stage, it selects the best choice concerning the local considerations.
- A locally optimal partial solution cannot be altered later. Such locally optimal partial solutions generated at each stage, finally build a global solution.
- Thus, the greedy approach provides a step-by-step progressive solution by expanding a partial solution at every stage until the global solution is built. The global solution is not always optimal. However, many times it proves to be an optimal solution.
- The greedy approach is simpler and quite powerful to solve a wide variety of optimization problems.
- **Some of the classic problems** that can be solved effectively by the greedy method are listed below:
  (1) Knapsack problem
  (2) Job sequencing problem
  (3) Activity selection problem.
  (4) Minimum spanning tree problem
  (5) Tree vertex splitting problem
  (6) Optimal storage on tapes problem
  (7) Optimal merge patterns problem
  (8) Single source shortest path problem

3.1.1 Characteristic Components of Greedy Algorithm

GQ. What is the basic nature of the greedy strategy? **(3 Marks)**

GQ. Define Feasible Solution. **(1 Mark)**

GQ. Explain the general characteristics of greedy algorithms. **(4 Marks)**

GQ Discuss general characteristics of the greedy method. Mention any two examples of the greedy method that we are using in real life. **(6 Marks)**

GQ. Briefly describe the greedy choice property and optimal substructure. **(4 Marks)**

GQ. Explain the following terms with reference to the Greedy Technique.
(i) Feasible solution and Optimal solution
(ii) Subset paradigm and Ordering paradigm. **(4 Marks)**

Different basic components characterize the greedy algorithms.

- **The feasible solution :** A subset of given inputs that satisfies all specified constraints of a problem is known as a "feasible solution".
- **Optimal solution :** The feasible solution that achieves the desired extremum is called an **"optimal solution"**. In other words, the feasible solution that either minimizes or maximizes the objective function specified in a problem is known as an **"optimal solution"**.
- **Feasibility check :** It investigates whether the selected input fulfils all constraints mentioned in a problem or not. If it fulfils all the constraints then it is added to a set of feasible solutions; otherwise, it is rejected.
- **Optimality check :** It investigates whether a selected input produces either a minimum or maximum value of the objective function by fulfilling all the specified constraints. If an element in a feasible solution set produces the desired extremum, then it is added to a set of optimal solutions.
- **Optimal substructure property :** The globally optimal solution to a problem includes the optimal sub solutions within it.

- **Greedy choice property :** The globally optimal solution is assembled by selecting locally optimal choices. The greedy approach applies some locally optimal criteria to obtain a partial solution that seems to be the best at that moment and then find out the solution for the remaining sub-problem.
- The local decisions (or choices) must possess three characteristics as mentioned below :
  - (i) **Feasibility :** The selected choice must fulfil local constraints.
  - (ii) **Optimality :** The selected choice must be the best at that stage (locally optimal choice).
  - (iii) **Irrevocability :** The selected choice cannot be changed once it is made.

3.1.2 Versions of Greedy Method

GQ. Write an abstract algorithm for the greedy design method.

(i) **Subset paradigm :** The greedy approach that determines a subset of inputs that gives the optimal solution is known as a subset paradigm.

E.g., knapsack problem, job sequencing with deadlines, minimum spanning tree, tree vertex splitting, etc.

(ii) **Ordering paradigm :** The greedy approach that determines some ordering of inputs that gives an optimal solution is called an ordering paradigm.

E.g., optimal storage on tapes, optimal merge patterns, single-source shortest paths.

Control Abstraction for the Subset Paradigm

GQ. Write and explain an algorithm for greedy design method. **(6 Marks)**

UQ. Write control abstraction for greedy strategy.

SPPU - Q. 4(b), Aug. 17, 4 Marks

```
Algorithm Greedy_subset (P, n)
/* Input: P [1 : n] is a set of n inputs of a given problem instance.
Output: Optimal solution set S to a given problem instance. */
{
    S : = Φ ;              // Initialise the solution set
    for (i : = 1, i ≤ n ; i + +)
    {
        q : = Select (P) ;      /* Function Select() selects an input from P[ ] and removes it.*/
        if (Feasible (S, q) )   /* Function Feasible () returns TRUE if the selected input can be added in a solution vector, otherwise returns FALSE.*/
            S : = Union (S, q) ;   /* Function Union() combines the selected input with the solution and updates the objective function. */
    }
    return S ;          /* Returns the final solution.*/
}
```

3.2 KNAPSACK PROBLEM

GQ. Explain the fractional knapsack problem with an example. **(7 Marks)**

GQ. Prove that the fractional knapsack problem has the greedy-choice property. **(4 Marks)**

- The fractional knapsack problem follows the subset paradigm of the greedy approach. It is also known as a **"continuous knapsack problem"**.
- This variant of the knapsack problem allows keeping the fractional part of any item into a knapsack. [It is not allowed in the 0/1 knapsack problem.]

Problem description

- Consider a knapsack or bag with capacity M. Suppose, n items are given to fill a knapsack. Each item i, $1 \le i \le n$ has a weight w_i and it gives a profit of $p_i \cdot x_i$ if its fraction x_i, $0 \le x_i \le 1$ is kept into the knapsack.
- The fractional knapsack problem is the problem to earn maximum profit by filling the knapsack with items considering the constraint of knapsack capacity.
- The problem can be mathematically given as :

$$\text{Maximize} \sum_{i=1}^{n} p_i x_i$$

$$\text{Subject to} \sum_{i=1}^{n} w_i x_i \le M$$

and $0 \le x_i \le 1$, $1 \le i \le n$. All the values of profit and weights are positive real numbers.

The general procedure

- To get the maximum profit by filling a knapsack with objects, the greedy approach tries to select the objects with more profit value.
- However, those objects should be of lighter weights, so that more objects can be kept into the knapsack considering its capacity.
- This is achieved by checking the ratio of profit to the weight of each object. The greedy algorithm arranges all the items in descending order of the ratios of their profits to weights.
- The items are kept in a knapsack as per this order. If no sufficient space is available in a knapsack to add a whole item as per this order, then its fractional part is added into the knapsack to make it full.

The basic steps followed by the greedy algorithm to solve the knapsack problem are given as :

(1) Sort all n items in descending order of their profit to weight ratios.

(2) Select each item as per the sorted list and check for the available knapsack capacity. If sufficient space is available in the knapsack then keep the whole item into the knapsack.

(3) If the space in the knapsack is insufficient to keep a whole item in it then the fractional part of that item equal to the remaining capacity of the knapsack is placed into it. Thus only one item that is added at the end is in fractional part and the rest added items are the whole items.

(4) When the knapsack is full, the algorithm terminates.

Greedy algorithm

GQ. Write an algorithm for Knapsack problem using Greedy Strategy and find out its time complexity. **(4 Marks)**

```
Algorithm Fr-Knapsack(M, n)
/* Input: Knapsack capacity = M. n is the number of available
items with associated profits P[1 : n] and weights W[1 : n].
These items are arranged such that P[i] / W[i] ≥ P[i + 1] / W[i
+ 1] where 1 ≤ i ≤ n.
Output: S[1 : n] is the fixed size solution vector. S[i] gives the
fractional part x_i of item i placed into a knapsack, 0 ≤ x_i ≤ 1
and 1 ≤ i ≤ n. */
{
   for (i := 1 ; i ≤ n ; i ++)
   {
      S[i] := 0.0 ; /* Initialization of a solution vector. */
   }
   balance := M ;          /* 'balance' describes the
                              remaining capacity of
                              knapsack. Initially it is equal to
                              the knapsack capacity M. */
   for (i := 1 ; i ≤ n ; i ++)
   {
      if (W[i] > balance)      /*Insufficient capacity of
                                 a knapsack */
         break ;
      S[i] := 1.0 ;            // add whole item i.
      balance := balance - W[i] ; /*Update the remaining
                                    capacity after adding item
                                    i with weight W[i] into a
                                    knapsack.*/
   }
   if (i ≤ n)                  /*Due to insufficient capacity
                                 add fractional part of item i in
                                 the knapsack. */
      S[i] := (balance / W[i]) ;
   return S ;
}
```

Complexity analysis

- Any efficient sorting algorithm takes O(n log n) time to sort n items in descending order of their profit to weight ratios.
- If we assume that the given n items are already arranged in decreasing order of P_i/W_i, $1 \le i \le n$, then by ignoring the complexity O(n log n) of sorting, we get the time complexity, O(n) of the greedy algorithm for fractional knapsack.

UEx. 3.2.1 SPPU - Q. 2(a), Dec. 19, 6 Marks

Consider the following instances of knapsack problem n = 3, M = 50, (P1, P2, P3): (60, 100, 120), and (W1, W2, W3) : (10, 20, 30). Find the optimal solution using Greedy approach ?

Soln. :

- Let I1, I2, and I3 be the 3 given items with profits (P1, P2, P3) = (60, 100, 120) and (W1, W2, W3) = (10, 20, 30).
- To solve this knapsack instance by the greedy algorithm we first calculate the ratios of Pi / Wi where $1 \le i \le n$.

Item (i)	Pi	Wi	Pi / Wi
1	60	10	6
2	100	20	5
3	120	30	4

- The given items are in decreasing order of Pi / Wi.
- As available knapsack capacity = 50, item I1and I2 can be placed as a whole in a knapsack by earning a profit of 60 + 100 = 160 units. Now available knapsack capacity= 50 – (10 + 20) = 20..
- Due to insufficient capacity of a knapsack, the next item I3 cannot be added as a whole into a knapsack. So the fractional part of I3 that can be placed into the knapsack = $\frac{20}{30} = \frac{2}{3}$. It gives the profit of (120*2/3) = 80 units.
- Thus the total profit earned by placing given 3 items as (I1= 1, I2 = 1, I3 = 2/3) in a knapsack = 60 + 100 + 80 = **240 units**. So fixed size solution vector **S = (1, 1, 2/3).**

UEx. 3.2.2
SPPU - Q. 1(b), May 18, Q. 3(a), Aug. 17, 6 Marks
How does Fractional greedy algorithm solve the following knapsack problem with capacity 20, P = (25, 24, 15) and W = (18, 15, 10).

☑ **Soln. :**

- Let i_1, i_2, and i_3 be the 3 given items with profits (p_1, p_2, p_3) = (25, 24, 15) and (w_1, w_2, w_3) = (18, 15, 10).
- To solve this knapsack instance by the greedy algorithm we first calculate the ratios of p_i / w_i where $1 \le i \le n$.

Item (i)	p_i	w_i	p_i / w_i
1	25	18	1.389
2	24	15	1.6
3	15	10	1.5

- We arrange the given items in decreasing order of p_i / w_i as i_2, i_3, and i_1.
- As available knapsack capacity = 20, item i_2 with weight 15 can be placed as a whole in a knapsack by earning a profit of 24 units. Now available knapsack capacity = 20 – 15 = 5.
- Due to insufficient capacity of a knapsack, the next item i_3 cannot be added as a whole into a knapsack. So the fractional part of i_3 that can be placed into the knapsack = $\frac{5}{10} = \frac{1}{2}$. It gives the profit of $(\frac{15}{2})$ = 7.5 units.
- Thus the total profit earned by placing given 3 items as $(i_1 = 0, i_2 = 1, i_3 = 1/2)$ in a knapsack = 24 + 7.5 = **31.5 units**. So fixed size solution vector **S = (0, 1, 1/2).**

UEx. 3.2.3 SPPU - Q. 3(a), March 19, 5 Marks
How does fractional greedy algorithm solve the following knapsack problem with capacity 8, Items (I1, I2, I3, I4), profit P = (25, 24, 15, 40) and weight w = (3, 2, 2, 5).

☑ **Soln. :**

- I1, I2, I3, I4 are the 3 given items with profits (p_1, p_2, p_3, p_4) = (25, 24, 15,40) and (w_1, w_2, w_3, w_4) = (3, 2, 2, 5).
- To solve this knapsack instance by the greedy algorithm we first calculate the ratios of p_i / w_i where $1 \le i \le n$.

Item (i)	p_i	w_i	p_i / w_i
1	25	3	8.33
2	24	2	12
3	15	2	7.5
4	40	5	8

- We arrange the given items in decreasing order of p_i / w_i as I2, I1, I4 and I3.
- As available knapsack capacity = 8, item I2 and I1 can be placed as a whole in a knapsack by earning a profit of 24 + 25 = 49 units. Now available knapsack capacity = 8 – (2 + 3) = 3.
- Due to insufficient capacity of a knapsack, the next item I4 cannot be added as a whole into a knapsack. So the fractional part of I4 that can be placed into the knapsack = (3/5). It gives the profit of (40*3/5) = 24 units. Item I3 cannot be added.
- Thus the total profit earned by placing 3 items as (I2 = 1, I1 = 1, I4 = 3/5) in a knapsack = 24 + 25 + 24 = **73 units**. So fixed size solution vector **S = (1, 1, 0, 3/5).**

UEx. 3.2.4 SPPU - Q. 1(c), May 16, 8 Marks
Find an optimal solution for the following knapsack instance using greedy method. Number of objects n = 5, capacity of knapsack m = 100, profits = (10, 20, 30, 40, 50), weights = (20, 30, 66, 40, 60).

☑ **Soln. :**

- Let I1, I2, I3, I4, I5 be the given items with profits $(p_1, p_2, p_3, p_4, p_5) = (10, 20, 30, 40, 50)$ and $(w_1, w_2, w_3, w_4, w_5) = (20, 30, 66, 40, 60)$.
- To solve this knapsack instance by the greedy algorithm we first calculate the ratios of p_i / w_i where $1 \le i \le n$.

Item (i)	p_i	w_i	p_i / w_i
1	10	20	0.5
2	20	30	0.67
3	30	66	0.45
4	40	40	1
5	50	60	0.83

- We arrange the given items in decreasing order of p_i / w_i as I4, I5, I2, I1 and I3.
- As available knapsack capacity = 100, item I4 and I5 can be placed as a whole in a knapsack by earning a profit of 40 + 50 = 90 units. Now available knapsack capacity = 100 – (40 + 60) = 0.
- As a knapsack is full, the remaining items I2, I1 and I3 cannot be added into a knapsack.
- Thus the total profit earned by placing the items as (I4 = 1, I5 = 1) in a knapsack = 40 + 50 = **90units**. So fixed size solution vector **S = (0, 0, 0, 1, 1).**

Ex. 3.2.5 : Consider the instance of the 0/1 (binary) knapsack problem as below with P depicting the value and W depicting the weight of each item whereas M denotes the total weight carrying capacity of the knapsack. Find the optimal answer using the greedy design technique. Also, write the time complexity of the greedy approach for solving the knapsack problem.

P = [40 10 50 30 60] W = [80 10 40 20 90] M = 110,

(5 Marks)

☑ **Soln. :**

- Let i_1, i_2, i_3, i_4, i_5 be the given 6 items with profits $P(1:5) = \{40, 10, 50, 30, 60\}$ and weights $W(1:5) = \{80, 10, 40, 20, 90\}$.
- To solve this knapsack instance by the greedy method we first calculate the ratios P_i / W_i; $1 \le i \le 5$.

Item (i)	P_i	W_i	P_i / W_i
1	40	80	0.5
2	10	10	1
3	50	40	1.25
4	30	20	1.5
5	60	90	0.67

- We arrange the given items in decreasing order of P_i / W_i as i_4, i_3, i_2, i_5 and i_1.
- Considering the available capacity M = 110 of a knapsack we can add items i_4, i_3, i_2 as a whole.

  ∴ The remaining capacity of a knapsack

  = 110 – (20 + 40 + 10) = 40
- Due to insufficient capacity of a knapsack item i_5 is added as a fractional part = (4/9) by giving profit = (4/9) ×60 = 26.67 units. Item i_1 cannot be added.

  ∴ Total profit earned = 30 + 50 + 10 + 26.67

  = 116.67units

The fixed size solution vector of 5 items = **(0, 1, 1, 1, 4/9).**

Ex. 3.2.6 : Consider Knapsack capacity W = 50, w = (10, 20, 40) and v = (60, 80, 100) find the maximum profit using greedy approach. **(4 Marks)**

☑ **Soln. :**

- Let i_1, i_2, i_3 be the given 3 items with the profits $v(1:3) = \{60, 80, 100\}$ and weights $w(1:3) = \{10, 20, 40\}$.
- To solve this knapsack instance by the greedy method we first calculate the ratios v_i / w_i; $1 \le i \le 3$.

Item (i)	v_i	w_i	v_i / w_i
1	60	10	6
2	80	20	4
3	100	40	2.5

- The given items are already arranged in decreasing order of v_i / w_i as i_1, i_2, and i_3.
- Considering the available capacity W = 50 of a knapsack we can add items i_1 and i_2 as a whole.

  ∴ The remaining capacity of a knapsack

  = 50 – (10 + 20) = 20
- Due to insufficient capacity of a knapsack item i_3 is added as a fractional part = (1/2) by giving profit

  = (1/2) × 100 = 50 units.

  ∴ Total profit earned = 60 + 80 + 50 = **190 units**
- The fixed size solution vector of 3 items = **(1, 1, 1/2).**

Ex. 3.2.7 : Solve the following Knapsack Problem using greedy method. Number of items = 5, knapsack capacity W = 100, weight vector = {50, 40, 30, 20, 10} and profit vector = {1, 2, 3, 4, 5} **(5 Marks)**

☑ **Soln. :**

- Let i_1, i_2, i_3, i_4, i_5 be the given 5 items with profits $p(1:5) = \{1, 2, 3, 4, 5\}$ and weights $w(1:5) = \{50, 40, 30, 20, 10\}$.
- To solve this knapsack instance by the greedy method we first calculate the ratios p_i / w_i; $1 \le i \le n$.

Item (i)	p_i	w_i	p_i / w_i
1	1	50	0.02
2	2	40	0.05
3	3	30	0.1
4	4	20	0.2
5	5	10	0.5

- We arrange the given items in decreasing order of p_i / w_i as i_5, i_4, i_3, i_2, and i_1.
- Considering the available capacity W = 100 of a knapsack we can add items i_5, i_4, i_3, i_2 as a whole.

  ∴ The remaining capacity of a knapsack

  = 100 – (10 + 20 + 30+ 40) = 0
- Due to insufficient capacity of a knapsack item i_1 cannot be added.

  ∴ Total profit earned = 5 + 4 + 3 + 2 = **14 units**

The fixed size solution vector of 5 items = **(0, 1, 1, 1, 1).**

Ex. 3.2.8 : Find an optimal solution for the following knapsack instance using greedy method: Number of objects n = 5, capacity of knapsack m = 100,
profits = (10, 20, 30, 40, 50), weights = (20, 30, 66, 40, 60).

(5 Marks)

☑ **Soln. :**

- Let I1, I2, I3, I4, I5 be the given items with profits $(p_1, p_2, p_3, p_4, p_5) = (10, 20, 30, 40, 50)$ and $(w_1, w_2, w_3, w_4, w_5) = (20, 30, 66, 40, 60)$.
- To solve this knapsack instance by the greedy algorithm we first calculate the ratios of p_i / w_i where $1 \le i \le n$.

Item (i)	p_i	w_i	p_i / w_i
1	10	20	0.5
2	20	30	0.67
3	30	66	0.45
4	40	40	1
5	50	60	0.83

- We arrange the given items in decreasing order of p_i / w_i as I4, I5, I2, I1 and I3.
- As available knapsack capacity = 100, item I4 and I5 can be placed as a whole in a knapsack by earning a profit of 40 + 50 = 90 units. Now available knapsack capacity = 100 – (40 + 60) = 0.
- As a knapsack is full, the remaining items I2, I1 and I3 cannot be added into a knapsack.
- Thus the total profit earned by placing the items as (I4 = 1, I5 = 1) in a knapsack = 40 + 50 = **90units**. So fixed size solution vector **S = (0, 0, 0, 1, 1).**

⏭ 3.3 JOB SCHEDULING PROBLEM

GQ. Write short note on Job Sequencing with Deadlines. **(4 Marks)**

GQ. Describe in brief Job Scheduling problem. **(4 Marks)**

- It is a type of scheduling problem that follows the subset paradigm of the greedy approach.
- It involves the scheduling of jobs such that they get completed before their deadlines giving the profit associated with their on-time completion.
- It has an objective to maximize the total profit gained by the completion of jobs.

☞ Problem description

- There is a set J of n jobs to be scheduled. Each job j, $1 \le j \le n$ gives profit $P_j > 0$ if it is completed by its associated deadline $D_j > 0$.
- It is assumed that there is only one machine on which the job j is to be scheduled. It is also assumed that the job j has to be processed for one unit of time on that machine for its completion.
- The problem of job sequencing with deadlines is to earn the maximum profit by completing a subset J′ of J (set of given jobs) by its deadline. It can be given mathematically as below.

  J: a set of n jobs to be scheduled, $n > 0$,

  J′: a feasible solution $\rightarrow J' \subseteq J$ such that all jobs included in J′ are completed by their deadlines,

  $\mathbf{P_j > 0}$ is a profit associated with a job j, $1 \le j \le n$ for its on-time completion,

  $\mathbf{D_j > 0}$ is a deadline associated with a job j, $1 \le j \le n$ for its completion, then the objective function is to maximize $\sum_{j \in J'} P_j$.

The general procedure

(1) With the objective of profit maximization, the greedy algorithm tries to schedule a job with the maximum profit on the highest priority.

- To achieve this, arrange all n jobs, $n > 0$ in descending order of their associated profits.
- Let J be the set of jobs arranged in decreasing order of their profits. P is the set of profits associated with each job in J so that $P_1 \geq P_2 \geq P_3 \ldots \geq P_n$. D is the set of deadlines associated with each job in J. D_j is the associated deadline of a j^{th} object in J.

(2) Assign a processing time slot [t – 1, t] if it is free to each job $j \in J$ where $1 \leq j \leq n$ and t is the largest integer m such that $1 \leq m \leq D_j$. If such a time slot is assigned to a job $j \in J$ then its associated profit P_j is added to the profit earned and that job j is included in the solution set $J' \subseteq J$.

(3) If there is no feasible time slot available to process a job by its deadline then reject that job. No profit is then earned.

(4) Repeat steps (2) and (3) to schedule all jobs in J without violating the constraint of their deadlines. The final solution vector $J' \subseteq J$ and the total profit earned associated with J′ are returned.

Greedy algorithm

```
Algorithm Greedy_Jobseq (D, P, J, n, J', T)
/* Input: n > 0 is the number of jobs to be scheduled. J[1: n]
is the set of n jobs arranged in descending order of their
profits. P is the set of profits associated with each job in J such
that P[1] ≥ P[2] ≥ .... ≥ P[n] > 0. D is the array of deadlines
associated with each job in J. D[ j] > 0 where 1 ≤ j ≤ n.
Output: J' is the dynamic array that contains a subset of given
jobs that can be finished by their deadlines. It gives a
variable-size solution set. T is the dynamic array of time slots
assigned to each job k in solution vector J' where 1 ≤ k ≤ n.
T[t] gives the job number k that is scheduled in slot t = [t – 1,
t] where 1 ≤ t ≤ maximum (D[1: n]). The algorithm returns the
maximum profit earned by the completion of jobs in J'. */
{
  D_max := maximum (D [1 : n]) ;
                        /* Find out maximum deadline
                           among deadlines of n jobs. */
  max_profit := 0 ; k := 1 ; //Initialization.
  for (t := 1 ; t ≤ D_max ; t ++)
  {
    T[t] := null ;          //Initialization.
  }
  for (j := 1 ; j ≤ n ; j ++)
  {
    t := D[j] ;     /* Get the deadline of j^th job.*/
    repeat
    {
      if (T[t] = null)        // time slot [t – 1, t is empty]
      {
        T[t] := J[j] ;   /*Schedule j^th job to t^th time slot
                           where t is the largest number m
                           such that 1 ≤ m ≤ D[j].*/
        J'[k] := J[j] ;  /*Create a feasible solution set
                           J' ⊆ J. It gives a variable size
                           solution set.*/
        k + + ;
        max_profit := max_profit + P[ j ] ;
                  /*Update maximum profit earned by
                    scheduling job j in t^th time slot. */
      }
      else
        t – – ;          /* Decrement t to search for
                    feasible t^th slot for job j where t is the
                    largest number m such that 1 ≤ m ≤
                    D[j] */
    } until (t = 0) ;
  }      // end for
  return max_profit ;
}
```

Complexity analysis

- It is assumed that each job requires processing for one unit of time for its completion.
- There are n such jobs to be scheduled. Hence, the upper bound on time slots can be necessarily considered to be n.
- Thus, for scheduling of each job maximum n time slots are to be scanned. Searching for feasible time slots to schedule at most n jobs needs $O(n^2)$ time.
- In addition to it, the algorithm takes O(n log n) time to sort n jobs according to the descending order of their profits by using any efficient sorting algorithm like heap sort or merge sort.

- Thus, the total time complexity of the greedy algorithm to solve a job sequencing problem with n jobs is given as $O(n^2)$. This complexity can be reduced to O(n log n) by efficiently using the set representation that includes the algorithms for disjoint set union and finds.

Ex. 3.3.1 ; Using a greedy algorithm find an optimal schedule for following jobs with n = 6.

Profits: $(P_1, P_2, P_3, P_4, P_5, P_6) = (20, 15, 10, 7, 5, 3)$

Deadline: $(d_1, d_2, d_3, d_4, d_5, d_6) = (3, 1, 1, 3, 1, 3)$ **(6 Marks)**

☑ **Soln. :**

- The given set P(1:6) = (20, 15, 10, 7, 5, 3) is arranged in decreasing order of profits associated with 5 jobs, say J(1 : 6). We have an associated set of deadlines d(1:6) = (3, 1, 1, 3, 1, 3).
- A feasible solute $J' \subseteq J$ is obtained by scheduling each job $j \in J$, $1 \le j \le 6$ in time slots $[t-1, t]$ where t is the largest integer m such that $1 \le m \le d[j]$. Thus, we get,

J'	Assigned time slots	Job under consideration	Action	Total profit earned
Φ	none	1	assign to [2, 3]	0
{1}	[2,3]	2	assign to [0, 1]	20
{1, 2}	[0, 1], [2, 3]	3	Not feasible, so reject	20 + 15 = 35
{1, 2}	[0, 1], [2, 3]	4	assign to [1, 2]	35
{1, 2, 4}	[0, 1], [1, 2], [2,3]	5	Not feasible, so reject	35 + 7 = 42
{1, 2, 4}	[0, 1], [1, 2], [2,3]	6	Not feasible, so reject	42

∴ The optimal solution J' = {1, 2, 4} with maximum profit = 42 units.

UEx. 3.3.2 SPPU - Q. 1(a), Dec. 19, 6 Marks

Find the correct sequence for jobs that maximizes profit using following instances, Job ID (1, 2, 3, 4, 5), Deadline (2, 1, 2, 1, 3) and Profit (100, 19, 27, 25, 15).

☑ **Soln. :**

- We arrange given 5 jobs as per decreasing order of their profits. Let J be this set of 5 jobs as below:

  J(1 : 5) = {1, 3, 4, 2, 5}.

Accordingly the sets P and D are rearranged as below:

P(1 : 4) = (100, 27, 25, 19, 15)

d(1 : 4) = (2, 2, 1, 1, 3)

- A feasible solute $J' \subseteq J$ is obtained by scheduling each job $j \in J$, $1 \le j \le 5$ in time slots $[t-1, t]$ where t is the largest integer m such that $1 \le m \le d[j]$. Thus we get :

J'	Assigned time slots	Job under consideration	Action	Total profit earned
Φ	none	J[1]=1	assign to [1, 2]	0
{1}	[1, 2]	J[2]=3	assign to [0, 1]	100
{1, 3}	[0, 1], [1, 2]	J[3]=4	Not feasible, so reject	100 + 27 = 127
{1, 3}	[0, 1], [1, 2]	J[4]=2	Not feasible, so reject	127
{1, 3}	[0, 1], [1, 2]	J[4]=5	assign to [2, 3]	127
{1, 3, 5}	[0, 1], [1, 2], [2, 3]	--	--	127+15=142

∴ The optimal solution J' = {1, 3, 5} with maximum profit = 142 units.

UEx. 3.3.3 SPPU - Q. 5(a), May 19, 5 Marks

Find the correct sequence for jobs using following instances,

Job	J1	J2	J3	J4	J5
Profit	20	15	10	5	1
Deadline	2	2	1	3	3

☑ **Soln. :**

- The given set P is arranged in decreasing order of profits associated with 5 jobs say J(1 : 5).
- A solution set $J' \subseteq J$ is obtained by scheduling each job $j \in J$, $1 \le j \le 5$ in time slots $[t-1, t]$ where t is the largest integer m such that $1 \le m \le D[j]$. Thus we get,

J'	Assigned time slots	Job under consideration	Action	Total profit earned
Φ	none	1	assign to [1, 2]	0
{1}	[1, 2]	2	assign to [0, 1]	20
{1, 2}	[0, 1], [1, 2]	3	Not feasible, so reject	20 + 15 = 35
{1, 2}	[0, 1], [1, 2]	4	assign to [2, 3]	35
{1, 2, 4}	[0, 1], [1, 2], [2,3]	5	Not feasible, so reject	35 + 5 = 40

∴ The optimal solution J′ = {1,2,4} with maximum profit = 40 units.

UEx. 3.3.4 SPPU - Q. 1(a), Dec. 17, 6 Marks

Find an optimal solution for the following instance using job sequencing with scheduling: Number of jobs n = 4, profits = (100, 27, 15, 10), deadlines = (2, 1, 2, 1)

☑ **Soln. :**

- The given set P is arranged in decreasing order of profits associated with 4 jobs say J(1 : 4).
- A solution set J′⊆ J is obtained by scheduling each job j∈J, 1 ≤ j ≤ 4 in time slots [t − 1, t] where t is the largest integer m such that 1 ≤ m ≤ D[j]. Thus we get,

J′	Assigned time slots	Job under consideration	Action	Total profit earned
Φ	none	1	assign to [1, 2]	0
{1}	[1, 2]	2	assign to [0, 1]	100
{1, 2}	[0, 1] , [1, 2]	3	Not feasible, so reject	100 + 27 = 127
{1, 2}	[0, 1], [1, 2]	4	Not feasible, so reject	100 + 27 = 127

∴ The optimal solution J′ = {1, 2} with maximum profit = 127 units.

UEx. 3.3.5 SPPU - Q. 1(c), Dec. 16, 8 Marks

Solve following job sequencing problem using Greedy approach. N = 7, Profit (P1 .... P7) = (3, 5, 20, 18, 1, 6, 30) dead: ne (d1 ..., d7) = (1, 3, 4, 2, 3, 2, 1)

☑ **Soln. :**

- We arrange given 7 jobs as per decreasing order of their profits. Let J be this set of 7 jobs as below:

  $J(1:7) = \{7, 3, 4, 6, 2, 1, 5\}$.

  Accordingly the sets P and D are rearranged as below :

  $P(1:7) = (30, 20, 18, 6, 5, 3, 1)$

  $d(1:7) = (1, 4, 2, 2, 3, 1, 3)$

- A feasible solute J′⊆ J is obtained by scheduling each job j∈J, 1 ≤ j ≤ 7 in time slots [t − 1, t] where t is the largest integer m such that 1 ≤ m ≤ d[j].
- Thus we get :

J′	Assigned time slots	Job under consideration	Action	Total profit earned
Φ	none	J[1] = 7	assign to [0, 1]	0
{7}	[0, 1]	J[2] = 3	assign to [3, 4]	30
{7, 3}	[0, 1], [3, 4]	J[3] = 4	assign to [1, 2]	30 + 20 = 50
{7, 3, 4}	[0, 1], [1, 2], [3, 4]	J[4] = 6	Not feasible, so reject	50 + 18 = 68
{7, 3, 4}	[0, 1], [1, 2], [3, 4]	J[5] = 2	assign to [2, 3]	68
{7, 3, 4, 2}	[0, 1], [1, 2], [2, 3], [3,4]	J[6] = 1	Not feasible, so reject	68 + 5 = 73
{7, 3, 4, 2}	[0, 1], [1, 2], [2, 3], [3,4]	J[7] = 5	Not feasible, so reject	74

∴ The optimal solution J′ = {2, 3, 4, 7} with maximum profit = 73 units.

⏭ 3.4 ACTIVITY SELECTION PROBLEM

GQ. Suppose that we have a set of activities to schedule among a large number of lecture halls, where any activity can take place in any lecture hall. We wish to schedule all the activities using as few lecture halls as possible. Give an efficient greedy algorithm to determine which activity should use which lecture hall. **(4 Marks)**

- The activity selection problem is a classic optimization problem solvable by the greedy approach. It follows the subset paradigm of greedy algorithms.
- It schedules a resource among multiple activities competing for the same with the objective of selection of a maximum number of compatible activities.
- E.g. Scheduling activities in a lecture hall, scheduling multiple production activities on a machine, booking for multiple events in the same auditorium, etc.

Problem description

- Suppose $A = \{a_1, a_2, ..., a_n\}$ is a set of n proposed activities that need a common resource. Each activity a_i, $1 \le i \le n$ has its start time s_i and a finish time f_i, where $0 \le s_i < f_i < \infty$.
- When we select any activity a_i, $1 \le i \le n$, it gets completed during the half-open time interval $[s_i, f_i)$.
- Any two activities a_i and a_j are said to be compatible if they have non-overlapping time intervals i.e. if $s_i \ge f_j$ or $s_j \ge f_i$ then a_i and a_j are known as compatible activities.
- The activity selection problem is to investigate a maximum-size subset of mutually compatible activities.
- The greedy approach selects the next activity a_i with minimum finish time f_i among rest activities and with the start time s_i more than or equal with the finish time f_{i-1} of the last chosen activity a_{i-1}.

The general procedure

1. Sort all activities in ascending order of finish time.
2. Choose the activity with the minimum finish time.
3. Eliminate the non-compatible activities that could not be scheduled.
4. Repeat steps (2) and (3) until decisions on selections of all activities are made.

Greedy Algorithm

```
Algorithm Activity_Select (A, S, F, n)
/* It describes the greedy algorithm to solve the activity
selection problem.
Input: A[1: n] is a list of proposed activities arranged in
ascending order of their finish time and n is the number of
proposed activities. S[1: n] and F[1: n] are lists giving the
associated start and finish time of activities in A[1: n]. F[i] ≤
F[i+1], 1 ≤ i ≤ n.
Output : An optimal solution set A' that gives a maximum-
size subset of compatible activities from A[1: n]. */
{
  A' := A[1] ;          /*Select the first activity with
                          a minimum finish time.*/
  for (j := 2, i ≤ n ; j ++)
  {
    if (S[j] ≥ F[j])  /* Choose the activity if its start time
                         is more than or equal to the finish
                         time of the last chosen activity.*/
    {
      A' := A' ∪ A[j] ;   /* Add next selected activity
                             in a final solution set.*/
      i := j;
    }
  }
  return A'; /* Return the final optimal solution set */
}
```

Complexity Analysis

Any efficient sorting algorithm takes O(n log n) time to sort n proposed activities in ascending order of their finish time. So, the time complexity of the activity selection greedy algorithm will be bounded by O(n log n).

- If a given set of n proposed activities is already sorted in ascending order of their finish time, then the time complexity of the algorithm will be O(n).

Ex. 3.4.1 : Write greedy algorithm for activity selection problem. Give its time complexity. For the following intervals, select the activities according to your algorithm. I1 (1-3), I2 (0-2), I3 (3-6), I4 (2-5), I5 (5-8), I6 (3-10), I7(7-9) **(9 Marks)**

Soln. :

- **The greedy algorithm :** Refer to section 3.4.
- **Complexity analysis :** Refer to section 3.4.
- **A solution** to the given instance of activity selection problem :

- The given data is :

Activities (a)	I1	I2	I3	I4	I5	I6	I7
Start time (s)	1	0	3	2	5	3	7
Finish time (f)	3	2	6	5	8	10	9

- We first arrange all the activities in ascending order of their finish time. So, we get,

Activities (a)	I2	I1	I4	I3	I5	I7	I6
Start time (s)	0	1	2	3	5	7	3
Finish time (f)	2	3	5	6	8	9	10

- A feasible solution $S \subseteq A$ is obtained by selecting compatible activities $a_i \in A$, $1 \le i \le 7$ in such that start time $s_{i+1} \ge f_i$.

- Thus, we get,

A'	Activity under consideration	Action
{}	A[1] =I2	Select I2 as the first activity with a minimum finish time.
{I2}	A[2] = I1	As s(I1) < f(I2) (i.e. 1 < 2) we cannot select I1.
{I2}	A[3] = I4	As s(I4) = f(I2) (i.e. 2 = 2) we select I4.
{I2, I4}	A[4] = I3	As s(I3) < f(I4) (i.e. 3 < 5) we cannot select I3.
{I2, I4}	A[5] = I5	As s(I5) < f(I4) (i.e. 5 = 5) we select I5.
{I2, I4, I5}	A[6] = I7	As s(I7) < f(I5) (i.e. 7 < 8) we cannot select I7.
{I2, I4, I5}	A[7] = I6	As s(I6) < f(I5) (i.e. 3 < 8) we cannot select I6.
{I2, I4, I5}	--	--

∴ **The optimal solution to the given activity selection problem is A'= {I2, I4, I5}.**

Ex. 3.4.2 : For following intervals, select the activities according to greedy algorithm. a1 (5-7), a2 (1-2), a3 (4-6), a4 (0-5), a5 (3-4), a6 (7-10), a7(5-9), a8(0-1). **(4 Marks)**

☑ **Soln. :**

- The given data is :

Activities (a)	a1	a2	a3	a4	a5	a6	a7	a8
Start time (s)	5	1	4	0	3	7	5	0
Finish time (f)	7	2	6	5	4	10	9	1

- We first arrange all the activities in ascending order of their finish time. So, we get,

Activities (a)	a8	a2	a5	a4	a3	a1	a7	a6
Start time (s)	0	1	3	0	4	5	5	7
Finish time (f)	1	2	4	5	6	7	9	10

- A feasible solution $S \subseteq A$ is obtained by selecting compatible activities $a_i \in A$, $1 \le i \le 8$ in such that start time $s_{i+1} \ge f_i$. Thus, we get,

A'	Activity under consideration	Action
{}	A[1] = a8	Select a8 as the first activity with a minimum finish time.
{a8}	A[2] = a2	As s(a2) = f(a8) (i.e. 2 = 2) we select a2.
{a8, a2}	A[3] = a5	As s(a5) > f(a2) (i.e. 3 > 2) we select a5.
{a8, a2, a5}	A[4] = a4	As s(a4) < f(a5) (i.e. 0 < 4) we cannot select a4.
{a8, a2, a5}	A[5] = a3	As s(a3) = f(a5) (i.e. 4 = 4) we select a3.
{a8, a2, a5, a3}	A[6] = a1	As s(a1) < f(a3) (i.e. 5 < 6) we cannot select a1.
{a8, a2, a5, a3}	A[7] = a7	As s(a7) < f(a3) (i.e. 5 < 6) we cannot select a7.
{a8, a2, a5, a3}	A[7] = a6	As s(a6) > f(a3) (i.e. 7 > 6) we select a6.
{a8, a2, a5, a3, a6}	--	--

∴ **The optimal solution to given activity selection problem is : A' = {a8, a2, a5, a3, a6}.**

Summary

- The greedy method solves many problems in combinatorial optimizations by constructing multistep progressive solutions.
- The **greedy method** builds a solution in stages. At every stage, it selects the best choice concerning the local considerations.
- **A feasible solution** to a problem gives a subset of inputs that fulfils all specified constraints of a given problem.
- **An optimal solution** to a problem is a feasible solution that achieves the desired extremes.
- Greedy algorithms have two properties :
  - (i) **Optimal substructure property**: It claims that the globally optimal solution to a problem includes the optimal sub solutions within it.
  - (ii) **Greedy choice property**: Greedy algorithm obtains an optimal decision sequence by enumerating locally optimal decision sequences at every step.
- The greedy approach performs feasibility and optimality check at each step to select the currently best choice of inputs.
- The selected choice must have 3 characteristics: **feasibility, optimality** and **irrevocability**.
- **A subset paradigm** of greedy approach determines a subset of inputs that produces the optimal solution.
- **An ordering paradigm** of greedy approach determines a permutation of inputs that produces the optimal solution.
- **Some of the classic problems** that can be solved effectively by the greedy method are listed below:
  - (i) **Subset Paradigm:** Minimum spanning tree problem, knapsack problem, job sequencing with deadlines problem, tree vertex splitting problem etc.
  - (ii) **Ordering Paradigm:** Optimal storage on tapes problem, optimal merge patterns problem, single source shortest path problem etc.

Chapter Ends...

❑❑❑

UNIT

IV

Backtracking and Branch-n-Bound

>> **Syllabus :**

Backtracking : Principle, control abstraction, time analysis of control abstraction, 8-queen problem, graph coloring problem, sum of subsets problem.

Branch-n-Bound : Principle, control abstraction, time analysis of control abstraction, strategies- FIFO, LIFO and LC approaches, TSP, knapsack problem.

- Chapter 4 : Dynamic Programming
- Chapter 5 : Backtracking
- Chapter 6 : Branch and Bound

UNIT IV

CHAPTER 4

Dynamic Programming

Syllabus

Dynamic Programming: Principle, Control abstraction, time analysis of control abstraction, Binomial Coefficients, OBST, 0/1 Knapsack, Chain Matrix multiplication.

4.1 Principle and General Method 4-2

UQ. Explain dynamic programming strategy. Enlist few applications which can be solved by using dynamic programming. SPPU - Q. 6(a), March 19, 5 Marks. 4-2

4.1.1 Characteristics Components of Dynamic Programming 4-2

UQ. With respect to dynamic programming, explain in brief the following : (i) Optimal Substructure (ii) Overlapping Subproblem. SPPU - Q. 2(a), Dec. 17, 6 Marks. 4-2

4.1.2 Advantages and Disadvantages of Dynamic Programming 4-3

4.1.3 Comparison between Dynamic Programming and Greedy Approach 4-4

4.2 Calculating the Binomial Coefficient 4-4

4.3 Optimal Binary Search Trees 4-7

RQ. Define OBST. (Ref. Q. 6(b), May 14, Q.4(a), Dec. 18, Q.6(a), Feb. 15, 2 Marks) 4-7

UEx. 4.3.1 (SPPU - Q. 6, May 11,16 Marks, Q. 6(a), Dec. 11, 10 Marks, Q. 6(b) Feb. 15, Q. 5(a) Mar. 18, 8 Marks) 4-10

4.4 0/1 Knapsack Problem 4-16

4.4.1 Computations using Set Representation 4-17

4.4.2 Computations using Tabular Representation 4-21

4.5 Matrix Chain Multiplication Problem 4-24

▶ Chapter Ends 4-31

When you play chess you cannot decide any move by just considering the best move at that moment, rather you think for all future consequent moves and then only you make an optimal decision for a move at that moment. The strategy you applied here is dynamic programming.

- Dynamic programming (DP) is suitable for solving optimization problems where the optimal decision sequence cannot be obtained through stepwise decisions based on locally optimal criteria.
- The optimal decision sequence is selected from all sequences of decisions based on globally optimal criteria.
- DP applies to optimization problems having overlapping sub-problems.
- It solves each overlapping sub-problem exactly once and stores its result in a table so that this result can be reused to get the solution to the original problem.
- It avoids the re-computations of overlapping sub-problems.

4.1 PRINCIPLE AND GENERAL METHOD

UQ. Explain dynamic programming strategy. Enlist few applications which can be solved by using dynamic programming. SPPU - Q. 6(a), March 19, 5 Marks

GQ. What is dynamic programming? Is this the optimization technique? Give reasons. What are its drawbacks? Explain memory functions. (7 Marks)

(1) Recognize the objective function of a specified problem and also find out the decision variables that describe the objective function.

(2) Obtain the overlapping sub-problems of a given problem by applying polynomial breakup.

(3) Develop a recurrence formula to find out an optimal solution to a given optimization problem in terms of its overlapping sub-problems.

(4) Find out the optimal sub sequences of decisions by solving recurrence relations with overlapping sub-problems in a bottom-up manner.

(5) Construct the final optimal solution of a given problem by combining the optimal sub sequences of decisions.

4.1.1 Characteristics Components of Dynamic Programming

UQ. With respect to dynamic programming, explain in brief the following :
(i) Optimal Substructure.
(ii) Overlapping Subproblem.
SPPU - Q. 2(a), Dec. 17, 6 Marks

GQ. State true or false : "Dynamic programming always leads to an optimal solution." (1 Mark)

GQ. What is the Principle of Optimality? Explain it with an example. (4 Marks)

GQ. What are the steps for dynamic programming? Explain the principle of optimality. (6 Marks)

GQ. Justify with an example that the shortest path problem satisfies the principle of optimality. (3 Marks)

GQ. Which are the basic steps of the development of the dynamic programming algorithm? Mention any two examples of dynamic programming that we are using in real life. (6 Marks)

Overlapping sub-problems

- If a recursive algorithm for a given optimization problem solves the same sub-problems repeatedly without producing new sub-problems every time then that problem has overlapping sub-problems.

Stages

- The given optimization problem is divided into interrelated sub-problems, known as **"stages".** This division is carried out in polynomial time and hence it is known as **"polynomial breakup"**.

Stage decision

- At every stage, an optimal decision sequence is selected from different decision sequences based on global information.

Policy

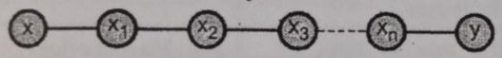
- A rule applied to make stage decisions is known as a **"policy"**.
- Dynamic programming computes all decision sequences and chooses the best one. This increases the number of computations.
- By applying some policy, dynamic programming avoids many computations of some decision sequences that do not lead to a globally optimal solution.

Principle of optimality

- Considering m choices for each of the n decisions in dynamic programming, there are total m^n possible decision sequences to be evaluated. It leads to exponential complexity.
- Some policy is needed to eliminate some non-promising decision sequences that eventually do not produce an optimal solution. This policy used in dynamic programming is known as the **"principle of optimality".**

- It states that **an optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision.**
- The principle of optimality assures that an optimal decision sequence includes all optimal sub sequences of decisions. Thus it satisfies the optimal substructure property.
- It does not generate the sub sequences of decisions that cannot be optimal.
- Though the enumeration of all decision sequences leads to an exponential complexity, the principle of optimality often reduces the complexity of dynamic programming algorithms to a polynomial-time by discarding sub-optimal sub sequences of decisions.

E.g., The shortest path problem satisfies the principle of optimality. If x, x_1, x_2, ..., x_n, y is the shortest path from a vertex x to a vertex y in a graph, then any sub-path from x_i to x_j, $1 \le i < n$, $1 < j \le n$ on that path is the shortest path from x_i to x_j.



(1D22)**Fig. 4.1.1 : The shortest path problem**

Optimal substructure

- An optimal decision sequence includes all optimal sub sequences of decisions.

Memory functions or memoization

GQ. What is memory function? Explain why it is advantageous to use memory functions. **(4 Marks)**

- Dynamic programming algorithms determine solutions by satisfying a recurrence equation with overlapping sub-problems. Such a recurrence equation is solved by the top-down approach in which the overlapping sub-problems are computed more than once giving the exponential complexity.
- The bottom-up approach followed by a classic dynamic programming algorithm computes all smaller
- sub-problems only once and stores all those results in a table. These values in a table are reused to compute the solution of the original problem. This is known as the **"tabulation method"**. But it is having one drawback of computing some unnecessary sub-problems.
- The drawbacks of both the top-down and bottom-up approaches are eliminated by the method using **"memory functions"**. It is also known as **"memoization"**. It stores the results of necessary sub-problems in a table by computing them only once.
- The dynamic programming algorithms using memory functions follow the steps below :
  (1) All entries in the table are initially marked as "null" to indicate that the sub-problem is not yet solved.
  (2) Whenever at each state a new sub-problem is needed to be computed, its associated table entry is checked. If that entry is not null then its value is directly reused to enumerate decision sequences at that stage. Otherwise, the sub-problem is computed and its result is stored in the table.
- The memoization approach does not recur completely through all overlapping sub-problems as there is no strict order of solving sub-problem.
- If the subsequence of decisions is possibly not producing an optimal result then there is no need to compute that subsequence of decisions and also its succeeding recursion tree.

Some of the classic problems that can be solved by dynamic programming are as below:

(1) Binomial coefficients problem
(2) Optimal Binary Search Tree (OBST) problem
(3) 0/1 knapsack problem
(4) Matrix chain multiplication problem
(5) Single source shortest problem
(6) All pairs shortest path problem
(7) Multistage graph problem
(8) Reliability design problem
(9) Travelling Salesman Problem (TSP)
(10) Assembly-line scheduling problem
(11) Longest common subsequence problem

4.1.2 Advantages and Disadvantages of Dynamic Programming

GQ. Explain dynamic programming and describe its advantages and disadvantages. **(7 Marks)**

Advantages of dynamic programming

(1) It is suitable for solving optimization problems with overlapping sub-problems where the optimal decision sequence cannot be generated through stepwise decisions based on local information.
(2) It solves each overlapping sub-problem exactly once and avoids the re-computations of overlapping sub-problems.
(3) It uses a sort of book-keeping and remembers the results of already solved sub-problems.

(4) Memoization technique eliminates the disadvantages of both the top-down and the bottom-up approaches. It avoids redundant computations of the top-down approach and reduces computations of unnecessary sub-problems, unlike, the bottom-up approach.

(5) It gives a globally optimal solution based on the sub sequences of decisions produced by solving overlapping sub-problems.

(6) Application of the principle of optimality avoids computations that are not leading to an optimal solution.

☞ Disadvantages of dynamic programming

(1) It cannot produce stepwise decisions based on local information.

(2) It makes decisions by enumerating all decision sequences to the sub-problems and hence does more computations.

(3) It leads to pseudo-polynomial algorithms that work efficiently only for smaller instances of a problem.

(4) For larger instances of a problem, it leads to exponential complexity.

(5) It needs some book-keeping to store results of prior calculations of sub-problems.

4.1.3 Comparison between Dynamic Programming and Greedy Approach

GQ. Differentiate dynamic programming and greedy approach. **(4 Marks)**

GQ. What are the disadvantages of the greedy method over the dynamic programming method ? **(4 Marks)**

Sr. No.	Dynamic Programming	Greedy Approach
1.	It may generate multiple decision sequences for a given problem.	It generates only one decision sequence for a given problem.
2.	It cannot produce stepwise decisions based on local information.	Stepwise decisions are made on local information.
3.	It makes globally optimal decisions that depend on the sub-sequences of decisions obtained by solving sub-problems.	While making locally optimal decisions, it does not consider results from sub-problems.
4.	It checks for the principle of optimality to make a stage decision.	It checks for feasibility and optimality to make a locally optimal decision.
5.	It computes the sub-problems before making the first choice.	It determines the first choice before computing any sub-problems.
6.	It proceeds in a bottom-up fashion.	It generally proceeds in a top-down fashion.
7.	It makes decisions by enumerating all decision sequences to the sub-problems, thus the decisions are made by considering future decisions.	It makes a decision that seems to be the best in the current stage. The decisions never depend on future decisions.
8.	E.g. Playing chess, Bellman-Ford algorithm, 0/1 knapsack problem, etc.	E.g. Playing cricket, Dijkstra's algorithm, Fractional knapsack problem etc.

4.2 CALCULATING THE BINOMIAL COEFFICIENT

GQ. Explain Binomial Coefficient algorithm using dynamic programming. **(4 Marks)**

- We can easily expand binomials $(x + y)^2$ and $(x + y)^3$ but to expand a binomial $(x + y)^n$ we need to compute its binomial coefficients ${}^nC_r$ or $\binom{n}{r}$.
- The binomial expansion of $(x + y)^n$ is given by:

$$(x+y)^n = {}^nC_0 \cdot x^n \cdot y^0 + \ldots\ldots + {}^nC_k \cdot x^{n-k} \cdot y^k + {}^nC_n \cdot x^0 \cdot y^n$$
$$= {}^nC_0 \cdot x^n + \ldots\ldots + {}^nC_k \cdot x^{n-k} \cdot y^k + {}^nC_n \cdot y^n$$

- Binomial coefficients ${}^nC_r$ or $\binom{n}{r}$ are useful to select r elements from a set of n elements.

☞ Problem description

- For a given binomial $(x + y)^n$, its binomial coefficients ${}^nC_r$ or $\binom{n}{r}$ can be calculated using the formula given below:

$\binom{n}{r} = \frac{n!}{r!(n-r)!}$ for non-negative integers n and r.

- As the factorial of a number can be a larger value, binomial coefficients $\binom{n}{r}$ or C[n, r] can be alternatively calculated using its Pascal identity. So, the recursive formula of $\binom{n}{r}$ is given by the following equation (1).

$$\binom{n}{r} \text{ or } c[n, r] = \begin{cases} 1 & ; \text{if } r = 0 \\ 1 & ; \text{if } r = n \\ \binom{n-1}{r} + \binom{n-1}{r-1} & ; 0 < r < n \end{cases} \quad \ldots(1)$$

The general procedure

- To compute binomial coefficients $\binom{n}{r}$ by dynamic programming, we need to solve the overlapping subproblems $\binom{0}{0}, \binom{1}{0}, \binom{1}{1}, \binom{2}{0}, \ldots\ldots, \binom{n}{n}$ in a bottom-up manner.

$S_0 = \left\{\binom{0}{0}\right\}$,

$S_1 = \left\{\binom{0}{0}, \binom{1}{1}\right\}$,

$S_2 = \left\{\binom{2}{0}, \binom{2}{1}, \binom{2}{2}\right\}$,

:

:

$S_n = \left\{\binom{n}{0}, \binom{n}{1}, \binom{n}{2}, \ldots\ldots, \binom{n}{n}\right\}$

- We use memoization for storing solutions to the overlapping subproblems to avoid recomputation of the subproblems. S_{i+1} is computed by using S_i as defined in equation (1). The solutions to the subproblems are stored in the table in which the i^{th} row corresponds to S_i. It is shown in Table 4.2.1. This representation of the binomial coefficients in tabular form is known as Pascal's triangle.

Table 4.2.1 : Pascal's Triangle

n \ r	0	1	2	3	.....	r-1	r
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
:							
:							
n-1	$\binom{n-1}{0}$	$\binom{n-1}{1}$	$\binom{n-1}{2}$	$\binom{n-1}{3}$	.....	$\binom{n-1}{r-1}$	$\binom{n-1}{r}$
n	$\binom{n}{0}$	$\binom{n}{1}$	$\binom{n}{2}$	$\binom{n}{3}$	.....	$\binom{n}{r-1}$	$\binom{n}{r}$

Algorithm

```
Algorithm Binomial_Coeff(n, r)
/* It computes the binomial coefficient C(n, r) by dynamic programming.
Input : Non-negative numbers n and r.
Output : C[n, r]: the binomial coefficient C(n, r).*/
{
    for (i := 0 ; i ≤ n ; i++)
    {
        for(j := 0 ; j ≤min(i, r) ; j++)   /* min(i, r) gives the
                                              minimum of i and r.*/
```

```
    {
    if (j=0 || j=i)
        C[i, j] := 1; /*Base condition of recursion.*/
    else
        C[i, j] := C[i-1, j-1] + C[i-1, j];
                /* Recursive formula to compute (n r). */
    }
}
return C[n, r];
}
```

☞ Complexity analysis

- To calculate binomial coefficient by DP it needs to fill the table by computing overlapping subproblems. The basic operation in this computation is addition.
- Since $r \leq n$, the sum is divided into two parts as for $i < r$ the algorithm fills only the half table, and it fills the entire row for the remaining part of the table.

$C[n, r]$ = sum for upper triangle + sum for the lower rectangle

$$= \sum_{1 \leq i \leq r} \sum_{1 \leq j \leq i-1} 1 + \sum_{1 \leq i \leq n} \sum_{1 \leq j \leq r} 1$$

$$= \sum_{1 \leq i \leq r} (i-1) + \sum_{1 \leq i \leq n} r$$

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

$$= \Theta(nr)$$

- The algorithm Binomial_Coeff(n, r) based on dynamic programming takes time $\Theta(nr)$.
- The worst-case running time for computing the binomial coefficient is $O(n^2)$.

Ex. 4.2.1 : Find out the binomial coefficient $\binom{6}{2}$ using dynamic programming. **(4 Marks)**

☑ Soln. :

- To compute binomial coefficient $\binom{n}{r}$ or C[n, r] using dynamic programming, we use the following formula :

$$\binom{n}{r} \text{ or } C[n, r] = \begin{cases} 1 & ; \text{if } r = 0 \\ 1 & ; \text{if } r = n \\ \binom{n-1}{r} + \binom{n-1}{r-1} & ; 0 < r < n \end{cases}$$

- Here, n = 6 and r = 2. We fill Pascal's triangle to find out $\binom{6}{2}$ or C[6, 2].

Initially, C[0, 0] = 1;

C[1, 0] = 1; C[1, 1] = 1;

C[2, 0] = 1;

C[2, 1] = C[1, 0] + C[1, 1] = 1 + 1 = 2;

C[2, 2] = 1;

C[3, 0] = 1;

C[3, 1] = C[2, 1] + C[2, 0] = 2 + 1 = 3 ;

C[3, 2] = C[2, 2] + C[2, 1] = 1 + 2 = 3 ;

C[4, 0] = 1;

C[4, 1] = C[3, 1] + C[3, 0] = 3 + 1 = 4 ;

C[4, 2] = C[3, 2] + C[3, 1] = 3 + 3 = 6 ;

C[5, 0] = 1 ;

C[5, 1] = C[4, 1] + C[4, 0] = 4 + 1 = 5 ;

C[5, 2] = C[4, 2] + C[4, 1] = 6 + 4 = 10 ;

C[6, 0] = 1;

C[6, 1] = C[5, 1] + C[5, 0] = 5 + 1 = 6 ;

C[6, 2] = C[5, 2] + C[5, 1] = 10 + 5 = 15 ;

Pascal's Triangle for $\binom{6}{2}$

n \ r	0	1	2
0	1		
1	1	1	
2	1	2	1
3	1	3	3
4	1	4	6
5	1	5	10
6	1	6	15

Thus, $\binom{6}{2} = 15$.

Ex. 4.2.2 : Find out the NCR $\binom{5}{3}$ using Dynamic Method. **(4 Marks)**

Soln. :

- To compute binomial coefficient $\binom{n}{r}$ or C[n, r] using dynamic programming, we use the following formula:

$$\binom{n}{r} \text{ or } C[n, r] = \begin{cases} 1 & ; \text{if } r = 0 \\ 1 & ; \text{if } r = n \\ \binom{n-1}{r} + \binom{n-1}{r-1} & ; 0 < r < n \end{cases}$$

- Here, n = 6 and r = 2. We fill Pascal's triangle to find out $\binom{5}{3}$ or C[5, 3].

Initially, C[0, 0] = **1**;

C[1, 0] = **1**; C[1, 1] = **1**;

C[2, 0] = **1**;

C[2, 1] = C[1, 0] + C[1, 1] = 1 + 1 = **2** ;

C[2, 2] = **1**;

C[3, 0] = **1**;

C[3, 1] = C[2, 1] + C[2, 0] = 2 + 1 = **3** ;

C[3, 2] = C[2, 2] + C[2, 1] = 1 + 2 = **3** ;

C[3, 3] = **1**;

C[4, 0] = 1;

C[4, 1] = C[3, 1] + C[3, 0] = 3 + 1 = **4** ;

C[4, 2] = C[3, 2] + C[3, 1] = 3 + 3 = **6** ;

C[4, 3] = C[3, 3] + C[3, 2] = 1 + 3 = **4** ;

C[5, 0] = **1**;

C[5, 1] = C[4, 1] + C[4, 0] = 4 + 1 = **5** ;

C[5, 2] = C[4, 2] + C[4, 1] = 6 + 4 = **10** ;

C[5, 3] = C[4, 3] + C[4, 2] = 4 + 6 = **10** ;

Pascal's Triangle for $\binom{5}{3}$

n \ r	0	1	2	3
0	1			
1	1	1		
2	1	2	1	
3	1	3	3	1
4	1	4	6	4
5	1	5	10	10

Thus, $\binom{5}{3} = 10$.

4.3 OPTIMAL BINARY SEARCH TREES

- Consider a program that takes a common name of any plant as an input and gives the corresponding botanical (scientific) name of that plant as an output.
- To have fast lookup operations we can construct a binary search tree with n common names of plants as keys and their equivalent botanical names as payload (satellite) data. By constructing a balanced binary search tree we can have the time complexity of searching as O(log n) for n common names of plants.
- Suppose a document includes different plant names with different frequencies then, to have efficient searching for plant names in a binary search tree the frequently referred plant names should be arranged nearer to the root than the rarely referred plant names, because in a binary search tree number of comparisons done for searching a key element is equal to the level of a node with that key (The level is considered w.r.t. level of the root = 1). So, the binary search tree with n common names of plants as keys should be organized by considering their frequency of occurrences in a document in such a way that it performs the minimum number of comparisons in all searches. This is a real-life example of optimal binary search tree (OBST) problem.

Problem description

RQ. Define OBST.

Ref. Q. 6(b), May 14, Q.4(a), Dec. 18, Q.6(a), Feb. 15, 2 Marks

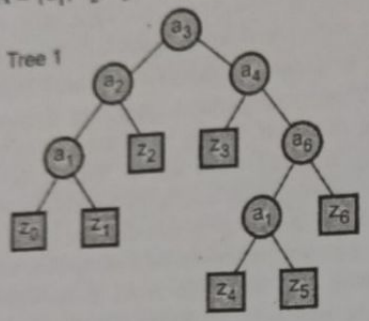
- Consider a set of n distinct key elements $A = \{a_1, a_2, \ldots, a_n\}$ arranged in ascending order $(a_1 < a_2 < \ldots\ldots < a_n)$ and $(p_1, p_2, \ldots, p_n)$ be the respective probabilities of searching for them.
- Let z is a dummy key such that $a_i < z < q_i$ where $0 \le i \le n$, $a_0 = -\infty$ and $a_{n+1} = +\infty$. So, search for any such z in a given set A results in an unsuccessful search. The probability of an unsuccessful search can be given as q_i, $0 \le i \le n$.
- Then the total probability of a search can be given as :

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1 \qquad \ldots(4.3.1)$$

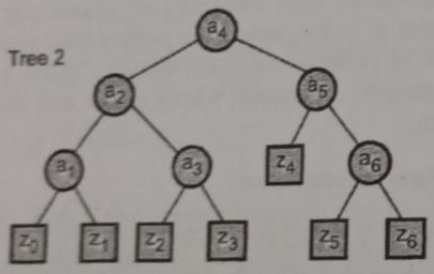
Probability of a successful search | Probability of an unsuccessful search

- With this data a binary search tree T can be built where each key element a_i, $1 \le i \le n$ in a set A represents an internal node and each dummy key element z_i, $0 \le i \le n$ represents an external (a leaf) node.
- A successful search terminates at an internal node; however, an unsuccessful search terminates at an external node in that binary search tree.

E.g. $A = \{a_1, a_2, a_3, a_4, a_5, a_6\}$



(1D9)(a)



(1D10)(b)

Fig. 4.3.1 : Two possible representations of binary search trees for A

- Considering the probabilities for a successful and an unsuccessful search in given binary search tree T, the expected cost of a search in T can be estimated.
- The actual cost of a search in T can be defined in terms of the number of comparisons done to search an element in T. It is equal to the level of a node with that element. Consider the root is at level 1. Thus the expected search cost in T can be calculated as :

$$E[\text{search-cost in T}] = \sum_{i=1}^{n} p_i * (\text{level}(a_i)) + \sum_{i=0}^{n} q_i * (\text{level}(z_i) - 1)$$

p_i : probability of a successful search

(level (a_i)) : level of an internal node (the end point of a successful search)

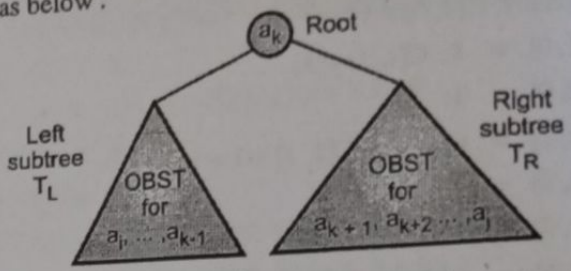
q_i : probability of an unsuccessful search

(level $(z_i) - 1$) : level of an external node (the end point of an unsuccessful search)

- The optimal binary search tree (OBST) problem is a problem to construct a binary search tree for a given set of key elements $A = \{a_1, a_2, \ldots\ldots, a_n\}$ with associated probabilities of successful searches $\{p_1, p_2, \ldots\ldots, p_n\}$ and probabilities of unsuccessful searches $\{q_0, q_1, q_2, \ldots\ldots q_n\}$ such that the expected cost of the searches in it is minimum.

The general procedure

- The optimal substructure of an OBST can be depicted as below :



(1D11)**Fig. 4.3.2 : OBST $T_{i,j}$ for key elements $a_i, a_{i+1}, \ldots\ldots, a_j$ where $1 \le i \le j \le n$ and $i \le k \le j$**

- Here, the left subtree T_L is an OBST with keys $a_i, \ldots, a_{k-1}$ and dummy keys $z_{i-1}, \ldots, z_{k-1}$. The right subtree T_R is an OBST with keys $a_{k+1}, \ldots\ldots, a_j$ and dummy key $z_k, \ldots\ldots, z_j$, $1 \le i \le j \le n$ and $i \le k \le j$. Any key a_k, $i \le k \le j$, is the root of T_L and T_R.
- If a key element a_i is selected as the root then T_L will have only one dummy key z_{i-1}.
- Similarly, if a_j is chosen as the root then T_R will have only one dummy key z_j.
- The dynamic programming approach generates an OBST through a sequence of decisions by selecting different a_k's , $1 \le k \le n$ as the root of the optimal subtree.
- Every time it is confirmed that the principle of optimality holds for the resultant problem state.
- When any key a_k , $1 \le k \le n$, among all n key elements $a_1, a_2, \ldots, a_n$ is selected as the root of the optimal subtree then keys $a_1, \ldots, a_{k-1}$ and fictitious keys $z_0, \ldots, z_{k-1}$ lie in the left subtree T_L and remaining keys and fictitious keys lie in the right subtree T_R. Hence the costs of both T_L and T_R can be given as:

$$\text{Cost}(T_L) = \sum_{1 \le i < k} p_i * (\text{level}(a_i)) + \sum_{0 \le i < k} q_i * (\text{level}(z_i) - 1)$$

and

$$\text{Cost}(T_R) = \sum_{k < i \le n} p_i * (\text{level}(a_i)) + \sum_{k < i \le n} q_i * (\text{level}(z_i) - 1)$$

- The level is considered w.r.t. the root of each subtree at level 1.
- As per the optimal substructure property, both the T_L and T_R should be developed as OBSTs. Since T_L and T_R are connected to the root a_k, the level of each of the node in T_L and T_R increases by 1.
- Thus the expected search cost in both T_L and T_R get increased by the sum of successful and unsuccessful probabilities of keys in them.
- The sum of probabilities (successful and unsuccessful) of any subtree with key elements a_i, ..., a_j is defined as :

$$w(i, j) = \sum_{h=i}^{j} p_h + \sum_{h=i-1}^{j} q_h$$

$$\therefore \quad w(i, j) = q_{i-1} + \sum_{h=i}^{j} (p_i + q_i) \qquad ; 1 \le i \le j \le n.$$

- Let c(i, j) be the expected search cost of an OBST $T_{i,j}$ having key elements a_i, ..., a_j. The root a_k is selected such that the cost of T_L and the cost of T_R are the smallest values resulting in minimum cost of $T_{i,j}$.

$$\therefore \quad c(0, n) = \min_{1 < k \le j} \{c(0, k-1) + c(k, n) + p_k + w(0, k-1) + w(k, n)\}$$

Here, c(0, n) is the expected search cost of an OBST with n keys a_1, ..., a_n and n + 1dummy keys z_0, ..., z_n.

c(0, k – 1) is the cost of T_L and c(k, n) is the cost of T_R.

p_k is the probability of successful search of the root element a_k.

w(0, k – 1) is the sum of all probabilities in T_L and w(k, n) is the sum of all probabilities in T_R.

- Thus generalized recursive formula for any c(i, j) is given as:

$$c(i, j) = \min_{i < k \le j} \{c(i, k-1) + c(k, j) + p_k + w(i, k-1) + w(k, j)\}$$

$$\therefore c(i, j) = \min_{i < k \le j} \{c(i, k-1) + c(k, j) + w(i, j)\} \quad ; 0 \le i \le j \le n.$$

$$c(i, i) = 0; \; w(i, i) = q_i;$$

$$w(i, j) = p_j + q_j + w(i, j-1) \quad ; 0 \le i \le j \le n$$

- The cost c(0, n) for an OBST with n key elements is estimated by first calculating all c(i, j)'s such that j – i = 1, then calculating all c(i, j) 's such that j – i = 2 and so on.
- During these calculations, the selected root element of an optimal subtree obtained at every decision step is noted. Let r(i, j) be the root of an optimal subtree $T_{i,j}$ with keys a_i,..., a_j. Then r(i, j) = k so that the tree with the root a_k, $i \le k \le j$ has the smallest value of expected search cost.
- By tracing all r(i, j) values of all subtrees $T_{i,j}$ the final OBST can be constructed with n key elements. Initially r(i, j) = 0.

Algorithm

```
Algorithm OBST_DP(A, p, q, n)
/*Input: A is a set of n distinct keys arranged in increasing
order so that a1 < a2 < ……< an. p[1:n] is a set of associated
probabilities of successful searches and q[0:n] is a set of
associated probabilities of unsuccessful searches of given
keys.
Output: c[i, j] is the optimal search cost of an OBST Tij for
keys ai+1, ….,aj , 0 ≤ i ≤ j ≤ n. r[i, j] is the selected root of Tij
that minimizes the c[i, j] and w[i, j] is the weight (sum of all
probabilities) of Tij.*/
{
  // Initialization
  for(i:=0; i < n; i++)
  {
     w[i, i]:= q[i];
     c[i, i]:= 0.0;
     r[i, i]:= 0;
  // optimal subtree with a single node
     w[i, i + 1]:= p[i + 1] + q[i + 1] + w[i, i];
     c[i, i + 1]:= c[i, i] + c[i + 1, i + 1] + w[i, i + 1];
     r[i, i + 1]:= i + 1;
  }
  // Initialization for the last key
  w[n, n]:= q[n];
  c[n, n]:= 0.0;
  r[n, n]:=0;
  // Construct optimal trees with m nodes
  for (m:=2; m ≤ n; m++)
```

```
{
  for (i:=0; i ≤ n – m; i++)
  {
    j:=i + m;
    w[i, j] = p[j] + q[j] + w[i, j – 1];
    k = Select_Root(i, j, c, r);
                    //Select_Root() selects the root that
                    minimizes the search cost in OBST T_ij.*/
    c[i, j]:= c[i, k – 1] + c[k, j] + w[i, j]
    r[i, j]:= k;
  }
}
write (c[0, n], w[0, n], r[0, n]);
}

Algorithm Select_Root (i, j, c, r)
{
/* It selects the root element that minimizes the search cost in
OBST T_ij. It considers search range proposed by D. E. Knuth
for the roots as r[i, j – 1] ≤ k ≤ r[i + 1, j] that minimizes
c[i, k – 1] + c[k, j].*/
  mincost := ∞;
  for(h:= r[i, j – 1]; h ≤ r [i + 1, j]; h++)
  {
    if (c[i, h – 1] + c[h, j] < mincost)
    {
      mincost := c[i, h – 1] + c[h, j];
      Root:= h;
    }
  }
  return Root;
}
```

☞ Complexity analysis

- The expected search cost c(i, j) for an OBST $T_{i,j}$ is calculated for (j – i) = 1, 2, 3, ..., n in the sequence. Thus, for j – i = m the algorithm calculates (n – m + 1) c(i, j)'s.
- Each c(i, j) is computed by comparing m values to get the smallest value of search cost. So the time complexity of calculating each c(i, j) = O(m). Thus the time complexity of computations of all c(i, j)'s with m = (j – i) is $O(mn - m^2)$.
- Since, the algorithm calculates c(i, j)'s for(j – i) = 1, 2, ..., n in the sequence, the total time required to compute all the c(i, j)'s and r(i, j)'s is given as:

$$\sum_{1 \le m \le n} (mn - m^2) = O(n^3)$$

- This complexity can be reduced by following the search range for the root as proposed by D. E. Knuth. He claims that the search range for the root selection can be limited to $r(i, j-1) \le k \le r(i+1, j)$ to get the time complexity of $O(n^2)$.
- Once the all computations of c(i, j)'s and r(i, j)'s are completed to get c(0, n) and r(0, n) then the OBST $T_{0,n}$ can be developed in time O(n).
- Thus the total time complexity of OBST by dynamic programming will be either:

$$O(n^3) + O(n) = O(n^3)$$

To find c (i, j)'s, r (i, j)'s | To construct, final OBST $T_{0,n}$

or by D. E. Knuth's algorithm

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

To find c (i, j)'s, & r (i, j)'s | To construct, final OBST $T_{0,n}$

UEx. 4.3.1 SPPU - Q. 6, May 11,16 Marks, Q. 6(a), Dec. 11, 10 Marks, Q. 6(b) Feb. 15, Q. 5(a) Mar. 18, 8 Marks

Let N = 3 and (a1, a2, a3) = (do, if, while). Let p(1:3) = (0.5, 0.1, 0.05), q(0:3) = (0.15, 0.1, 0.05, 0.05). Compute and construct OBST for above values using dynamic programming.

☑ **Soln. :**

– To solve a given instance of the OBST by dynamic programming we use following recursive formulae to calculate c(i, j), w(i, j) and r(i, j) for an optimal subtree $T_{i,j}$, $0 \le i \le j \le n$.

$$c(i, j) = \min_{i < k \le j} \{ c(i, k-1) + c(k, j) \} + w(i, j)$$

Expected search, cost of left, subtree T_L | **Expected search, cost of right, subtree T_R** | **sum of all, probabilities**

$$c(i, i) = 0; \quad w(i, i) = q_i; \quad r(i, i) = 0;$$

$$w(i, j) = p_j + q_j + w(i, j-1), \quad 0 \le i \le j \le n$$

We recursively compute w(0, 3), c(0, 3) and r(0, 3) and represent all values in a tabular form as depicted in Fig. Ex. 4.3.1

$w(0, 0) = q_0 = \mathbf{0.15}$; $c(0, 0) = \mathbf{0}$; $r(0, 0) = \mathbf{0}$

$w(1, 1) = q_1 = \mathbf{0.1}$; $c(1, 1) = \mathbf{0}$; $r(1, 1) = \mathbf{0}$

$w(2, 2) = q_2 = \mathbf{0.05}$; $c(2, 2) = \mathbf{0}$; $r(2, 2) = \mathbf{0}$

$w(3, 3) = q_3 = \mathbf{0.05}$; $c(3, 3) = \mathbf{0}$; $r(3, 3) = \mathbf{0}$

$w(0, 1) = p_1 + q_1 + w(0, 0) = 0.5 + 0.1 + 0.15 = \mathbf{0.75}$

$c(0, 1) = c(0, 0) + c(1, 1) + w(0, 1) = 0 + 0 + 0.75$

$= \mathbf{0.75}$

$r(0, 1) = \mathbf{1}$

$w(1, 2) = p_2 + q_2 + w(1, 1) = 0.1 + 0.05 + 0.1 = \mathbf{0.25}$

$c(1, 2) = c(1, 1) + c(2, 2) + w(1, 2) = 0 + 0 + 0.25$

$= \mathbf{0.25}$

$r(1, 2) = \mathbf{2}$

$w(2, 3) = p_3 + q_3 + w(2, 2)$

$= 0.05 + 0.05 + 0.05 = \mathbf{0.15}$

$c(2, 3) = c(2, 2) + c(3, 3) + w(2, 3) = 0 + 0 + 0.15$

$= \mathbf{0.15}$

$r(2, 3) = \mathbf{3}$

$w(0, 2) = p_2 + q_2 + w(0, 1) = 0.1 + 0.05 + 0.75 = \mathbf{0.9}$

$c(0, 2) = \min \{c(0, 0) + c(1, 2), c(0, 1) + c(2, 2)\} + w(0, 2)$

$= \min \{0 + 0.25, 0.75 + 00\} + 0.9$

$= 0.25 + 0.9 = \mathbf{1.15}$

$r(0, 2) = \mathbf{1}$

$w(1, 3) = p_3 + q_3 + w(1, 2) = 0.05 + 0.05 + 0.25$

$= \mathbf{0.35}$

$c(1, 3) = \min \{c(1, 1) + c(2, 3), c(1, 2) + c(3, 3)\} + w(1, 3)$

$= \min \{(0 + 0.15), (0.25 + 0)\} + 0.35$

$= 0.15 + 0.35 = \mathbf{0.5}$

$r(1, 3) = \mathbf{2}$

$w(0, 3) = p_3 + q_3 + w(0, 2) = 0.05 + 0.05 + 0.9 = \mathbf{1}$

$$c(0, 3) = \min \left\{ \begin{array}{l} c(0, 0) + c(1, 3), \\ c(0, 1) + c(2, 3), \\ c(0, 2) + c(3, 3) \end{array} \right\} + w(0, 3)$$

$= \min \{0 + 0.5, 0.75 + 0.15, 1.15 + 0\} + 1$

$= \min \{0.5, 0.9, 1.15\} + 1 = 0.5 + 1 = \mathbf{1.5}$

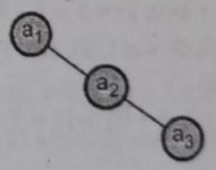
$r(0, 3) = \mathbf{1}$

- From the computations table in Fig. Ex. 4.3.1 we have **c(0, 3) = 1.5** as the minimum search cost and **r(0,3)=1** as the root of an OBST $T_{0,3}$ for given 3 keys.
- The root of a tree $T_{0,3}$ is a_1 giving a left subtree $T_L = T_{0,0}$ (i.e. no key element, only fictitious node in T_L) and a right subtree $T_R = T_{1,3}$.
- The root of a tree $T_{1,3}$ is r(1,3)=2, so its root is a_2 giving a left subtree $T_L = T_{1,1}$ and a right subtree $T_R = T_{2,3}$.
- The root of $T_{2,3}$ is r(2,3)=3, so its root is a_3 giving a left subtree $T_L = T_{2,2}$ and a right subtree $T_R = T_{3,3}$.

i ↓ / j →	0	1	2	3
0	$w_{00} = 0.15$ $c_{00} = 0$ $r_{00} = 0$	$w_{11} = 0.1$ $c_{11} = 0$ $r_{11} = 0$	$w_{22} = 0.05$ $c_{22} = 0$ $r_{22} = 0$	$w_{33} = 0.05$ $c_{33} = 0$ $r_{33} = 0$
1	$w_{01} = 0.75$ $c_{01} = 0.75$ $r_{01} = 1$	$w_{12} = 0.25$ $c_{12} = 0.25$ $r_{12} = 2$	$w_{23} = 0.15$ $c_{23} = 0.15$ $r_{23} = 3$	
2	$w_{02} = 0.9$ $c_{02} = 1.15$ $r_{02} = 1$	$w_{13} = 0.35$ $c_{13} = 0.5$ $r_{13} = 2$		
3	$w_{03} = 1$ $c_{03} = \mathbf{1.5}$ $r_{03} = 1$	Here each cell in row i and column j represents the values w(j, j + i), c(j, j + i) and r(j, j + i), $0 \le i \le j \le n$		

Fig. Ex. 4.3.1 : Computation of w(0, 3) c(0, 3) and r(0, 3)

- So the final OBST $T_{0,3}$ is depicted as below :



(1D13)**Fig. Ex. 4.3.1(a) : OBST $T_{0,3}$**

- **The expected minimum search cost c(0, 4) = 1.5 units.**

Ex. 4.3.2 : N = 4 and (a1, a2, a3, a4) = (DAA, ITPM, OS, SP). Let p(1:4) = (3, 3, 1, 1) and q(0:4) = (2, 3, 1, 1, 1). Compute and construct OBST for above values using dynamic programming. **(10 Marks)**

☑ **Soln. :**

To solve a given instance of the OBST by dynamic programming we use following recursive formulae to calculate c(i, j), w(i, j) and r(i, j) for an optimal subtree $T_{i,j}$, $0 \le i \le j \le n$.

$$c(i, j) = \min_{i<k\le j} \{ c(i, k-1) + c(k, j) \} + w(i, j)$$

Expected search, cost of left, subtree T_L | Expected search, cost of right, subtree T_R | sum of all, probabilities

$c(i, i) = 0;\ w(i, i) = q_i;\ r(i, i) = 0;$

$w(i, j) = p_j + q_j + w(i, j-1),\ 0 \le i \le j \le n$

We recursively compute w(0, 4), c(0, 4) and r(0, 4) and represent all values in a tabular form as depicted in Fig. Ex. 4.3.2

$w(0, 0) = q_0 = 2;\ c(0, 0) = 0;\ r(0, 0) = 0$

$w(1, 1) = q_1 = 3;\ c(1, 1) = 0;\ r(1, 1) = 0$

$w(2, 2) = q_2 = 1;\ c(2, 2) = 0;\ r(2, 2) = 0$

$w(3, 3) = q_3 = 1;\ c(3, 3) = 0;\ r(3, 3) = 0$

$w(4, 4) = q_4 = 1,\ c(4, 4) = 0,\ r(4, 4) = 0$

$w(0, 1) = p_1 + q_1 + w(0, 0) = 3 + 3 + 2 = 8$

$c(0, 1) = c(0, 0) + c(1, 1) + w(0, 1) = 0 + 0 + 8 = 8$

$r(0, 1) = 1$

$w(1, 2) = p_2 + q_2 + w(1, 1) = 3 + 1 + 3 = 7$

$c(1, 2) = c(1, 1) + c(2, 2) + w(1, 2) = 0 + 0 + 7 = 7$

$r(1, 2) = 2$

$w(2, 3) = p_3 + q_3 + w(2, 2) = 1 + 1 + 1 = 3$

$c(2, 3) = c(2, 2) + c(3, 3) + w(2, 3) = 0 + 0 + 3 = 3$

$r(2, 3) = 3$

$w(3, 4) = p_4 + q_4 + w(3, 3) = 1 + 1 + 1 = 3$

$c(3, 4) = c(3, 3) + c(4, 4) + w(3, 4) = 0 + 0 + 3 = 3$

$r(3, 4) = 4$

$w(0, 2) = p_2 + q_2 + w(0, 1) = 3 + 1 + 8 = 12$

$c(0, 2) = \min \{c(0, 0) + c(1, 2), c(0, 1) + c(2, 2)\} + w(0, 2)$

$= \min \{0 + 7, 8 + 0\} + 12 = 7 + 12 = 19$

$r(0, 2) = 1$

$w(1, 3) = p_3 + q_3 + w(1, 2) = 1 + 1 + 7 = 9$

$c(1, 3) = \min \{c(1, 1) + c(2, 3), c(1, 2) + c(3, 3)\} + w(1, 3)$

$= \min \{0 + 3, 7 + 0\} + 9 = 12$

$r(1, 3) = 2$

$w(2, 4) = p_4 + q_4 + w(2, 3) = 1 + 1 + 3 = 5$

$c(2, 4) = \min \{c(2, 2) + c(3, 4), c(2, 3) + c(4, 4)\} + w(2, 4)$

$= \min \{0 + 3, 3 + 0\} + 5 = 3 + 5 = 8$

$r(2, 4) = 3 \text{ or } 4$

$w(0, 3) = p_3 + q_3 + w(0, 2) = 1 + 1 + 12 = 14$

$$c(0, 3) = \min \begin{Bmatrix} c(0, 0) + c(1, 3), \\ c(0, 1) + c(2, 3), \\ c(0, 2) + c(3, 3) \end{Bmatrix} + w(0, 3)$$

$= \min \{0 + 12, 8 + 3, 19 + 0\} + 14$

$= \min (12, 11, 19) + 14 = 11 + 14 = 25$

$r(0, 3) = 2$

$w(1, 4) = p_4 + q_4 + w(1, 3) = 1 + 1 + 9 = 11$

$$c(1, 4) = \min \begin{Bmatrix} c(1, 1) + c(2, 4), \\ c(1, 2) + c(3, 4), \\ c(1, 3) + c(4, 4) \end{Bmatrix} + w(1, 4)$$

$= \min \{0 + 8, 7 + 3, 12 + 0\} + 11$

$= \min \{8, 10, 12\} + 11 = 8 + 11 = 19$

$r(1, 4) = 2$

$w(0, 4) = p_4 + q_4 + w(0, 3) = 1 + 1 + 14 = 16$

$$c(0, 4) = \min \begin{Bmatrix} c(0, 0) + c(1, 4), \\ c(0, 1) + c(2, 4), \\ c(0, 2) + c(3, 4), \\ c(0, 3) + c(4, 4) \end{Bmatrix} + w(0, 4)$$

$= \min \{0 + 19, 8 + 8, 19 + 3, 25 + 0\} + 16$

$= \min \{19, 16, 22, 25\} + 16 = 16 + 16 = 32$

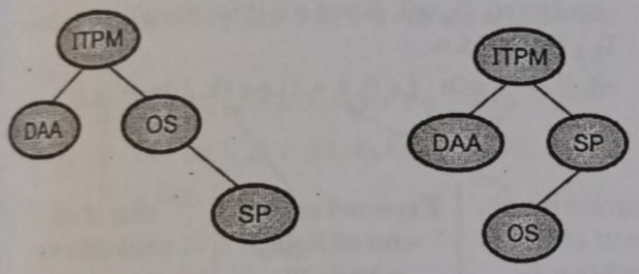
$r(0, 4) = 2$

i ↓ / j →	0	1	2	3	4
0	$w_{00} = 2$ $c_{00} = 0$ $r_{00} = 0$	$w_{11} = 3$ $c_{11} = 0$ $r_{11} = 0$	$w_{22} = 1$ $c_{22} = 0$ $r_{22} = 0$	$w_{33} = 1$ $c_{33} = 0$ $r_{33} = 0$	$w_{44} = 1$ $c_{44} = 0$ $r_{44} = 0$
1	$w_{01} = 8$ $c_{01} = 8$ $r_{01} = 1$	$w_{12} = 7$ $c_{12} = 7$ $r_{12} = 2$	$w_{23} = 3$ $c_{23} = 3$ $r_{23} = 3$	$w_{34} = 3$ $c_{34} = 3$ $r_{34} = 4$	
2	$w_{02} = 12$ $c_{02} = 19$ $r_{02} = 1$	$w_{13} = 9$ $c_{13} = 12$ $r_{13} = 2$	$w_{24} = 5$ $c_{24} = 8$ $r_{24} = 3$ or 4		
3	$w_{03} = 14$ $c_{03} = 25$ $r_{03} = 2$	$w_{14} = 11$ $c_{14} = 19$ $r_{14} = 2$			
4	$w_{04} = 16$ $c_{04} = 32$ $r_{04} = 2$				

Here each cell in row i and column j represents w(j, j + i), c(j, j + i) and r(j, j + i)

Fig. Ex. 4.3.2 : Computation of w(0, 4), c(0, 4) and r(0, 4)

- From the computation table in Fig. Ex. 4.3.2 we have $c(0, 4) = 32$ as the minimum search cost and $r(0,4)=2$ as the root of an OBST $T_{0,4}$ for given 4 keys.
- The root of a tree $T_{0,4}$ is a_2 giving a left subtree $T_L = T_{0,1}$ and a right subtree $T_R = T_{2,4}$.
- The root of a tree $T_{0,1}$ is $r(0,1)=1$, so its root is a_1 giving a left subtree $T_L = T_{0,0}$ and a right subtree $T_R = T_{1,1}$.
- The root of a tree $T_{2,4}$ is $r(2,4)=3$ or 4, so its root is a_3 or a_4.
- If a_3 is the root of $T_{2,4}$ then we get $T_L = T_{2,2}$ and $T_R = T_{3,4}$. The root of $T_{3,4}$ is $r(3,4) = 4$, so its root is a_4 giving left subtree $T_L = T_{3,3}$ and $T_R = T_{4,4}$.
- If a_4 is the root of $T_{2,4}$ then we get a left subtree $T_L = T_{2,3}$ and right subtree as $T_{4,4}$. The root of $T_{2,3}$ is $r(2,3) = 3$, so its root is a_3 giving a left subtree $T_L = T_{2,2}$ and right subtree as $T_R = T_{3,3}$.
- So the final OBST $T_{0,4}$ is depicted as below :



(1D14) (1D15)

Fig. Ex. 4.3.2(a) : OBST $T_{0,4}$

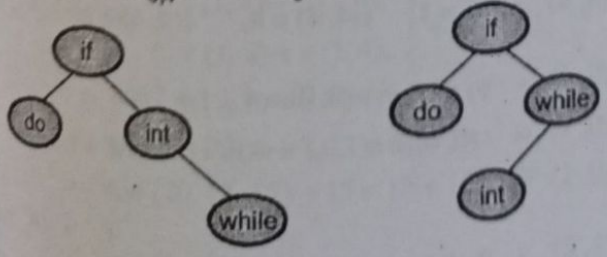
- **The expected minimum search cost c(0, 4) = 32 units.**

Ex. 4.3.3 : Construct the optimal binary search tree for data n = 4 and (a1, a2, a3, a4) = (do, if, int, while). Let p(1:4) = (3, 3, 1, 1) and q(0:4) = (2, 3, 1, 1, 1) **(10 Marks)**

☑ **Soln. :**

Refer the solution to UEx. 4.3.3 for all computations.

- Final OBST $T_{0,4}$ can be depicted as



OR

(1D16) (1D17)

Fig. Ex. 4.3.3 : OBST $T_{0,4}$

- **The expected minimum search cost c(0, 4) = 32 units.**

Ex. 4.3.4 : Compute and construct OBST for the given values using dynamic programming. Let N = 3 and (a1, a2, a3) = (do, if, int). Let p(1:3) = (4, 2, 1), q(0:3) = (2, 3, 1, 5).

☑ **Soln. :**

- To solve a given instance of the OBST by dynamic programming we use following recursive formulae to calculate c(i, j), w(i, j) and r(i, j) for an optimal subtree $T_{i,j}$, $0 \le i \le j \le n$.

$$c(i, j) = \min_{i<k\le j} \{ c(i, k-1) + c(k, j)\} + w(i, j)$$

Expected search, cost of left, subtree T_L | Expected search, cost of right, subtree T_R | sum of all, probabilities

$$c(i, i) = 0;\quad w(i, i) = q_i;\quad r(i, i) = 0;$$

$$w(i, j) = p_j + q_j + w(i, j-1),\ 0 \le i \le j \le n$$

We recursively compute w(0, 4), c(0, 4) and r(0, 4) and represent all values in a tabular form as depicted in Fig. Ex. 4.3.4

$w(0, 0) = q_0 = 2;\quad c(0, 0) = 0;\quad r(0, 0) = 0$

$w(1, 1) = q_1 = 3;\quad c(1, 1) = 0;\quad r(1, 1) = 0$

$w(2, 2) = q_2 = 1;\quad c(2, 2) = 0;\quad r(2, 2) = 0$

$w(3, 3) = q_3 = 5;\quad c(3, 3) = 0;\quad r(3, 3) = 0$

$w(0, 1) = p_1 + q_1 + w(0, 0) = 4 + 3 + 2 = 9$

$c(0, 1) = c(0, 0) + c(1, 1) + w(0, 1) = 0 + 0 + 9 = 9$

$r(0, 1) = 1$

$w(1, 2) = p_2 + q_2 + w(1, 1) = 2 + 1 + 3 = 6$

$c(1, 2) = c(1, 1) + c(2, 2) + w(1, 2) = 0 + 0 + 6 = 6$

$r(1, 2) = 2$

$w(2, 3) = p_3 + q_3 + w(2, 2) = 1 + 5 + 1 = 7$

$c(2, 3) = c(2, 2) + c(3, 3) + w(2, 3) = 0 + 0 + 7 = 7$

$r(2, 3) = 3$

$w(0, 2) = p_2 + q_2 + w(0, 1) = 2 + 1 + 9 = 12$

$c(0, 2) = \min \{c(0, 0) + c(1, 2), c(0, 1) + c(2, 2)\} + w(0, 2)$

$= \min \{0 + 6, 9 + 0\} + 12 = 6 + 12 = 18$

$r(0, 2) = 1$

$w(1, 3) = p_3 + q_3 + w(1, 2) = 1 + 5 + 6 = \mathbf{12}$

$c(1, 3) = \min \{c(1, 1) + c(2, 3), c(1, 2) + c(3, 3)\} + w(1, 3)$

$= \min \{0 + 7, 6 + 0\} + 12 = 6 + 12 = \mathbf{18}$

$r(1, 3) = \mathbf{3}$

$w(0, 3) = p_3 + q_3 + w(0, 2) = 1 + 5 + 12 = \mathbf{18}$

$c(0, 3) = \min \left\{ \begin{matrix} c(0, 0) + c(1, 3), \\ c(0, 1) + c(2, 3), \\ c(0, 2) + c(3, 3) \end{matrix} \right\} + w(0, 3)$

$= \min \{0 + 18, 9 + 7, 18 + 0\} + 18$

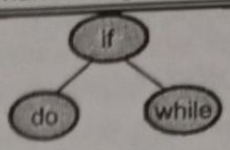
$= \min \{18, 16, 18\} + 18 = 16 + 18 = \mathbf{34}$

$r(0, 3) = \mathbf{2}$

i ↓	j → 0	1	2	3
0	$w_{00} = 2$ $c_{00} = 0$ $r_{00} = 0$	$w_{11} = 3$ $c_{11} = 0$ $r_{11} = 0$	$w_{22} = 1$ $c_{22} = 0$ $r_{22} = 0$	$w_{33} = 5$ $c_{33} = 0$ $r_{33} = 0$
1	$w_{01} = 9$ $c_{01} = 9$ $r_{01} = 1$	$w_{12} = 6$ $c_{12} = 6$ $r_{12} = 2$	$w_{23} = 7$ $c_{23} = 7$ $r_{23} = 3$	
2	$w_{02} = 12$ $c_{02} = 18$ $r_{02} = 1$	$w_{13} = 12$ $c_{13} = 18$ $r_{13} = 3$		
3	$w_{03} = 18$ $c_{03} = 34$ $r_{03} = 2$	Here each cell in row i and column j represents the values w(j, j + i), c(j, j + i) and r(j, j + i)		

Fig. Ex. 4.3.4 : Computation of c(0, 3), w(0, 3) and r(0, 3)

- From the computations table in Fig. Ex. 4.3.4 we have **c(0, 3) = 1.5** as the minimum search cost and **r(0,3) = 2** as the root of an OBST $T_{0,3}$ for given 3 keys.
- The root of a tree $T_{0,3}$ is a_2 giving a left subtree $T_L = T_{0,1}$ and a right subtree $T_R = T_{2,3}$.
- The root of a tree $T_{0,1}$ is r(0,1) = 1, so its root is a_1 giving a left tree $T_L = T_{0,0}$ and a right subtree $T_R = T_{1,1}$.
- The root of a tree $T_{2,3}$ is r(2,3)=3, so its root is a_3 giving a left subtree $T_L = T_{2,2}$ and a right subtree $T_R = T_{3,3}$.
- So, the final OBST $T_{0,3}$ is depicted as below :



(1D18)**Fig. Ex. 4.3.4(a) : OBST $T_{0,3}$**

- **The expected minimum search cost of c(0, 3) = 34 units.**

Ex. 4.3.5 : Compute and construct OBST for the given values using dynamic programming. Let N = 4 and (a1, a2, a3, a4) = (Japan, Korea, Nepal, Singapore).
Let p(1:4) = (1/5, 1/20, 1/10, 1/4) and q(0:4) = (1/20, 1/20, 1/5,1/20, 1/20). **(8 Marks)**

☑ **Soln. :**

- For the simplified calculations we multiply all probabilities by 20 so that we get,
  p (1:4) = (4, 1, 2, 5) , and q (0:4) = (1, 1, 4, 1, 1)
- To solve a given instance of the OBST by dynamic programming we use following recursive formulae to calculate c(i, j), w(i, j) and r(i, j) for an optimal subtree $T_{i,j}$, $0 \le i \le j \le n$.

$$\mathbf{c(i, j) = \min_{i < k \le j} \{c(i, k - 1) + c(k, j)\} + w(i, j)}$$

Expected search, cost of left, subtree T_L | **Expected search, cost of right, subtree T_R** | **sum of all, probabilities**

$$\mathbf{c(i, i) = 0;\ w(i, i) = q_i;\ r(i, i) = 0;}$$

$$\mathbf{w(i, j) = p_j + q_j + w(i, j - 1),\ 0 \le i \le j \le n}$$

We recursively compute w(0, 4), c(0, 4) and r(0, 4) and represent all values in a tabular form as depicted in Fig. Ex. 4.3.5.

$w(0, 0) = q_0 = \mathbf{1};\quad c(0, 0) = \mathbf{0};\quad r(0, 0) = \mathbf{0}$

$w(1, 1) = q_1 = \mathbf{1};\quad c(1, 1) = \mathbf{0};\quad r(1, 1) = \mathbf{0}$

$w(2, 2) = q_2 = \mathbf{4};\quad c(2, 2) = \mathbf{0};\quad r(2, 2) = \mathbf{0}$

$w(3, 3) = q_3 = \mathbf{1};\quad c(3, 3) = \mathbf{0};\quad r(3, 3) = \mathbf{0}$

$w(4, 4) = q_4 = \mathbf{1},\quad c(4, 4) = \mathbf{0},\quad r(4, 4) = \mathbf{0}$

$w(0, 1) = p_1 + q_1 + w(0, 0) = 4 + 1 + 1 = \mathbf{6}$

$c(0, 1) = c(0, 0) + c(1, 1) + w(0, 1) = 0 + 0 + 6 = \mathbf{6}$

$r(0, 1) = \mathbf{1}$

$w(1, 2) = p_2 + q_2 + w(1, 1) = 1 + 4 + 1 = \mathbf{6}$

$c(1, 2) = c(1, 1) + c(2, 2) + w(1, 2) = 0 + 0 + 6 = \mathbf{6}$

$r(1, 2) = \mathbf{2}$

$w(2, 3) = p_3 + q_3 + w(2, 2) = 2 + 1 + 4 = 7$

$c(2, 3) = c(2, 2) + c(3, 3) + w(2, 3) = 0 + 0 + 7 = 7$

$r(2, 3) = 3$

$w(3, 4) = p_4 + q_4 + w(3, 3) = 5 + 1 + 1 = 7$

$c(3, 4) = c(3, 3) + c(4, 4) + w(3, 4) = 0 + 0 + 7 = 7$

$r(3, 4) = 4$

$w(0, 2) = p_2 + q_2 + w(0, 1) = 1 + 4 + 6 = \mathbf{11}$

$c(0, 2) = \min \{c(0, 0) + c(1, 2), c(0, 1) + c(2, 2)\} + w(0, 2)$

$= \min\{0 + 6, 6 + 0\} + 11 = 6 + 11 = \mathbf{17}$

$r(0, 2) = \mathbf{1 \text{ or } 2}$

$w(1, 3) = p_3 + q_3 + w(1, 2) = 2 + 1 + 6 = \mathbf{9}$

$c(1, 3) = \min \{c(1, 1) + c(2, 3), c(1, 2) + c(3, 3)\} + w(1, 3)$

$= \min \{0 + 7, 6 + 0\} + 9 = 6 + 9 = \mathbf{15}$

$r(1, 3) = \mathbf{3}$

$w(2, 4) = p_4 + q_4 + w(2, 3) = 5 + 1 + 7 = \mathbf{13}$

$c(2, 4) = \min \{c(2, 2) + c(3, 4), c(2, 3) + c(4, 4) + w(2, 4)\}$

$= \min \{0 + 7, 7 + 0\} + 13 = 7 + 13 = \mathbf{20}$

$r(2, 4) = \mathbf{3 \text{ or } 4}$

$w(0, 3) = p_3 + q_3 + w(0, 2) = 2 + 1 + 11 = \mathbf{14}$

$$c(0, 3) = \min \begin{Bmatrix} c(0, 0) + c(1, 3), \\ c(0, 1) + c(2, 3), \\ c(0, 2) + c(3, 3) \end{Bmatrix} + w(0, 3)$$

$= \min \{0 + 15, 6 + 7, 11 + 0\} + 14$

$= \min \{15, 13, 11\} + 14 = 11 + 14 = \mathbf{25}$

$r(0, 3) = \mathbf{3}$

$w(1, 4) = p_4 + q_4 + w(1, 3) = 5 + 1 + 9 = \mathbf{15}$

$$c(1, 4) = \min \begin{Bmatrix} c(1, 1) + c(2, 4), \\ c(1, 2) + c(3, 4), \\ c(1, 3) + c(4, 4) \end{Bmatrix} + w(1, 4)$$

$= \min \{0 + 20, 6 + 7, 15 + 0\} + 15$

$= \min \{20, 13, 15\} + 15 = 13 + 15 = \mathbf{28}$

$r(1, 4) = \mathbf{3}$

$w(0, 4) = p_4 + q_4 + w(0, 3) = 5 + 1 + 14 = \mathbf{20}$

$$c(0, 4) = \min \begin{Bmatrix} c(0, 0) + c(1, 4), \\ c(0, 1) + c(2, 4), \\ c(0, 2) + c(3, 4), \\ c(0, 3) + c(4, 4) \end{Bmatrix} + w(0, 4)$$

$= \min \{0 + 28, 6 + 20, 17 + 7, 25 + 0\} + 20$

$= \min \{28, 26, 24, 25\} + 20 = 24 + 20 = \mathbf{44}$

$r(0, 4) = \mathbf{3}$

- From the computation table in Fig. Ex. 4.3.5 we have **c(0, 4) = 44** as the minimum search cost and **r(0,4)=3** as the root of an OBST $T_{0,4}$ for given 4 keys.
- The root of a tree $T_{0,4}$ is a_3 giving a left subtree $T_L = T_{0,2}$ and a right subtree $T_R = T_{3,4}$.
- The root of a tree $T_{0,2}$ is r(0,2)=1 or 2, so its root is a_1 or a_2.

i ↓ / j →	0	1	2	3	4
0	$w_{00} = 1$ $c_{00} = 0$ $r_{00} = 0$	$w_{11} = 1$ $c_{11} = 0$ $r_{11} = 0$	$w_{22} = 4$ $c_{22} = 0$ $r_{22} = 0$	$w_{33} = 1$ $c_{33} = 0$ $r_{33} = 0$	$w_{44} = 1$ $c_{44} = 0$ $r_{44} = 0$
1	$w_{01} = 6$ $c_{01} = 6$ $r_{01} = 1$	$w_{12} = 6$ $c_{12} = 6$ $r_{12} = 2$	$w_{23} = 7$ $c_{23} = 7$ $r_{23} = 3$	$w_{34} = 7$ $c_{34} = 7$ $r_{34} = 4$	
2	$w_{02} = 11$ $c_{02} = 17$ $r_{02} = 1 \text{ or } 2$	$w_{13} = 9$ $c_{13} = 15$ $r_{13} = 3$	$w_{24} = 13$ $c_{24} = 20$ $r_{24} = 3 \text{ or } 4$		
3	$w_{03} = 14$ $c_{03} = 25$ $r_{03} = 3$	$w_{14} = 15$ $c_{14} = 28$ $r_{14} = 3$			
4	$w_{04} = 20$ $c_{04} = 44$ $r_{04} = 3$	Here each cell in row i and column j represents w(j, j + i), c(j, j + i) and r(j, j + i)			

Fig. Ex. 4.3.5 : Computation of w(0, 4), c(0, 4) and r(0, 4)

- If a_1 is the root of $T_{0,2}$ then we get giving a left subtree $T_L = T_{0,0}$ and a right subtree $T_R = T_{1,2}$. The root of $T_{1,2}$ is r(1,2) = 2, so its root is a_2 giving a left subtree $T_L = T_{1,1}$ and right subtree as $T_R = T_{2,2}$.
- If a_2 is the root of $T_{0,2}$ then we get giving a left subtree $T_L = T_{0,1}$ and a right subtree $T_R = T_{2,2}$. The root of $T_{0,1}$ is r(0,1) = 1, so its root is a_1 giving a left subtree $T_L = T_{0,0}$ and right subtree as $T_R = T_{1,1}$.

- The root of a tree $T_{3,4}$ is $r(3,4) = 4$, so its root is a_4 giving a left subtree $T_{3,3}$ and a right subtree $T_{4,4}$.
- So the final OBST $T_{0,4}$ is depicted as below :

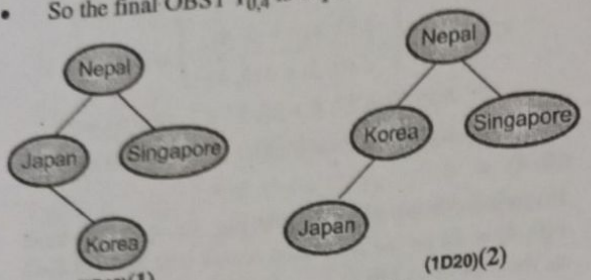


Fig. Ex. 4.3.5(a) : OBST $T_{0,4}$

- **The expected minimum search cost of c(0, 4) = 44 units.**

4.4 0/1 KNAPSACK PROBLEM

GQ. Discuss and derive an equation for solving the 0/1 Knapsack problem using the dynamic programming method. **(4 Marks)**

GQ. Explain 0/1 knapsack using a suitable example. **(7 Marks)**

- The knapsack problem is one of the classic problems in combinatorial optimization that can be solved by different algorithmic strategies.
- There are two variants of this problem: (1) 0/1 the knapsack problem and (2) the Fractional knapsack problem.
- Dynamic programming is used to solve the 0/1 knapsack problem.
- The dynamic programming approach enumerates a sequence of decisions on the inclusion or the exclusion of each item to get an optimal solution.

Problem description

- Suppose, n items are provided to fill a knapsack of capacity M. Each item i, $1 \le i \le n$, has weight w_i and it gives profit $p_i \cdot x_i$ if its part x_i, $x_i = 0$ or 1 is kept into the knapsack.
- In the 0/1 knapsack problem, the item can either be added as a whole ($x_i = 1$) or rejected ($x_i = 0$) depending on the available capacity of a given knapsack.
- The objective of this problem is to earn the maximum profit by filling a knapsack with the given items by considering a constraint of knapsack capacity.
- The problem can be mathematically given as:

$$\text{Maximize} \quad \sum_{i=1}^{n} p_i x_i$$

$$\text{Subject to} \quad \sum_{i=1}^{n} w_i x_i \le M$$

and $x_i = 0$ or 1, $1 \le i \le n$.

all the values of profits and weights are positive real numbers.

The general procedure

- If the i^{th} item with weight w_i is selected ($x_i = 1$) then the profit earned increases by p_i and the available capacity of the knapsack decreases by w_i.
- However, if the i^{th} item is rejected ($x_i = 0$) then the associated profit p_i is not earned and there is no change in the available capacity of a knapsack.
- Suppose, decisions on x_i are made in the sequence
- x_n, x_{n-1}, ……x_1. If a decision sequence x_n, x_{n-1}, ……, x_1 is optimal then the sequence of decisions
- x_{n-1}, ……., x_1 must be optimal with respect to the resultant problem state of a decision on x_n. This satisfies the principle of optimality.
- Thus, the optimal profit earned by deciding on x_n w.r.t. knapsack capacity M can be formulated as :

$$V_n(M) = \max \{V_{n-1}(M), V_{n-1}(M - w_n) + p_n\}$$

The exclusion of an n^{th} item, ($x_n = 0$) does not earn the profit p_n and does, not change the capacity M.

The inclusion of an n^{th} item ($x_n = 1$) increases the profit by p_n and decreases the knapsack capacity by w_n.

- This recursive formula can be generalized as

$$V_i(j) = \max\{V_{i-1}(j), V_{i-1}(j - w_i) + p_i\}$$

optimal profit earned by making decision, on x_i w.r.t. available, capacity j.

by excluding an i^{th} item ($x_i = 0$).

by including an i^{th} item ($x_i = 1$).

- The values of V_1, V_2, ……, V_n are successively calculated. $V_i(j) = -\infty$ if $j < 0$ and $V_0(j) = 0$ for all $j \ge 0$.

- The function $V_i(j)$ is an increasing step function. Thus, there exists a finite number of j's, $0 = j_1 < j_2 < \ldots < j_k$ such that

$V_i(j_1) < V_i(j_2) < \ldots < V_i(j_k)$;

$V_i(j) = -\infty$ if $j < j_1$;

$V_i(j) = V_i(j_k)$ if $j \geq j_k$;

$V_i(j) = V_i(j_h)$ if $j_h \leq j \leq j_{h+1}$

- Hence, we need to calculate only $V_i(j_h)$ where, $1 \leq h \leq k$, $1 \leq i \leq n$.
- There are two methods to represent the computation of $V_i(j)$:
  (1) Using sets representation
  (2) Using tabular representation

Note : The examples can be solved either by set representation or by tabular representation.

4.4.1 Computations using Set Representation

- Here, $V_i(j)$ is represented as the ordered set

$$S^i = \left\{ (V_i(j), j) \;\middle|\; \begin{matrix} 0 \leq i \leq n,\ 0 \leq j \leq M \\ \text{n = number of items, M = knapsack capacity} \end{matrix} \right\}$$

- Each member in the set S^i is a pair(P, W) where
- $P = V_i(j)$ and $W = j$

$$S^0 = \{(0, 0)\}$$

$$S_1^i = \left\{ (P, W) \;\middle|\; (P - p_{i+1}, W - w_{i+1}) \in S^i \right\}$$

By merging pairs in S^i and S_1^i we get S^{i+1}

i.e. $S^{i+1} = S^i \cup S_1^i$

Purging rule or Dominance rule

- If S^{i+1} has two pairs (P_a, W_a) and (P_b, W_b) such that
- $P_a \leq P_b$ and $W_a \geq W_b$ then the pair(P_a, W_a) can be discarded. Here, due to the dominance of (P_b, W_b) the dominated pair (P_a, W_a) gets purged.

Tracing through all S^i's to get a decision sequence on x_i's, $1 \leq i \leq n$

- Suppose (P', W') is the last tuple in S^n and $P' = \sum p_i x_i$ and $W' = \sum w_i x_i$, then the 0 or 1 values for decisions on x_i's is obtained by searching through S^i's.
- Set $x_n = 0$; if $(P', W') \in S^{n-1}$ ....(No change in profit and weight indicates the exclusion of an n^{th} item)
- Set $x_n = 1$; if $(P', W') \notin S^{n-1}$ and $(P' - p_n, W' - w_n) \in S^{n-1}$ ....(Increase in profit by p_n and weight by w_n indicates the inclusion of an n^{th} item)
- Thus we can get a decision sequence on x_i's by recursively tracing how either (P', w') or $(P' - p_n, W' - w_n)$ was obtained in a set S^{n-1}.

Control abstraction for set representation

```
Algorithm Knapsack_DP1(p, w, n, M)
/*Input : p=(p1,...pn) is a set of profits, and w=(w1,...wn) is
a set of weights associated with n items. M is the capacity of a
given knapsack.
Output: Optimal profit earned by filling a knapsack with n
items.*/
{
   S^0 := {(0, 0)};
   for (i:=1; i < n; i++)
   {
   S_1^{i-1} := { (P, W) | (P - p_{i+1}, W - w_{i+1}) ∈ S^{i-1} && W ≤ M };
      S^i := Union_with_Purge(S^{i-1}, S_1^{i-1});
                  /*Union_with_Purge () merges sets S^{i-1}
                  and S_1^{i-1} to get S^i. It also applies purging
                  rule while merging of two sets.*/
   }
   (P'',W'') := last pair in S^{n-1};
   (P', W') := (P_L + p_n, W_L + w_n) where W_L is the
               greatest value of W in any pair in S^{n-1} such
               that W_L + w_n ≤ M;
   if (P'>P'')
      x_n := 1;
   else  x_n := 0;
      Trace_Seq(x_{n-1}, ....., x_1);
                  /* Trace_Seq() traces all S^i's
                  for a decision sequence on x_i's.*/
}
```

Complexity analysis

- The time required for computing S^i from S^{i-1} is
- $\Theta(|S^{i-1}|)$. Thus the total time to calculate all the S^i's,

$0 \le i < n$ is $\Theta\left(\sum_{i=1}^{n} |S^{i-1}|\right)$.

- For each item, we decide on either to include it or to exclude it. Thus, $|S^i| \le 2^i$. Hence the time for calculation of all S^i's, $0 \le i < n$ is $O(2^n)$.
- The time needed to trace all S^i's for a decision sequence on x_i's is $O(n^2)$.
- When the profits P_i's and weights W_i's of all n items are integers then we get,

$$|S^i| \le 1 + \sum_{j=1}^{i} p_j \text{ and } |S^i| \le 1 + \min\left\{M, \sum_{j=1}^{i} w_j\right\}$$

- Hence the time complexity of calculating all S^i's is

$$O\left(\min\left\{2^n, n\sum_{i=1}^{n} p_i, nM\right\}\right)$$

- Replacing $\sum_{i=1}^{n} p_i$ by $\sum_{i=1}^{n} p_i / \text{gcd}\ (p_1, p_2, \ldots\ldots, p_n)$

And replacing M by

$$1 + \min\left\{M, \sum_{j=1}^{i} w_j\right\} / \text{gcd}\ (w_1, w_2, \ldots\ldots, w_n, M)$$

we get the bound on the time needed to calculate all S^i's as below :

$$O\left(\min\left\{2^n, n\sum_{i=1}^{n} p_i / \text{gcd}\ (p_1, p_2, \ldots\ldots, p_n),\ 1 + \min\left\{M, \sum_{j=1}^{i} w_j\right\} / \text{gcd}\ (w_1, w_2, \ldots, w_n, M)\right\}\right)$$

Ex. 4.4.1 : Consider Knapsack capacity W = 9, w = (3, 4, 5, 7) and v = (12, 40, 25, 42) find the maximum profit using dynamic method. **(8 Marks)**

Soln. :

To solve the given instance of the 0/1 knapsack problem by dynamic programming we use the following formulae :

$$V_i(j) = \max\{V_{i-1}(j), V_{i-1}(j - w_i) + p_i\}$$

- $V_i(j)$: optimal profit earned by making a decision, on x_i w.r.t. available capacity j.
- $V_{i-1}(j)$: by excluding an i^{th} item ($x_i = 0$).
- $V_{i-1}(j - w_i) + p_i$: by including an i^{th} item ($x_i = 1$).

Using set representation we have,

$S^i = \{(P, W)\}$ where $P = V_i(j)$ and $W = j$, $0 \le i \le n$, $0 \le j \le M$, n = the number of items and M = capacity of a knapsack.

$$S^0 = \{(0, 0)\}$$

$$S_1^i = \{(P, W) \mid (P - p_{i+1}, W - w_{i+1}) \in S^i\}$$

$$S^{i+1} = S^i \cup S_1^i$$

Given : values P = v(1:4) = {12, 40, 25, 42}, w(1:4) = {3, 4, 5, 7}, M = W = 3.

- Using above formulae we do following computations :

$S^0 = \{(0, 0)\}$

$S_1^0 = \{0 + 12, 0 + 3\} = \{(12, 3)\}$

$S^1 = S^0 \cup S_1^0 = \{(0, 0), (12, 3)\}$

$S_1^1 = \{(0 + 40, 0 + 4), (12 + 40, 3 + 4)\}$

$= \{(40, 4), (52, 7)\}$

$S^2 = S^1 \cup S_1^1 = \{(0, 0), (12, 3), (40, 4), (52, 7)\}$

$S_1^2 = \{(0 + 25, 0 + 5), (12 + 25, 3 + 5), (40 + 25, 4 + 5), (52 + 25, 7 + 5)\}$

$\therefore S_1^2 = \{(25, 5), (37, 8), (65, 9), (77,12)\}$

$S^3 = S^2 \cup S_1^2 = \{(0, 0), (12, 3), \mathbf{(25, 5)}, \mathbf{(37, 8)}, (40, 4), (52, 7), (65, 9), \mathbf{(77,12)}\}$

...(77,12) is not feasible as it exceeds the knapsack capacity.

Also, (25, 5) and (37, 8) are purged due to dominance of (40, 4).

$\therefore S^3 = \{(0, 0), (12, 3), (40, 4), (52, 7), (65, 9)\}$

$S_1^3 = \{(0 + 42, 0 + 7), (12 + 42, 3 + 7), (40 + 42, 4 + 7), (52 + 42, 7 + 7), (65 + 42, 9 + 7)\}$

$\therefore S_1^3 = \{(42, 7), (54, 10), (82, 11), (94, 14), (107, 16)\}$

$S^4 = S^3 \cup S_1^3 = \{(0, 0), (12, 3), (40, 4), \mathbf{(42, 7)}, (52, 7), \mathbf{(54, 10)}, (65, 9), \mathbf{(82, 11)}, \mathbf{(94, 14)}, \mathbf{107, 16)}\}$

...(54,10), (82,11), (94,14), (107,16) are not feasible as they exceed the knapsack capacity.

Also, (42, 7) is purged due to dominance of (52, 7);

$\therefore S^4 = \{(0, 0), (12, 3), (40, 4), (52, 7), (65, 9)\}$

- Now (65, 9) is a pair with the maximum profit and the largest weight. So, the maximum profit earned is 65 units with the weight of 9 units.

Here,

$(65, 9) \in S^4, (65, 9) \in S^3$

$\therefore x_4 = 0;$

$(65, 9) \in S^3$ and $(65 - 25, 9 - 5) = (40, 4) \in S^2$

$\therefore x_3 = 1;$

$(40, 4) \in S^2$ and $(40 - 40, 4 - 4) = (0, 0) \in S^1$

and $(0, 0) \in S^0$

$\therefore x_2 = 1; x_1 = 0.$

- **Thus, the maximum profit of 65 units is earned by selecting items 2, 3.**

$\therefore$ **x (1 : 4) = (0, 1, 1, 0)**

Ex. 4.4.2 : Solve the following Knapsack Problem using the greedy method. Number of items = 5, knapsack capacity W = 100, weight vector = {50, 40, 30, 20,10} and profit vector = {1, 2, 3, 4, 5}. **(8 Marks)**

☑ **Soln. :**

To solve the given instance of the 0/1 knapsack problem by dynamic programming we use the following formulae :

$$V_i(j) = \max \{V_{i-1}(j), V_{i-1}(j - w_i) + p_i\}$$

- $V_i(j)$: optimal profit earned by making decision, on x_i w.r.t. available capacity j.
- $V_{i-1}(j)$: by excluding an i^{th} item ($x_i = 0$).
- $V_{i-1}(j - w_i) + p_i$: by including an i^{th} item ($x_i = 1$).

- Using set representation we have,

$S^i = \{(P, W)\}$ where $P = V_i(j)$ and $W = j$, $0 \le i \le n$, $0 \le j \le M$, n = the number of items and M = capacity of a knapsack.

$S^0 = \{(0, 0)\}$

$S_1^i = \{(P, W) \mid (P - p_{i+1}, W - w_{i+1}) \in S^i\}$

$S^{i+1} = S^i \cup S_1^i$

Given : $P(1:5) = \{1, 2, 3, 4, 5\}$,

$W(1:5) = 50, 40, 30, 20, 10\}$, $M = 100$.

- Using above formulae we do following computations :

$S^0 = \{(0, 0)\}$

$S_1^0 = \{0 + 1, 0 + 50\} = \{(1, 50)\}$

$S^1 = S^0 \cup S_1^0 = \{(0, 0), (1, 50)\}$

$S_1^1 = \{(0 + 2, 0 + 40), (1 + 2, 50 + 40)\}$

$= \{(2, 40), (3, 90)\}$

$S^2 = S^1 \cup S_1^1 = \{(0, 0), (1, 50), (2, 40), (3, 90)\}$

...(1, 50) is purged due to dominance of (2, 40).

$\therefore S^2 = \{(0, 0), (2, 40), (3, 90)\}$

$S_1^2 = \{(0+3, 0+30), (2+3, 40+30), (3+3, 90+30)\}$

$\therefore S_1^2 = \{(3, 30), (5, 70), (6, 120)\}$

$S^3 = S^2 \cup S_1^2 = \{(0, 0), (2, 40), (3, 90), (3, 30), (5, 70), (6, 120)\}$

...(6, 120) is not feasible as it exceeds the knapsack capacity.

Also, (2, 40) and (3, 90) are purged due to dominance of (3, 30).

$\therefore S^3 = \{(0, 0), (3, 30), (5, 70)\}$

$S_1^3 = \{(0 + 4, 0 + 20), (3 + 4, 30 + 20), (5 + 4, 70 + 20)\}$

$\therefore S_1^3 = \{(4, 20), (7, 50), (9, 90)\}$

$\therefore S^4 = S^3 \cup S_1^3 = \{(0, 0), (3, 30), (4, 20), (5, 70), (7, 50), (9, 90)\}$

... (3, 30) is purged due to dominance of (4, 20).

Also, (5, 70) is purged due to dominance of (7, 50).

$\therefore S^4 = \{(0, 0), (4, 20), (7, 50), (9, 90)\}$

$\therefore S_1^4 = \{(0 + 5, 0 + 10), (4 + 5, 20 + 10), (7 + 5, 50 + 10), (9 + 5, 90 + 10)\}$

$\therefore S_1^4 = \{(5, 10), (9, 30), (12, 60), (14, 100)\}$

$S^5 = S^4 \cup S_1^4 = \{(0, 0), (4, 20), (5, 10), (7, 50), (9, 90), (9, 30), (12, 60), (14, 100)\}$

...(4, 20) is purged due to dominance of (5, 10).

Also, (9, 90) is purged due to the dominance of (9, 30).

$\therefore S^5 = \{(0, 0), (5, 10), (7, 50), (9, 30), (12, 60), (14, 100)\}$

- Now (14, 100) is a pair with the maximum profit and the largest weight. So, the maximum profit is 14 units with a weight of 100 units.

Here,

$(14, 100) \in S^5$ and $(14 - 5, 100 - 10) = (9, 90) \in S^4$

$\therefore x_5 = 1$;

$(9, 90) \in S^4$ and $(9 - 4, 90 - 20) = (5, 70) \in S^3$

$\therefore x_4 = 1$;

$(5, 70) \in S^3$ and $(5 - 3, 70 - 30) = (2, 40) \in S^2$

$\therefore x_3 = 1$;

$(2, 40) \in S^2$ and $(2 - 2, 40 - 40) \in S^0$

$\therefore x_2 = 1$ and $x_1 = 0$;

- **Thus, the maximum profit of 14 units with a total weight of 100 units is earned by selecting items (2, 3, 4, 5)**

  $\therefore$ x(1 : 5) = (0, 1, 1, 1, 1)

Ex. 4.4.3 : Discuss the knapsack problem using dynamic programming. Solve the following knapsack problem using dynamic programming. There are three objects, whose weights w(w1, w2, w3) = {1, 2, 3} and values v(v1,v2,v3) = {2, 3, 4} are given. The knapsack capacity M is 3 units. **(8 Marks)**

☑ **Soln. :**

▶ **The knapsack problem by dynamic programming:**

Refer to problem description under section 4.5.

▶ **The solution to a given instance of knapsack problem by dynamic programming:**

- To solve the given instance of the 0/1 knapsack problem by dynamic programming we use the following formulae :

$$V_i(j) = \max \{V_{i-1}(j), V_{i-1}(j - w_i) + p_i\}$$

- $V_i(j)$: optimal profit earned by making decision, on x_i w.r.t. available capacity j.
- $V_{i-1}(j)$: by excluding an i^{th} item ($x_i = 0$).
- $V_{i-1}(j - w_i) + p_i$: by including an i^{th} item ($x_i = 1$).

- Using set representation we have,

S^i = {(P, W)} where P = $V_i(j)$ and W = j,

$0 \le i \le n, 0 \le j \le M$, n = the number of items and M = capacity of a knapsack.

$S^0 = \{(0, 0)\}$

$S_1^i = \{(P, W) \mid (P - p_{i+1}, W - w_{i+1}) \in S^i\}$

$S^{i+1} = S^i \cup S_1^i$

Given : w(w1, w2, w3) = {1, 2, 3},

values P = v(v1,v2,v3) = {2, 3, 4}, M = 3.

- Using above formulae we do following computations :

$S^0 = \{(0, 0)\}$

$S_1^0 = \{0 + 2, 0 + 1\} = \{(2, 1)\}$

$S^1 = S^0 \cup S_1^0 = \{(0, 0), (2, 1)\}$

$S_1^1 = \{(0 + 3, 0 + 2), (2 + 3, 1 + 2)\}$

$= \{(3, 2), (5, 3)\}$

$S^2 = S^1 \cup S_1^1 = \{(0, 0), (2, 1), (3, 2), (5, 3)\}$

$S_1^2 = \{(0 + 4, 0 + 3), (2 + 4, 1 + 3), (3 + 4, 2 + 3), (5 + 4, 3 + 3)\}$

$\therefore S_1^2 = \{(4, 3), (6, 4), (7, 5), (9,6)\}$

$S^3 = S^2 \cup S_1^2$

$= \{(0, 0), (2, 1), \mathbf{(3, 2)}, \mathbf{(4, 3)}, (5, 3), (6, 4), (7, 5), \mathbf{(9, 6)}\}$

...(6, 4), (7, 5), (9, 6) are not feasible as they exceed the knapsack capacity.

Also, (4, 3) is purged due to dominance of (5, 3).

$\therefore S^3 = \{(0, 0), (2, 1), (3, 2), (5, 3)\}$

- Now (5, 3) is a pair with the maximum profit and the largest weight. So, the maximum profit earned is 5 units with a weight of 3 units.

Here,

$(5, 3) \in S^3, (5, 3) \in S^2$

$\therefore x_3 = 0$;

$(5, 3) \in S^2$ and $(5 - 3, 3 - 2) = (2, 1) \in S^1 \therefore x_2 = 1$;

$(2, 1) \in S^1$ and $(2 - 2, 1 - 1) = (0, 0) \in S^0 \therefore x_1 = 1$.

- **Thus, the maximum profit of 5 units is earned by selecting items 1, 2.**

  $\therefore$ x (1 : 3) = (1, 1, 0)

Ex. 4.4.4 : Consider the instance of the 0/1 (binary) knapsack problem as below with P depicting the value and W depicting the weight of each item whereas M denotes the total weight carrying capacity of the knapsack. Find the optimal answer using dynamic programming. P = [40 10 50 30 60] W = [80 10 40 20 90] M = 110. **(7 Marks)**

☑ **Soln. :**

To solve the given instance of the 0/1 knapsack problem by dynamic programming we use the following formulae :

$V_i(j) = \max \{V_{i-1}(j), V_{i-1}(j - w_i) + p_i\}$

- optimal profit earned by deciding on x_i w.r.t. available capacity j.
- by excluding an i^{th} item ($x_i = 0$).
- by including an i^{th} item ($x_i = 1$).

- Using set representation we have,

$S^i = \{(P, W)\}$ where $P = V_i(j)$ and $W = j$, $0 \le i \le n$, $0 \le j \le M$, n = the number of items and M = capacity of a knapsack.

$S^0 = \{(0, 0)\}$

$S_1^i = \{(P, W) \mid (P - p_{i+1}, W - w_{i+1}) \in S^i\}$

$S^{i+1} = S^i \cup S_1^i$

Given :

$P(1:5) = \{40, 10, 50, 30, 60\}$, $W(1:5)$

$= \{80, 10, 40, 20, 90\}$, $M = 110$.

- Using above formulae we do following computations :

$S^0 = \{(0, 0)\}$

$S_1^0 = \{0 + 40, 0 + 80\} = \{(40, 80)\}$

$S^1 = S^0 \cup S_1^0 = \{(0, 0), (40, 80)\}$

$S_1^1 = \{(0 + 10, 0 + 10), (40 + 10, 80 + 10)\}$

$= \{(10, 10), (50, 90)\}$

$S^2 = S^1 \cup S_1^1 = \{(0, 0), (10, 10), (40, 80), (50, 90)\}$

$S_1^2 = \{(0 + 50, 0 + 40), (100 + 50, 10 + 40), (40 + 50, 80 + 40), (50 + 50, 90 + 40)\}$

$\therefore S_1^2 = \{(50, 40), (60, 50), (90, 120), (100,130)\}$

$S^3 = S^2 \cup S_1^2 = \{(0, 0), (10, 10), \mathbf{(40, 80)}, \mathbf{(50, 90)}, (50, 40), (60, 50), \mathbf{(90, 120)}, \mathbf{(100,130)}\}$

...(90, 120), (100,130) are not feasible as they exceed the knapsack capacity.

Also, (40, 80), (50, 90) are purged due to dominance of (50, 40).

$\therefore S^3 = \{(0, 0), (10, 10), (50, 40), (60, 50)\}$

$S_1^3 = \{(0 + 30, 0 + 20), (10 + 30, 10 + 20), (50 + 30, 40 + 20), (60 + 30, 50 + 20)\}$

$\therefore S_1^3 = \{(30, 20), (40, 30), (80, 60), (90,70)\}$

$\therefore S^4 = S^3 \cup S_1^3 = \{(0, 0), (10, 10), (30, 20), (40, 30), (50, 40), (60, 50), (80, 60), (90,70)\}$

$\therefore S_1^4 = \{(0 + 60, 0 + 90), (10 + 60, 10 + 90), (30 + 60, 20 + 90), (40 + 60, 30 + 90), (50 + 60, 40 + 90), (60 + 60, 50 + 90), (80 + 60, 60 + 90), (90 + 60, 70 + 90)\}$

$\therefore S_1^4 = \{(60, 90), (70, 100), (90, 110), (100, 120), (110, 130), (120, 140), (140, 150), (150,160)\}$

...(100, 120),(110, 130), (120, 140), (140, 150), (150,160) are not feasible as they exceed the knapsack capacity.

$\therefore S_1^4 = \{(60, 90), (70, 100), (90, 110)\}$

$S^5 = S^4 \cup S_1^4$

$\therefore S^5 = \{(0, 0), (10, 10), (30, 20), (40, 30), (50, 40), (60, 50), \mathbf{(60, 90)}, \mathbf{(70, 100)}, (80, 60), (90,70), \mathbf{(90, 110)}\}$

...(60, 90), (70, 100) are purged due to dominance of (80, 60). Also, (90, 110) is purged due to dominance of (90,70).

$\therefore S^5 = \{(0, 0), (10, 10), (30, 20), (40, 30), (50, 40), (60, 50), (80, 60), (90,70)\}$

- Now (90, 70) is a pair with the maximum profit and the largest weight. So, the maximum profit is 90 units with a weight of 70 units.

Here,

$(90, 70) \in S^5$ and $(90, 70) \in S^4$ $\therefore x_5 = 0$;

$(90, 70) \in S^4$ and $(90 - 30, 70 - 20) = (60, 50) \in S^3$

$\therefore x_4 = 1$;

$(60, 50) \in S^3$ and $(60 - 50, 50 - 40) = (10, 10) \in S^2$

$\therefore x_3 = 1$;

$(10, 10) \in S^2$ and $(10 - 10, 10 - 10) \in S^0$

$\therefore x_2 = 1$ and $x_1 = 0$;

- **Thus, the maximum profit of 90 units with total weight of 70 units is earned by selecting items (2, 3, 4)**

$\therefore$ **x(1 : 5) = (0, 1, 1, 1, 0)**

4.4.2 Computations using Tabular Representation

- It is suitable when the given capacity of a knapsack is small.
- It represents the value $V_i(j)$ as a tabular entry V[i, j] where $0 \le i \le n$ and $0 \le j \le M$, n is the number of given items and M is the capacity of a knapsack.

- V[i, j] represents the optimal profit earned by deciding on x_i w.r.t. available capacity j. It is computed by the following recurrence formula:

$$V[i, j] = \begin{cases} \max\{V[i-1, j], V[i-1, j-w_i] + p_i\} & ; \text{if } j - w_i \geq 0 \\ V[i-1, j] & ; \text{if } j - w_i < 0 \end{cases}$$

$V[0, j] = 0$ if $j \geq 0$ and $V[i, 0] = 0$ if $i \geq 0$

$V[i-1, j]$: Exclusion of an i^{th} item

$V[i-1, j-w_i]$: Inclusion of an i^{th} item

- To calculate V[i, j] i.e. the entry in the i^{th} row and the j^{th} column, i, j > 0, we determine the largest value between:
  (1) The entry in the prior row and the same column (i.e. V[i – 1, j]) and
  (2) The sum of p_i and the entry in the prior row and w_i columns to the left (i.e. $V[i-1, j-w_i] + p_i$).
- The entries in the table are filled either row-wise or column-wise.

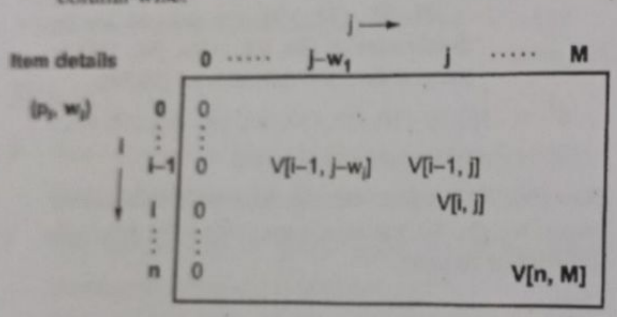


Fig. 4.4.1 : Tabular representation to solve $\frac{0}{1}$ knapsack problem by DP

- **Tracing through V[i, j]' s to get a decision sequence on x_i' s, $1 \leq i \leq n$:**
  - V[n, M] gives the final optimal solution.
  - Set $x_n = 0$ if V[n, M] = V[n – 1, M] ...No changes in the earned profit and weight indicate the exclusion of an n^{th} item.
  - Set $x_n = 1$ if V[n, M] ≠ V[n – 1, M] ...A change in the profit indicates the inclusion of an n^{th} item. In this case the remaining decision sequence on x_i, $1 \leq i < n$ is specified by tracing $V[n-1, M-w_n]$, and so on.

Control abstraction for a tabular representation

```
Algorithm Knapsack_DP2(P, W, n, M)
/* Input: P[1 : n] is a vector of profits and W[1 : n] is a vector
of weights associated with n items. M is the capacity of a
knapsack.
Output: Optimal profit earned by filling a knapsack with the
subset of n items. */
{
 // Initialize the table
  for (j:= 0 ; j ≤ M; j ++)
  V[0, j] : = 0;
  for (i:= 0 ; i ≤ n ; i ++)
     V[i, 0] : = 0;
  for (i:= 1 ; i ≤ n  ; i ++)
  {
     for (j:= 1 ; j ≤ M ; j ++)
     {
        diff:= j – W[i] ;
        if (diff ≥ 0)              {
           V[i, j] := max { V[i – 1, j], V[i – 1, diff] + P[i]};
        }
        else
           V[i, j] := V[i – 1, j] ;
     } // end for j
  } // end for i
  if (V[n, M] > V[n – 1, M])
     x_n := 1;
  else x_n : = 0 ;
     Trace_Seq(x_n-1, ....., x_1);
              /* Trace_Seq() traces all table values for
              a decision sequence on x_i' s.*/
  return V[n, M] ;        /* returns maximum profit earned.*/
}
```

Complexity analysis

- The number of items = n and the capacity of the knapsack = M decide the complexity of the algorithm Knapsack_DP2(P, W, n, M).
- The time to compute all table entries is O(nM) and the time to trace the decision sequence on x_i's that gives an optimal solution is O(n).
- Thus, the time complexity of Knapsack_DP2 becomes O(nM).

Ex. 4.4.5 : Consider 0/1 knapsack problem: N = 3; W = (4, 6, 8) and P = (10, 12, 15). By using dynamic programming determine the optimal profit for the knapsack of capacity 10. **(6 Marks)**

Soln. :

- To solve a given instance of 0/1 knapsack problem by dynamic programming - tabular representation we use following recurrence formulae:

$$V[i,j] = \begin{cases} \max\{V[i-1, j], V[i-1, j-w_i] + p_i\} & ; \text{if } j - w_i \geq 0 \\ V[i-1, j] & ; \text{if } j - w_i < 0 \end{cases}$$

$V[0,j] = 0$ if $j \geq 0$ and $V[i, 0] = 0$ if $i \geq 0$

- All values of V[i, j], $0 \leq i \leq 3$ and $0 \leq j \leq 10$ are computed and represented in a tabular form as below.

Item details (p_i, w_i)	i \ j	0	1	2	3	4	5	6	7	8	9	10
	0	0	0	0	0	0	0	0	0	0	0	0
(10, 4)	1	0	0	0	0	10	10	10	10	10	10	10
(12, 6)	2	0	0	0	0	10	10	12	12	12	12	22
(15, 8)	3	0	0	0	0	10	10	12	12	15	15	**22**

V[1, 1] = V[0, 1] = 0 ...{(j − w_i) = 1 − 4 = − 3 < 0}

V[1, 2] = V[0, 2] = 0 ...{(j − w_i) = 2 − 4 = − 2 < 0}

V[1, 3] = V[0, 3] = 0 ...{(j − w_i) = 3 − 4 = − 1 < 0}

V[1, 4] = max {V[0, 4], 10 + V[0, 0]}
...{j − w_i = 4 − 4 = 0}
= max {0, 10 + 0} = 10

V[1, 5] = max {V[0, 5], 10 + V[0, 1]}
...{j − w_i = 5 − 4 = 1 > 0}
= max {0, 10 + 0} = 10

V[1, 6] = max {V[0, 6], 10 + V[0, 2]}
...{j − w_i = 6 − 4 = 2 > 0}
= max {0, 10 + 0} = 10

V[1, 7] = max {V (0, 7), 10 + V[0, 3]}
...{j − w_i = 7 − 4 = 3 > 0}
= max {0, 10 + 0} = 10

V[1, 8] = max {V[0, 8], 10 + V[0, 4]}
...{j − w_i = 8 − 4 = 4 > 0}
= max {0, 10 + 0} = 10

V[1, 9] = max {V[0, 9] 10 + V[0, 5]}
...{j − w_i = 9 − 5 = 4 > 0}
= max {0, 10 + 0} = 10

V[1, 10] = max {V[0, 10], 10 + V[0, 6]}
...{j − w_i = 10 − 4 = 6 > 0}
= max {0, 10 + 0} = 10

V[2, 1] = V[1, 1] = 0 ...{j − w_i = 1 − 6 = − 5 < 0}

V[2, 2] = V[1, 2] = 0 ...{j − w_i = 2 − 6 = − 4 < 0}

V[2, 3] = V[1, 3] = 0 ...{j − w_i = 3 − 6 = − 3 < 0}

V[2, 4] = V[1, 4] = 10 ...{j − w_i = 4 − 6 = − 2 < 0}

V[2, 5] = V[1, 5] = 10 ...{j − w_i = 5 − 6 = − 1 < 0}

V[2, 6] = max {V[1, 6], 12 + V[1, 0]}
...{j − w_i = 6 − 6 = 0}
= max {10, 12 + 0}
= max {10, 12} = 12

V[2, 7] = max {V[1, 7], 12 + V[1, 1]}
...{j − w_i = 7 − 6 = 1 > 0}
= max {10, 12 + 0} = max {10, 12} = 12

V[2, 8] = max {V[1, 8], 12 + V[1, 2]}
...{j − w_i = 8 − 6 = 2 > 0}
= max {10, 12 + 0} = max {10, 12} = 12

V[2, 9] = max {V[1, 9], 12 + V[1, 3]}
...{j − w_i = 9 − 6 = 3 > 0}
= max {10, 12 + 0}
= max {10, 12} = 12

V[2, 10] = max {V[1, 10], 12 + V[1, 4]}
...{j − w_i = 10 − 6 = 4 > 0}
= max {10, 12 + 10}
= max {10, 22} = 22

V[3, 1] = V[2, 1] = 0 ... {j − w_i = 1 − 8 = − 7 < 0}

V[3, 2] = V[2, 2] = 0 ... {j − w_i = 2 − 8 = − 6 < 0}

V[3, 3] = V[2, 3] = 0 ... {j − w_i = 3 − 8 = −5 < 0}

V[3, 4] = V[2, 4] = 10 ...{j − w_i = 4 − 8 = − 4 < 0}

V[3, 5] = V[2, 5] = 10 ...{j − w_i = 5 − 8 = − 3 < 0}

V[3, 6] = V[2, 6] = 12 ... {j − w_i = 6 − 8 = − 2 < 0}

V[3, 7] = V[2, 7] = 12 ... {j − w_i = 7 − 8 = − 1 < 0}

V[3, 8] = max V[2, 8], 15 + V[2, 0]}
... {j − w_i = 8 − 8 = 0}
= max {12, 15 + 0}
= max {12, 15} = 15

V[3, 9] = max {V[2, 9], 15 + V[2, 1]}
...{j − w_i = 9 − 8 = 1 > 0}
= max {12, 15 + 0}
= max {12, 15} = 15

$V[3, 10] = \max\{V[2, 10], 15 + V[2, 2]\}$ ...$\{j - w_i = 10 - 8 = 2 > 0\}$

$= \max\{22, 15 + 0\} = \max\{22, 15\} = 22$

- So the maximum profit earned is 22 units.
- Now we trace the composition of an optimal solution:

Since $V[3, 10] = V[2, 10]$, $x_3 = 0$.

Since $V[2, 10] \neq V[1, 10]$, $x_2 = 1$.

$\therefore 10 - w_2 = 10 - 6 = 4$ ....Available capacity j.

So, we check V[1, 4].

Since $V[1, 4] \neq V[0, 4]$, $x_1 = 1$

- **Thus, the maximum profit of 22 units is earned by selecting items 1 and 2 and the weight of a knapsack is 10 units.**

$\therefore$ **x(1 : 3) = (1, 1, 0)**

4.5 MATRIX CHAIN MULTIPLICATION PROBLEM

GQ. Explain chained matrix multiplication with an example. **(9 Marks)**

Dynamic programming presents an efficient algorithm to multiply a chain of matrices.

Problem description

- Consider a chain of n matrices $M_1, M_2, \ldots, M_n$ to be multiplied to get a product :

$$M = M_1 \cdot M_2 \cdot M_3 \cdot \ldots \cdot M_n$$

- The matrix chain multiplication problem asks to determine the optimal parenthesization of a given sequence (chain) of matrices to be multiplied so that the total number of scalar multiplications is minimized.
- E.g. To multiply two matrices $X_{i \times j}$ and $Y_{j \times k}$ we need to perform (i * j * k) number of scalar multiplications to get a resultant matrix $Z_{i \times k}$.
- Suppose we want to multiply three matrices $W_{10 \times 50}$, $X_{50 \times 100}$, $Y_{100 \times 5}$.
- Then for $(W_{10 \times 50} * X_{50 \times 100}) * Y_{100 \times 5} \Rightarrow (10 * 50 * 100) + (10 * 100 * 5) = 50{,}000 + 5000 = 55000$ scalar multiplications are to be done.
- If we do multiplication as $W_{10 \times 50} * (X_{50 \times 100} * Y_{100 \times 5})$ then $(10 * 50 * 5) + (50 * 100 * 5) = 2500 + 25000 = 27500$ scalar multiplications are to be done.
- Thus, optimal parenthesization dramatically reduces the number of scalar multiplications and hence improves the performance of an algorithm.

General procedure

- By brute force method, we can recursively try all possible parenthesizations of n matrices. The time for parenthesization by brute force method can be given by a recurrence formula as below :

$$T(n) = \begin{cases} \sum_{1 \le k \le n-1} T(k)\, T(n-k) & ; \text{ if } n > 1 \\ 1 & ; \text{ if } n = 1 \end{cases}$$

$$= \frac{1}{n} \cdot {}^{2n-2}C_{n-1}$$

$$= \Omega\left(\frac{4^{n-1}}{(n-1)^{3/2}}\right) \Rightarrow \text{Exponential in n.}$$

The basic steps in dynamic programming algorithm:

(1) **Characterise optimal substructure of a solution**

Parenthesization of two sub-sequences $(M_1, M_2, \ldots, M_k)$ and $(M_{k+1}, M_{k+2}, \ldots M_n)$ must each be optimal so that a final sequence of matrices $(M_1, M_2, \ldots, M_n)$ is optimal.

(2) **Devise a recursive formula for an optimal solution by solving overlapping sub-problems**

- Let m(i, j) be the minimum number of scalar multiplications required to calculate multiplication of a sub-sequence $M_{i,\ldots,j} = M_i, M_{i+1}, \ldots M_j$, $1 \le i \le j \le n$.
- Let each matrix M_i is of dimensions $p_{i-1} \times p_i$.
- m(i, j) can be defined by a recurrence relation.
- If i = j, m(i, j) = m(i, i), so no scalar multiplication is required. Hence m(i, i) = 0, $1 \le i \le n$.
- If i < j, we use the optimal substructure property to compute m (i, j) as below.

$m(i, j) = m(i, k) + m(k + 1, j) + p_{i-1} p_k p_j$

where m(i, k) is the cost of multiplication of an optimal sub-sequence $M_i, \ldots M_k = M_{i,\ldots,k}$; m(k + 1, j) is the cost of multiplication of an optimal sub-sequence $M_{k+1}, \ldots, M_j = M_{k+1,\ldots,j}$, $1 \le i \le k \le j \le n$ and $(p_{i-1} p_k p_j)$ is the number of scalar multiplications required to multiply two matrices $M_{i,\ldots,k}$ and $M_{k+1,\ldots,j}$ together.

- As k can take any value between i to j – 1, we need to evaluate all sub-sequences of decisions for different values of k, $1 \le i \le k \le j \le n$. Thus a recursive formula for the minimum cost of parenthesizing a sub-sequence $M_i, \ldots, M_j$ is given as :

$$m(i, j) = \begin{cases} 0 & ; \text{ if } i = j \\ \min\limits_{i \le k \le j} \{m(i, k) + m(k+1, j) + p_{i-1} p_k p_j\} & ; \text{ if } i < j \end{cases}$$

and

$s(i, j) = k$ that gives a position at which we divide a sequence $M_i, \ldots M_j$ in an optimal parenthesization.

(3) Compute the optimal costs using the tabulation method

- Using the recursive formula for m(i, j) all optimal costs are computed by the bottom-up approach. (tabulation method).

(4) Construct the final optimal sequence of decisions by using the results of optimal sub-sequences

- Using all optimal sub-sequences of parenthesization, the final optimal sequence of parenthesization of a given chain of matrices is devised.

☞ Algorithm

```
Algorithm Matrix_Chain_DP (p, n, m, s)
/* Input : p is a sequence of dimensions of n matrices to be
multiplied. Each matrix M_i has dimensions p_{i-1} × p_i.
m[1: n, 1: n] is an auxiliary 2D-array used to store minimum
costs m(i, j). s[1: n – 1, 2: n] is an auxiliary 2D-array used to
store an index k contributed to the optimal cost in
computations of m(i, j).
Output: A table m[1: n, 1: n] of minimum costs m(i, j)'s for all
optimal subsequences. A table s[1: n – 1, 2: n] of all k values
corresponding to each m(i, j). */
{
   n := Length(p) – 1;
                  /* Length(p) gives the length
                     of a sequence p = <p0, p1, ...., pn> */
Build new tables m[1: n, 1 : n] and s[1: n – 1, 2: n];
for (i := 1;  i ≤ n ; i++ )
   m [i, i] := 0;
for (h := 2; h ≤ n; h++)
                        /* h > 1  is a chain length. */
{
   for (i := 1; i ≤ n – h + 1; i++)
   {
      j := i + h – 1;
      m[i, j] := ∞;
      for (k := i ; k < j; k + +)
      {
         cost := m[i, k] + m[k + 1, j] + p_{i-1} · p_k · p_j;
         if (cost < m [i j] )
                        /* Check for minimum cost. */
         {
            m[i, j] := cost;
            s[i, j] := k;
         }
      }
   }
}
```

```
Algorithm Parenthesization (s, i, j)
/* It gives optimal parenthesization values computed by the
algorithm Matrix_Chain_DP( ).
Input : Table s[1:n – 1, 2 : n] that stores the values of k
corresponding to each minimum cost m(i, j). i is the row index
and j is the column index to access the value s[i, j].
Output: An optimal parenthesization for a sequence M_1, M_2,
... M_n of n matrices to be multiplied.*/
{
   if (i = j)
      write ("M"i) ;
   else
   {  write ("(");
      Parenthesization (s, i, s[i, j]);
                        /* Parenthesization for a
                           subsequence M_1, ....., M_k . */
      Parenthesization (s, s[i, j] + 1 , j);
                        /* Parenthesization for a
                           subsequence M_{k+1}, ... M_j .*/
      write (")");
   }
}
```

☞ Complexity analysis

- Considering the nested loop structure (three for loops) in the algorithm Matrix_Chain_DP(), its time complexity is given as $O(n^3)$.
- To access each s[i, j] from a 2D-array s [1 : n – 1, n], it takes $O(n^2)$ time. So the time complexity of the algorithm Parenthesization() is $O(n^2)$.
- Thus the total running time of matrix chain multiplication by dynamic programming is bounded by $O(n^3)$.
- It is much lesser than the exponential complexity of a brute force solution to the same problem.

Ex. 4.5.1 : Write an equation for Chained matrix multiplication using dynamic programming. Find out optimal sequence for multiplication: A_1 [5 × 4], A_2 [4 × 6], A_3 [6 × 2], and A_4 [2 × 7]. Also, give the optimal parenthesization of matrices. **(8 Marks)**

☑ **Soln. :**

- We use the following recursive formula to multiply the chain of matrices M_i ... M_j by dynamic programming where a matrix M_i has dimensions $p_{i-1} \times p_i$.

$$m(i, j) = \begin{cases} 0 \ ; \ \text{if } i = j \\ \min\limits_{i \le k \le j} \{m(i, k) + m(k + 1, j) + p_{i-1} p_k p_j\} \ ; \ \text{if } i < j \end{cases}$$

and

s(i, j) = k that gives a position at which we divide a sequence $M_i, \ldots M_j$ in an optimal parenthesization.

Given: $A_1: (p_0 \times p_1) = (5 \times 4)$,
$A_2: (p_1 \times p_2) = (4 \times 6)$,
$A_3: (p_2 \times p_3) = (6 \times 2)$,
$A_4: (p_3 \times p_4) = (2 \times 7)$.

- So, here we have, $p_0 = 5$, $p_1 = 4$, $p_2 = 6$, $p_3 = 2$, and $p_4 = 7$.

i ↓ / j→1	2	3	4
1	$m_{11}=0$ $s_{11}=0$	$m_{12}=120$ $s_{12}=1$	$m_{13}=88$ $s_{13}=1$	$m_{14}=\mathbf{158}$ $s_{14}=\mathbf{3}$
2		$m_{22}=0$ $s_{22}=0$	$m_{23}=48$ $s_{23}=2$	$m_{24}=132$ $s_{24}=3$
3			$m_{33}=0$ $s_{33}=0$	$m_{34}=84$ $s_{34}=3$
4				$m_{44}=0$ $s_{44}=0$

Initially, m(1,1) = m(2,2) = m(3,3) = m(4,4) = 0 ...(as i = j)

- Now, we compute the first super diagonal as below:

$m(1,2) = m(1,1) + m(2,2) + p_0 \cdot p_1 \cdot p_2$
$= 0 + 0 + 5 \times 4 \times 6 = \mathbf{120}$; $s(1,2) = \mathbf{1}$

$m(2,3) = m(2,2) + m(3,3) + p_1 \cdot p_2 \cdot p_3$
$= 0 + 0 + 4 \times 6 \times 2 = \mathbf{48}$; $s(2,3) = \mathbf{2}$

$m(3,4) = m(3,3) + m(4,4) + p_2 \cdot p_3 \cdot p_4$
$= 0 + 0 + 6 \times 2 \times 7 = \mathbf{84}$; $s(3,4) = \mathbf{3}$

- Now, we compute the second super diagonal as below:

$m(1,3) = \min\{(m(1,1) + m(2,3) + p_0 \cdot p_1 \cdot p_3), (m(1,2) + m(3,3) + p_0 \cdot p_2 \cdot p_3)\}$
$= \min\{(0 + 48 + 5 \times 4 \times 2), (120 + 0 + 5 \times 6 \times 2)\}$
$= \min\{88, 180\} = \mathbf{88}$; $s(1,3) = \mathbf{1}$

$m(2,4) = \min\{(m(2,2) + m(3,4) + p_1 \cdot p_2 \cdot p_4), (m(2,3) + m(4,4) + p_1 \cdot p_3 \cdot p_4)\}$
$= \min\{(0 + 84 + 4 \times 6 \times 7), (48 + 0 + 4 \times 3 \times 7)\}$
$= \min\{252, 132\} = \mathbf{132}$; $s(2,4) = \mathbf{3}$

- Now, we compute the final value m(1,4)as below:

$m(1,4) = \min\{(m(1,1) + m(2,4) + p_0 \cdot p_1 \cdot p_4), (m(1,2) + m(3,4) + p_0 \cdot p_2 \cdot p_4), (m(1,3) + m(4,4) + p_0 \cdot p_3 \cdot p_4)\}$
$= \min\{(0 + 132 + 5 \times 4 \times 7), (120 + 84 + 5 \times 6 \times 7), (88 + 0 + 5 \times 2 \times 7)\}$
$= \min\{272, 414, 158\} = \mathbf{158}$; $s(1,4) = \mathbf{3}$

- Since s(1,4) = 3 the split is at k = 3 with the resultant order $(A_1A_2A_3)A_4$.
- To split $A_1 A_2 A_3$ we check s(1,3). As it is 1, we get the resultant order $(((A_1)\ (A_2A_3))\ A_4)$.
- **Thus, the optimal order of parenthesization for the given chain of matrices is $(((A_1)\ (A_2\ A_3))A_4)$ and the optimal number of scalar multiplications are 158.**

Ex. 4.5.2 : Find the optimal way of multiplying the following matrices using dynamic programming. Also, indicate the optimal number of multiplications required. A : 3 x 2, B: 2 x 5, C:5 x 4, D: 4 x 3, E: 3 x 3. **(8 Marks)**

☑ **Soln. :**

- We use the following recursive formula to multiply the chain of matrices $M_i, \ldots, M_j$ by dynamic programming where a matrix M_i has dimensions $p_{i-1} \times p_i$.

$$m(i, j) = \begin{cases} 0 \ ; \ \text{if } i = j \\ \min\limits_{i \le k \le j} \{m(i, k) + m(k+1, j) + p_{i-1} p_k p_j\} \ ; \ \text{if } i < j \end{cases}$$

and

s(i, j) = k that gives a position at which we divide a sequence $M_i, \ldots M_j$ in an optimal parenthesization.

Given : A: $(p_0 \times p_1) = (3 \times 2)$,
B: $(p_1 \times p_2) = (2 \times 5)$,
C: $(p_2 \times p_3) = (5 \times 4)$,
D: $(p_3 \times p_4) = (4 \times 3)$,
E: $(p_4 \times p_5) = (3 \times 3)$.

- So, here we have, $p_0 = 3$, $p_1 = 2$, $p_2 = 5$, $p_3 = 4$, $p_4 = 3$ and $p_5 = 3$.

i ↓ / j→1	2	3	4	5
1	$m_{11}=0$ $s_{11}=0$	$m_{12}=30$ $s_{12}=1$	$m_{13}=64$ $s_{13}=1$	$m_{14}=82$ $s_{14}=1$	$m_{15}=\mathbf{100}$ $s_{15}=\mathbf{1}$
2		$m_{22}=0$ $s_{22}=0$	$m_{23}=40$ $s_{23}=2$	$m_{24}=64$ $s_{24}=3$	$m_{25}=82$ $s_{25}=4$
3			$m_{33}=0$ $s_{33}=0$	$m_{34}=60$ $s_{34}=3$	$m_{35}=96$ $s_{35}=3$
4				$m_{44}=0$ $s_{44}=0$	$m_{45}=36$ $s_{45}=4$
5					$m_{55}=0$ $s_{55}=0$

Initially, $m(1,1) = m(2,2) = m(3,3) = m(4,4) = m(5,5) = 0$ ...(as i = j)

- Now, we compute the first super diagonal as below:

$m(1,2) = m(1,1) + m(2,2) + p_0 \cdot p_1 \cdot p_2$
$= 0 + 0 + 3 \times 2 \times 5 = \mathbf{30}$; $s(1,2) = \mathbf{1}$

$m(2,3) = m(2,2) + m(3,3) + p_1 \cdot p_2 \cdot p_3$
$= 0 + 0 + 2 \times 5 \times 4 = \mathbf{40}$; $s(2,3) = \mathbf{2}$

$m(3,4) = m(3,3) + m(4,4) + p_2 \cdot p_3 \cdot p_4$
$= 0 + 0 + 5 \times 4 \times 3 = \mathbf{60}$; $s(3,4) = \mathbf{3}$

$m(4,5) = m(4,4) + m(5,5) + p_3 \cdot p_4 \cdot p_5$
$= 0 + 0 + 4 \times 3 \times 3 = \mathbf{36}$; $s(4,5) = \mathbf{4}$

- Now, we compute the second super diagonal as below:

$m(1,3) = \min\{(m(1,1) + m(2,3) + p_0 \cdot p_1 \cdot p_3), (m(1,2) + m(3,3) + p_0 \cdot p_2 \cdot p_3)\}$
$= \min\{(0 + 40 + 3 \times 2 \times 4), (30 + 0 + 3 \times 5 \times 4)\}$
$= \min\{64, 90\} = \mathbf{64}$; $s(1,3) = \mathbf{1}$

$m(2,4) = \min\{(m(2,2) + m(3,4) + p_1 \cdot p_2 \cdot p_4), (m(2,3) + m(4,4) + p_1 \cdot p_3 \cdot p_4)\}$
$= \min\{(0 + 60 + 2 \times 5 \times 3), (40 + 0 + 2 \times 4 \times 3)\}$
$= \min\{90, 64\} = \mathbf{64}$; $s(2,4) = \mathbf{3}$

$m(3,5) = \min\{(m(3,3) + m(4,5) + p_2 \cdot p_3 \cdot p_5), (m(3,4) + m(5,5) + p_2 \cdot p_4 \cdot p_5)\}$
$= \min\{(0 + 36 + 5 \times 4 \times 3), (60 + 0 + 5 \times 3 \times 3)\}$
$= \min\{96, 105\} = \mathbf{96}$; $s(3,5) = \mathbf{3}$

- Now, we compute the third super diagonal as below:

$m(1,4) = \min\{(m(1,1) + m(2,4) + p_0 \cdot p_1 \cdot p_4), (m(1,2) + m(3,4) + p_0 \cdot p_2 \cdot p_4), (m(1,3) + m(4,4) + p_0 \cdot p_3 \cdot p_4)\}$
$= \min\{(0 + 64 + 3 \times 2 \times 3), (30 + 60 + 3 \times 5 \times 3), (64 + 0 + 3 \times 4 \times 3)\}$
$= \min\{82, 135, 100\} = \mathbf{82}$; $s(1,4) = \mathbf{1}$

$m(2,5) = \min\{(m(2,2) + m(3,5) + p_1 \cdot p_2 \cdot p_5), (m(2,3) + m(4,5) + p_1 \cdot p_3 \cdot p_5), (m(2,4) + m(5,5) + p_1 \cdot p_4 \cdot p_5)\}$
$= \min\{(0 + 96 + 2 \times 5 \times 3), (40 + 36 + 2 \times 4 \times 3), (64 + 0 + 2 \times 3 \times 3)\}$
$= \min\{126, 100, 82\} = \mathbf{82}$; $s(2,5) = \mathbf{4}$

- Now, we compute the final value m(1,5) as below:

$m(1,5) = \min\{(m(1,1) + m(2,5) + p_0 \cdot p_1 \cdot p_5), (m(1,2) + m(3,5) + p_0 \cdot p_2 \cdot p_5), (m(1,3) + m(4,5) + p_0 \cdot p_3 \cdot p_5), (m(1,4) + m(5,5) + p_0 \cdot p_4 \cdot p_5)\}$
$= \min\{(0 + 82 + 3 \times 2 \times 3), (30 + 96 + 3 \times 5 \times 3), (64 + 36 + 3 \times 4 \times 3), (82 + 0 + 3 \times 3 \times 3)\}$
$= \min\{100, 171, 136, 109\} = \mathbf{100}$; $s(1,5) = \mathbf{1}$

- Since s(1,5) = 1 the split is at k = 1 with the resultant order (A)BCDE.
- To split BCDE we check s(2,5). As it is 4, we get the resultant order (A)((BCD)E).
- To split BCD we check s(2,4). As it is 3, we get the resultant order (A)(((BC)D)E).
- **Thus, the optimal order of parenthesization for the given chain of matrices is ((A)(((BC)D)E))and the optimal number of scalar multiplications are 100.**

Ex. 4.5.3 : For the following chain of matrices find the order of parenthesization for the optimal chain multiplication (15,5,10,20,25). **(8 Marks)**

☑ **Soln. :**

- We use the following recursive formula to multiply the chain of matrices $M_i, ..., M_j$ by dynamic programming where a matrix M_i has dimensions $p_{i-1} \times p_i$.

$$m(i,j) = \begin{cases} 0 \ ; \ \text{if } i = j \\ \min\limits_{i \le k \le j} \{m(i,k) + m(k+1,j) + p_{i-1} p_k p_j\} \ ; \ \text{if } i < j \end{cases}$$

and

s(i, j) = k that gives a position at which we divide a sequence $M_i, \ldots M_j$ in an optimal parenthesization.

Given : $p_0 = 15$, $p_1 = 5$, $p_2 = 10$, $p_3 = 20$, and $p_4 = 25$.

i ↓ / j→1	2	3	4
1	$m_{11}= 0$ $s_{11}= 0$	$m_{12}= 750$ $s_{12}= 1$	$m_{13}= 2500$ $s_{13}= 1$	$m_{14}=5375$ $s_{14}= 1$
2		$m_{22}= 0$ $s_{22}= 0$	$m_{23}= 1000$ $s_{23}= 2$	$m_{24}= 3500$ $s_{24}= 3$
3			$m_{33}= 0$ $s_{33}= 0$	$m_{34}= 5000$ $s_{34}= 3$
4				$m_{44}= 0$ $s_{44}= 0$

Initially, m(1,1) = m(2,2) = m(3,3) = m(4,4) = 0 ...(as i = j)

- Now, we compute the first super diagonal as below:

$m(1,2) = m(1,1) + m(2,2) + p_0 \cdot p_1 \cdot p_2$
$= 0 + 0 + 15 \times 5 \times 10 = \mathbf{750}$; $s(1,2) = \mathbf{1}$

$m(2,3) = m(2,2) + m(3,3) + p_1 \cdot p_2 \cdot p_3$
$= 0 + 0 + 5 \times 10 \times 20 = \mathbf{1000}$; $s(2,3) = \mathbf{2}$

$m(3,4) = m(3,3) + m(4,4) + p_2 \cdot p_3 \cdot p_4$
$= 0 + 0 + 10 \times 20 \times 25 = \mathbf{5000}$; $s(3,4) = \mathbf{3}$

- Now, we compute the second super diagonal as below:

$m(1,3) = \min\{(m(1,1) + m(2,3) + p_0 \cdot p_1 \cdot p_3), (m(1,2) + m(3,3) + p_0 \cdot p_2 \cdot p_3)\}$
$= \min\{(0 + 1000 + 15 \times 5 \times 20), (750 + 0 + 15 \times 10 \times 20)\}$
$= \min\{2500, 3750\} = \mathbf{2500}$; $s(1,3) = \mathbf{1}$

$m(2,4) = \min\{(m(2,2) + m(3,4) + p_1 \cdot p_2 \cdot p_4), (m(2,3) + m(4,4) + p_1 \cdot p_3 \cdot p_4)\}$
$= \min\{(0 + 5000 + 5 \times 10 \times 25), (1000 + 0 + 5 \times 20 \times 25)\}$
$= \min\{6250, 3500\} = \mathbf{3500}$; $s(2,4) = \mathbf{3}$

- Now, we compute the final value m(1,4) as below:

$m(1,4) = \min\{(m(1,1) + m(2,4) + p_0 \cdot p_1 \cdot p_4), (m(1,2) + m(3,4) + p_0 \cdot p_2 \cdot p_4), (m(1,3) + m(4,4) + p_0 \cdot p_3 \cdot p_4)\}$
$= \min\{(0 + 3500 + 15 \times 5 \times 25), (750 + 5000 + 15 \times 10 \times 25), (2500 + 0 + 15 \times 20 \times 25)\}$
$= \min\{5375, 9500, 10000\} = \mathbf{5375}$; $s(1,4) = \mathbf{1}$

- Since s(1,4) = 1 the split is at k = 1 with the resultant order $(M_1)M_2 M_3 M_4$.
- To split $M_2M_3M_4$ we check s(2,4). As it is 3, we get the resultant order $((M_1)((M_2M_3)M_4))$.
- **Thus, the optimal order of parenthesization for the given chain of matrices is $((M_1)((M_2 M_3) M_4))$ and the optimal number of scalar multiplications is 5375.**

Ex. 4.5.4 : Find an optimal sequence of multiplication using dynamic programming of the following matrices : A1[10 x 100], A2[100 x 5], A3[5 x 50] and A4[50 x 1]. List the optimal number of multiplication and parenthesization of matrices. **(8 Marks)**

☑ **Soln. :**

- We use the following recursive formula to multiply the chain of matrices $M_i, ..., M_j$ by dynamic programming where a matrix M_i has dimensions $p_{i-1} \times p_i$.

$$m(i,j) = \begin{cases} 0 \ ; \ \text{if } i = j \\ \min\limits_{i \le k \le j} \{m(i,k) + m(k+1,j) + p_{i-1} p_k p_j\} \ ; \ \text{if } i < j \end{cases}$$

and

s(i, j) = k that gives a position at which we divide a sequence $M_i, ... M_j$ in an optimal parenthesization.

Given: $A_1 : (p_0 \times p_1) = (10 \times 100)$,
$A_2 : (p_1 \times p_2) = (100 \times 5)$,
$A_3 : (p_2 \times p_3) = (5 \times 50)$,
$A_4 : (p_3 \times p_4) = (50 \times 1)$.

So, here we have, $p_0 = 10$, $p_1 = 100$, $p_2 = 5$, $p_3 = 50$ and $p_4 = 1$.

i ↓ / j→	1	2	3	4
1	$m_{11} = 0$ $s_{11} = 0$	$m_{12} = 5000$ $s_{12} = 1$	$m_{13} = 7500$ $s_{13} = 2$	$m_{14} = \mathbf{1750}$ $s_{14} = \mathbf{1}$
2		$m_{22} = 0$ $s_{22} = 0$	$m_{23} = 25000$ $s_{23} = 2$	$m_{24} = 750$ $s_{24} = 2$
3			$m_{33} = 0$ $s_{33} = 0$	$m_{34} = 250$ $s_{34} = 3$
4				$m_{44} = 0$ $s_{44} = 0$

Initially, m(1,1) = m(2,2) = m(3,3) = m(4,4) = 0 ...(as i = j)

- Now, we compute the first super diagonal as below:

$m(1,2) = m(1,1) + m(2,2) + p_0 \cdot p_1 \cdot p_2$
$= 0 + 0 + 10 \times 100 \times 5 = \mathbf{5000}$; $s(1,2) = \mathbf{1}$

$m(2,3) = m(2,2) + m(3,3) + p_1 \cdot p_2 \cdot p_3$
$= 0 + 0 + 100 \times 5 \times 50 = \mathbf{25000}$; $s(2,3) = \mathbf{2}$

$m(3,4) = m(3,3) + m(4,4) + p_2 \cdot p_3 \cdot p_4$
$= 0 + 0 + 5 \times 50 \times 1 = \mathbf{250}$; $s(3,4) = \mathbf{3}$

- Now, we compute the second super diagonal as below:

$m(1,3) = \min\{(m(1,1) + m(2,3) + p_0 \cdot p_1 \cdot p_3), (m(1,2) + m(3,3) + p_0 \cdot p_2 \cdot p_3)\}$
$= \min\{(0 + 25000 + 10 \times 100 \times 50), (5000 + 0 + 10 \times 5 \times 50)\}$
$= \min\{75000, 7500\} = \mathbf{7500}$; $s(1,3) = \mathbf{2}$

$m(2,4) = \min\{(m(2,2) + m(3,4) + p_1 \cdot p_2 \cdot p_4), (m(2,3) + m(4,4) + p_1 \cdot p_3 \cdot p_4)\}$
$= \min\{(0 + 250 + 100 \times 5 \times 1), (25000 + 0 + 100 \times 50 \times 1)\}$
$= \min\{750, 30000\} = \mathbf{750}$; $s(2,4) = \mathbf{2}$

- Now, we compute the final value m(1,4) as below:

$$m(1,4) = \min\{(m(1,1) + m(2,4) + p_0 p_1 p_4), (m(1,2) + m(3,4) + p_0 p_2 p_4), (m(1,3) + m(4,4) + p_0 p_3 p_4)\}$$
$$= \min\{(0 + 750 + 10 \times 100 \times 1), (5000 + 250 + 10 \times 5 \times 1), (7500 + 0 + 10 \times 50 \times 1)\}$$
$$= \min\{1750, 5300, 8000\} = \mathbf{1750} \; ; \; s(1,4) = \mathbf{1}$$

- Since $s(1,4) = 1$ the split is at k=1 with the resultant order $(A_1)\, A_2 A_3 A_4$.
- To split $A_2 A_3 A_4$ we check s(2,4). As it is 2, we get the resultant order $((A_1)\,((A_2)(A_3A_4)))$
- **Thus, the optimal order of parenthesization for the given chain of matrices is $((A_1)\,((A_2)(A_3\,A_4)))$and the optimal number of scalar multiplications is 1750.**

Ex. 4.5.5 : For the following chain of matrices find the order of parenthesization for the optimal chain multiplication (13,5,89,3,34). **(8 Marks)**

☑ **Soln. :**

- We use the following recursive formula to multiply the chain of matrices $M_i, .., M_j$ by dynamic programming where a matrix M_i has dimensions $p_{i-1} \times p_i$.

$$m(i,j) = \begin{cases} 0 \; ; \; \text{if } i = j \\ \min\limits_{i \le k \le j} \{m(i,k) + m(k+1,j) + p_{i-1} p_k p_j\} \; ; \; \text{if } i < j \end{cases}$$

and

$s(i, j) = k$ that gives a position at which we divide a sequence $M_i, \ldots M_j$ in an optimal parenthesization.

Given: $p_0 = 15$, $p_1 = 5$, $p_2 = 89$, $p_3 = 3$, and $p_4 = 34$.

i ↓ J →	1	2	3	4
1	$m_{11} = 0$ $s_{11} = 0$	$m_{12} = 6675$ $s_{12} = 1$	$m_{13} = 1560$ $s_{13} = 1$	$m_{14} = 3090$ $s_{14} = 3$
2		$m_{22} = 0$ $s_{22} = 0$	$m_{23} = 1335$ $s_{23} = 2$	$m_{24} = 1845$ $s_{24} = 3$
3			$m_{33} = 0$ $s_{33} = 0$	$m_{34} = 9078$ $s_{34} = 3$
4				$m_{44} = 0$ $s_{44} = 0$

- Initially, $m(1,1) = m(2,2) = m(3,3) = m(4,4) = 0$ ...(as i = j)
- Now, we compute the first super diagonal as below:

$$m(1,2) = m(1,1) + m(2,2) + p_0 p_1 p_2 = 0 + 0 + 15 \times 5 \times 89 = \mathbf{6675} \; ; \; s(1,2) = 1$$
$$m(2,3) = m(2,2) + m(3,3) + p_1 p_2 p_3 = 0 + 0 + 5 \times 89 \times 3 = \mathbf{1335} \; ; \; s(2,3) = 2$$
$$m(3,4) = m(3,3) + m(4,4) + p_2 p_3 p_4 = 0 + 0 + 89 \times 3 \times 34 = \mathbf{9078} \; ; \; s(3,4) = 3$$

- Now, we compute the second super diagonal as below:

$$m(1,3) = \min\{(m(1,1) + m(2,3) + p_0 p_1 p_3), (m(1,2) + m(3,3) + p_0 p_2 p_3)\}$$
$$= \min\{(0 + 1335 + 15 \times 5 \times 3), (6675 + 0 + 15 \times 89 \times 3)\}$$
$$= \min\{1560, 10680\} = \mathbf{1560} \; ; \; s(1,3) = 1$$
$$m(2,4) = \min\{(m(2,2) + m(3,4) + p_1 p_2 p_4), (m(2,3) + m(4,4) + p_1 p_3 p_4)\}$$
$$= \min\{(0 + 9078 + 5 \times 89 \times 34), (1335 + 0 + 5 \times 3 \times 34)\}$$
$$= \min\{24208, 1845\} = \mathbf{1845} \; ; \; s(2,4) = 3$$

- Now, we compute the final value m(1,4)as below :

$$m(1,4) = \min\{(m(1,1) + m(2,4) + p_0 p_1 p_4), (m(1,2) + m(3,4) + p_0 p_2 p_4), (m(1,3) + m(4,4) + p_0 p_3 p_4)\}$$
$$= \min\{(0 + 1845 + 15 \times 5 \times 34), (6675 + 9078 + 15 \times 89 \times 34), (1560 + 0 + 15 \times 3 \times 34)\}$$
$$= \min\{4395, 61143, 3090\} = \mathbf{3090} \; ; \; s(1,4) = 3$$

- Since s(1, 4) = 1 the split is at k = 3 with the resultant order $(A_1A_2A_3)A_4$.
- To split $A_1A_2A_3$ we check s(1,3). As it is 1, we get the resultant order $(((A_1)\,(A_2A_3))A_4)$.
- **Thus, the optimal order of parenthesization for the given chain of matrices is $(((A_1)\,(A_2\,A_3))A_4)$ and the optimal number of scalar multiplications are 3090.**

Ex. 4.5.6 : Using dynamic programming find out the optimal sequence for the matrix chain multiplication of A4x10, B10x3, C3x12, D12x20, and E20x7 matrices. **(8 Marks)**

☑ **Soln. :**

- We use the following recursive formula to multiply the chain of matrices $M_i, .., M_j$ by dynamic programming where a matrix M_i has dimensions $p_{i-1} \times p_i$.

$$m(i,j) = \begin{cases} 0 \; ; \; \text{if } i = j \\ \min\limits_{i \le k \le j} \{m(i,k) + m(k+1,j) + p_{i-1} p_k p_j\} \; ; \; \text{if } i < j \end{cases}$$

and $s(i, j) = k$ that gives a position at which we divide a sequence $M_i, \ldots M_j$ in an optimal parenthesization.

Given : A : $(p_0 \times p_1) = (4 \times 10)$,
B : $(p_1 \times p_2) = (10 \times 3)$,
C : $(p_2 \times p_3) = (3 \times 12)$,
D : $(p_3 \times p_4) = (12 \times 20)$,
E : $(p_4 \times p_5) = (20 \times 7)$.

So, here we have, $p_0 = 4$, $p_1 = 10$, $p_2 = 3$, $p_3 = 12$, $p_4 = 20$ and $p_5 = 7$.

i ↓ \ j →	1	2	3	4	5
1	$m_{11}=0$ $s_{11}=0$	$m_{12}=120$ $s_{12}=1$	$m_{13}=264$ $s_{13}=2$	$m_{14}=1080$ $s_{14}=2$	$m_{15}=630$ $s_{15}=1$
2		$m_{22}=0$ $s_{22}=0$	$m_{23}=360$ $s_{23}=2$	$m_{24}=1320$ $s_{24}=2$	$m_{25}=350$ $s_{25}=2$
3			$m_{33}=0$ $s_{33}=0$	$m_{34}=720$ $s_{34}=3$	$m_{35}=1140$ $s_{35}=4$
4				$m_{44}=0$ $s_{44}=0$	$m_{45}=1680$ $s_{45}=4$
5					$m_{55}=0$ $s_{55}=0$

Initially, $m(1,1) = m(2,2) = m(3,3) = m(4,4) = m(5,5) = 0$ ...(as i = j)

- Now, we compute the first super diagonal as below :

$m(1,2) = m(1,1) + m(2,2) + p_0 \cdot p_1 \cdot p_2$
$= 0 + 0 + 4 \times 10 \times 3 = \mathbf{120}$; $s(1,2) = \mathbf{1}$

$m(2,3) = m(2,2) + m(3,3) + p_1 \cdot p_2 \cdot p_3$
$= 0 + 0 + 10 \times 3 \times 12 = \mathbf{360}$; $s(2,3) = \mathbf{2}$

$m(3,4) = m(3,3) + m(4,4) + p_2 \cdot p_3 \cdot p_4$
$= 0 + 0 + 3 \times 12 \times 20 = \mathbf{720}$; $s(3,4) = \mathbf{3}$

$m(4,5) = m(4,4) + m(5,5) + p_3 \cdot p_4 \cdot p_5$
$= 0 + 0 + 12 \times 20 \times 7 = \mathbf{1680}$; $s(4,5) = \mathbf{4}$

- Now, we compute the second super diagonal as below:

$m(1,3) = \min\{(m(1,1) + m(2,3) + p_0 \cdot p_1 \cdot p_3),$
$(m(1,2) + m(3,3) + p_0 \cdot p_2 \cdot p_3)\}$
$= \min\{(0 + 360 + 4 \times 10 \times 12),$
$(120 + 0 + 4 \times 3 \times 12)\}$
$= \min\{840, 264\} = \mathbf{264}$; $s(1,3) = \mathbf{2}$

$m(2,4) = \min\{(m(2,2) + m(3,4) + p_1 \cdot p_2 \cdot p_4),$
$(m(2,3) + m(4,4) + p_1 \cdot p_3 \cdot p_4)\}$
$= \min\{(0 + 720 + 10 \times 3 \times 20),$
$(360 + 0 + 10 \times 12 \times 20)\}$
$= \min\{1320, 2760\} = \mathbf{1320}$; $s(2,4) = 2$

$m(3,5) = \min\{(m(3,3) + m(4,5) + p_2 \cdot p_3 \cdot p_5),$
$(m(3,4) + m(5,5) + p_2 \cdot p_4 \cdot p_5)\}$
$= \min\{(0 + 1680 + 3 \times 12 \times 7),$
$(720 + 0 + 3 \times 20 \times 7)\}$
$= \min\{1932, 1140\} = \mathbf{1140}$; $s(3,5) = \mathbf{4}$

- Now, we compute the third super diagonal as below:

$m(1,4) = \min\{(m(1,1) + m(2,4) + p_0 \cdot p_1 \cdot p_4),$
$(m(1,2) + m(3,4) + p_0 \cdot p_2 \cdot p_4),$
$(m(1,3) + m(4,4) + p_0 \cdot p_3 \cdot p_4)\}$
$= \min\{(0 + 1320 + 4 \times 10 \times 20),$
$(120 + 720 + 4 \times 3 \times 20),$
$(264 + 0 + 4 \times 20 \times 20)\}$
$= \min\{2120, 1080, 1864\} = \mathbf{1080}$; $s(1,4) = 2$

$m(2,5) = \min\{(m(2,2) + m(3,5) + p_1 \cdot p_2 \cdot p_5),$
$(m(2,3) + m(4,5) + p_1 \cdot p_3 \cdot p_5),$
$(m(2,4) + m(5,5) + p_1 \cdot p_4 \cdot p_5)\}$
$= \min\{(0 + 140 + 10 \times 3 \times 7),$
$(360 + 1680 + 10 \times 12 \times 7),$
$(1320 + 0 + 10 \times 20 \times 7)\}$
$= \min\{350, 2880, 1720\} = \mathbf{350}$; $s(2,5) = 2$

- Now, we compute the final value m(1,5) as below:

$m(1,5) = \min\{(m(1,1) + m(2,5) + p_0 \cdot p_1 \cdot p_5),$
$(m(1,2) + m(3,5) + p_0 \cdot p_2 \cdot p_5),$
$(m(1,3) + m(4,5) + p_0 \cdot p_3 \cdot p_5),$
$(m(1,4) + m(5,5) + p_0 \cdot p_4 \cdot p_5)\}$
$= \min\{(0 + 350 + 4 \times 10 \times 7),$
$(120 + 1140 + 4 \times 3 \times 7),$
$(264 + 1680 + 4 \times 12 \times 7),$
$(1080 + 0 + 4 \times 20 \times 7)\}$
$= \min\{630, 1344, 2280, 1640\}$
$= \mathbf{630}$; $s(1,5) = \mathbf{1}$

Since s(1,5) = 1 the split is at k = 1 with the resultant order (A)BCDE.

- To split BCDE we check s(2,5). As it is 2, we get the resultant order (A)((B)CDE).
- To split CDE we check s(3,5). As it is 4, we get the resultant order (A)((B)((CD)E)).
- **Thus, the optimal order of parenthesization for the given chain of matrices is ((A)((B)((CD)E)))and the optimal number of scalar multiplications are 630.**

Summary

- **Dynamic programming (DP)** is suitable for solving optimization problems with overlapping sub-problems.
- DP applies to problems when the optimal decision sequence cannot be generated through stepwise decisions based on locally optimal criteria.

- DP avoids redundant computations by solving overlapping sub-problems exactly once and storing their results for further calculations.
- **Principle of optimality :** It claims that an optimal decision sequence to any of the problems must be produced through the optimal decision sequences to its sub-problems. It avoids computations that do not lead to an optimal solution.
- **The memoization** technique eliminates the weaknesses of both the top-down and the bottom-up approaches to find out a solution to problems with overlapping sub-problems. It avoids redundant computations of the top-down approach and reduces computations of unnecessary subproblems, unlike the bottom-up approach.
- DP leads to pseudo-polynomial algorithms that work efficiently for smaller instances of a problem; but for larger instances of a problem, it leads to exponential complexity.
- **Some of the classic problems** that can be solved effectively by dynamic programming are listed below:

Replace it by :

(1) Binomial coefficients problem,
(2) Optimal Binary Search Tree (OBST) problem
(3) 0/1 knapsack problem
(4) Matrix chain multiplication problem
(5) Single source shortest problem
(6) All pairs shortest path problem
(7) Multistage graph problem
(8) Reliability design problem
(9) Travelling Salesman Problem (TSP)
(10) Assembly-line scheduling problem
(11) Longest common subsequence problem

- **Recursive formulae** to solve some of the classic problems using dynamic programming :

Classic Problems and their Recursive Formulae by DP

Calculating the binomial coefficient :

$\binom{n}{r}$ or C[n, r] =

$$\begin{cases} 1 & ; \text{if } r = 0 \\ 1 & ; \text{if } r = n \\ \binom{n-1}{r} + \binom{n-1}{r-1} & ; 0 < r < n \end{cases}$$

Classic Problems and their Recursive Formulae by DP

Optimal Binary Search Tree problem:

$$c(i, j) = \min_{i < k \le j} \{c(i, k-1) + c(k, j) + w(i, j)\}; 0 \le i \le j \le n.$$

$c(i, i) = 0$; $w(i, i) = q_i$; $w(i, j) = p_j + q_j + w(i, j-1)$;

$0 \le i \le j \le n$

0/1 Knapsack problem

n = the number of items, M = capacity of a knapsack, $0 \le i \le n$, $0 \le j \le M$.

(1) Set Representation:

$V_i(j) = \max\{V_{i-1}(j), V_{i-1}(j-w_i) + p_i\}$; $S^i = \{(P, W)\}$

where $P = V_i(j)$ and $W = j$, $0 \le j \le n$, $0 \le j \le M$,

n = number of items and M = capacity of a knapsack

$S^0 = \{(0, 0)\}$

$S_1^i = \{(P, w) \mid (P - p_{i+1}, W - w_{i+1}) \in S^i\}$

$S^{i+1} = S^i \cup S_1^i$

(2) Tabular Representation:

$$V[i, j] = \begin{cases} \max\{V[i-1, j], V[i-1, j-w_i] + p_i\} & ; \text{if } j - w_i \ge 0 \\ V[i-1, j] & ; \text{if } j - w_i < 0 \end{cases}$$

$V[0, j] = 0$ if $j \ge 0$ and $V[i, 0] = 0$ if $i \ge 0$

Matrix chain multiplication:

M_i ,.., M_j is the given chain of matrices where a matrix M_i has dimensions $p_{i-1} \times p_i$.

$$m(i, j) = \begin{cases} 0 & ; \text{if } i = j \\ \min_{i \le k \le j} \{m(i, k) + m(k+1, j) + p_{i-1} p_k p_j\} & ; \text{if } i < j \end{cases}$$

and

$s(i, j) = k$ gives a position at which we divide a sequence $M_i, \ldots M_j$ in an optimal parenthesization.

Chapter Ends...

□□□

UNIT IV

CHAPTER 5

Backtracking

Syllabus

Backtracking : Principle, control abstraction, time analysis of control abstraction, 8-queens problem, Graph coloring, Sum of subsets problem.

5.1 Principle and General Method of Backtracking 5-2

5.1.1 Basics of Backtracking Strategy 5-2

UQ. State the principle of backtracking. Explain the constraints used in backtracking with an example. **(SPPU - Q. 5(a), Dec. 17, 8 Marks)** 5-2

UQ. With respect to backtracking what do you mean by explicit and implicit constraints? **(SPPU - Q. 7(b), May 14, 6 Marks)** 5-2

UQ. Write short notes on : (i) State space tree (ii) Live node (iii) Expanding node (E-node) (iv) Bounding function **(SPPU - Q. 6(b), May 15, 8 Marks)** 5-2

UQ. Explain the following term: (i) State space tree (ii) Live node (iii) E-node (iv) Dead node **(SPPU - Q. 6(b), May 18, 8 Marks)** 5-2

UQ. What is the 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 **(SPPU - Q. 10(b), May 11, 8 Marks)** 5-2

5.1.2 Recursive Backtracking Method 5-4

UQ. What is backtracking? Write a general recursive algorithm for backtracking. **(SPPU - Q. 6(a), Dec. 15, 8 Marks)** 5-4

UQ. Write a recursive algorithm which shows a recursive formulation of the backtracking technique and explain it. **(SPPU - Q. 5(a), May 16, Q. 5(a), Dec. 18, 8 Marks)** 5-4

UQ. Write a recursive and iterative algorithm of the backtracking method. **(SPPU - Q. 5(a), May 18, 8 Marks)** 5-4

5.1.3 Iterative Backtracking Method 5-5

UQ. What is backtracking? Write a general iterative algorithm for backtracking. **(SPPU - May 15, Q. 6(a), 8 Marks)** 5-5

5.1.4 Complexity Analysis of the Backtracking Algorithms 5-5

5.2 8-Queens Problem 5-6

UQ. State the principle of backtracking and write backtracking algorithm for N-queens problem. **(SPPU - Q. 5(b), May 19, 8 Marks)** 5-8

UQ. Write an iterative and recursive backtracking algorithm for N-Queens problem. **(SPPU - Q. 6(a), Dec. 19, 8 Marks)** 5-8

5.3 Graph Coloring Problem 5-10

5.4 Sum of Subsets Problem 5-15

Chapter Ends 5-21

To solve a Rubik's cube, we start with some rotations and try to arrange some cubes. We test all of its faces. If that arrangement is not satisfying the constraints of the puzzle, then we discard those rotations, go back to the prior state, and retry another rotation. We continue this process until the puzzle gets solved. The strategy we apply to solve a Rubik's cube is backtracking.

Typical algorithmic strategies like divide and conquer, greedy method, dynamic programming, etc. are not suitable to solve some problems. E.g., Sudoku, maze puzzle, pentomino configuration problem, graph colouring problem, the sum of subsets problem. Considering the significance of such problems we cannot just leave them due to algorithmic limitations. Backtracking and Branch-and-bound methods provide solutions to at least some large instances of such difficult but important combinatorial problems.

5.1 PRINCIPLE AND GENERAL METHOD OF BACKTRACKING

- The backtracking technique is used to solve constraint satisfaction problems.
- It incrementally constructs a solution to a given problem by considering one candidate solution at a time.
- It discards the candidate solution if it violates the constraints mentioned in the problem.
- Thus, it is the improved or smart exhaustive search. However, in the worst case, it also reaches exponential complexity.
- Backtracking follows a depth first search.

5.1.1 Basics of Backtracking Strategy

UQ. State the principle of backtracking. Explain the constraints used in backtracking with an example.
(SPPU - Q. 5(a), Dec. 17, 8 Marks)

UQ. With respect to backtracking what do you mean by explicit and implicit constraints?
(SPPU - Q. 7(b), May 14, 6 Marks)

UQ. Write short notes on :
(i) State space tree (ii) Live node
(iii) Expanding node (E-node)
(iv) Bounding function
(SPPU - Q. 6(b), May 15, 8 Marks)

UQ. Explain the following term:
(i) State space tree (ii) Live node
(iii) E-node (iv) Dead node
(SPPU - Q. 6(b), May 18, 8 Marks)

UQ. What is the 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
(SPPU - Q. 10(b), May 11, 8 Marks)

GQ. What are the constraints that must be satisfied while solving any problem using backtracking? Explain briefly. **(4 Marks)**

GQ. Define backtracking. State types of constraints used in backtracking. **(5 Marks)**

GQ. What is meant by the dead node in backtracking? **(1 Mark)**

Basic terminologies

Solution vector

- The desired solution X to a problem instance P of input size n is expressed as a vector of candidate solutions x_i that are selected from some finite set of possible solutions S.
- Thus, a solution can be represented as an n-tuple $(x_1, x_2, \ldots, x_n)$ and its partial solution is given as $(x_1, x_2, \ldots, x_i)$ where $i < n$.
- E.g. for a 4-queens problem X = {2,4,1,3} is a solution vector.

Constraints

- Constraints are the rules to confine the solution vector $(x_1, x_2, \ldots, x_n)$.
- They determine the values of candidate solutions and their relationship with each other.
- There are two types of constraints:
  (1) implicit constraints and (2) explicit constraints.

Implicit constraints

- These are the rules that identify the tuples in the solution space S that satisfy the specified criterion function of a problem instance P.
- They give the directives of relating all candidate solutions x_i's to each other.
- E.g., in the case of the N-queens problem, all x_i's must be distinct satisfying the criterion function of non-attacking queens, in the case of the 0/1 knapsack problem all x_i's with value '1' must represent the items giving overall maximum profit and having total weight ≤ knapsack capacity.

Explicit constraints

- These are the rules by which all candidate solutions x_i's are restricted to take on values only from a specified set in a problem instance P.
- Explicit constraints vary with the instances of the problem.
- E.g., in case of the N-queens problem, if N = 4 then $x_i \in \{1, 2, 3, 4\}$ and if N = 8 then $x_i \in \{1, 2, 3, 4, 5, 6, 7, 8\}$; in case of 0/1 knapsack problem $x_i \in \{0,1\}$ where $x_i = 0$ represents the exclusion of an item i and $x_i = 1$ represents the inclusion of an item i.

Solution space

- All candidate solutions x_i's satisfying the explicit constraints form the solution space S of a problem instance P.
- In a state space tree, all paths from the root node to a leaf node describe the solution space.
- E.g., in the case of N-queens problem, all n! orderings of $(x_1, x_2, \ldots, x_n)$ form the solution space of that problem instance.

State space tree

- A representation of the solution space S of a problem instance P in the form of a tree is defined as the state space tree.
- It facilitates systematic search in the solution space to determine the desired solution to a problem.
- A solution space of a given problem can be represented by different state space trees.

State space

- The state space of a problem is described by all paths from a root node to other nodes in a state space tree.

Problem state

- Each node in a state space tree describes a problem state or a partial solution formed by making choices from the root of the tree to that node.

Solution states

- These are the problem states producing a tuple in the solution space S.
- At every internal node, the solution states are partitioned into disjoint sub-solution spaces.
- In a state space tree for a variable tuple size, all nodes are solution states.
- In a state space tree for a fixed tuple size, only the leaf nodes are solution states.

Answer states

- These are the solution states that satisfy the implicit constraints.
- These states thus describe the desired solution-tuple (or answer-tuple).

Promising node

- A node is promising if it eventually leads to the desired solution.
- A promising node corresponds to a partial solution that is still feasible.
- Any time the partial node becomes infeasible, that branch will no longer be pursued.

Non-promising node

- A node is non-promising if it eventually leads to a state that cannot produce the desired solution.
- A non-promising node corresponds to a partial solution that shows infeasibility to get a complete solution.
- Such nodes are killed by a bounding function without further exploration.

Live node

- It is a node that has been generated and all of whose children are not yet been generated.

E-node

- It is a live node whose children are currently being generated.

Dead node

- A node that is either not to be expanded further or for which all its children have been generated is known as a dead node.

Depth first node generation

- Here, the latest live node becomes the next E-node.
- The moment a new child of the current E-node is generated, that child will be the new E-node.
- In backtracking, a state space tree is constructed by using the depth first node generation approach.

Bounding function

- It is also known as a "validity function", or "criterion function", or "promising function".
- It is an optimization function $B(x_1, x_2, \ldots, x_n)$ which is to be either maximized or minimized for a given problem instance P.
- It optimizes the search of a solution vector $(x_1, x_2, \ldots, x_n)$ in the solution space S of a problem instance P.
- It helps to reject the candidate solutions not leading to

the desired solution to the problem. Thus, it kills the live nodes without exploring their children if constraints are not satisfied.

- E.g., in the case of the knapsack problem, the criterion function is the maximization of the profit by filling a knapsack.

Static trees

- These are the state space trees whose tree formulation is independent of the problem instance being solved.

Dynamic trees

- These are the state space trees whose tree formulation varies with the problem instance being solved.

Principle of backtracking

- In backtracking a solution-tuple is determined by evaluating each possible solution in the solution space one by one.
- The solution space is represented through the state space tree that follows depth first node generation.
- The bounding function is applied to check the promising and non-promising nodes.
- Whenever any node violates the bounding function and shows infeasibility, the algorithm stops pursuing that branch further.
- It then "backtracks" and explores an alternative branch.
- By discarding non-promising solutions, backtracking does fewer trials to determine the solution.

Some classic problems solvable by backtracking

(1) N-queens problem
(2) Graph coloring
(3) Sum of subsets
(4) Knapsack problem
(5) Hamiltonian cycle problem
(6) Puzzles like Sudoku, Rubik's cube, crosswords, maze puzzle, etc.
(7) Problems in gamification
(8) Problems in artificial intelligence
(9) PROLOG- a logic programming language

5.1.2 Recursive Backtracking Method

- The backtracking strategy typically uses recursion to incrementally build a complete solution from the partial solutions that satisfy the criterion function.
- Consider a partial solution $(x_1, x_2, \ldots, x_i)$, $i < n$, to a given problem instance of size n.
- Let $T(x_1, x_2, \ldots, x_{i+1})$ be the set of all possible values of x_{i+1} such that $(x_1, x_2, \ldots, x_{i+1})$ is also a path (or a partial solution) from the root to a problem state. Initially, $T(x_1, x_2, \ldots, x_n) = \Phi$
- Let a bounding function $B_{i+1}(x_1, x_2, \ldots, x_{i+1})$ checks whether a partial solution corresponds to a complete solution.
- $B_{i+1}(x_1, x_2, \ldots, x_{i+1})$ returns FALSE for a partial solution $(x_1, x_2, \ldots, x_{i+1})$ if it is non-promising. Then that path from a root node to a problem state cannot be pursued further to reach an answer node.
- Thus, candidate solutions for the position $i + 1$ in a solution vector $(x_1, x_2, \ldots, x_n)$ are those values that satisfy a bounding function B_{i+1} and are generated by T.

Control abstraction of a recursive backtracking algorithm

GQ. Write a recursive algorithm that shows a recursive formulation of the backtracking technique and explain it. **(4 Marks)**

UQ. What is backtracking? Write a general recursive algorithm for backtracking.

(SPPU - Q. 6(a), Dec. 15, 8 Marks)

UQ. Write a recursive algorithm which shows a recursive formulation of the backtracking technique and explain it.

(SPPU - Q. 5(a), May 16, Q. 5(a), Dec. 18, 8 Marks)

UQ. Write a recursive and iterative algorithm of the backtracking method.

(SPPU - Q. 5(a), May 18, 8 Marks)

Algorithm Backtrack_R(i)

```
/* Input: X[1: n] is a solution vector to a problem instance of
size n. X[] and n are globally declared. The first (i –1) values
X[1], X[2], ..., X[i-1] of the solution vector X[1: n] have been
assigned.

Output: A final solution vector X[1: n] to a given problem
instance that satisfies a bounding function.*/
{
  for(each X[i] ∈ T(X[1], ..., X[i-1])
  {
    if (B_i(X[1], X[2], ..., X[i]) ≠ 0)
    {
      if ((X[1], X[2], ..., X[i]) is a path to an answer
      node)
```

```
        write (X[1: i]);
    if ( i < n )
        Backtrack_R(i + 1 ); //Recursive call
    }
}
```

- This recursive algorithm Backtrack_R produces all possible values for the i^{th} position of the solution-tuple X[1: n] that satisfy a bounding function B_i.
- The algorithm produces all the values, one by one, and appends them to the current partial solution vector $(x_1, x_2, ..., x_{i-1})$.
- Each time x_i is adjoined, the algorithm checks whether the desired solution has been found.
- On exit of *for* loop, as no more values for x_i are available, the current copy of Backtrack_R terminates.
- Then, the last unresolved call that continues to check remaining values assuming only (i – 2) values have been set is resumed.
- The algorithm can be updated to quit after getting a single solution.

5.1.3 Iterative Backtracking Method

- The backtracking algorithm can be implemented iteratively.
- A set T() includes all possible values of the first component x_1 for which $B_1(x_1)$ returns TRUE.
- It generates all values in a depth first way.

Control abstraction of an iterative backtracking algorithm

GQ. Write a schema for the iterative backtracking method. **(4 Marks)**

UQ. What is backtracking? Write a general iterative algorithm for backtracking.

(SPPU - May 15, Q. 6(a), 8 Marks)

GQ. Write a schema for iterative backtracking method.

(8 Marks)

```
Algorithm Backtrack_I(n)
// Input: X[1: n] is a solution vector to a problem instance of
size n. X[ ] and n are globally declared.
// Output: A final solution vector X[1: n] to a given problem
instance that satisfies a bounding function. All solutions in
X[1 : n] are printed, the moment, they are produced. */
{
    i := 1;
    while ( i ≠ 0 )
    {
        if (there remains an untried X[i] in T(X[1], ...,
            X[i –1]) && Bi(X[1], ..., X[i]) is true )
        {
            if((X[1], ..., X[i]) is a path to an answer
                node)
                write (X[1: i] );
                    i := i + 1;  /*Takes the next set for
                                    consideration. */
        }
        else i := i – 1;          /* Backtrack to the
                                    previous set . */
    }
}
```

- Algorithm Backtrack_I, continually increments i to expand the solution vector until either a complete solution-tuple is found or there is no untried value of candidate x_i remains.
- When i is decremented, the algorithm must continue the generation of untried values for the i^{th} position.

5.1.4 Complexity Analysis of the Backtracking Algorithms

- Following four performance metrics affect the efficiency of the backtracking algorithms:
  (1) Time to compute the next candidate solution X[i],
  (2) Number of candidate solutions X[i] fulfilling the explicit constraints,
  (3) The time taken by the bounding function B_i to check a promising candidate solution,
  (4) The number of candidate solutions X[i] satisfying the bounding function B_i.
- Among these metrics, the first three metrics are independent of an instance of a given problem and the last metric of the number of promising nodes varies with an instance of a given problem.
- For a problem instance of size n, if the backtracking algorithm produces a solution space of 2^n or n! nodes, then it has the worst-case running times $O(p(n)2^n)$ or $O(q(n)n!)$ respectively where p(n) and q(n) are polynomials in n.
- The backtracking algorithms give solutions to some larger problem instances in a very small amount of time.

5.2 8-QUEENS PROBLEM

GQ. Explain the backtracking method. What is the 8-queens problem? Give a solution to the 4-queens problem using the backtracking method. **(7 Marks)**

GQ. What do you mean by backtracking? Explain in brief. Discuss the eight queens problem using backtracking. **(7 Marks)**

It is a famous chess puzzle based on combinatorial logic. The efficient solution to this problem is given by the backtracking strategy. It is a classic example of a backtracking algorithm.

Problem description

- Place 8 queens on an 8×8 chessboard such that none of them can attack any other using the standard chess queen's moves. This implies that no two queens should be placed on the same row, column, or diagonal.
- Mathematically, it can be expressed as below:

  If 2 queens are placed at position (i, j) and (k, l) where i and k are the row indices and j and l are the column indices then,

  $i \neq k$ ....(No same row),

  $j \neq l$ ....(No same column) and

  $|i - k| \neq |j - l|$...(No same diagonal)

  where $i, j, k, l \in \{1, 2, ..., 8\}$.
- It is generalised to the n-queens problem that asks to place n queens on an $n \times n$ chessboard such that none of them can attack any other using the standard chess queen's moves.

The general procedure

(1) Identification of the data structures

- An 8×8 chessboard can be represented as a 2-D array of dimensions 8×8.
- Since no two queens can ever be placed on the same row, instead of a 2-D array a single dimensional array X of size 8 can be used to store 8-tuple $(x_1, x_2, ..., x_8)$. X[i] is the column number for an i^{th} queen placed on an i^{th} row, $1 \leq i \leq 8$.

(2) Identification of the explicit constraints

The explicit constraints using 8-tuple formulation are $x_i \in S_i = \{1, 2, ..., 8\}$, $1 \leq i \leq 8$. It has a solution space of 8^8 8-tuples.

(3) Identification of the implicit constraints

- Since all queens are to be placed essentially on different columns; no two x_i's can have the same value. All solutions are obtained by permutations of the 8-tuple (1, 2, ..., 8) and thus reducing the solution space of 8^8 to 8! tuples.
- Also, no two queens can ever be placed on the same diagonal.

(4) Building a solution using backtracking

- Place the queens on a chessboard row by row, starting at the top.
- $X[i] = c$, $1 \leq c \leq 8$ is a valid column number of the i^{th} queen placed on the i^{th} row. $X[i] = 0$ if the i^{th} queen is not yet placed on the i^{th} row (i.e. no valid assignment of a column number to the i^{th} queen).
- A partial solution is an array X[1: n] whose first k–1 entries are positive and whose last n–k +1 entries are all zeros, for some integer k.
- Construct a complete solution from the partial solutions that satisfy the bounding function.
- Discard the non-promising partial solutions, backtrack and check for an alternative path.
- Construct a state space tree by depth first node generation for the same.
- E.g. One of the solution to 8-queens problem is X = {3, 6, 2, 7, 1, 4, 8, 5} as depicted in Fig. 5.2.1.

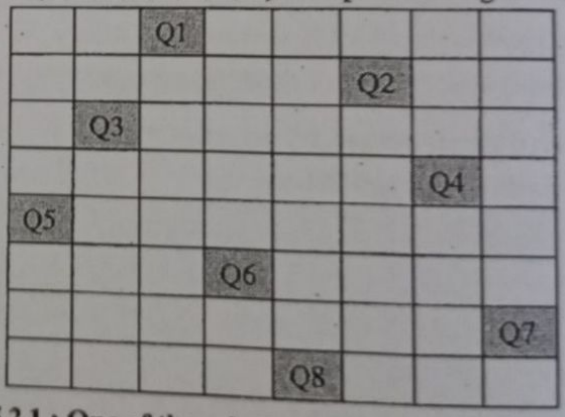


Fig. 5.2.1 : One of the solutions to the 8-queens problem

The state space tree of the 8-queens problem

- The solution space of the 8-queens problem by the backtracking strategy contains 8! permutations for an 8-tuple $(x_1, x_2, ..., x_8)$.
- It is represented by a state space tree that follows depth first node generation. This tree is also called a **permutation tree**. All paths from the root to leaf nodes describe the solution space.

- The edges in the tree are labelled by possible values of x_i. The nodes represent the problem states.
- The state space tree consists of 8! leaf nodes.

The basic steps

(1) Start assigning a position to the 1st queen on the 1st row and 1st column.

(2) Place the next queen on a particular column of the next row by checking the valid non-attacking position.

(3) Similarly, assign the valid non-attacking column position of each row to each queen.

(4) If such a valid column position is infeasible, discard that path without further exploration.

(5) Backtrack to the previous queen's column position and assign an alternative valid column position to that queen.

(6) Repeat steps (4) and (5) until the complete solution is found. If the desired solution is not found, then report the same.

GQ. Explain the 4 queen problem with one of the solutions. **(4 Marks)**

GQ. Draw the state space tree Diagram for the 4 Queen problem. **(4 Marks)**

- The 4-queens problem is about placing 4 queens on a 4 × 4 chessboard such that none of them can attack any other using the standard chess queen's moves. This implies that no two queens should be placed on the same row, column, or diagonal.
- It is efficiently solved by the backtracking algorithm.
- The solution space of the 4-queens problem by backtracking consists of 4! = 24 orderings for (x_1, x_2, x_3, x_4).
- Let X[1:4] = (x_1, x_2, x_3, x_4) be the 4-tuple solution vector to the 4-queens problem.
- There are two legal solutions X1 = (2,4,1,3) and X2 = (3,1,4,2) to the 4-queens problem. These solutions are shown in Fig. 5.2.2.

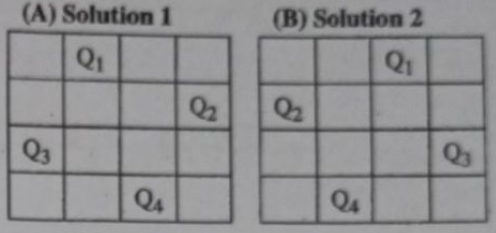
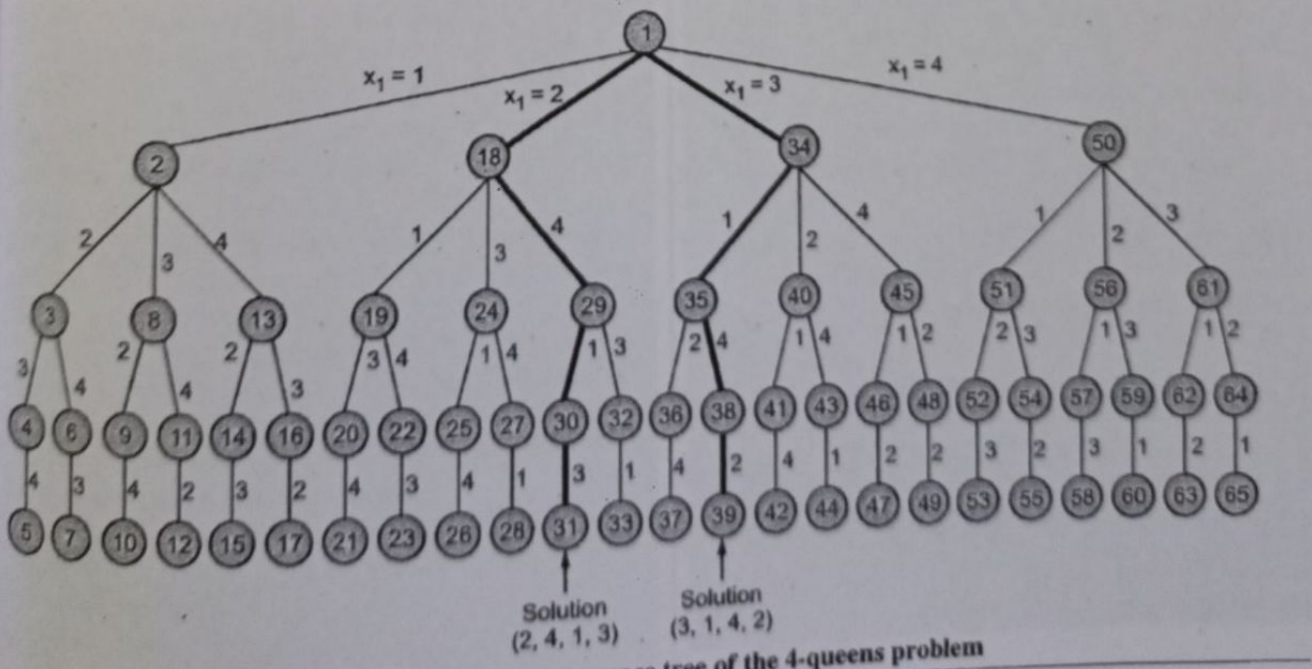


Fig. 5.2.2: Solutions to the 4-queens problem

The state space tree of the 4-queens problem

- The solution space of the 4-Queens problem by backtracking is represented by a state space tree.
- It is constructed by applying depth first node generation policy.
- It consists of 4! = 24 leaf nodes.
- The edges in the tree are labelled by possible values of x_i. The nodes represent the problem states.
- The state space tree of the 4-queens problem is depicted in Fig. 5.2.3. Here, the edges in the tree are labelled by possible column positions (values of x_i).



(1E28)Fig. 5.2.3 : The state space tree of the 4-queens problem

☞ Recursive backtracking algorithm

GQ. Write a short note on the 8 queens problem. Write an algorithm for the same. **(8 Marks)**

GQ. Write an iterative and recursive backtracking algorithm for the N-Queens problem. **(8 Marks)**

UQ. State the principle of backtracking and write backtracking algorithm for N-queens problem.

(SPPU - Q. 5(b), May 19, 8 Marks)

UQ. Write an iterative and recursive backtracking algorithm for N-Queens problem.

(SPPU - Q. 6(a), Dec. 19, 8 Marks)

GQ. Explain how backtracking strategy can be used to solve n-queens problem. Give pseudo code for the same. Discuss the time complexity for this problem. **(12 Marks)**

Algorithm NQueens_Bk_R(k, n)

/*It is a recursive backtracking algorithm to find all solutions to the n-queens problem.

Input: n is the number of queens to be placed on an n × n chessboard. k is the queen number that is to be placed on the chessboard currently.

Output: An n-tuple solution X[1: n] representing the valid column positions of n queens.*/

```
{
  k := 1; //Initialization
  for (i := 1;i ≤ n; i++)
  {
    if PlaceQ(k, i)                    /*if a bounding
                                         function is satisfied*/
    {
      X[k] := i;
      if (k = n)
        write (X[1: n]);
      else NQueens_Bk_R(k + 1, n);
    }
  }
}
```

Algorithm PlaceQ(k, i)

/* It is a bounding function for the n-queens problem.

Input: k is the queen number that is to be placed on the k^{th} row. i is the column number of the k^{th} queen. A solution vector X[1: n] is a global array whose first (k – 1) values are decided. X[1: n] is initialized with all 0 values. The function ABS(y) returns the absolute value of y.

Output: The algorithm returns TRUE if the k^{th} queen is placed in the k^{th} row and the i^{th} column on an n × n chessboard, otherwise returns FALSE.*/

```
{
  for (j := 1; j ≤ k –1; j++)
  {
    if ((X[j] = i) || (ABS(X[j] – i) = ABS(j – k)))
                                       /*Tests whether 2 queens are
                                         placed on the same column or on
                                         the same diagonal*/
      return (FALSE);
  }
  return (TRUE);
}
```

☞ An iterative backtracking algorithm

Algorithm NQueens_Bk_I(n)

/*It is an iterative backtracking algorithm to find all solutions to the n-queens problem.

Input: n is the number of queens to be placed on an n × n chessboard. A solution vector X[1: n] is a global array. X[1: n] is initialized with all 0 values.

Output: An n-tuple solution X[1: n] representing the valid column positions of n queens.*/

```
{
  k := 1;
  while (k ≠ 0)
  {
    X[k] := X[k] +1;
    if (X[k] ≤ n)
    {
      if PlaceQ(k, X[k])          /*if a bounding function
                                    is satisfied*/
      {
        if (k = n)
          write (X[1: n]);
        else k := k + 1;
      }
    }
    else
    {
      X[k] := 0;
      k := k – 1;
    }
  }
}
```

```
Algorithm PlaceQ(k, i)
/* It is a bounding function for the n-queens problem.
Input: k is the queen number that is to be placed on the k^th
row. i is the column number of the k^th queen. A solution vector
X[1: n] is a global array whose first (k – 1) values are decided.
X[1: n] is initialized with all 0 values. The function ABS(y)
returns the absolute value of y.
Output: The algorithm returns TRUE if the k^th queen is placed
in the k^th row and the i^th column on an n ×n chessboard,
otherwise returns FALSE.*/

for (j := 1; j ≤ k –1; j++)
 {
  if ((X[j] = i) || (ABS(X[j] – i) = ABS(j – k)))
                    /*Tests whether 2 queens are
                    placed on the same column or on
                    the same diagonal.*/
   return (FALSE);
 }
return (TRUE);
```

☞ Complexity analysis

GQ. Analyze the 8-queens problem using the backtracking strategy of problem-solving. **(4 Marks)**

- In backtracking a bounding function is applied to discard the infeasible partial solutions without exploring them further. It reduces the solution space.
- In the n-queens problem, no two queens can ever be placed on the same row or same column, or same diagonal. So, each placement of the i^{th} queen on a particular column on an i^{th} row leaves at most $(n - 1)$ possible column positions on an $(i+1)^{st}$ row for the next queen.
- Thus, the placement of every queen on a row develops another problem of size at most $(n - 1)$ to place the next queen in the next row.
- To test the legal placement of each queen on the chessboard, the algorithm PlaceQ() takes O(n) time. To test the valid column number of a queen on each row, at most n times PlaceQ() is invoked. So each placement of a queen in a row takes time $O(n^2)$.
- As total n queens are to be placed, the total time required to solve the n-queens problem by a recursive backtracking algorithm is given as

  $T(n) = nT(n-1) + n^2 = O(n!)$.
- By the brute force method, for an 8 × 8 chessboard, there are ${}^{64}C_8$ ways to place 8 queens. Thus, the brute force method can give all solutions to the 8-queens problem by testing the solution space of 4.4 billion 8-tuples.
- By rejecting non-promising candidate solutions, the backtracking algorithm examines at most 8! = 40,320 solution tuples. Thus, the backtracking algorithm is more effective than the brute force approach.

Ex. 5.2.1 : Find all possible solutions for the 5-queens problem using the backtracking method. **(7 Marks)**

☑ **Soln. :**

- Let $X[1:5] = (x_1, x_2, x_3, x_4, x_5)$ be a 5-tuple solution to the 5-queens problem. When an i^{th} queen Q_i is correctly placed on an i^{th} row and a j^{th} column of a 5×5 chessboard, $X[i] = x_i = j$.
- Start the placement of the first queen on the first row.

(1) **Place Q_1 on a position (1, 1) ⇒Valid placement ⇒ X[1] = 1.**

(2) Place Q_2 on a position (2, 1) ⇒ Invalid placement on the same column, so backtrack.

(3) Place Q_2 on a position (2, 2) ⇒ Invalid placement on the same diagonal, so backtrack.

(4) Place **Q_2 on a position (2, 3) ⇒ Valid placement ⇒ X[2] = 3.**

(5) Place Q_3 on a position (3, 1)⇒ Invalid placement on the same column, so backtrack.

(6) Place Q_3 on a position (3, 2)⇒ Invalid placement on the same diagonal, so backtrack.

(7) Place Q_3 on a position (3, 3)⇒ Invalid placement on the same column, so backtrack.

(8) Place Q_3 on a position (3, 4)⇒ Invalid placement on the same diagonal, so backtrack.

(9) Place **Q_3 on a position (3, 5) ⇒ Valid placement ⇒ X[3] = 5.**

(10) Place Q_4 on a position (4, 1)⇒ Invalid placement on the same column, so backtrack.

(11) Place **Q_4 on a position (4, 2) ⇒Valid placement ⇒ X[4] = 2.**

(12) Place Q_5 on a position (5, 1)⇒ Invalid placement on the same column and same diagonal, so backtrack.

(13) Place Q_5 on a position (5, 2)⇒ Invalid placement on the same column, so backtrack.

(14) Place Q_5 on a position (5, 3)⇒ Invalid placement on the same column and same diagonal, so backtrack.

(15) Place **Q_5 on a position (5, 4) ⇒ Valid placement ⇒ X[5] = 4.**

- Thus, one solution to the 5-queens problem is X1[1:5] = (x_1, x_2, x_3, x_4, x_5) = (1, 3, 5, 2, 4). It is shown in Fig. Ex. 5.2.1.
- Similarly, other solutions to the 5-queens problem can be determined. These solutions are depicted in Fig. Ex. 5.2.1(a).

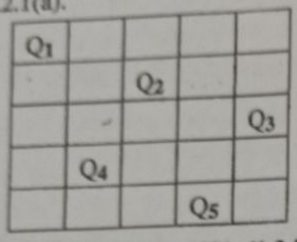
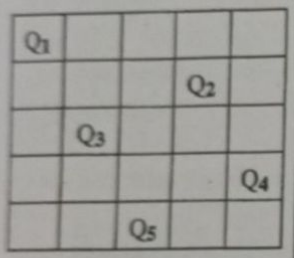
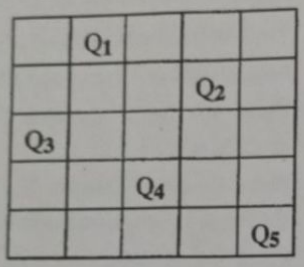


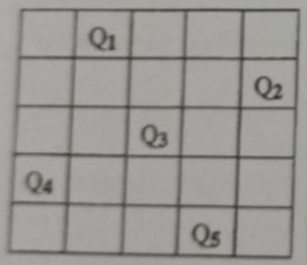
Fig. Ex. 5.2.1: Solution1X1=(1,3,5,2,4) to the 5-queens problem



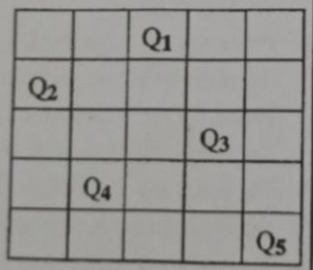
(i) Solution2 X2 = (1,4,2,5,3)



(ii) Solution3 X3 = (2,4,1,3,5)

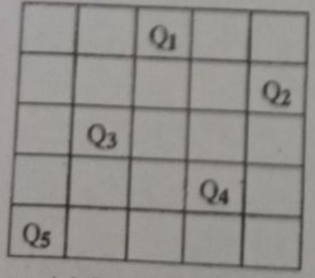


(iii) Solution4 X4 = (2,5,3,1,4)

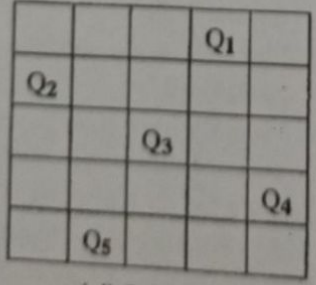


(iv) Solution5

X5 = (3,1,4,2,5)



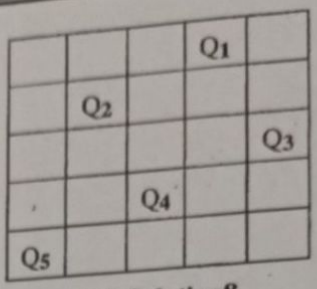
(v) Solution6 X6 = (3,5,2,4,1)



(vi) Solution7

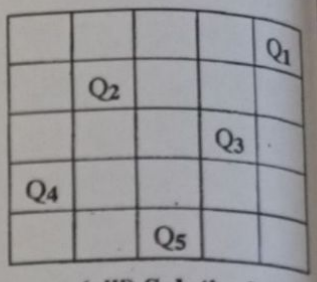
X7 = (4,1,3,5,2)

Fig. Ex. 5.2.1(a) (Contd....)



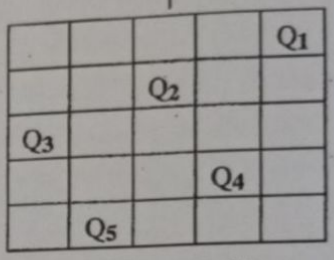
(vii) Solution8

X8 = (4,2,5,3,1)



(viii) Solution9

X9 = (5,2,4,1,3)



(ix) Solution10

X10 = (5,3,1,4,2)

Fig. Ex. 5.2.1(a) : Other solutions to the 5-queens problem

5.3 GRAPH COLORING PROBLEM

The graph colouring is a classic problem in combinatorics. There are several variants of coloring problems. It is efficiently solved by backtracking strategy.

Problem description

GQ. What is m-colorability optimization problem? Explain with an example.

GQ. What are planar graphs? Explain graph coloring.

- Consider G = (V, E) be an undirected graph where V is a set of nodes and E is a set of edges.
- Then the graph coloring problem asks to assign m colors to the vertices of G such that no two neighboring vertices have a similar color. It is also called a **vertex coloring problem.**
- If the same problem is applied to color the edges of a graph G so that any two adjacent edges are colored with different colors, then it is known as an **edge coloring problem.**

Planar graph : A graph G is called as a planar graph iff it is drawn in a plane such that any two edges intersect only at their endpoints without crossing each other.

Region or face coloring problem : It colors all the regions or faces of a planar graph so that any two regions sharing a boundary have different colors.

m-colorability decision problem : It is a decision problem that checks whether all the vertices or edges or regions of G can be colored using only m colors such that no two neighboring vertices or edges or regions have the same color, where G is a given graph and m is a given positive integer.

4-coloring problem : It is a popular specific case of an m-colorability decision problem that checks whether all the regions of a given map can be colored using only 4 colors such that no two neighboring regions in the map have the same color.

m-colorability optimization problem : It is an optimization problem that asks to determine the minimum number of colors m required to color all the vertices or edges or regions of G such that no two neighboring vertices or edges or regions have the same color, where G is a given graph.

A chromatic number of a graph G is the smallest integer m that is required to color a graph G.

The general procedure

(1) Construction of a planar graph from a given map :

- Represent the regions of a given map as the vertices in the corresponding graph.
- If any two regions are sharing a boundary in a given map, then add an edge between the corresponding vertices in the graph.

(2) Identification of a data structure to represent a graph :

- Let G = (V, E) is a given graph where V is a set of n vertices and E is a set of edges in it.
- Represent a given graph G as an adjacency matrix G[1:n,1:n] where G[i, j] = 1 if an edge <i, j>∈ E and i, j∈ V; G[i,j] = 0 if an edge <i, j>∉ E and i, j∈ V.

(3) Identification of explicit constraints :

- The colors are given by the numbers 1, 2, …., m.
- The solution vector $X = (x_1, x_2, \ldots, x_n)$ represents an n-tuple where x_i gives the assigned color to a node i.
- For valid color assignment, $x_i \in \{c \mid c$ is an integer and $1 \le c \le m\}$; otherwise $x_i = 0$.

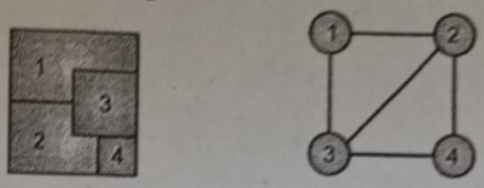
(4) Identification of implicit constraints :

No two adjacent vertices are assigned the same color. So, if an edge <i, j>∈ E and i, j∈ V then $x_i \neq x_j$.

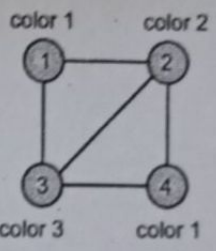
(5) Building a solution using backtracking :

- Start coloring from node1 by assigning it the first color.
- A partial solution is an array X[1:n] whose first k–1 entries are determined, for some integer k.
- Construct a complete solution from the partial solutions that satisfy the bounding function.
- Then bounding function B_k can be given as:

  $B_k(x_1, x_2, \ldots, x_k)$ = TRUE iff G[k, j] = 1 and (X[k] ≠ X[j]), $1 \le j \le n$ for some integer $k \le n$.
- Discard the non-promising partial solutions, backtrack and check for an alternative solution.
- Construct a state space tree by depth first node generation for the same.
- E.g., Consider an example of the m-coloring problem as depicted in Fig. 5.3.1.



(1E30)(a) A map (1E31) (b) Corresponding planar graph



(1E32)(c) : Solution with m = 3, X = (1, 2, 3, 1)

Fig. 5.3.1 : An example of the m-coloring problem

The state space tree of the graph coloring problem

- The solution space of the graph coloring problem is represented by a state space tree that follows the depth first node generation.

The basic steps

(1) Select any single node (generally node1) of a given graph and assign it the colour number1.

(2) Then select remaining nodes of a graph one by one and assign the higher numbered color to them from the list of available colors.

(3) For each color assignment apply the bounding function to check whether two adjacent nodes have different colors.
(4) If the partial solution is infeasible, discard that path without further exploration.
(5) Backtrack to the previous level and assign another valid color to the node.
(6) Repeat step (2) to (5) till the complete solution is found. If the desired solution is not found, then report the same.

☞ Recursive backtracking algorithm

GQ. Write an algorithm for graph coloring problem using backtracking method.

GQ. Write a recursive backtracking algorithm for m-colorings of the graph.

GQ. Write a recursive backtracking schema for m-colorings of the graph.

```
Algorithm mColorability_Bk(k)
/* It solves the m-colorability decision problem by recursive
backtracking approach.
Input: Let m is the number of available colors to color a given
graph G=(V, E) with n vertices.
Integers m and n are globally defined. A graph G is
represented as a boolean adjacency matrix G[1:n,1:n] where
G[i, j]=1 if an edge <i, j>∈E and i, j∈V; G[i, j]=0 if an
edge <i, j>∉E and i, j∈V. A fixed-size solution vector X[1:n]
is a global array whose first (k-1) values are decided. k is the
next node to be colored.
Output: A fixed-size solution vector X[1:n] that gives all legal
color assignments to n nodes so that no two adjacent vertices
are assigned the same color.*/
{
  repeat
  {
      /*Give all legal color assignments to X[k].*/
    NextColor(k);          /*Check a bounding function
                           to assign a legal color to
                           X[k].*/
    if (X[k] = 0)
      return;              /* No new color is
                              feasible.*/
    if (k = n)    /* n vertices are colored by using at
                     most m colors.*/
      write(X[1:n]);       /* The desired solution. */
    else
      mColorability_Bk(k+1);
  }
  until(FALSE);
}
```

```
Algorithm NextColor(k)
/* It is a bounding function for m-colorability problem.
Input: k is the node number that is to be colored. A solution
vector X[1:n] is a global array whose first (k-1) values in the
range [1,m] are decided.
Output: The desired value of X[k] is decided from the
range[0,m]. X[k] is assigned the highest-numbered valid color
so that a node k gets a distinct color from the colors of its
neighboring nodes. If no such valid color is available, then
X[k]=0.*/
{
  repeat
  {
    X[k]:=(X[k] +1) %(m+1);
                  /*Assignment of next highest color. */
    if (X[k]=0)
      return;
                  /* All colors have been assigned.*/
    for (j:= 1; j≤ n; j++)      {
                  /*Check whether the assigned color
                  to a node k is distinct color from the
                  colors of its neighboring nodes.*/
      if((G[k,j]≠0) && (X[k]=X[j]))
                  /* if there is an edge <k, j>∈E and
                  nodes k and j are assigned the same
                  color the discard the solution*/
        break;
    }
    if (j = n+1)
      return;
                  /* New color is found for node k*/
  } until (FALSE);
}
```

Complexity analysis

Q2. Discuss and analyze the problem of graph coloring using backtracking with the help of an example.

- Consider a given graph G has n vertices and at most m colors are available to colors it.
- In the state space tree of m-colorability decision problem, at level i, the tree has m^i nodes representing problem states.
- So, the total number of internal nodes in a state space tree is $\sum_{0 \le k \le n-1} m^k$.
- For checking legal color assignment at each internal node, the bounding function NextColor() takes O(mn) time. The bounding function checks the feasibility of solution at each internal node.
- There are total $\sum_{0 \le k \le n-1} m^k$ internal nodes. So, the total time taken by an algorithm is:

$$T(n) = \sum_{0 \le k \le n-1} m^{k+1} n = \sum_{1 \le k \le n} m^k n$$
$$= n(m^{n+1} - 2)/(m-1) = O(nm^n)$$

- Thus, the graph coloring algorithm by backtracking has an exponential time complexity $O(nm^n)$.

Examples

Ex. 5.3.1 : Construct planar graph for the following map. Explain how to find m-colorings of this planar graph by using m-colorings backtracking algorithm. **(8 Marks)**

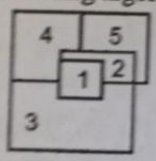
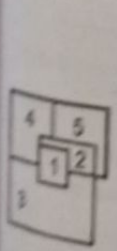


Fig. Ex. 5.3.1

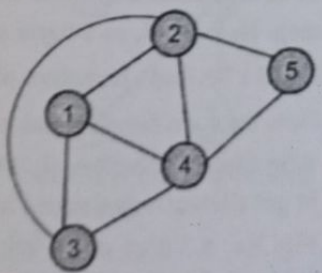
Soln. :

Step 1 : Construction of a planar graph :

Fig. Ex. 5.3.1(a) shows a planar graph corresponding to a given map.



(i) A given map planar



(ii) A corresponding graph

Fig. Ex. 5.3.1(a)

Step 2 : To find m for a planar graph :

- For the given instance of m-coloring problem, we have to find m = number of colors needed to color the graph, so that no two adjacent nodes have the same colour.
- We will construct a state space tree for the same problem instance.
- Let x_i represent the color number assigned to node i, $1 \le i \le 5$ as the number of nodes n = 5.
- Each edge in the state space tree is labelled by value of x_i (i.e. assigned color number).
- Let us start with node 1 and assign color number 1 to it.
- Then consider nodes 2, 3, 4 and 5 one after another.
- For each node, assign the colour by testing whether the adjacent nodes have different colours.
- Initially, consider only 1 colour and then introduce a new colour as and when needed to colour the adjacent nodes with different colours.
- Fig. Ex. 5.3.1(b) shows the portion of the state space tree to find 'm' for the given graph colouring instance. Label 'K' indicates that a node is killed by a bounding function.

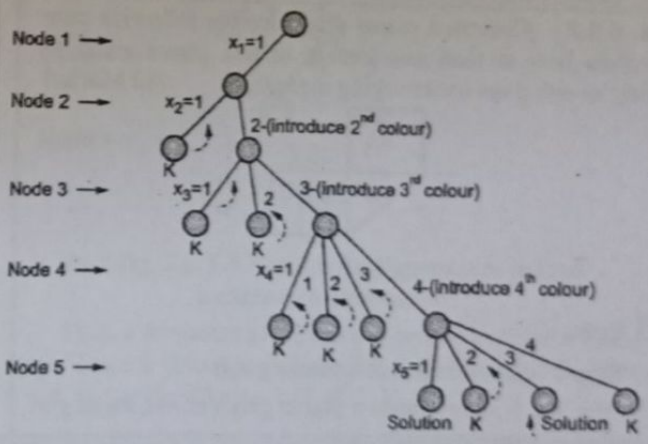


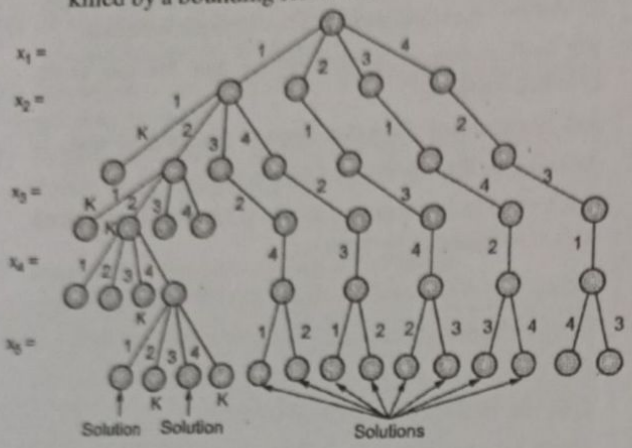
Fig. Ex. 5.3.1(b) : A state space tree to find m for UEx. 5.3.1

- Thus a minimum of 4 colors is needed to color a given map. Here n = 5, so 5-tuple solution is (1, 2, 3, 4, 1) or (1, 2, 3, 4, 3).

Step 3 : To find alternative solutions to m-colorability :

- Now once we have a value of m = 4, we can construct a state space tree with each internal node having degree 4 to get alternative solutions to color the given map.

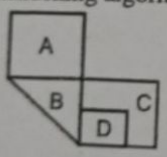
- Fig. Ex. 5.3.1(c) shows the portion of the state space tree with m = 4. Label 'K' indicates that a node is killed by a bounding function.



(1E8)**Fig. Ex. 5.3.1(c) : The portion of a state space tree with m = 4**

- Some 5-tuple solutions are : (1, 2, 3, 4, 1), (1, 2, 3, 4, 3), (1, 3, 2, 4, 1), (1, 3, 2, 4, 2), (1, 4, 2, 3, 1), (1, 4, 2, 3, 2), (2, 1, 3, 4, 2), (2, 1, 3, 4, 3), (3, 1, 4, 2, 3), (3, 1, 4, 2, 4), (4, 2, 3, 1, 4), (4, 2, 3, 1, 3) and so on.

Ex. 5.3.2 : Construct planar graph for the following map. Explain how to find m-colorings of this planar graph by using m-colorings backtracking algorithm. **(10 Marks)**

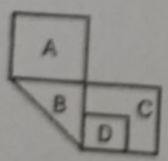


(1E9)**Fig. Ex. 5.3.2**

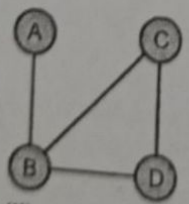
☑ **Soln. :**

▶ **Step 1 :** Construction of a planar graph

- Fig. Ex. 5.3.2(a) shows a planar graph corresponding to a given map.



(1E10)**(i) : A given map**

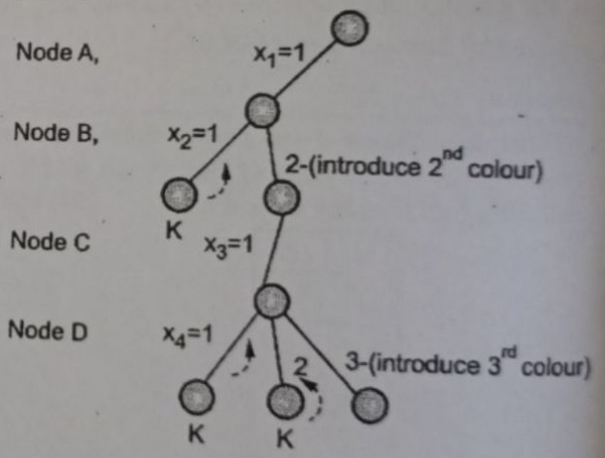


(1E11)**(ii) : A corresponding planar graph**

Fig. Ex. 5.3.2(a)

▶ **Step 2 :** To find m for a planar graph :

- For the given instance of an m-coloring problem, we have to first find m i.e. number of colours needed to colour the graph, so that no two adjacent nodes have the same colour.
- We will construct a state space tree.
- Let x_i represent the colour number assigned to node i, $1 \le i \le 4$ as n = 4 for this instance.
- Each edge in the state space tree is labelled by value x_i(i.e. assigned color number).
- Let us start with node A and assign colour number 1 to it.
- Then consider nodes B, C and D one after another.
- For each node, assign the colour by testing whether the adjacent nodes have different colours.
- Initially, consider only 1 colour and then introduce a new colour as and when needed to colour the adjacent nodes with different colours.
- Fig. Ex. 5.3.2(b) shows the portion of the state space tree to find 'm' for the given graph colouring instance. Label 'K' indicates that a node is killed by a bounding function.

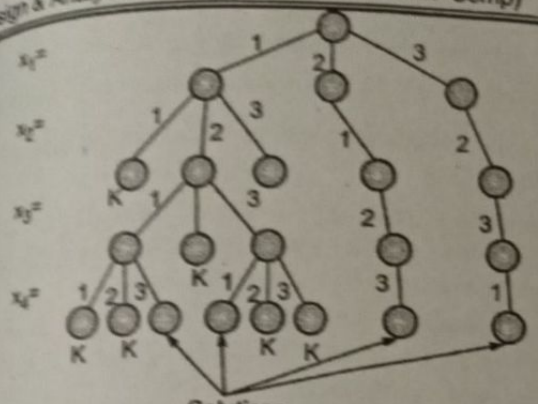


(1E12)**Fig. Ex. 5.3.2(b) : A state space tree to find m**

- Thus a minimum of 3 colors is needed to color a given map. Here n = 4, so 4-tuple solution is (1, 2, 1, 3).

▶ **Step 3 :** To find alternative solutions to m-colorability

- Now once we have a value of m = 3, we can construct a state space tree with each internal node having degree 3 to get alternative solutions to color the given map.
- Fig. Ex. 5.3.2(c) shows the portion of the state space tree with m = 3. Label 'K' indicates that a node is killed by a bounding function.



(1,2,1,3), (1,2,3,1), (2,1,2,3), (3,2,3,1),

(IE15)Fig. Ex. 5.3.2(c) : The portion of state space tree with m = 3

- Some 4-tuple solutions are : (1, 2, 1, 3), (1, 2, 3, 1), (2, 1, 2, 3), (3, 2, 3, 1) and so on.

Ex. 5.3.3 : What is a graph coloring problem? Define a chromatic number of a graph. Find the chromatic number of the following map and draw the necessary state space tree.

(8 Marks)

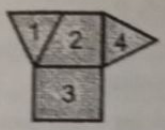


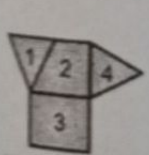
Fig. Ex. 5.3.3

☑ **Soln. :**

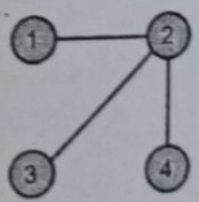
- **A graph coloring problem :** Refer to a problem description under section 5.3.
- **A chromatic number of a graph :** Refer to a definition given in problem description under section 5.3.

Step 1 : Construction of a planar graph :

- Fig. Ex. 5.3.3(a) shows a planar graph corresponding to a given map.



(IE16) (i) A given map planar

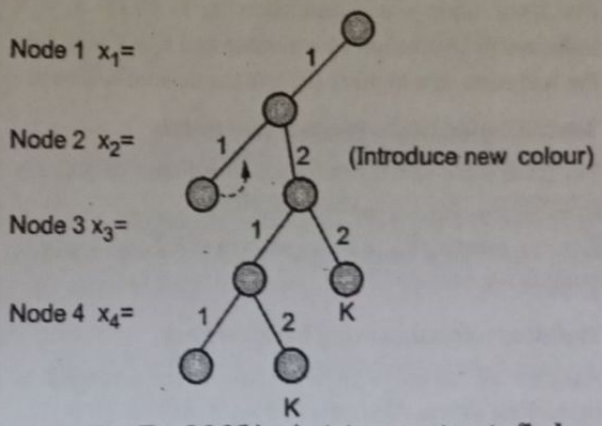


(IE16)(ii) A corresponding graph

Fig. Ex. 5.3.3(a)

Step 2 : To find a chromatic number m for a planar graph

- For the given instance of the m-coloring problem, we have to find m = number of colors needed to color the graph, so that no two adjacent nodes have the same colour.
- We will construct a state space tree for the same problem instance.
- Let x_i represent the colour number assigned to node i, $1 \le i \le 4$ as the number of nodes n = 4.
- Each edge in the state space tree is labelled by value of x_i (i.e. assigned color number).
- Let us start with node 1 and assign colour number 1 to it.
- Then consider nodes 2, 3 and 4 one after another.
- For each node, assign the colour by testing whether the adjacent nodes have different colours.
- Initially, consider only 1 colour and then introduce a new colour as and when needed to colour the adjacent nodes with different colours.
- Fig. Ex. 5.3.3(b) shows the portion of the state space tree to find 'm' for the given graph colouring instance. Label 'K' indicates that a node is killed by a bounding function.



(IE17)Fig. Ex. 5.3.3(b) : A state space tree to find a chromatic number m

Thus, a minimum of 2 colors is needed to color a given map. Hence a chromatic number of a given map is 2. Here n = 4, so 4-tuple solution is (1, 2, 1, 1).

5.4 SUM OF SUBSETS PROBLEM

Problem description

- Consider $W = (w_1, w_2, \ldots, w_n)$ is a set of n positive numbers or weights, $n > 0$. Let m is another positive number. The sum of subsets problem is to determine all subsets of W whose sums are m.

The general procedure

(1) Formulations of the sum of subsets problem

- There are two formulations of the sum of subsets problem: (1) Variable tuple size formulation and (2) Fixed tuple size formulation.

(i) Variable tuple size formulation

- An edge from a level i node to a level i + 1 node in a state space tree represents value for $x_i \in W$.
- All solutions are k-tuples, $1 \le k \le n$.

(ii) Fixed tuple size formulation

- An edge from a level i node to a level i + 1 node in a state space tree represents value for x_i, $x_i \in \{0,1\}$.
- The left child of a level i node corresponds to the inclusion of an item ($x_i = 1$) and its right child corresponds to the exclusion of an item ($x_i = 0$).
- All solutions are n-tuples.
- 2^n leaf nodes represent all possible tuples.
- We consider fixed tuple size formulation for further discussion.

(2) Identification of the explicit constraints

For fixed tuple size formulation: $x_i \in \{0,1\}$. $x_i = 0$ indicates the exclusion of a number and $x_i = 1$ indicates the inclusion of a number to form the desired solution.

(3) Identification of the implicit constraints

- For fixed tuple size formulation, all solution subsets are n-tuples $(x_1, x_2, \ldots, x_n)$ and $x_i \in \{0,1\}$.
- $\sum_{1 \le i \le k} w_i x_i + \sum_{k+1 \le i \le n} w_i \ge m$ && $\sum_{1 \le i \le k} w_i x_i + w_{k+1} \le m$.

(4) Building a solution using backtracking

- Assume all numbers in a given set are arranged in increasing order. Also, assume $w_1 \le m$ and $\sum_{1 \le i \le n} w_i \ge m$.
- A partial solution is an array X[1: n] whose first k–1 entries are determined, for some integer k.
- Construct a complete solution from the partial solutions that satisfy the bounding function.
- Then bounding function B_k can be given as :

$B_k(x_1, x_2, \ldots, x_k)$ = **TRUE iff** $\sum_{1 \le i \le k} w_i x_i + \sum_{k+1 \le i \le n} w_i \ge m$ **and** $\sum_{1 \le i \le k} w_i x_i + w_{k+1} \le m$.

- Discard the non-promising partial solutions, backtrack and check for an alternative path.
- Construct a state space tree by depth first node generation for the same.
- E.g. W = (7, 24, 11, 13) and m = 31. Then the variable size solution to the sum of subsets problem is (7, 24) and (7, 11, 13). The solution can also be represented by the indices of the numbers as (1, 2) and (1, 3, 4) respectively. The fixed size solution tuple for the same problem can be given as (1, 1, 0, 0) and (1, 0, 1, 1) respectively where 1 represents the inclusion of an i^{th} number and 0 represents the exclusion of an i^{th} number.

☞ The state space tree of the sum of subsets problem

- The solution space of the sum of subsets problem is represented by a state space tree that follows depth first node generation.
- In the variable size tree formulation, edges in the tree are labelled by possible values of x_k, $1 \le k \le n$. The nodes represent the current sum.
- In the fixed size tree formulation, edges in the tree are labelled by $x_k = 0$ for the exclusion of a k^{th} number and $x_k = 1$ for the inclusion of a k^{th} number, $1 \le k \le n$.

▶ The basic steps : (fixed size tree formulation)

(1) Select the numbers in a given set one by one.
(2) Apply the bounding function to check the feasibility of a resultant partial solution.
(3) If it satisfies the bounding function then create a left child by including it ($x_k = 1$).
(4) If the partial solution is infeasible, discard that path without further exploration.
(5) Backtrack to the previous level then create a right child excluding that number ($x_k = 0$).
(6) Repeat steps (1) to (5) till the complete solution is found. If the desired solution is not found, then report the same.

☞ Recursive backtracking algorithm

GQ. Write a recursive backtracking algorithm for the sum of subsets problem.

```
Algorithm SumOfSubsets_Bk(s, k,r)
/* The algorithm gives all subsets of W[1:n]that sum to m; All
numbers in a set W are arranged in increasing order.
Input: A fixed size solution vector X[1:n] is a global array
whose first (k-1) values are decided. X[1:n] is initialized with
all 0 values. Consider W[1] ≤ m and Σ(1≤i≤n) W[i] ≥ m.
s= Σ(1≤i≤k-1) W[i] *X[i] and r= Σ(k≤i≤n) W[i].
Output: A fixed size solution vector X[1:n] that determines the
desired solution to the sum of
subsets problem.*/
{
//Generate left child.
   X[k] := 1;                    // Includes the number
   if (s+W[k] = m)
```

```
    write (X[1:n] );
                    /* Subset is found. No recursive call
                       is made as W[i]>0, 1 ≤ i ≤ n*/
else if (s+W[k]+W[k+1] ≤ m)
SumOfSubsets_Bk(s+W[k], k+1, r-W[k]);
    /* Generate right child and check the bounding
    function Bk .*/
if ((s+r-W[k] ≥ m) && (s+W[k+1] ≤ m))
{
  X[k] := 0;          /*Excludes the number.*/
  SumOfSubsets_Bk(s, k+1, r-W[k]);
}
```

Complexity analysis

- In a state space tree of fixed size formulation of the sum of subsets problem, at level i, the tree has 2^i nodes representing problem states.
- So, for n numbers, the total number of internal nodes in a state space tree is $\sum_{0 \le k \le n} 2^k$.
- The bounding function is applied to each problem state to check the feasibility of a partial solution.
- Thus, the sum of subsets problem runs in exponential order $O(2^n)$.

Examples

> Note : Examples can be solved either by variable size or fixed size formulation of a state space tree of the sum of subsets problem.

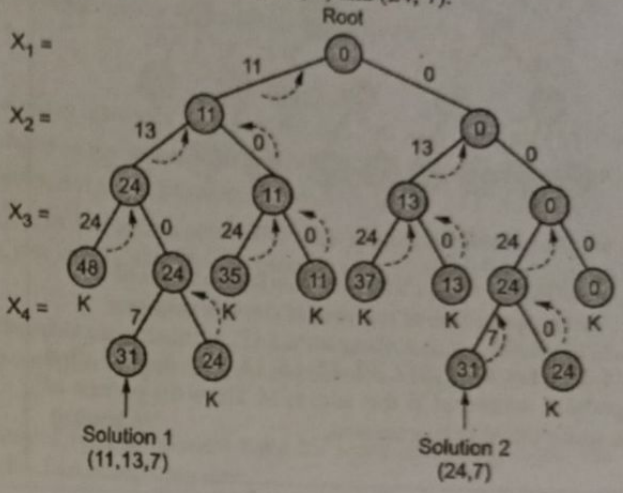
Ex. 5.4.1 : What is the backtracking method of algorithmic design? Solve the sum of subsets problem using backtracking algorithmic strategy for the following data: N = 4; (w1, w2, w3, w4) = (11, 13, 24, 7) and M = 31. **(12 Marks)**

Soln. :

- To solve the given instance of the sum of subsets problem using backtracking we use the following bounding function:

$B_k(x_1,x_2,\ldots,x_k)$ = **TRUE iff** $\sum_{1 \le i \le k} w_i x_i + \sum_{k+1 \le i \le n} w_i \ge m$ and $\sum_{1 \le i \le k} w_i x_i + w_{k+1} \le m$.

- We draw a **variable size formulation** of a state space tree where value of $x_k, 1 \le k \le n$ is assigned to the edges of a tree.
- The left child is corresponding to the inclusion of the k^{th} item while the right child is corresponding to its exclusion.
- The state space tree is shown in Fig. Ex. 5.4.1. The nodes marked by 'K' are the nodes killed by a bounding function.
- It has 2 solutions. The variable size solutions to a given instance of the sum of subsets problem using backtracking are (11, 13, 7) and (24, 7).



(1E1)Fig. Ex. 5.4.1 : Variable size formulation of a state space tree of the sum of subsets problem

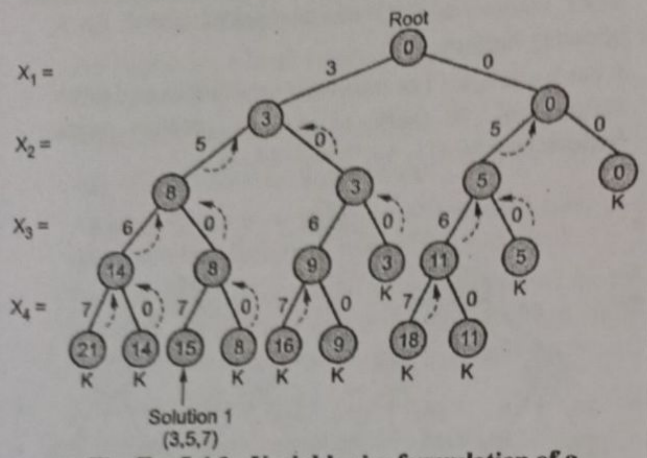
Ex. 5.4.2 : Differentiate between backtracking and branch and bound. Draw state space tree for a given sum of subsets problem: Set of elements = {3, 5, 6, 7} and d = 15.**(8 Marks)**

Soln. :

- **Difference between backtracking and branch and bound : Refer to section 6.1.2.**
- To solve the given instance of the sum of subsets problem using backtracking we use the following bounding function :

$B_k(x_1,x_2,\ldots,x_k)$ = **TRUE iff** $\sum_{1 \le i \le k} w_i x_i + \sum_{k+1 \le i \le n} w_i \ge m$ **and** $\sum_{1 \le i \le k} w_i x_i + w_{k+1} \le m$.

- We draw a **variable size formulation** of a state space tree where the value of $x_k, 1 \le k \le n$ is assigned to the edges of a tree.
- The left child is corresponding to the inclusion of the k^{th} item while the right child is corresponding to its exclusion.
- The state space tree is shown in Fig. Ex. 5.4.2. The nodes marked by 'K' are the nodes killed by a bounding function.
- It has only one solution. **The variable size solution to a given instance of the sum of subsets problem using backtracking is : (3, 5, 7).**



(1E2)**Fig. Ex. 5.4.2 : Variable size formulation of a state space tree of the sum of subsets problem**

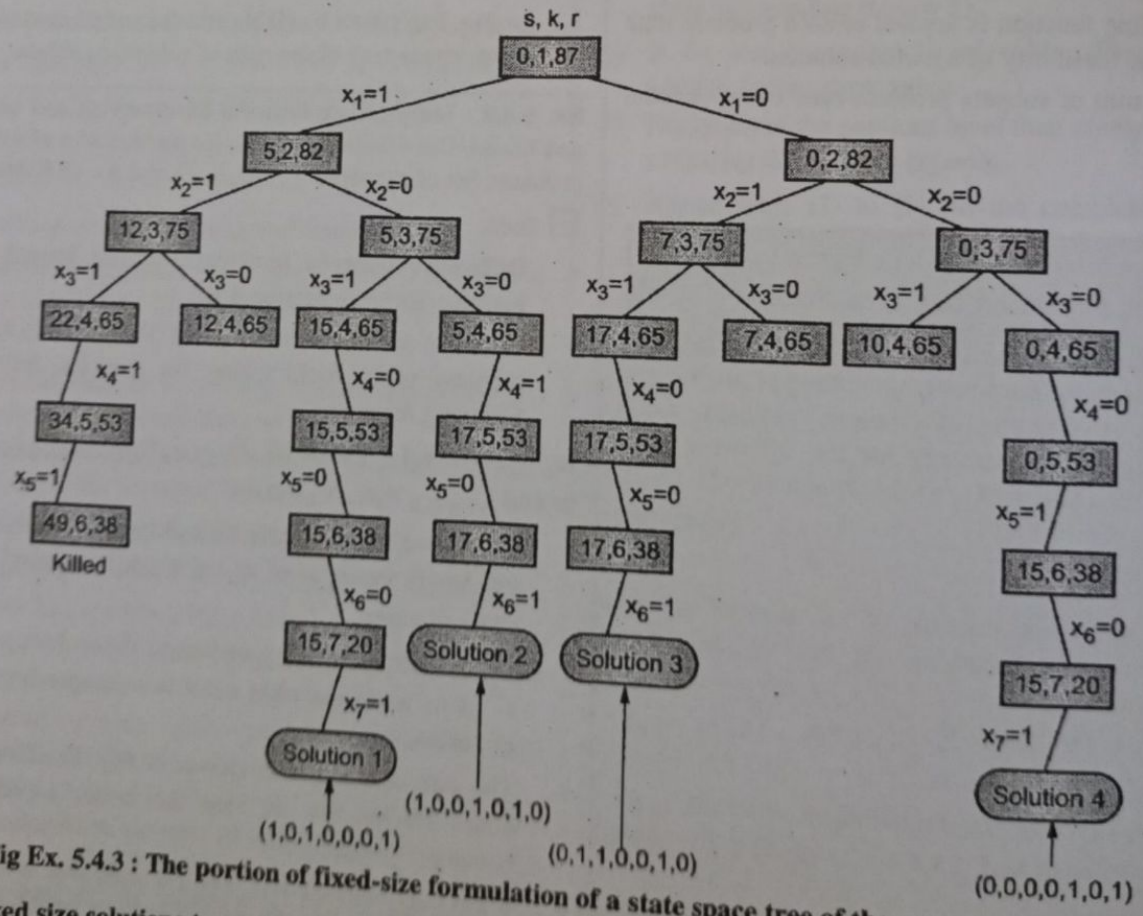
Ex. 5.4.3 : Let W = {5, 7, 10, 12, 15, 18, 20}, M = 35 Find all possible subsets of W that sum to M. Draw the portion of state space tree that is generated. **(8 Marks)**

☑ **Soln. :**

- To solve the given instance of the sum of subsets problem using backtracking we use the following bounding function :

$B_k(x_1,x_2,\ldots,x_k)$ = **TRUE iff** $\sum_{1 \le i \le k} w_i x_i + \sum_{k+1 \le i \le n} w_i \ge m$ **and** $\sum_{1 \le i \le k} w_i x_i + w_{k+1} \le m$.

- We consider $W[1] \le m$ and $\sum_{1 \le i \le n} W[i] \ge m$. $s = \sum_{1 \le i \le k-1} W[i] * X[i]$ and $r = \sum_{k \le i \le n} W[i]$.
- We draw a **fixed size formulation** of a state space tree where the value of $x_k \in \{0,1\}, 1 \le k \le n$ is assigned to the edges of a tree.
- The left child is corresponding to the inclusion of the k^{th} item ($x_k = 1$) while the right child is corresponding to its exclusion ($x_k = 0$).
- The portion of a state space tree is shown in Fig. Ex. 5.4.3.



(1E3)**Fig Ex. 5.4.3 : The portion of fixed-size formulation of a state space tree of the sum of subsets problem**

- **The fixed size solutions to a given instance of sum of subsets problem using backtracking are : (1, 0, 1, 0, 0, 0, 1), (1, 0, 0, 1, 0, 1, 0), (0, 1, 1, 0, 0, 1, 0), (0, 0, 0, 0, 1, 0, 1).**

Ex. 5.4.4 : Analyze sum of subsets algorithm on data : M = 35 and
(a) W = {5, 7, 10, 12, 15, 18, 20} (b) W = {20, 18, 15, 12, 10, 7, 5}
(c) W = {15, 7, 20, 5, 18, 10, 12}
Are there any discernible differences in computing time ?

Soln. : **(10 Marks)**

Refer to the solution to Ex. 5.3.3 for W = {5, 7, 10, 12, 15, 18, 20}. Similarly, the solutions for W = {20, 18, 15, 12, 10, 7, 5} and W = {15, 7, 20, 5, 18, 10, 12} can be obtained by generation of the corresponding state space trees.

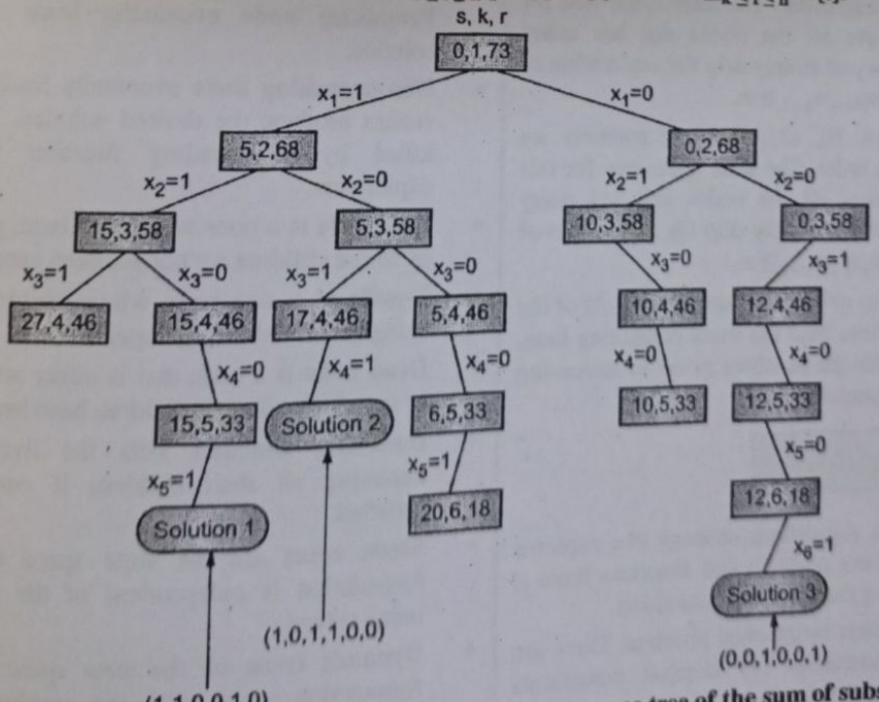
Differences in computing time

- There will be differences in computing time for all the three given instances.
- (a) W = {5, 7, 10, 12, 15, 18, 20}: Here the numbers are arranged in increasing order. So if $\sum_{1 \le i \le k} w_i x_i + w_{k+1} \ge m$ then there is no need to explore a path with $x_{k+1} = 1$. So all further nodes are not explored at all.
- (b) W = {20, 18, 15, 12, 10, 7, 5}: Here the numbers are arranged in decreasing order. The state space tree for this instance explores all the nodes and has many backtracks. It cannot just simply skip the exploration of $x_{k+1} = 1$ if $\sum_{1 \le i \le k} w_i x_i + w_{k+1} \ge m$.
- (c) W = {15, 7, 20, 5, 18, 10, 12}: Here the numbers are arranged in random order. The state space tree for this instance also explores all the nodes and has many backtracks. It cannot just simply skip the exploration of $x_{k+1} = 1$ if $\sum_{1 \le i \le k} w_i x_i + w_{k+1} \ge m$.
- Thus, the solutions to given instances (b) and (c) of the sum of subsets problem have the worst computing time. The instance (c) with the numbers given in increasing order has the best computing time.

Ex. 5.4.5 : 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. **(8 Marks)**

Soln. :

- To solve the given instance of the sum of subsets problem using backtracking we use the following bounding function :
  $B_k(x_1, x_2, \ldots, x_k)$ = **TRUE iff** $\sum_{1 \le i \le k} w_i x_i + \sum_{k+1 \le i \le n} w_i \ge m$ **and** $\sum_{1 \le i \le k} w_i x_i + w_{k+1} \le m$.
- We consider $W[1] \le m$ and $\sum_{1 \le i \le n} W[i] \ge m$. $s = \sum_{1 \le i \le k-1} W[i] * X[i]$ and $r = \sum_{k \le i \le n} W[i]$.



(1E4) **Fig. Ex. 5.4.5 : The portion of fixed-size formulation of state space tree of the sum of subsets problem**

- We draw a **fixed size formulation** of a state space tree where the value of $x_k \in \{0,1\}, 1 \leq k \leq n$ is assigned to the edges of a tree.
- The left child is corresponding to the inclusion of the k^{th} item ($x_k = 1$) while the right child is corresponding to its exclusion ($x_k = 0$).
- The portion of a state space tree is shown in Fig. Ex. 5.4.5.
- **The fixed size solutions to a given instance of sum of subsets problem using backtracking are: (1, 1, 0, 0, 1, 0), (1, 0, 1, 1, 0, 0), (0, 0, 1, 0, 0, 1).**

Ex. 5.4.6 : If M = 30, given data set W = {5, 10, 12, 13, 15, 18} find all possible subsets of W that sum to M. Draw the portion of state space tree that is generated. Are there any differences in the computing time in given set of elements ? W = {18, 15, 13, 12, 10, 5} and W = {15, 13, 5, 18, 10, 12} **(8 Marks)**

☑ **Soln. :** Refer to the solution to UEx.5.3.5 for W = {5, 10, 12, 13,15, 18}. Similarly, the solutions for W = {18, 15, 13, 12,10,5} and W = {15, 13, 5, 18, 10, 12} can be obtained by generation of the corresponding state space trees.

▶ **Differences in computing time**

- There will be differences in computing time for all the three given instances.

(1) W = {5, 10, 12, 13, 15, 18}: Here the numbers are arranged in increasing order. So if $\sum_{1 \leq i \leq k} w_i x_i + w_{k+1} \geq m$ then there is no need to explore a path with $x_{k+1} = 1$. So all further nodes are not explored at all.

(2) W = {18, 15, 13, 12, 10, 5}: Here the numbers are arranged in decreasing order. The state space tree for this instance explores all the nodes and has many backtracks. It cannot just simply skip the exploration of $x_{k+1} = 1$ if $\sum_{1 \leq i \leq k} w_i x_i + w_{k+1} \geq m$.

(3) W = {15, 13, 5, 18, 10, 12}: Here the numbers are arranged in random order. The state space tree for this instance also explores all the nodes and has many backtracks. It cannot just simply skip the exploration of $x_{k+1} = 1$ if $\sum_{1 \leq i \leq k} w_i x_i + w_{k+1} \geq m$.

Thus, the solutions to given instances (2) and (3) of the sum of subsets problem have the worst computing time. The instance (1) with the numbers given in increasing order has the best computing time.

Summary

- **Backtracking** is an algorithmic strategy that explores all solutions to a given problem and abandons them if they are not fulfilling the specified constraints.
- It applies to constraints satisfaction problem. There are two types of constraints: (1) implicit constraints and (2) explicit constraints.
- **Implicit constraints** give the directives of relating all candidate solutions x_i's to each other.
- **Explicit constraints** are the rules by which all candidate solutions x_i's are restricted to take on values only from a specified set in a problem instance P. They vary with the instances of the problem.
- **Solution Space** is described by all paths from the root node to a leaf node in a state space tree.
- **State space tree** is a representation of the solution space S of a problem instance P in the form of a tree. It is constructed by depth first node generation policy.
- **State space** of a problem is described by all paths from a root node to other nodes in a state space tree.
- Each node in a state space tree describes a **problem state** or a partial solution formed by making choices from the root of the tree to that node.
- **Solution states** are the problem states producing a tuple in the solution space S.
- **Answer states** are the solution states that describe the desired solution- tuple (or answer-tuple).
- **Promising node** eventually leads to the desired solution.
- **Non-promising node** eventually leads to a state that cannot produce the desired solution. Such nodes are killed by a bounding function without further exploration.
- **Live node** is a node which has been generated and all of whose children are not yet been generated.
- **E-node** is a live node whose children are currently being generated (being expanded).
- **Dead node** is a node that is either not to be expanded or for which all of its children have been generated.
- **Bounding function** kills the live nodes without exploring all their children if constraints are not satisfied.
- **Static trees** are the state space trees whose tree formulation is independent of the problem instance being solved.
- **Dynamic trees** are the state space trees whose tree formulation varies with the problem instance being solved.

- Some classic problems solvable by backtracking are as below:
  - N-Queens problem
  - Graph coloring problem
  - Sum of subset problem
  - Knapsack problem
  - Hamiltonian cycle problem
  - Puzzles like Sudoku, Rubik's cube, crosswords, maze puzzle etc.
  - Problems in gamification
  - Problems in Artificial Intelligence
  - PROLOG- a logic programming language
- **The general procedure to solve a problem by backtracking strategy:**
  (1) Identification of the data structures to solve a given problem.
  (2) Identification of the explicit constraints of a given problem.
  (3) Identification of the implicit constraints of a given problem.
  (4) Building a solution using backtracking.
- Construct a complete solution from partial solutions that satisfy the bounding function.
- Discard the non-promising partial solutions, backtrack and check for an alternative.
- Construct a state space tree by depth first node generation.
- **The 8-queens problem** asks to place 8 queens on an 8× 8 chessboard such that none of them can attack any other using the standard chess queen's moves. This implies that no two queens should be placed on the same row, column or diagonal.
- **m-coloring decision problem** checks whether all the vertices or edges or regions of G can be colored using only m colors such that no two neighboring vertices or edges or regions have the same color, where G is a given graph and m is a given positive integer.
- **m-coloring optimization problem** asks to determine the minimum number of colors m required to color all the vertices or edges or regions of G such that no two neighboring vertices or edges or regions have the same color, where G is a given graph.
- **A chromatic number** of a graph G is the smallest integer m that is required to color a graph G.
- **Sum of subsets problem** : Consider W = $(w_1, w_2, \ldots, w_n)$ is a set of n positive numbers or weights, $n > 0$. Let m is another positive number. Sum of subsets problem is to determine all subsets of W whose sums are m.

Chapter Ends...

❑❑❑

UNIT IV

CHAPTER 6

Branch and Bound

Syllabus

Branch-n-Bound: Principle, control abstraction, time analysis of control abstraction, Strategies- FIFO, LIFO and LC approaches, TSP, 0/1 Knapsack problem.

6.1 Principle and General method 6-2

UQ. Explain branch and bound approach with suitable example. What are the general characteristics of the branch and bound? SPPU - Q. 1(c), May 19, 6 Marks. 6-2

UQ. What are the general characteristics of the branch and bound approach? SPPU - Q. 1(b), May 16, 4 Marks. 6-2

6.1.1 Basic Concepts of Branch and Bound 6-2

6.1.2 Comparison between Backtracking and Branch and Bound 6-3

6.1.3 4-queens Problem 6-3

6.1.4 General Algorithm for Branch and Bound 6-5

6.2 Different search techniques in Branch and Bound 6-6

6.2.1 Least Cost Search 6-7

6.2.2 BFS or FIFO Search 6-8

6.2.3 DFS or LIFO Search 6-9

6.2.4 Bounding Function 6-10

6.3 Different strategies of Branch and Bound 6-10

6.3.1 Least Cost Branch and Bound (LCBB) 6-11

6.3.2 First In First Out Branch and Bound(FIFOBB) 6-12

6.3.2(A) Comparison between LCBB and FIFOBB 6-13

6.3.3 Last in First Out Branch and Bound (LIFOBB) 6-13

6.3.3 (A) Comparison between LCBB and LIFOBB 6-15

6.4 Traveling salesperson Problem 6-15

UQ. Explain in detail with one example Travelling Salesperson Problem using branch and bound method. SPPU - Q. 2(c), Dec. 15, 8 Marks. 6-15

6.5 0/1 Knapsack Problem 6-26

UQ. Write and explain the upper bound function for 0/1 Knapsack problem. (SPPU - Q. 7(b), Dec. 17, 8 Marks) 6-27

6.5.1 0/1 Knapsack Problem by LCBB 6-27

6.5.2 0/1 Knapsack Problem by FIFOBB 6-38

Chapter Ends 6-44

6.1 PRINCIPLE AND GENERAL METHOD

- Some optimization problems like knapsack problem, travelling salesperson problem, job sequencing problem, etc. cannot be efficiently solved by algorithmic strategies like greedy technique and dynamic programming. Such problems can be solved by the branch and bound (B&B) technique in a better way.
- In the worst case, B&B algorithms also suffer the problem of the exponential explosion, but by choosing appropriate bounds to discard the non-promising partial solutions B&B algorithms can run in reasonably lesser time.
- It gives efficient solutions to at least some large instances of difficult combinatorial, optimization problems.

UQ. Explain branch and bound approach with suitable example. What are the general characteristics of the branch and bound? **SPPU - Q. 1(c), May 19, 6 Marks**

UQ. What are the general characteristics of the branch and bound approach? **SPPU - Q. 1(b), May 16, 4 Marks**

GQ. Write a brief note on the branch and bound strategy.

- Like the backtracking technique, B&B is used to solve constraint satisfaction problems, but typically the optimization problems.
- Both backtracking and B&B are state-space algorithms as they construct the state-space trees to solve the problem.
- The basic terminologies described for backtracking (Refer to section 5.1.1) are also applicable to the B&B state space tree.
- It also incrementally constructs a solution to a given problem by evaluating one candidate solution at a time.
- Whenever a partial solution is incompetent to produce the desired optimal solution, it is rejected, and the algorithm evaluates an alternative solution.

6.1.1 Basic Concepts of Branch and Bound

Generation of a state space tree

- As in the case of backtracking, B&B generates a state space tree to efficiently search the solution space of a given problem instance.
- In B&B, all children of an E-node in a state space tree are produced before any live node gets converted into an E-node. Thus, the E-node remains an E-node until it becomes a dead node.

Evaluation of a candidate solution

- Unlike backtracking, B&B needs additional factors to evaluate a candidate solution:
  (i) **A way to assign a bound on the best values of the given criterion functions to each node in a state space tree :** It is produced by the addition of further components to the partial solution given by that node.
  (ii) **The best values of a given criterion function obtained so far :** It describes the upper bound for the maximization problem and the lower bound for the minimization problem.

A feasible solution is defined by the problem states that satisfy all the given constraints.

An **optimal solution** is a feasible solution, which produces the best value of a given objective function.

Bounding function

- It optimizes the search for a solution vector in the solution space of a given problem instance.
- It is a heuristic function that evaluates the lower and upper bounds on the possible solutions at each node.
- The bound values are used to search the partial solutions leading to an optimal solution. If a node does not produce a solution better than the best solution obtained thus far, then it is abandoned without further exploration.
- The algorithm then branches to another path to get a better solution.
- The desired solution to the problem is the value of the best solution produced so far.

The reasons to dismiss a search path at the current node

(i) The bound value of the node is lower than the upper bound in the case of the maximization problem and higher than the lower bound in the case of the minimization problem. (i.e. the bound value of the node is not better than the value of the best solution obtained until that node).

(ii) The node represents infeasible solutions, due to violation of the constraints of the problem.

(iii) The node represents a subset of a feasible solution containing a single point. In this case, if the latest solution is better than the best solution obtained so far, the best solution is modified to the value of a feasible solution at that node.

GQ. Write short notes on:
(i) Various searching techniques in branch and bound
(ii) Bounding function in branch and bound
(iii) Backtracking Vs branch and bound

GQ. Explain the term:
(a) Branch and Bound
(b) Bounding Function
(c) Various searching techniques in branch and bound
(d) Heuristic function
(e) How 0/1 Knapsack problem can be solved using branch and bound

6.1.2 Comparison between Backtracking and Branch and Bound

GQ. Differentiate branch and bound and backtracking algorithm. **(5 Marks)**

GQ. Explain the terms :
(i) Least Cost Branch and Bound
(ii) Comparison between Backtracking and Branch and Bound

GQ. Differentiate between backtracking and branch and bound method. Illustrate with an example of the 4-queens problem.

GQ. What is the difference between backtracking and branch and bound approach? Illustrate using the 8-queens problem.

GQ. What is the difference between backtracking and branch and bound approach? Illustrate using the knapsack problem.

Similarities

(1) Both of the algorithmic strategies - Backtracking and B&B are considered as the improved exhaustive search.

(2) They find all possible solutions available to the given constraint satisfaction problem.

(3) They generate a state space tree to efficiently search the solution space of a given problem instance.

(4) The bounding function is used to optimize the search of the desired solution in the solution space of a given problem instance.

(5) The bounding function kills the nodes that do not lead to the desired solution. Then the alternative paths are evaluated to get the desired solution.

(6) In the worst case, both algorithms have exponential time complexity.

GQ. Differentiate between backtracking and branch and bound method. **(5 Marks)**

Sr. No.	Backtracking	Branch and Bound
1.	It is generally used to solve non-optimization problems.	It is generally used to solve optimization problems.
2.	A state space tree is generated by using the depth first node generation policy.	A state space tree is generated by using different rules like the best-first rule, breadth first search, or depth first search.
3.	All backtracking algorithms search a state space tree by applying DFS (depth first search) only.	Based on different search techniques B&B algorithms have different variations as : • **LCBB** (Least Cost Branch and Bound): It uses the least cost search. • **FIFOBB** (First In First Out Branch and Bound) : It uses BFS (breadth first search) • **LIFOBB** (Last In First Out Branch and Bound) : It uses DFS (depth first search).
4.	The bounding function is typically a feasibility function that does not make use of the value of the best solution seen so far.	The bounding function is a heuristic function that computes the bound values at each node and compares them with the value of the best solution seen so far.

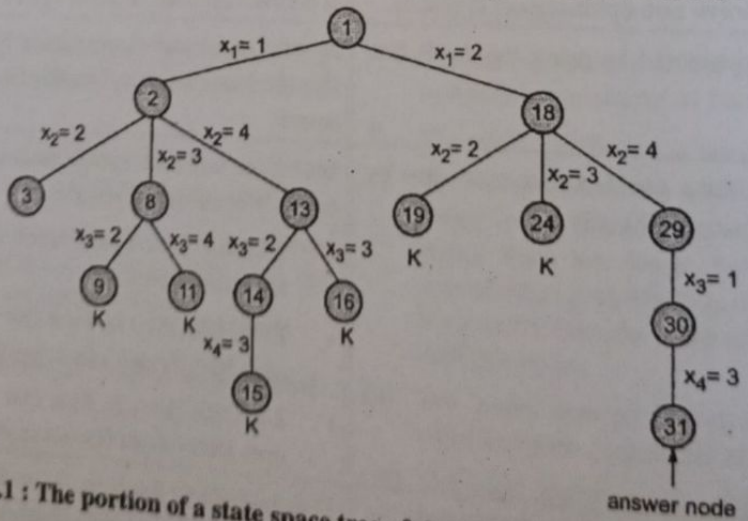
Sr. No.	Backtracking	Branch and Bound
5.	If a node does not satisfy the implicit constraints and does not lead to the desired solution, then that node is killed without pursuing that path further. The algorithm backtracks and evaluates an alternative path.	If the node's bound value is not better than the value of the best solution obtained until that node or if it is violating the constraints, then that node is killed without pursuing that path further. The algorithm then jumps to another branch to evaluate an alternative solution.
6.	It searches the state space tree until it gets a solution.	It searches the state space tree until it gets an optimal solution.
7.	Some typical problems solved by backtracking are the N-queens problem, graph colouring problem, Hamiltonian cycle problem, etc.	Some typical problems solved by B&B are the Job sequencing problem, 0/1 knapsack problem, Travelling salesperson problem, etc.

6.1.3 4-queens Problem

GQ. Illustrate with an example of the 4-queens problem. **(10 Marks)**

Example of the 4-queens problem

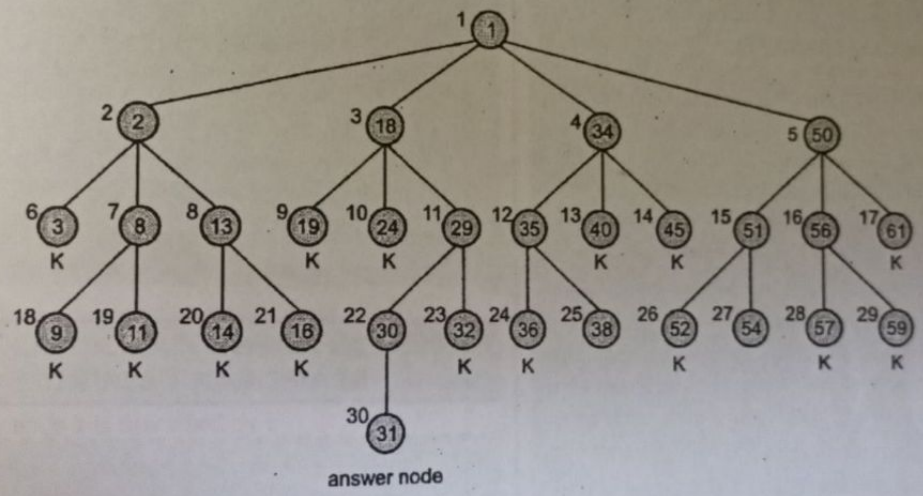
- In backtracking a state space tree is constructed by using depth first node generation policy.
- The state space tree of the 4-queens problem by backtracking is depicted in Fig. 5.2.3. Node numbers in Fig. 5.2.3 are given as per the sequence of their generation. The edges are labelled by the values of a candidate solution x_i, $1 \le i \le n$. It represents the column number on row i on which the i^{th} queen can be placed.
- The portion of a tree in Fig. 5.2.3 is shown in Fig. 6.1.1. Here, the node numbers are corresponding to the node numbers in Fig. 5.2.3. The nodes marked with 'K' indicate that the nodes are killed by a bounding function.
- In backtracking, node1 is generated when no queen is placed on a chessboard. It then becomes an E-node and generates node2 by placing the first queen on the 1^{st} row and the 1^{st} column.
- At the moment, node2 is generated it becomes the next E-node and generates node3 by placing the 2^{nd} queen on the 2^{nd} row and 2^{nd} column. This places two queens on the same diagonal, so the bounding function kills node3 and the algorithm backtracks to generate node8 by placing the 2^{nd} queen on the 2^{nd} row and 3^{rd} column.
- The backtracking algorithm thus continues until the desired solution is obtained. One of the answer states is node 31.



(1F11)**Fig. 6.1.1 : The portion of a state space tree of the 4-queens problem by backtracking**

- The FIFOBB generates a state space tree using BFS where the E-node remains the E-node until it becomes a dead node.

- The FIFOBB maintains the list of live nodes as a queue. It thus follows the First-In-First-Out policy to evaluate all the nodes by applying a bounding function.
- The portion of a state space tree of the 4-queens problem by FIFOBB is shown in Fig. 6.1.2.
- In this state space tree, the numbers inside the circles represent the node numbers corresponding to Fig. 5.2.3, and the numbers outside the nodes are given as per the sequence of their generation by FIFOBB. The nodes marked with 'K' indicate that the nodes are killed by a bounding function.
- In FIFOBB, node1 is generated when no queen is placed on a chessboard. It then becomes an E-node and generates nodes 2, 18, 34, and 50. These nodes are added to the FIFO list of live nodes. Node1 becomes a dead node after generating all of its children.
- As per the FIFO list of live nodes, node2 becomes the next E-node and generates nodes 3, 8 13 which are then added to the FIFO list of live nodes.
- The bounding function is applied to kill the non-promising nodes.
- Each time the next E-node is selected from the FIFO list of live nodes.
- The FIFOBB algorithm thus continues until the desired solution is obtained. One of the answer states is node31. When it is reached, the only live nodes present in the FIFO list are nodes 38 and 54.
- By comparing the portions of state space trees shown in Fig. 6.1.1 and 6.1.2, we can conclude that backtracking is a better search method than FIFOBB to solve the 4-queens problem.



(1F12)**Fig. 6.1.2 : The portion of a state space tree of the 4-queens problem by FIFOBB**

6.1.4 General Algorithm for Branch and Bound

GQ. Write just steps for Backtracking and Branch-and-Bound algorithms. **(6 Marks)**

GQ. Describe in brief general strategy used in branch and bound method. Write a general algorithm for branch and bound method.

The general procedure

(1) Construct a state space tree such that the E-node remains an E-node until it becomes a dead node.

(2) Maintain the list of live nodes using a suitable data structure according to the search technique incorporated in the B&B algorithm. (E.g., for LC search: a min-heap data structure, for FIFO search: a queue data structure, for LIFO search : a stack data structure).

(3) Select the next E-node from the list of live nodes by following one of the search techniques (E.g., LC search, FIFO search, LIFO search).

(4) At each node, compute the values of lower and upper bounds on the possible solutions from that node.

(5) Apply the bounding functions to check whether a node produces a better solution than the best solution obtained thus far. Otherwise, that node is abandoned without its further exploration.

(6) Enumerate another branch to get a better solution.

(7) Repeat steps (1) to (6) until an optimal solution to the problem is obtained.

☞ Control abstraction for B&B method

```
typedef struct                  /*Defines a structure object to
                                represent a node in a T.*/
{
   listNode_T * next;
   listNode_T * parent;         /*It is useful to trace the path
                                from an answer state to the
                                root.*/
   float cost;
} listNode_T;
listNode_T  Node;
Algorithm B&B(T)
/* The algorithm follows B&B strategy to get an answer node
in a state space tree.
Input : T is the root node of a state space tree. The function
Explore(E) generates all children of a current E-node E. The
function Evaluate_Bounds(k) computes the bound values at
each node k and kills the incompetent node by applying a
bounding function. The function Insert(k) inserts node k into
the list of live nodes. The function Next_E() identifies a next
E-node from the list of live nodes, then removes it from that
list and returns it.
Output : A path from an answer node to the root T in a state
space tree.*/
{
  if ( *T is an answer state )
  {
     write(*T);
     return;
  }
  E :=T;                                  // E-node
  Initialize the list of live nodes to be empty;
  repeat
  {
     Explore(E);
          /* Generate all children of a current E-node E.*/
     for(each child k of E)
     {
        Evaluate_Bounds(k);
                                /*Compute the bound values at each
                                node k and kill the incompetent node
                                by applying a bounding function.*/
        if (k is a minimum-cost answer state)
        {
           write the path from k to T;
           Update the upper bound according to a
           minimum cost answer state;
           return;
        }
        Insert(k);   /*Insert k into the list of live nodes.*/
        (k→parent):= E;
     }
     if(there are no more live nodes)
     {
        write("No answer state." );
        return;
     }
     E :=Next_E();          /*Identify a next E-node from the
                            list of live nodes, then remove it
                            from that list and return it.*/
  }until(FALSE);
}
```

⏭ 6.2 DIFFERENT SEARCH TECHNIQUES IN BRANCH AND BOUND

GQ. Explain for branch and bound:
(i) LIFO search (ii) FIFO search
(iii) LC search

GQ. Explain following terms:
(i) Branch and Bound
(ii) LC Search
(iii) Bounding function

The B&B algorithms incorporate different search techniques to traverse a state space tree. Different search techniques used in B&B are listed below:

(1) LC search (Least Cost Search)

- It uses a heuristic cost function to compute the bound values at each node.
- Nodes are added to the list of live nodes as soon as they get generated.

- The node with the least value of a cost function is selected as a next E-node.

(2) **BFS(Breadth First Search)**

- It is also known as a FIFO search.
- It maintains the list of live nodes in first-in-first-out order i.e. in a queue.
- The live nodes are searched in the FIFO order to make them next E-nodes.

(3) **DFS (Depth First Search)**

- It is also known as a LIFO search.
- It maintains the list of live nodes in last-in-first-out order i.e. in a stack.
- The live nodes are searched in the LIFO order to make them next E-nodes.

6.2.1 Least Cost Search

GQ. What is LC Search? How does it help in finding a solution for the branch and bound algorithm?

- Both DFS and BFS are rigid searching techniques. Without applying any heuristics, they blindly select a next E-node as per DFS or BFS policy respectively.
- However, Least Cost (LC) search uses a heuristic cost function $\hat{c}(\cdot)$ to assign the ranks to the live nodes.
- It estimates the extra computational work (cost) to reach an answer state from the current live node and accordingly assigns the rank to that live node.
- The cost at any node k is described by :
  - (i) The minimum number of nodes required to be generated in a subtree of k to reach an answer state.
  - (ii) The number of levels the nearest answer state is distant from k.
- Assume $\hat{g}(k)$ is the estimation of the added cost required to reach an answer state from k and h(k) is the cost of reaching k from the root. Then the rank of k is given by :

$\hat{c}(k) = f(h(k)) + \hat{g}(k)$; where $f(\cdot)$ is an increasing function.

- The live node with a minimum $\hat{c}(\cdot)$ value is always selected as a next E-node, hence this searching method is named as **"Least Cost(LC) search".**
- BFS and DFS can be described as special cases of LC search :
  - (i) **BFS :** When we consider $\hat{g}(k) \equiv 0$ and f(h(k)) as the level of node k then the node with a minimum level will generate the least $\hat{c}(\cdot)$ value and hence it becomes a next E-node. Thus it generates nodes by level resulting in BFS.
  - (ii) **DFS :** When we consider $(h(k)) \equiv 0$ and if k has a child j, resulting $\hat{g}(k) \geq \hat{g}(j)$, then the node j (a child of node k) will have the least $\hat{c}(\cdot)$ value and hence it becomes a next E-node. Thus it generates nodes by depth resulting in DFS.
- The cost function $c(\cdot)$ in the LC search is described as below :
  - (i) If k is an answer state, then c(k) is the cost determined by a path from the root to the node k in a state space tree.
  - (ii) If k is not an answer state so that the subtree of k also does not contain any answer state, then $c(k) = \infty$.
  - (iii) If k is not an answer state, but a subtree of k contains an answer state, then c(k) is the cost of the least cost answer state in subtree k.
  - (iv) $\hat{c}(k)$ with $f(h(k)) = h(k)$ gives an approximation to $c(\cdot)$.

☞ Control abstraction for Least Cost Search

GQ. What is LC Search? Explain in detail Control abstraction for LC Search.

- Consider S is a state space tree and $c(\cdot)$ is a cost function in LC search. Let k be a node in S, then c(k) gives the least cost of any answer state in the subtree rooted at node k. So, c(S) can be considered as a cost of the least-cost answer state in S.
- As the computation of $c(\cdot)$ is complex, we can replace it by a heuristic function as $\hat{c}(\cdot)$ to estimate $c(\cdot)$ $\hat{c}(\cdot)$ should be easily computable and if k is an answer state or a leaf node then $c(\cdot) = \hat{c}(k)$.
- The algorithm LCS(T) finds an answer state in a state space tree with the root node T by using $\hat{c}(\cdot)$.

```
typedefstruct                /*Defines a structure object to
                              represent a node in T.*/
{
listNode_T * next;
listNode_T * parent; /*It is useful to trace the path
                              from an answer state to the
root.*/
float cost;
} listNode_T;
```

```
listNode_TNode;

Algorithm LCS(T)
/* The algorithm follows LC search to get an answer node in a
state space tree.
Input : T is the root node of a state space tree. The function
Insert_MinHeap(k) inserts node k into the list of live nodes
which is maintained as a min-heap. The function Find_Least()
identifies a live node with the least ĉ(·) value (i.e. the root
node of a min-heap of live nodes), removes it from a min-heap
of live nodes and returns it.
Output : Path from an answer node to the root T in a state
space tree. */
{
  if ( *T is an answer state )
  {
     write(*T);
     return;
  }
  E :=T;    // E-node
  Initializea min-heap of live nodes to be empty;
  repeat
  {
     for(each child k of E)
     {
        if (k is an answer state)
        {
           write the path from k to T;
           return;
        }
        Insert_MinHeap(k);  /*Insert k into a min-
                                    heap of live nodes.*/
        (k→parent):= E;
     }
     if(there are no more live nodes)
     {
        write("No answer state.");
        return;
     }
     E :=Find_Least();
                  /*Identify a live node with the least
                  ĉ(·) value, remove it from a min-
                        heap of live nodes and return it.*/
  }until(FALSE);
}
```

Correctness of an algorithm LCS()

- Initially, the root node T is the first E-node. A variable E always points to the current E-node. So, we assign E := T.
- The algorithm LCS() always maintains the list of live nodes as a min-heap.
- When a current E-node E is completely explored, E it becomes a dead node. It is possible only when none of the children of E is an answer state.
- If one of the children of E is an answer state, then by writing the path from that answer state to the root the algorithm terminates.
- If a child of E is not an answer state, then it becomes a live node and its parent is set to E.
- When there are no live nodes left, the algorithm terminates; otherwise, Find_Least() identifies the least cost node that becomes a next E-node and the algorithm continues.
- The algorithm LCS() ends only when either an answer state is found or the entire state space search tree has been searched.
- The algorithm LCS() guarantees its termination only for finite state space trees.

6.2.2 BFS or FIFO Search

- In FIFO search, there is no need for any heuristic function to select the next E-node from the list of live nodes. It follows breadth first search to traverse a state space tree.
- It implements a list of live nodes as a queue and always picks out the node at the front end of that queue as a next E-node.
- It simply selects the live nodes in the first-in-first-out (FIFO) order.

Control Abstraction for FIFO Search

GQ. What is FIFO Search? Explain in detail Control abstraction for FIFO Search. **(8 Marks)**

```
typedefstruct                /*Defines a structure object to
                               represent a node in a T.*/
{
  listNode_T * next;
  listNode_T * parent;       /*It is useful to trace the path
                               from an answer state to the root.*/
  float cost;
} listNode_T;
listNode_TNode;

Algorithm BFS(T)
/* The algorithm follows BFS to get an answer node in a state
space tree.
Input : T is the root node of a state space tree. The list of live
nodes is maintained as a queue. The function Append_Q(k)
appends(inserts) a node k to the rear end of a queue of live
nodes. The function Delete_Q() deletesa live node at the front
end of a queue of live nodes and returns it.
Output : Path from an answer node to the root T in a state
space tree.*/

if ( *T is an answer state )
{
write(*T);
  return;
}
E :=T;          // E-node
Initialize a queue of live nodes to be empty;
repeat
{
  for(each child k of E)
  {
    if (k is an answer state)
    {
      write the path from k to T;
      return;
    }
    Append_Q(k);     /*Insert k to the rear end of a
                       queue of live nodes.*/
    (k→parent):= E;
  }
  if(there are no more live nodes)
  {
    write("No answer state." );
    return;
  }
  E :=Delete_Q();    /*Delete a live node at the front
                       end of a queue of live nodes and
                       return it */
}until(FALSE);
}
```

6.2.3 DFS or LIFO Search

- In LIFO search there is no need for any heuristic function to select the next E-node from the list of live nodes. It follows depth first search to traverse a state space tree.
- It implements a list of live nodes as a stack and always picks out the node at the top of the stack as a next E-node.
- It simply selects the live nodes in the last-in-first-out (LIFO) order.

Control abstraction for LIFO search

GQ. What is LIFO Search? Explain in detail Control abstraction for LIFO Search. **(8 Marks)**

```
typedefstruct          /*Defines a structure object to represent
                         a node in a state space tree.*/
{
  listNode_T * next;
  listNode_T * parent;  /*It is useful to trace the path
                          from an answer state to the root.*/
  float cost;
} listNode_T;
listNode_TNode;
Algorithm DFS(T)
/* The algorithm follows DFS to get an answer node in a state
space tree.
Input : T is the root node of a state space tree. The list of live
nodes is maintained as a stack. The function Push_Stack(k)
inserts (pushes) a node k to the top of a stack of live nodes.
```

```
The function Pop_Stack() removes(pops) a live node at the top
of a stack of live nodes and returns it.
Output : Path from an answer node to the root T in a state
space tree.*/
{
  if ( *T is an answer state )
  {
    write(*T);
    return;
  }
  E :=T;              // E-node
  Initialize a stack of live nodes to be empty;
  repeat
  {
    for(each child k of E)
    {
      if (k is an answer state)
      {
        write the path from k to T;
        return;
      }
      Push_Stack(k);
            /*Push k to the top of a stack of live nodes.*/
      (k→parent):= E;
    }
    if(there are no more live nodes)
    {
      write("No answer state." );
      return;
    }
    E :=Pop_Stack();
                  /*Pop a live node at the top of
                  a stack of live nodes and return it */
  }until(FALSE);
}
```

6.2.4 Bounding Function

- The usage of the bounding function prunes the subtrees in a state space tree that do not have an answer state.
- Each answer state k has an associated cost c(k) and the least-cost answer state is defined as an optimal solution.
- At each node k in a state space tree, lower and upper bounds on solutions feasible from k are computed.
- The estimation of the added cost reaches an answer state from a node k is described by $\hat{c}(k)$, so that, $\hat{c}(k) \leq c(k)$. It defines lower bound on solutions feasible from node k.
- Consider *upper* gives an upper bound on the cost of a least-cost solution. Then all live nodes k with
- $\hat{c}(k) > upper$ can be killed without further exploration since all answer states reachable from a node k have cost $c(k) \geq \hat{c}(k) > upper$.
- Initially $upper = \infty$ and whenever a new answer state is obtained, the value of *upper* is updated accordingly.
- Since a minimum-cost answer state is selected as an optimal solution, B&B algorithms are directly applicable to solve the minimization problems.
- However, to solve the maximization problems by B&B algorithms the signs of objective functions are changed to describe the maximization problems as the minimization problems.

6.3 DIFFERENT STRATEGIES OF BRANCH AND BOUND

GQ. Explain the branch & bound technique and different strategies used in it like LCBB, FIFOBB, compare LCBB, FIFOBB

GQ. Explain the term:

(i) Branch and Bound
(ii) LC search
(iii) FIFO branch & bound
(iv) Bounding Function
(v) Difference in LIFOBB and LCBB

- The B&B algorithms traverse a state space tree by following any of the search techniques like LC search, DFS or BFS in which a current E-node is completely explored by generating all its children before making any other node as an E-node.
- Based on different search techniques the B&B algorithms have different variations as:
  (1) **LCBB (Least Cost Branch and Bound)**
  (2) **FIFOBB (First In First Out Branch and Bound)**
  (3) **LIFOBB (Last In First Out Branch and Bound)**

6.3.1 Least Cost Branch and Bound (LCBB)

- This B&B algorithm uses LC search for the selection of next E-node from the list of live nodes.
- LCBB uses a heuristic cost function $\hat{c}(\cdot)$ to assign the ranks to the live nodes.
- It estimates the extra computational work (cost) needed to reach an answer state from the current live node and accordingly assigns the rank to that live node.
- A live node with the least $\hat{c}(\cdot)$ value is selected as a next E-node from the list of live nodes.
- A minimum-cost answer state gives an optimal solution.
- Initially, *upper* = ∞ and whenever a new answer state is obtained, the value of the *upper* is updated accordingly.
- To solve the maximization problems by the LCBB algorithms the signs of the objective functions are changed to describe the maximization problems as the minimization problems.

Control abstraction for LCBB

GQ. Write an algorithm for Least Cost (LC) branch and bound. **(8 Marks)**

```
typedefstruct          /*Defines a structure object to represent
                         a node in a state space tree.*/
{
  listNode_T * next;
  listNode_T * parent;  /*It is useful to trace the path
                          from an answer state to the root.*/
  float cost;
} listNode_T;
listNode_TNode;
```

Algorithm LCBB(T)

/* It describes the least cost search based B&B strategy to get an answer node in a state space tree.

Input : T is the root node of a state space tree. *upper* is the upper bound. The function Explore(E) generates all children of a current E-node E. The function Evaluate_Bounds(k) computes the bound values at each node k and kills the incompetent node if $\hat{c}(k) \geq upper$. The function Insert_MinHeap(k) inserts node k into the list of live nodes which is maintained as a min-heap.The function Find_Least() identifies a live node with least $\hat{c}(\cdot)$ value(i.e. the root node of a min-heap of live nodes), removes it from a min-heap of live nodes and returns it.

Output : Path from an answer node to the root T in a state space tree.*/

```
{
  upper:=∞;
  if ( *T is an answer state )
  {
    write(*T);
    return;
  }
  E :=T; // E-node
  Initialize a min-heap of live nodes to be empty;
  repeat
  {
    Explore(E);          /* Generate all children of a
                            current E-node E.*/
    for(each child k of E)
    {
    Evaluate_Bounds(k);
                /*Compute the bound values at each
                  node k and kill the incompetent node
                  by applying a bounding function.*/
      if (k is a minimum-cost answer state)
      {
        write the path from k to T;
        Update upper according to a minimum cost
        answer state;
        return;
      }
      Insert_MinHeap(k);
            /*Insert k into a min-heap of live nodes. */
      (k→parent):= E;
    }
    if(there are no more live nodes)
    {
      write("No answer state.");
      return;
    }
    E :=Find_Least();    /*Identify a live node with least
                           ĉ(·) value, remove it from a
                           min-heapof live nodes and
                           return it.*/
  }until(FALSE);
}
```

6.3.2 First In First Out Branch and Bound(FIFOBB)

- This B&B algorithm uses BFS for the selection of a next E-node from the list of live nodes.
- It implements the list of live nodes as a queue and always selects the node at the front end of that queue as a next E-node.
- A minimum-cost answer state gives an optimal solution.
- Initially, *upper* = ∞ and whenever a new answer state is obtained, the value of the *upper* is updated accordingly.
- To solve the maximization problems by the FIFOBB algorithms the signs of the objective functions are changed to describe the maximization problems as the minimization problems.

Control abstraction for FIFOBB

GQ. Write an algorithm for FIFO branch and bound. **(8 Marks)**

```
typedefstruct      /*Defines a structure object to represent a
                   node in a state space tree.*/
{
  listNode_T * next;
  listNode_T * parent;   /*It is useful to trace the
                         path from an answer state to
                         the root.*/
  float cost;
}listNode_T;
listNode_TNode;
Algorithm FIFOBB(T)
/* It describes the BFS based B&B strategy to get an answer
node in a state space tree.
Input : T is the root node of a state space tree. upper is the
upper bound. The function Explore(E) generates all children
of a current E-node E. The function Evaluate_Bounds(k)
computes the bound values at each node k and kills the
incompetent node if ĉ(k)>upper. The function Append_Q(k)
inserts a node k to the rear end of a queueof live nodes.The
function Delete_Q() removes(deletes) a live node at the front
end of a queue of live nodes and returns it.
Output : Path from an answer node to the rootT in a state space
tree.*/
{
  upper:= ∞;
  if ( *T is an answer state )
  {
    write(*T);
    return;
  }
  E :=T;                                   // E-node
  Initialize a queue of live nodes to be empty;
    repeat
    {
      Explore(E);
        /* Generate all children of a current E-node E.*/
      for(each child k of E)
      {
        Evaluate_Bounds(k);
                /*Compute the bound values
                at each node k and kill the
                incompetent node by
                applying a bounding function.*/
        if (k is a minimum-cost answer state) then
        {
          write the path from k to the root in T;
          Update upper according to a minimum cost
          answer state;
          return;
        }
        Append_Q(k);    /*Insert k to therear end
                        of a queue of live nodes.*/
        (k→parent):= E;
      }
      if(there are no more live nodes)
      {
        write("No answer state.");
        return;
      }
      E :=Delete_Q();
                /*Delete a live node at the front end of a
                        queue of live nodesand return it*/
    }until(FALSE);
}
```

6.3.2(A) Comparison between LCBB and FIFOBB

GQ. Compare LCBB and FIFOBB. (4 Marks)

Sr. No.	LCBB	FIFOBB
1.	It is a Least Cost Branch and Bound technique.	It is a First In First Out Branch and Bound technique.
2.	This B&B algorithm follows LC search to traverse a state space tree.	This B&B algorithm follows breadth first search (BFS) to traverse a state space tree.
3.	LCBB uses a heuristic cost function $\hat{c}(\cdot)$ to assign the ranks to the live nodes. It selects the next E-node based on the rank of a live node.	FIFOBB does not use any heuristic ranking function to select the next E-node from the list of live nodes.
4.	A live node with the least $\hat{c}(\cdot)$ value is selected a next E-node from the list of live nodes.	It simply selects the live nodes in the first-in-first-out (FIFO) order.
5.	A data structure used to store the list of live nodes is a min-heap.	A data structure used to store the list of live nodes is a queue.
6.	It always selects the root node of a min-heap of live nodes a next E-node.	It always selects the node at the front end of a queue of live nodes as a next E-node.
7.	The time complexity of insertion and deletion of a live node in a min-heap of m live nodes is O(log m).	The time complexity of insertion and deletion of a live node in a queue of m live nodes is $\Theta(1)$.
8.	Though LCBB examines fewer nodes than FIFOBB, the time needed to find each E-node is more.	Though FIFOBB examines more nodes than LCBB, the time needed to find each E-node is lesser.
9.	Considering the real computational time, LCBB will be superior to FIFOBB only when it has far fewer E-nodes than FIFOBB.	Considering the real computational time, FIFOBB will be superior to LCBB unless LCBB has far fewer E-nodes than FIFOBB.

6.3.3 Last in First Out Branch and Bound (LIFOBB)

- This B&B algorithm uses DFS for the selection of next E-node from the list of live nodes.
- It implements the list of live nodes as a stack and always selects the node at the top of that stack as a next E-node.
- A minimum-cost answer state gives an optimal solution.
- Initially, *upper* = ∞ and whenever a new answer state is obtained, the value of the *upper* is updated accordingly.
- To solve the maximization problems by the LIFOBB algorithms the signs of the objective functions are changed to describe the maximization problems as the minimization problems.

Control abstraction for LIFOBB

GQ. Write an algorithm for LIFO branch and bound.

```
typedefstruct        /*Defines a structure object to
                     represent a node in a state space tree.*/
{
listNode_T * next;
listNode_T * parent;  /*It is useful to trace the path
                      from an answer state to
                      the root.*/
```

```
    float cost;
}listNode_T;
listNode_TNode;
```

Algorithm LIFOBB(T)

/* It describes the DFS based B&B strategy to get an answer node in a state space tree.

Input: T is the root node of a state space tree. *upper* is the upper bound. The function Explore(E) generates all children of a current E-node E. The function Evaluate_Bounds(k) computes the bound values at each node k and kills the incompetent node if ĉ(k)>*upper*. The function Push_Stack(k) inserts (pushes) a node k to the top of a stack of live nodes. The functionPop_Stack()removes (pops) a live node at the top of a stack of live nodes and returns it.

Output: Path from an answer node to the root T in a state space tree.*/

```
{
  upper:= ∞;
  if ( *T is an answer state )
  {
    write(*T);
    return;
  }
    E :=T;     // E-node
    Initialize a stack of live nodes to be empty;
    repeat
    {
      Explore(E);
                                                    /* Generate all children of a current E-node E.*
      for(each child k of E)
      {
        Evaluate_Bounds(k);
                /*Compute the bound values at each node k and kill the incompetent node by applying bounding
                function.*/
        if (k is a minimum-cost answer state)
        {
          write the path from k to T;
          Update upper according to a minimum cost answer state;
          return;
        }
        Push_Stack(k);

        (k→parent):= E;                                  /*Push k to thetop of a stack of live nodes.*/
    }
    if(there are no more live nodes)
    {
```

```
        write("No answer state.");
        return;
    }
    E := Pop_Stack();
                                    /*Pop a live node at the top of a stack of live nodes and return it. */
}until(FALSE);
```

6.3.3 (A) Comparison between LCBB and LIFOBB

GQ. Compare LCBB and FIFOBB. (4 Marks)

Sr. No.	LCBB	LIFOBB
1.	It is the Least Cost Branch and Bound technique.	It is the Last in First Out Branch and Bound technique.
2.	This B&B algorithm follows LC search to traverse a state space tree.	This B&B algorithm follows depth first search (DFS) to traverse a state space tree.
3.	LCBB uses a heuristic cost function $\hat{c}(\cdot)$to assign the ranks to the live nodes. It selects the next E-node based on the rank of a live node.	LIFOBB does not use any heuristic ranking function to select the next E-node from the list of live nodes.
4.	A live node with the least $\hat{c}(\cdot)$ value is selected a next E-node from the list of live nodes.	It simply selects the live nodes in the last-in-first-out (LIFO) order.
5.	A data structure used to store the list of live nodes is a min-heap.	A data structure used to store the list of live nodes is a stack.
6.	It always selects the root node of a min-heap of live nodes a next E-node.	It always selects the node at the top of a stack of live nodes as a next E-node.
7.	The time complexity of insertion and deletion of a live node in a min-heap of m live nodes is O(log m).	The time complexity of push and pop operation of a live node on a stack of m live nodes is $\Theta(1)$.
8.	Though LCBB examines fewer nodes than LIFOBB, the time needed to find each E-node is more.	Though LIFOBB examines more nodes than LCBB, the time needed to find each E-node is lesser.
9.	Considering the real computational time, LCBB will be superior to LIFOBB only when it has far fewer E-nodes than LIFOBB.	Considering the real computational time, LIFOBB will be superior to LCBB unless LCBB has far fewer E-nodes than LIFOBB.

6.4 TRAVELING SALESPERSON PROBLEM

UQ. Explain in detail with one example Travelling Salesperson Problem using branch and bound method. SPPU - Q. 2(c), Dec. 15. 8 Marks

The traveling salesperson problem asks to minimize the cost of a tour that visits all cities only once and ends at the starting city. Since it is a minimization problem, it is directly solved by the B&B algorithm without changing the sign of an objective function.

Problem description

GQ. What is a traveling sales person problem ? (2 Marks)

- Consider G = (V, E) be a weighted graph where V is a set of vertices and E is a set of edges. |V| = n. Assume C_{ij} be the weight (cost) of an edge <i, j>, $C_{ij} = \infty$ if <i, j>$\notin$E, C_{ij}= w if <i, j>$\in$E, w is any positive real number.
- Traveling salesperson problem is to find the shortest tour(i.e. a tour with the least cost) of G that begins at any node i$\in$V and ends at i by visiting all remaining nodes in (V-{i}) only once.

- Let us assume that every tour begins and ends at node 1. Thus the solution space $S = \{1, \pi, 1 \mid \pi$ is an ordering of $(2, 3, \ldots, n)\}$. $|S| = (n-1)!$
- The size of a solution space S can be reduced by constraining S so that $(1, i_1, i_2, \ldots, i_{n-1}, 1) \in S$ iff an edge $\langle i_k, i_{k+1} \rangle \in E$, $0 \le k \le n-1$, $i_0 = i_n = 1$. S can be represented by a state space tree. Each leaf node in it represents the tour described by the path from the root to that leaf node.

☞ The general procedure of TSP by LCBB

GQ. Explain the branch & bound algorithmic strategy for solving the problem; take an example of traveling salesperson problem using branch and bound. **(10 Marks)**

GQ. Explain the dynamic reduction with all steps with respect to traveling salesperson problem. **(6 Marks)**

(1) Represent a given graph G = (V, E) as a cost adjacency matrix.

(2) Construct a state space tree to solve a TSP instance by the B&B method.

(3) Consider *upper* as an upper bound on the cost of the least-cost solution. Initially, set *upper* = ∞.

(4) At each live node k, compute the lower and upper bounds on solutions feasible from k.

(i) A heuristic function ĉ(k) is the estimation of the cost c(k) and it defines the lower bound.

(ii) A function u(k)gives a simple upper bound on the cost of the least cost answer state in the subtree of k. $\hat{c}(k) \le c(k) \le u(k)$.

(5) Use the dynamic reduction of a cost matrix corresponding to the graph G to compute ĉ(k) at each node k in the state space tree.

(6) **Computation of ĉ(k) using the dynamic reduction of a cost matrix :**

- Compute a reduced cost matrix for each node k in a state space tree of TSP as follows :

(i) Row reduction: Each row in a cost matrix must have at least one zero and all other entries are positive; if not so, subtract the minimum valued entry in a row from all entries in that row.

(ii) Column reduction: Each column in a cost matrix must have at least one zero and all other entries are positive; if not so, subtract the minimum valued entry in a column from all entries in that column.

(iii) The total reduced cost r is the total amount of subtracted cost during row and column reduction.

- Let A be the reduced cost matrix for node k. Consider node k has a child j such that branch(k, j) in a state space tree of TSP corresponds to the inclusion of an edge $\langle p, q \rangle \in E$ in the tour.
- If child j is not a leaf node, then the reduced cost matrix for j can be determined as follows:

(i) Mark all entries in row p and column q of A to ∞. It avoids the use of any more paths with source p or destination q(except for an edge <p, q>) in any tour of the subtree of a child j.

(ii) Mark A(q, 1) = ∞. It avoids the use of an edge <q, 1> in any tour of the subtree of a child j.

(iii) Apply row and column reduction to the resultant matrix except for rows and columns with all ∞.

- Let B be the resultant matrix. If r is the total reduced cost in step (iii) then $\hat{c}(j) = \hat{c}(k) + A[p, q] + r$.
- If child j is a leaf node, then $\hat{c}(j) = c(j)$ is easily computed as each leaf defines a unique tour.

(7) For the upper bound function u(k) at each node k, use u(k) = ∞ for all nodes.

(8) Kill the node k if $\hat{c}(k) > upper$.

(9) Select the next E-node by applying any of the suitable search techniques(E.g. LC search for LCBB, DFS for FIFOBB and BFS for LIFOBB)

(10) Whenever a new answer state is computed, the value of the *upper* is updated accordingly.

(11) Repeat the steps (4) to (10) till all live nodes become dead. The least-cost answer state gives an optimal solution (cost of the shortest tour) to the problem.

☞ Examples

➠ **Note :** For TSP by DP, mark all diagonal entries = 0 and for TSP by B&B, mark all diagonal entries = ∞ in a cost adjacency matrix of a given graph.

Ex. 6.4.1 : What is traveling salesperson problem? Find the solution to the following traveling salesperson problem using branch and bound method. cost matrix

$$= \begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

(18 Marks)

Soln. :

Traveling salesperson problem : Refer to problem description described above.

TSP solution by branch and bound :

- We solve a given instance of TSP by LCBB. For the same, we use the dynamic reduction of a cost matrix. We maintain the lists of live nodes (L), E-nodes (E) and dead nodes (D).
- We construct a state space tree in which each node has a reduced cost matrix assigned to it. The portion of the state space tree for the same is shown in Fig. Ex. 6.4.1.
- Initially, *upper* = ∞. A node k is killed if $\hat{c}(k) >$ upper.
- To compute the lower bound $\hat{c}(\cdot)$ we use a formula $\hat{c}(j) = \hat{c}(k) + A[k, j] + r$ where j is a child of k obtained by including an edge < p, q > of a graph in the tour, A is a reduced matrix associated with parent k and r is a reduced cost of matrix associated with j.
- The computations are as follows :

(1) We first determine a reduced matrix A_1 for node 1 from a given cost matrix. Initially, *upper* = ∞.

L	1
E	–
D	–

r

$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix} \begin{matrix} 10 \\ 2 \\ 2 \\ 3 \\ 4 \end{matrix}$$

21

Row reduction

$$\Downarrow$$

$$\begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 2 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix}$$

r = 1 3 = 4

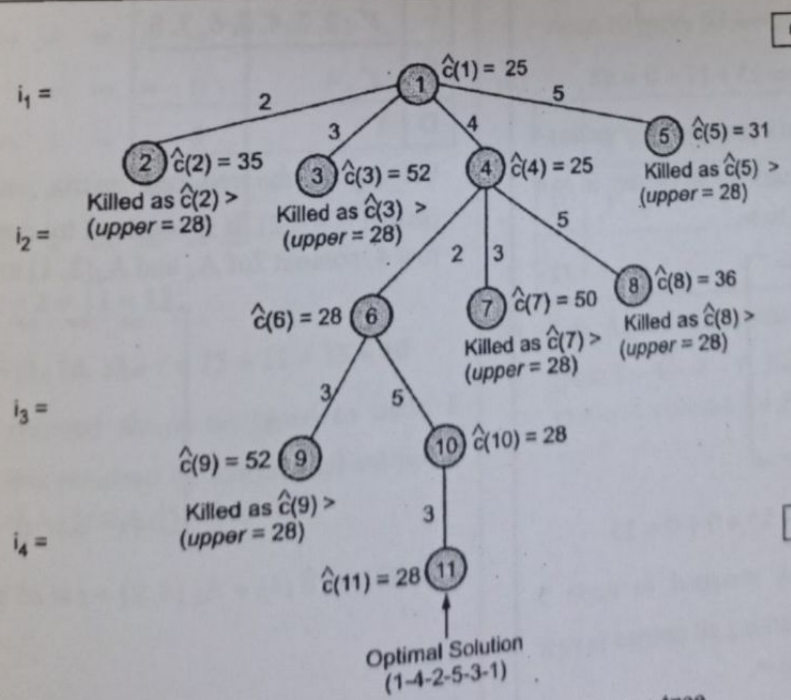
Column reduction

$$\Downarrow$$

$$A_1 = \begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 14 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

$\therefore r = 21 + 4 = 25$

$\therefore \hat{c}(1) = r = 25$



(1F7) **Fig. Ex. 6.4.1 : The state space tree**

- Since node 1 is the only live node it is explored by generating node 2 (path 1 – 2), node 3 (path 1 – 3), node 4 (path 1 – 4) and node 5 (path 1 – 5).

(2)

L	1, 2, 3, 4, 5
E	1
D	

(i) Let A_2 be the reduced matrix assigned to node 2 (path 1 – 2). It is obtained by marking all entries in row 1, column 2 of A_1 and $A_1[2, 1]$ to ∞.

$$A_2 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 14 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

$$\hat{c}(2) = \hat{c}(1) + A_1[1, 2] + r = 25 + 10 + 0 = 35$$

(ii) Let A_3 be the reduced matrix assigned to node 3 (path 1 – 3). It is obtained by marking all entries in row 1, column 3 of A_1 and $A_1[3, 1]$ to ∞.

$$A_3 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{bmatrix}$$

$$\hat{c}(3) = \hat{c}(1) + A_1[1, 3] + r = 25 + 17 + 0 = 52$$

(iii) Let A_4 be the reduced matrix assigned to node 4 (path 1 – 4). It is obtained by marking all entries in row 1, column 4 of A_1 and $A_1[4, 1]$ to ∞.

$$A_4 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

$$\hat{c}(4) = \hat{c}(1) + A_1[1, 4] + r = 25 + 0 + 0 = 25$$

(iv) Let A_5 be the reduced matrix assigned to node 5 (path 1 – 5). It is obtained by marking all entries in row 1, column 5of A_1 and $A_1[5, 1]$ to ∞.

$$A_5 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & 2 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 15 & 3 & 14 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix} \quad \begin{matrix} r \\ 2 \\ 3 \\ \\ 5 \end{matrix}$$

Row reduction

$$\Downarrow$$

$$A_5 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 12 & 0 & 11 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$$

$$\therefore r = 5$$

$$\therefore \hat{c}(5) = \hat{c}(1) + A_1[1, 5] + r = 25 + 1 + 5 = 31$$

- Since all children of node 1 are explored it is dead. The live nodes are 2, 3, 4, 5. Out of them node 4 with a minimum $\hat{c}(\cdot)$ value becomes a next E-node generating its children: node 6 (path 1 – 4 – 2), node 7 (path 1 – 4 – 3), node 8 (path 1 – 4 – 5).

(3)

L	✗, 2, 3, 4, 5, 6, 7, 8
E	✗, 4
D	1

(i) Let A_6 be the reduced matrix assigned to node 6 (path 1 – 4 – 2). It is obtained by marking all entries in row 4, column 2of A_4 and $A_4[2, 1]$ to ∞.

$$A_6 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

$$\hat{c}(6) = \hat{c}(4) + A_4[4, 2] + r = 25 + 3 + 0 = 28$$

(ii) Let A_7 be the reduced matrix assigned to node 7 (path 1 – 4 – 3). It is obtained by marking all entries in row 4, column 3 of A_4 and $A_4[3, 1]$ to ∞.

$$A_7 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ \infty & 3 & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & 0 & \infty & \infty & \infty \end{bmatrix} \begin{matrix} r \\ \\ 2 \\ \\ \hline 2 \end{matrix}$$

Row reduction

$$\Downarrow$$

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & 0 & \infty & \infty & \infty \end{bmatrix}$$

$r = 11$ $= \mathbf{11}$

Column reduction

$$\Downarrow$$

$$A_7 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & \infty \end{bmatrix}$$

$\therefore r = 2 + 11 = \mathbf{13}$

$\hat{c}(7) = \hat{c}(4) + A_4[4, 3] + r = 25 + 12 + 13 = \mathbf{50}$

(iii) Let A_8 be the reduced matrix assigned to node 8 (path 1 – 4 – 5). It is obtained by marking all entries in row 4, column 5 of A_4 and $A_4[5, 1]$ to ∞.

$$A_8 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix} \begin{matrix} r \\ \\ 11 \\ \\ \\ \hline 11 \end{matrix}$$

Row reduction

$$\Downarrow$$

$$A_8 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix}$$

$\therefore \mathbf{r = 11}$

$\therefore \hat{c}(8) = \hat{c}(8) + A_4[4, 5] + r = 25 + 0 + 11 = \mathbf{36}$

- Since all children of node 4 are explored it is dead. The live nodes are 2, 3, 5, 6, 7, 8. Out of them node 6 with a minimum $\hat{c}(\cdot)$ value becomes a next E-node generating its children: node 9 (path 1 – 4 – 2 – 3) and node 10 (path 1 – 4 – 2 – 3 – 5).

(4)

L	~~1~~, 2, 3, ~~4~~, 5, 6, 7, 8, 9, 10
E	~~1~~, ~~4~~, 6
D	1, 4

(i) Let A_9 be the reduced matrix assigned to node 9 (path 1 – 4 – 2 – 3). It is obtained by marking all entries in row 2, column 3of A_6 and $A_6[3, 1]$ to ∞.

$$A_9 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & \infty & \infty & \infty \end{bmatrix} \begin{matrix} r \\ \\ 2 \\ \\ 11 \\ \hline 13 \end{matrix}$$

Row reduction

$$\Downarrow$$

$$A_9 = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix}$$

$\therefore$ **r = 13**

$\therefore \hat{c}(9) = \hat{c}(6) + A_6[2, 3] + r = 28 + 11 + 13 = 52$

(ii) Let A_{10} be the reduced matrix assigned to node 10 (path 1 – 4 – 2 – 5). It is obtained by marking all entries in row 2, column 5 of A_6 and $A_6[5, 1]$ to ∞.

$$A_{10} = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$$

$\hat{c}(10) = \hat{c}(6) + A_6[2, 5] + r = 28 + 0 + 0 = 28$

- Since all children of node 6 are explored, it is dead. The live nodes are 2, 3, 5, 7, 8, 9, 10. Out of them node 11 with a minimum $\hat{c}(\cdot)$ becomes a next E-node generating its child as node 11 (path 1 – 4 – 2 – 5 – 3).

(5)

L	~~1~~, 2, 3, ~~4~~, 5, ~~6~~, 7, 8, 9, 10, 11
E	~~1~~, ~~4~~, ~~6~~, 10
D	1, 4, 6

(i) Let A_{11} be the reduced matrix assigned to node 11 (path 1 – 4 – 2 – 5 – 3). It is obtained by marking all entries in row 5, column 3 of A_{10} and $A_{10}[3, 1]$ to ∞.

$$A_{11} = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

$\hat{c}(11) = \hat{c}(10) + A_{10}[5, 3] + r = 28 + 0 + 0 = 28$

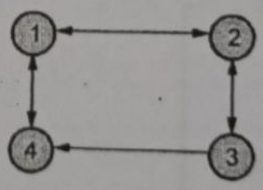
- Now *upper* is updated as ***upper*** $= \hat{c}(11) = 28$.
- Since all children of node 10 are explored it is dead. The live nodes are 2, 3, 5, 7, 8, 9, 11. Node 11 cannot be further explored, so it is dead. Remaining live nodes have their $\hat{c}(\cdot)$ values > (*upper* = 28). So all of them are killed. The least cost answer node 11 gives an optimal solution.

L	~~1~~, ~~2~~, ~~3~~, ~~4~~, ~~5~~, ~~6~~, ~~7~~, ~~8~~, ~~9~~, ~~10~~, ~~11~~
E	~~1~~, ~~4~~, ~~6~~, ~~10~~
D	1, 4, 6, 10, 2, 3, 5, 7, 8, 9, 11

- **Thus the final shortest tour is 1 – 4 – 2 – 5 – 3 – 1 with the cost of 28 units.**

Ex. 6.4.2 : What is a traveling salesperson problem? Find the solution to the following traveling salesperson problem using Branch and Bound approach.

(16 Marks)



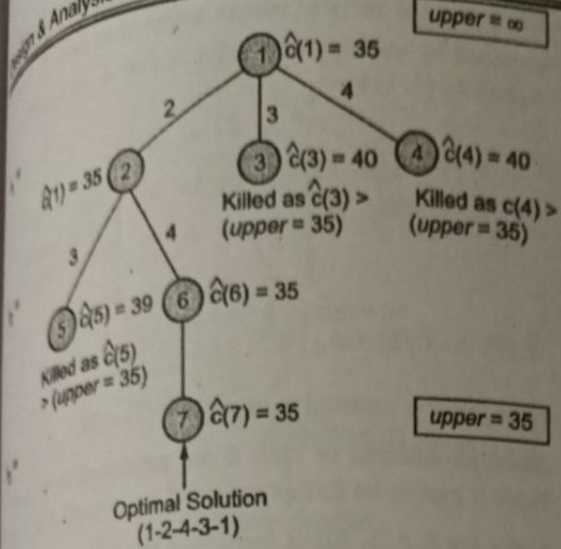
$$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

☑ **Soln. :**

Traveling salesperson Problem: Refer to the problem description described above.

TSP Solution by branch and bound :

- We solve the given instance of TSP by LCBB.
- For the same, we use the dynamic reduction of a cost matrix. We maintain the lists of live nodes(L), E-nodes(E) and dead nodes(D).
- The portion of a state space tree is shown in Fig. Ex. 6.4.2.



(TF8)Fig. Ex. 6.4.2 : The state space tree

1) We first determine a reduced matrix A_1 for node 1 from a given cost matrix .Initially *upper* = ∞.

L	1
E	-
D	-

$$\begin{bmatrix} \infty & 10 & 15 & 20 \\ 5 & \infty & 9 & 10 \\ 6 & 13 & \infty & 12 \\ 8 & 8 & 9 & \infty \end{bmatrix} \quad \begin{matrix} r \\ 10 \\ 5 \\ 6 \\ 8 \end{matrix}$$

$$29$$

Row reduction

$$\Downarrow$$

$$\begin{bmatrix} \infty & 0 & 5 & 10 \\ 0 & \infty & 4 & 5 \\ 0 & 7 & \infty & 6 \\ 0 & 0 & 1 & \infty \end{bmatrix}$$

$r \quad 1 \quad 5 = 6$

Column reduction

$$\Downarrow$$

$$A_1 = \begin{bmatrix} \infty & 0 & 4 & 5 \\ 0 & \infty & 3 & 0 \\ 0 & 7 & \infty & 1 \\ 0 & 0 & 0 & \infty \end{bmatrix}$$

$\therefore r = 29 + 6 = 35$

$\therefore \hat{c}(1) = r = 35$

Since node 1 is the only live node it is explored to generate its children: node 2 (path 1 – 2), node 3 (path 1 – 3), node 4 (path 1 – 4).

(2)

L	1, 2, 3, 4
E	1
D	--

(i) Let A_2 be the reduced matrix assigned to node 2. It is obtained by marking all entries in row 1, column 2 of A_1 and $A_1[2, 1]$ to ∞.

$$A_2 = \begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & 3 & 0 \\ 0 & \infty & \infty & 1 \\ 0 & \infty & 0 & \infty \end{bmatrix}$$

$\hat{c}(2) = \hat{c}(1) + A_1[1, 2] + r = 35 + 0 + 0 = 35$

(ii) Let A_3 be the reduced matrix assigned to node 3. It is obtained by marking all entries in row 1, column 3 of A_1 and $A_1[3, 1]$ to ∞.

$$A_3 = \begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 0 \\ \infty & 7 & \infty & 1 \\ 0 & 0 & \infty & \infty \end{bmatrix} \quad \begin{matrix} R \\ \\ \\ 1 \end{matrix}$$

$= 1$

Row reduction

$$\Downarrow$$

$$A_3 = \begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 0 \\ \infty & 6 & \infty & 0 \\ 0 & 0 & \infty & \infty \end{bmatrix}$$

$\therefore r = 1$

$\hat{c}(3) = \hat{c}(1) + A_1[1, 3] + r = 35 + 4 + 1 = 40$

(iii) Let A_4 be the reduced cost matrix assigned to node 4. It is obtained by marking all entries in row 1, column 4 of A_1 and $A_1[4, 1]$ to ∞.

$$A_4 = \begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & 3 & \infty \\ 0 & 7 & \infty & \infty \\ \infty & 0 & 0 & \infty \end{bmatrix}$$

$\hat{c}(4) = \hat{c}(1) + A_1[1, 4] + r = 35 + 5 + 0 = 40$

- Since all children of node 1 are generated, it is dead. The live nodes are 2, 3, 4. Out of them node 2 with a minimum $\hat{c}(\cdot)$ value is explored to generate its children: node 5 (path 1 – 2 – 3) and node 6 (path 1 – 2 – 4).

(3)

L	~~1~~, 2, 3, 4, 5, 6
E	~~1~~, 2
D	1

(i) Let A_5 be the reduced matrix assigned to node 5. It is obtained by marking all entries in row 2, column 3 of A_2 and $A_2[3, 1]$ to ∞.

$$A_5 = \begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 1 \\ 0 & \infty & \infty & \infty \end{bmatrix}$$

r 1 = 1

Column reduction

⇓

$$A_5 = \begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 0 \\ 0 & \infty & \infty & \infty \end{bmatrix}$$

∴ r = 1

$$\hat{c}(5) = \hat{c}(2) + A_2[2, 3] + r = 35 + 3 + 1 = 39$$

(ii) Let A_6 be the reduced cost matrix assigned to node 6. It is obtained by marking all entries in row 2, column 4 of A_2 and $A_2[4, 1]$ to ∞.

$$A_6 = \begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty \end{bmatrix}$$

$$\hat{c}(6) = \hat{c}(2) + A_2[2, 4] + r = 35 + 0 + 0 = 35$$

- Since all children of node 2 are generated, it is dead. The live nodes are 3, 4, 5, 6. Out of them node 6 with a minimum $\hat{c}(\cdot)$ is explored to generate its child: node 7 (path 1 – 2 – 4 – 3).

(4)

L	~~1~~, ~~2~~, 3, 4, 5, 6, 7
E	~~1~~, ~~2~~
D	1, 2

- Let A_7 be the reduced matrix assigned to node 7. It is obtained by marking all entries in row 4, column 3 of A_6 and $A_6[3, 1]$ to ∞.

$$A_7 = \begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \end{bmatrix}$$

$$\hat{c}(7) = \hat{c}(6) + A_6[4, 3] + r = 35 + 0 + 0 = 35$$

- Now *upper* is updated as ***upper*** $= \hat{c}(7) = 35$.
- Since all children of node 6 are generated it is dead. Node 7 cannot be further explored, so it dead. The live nodes are 3, 4, 5. All of them have their $\hat{c}(\cdot)$ value > (upper = 35). So they are killed. The least cost answer node 7 gives an optimal solution.

L	~~1~~, ~~2~~, ~~3~~, ~~4~~, ~~5~~, ~~6~~, ~~7~~
E	~~1~~, ~~2~~, ~~6~~
D	1, 2, 6, 7, 3, 4, 5

- **Thus the final shortest tour is 1 – 2 – 4 – 3 – 1 with the cost of 35 units.**

Ex. 6.4.3 : Describe the Traveling salesperson Problem. Solve the following instance of TSP by LC-branch and bound.

$$\begin{bmatrix} \infty & 15 & 6 & 20 \\ 5 & \infty & 9 & 15 \\ 6 & 13 & \infty & 12 \\ 8 & 8 & 19 & \infty \end{bmatrix}$$

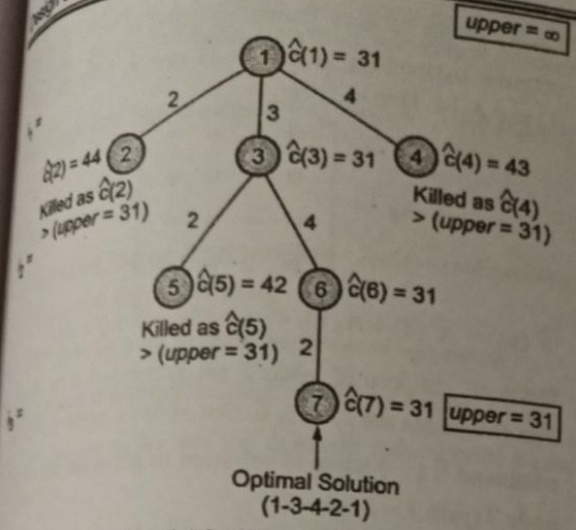
(18 Marks)

☑ **Soln. :**

▶ **Traveling salesperson problem:** Refer to problem description described above.

▶ **TSP Solution by Branch and Bound approach:**

- We solve the given instance of TSP by LCBB.
- For the same, we use the dynamic reduction of a cost matrix.
- We maintain the lists of live nodes (L), E-nodes (E) and dead node (D).
- The portion of a state space tree is shown in Fig. Ex. 6.4.3.



(1F8)**Fig. Ex. 6.4.3 : The state space tree**

(1) We first determine a reduced matrix A_1 for node 1 from a given cost matrix. Initially *upper* = ∞.

L	1
E	–
D	–

$$\begin{bmatrix} \infty & 15 & 6 & 20 \\ 5 & \infty & 9 & 15 \\ 6 & 13 & \infty & 12 \\ 8 & 8 & 19 & \infty \end{bmatrix} \quad \begin{matrix} r \\ 6 \\ 5 \\ 6 \\ 8 \end{matrix}$$

$$= 25$$

Row reduction

$$\Downarrow$$

$$\begin{bmatrix} \infty & 9 & 0 & 14 \\ 0 & \infty & 4 & 10 \\ 0 & 7 & \infty & 6 \\ 0 & 0 & 11 & \infty \end{bmatrix}$$

$r \qquad 6 = 6$

Column reduction

$$\Downarrow$$

$$A_1 = \begin{bmatrix} \infty & 9 & 0 & 8 \\ 0 & \infty & 4 & 4 \\ 0 & 7 & \infty & 0 \\ 0 & 0 & 11 & \infty \end{bmatrix}$$

$\therefore r = 25 + 6 = 31$

$\therefore \hat{c}(1) = r = 31$

- Since node 1 is the only live node, it is explored to generate its children: node 2 (path 1 – 2), node 3 (1 – 3), node 4 (path 1 – 4).

(2)

L	1, 2, 3, 4
E	1
D	--

(i) Let A_2 be the reduced matrix assigned to node 2. It is obtained by marking all entries in row 1, column 2 of A_1 and $A_1[2, 1]$ to ∞.

$$A_2 = \begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & 4 & 4 \\ 0 & \infty & \infty & 0 \\ 0 & \infty & 11 & \infty \end{bmatrix} \quad \begin{matrix} R \\ \\ 4 \\ \\ \end{matrix}$$

$$= 4$$

Row reduction

$$\Downarrow$$

$$A_2 = \begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & 0 \\ 0 & \infty & \infty & 0 \\ 0 & \infty & 11 & \infty \end{bmatrix}$$

$\therefore r = 4$

$\hat{c}(2) = \hat{c}(1) + A_1[1, 2] + r = 31 + 9 + 4 = 44$

(ii) Let A_3 be the reduced matrix assigned to node 3. It is obtained by marking all entries in row 1, column 3 of A_1 and $A_1[1, 3]$ to ∞.

$$A_3 = \begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & 4 \\ \infty & 7 & \infty & 0 \\ 0 & 0 & \infty & \infty \end{bmatrix}$$

$\hat{c}(3) = \hat{c}(1) + A_1[1, 3] + r = 31 + 0 + 0 = 31$

(iii) Let A_4 be the reduced matrix assigned to node 4. It is obtained by marking all entries in row 1, column 4 of A_1 and $A_1[4, 1]$ to ∞.

$$A_4 = \begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & 4 & \infty \\ 0 & 7 & \infty & \infty \\ \infty & 0 & 11 & \infty \end{bmatrix} \quad r \quad 4 \quad = 4$$

Column reduction

$$\Downarrow$$

$$A_4 = \begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & 0 & \infty \\ 0 & 7 & \infty & \infty \\ \infty & 0 & 7 & \infty \end{bmatrix}$$

$\therefore$ r = 4

$$\hat{c}(4) = \hat{c}(1) + A_1[1, 4] + r = 31 + 8 + 4 = 43$$

- Since all children of node 1 are generated it is dead. The live nodes are 2, 3, 4. Out of them node 3 with a minimum $\hat{c}(\cdot)$ value is explored to generate nodes 5 (path 1 – 3 – 2) and node 6 (path 1 – 3 – 4).

(3)

L	~~1~~, 2, 3, 4, 5, 6
E	~~1~~, 3
D	~~1~~

(i) Let A_5 be the reduced matrix assigned to node 5. It is obtained by marking all entries in row 3, column 2 of A_3 and $A_3[2, 1]$ to ∞.

$$A_5 = \begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 4 \\ \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty \end{bmatrix} \quad \begin{matrix} r \\ 4 \\ \\ \\ \hline = 5 \end{matrix}$$

Row reduction

$$\Downarrow$$

$$A_5 = \begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty \end{bmatrix}$$

$\therefore r = 5$

$$\hat{c}(5) = \hat{c}(3) + A_3[3, 2] + r = 31 + 7 + 4 = 42$$

(ii) Let A_6 be the reduced matrix assigned to node 6. It is obtained by marking all entries in row 3, column 4 of A_3 and $A_3[4, 1]$ to ∞.

$$A_6 = \begin{bmatrix} \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty \end{bmatrix}$$

$$\hat{c}(6) = \hat{c}(3) + A_3[3, 4] + r = 31 + 0 + 0 = 31$$

- Since all children of node 3 are generated it is dead. The live nodes are 2, 4, 5, 6. Out of them node 6 with a minimum $\hat{c}(\cdot)$ value is explored to generate its child node 7 (path 1 – 3 – 4 – 2).

(4)

L	~~1~~, 2, ~~3~~, 4, 5, 6, 7
E	~~1~~, ~~3~~, 6
D	1, 3

(i) Let A_7 be the reduced matrix of assigned to node 7. It is obtained by marking all entries in row 4, column 2 of A_6 and $A_6[2, 1]$ to ∞.

$$A_7 = \begin{bmatrix} \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \end{bmatrix}$$

$$\hat{c}(7) = \hat{c}(6) + A_6[4, 2] + r = 31 + 0 + 0 = 31$$

- Now *upper* is updated as ***upper*** $= \hat{c}(7) = 31$.
- Since all children of node 6 are explored it is dead. Node 7 cannot be further explored, so it is dead. The live nodes are 2, 4, 5. All of them are having their $\hat{c}(\cdot) >$ (upper = 31). So they are killed. The least cost answer node 7 gives an optimal solution.

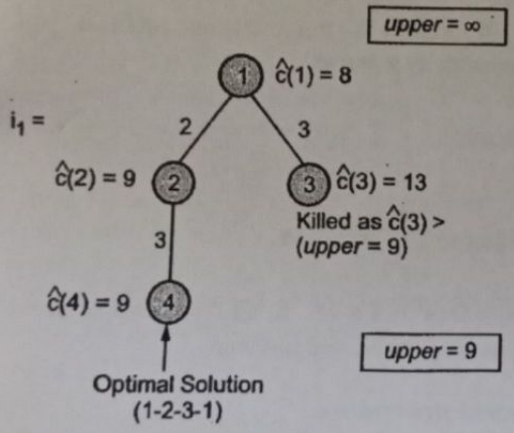
L	~~1~~, ~~2~~, ~~3~~, ~~4~~, ~~5~~, ~~6~~, ~~7~~
E	~~1~~, ~~3~~, ~~6~~
D	1, 3, 6, 7, 2, 4, 5

- **The final optimal tour by LCBB solution is 1-3-4-2-1 and the cost of tour is 31 units.**

Ex. 6.4.4 : What is a traveling salesperson problem? Find the solution to the following travelling salesperson problem using branch and bound method. $\begin{bmatrix} \infty & 4 & 2 \\ 3 & \infty & 4 \\ 1 & 8 & \infty \end{bmatrix}$ **(12 Marks)**

☑ **Soln. :**

- **Traveling salesperson problem :** Refer to the problem description described above.
- **TSP Solution by Branch and Bound approach :**
- We solve the given instance of TSP by LCBB.
- For the same, we use the dynamic reduction of a cost matrix.
- We maintain the lists of live nodes (L), E-nodes (E), dead nodes (D).
- The portion of a state space tree is shown in Fig. Ex. 6.4.4.



(1F10)**Fig. Ex. 6.4.4 : The state space tree**

(1) Initially upper = ∞. Node 1 is created. We first determine a reduced matrix A_1 for node 1 from a given cost matrix.

L	1
E	–
D	–

$$\begin{bmatrix} \infty & 4 & 2 \\ 3 & \infty & 4 \\ 1 & 8 & \infty \end{bmatrix} \quad \begin{matrix} r \\ 2 \\ 3 \\ 1 \end{matrix}$$

$$= 6$$

Row reduction

$$\Downarrow$$

$$\begin{bmatrix} \infty & 2 & 0 \\ 0 & \infty & 1 \\ 0 & 7 & \infty \end{bmatrix}$$

$r = 2$

Column reduction

$$\Downarrow$$

$$A_1 = \begin{bmatrix} \infty & 0 & 0 \\ 0 & \infty & 1 \\ 0 & 5 & \infty \end{bmatrix}$$

$\therefore r = 6 + 2 = 8$

$\therefore \hat{c}(1) = r = 8$

Since node 1 is the only live node, it is explored to generate node 2 (path 1-2), node 3 (path 1-3).

(2)

L	1, 2, 3
E	1
D	--

(i) Let A_2 be the reduced matrix assigned to node 2. It is obtained by marking all entries in row 1, column 2 of A_1 and A_1 [2, 1] to ∞.

$$A_2 = \begin{bmatrix} \infty & \infty & \infty \\ \infty & \infty & 1 \\ 0 & \infty & \infty \end{bmatrix} \quad \begin{matrix} r \\ 1 \end{matrix}$$

$$= 1$$

Row reduction

$$\Downarrow$$

$$A_2 = \begin{bmatrix} \infty & \infty & \infty \\ \infty & \infty & 0 \\ 0 & \infty & \infty \end{bmatrix}$$

$\therefore r = 1$

$\hat{c}(2) = \hat{c}(1) + A_1[1, 2] + r = 8 + 0 + 1 = 9$

(ii) Let A_3 be the reduced matrix assigned to node 3. It is obtained by marking all entries in row 1, column 3 of A_1 and A_1[3, 1] to ∞.

$$A_3 = \begin{bmatrix} \infty & \infty & \infty \\ 0 & \infty & \infty \\ \infty & 5 & \infty \end{bmatrix} \begin{matrix} r \\ 5 \\ = 5 \end{matrix}$$

Row reduction

$$\Downarrow$$

$$A_3 = \begin{bmatrix} \infty & \infty & \infty \\ 0 & \infty & \infty \\ \infty & 0 & \infty \end{bmatrix}$$

$$\therefore r = 5$$

$$\hat{c}(3) = \hat{c}(1) + A_1[1,3] + r = 8 + 0 + 5 = 13$$

- Since all children of node 1 are generated, it is dead. The live nodes are 2, 3. Out of them node 2 with a minimum $\hat{c}(\cdot)$ value is explored to generate node 4 (path 1-2-3).

(3)

L	✗, 2, 3
E	✗, 2
D	1

(i) Let A_4 be the reduced matrix assigned to node 4. It is obtained by marking all entries in row 2, column 3 of A_2 and $A_2[3, 1]$ to ∞.

$$A_4 = \begin{bmatrix} \infty & \infty & \infty \\ \infty & \infty & \infty \\ \infty & \infty & \infty \end{bmatrix}$$

$$\hat{c}(4) = \hat{c}(2) + A_2[2,3] + r = 9 + 0 + 0 = 9$$

- Now *upper* is updated as ***upper*** $= \hat{c}(4) = 9$.
- Since all children of node 2 are explored it is dead. Node 4 cannot be explored further, so it is dead. Node 3 is the only live node. But $\hat{c}(3) > (upper = 9)$, so it is killed. The least cost answer node 4 gives an optimal solution.

L	✗, ✗, ✗, ✗
E	✗, ✗
D	1, 2, 4, 3

- **Thus the final shortest tour is 1 - 2 - 3 - 1 with a cost of 9 units.**

6.5 0/1 KNAPSACK PROBLEM

The 0/1 knapsack problem is a maximization problem that asks to maximize the profit by adding the given items to a knapsack considering its capacity. To solve this maximization problem by the B&B algorithms the sign of objective function is changed to describe this "maximization of profit problem" as the "minimization of loss problem".

Problem description

- Consider a knapsack or bag with capacity M. Suppose there are n items available. Each item i, $1 \le i \le n$ has a weight w_i and it gives a profit of $p_i x_i$ if its portion x_i, $x_i = 0$ or 1 is kept into the knapsack.
- 0/1 knapsack problem is converted as a minimization problem by changing the objective function $\sum p_i x_i$ to $-\sum p_i x_i$, $1 \le i \le n$, $x_i = 0$ or 1.
- It implies that the objective function $\sum p_i x_i$ gets maximized iff $-\sum p_i x_i$ is minimized.
- The 0/1 knapsack minimization problem can be mathematically given as :

$$\text{Minimize: } -\sum_{i=1}^{n} p_i x_i$$

$$\text{Subject to } \sum_{i=1}^{n} w_i x_i \le M$$

- and $x_i = 0$ or 1, $1 \le i \le n$, all the values of profit and weights are positive real numbers.

General procedure

(1) Construct a state space tree to solve a 0/1 knapsack problem by the B&B method.

(2) Consider *upper* as an upper bound on the cost of the least-cost solution. Initially, set $upper = \infty$.

(3) At each live node k, compute the lower and upper bounds on solutions feasible from k.

(i) A heuristic function $\hat{c}(k)$ is the estimation of cost c(k) and it defines the lower bound.

(ii) A function u(k)gives a simple upper bound on the cost of the least cost answer state in the subtree of k. $\hat{c}(k) \le c(k) \le u(k)$.

(4) Since the objective function has a negative sign, define the lower bound $\hat{c}(k)$ by computing the greedy solution to a knapsack problem with condition $0 \le x_i \le 1$ and the upper bound u(k) by computing the solution to a knapsack problem with condition $x_i = 0$ or 1.

Note : The absolute value of a greedy solution to a knapsack problem defines the upper bound But with negative sign this larger absolute value becomes the lower bound in the case of 0/1 knapsack minimization problem.

(5) Kill all the live nodes k with $\hat{c}(k) > upper$.

(6) Select the next E-node by applying any of the suitable search techniques (E.g., LC search for LCBB, BFS for FIFOBB and DFS for LIFOBB)

(7) Whenever a new answer state is obtained, the value of the *upper* is updated accordingly.

(8) Repeat the steps (3) to (7) until all live nodes become dead. The least-cost answer state gives an optimal solution to the problem.

Algorithm to compute $\hat{c}(\cdot)$

```
Algorithm Bound_Bk(k, cp, cw)
/*The objective function of the 0/1 knapsack minimization problem has a negative sign. So, this algorithm is used to compute a lower bound ĉ(k) by finding the greedy solution to a knapsack problem with condition 0 ≤ x_i ≤ 1.
Input: M is the capacity of a knapsack; n is the number of given items. W[1: n] and P[1:n] are the weights and profits associated with n items. Items are arranged in decreasing order of their profit to weight ratio, so
P[i]/W[i] ≥ P[i+1]/W[i+1]. A solution vector X[1: n] is a global array whose first (k−1) values are decided. cp is a current total profit, cw is the current total weight. k is the last removed item.
Output: Gives a lower bound on the negative-signed profit feasible at node k in the state space tree.*/
{
  b := cp;
  c := cw;
  for (i := k+1; i ≤ n; i++)
  {
    c := c + W[i];
    if (c < M)
      b := b − P[i];          //It gives −∑ p_i x_i
    else
      return b + (1− (c − M)/W[i]) * P[i];
                /*Calculation as per the fractional knapsack i.e. greedy solution where 0 ≤ x_i ≤ 1.*/
  }
  return b;
}
```

Algorithm to compute an upper bound u(·)

UQ. Write and explain the upper bound function for 0/1 Knapsack problem.

(SPPU - Q. 7(b), Dec. 17, 8 Marks)

```
Algorithm Bound_B&B(k, cp, cw)
/* It is a bounding function that gives an upper bound on the negative-signed profit to solve a 0/1 knapsack minimization problem by B&B.
Input: M is the capacity of a knapsack; n is the number of given items. W[1: n] and P[1: n] are the weights and profits associated with n items. Items are arranged in decreasing order of their profit to weight ratio, so P[i]/W[i] ≥ P[i+1]/W[i+1]. A solution vector X[1: n] is a global array whose first (k-1) values are decided. cp is a current total profit, cw is the current total weight. k is the last removed item.
Output: Gives an upper bound on − ∑ p_i x_i where x_i=0 or 1at node k in state space tree of a 0/1 knapsack minimization problem.*/
{
    b := cp;
    c := cw;
    for (i := k+1; i ≤ n; i++)
    {
      if (c+ W[i] ≤ M)
      {
        c := c + W[i];
        b := b − P[i];          //It gives −∑ p_i x_i
      }
    }
  return b;
}
```

6.5.1 0/1 Knapsack Problem by LCBB

Here, a node with the least $\hat{c}(\cdot)$ is selected as a next E-node. All live nodes are arranged in a min-heap data structure.

Ex. 6.5.1 : Solve the following instance of 0/1 knapsack problem by Least Cost branch and bound approach : n = 4; p(1:4) = {10, 10, 12, 18}; w(1:4) = {2, 4, 6, 9}.M = 15.

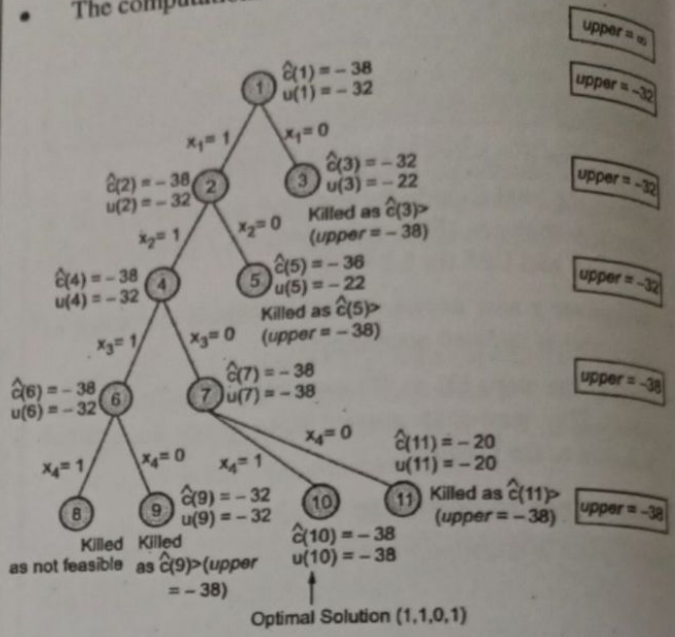
☑ **Soln. :**

- To solve a given instance of the knapsack problem by LCBB, we first check that whether the given items are arranged in decreasing order of the ratio of P_i/W_i, $1 \le i \le n$. Here n = 4 and M = 15.

Item	P_i	W_i	P_i/W_i
1	10	2	5
2	10	4	2.5
3	12	6	2
4	18	9	2

- The given items are arranged in decreasing order of the ratio of P_i/W_i, $1 \le i \le 4$.
- To solve a given instance of the knapsack problem by LCBB, we convert it into a minimization problem by setting the objective function as: Minimize-$\sum p_i x_i$ where $x_i = 0$ or 1.
- We construct a state space tree with fixed size tuple formulation as shown in Fig. Ex. 6.5.1. At each node k, the values of the lower bound $\hat{c}(k)$ and the upper bound u(k) is computed so that $\hat{c}(k) \le c(k) \le u(k)$.
- For computing $\hat{c}(k)$, we use an algorithm Bound_Bk(k,cp,cw) that gives the greedy solution to a knapsack problem where $0 \le x_i \le 1$. k,cp,cw have usual meanings.
- For computing u(k), we use an algorithm Bound_B&B(k,cp, cw) that gives the solution to a knapsack problem where $x_i = 0$ or 1.
- Initially, we set *upper* = ∞ and we kill the node k if $\hat{c}(k) > upper$.
- As per the LCBB, each time a live node with a minimum $\hat{c}(\cdot)$ is selected as a next E-node.
- Whenever a new answer state is obtained, the value of the *upper* is updated accordingly.
- Finally, the least-cost answer state gives an optimal solution to the problem.
- We maintain the lists of live nodes(L), E-nodes(E) and dead nodes(D).

- The computations of $\hat{c}(\cdot)$ and u(·) are as follows :



(1F1)**Fig. Ex. 6.5.1 : The state space tree**

(1) Initially, node1 is created.

L:	1
E:	--
D:	--

- $\hat{c}(1)$ = Bound_Bk(0, 0, 0) = 0 – 10 –10–12–(3/9*18) = **–38**...(Items 1, 2, 3 can be completely added and the fraction 3/9 of item4 can be added to a knapsack to give a greedy solution.)
- **u(1)** = Bound_B&B(0, 0, 0) = 0–10–10–12 = – 32 ...(Items 1, 2, 3 can be completely added and item 4 cannot be added as its addition exceeds a knapsack capacity M.)
- Since $\hat{c}(1) < upper$ (i.e. –38 < ∞), it is not killed.
- As node 1 is the only live node in the list, it is explored to generate its children: node2 (x_1 = 1) and node3 (x_1 = 0). Nodes2 and 3 are added in the list of live nodes. ***upper*** **= u(1) = – 32.**

(2)

L:	1, 2, 3
E:	1
D:	--

- $\hat{c}(2)$ = Bound_Bk(1, –10,2) = – 10 – 10 – 12 – (3/9*18) = **–38**...(Items 1 is already added. Items 2, 3 can be completely added and the fraction 3/9 of item4 can be added to a knapsack to give a greedy solution.)

- **u(2)** = Bound_B&B(1, –10,2) = – 10 – 10 – 12 = **–32**...(Items 1 is already added. Items 2, 3 can be completely added and item 4 cannot be added as its addition exceeds a knapsack capacity M.)
- Since ĉ(2) < *upper* (i.e. –38 < –32), it is not killed.
- **ĉ(3)** = Bound_Bk(1,0,0) = 0 – 10 – 12 – (5/9*18) = **–32**...(Item 1 is not added. Items 2,3can be completely added and the fraction 5/9 of item4 can be added to a knapsack to give a greedy solution.)
- **u(3)** = Bound_B&B(1, 0, 0) = 0 – 10 – 12 = **–22** ...(Item 1 is not added. Items 2, 3 can be completely added and item 4 cannot be added as its addition exceeds a knapsack capacity M.)
- Since ĉ(3) = *upper* (i.e. – 32 = –32), it is not killed.
- As all children of node1 are explored it becomes a dead node. So, there are two nodes2 and 3 in the list of live nodes. Since ĉ(2) < ĉ(3), node2 becomes the next E-node generating its children: node4 (x_2 = 1) and node5(x_2 = 0). *upper* = –32.

(3)

L:	✗, 2, 3, 4 ,5
E:	✗, 2
D	1

- **ĉ(4)** = Bound_Bk(2, –20,6) = –20 – 12 – (3/9*18) = **–38**...(Items 1,2 are already added. Item3 can be completely added and the fraction 3/9 of item4 can be added to a knapsack to give a greedy solution.)
- **u(4)** = Bound_B&B(2, –20,6) = –20 – 12 = **–32** ...(Items 1,2 are already added. Item3 can be completely added and item4 cannot be added as its addition exceeds a knapsack capacity M.)
- Since ĉ(4) < *upper* (i.e. –38 < –32), it is not killed.
- **ĉ(5)** = Bound_Bk(2,–10,2) = –10 – 12 – (7/9*18) = **–36**...(Item 1 is added and item 2 is not added. Item 3 can be completely added and the fraction 7/9 of item4 can be added to a knapsack to give a greedy solution.)
- **u(5)** = Bound_B&B(2, –10,2) = –10 – 12 = **–22** ...(Item 1 is added and item 2 is not added. Item 3 can be completely added and item 4 cannot be added as its addition exceeds a knapsack capacity M.)
- Since ĉ(5) < *upper* (i.e.–36 < –32), it is not killed.
- As all children of node2 are explored it becomes a dead node. Now the live nodes are 3, 4, 5. Out of them, node4 with a minimum ĉ(·)value becomes the next E-node generating its children: node6 (x_3 = 1) and node7 (x_3 = 0). *upper* = –32.

(4)

L:	✗, ✗, 3,4,5,6,7
E:	✗, ✗, 4,
D:	1, 2

- **ĉ(6)** = Bound_Bk(3, –32,12) = –32 – (3/9*18) = **–38**...(Items 1,2,3 are already added and the fraction 3/9 of item4 can be added to a knapsack to give a greedy solution.)
- **u(6)** = Bound_B&B(3, –32,12) = –32 – 0 = **–32** ...(Items 1,2,3 are already added and item 4 cannot be added as its addition exceeds a knapsack capacity M.)
- Since ĉ(6) < *upper* (i.e. –38 < –32), it is not killed.
- **ĉ(7)** = Bound_Bk(3, –20,6) = –20 –18 = **–38** ...(Items 1,2 are already added and item 3 is not added. Item 4 can be completely added to give a greedy solution.)
- **u(7)** = Bound_B&B(3, –20,6) = –20 –18 = **–38** ...(Items 1,2 are already added and item 3 is not added. Item 4 can be completely added.)
- Since ĉ(7) < *upper* (i.e. –38< –32), it is not killed.
- As all children of node 4 are explored it becomes a dead node. Now the live nodes are 3, 5, 6, 7. Out of them, nodes 6 and 7 have minimum ĉ(·) value = –38. Applying FIFO to break this tie node 6 becomes a next E-node generating its children: node 8 (x_4 = 1) and node9(x_4 = 0).***upper*** **= u(7) = –38.**

(5)

L:	✗, ✗, 3,4,5,6,7,8,9
E:	✗, ✗, ✗ 6
D:	1,2,4

- Node 8 with x_4 = 1 is killed as it is not feasible. The addition of item 4 exceeds a knapsack capacity M.
- **ĉ(9)** = Bound_Bk(4,–32,12) = –32 – 0 = **–32** ...(Items 1, 2, 3 are already added and the item4 is not added.)
- **u(9)** = Bound_B&B(4, –32,12) = –32 –0 = **–32** ...(Items 1,2,3 are already added and the item4 is not added.)
- Since ĉ(9) > *upper* (i.e. – 32 > –38), it is killed.
- As all children of node 6 are explored it becomes a dead node. Nodes 8 and 9 are also killed. Now, the live nodes are 3, 5, 7. Out of them, node 7 with a minimum ĉ(·) value becomes a next E-node generating its children: node 10 (x_4 = 1) and node 11(x_4 = 0). *upper* = –38.

(6)

L:	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
E:	1, 2, 4, 8, 7
D:	1,2,4,6,8,9

- $\hat{c}(10)$ = Bound_Bk(4, –38,15) = **–38**...(Items 1, 2, 4 are already added and item 3 is not added.)
- $u(10)$ = Bound_B&B(4, –38,15) = **–38**...(Items 1, 2, 4 are already added and item 3 is not added.)
- Since $\hat{c}(10) = upper$ (i.e. –38 = –38), it is not killed.
- $\hat{c}(11)$ = Bound_Bk(4, –20,6) = **–20**...(Items 1, 2 are already added and items 3,4arenot added.)
- $u(11)$ = Bound_B&B(4, –20,6) = **–20**...(Items 1, 2 are already added but Item 3 is not added.)
- Since $\hat{c}(11) > upper$ (i.e. –20 > –38), it is killed.
- As all children of node 7 are explored it becomes a dead node. Node 11 is also killed. Now the live nodes are 3, 5, 10. Out of them, nodes 3 and 5 have their $\hat{c}(\cdot)$ value > *upper* (i.e. $\hat{c}(3)$ = –32 > –38 and $\hat{c}(5)$ = –36 > –38). So nodes 3 and 5 are killed. Node 10 cannot be further explored. So the search terminates at node 10. It becomes dead. *upper* = –38.

L:	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
E:	1, 2, 4, 6, 7
D:	1,2,4,6,8,9,7,11,3,5,10

- Thus, an optimal solution is given by node10 as it is the least cost answer state. **The final solution tuple is (1, 1, 0, 1) with maximum profit of 38 units.**

Ex. 6.5.2 : Consider 0/1 knapsack instance n = 4 with capacity 10 kg such that (Refer to Table). Find maximum profit using Least Cost branch and bound (LCBB) method. Use fixed size formation of state space. **(8 Marks)**

Item	Profit (In Rs.)	Weight (in kg)
1	40	4
2	42	7
3	20	5
4	12	3

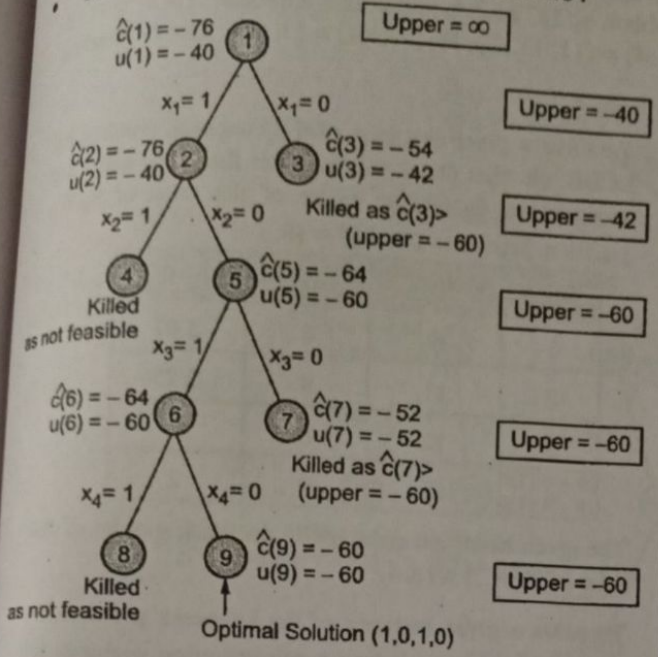
☑ **Soln. :**

- To solve a given instance of the knapsack problem by LCBB, we first check that whether the given items are arranged in decreasing order of the ratio of $P_i/W_i, 1 \le i \le n$. Here n = 4 and M = 10kg.

Item	P_i	W_i	P_i/W_i
1	40	4	10
2	42	7	6
3	20	5	4
4	12	3	4

- The given items are arranged in decreasing order of the ratio of $P_i/W_i, 1 \le i \le 4$.
- To solve a given instance of the knapsack problem by LCBB, we convert it into a minimization problem by setting the objective function as: Minimize-$\sum p_i x_i$ where $x_i = 0$ or 1.
- We construct a state space tree with fixed size tuple formulation as shown in Fig. Ex. 6.5.2. At each node k, the values of the lower bound $\hat{c}(k)$ and the upper bound $u(k)$ is computed so that $\hat{c}(k) \le c(k) \le u(k)$.
- For computing $\hat{c}(k)$, we use an algorithm Bound_Bk(k,cp,cw) that gives the greedy solution to a knapsack problem where $0 \le x_i \le 1$. k, cp, cw have usual meanings.
- For computing $u(k)$, we use an algorithm Bound_B&B(k,cp,cw) that gives the solution to a knapsack problem where $x_i = 0$ or 1.
- Initially, we set *upper* = ∞ and we kill the node k if $\hat{c}(k) > upper$.
- As per the LCBB, each time a live node with a minimum $\hat{c}(\cdot)$ is selected as a next E-node.
- Whenever a new answer state is obtained, the value of the *upper* is updated accordingly.
- Finally, the least-cost answer state gives an optimal solution to the problem.
- We maintain the lists of live nodes(L), E-nodes(E) and dead nodes(D).

The computations of $\hat{c}(\cdot)$ and $u(\cdot)$ are as follows :



(1F2)**Fig. Ex. 6.5.2 : The state space tree**

(1) **Initially, node1 is created.**

L:	1
E:	--
D:	--

- $\hat{c}(1)$ = Bound_Bk(0, 0, 0) = 0 – 40 – (6/7*42) = **–76** …(Item 1can be completely added and the fraction 6/7 of item 2 can be added to a knapsack to give a greedy solution.)
- **u(1)** = Bound_B&B(0, 0, 0) = 0 – 40 = **–40**…(Item 1can be completely added and item 2 cannot be added as its addition exceeds a knapsack capacity M.)
- Since $\hat{c}(1) <$ *upper* (i.e. –76 < ∞), it is not killed.
- As node 1 is the only live node in the list, it is explored to generate its children: node2 (x_1 = 1) and node3 (x_1 = 0). Nodes 2 and 3 are added in the list of live nodes. ***upper* = u(1) = –40.**

(2)

L:	1, 2, 3
E:	1
D:	--

- $\hat{c}(2)$ = Bound_Bk(1, –40, 4) = –40 – (6/7*42) = **–76** …(Item 1is already added and the fraction 6/7of item 2 can be added to a knapsack to give a greedy solution.)
- **u(2)** = Bound_B&B(1, –40, 4) = –40 + 0 = **–40** …(Item 1 is already added. Item 2 cannot be added as its addition exceeds a knapsack capacity M.)
- Since $\hat{c}(2) <$ *upper* (i.e. –76 < –40), it is not killed.
- $\hat{c}(3)$ = Bound_Bk(1, 0, 0) = 0 –42 – (3/5*20) = **–54**…(Item 1 is not added. Item 2 can be completely added and the fraction 3/5 of item 3 can be added to a knapsack to give a greedy solution.)
- **u(3)** = Bound_B&B(1, 0, 0) = 0 –42 = **–42** …(Item 1 is not added. Item 2 cannot be added as its addition exceeds a knapsack capacity M.)
- Since $\hat{c}(3) <$ *upper* (i.e. –54< – 40), it is not killed.
- As all children of node1 are explored it becomes a dead node. So, the live nodes are 2 and 3. Since $\hat{c}(2) < \hat{c}(3)$, node 2 becomes a next E-node generating its children: node 4 (x_2 = 1) and node 5(x_2 = 0). ***upper* = u(3) = –42.**

(3)

L:	~~1~~, 2, 3, 4, 5
E:	~~1~~, 2
D:	1

- Node 4 with x_2 = 1 is killed as it is not feasible. The addition of item 2 exceeds a knapsack capacity M.
- $\hat{c}(5)$ = Bound_Bk(2, –40,4) = –40 – 20 – (1/3*12) = **–64**…(Item 1 is already added and item 2 is not added. Item 3 can be completely added and the fraction 1/3 of item4 can be added to a knapsack to give a greedy solution.)
- **u(5)** = Bound_B&B(2, –40,4) = –40 – 20 = **–60**…(Item 1 is already added and item 2 is not added. Item 3 can be completely added and item 4 cannot be added as its addition exceeds a knapsack capacity M.)
- Since $\hat{c}(4) <$ *upper* (i.e. –64 < –42), it is not killed.
- As all children of node 2 are explored it becomes a dead node. Node 4 is also killed. Now the live nodes are 3, 5. Out of them, node 5 with a minimum $\hat{c}(\cdot)$ value becomes a next E-node generating its children: node 6 (x_3=1) and node7(x_3 = 0). ***upper* = u(5) = –60.**

(4)

L:	~~1~~, ~~2~~, 3, 4, 5, 6, 7
E:	~~1~~, ~~2~~, 5
D:	1, 4, 2

- $\hat{c}(6)$ = Bound_Bk(3, –60, 9) = –60 – (1/3*12) = **–64**…(Items 1, 3 are already added and item 2 is not added. The fraction 1/3 of item 4 can be added to a knapsack to give a greedy solution.)

- $u(6)$ = Bound_B&B(3, –60,9) = –60 – 0 = **–60**...(Items 1, 3 are already added and item 2 is not added. Item 4 cannot be added as its addition exceeds a knapsack capacity M.)
- Since $\hat{c}(6) = upper$ (i.e.–60 = –60), it is not killed.
- $\hat{c}(7)$ = Bound_Bk(3, –40, 4) = –40–12 = **–52**...(Item 1is already added and items2, 3 are not added. Item 4 can be completely added to give a greedy solution.)
- $u(7)$ = Bound_B&B(3, –40,4) = –40 –12= **–52**...(Item 1is already added and items 2, 3 are not added. Item 4 can be completely added.)
- Since $\hat{c}(7) > upper$ (i.e. –52 > –60), it is killed.
- As all children of node 5 are explored it becomes a dead node. Now the live nodes are 3, 6. Out of them, node 6 with a minimum $\hat{c}(\cdot)$value becomes a next E-node generating its children: node 8 ($x_4 = 1$) and node 9($x_4 = 0$). *upper* = –60.

(5)

L:	~~1~~, ~~2~~, 3, ~~4~~, ~~5~~,6, ~~7~~,8,9
E:	~~1~~,~~2~~,~~5~~,6
D:	1,4, 2, 7, 5

- Node 8 with $x_4 = 1$ is killed as it is not feasible. The addition of item 4 exceeds a knapsack capacity M.
- $\hat{c}(9)$ = Bound_Bk(4, – 60, 9) = **–60**...(Items 1,3 are already added and items 2,4 are not added.)
- $u(9)$ = Bound_B&B(4, –60, 9) = **–60**...(Items 1, 3 are already added and items 2, 4 are not added.)
- Since $\hat{c}(9) = upper$ (i.e. –60 = –60), it is not killed.
- As all children of node 6 are explored it becomes a dead node. Node 8 is also killed. Now, the live nodes are 3, 9. Out of them, node 3 has $\hat{c}(\cdot) > upper$ (i.e. $\hat{c}(3)$ = –54 > –60), so it is killed. Node 9 cannot be further explored. So the search terminates at node 9. It becomes dead. *upper* = –60.

L:	~~1~~, ~~2~~, ~~3~~, ~~4~~, ~~5~~, ~~6~~, ~~7~~, ~~8~~, ~~9~~
E:	~~1~~, ~~2~~, ~~5~~, ~~6~~
D:	1,4, 2, 7, 5, 8, 6,3,9

- Thus an optimal solution is given by node 9 as it is the least cost answer state. **The final solution tuple is (1, 0, 1, 0) with a maximum profit of 60Rs.**

Ex. 6.5.3 : Solve the following instance of 0/1 knapsack problem by LC branch and bound approach: n = 4; M = 18; p(1:4) = {11, 11, 12, 18}; w(1:4) = {3, 4, 6, 9}. **(10 Marks)**

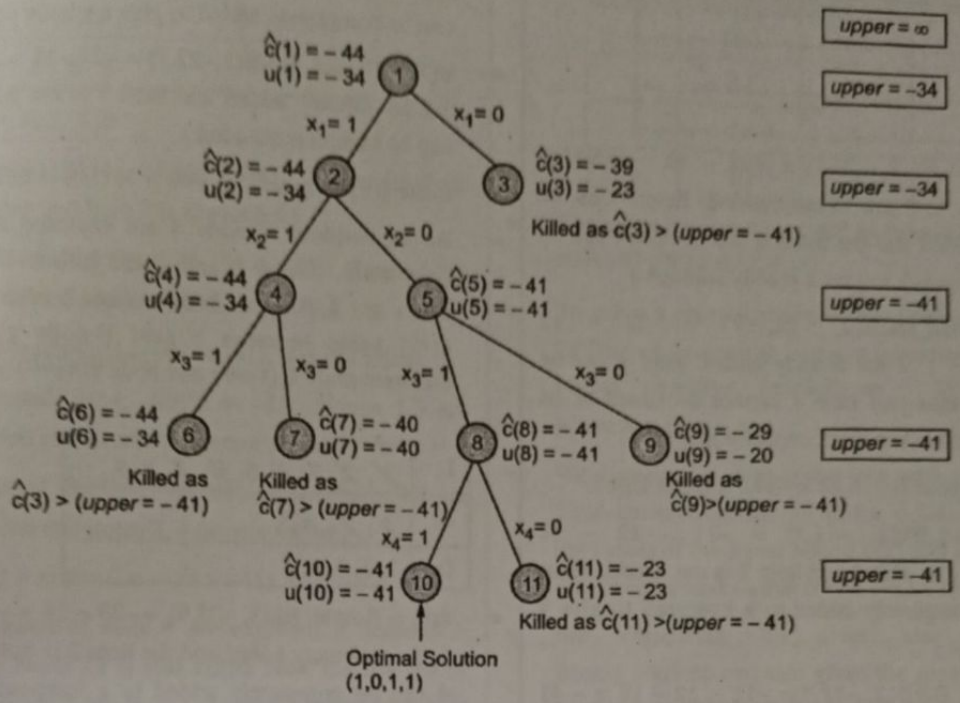
☑ **Soln. :**

- To solve a given instance of the knapsack problem by LCBB, we first check that whether the given items are arranged in decreasing order of the ratio of P_i/W_i, $1 \le i \le n$. Here n = 4 and M = 18.

Item	P_i	W_i	P_i/W_i
1	11	3	3.67
2	11	4	2.75
3	12	6	2
4	18	9	2

- The given items are arranged in decreasing order of the ratio of $P_i/W_i, 1 \le i \le 4$.
- To solve a given instance of the knapsack problem by LCBB, we convert it into a minimization problem by setting the objective function as: Minimize $-\sum p_i x_i$ where $x_i = 0$ or 1.
- We construct a state space tree with fixed size tuple formulation as shown in Fig. Ex. 6.5.3. At each node k, the values of the lower bound $\hat{c}(k)$ and the upper bound $u(k)$ is computed so that $\hat{c}(k) \le c(k) \le u(k)$.
- For computing $\hat{c}(k)$, we use an algorithm Bound_Bk(k,cp,cw) that gives the greedy solution to a knapsack problem where $0 \le x_i \le 1$. k, cp, cw have usual meanings.
- For computing u(k), we use an algorithm Bound_B&B(k,cp,cw) that gives the solution to a knapsack problem where $x_i = 0$ or 1.
- Initially, we set *upper* = ∞ and we kill the node k if $\hat{c}(k) > upper$.
- As per the LCBB, each time a live node with a minimum $\hat{c}(\cdot)$ is selected as a next E-node.
- Whenever a new answer state is obtained, the value of the *upper* is updated accordingly.
- Finally, the least-cost answer state gives an optimal solution to the problem.
- We maintain the lists of live nodes(L), E-nodes(E) and dead nodes(D).

The computations of ĉ(·) and u(·) are as follows :



(1F3)**Fig. Ex. 6.5.3 : The state space tree**

(1) Initially, node1 is created.

L:	1
E:	--
D:	--

- **ĉ(1)** = Bound_Bk(0, 0, 0) = 0 – 11 – 11 – 12 – (5/9*18) = **–44**…(Items 1, 2, 3 can be completely added and the fraction 5/9 of item 4 can be added to a knapsack to give a greedy solution.)
- **u(1)** = Bound_B&B(0, 0, 0) = 0 – 11 – 11 – 12 = **–34**…(Items 1,2,3 can be completely added and item 4 cannot be added as its addition exceeds a knapsack capacity M.)
- Since ĉ(1) < *upper* (i.e. – 38 < ∞), it is not killed.
- As node 1 is the only live node in the list, it is explored to generate its children: node2 (x_1 = 1) and node3 (x_1 = 0). Nodes 2 and 3 are added in the list of live nodes. ***upper*** **= u(1) = –34.**

(2)

L:	1, 2, 3
E:	1
D:	--

- **ĉ(2)** = Bound_Bk(1, –11,3) = –11 – 11 –12 – (5/9*18) = **–44**…(Item 1 is already added. Items 2,3can be completely added and the fraction 5/9 of item 4 can be added to a knapsack to give a greedy solution.)
- **u(2)** = Bound_B&B(1, – 11,3) = –1 – 11 – 12 = **–34**…(Item 1 is already added. Items 2,3 can be completely added and item 4 cannot be added as its addition exceeds a knapsack capacity M.)
- Since ĉ(2) < *upper* (i.e. – 44 < – 34), it is not killed.
- **ĉ(3)** = Bound_Bk(1,0,0)= 0 – 11 – 12 – (8/9*18) = **–39**…(Item 1 is not added. Items 2, 3 can be completely added and the fraction 8/9 of item 4 can be added to a knapsack to give a greedy solution.)
- **u(3)** = Bound_B&B(1, 0, 0) = 0 – 11 – 12 = **–23**…(Item 1 is not added. Items 2, 3 can be completely added and item 4 cannot be added as its addition exceeds a knapsack capacity M.)
- Since ĉ(3) < *upper* (i.e. –39 < –34), it is not killed.
- As all children of node1 are explored it becomes a dead node. So, the live nodes are 2 and 3. Since ĉ(2) < ĉ(3), node 2 becomes a next E-node generating its children: node 4 (x_2 = 1) and node 5(x_2 = 0). *upper* = –34.

(3)

L:	~~1~~, 2, 3, 4, 5
E:	~~1~~, 2
D:	1

- $\hat{c}(4)$ = Bound_Bk(2, –22,7) = –22 – 12 – (5/9*18) = **–4**...(Items 1, 2 are already added. Item 3 can be completely added and the fraction 5/9 of item 4 can be added to a knapsack to give a greedy solution.)
- **u(4)** = Bound_B&B(2, –22, 7) = –22 – 12 = **–34**...(Items 1, 2 are already added. Item 3 can be completely added and item 4 cannot be added as its addition exceeds a knapsack capacity M.)
- Since $\hat{c}(4) <$ *upper* (i.e. – 44 < –34), it is not killed.
- $\hat{c}(5)$ = Bound_Bk(2, –11,3) = –11 – 12 – 18 = **–41**...(Item 1 is added and item 2 is not added. Items 3, 4 can be completely added to a knapsack to give a greedy solution.)
- **u(5)** = Bound_B&B(2, –11,3)= –11 – 12 – 18 = **– 41** ...(Item 1 is added and item 2 is not added. Items 3, 4 can be completely added.)
- Since $\hat{c}(5) <$ *upper* (i.e. –41< –34), it is not killed.
- As all children of node 2 are explored it becomes a dead node. Now the live nodes are 3, 4, 5. Out of them, node 4 with a minimum $\hat{c}(\cdot)$ value becomes a next E-node generating its children: node 6 ($x_3 = 1$) and node7 ($x_3 = 0$). ***upper* = u(5) = –41.**

(4)

L:	~~1~~, ~~2~~, 3,4,5,6,7
E:	~~1~~, ~~2~~ ,4,
D:	1,2

- $\hat{c}(6)$ = Bound_Bk(3, –34, 13) = –34 – (5/9*18) = **–44**...(Items 1, 2, 3 are already added and the fraction 5/9 of item 4 can be added to a knapsack to give a greedy solution.)
- **u(6)**=Bound_B&B(3, –34,14) = –34 – 0 = **–34**...(Items 1, 2, 3 are already added and item 4 cannot be added as its addition exceeds a knapsack capacity M.)
- Since $\hat{c}(6) >$ *upper* (i.e. –34 > –41), it is killed.
- $\hat{c}(7)$ = Bound_Bk(3, –22,7) = –22 – 18)= **–40**...(Items 1,2 are already added and item 3 is not added. Item 4 can be completely added to give a greedy solution.)
- **u(7)** = Bound_B&B(3,–22,7) = –22 –18 = **–40**...(Items 1,2 are already added and item 3 is not added. Item 4 can be completely added.)
- Since $\hat{c}(7) >$ *upper* (i.e. –40 > –41), it is killed.
- As all children of node 4 are explored it becomes a dead node. Nodes 6 and 7 are killed. Now the live nodes are 3, 5. Out of them, node 5 with a minimum $\hat{c}(\cdot)$ value becomes a next E-node generating its children: node 8 (x_3=1) and node 9(x_3=0). *upper* = –41.

(5)

L:	~~1~~, ~~2~~, 3, ~~4~~, 5, ~~6~~, ~~7~~, 8, 9
E:	~~1~~, ~~2~~, ~~4~~,5
D:	1, 2, 4, 6, 7

- $\hat{c}(8)$ = Bound_Bk(3, –23,9) = –23 – 18 = **–41** ...(Items 1, 3 are already added and the item 2 is not added. Item 4 can be completely added to a knapsack to give a greedy solution.)
- **u(8)** = Bound_B&B(3,–23,9) = –23– 18 = **–41**...(Items 1, 3 are already added and the item 2 is not added. Item 4 can be completely added.)
- Since $\hat{c}(8) =$ *upper* (i.e. –41 = –41), it is not killed.
- $\hat{c}(9)$ = Bound_Bk(3, –11,3) = –11 – 18 = **–39**...(Item 1is already added and items 2, 3 are not added. Item 4 can be completely added to give a greedy solution.)
- **u(9)** = Bound_B&B(3, –11, 3) = –11 – 18 = **–39**...(Item 1 is already added and items 2, 3 are not added. Item 4 can be completely added.)
- Since $\hat{c}(9) >$ *upper* (i.e. –39 > –41), it is killed.
- As all children of node 5 are explored it becomes a dead node. Node 9 is killed. Now the live nodes are 3, 8. Out of them, node 8 with a minimum $\hat{c}(\cdot)$ value becomes a next E-node generating its children: node 10 ($x_4 = 1$) and node 11($x_4 = 0$). *upper* = –41.

(d)

L:	X, Z, 3, X, X, X, 7, 8, X, 10, 11
E:	X, Z, X, X, 8
D:	1, 2, 4, 6, 7, 5, 9

- ĉ(10) = Bound_Bk(4, –41, 18) = **–41**...(Items 1,3,4 are already added and item 2 is not added.)
- u(10) = Bound_B&B(4, –41, 18) = **–41**...(Items 1,3,4 are already added and item 2 is not added.)
- Since ĉ(10) = *upper* (i.e. –41 = –41), it is not killed.
- ĉ(11) = Bound_Bk(4, –23, 9) = **–23**...(Items 1,3 are already added and items 2,4 are not added.)
- u(11) = Bound_B&B(4, –23, 9) = **–23**...(Items 1,2 are already added and items 2, 4 are not added.)
- Since ĉ(11) > *upper* (i.e. – 23 > –41), it is killed.
- As all children of node 8 are explored it becomes a dead node. Node 11 is also killed. Now the live nodes are 3, 10. Node 3 has its ĉ(·) value > *upper* (i.e. ĉ(3) = –39 > –41), so node 3 is killed. Node 10 cannot be further explored. So the search terminates at node 10. It becomes dead. *upper* = –41.

L:	X, Z, X, X, X, X, 7, X, X, 10, 11
E:	X, Z, X, X, X,
D:	1, 2, 4, 6, 7, 5, 9, 8, 11, 3, 10

- Thus an optimal solution is given by node 10 as it is the least cost answer state. **The final solution tuple is (1,0,1,1) with a maximum profit of 41 units.**

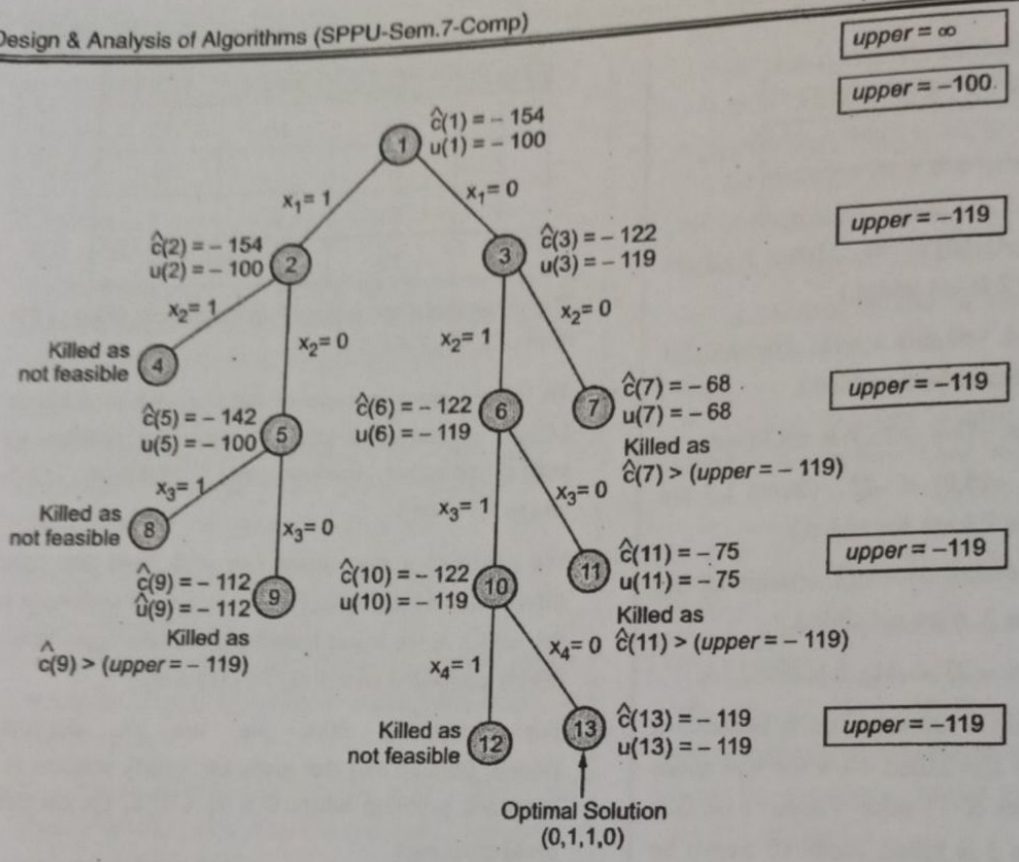
Ex. 6.5.4 : Solve the following instance of the knapsack problem by branch and bound algorithm : n = 4, W(1:4) = {10,7,8,4}, P(1:4) = {100, 63, 56,12}, knapsack capacity M = 16. **(8 Marks)**

Soln. :

- To solve a given instance of the knapsack problem by LCBB, we first check that whether the given items are arranged in decreasing order of the ratio of P_i/W_i, $1 \le i \le n$. Here n = 4 and M = 16.

Item	P_i	W_i	P_i/W_i
1	100	10	10
2	63	7	9
3	56	8	7
4	12	4	3

- The given items are arranged in decreasing order of the ratio of P_i/W_i, $1 \le i \le 4$.
- To solve a given instance of the knapsack problem by LCBB, we convert it into a minimization problem by setting objective function as : Minimize $-\sum p_i x_i$ where $x_i = 0$ or 1.
- We construct a state space tree with fixed size tuple formulation as shown in Fig. Ex. 6.5.4. At each node k, the values of the lower bound ĉ(k) and the upper bound u(k) is computed so that $\hat{c}(k) \le c(k) \le u(k)$.
- For computing ĉ(k), we use an algorithm Bound_Bk(k,cp,cw) that gives the greedy solution to a knapsack problem where $0 \le x_i \le 1$. k, cp, cw have usual meanings.
- For computing u(k), we use an algorithm Bound_B&B(k,cp,cw) that gives the solution to a knapsack problem where $x_i = 0$ or 1.
- Initially, we set *upper* = ∞ and we kill the node k if ĉ(k) > *upper*.
- As per the LCBB, each time a live node with a minimum ĉ(·) is selected as a next E-node.
- Whenever a new answer state is obtained, the value of the *upper* is updated accordingly.
- Finally, the least-cost answer state gives an optimal solution to the problem.
- We maintain the lists of live nodes(L), E-nodes(E) and dead nodes(D).
- The computations of ĉ(·) and u(·) are as follows :



(1F4)**Fig. Ex. 6.5.4 : The state space tree**

(1) Initially, node1 is created.

L:	1
E:	--
D:	--

- $\hat{c}(1)$ = Bound_Bk(0,0,0) = 0 – 100 – (6/7*63) = **–154** ...(Item 1 can be completely added and the fraction 6/7 of item 2 can be added to a knapsack to give a greedy solution n.)
- **u(1)** = Bound_B&B(0,0,0) = 0 – 100 = **–100** ...(Item 1can be completely added and item 2 cannot be added as its ad dition exceeds a knapsack capacity M.)
- Since $\hat{c}(1) < upper$ (i.e. –154 < ∞), it is not killed.
- As node 1 is the only live node in the list, it is explored to generate its children: node 2 ($x_1 = 1$) and node 3 ($x_1 = 0$). Nodes 2 and 3 are added in the list of live nodes. ***upper* = u(1) = –100.**

(2)

L:	1, 2, 3
E:	1
D:	--

- $\hat{c}(2)$ = Bound_Bk(1, –100,10) = –100– (6/7*63) = **–154** ...(Item 1 is already added. The fraction 6/7 of item 2 can be added to a knapsack to give a greedy solution.)
- **u(2)** = Bound_B&B(1, –100,10) = –100 – 0 = **–100**...(Item 1 is already added. Item 2 cannot be added as its addition exceeds a knapsack capacity M.)
- Since $\hat{c}(2) < upper$ (i.e. – 154 < –100), it is not killed.
- $\hat{c}(3)$ = Bound_Bk(1,0,0) = 0 – 63 – 56 – (1/4*12) = **–122** ...(Item 1 is not added. Items 2, 3 can be completely added and the fraction 1/4 of item 4 can be added to a knapsack to give a greedy solution.)
- **u(3)** = Bound_B&B(1,0,0) = 0 – 63 – 56 = **–119**...(Item 1 is not added. Items 2, 3 can be completely added and item 4 cannot be added as its addition exceeds a knapsack capacity M.)

- Since $\hat{c}(3) < upper$ (i.e. $-122 < -100$), it is not killed.
- As all children of node 1 are explored it becomes a dead node. So, the live nodes are 2 and 3. Since $\hat{c}(2) < \hat{c}(3)$, node 2 becomes a next E-node generating its children: node 4 ($x_2 = 1$) and node 5($x_2 = 0$). ***upper* = u(3) = –119.**

(3)

L:	~~1~~, 2, 3, 4, 5
E:	~~1~~, 2
D:	1

- Node 4 with $x_2 = 1$ is killed as it is not feasible. The addition of item 2 exceeds a knapsack capacity M.
- **$\hat{c}(5)$** = Bound_Bk(2, –100, 10) = –100 – (6/8*56) = **–142** ...(Item 1 is already added and item 2 is not added. The fraction 6/8 of item 3 can be added to a knapsack to give a greedy solution.)
- **u(5)** = Bound_B&B(2, –100, 10) = –100 – 0 = **–100**...(Item 1 is already added and item 2 is not added. Item 3 cannot be added as its addition exceeds a knapsack capacity M.)
- Since $\hat{c}(5) < upper$ (i.e.–142 < –119), it is not killed.
- As all children of node 2 are explored it becomes a dead node. Node 4 is also killed. Now the live nodes are 3, 5. Out of them, node 3 with a minimum $\hat{c}(\cdot)$ value becomes a next E-node generating its children: node 6 ($x_2 = 1$) and node7($x_2 = 0$). *upper* = –119.

(4)

L:	~~1~~, ~~2~~, 3, ~~4~~, 5, 6, 7
E:	~~1~~, ~~2~~, 3
D:	1, 4, 2

- **$\hat{c}(6)$** = Bound_Bk(2, –63, 7) = –63 – 56 – (1/4*12) = **–122**...(Item 1 is not added and item 2 is added. Item 3 can be completely added and the fraction 1/4 of item 4 can be added to a knapsack to give a greedy solution.)
- **u(6)** = Bound_B&B(2, –63,7) = – 63 – 56 = **–119**...(Item 1 is not added and item 2 is added. Item 3 can be completely added and item 4 cannot be added as its addition exceeds a knapsack capacity M.)
- Since $\hat{c}(6) = upper$ (i.e. –119 = –119), it is not killed.
- **$\hat{c}(7)$** = Bound_Bk(2, 0, 0) = 0 – 56 – 12 = **–68**...(Items 1, 2 are not added. Items 3, 4 can be completely added to give a greedy solution.)
- **u(7)** Bound_B&B(2, 0, 0) = 0 – 56 – 12 = **–68** ...(Items 1, 2 are not added. Items 3, 4 can be completely added.)
- Since $\hat{c}(7) > upper$ (i.e. –68 > –119), it is killed.
- As all children of node 3 are explored it becomes a dead node. Node 7 is also killed. So the live nodes are 5, 6. Out of them, nodes 5 with a minimum $\hat{c}(\cdot)$ value becomes a next E-node generating its children: node 8 ($x_3 = 1$) and node 9($x_3 = 0$). *upper* = –119.

(5)

L:	~~1~~, ~~2~~, ~~3~~, ~~4~~, 5, 6, ~~7~~, 8, 9
E:	~~1~~, ~~2~~, ~~3~~, 5
D:	1, 4, 2, 7, 3

- Node 8 with $x_3 = 1$ is killed as it is not feasible. The addition of item 4 exceeds a knapsack capacity M.
- **$\hat{c}(9)$** = Bound_Bk(3, –100, 10) = –100 – 12 = **–112**...(Item 1 is already added and items 2, 3 are not added. Item 4 can be completely added to give a greedy solution.)
- **u(9)** = Bound_B&B(3, –100, 10) = –100 – 12 = **–112** ...(Item 1 is already added and items 2, 3 are not added. Item 4 can be completely added.)
- Since $\hat{c}(9) > upper$ (i.e. –112 > –119), it is killed.
- As all children of node 5 are explored it becomes a dead node. Nodes 8 and 9 are also killed. Now, node 6 is the only live node, it becomes a next E-node generating its children: node 10 ($x_3 = 1$) and node 11($x_3 = 0$). *upper* = –119.

(6)

L:	~~1~~, ~~2~~, ~~3~~, ~~4~~, ~~5~~, 6, ~~7~~, ~~8~~, ~~9~~,,10,11
E:	~~1~~, ~~2~~, ~~3~~, ~~5~~, 6
D:	1,4, 2, 7, 3, 8, 9, 5

- **$\hat{c}(10)$** = Bound_Bk(3, –119, 15) = –119 – (1/4*12) = **–122**...(Item 1 is not added. Items 2, 3 are already added. The fraction 1/4 of item 4 can be added to give a greedy solution.)
- **u(10)** = Bound_B&B(3, –119, 15) = –119 – 0 = **–119**...(Item 1 is not added. Items 2, 3 are already added and item 4 is not added as its addition exceeds a knapsack capacity M.)
- Since $\hat{c}(10) < upper$ (i.e.– 122 < – 119), it is not killed.

- $\hat{c}(11)$ = Bound_Bk(3, –63,7) = –63 –12 = –75 ...(Items 1,3 are not added. Item 2 is already added. Item 4 can be completely added to give a greedy solution.)
- $u(11)$ = Bound_B&B(3, – 63, 7) = – 63 – 12 = – 75 ...(Items 1,3 are not added. Item 2 is already added. Item 4 can be completely added.)
- Since $\hat{c}(11) > upper$ (i.e. –75 > –119), it is killed.
- As all children of node 6 are explored it becomes a dead node. Node 11 is also killed. Now, node 10 is the only live node, it becomes a next E-node generating its children: node 12 ($x_4 = 1$) and node 13($x_4 = 0$). *upper* = –119.

(7)

L:	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,12,13
E:	1, 2, 3, 5, 6, 10
D:	1,4, 2, 7, 3, 8, 9, 5, 11,6

- Node 12 with $x_4 = 1$ is killed as it is not feasible. The addition of item 4 exceeds a knapsack capacity M.
- $\hat{c}(13)$ = Bound_Bk(4, –119, 15) = **–119** ...(Items 1, 4 are not added. Items 2,3 are already added.)
- $u(13)$ = Bound_B&B(4, –119, 15) = **–119** ...(Items 1, 4 are not added. Items 2, 3 are already added.)
- Since $\hat{c}(13) = upper$ (i.e. –119= –119), it is not killed.
- As all children of node 10 are explored it becomes a dead node. Node 12 is also killed. Node 13 cannot be further explored. So the search terminates at node 13. It becomes dead. *upper* = –119.

L:	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13
E:	1, 2, 3, 5, 6, 10
D:	1,4, 2, 7, 3, 8, 9, 5, 11,6,12,10,13

- Thus an optimal solution is given by node 13 as it is the least cost answer state. **The final solution tuple is (0,1,1,0) with a maximum profit of 119units.**

6.5.2 0/1 Knapsack Problem by FIFOBB

Here, a next E-node is selected by the first-in-first-out order of live nodes. It maintains a queue of live nodes.

Examples

Ex. 6.5.5 : Solve the following instance of 0/1 knapsack problem by FIFO branch and bound approach: n = 4; M = 15 and (p1,p2,p3,p4) = (10,10,12,18); (w1,w2,w3,w4) = (2,4,6,9). **(10 Marks)**

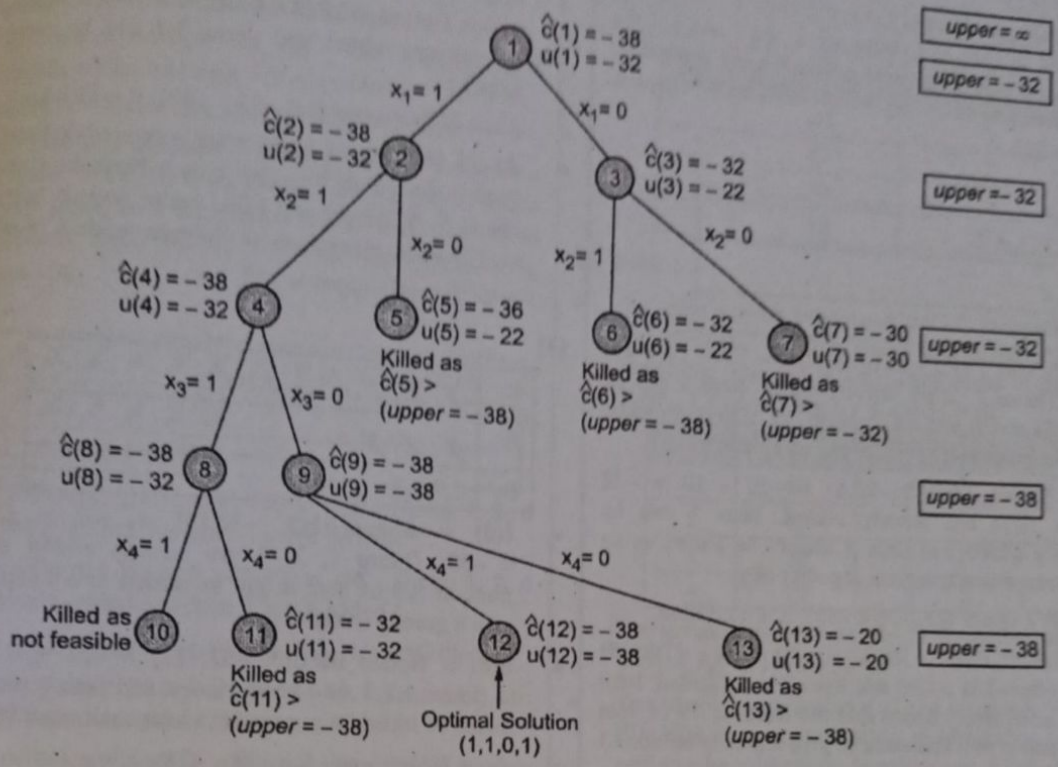
Soln. :

- To solve a given instance of the 0/1 knapsack problem by FIFOBB, we first check that whether the given items are arranged in decreasing order of the ratio of P_i/W_i, $1 \le i \le n$. Here n = 4 and M = 15.

Item	P_i	W_i	P_i/W_i
1	10	2	5
2	10	4	2.5
3	12	6	2
4	18	9	2

- The given items are arranged in decreasing order of the ratio of P_i/W_i, $1 \le i \le 4$.
- To solve a given instance of the knapsack problem by FIFOBB, we convert it into a minimization problem by setting objective function as: Minimize $-\sum p_i x_i$ where $x_i = 0$ or 1.
- We construct a state space tree with fixed size tuple formulation as shown in Fig. Ex. 6.5.5. At each node k, the values of the lower bound $\hat{c}(k)$ and the upper bound u(k) is computed so that $\hat{c}(k) \le c(k) \le u(k)$.
- For computing $\hat{c}(k)$, we use an algorithm Bound_Bk(k,cp,cw) that gives the greedy solution to a knapsack problem where $0 \le x_i \le 1$. k, cp, cw have usual meanings.
- For computing u(k), we use an algorithm Bound_B&B(k,cp,cw) that gives the solution to a knapsack problem where $x_i = 0$ or 1.
- Initially, we set *upper* = ∞ and we kill the node k if $\hat{c}(k) > upper$.
- As per the FIFOBB, each time a live node in the FIFO order is selected as a next E-node.
- Whenever a new answer state is obtained, the value of the *upper* is updated accordingly.
- Finally, the least-cost answer state gives an optimal solution to the problem.
- We maintain the lists of live nodes(L), E-nodes(E) and dead nodes(D).

The computations of ĉ(·) and u(·) are as follows:



(1F5)**Fig. Ex. 6.5.5 : The state space tree**

1) **Initially, node1 is created.**

L:	1
E:	--
D:	--

ĉ(1) = Bound_Bk(0, 0, 0) = 0 – 10 – 10 – 12 – (3/9*18) = **–38**...(Items 1, 2, 3 can be completely added and the fraction 3/9 of item 4 can be added to a knapsack to give a greedy solution.)

u(1) = Bound_B&B(0, 0, 0) = 0 – 10 – 10 – 12 = **–32** ...(Items 1, 2, 3 can be completely added and item 4 cannot be added as its addition exceeds a knapsack capacity M.)

Since ĉ(1) < *upper* (i.e. –38 < ∞), it is not killed.

As node 1 is the only live node in the list, it is explored to generate its children: node2 (x_1 = 1) and node3 (x_1 = 0). Nodes 2 and 3 are added in the list of live nodes. ***upper* = u(1) = –32.**

(2)

L:	1, 2, 3
E:	1
D:	

- **ĉ(2)** = Bound_Bk(1, –10,2) = –10 – 10 – 12 – (3/9*18) = **–38**...(Items 1 is already added. Items 2,3can be completely added and the fraction 3/9 of item 4 can be added to a knapsack to give a greedy solution.)
- **u(2)** = Bound_B&B(1, –10,2)= –10 – 10 – 12 = **–32** ...(Items 1 is already added. Items 2, 3 can be completely added and item 4 cannot be added as its addition exceeds a knapsack capacity M.)
- Since ĉ(2) < *upper* (i.e. –38 < –32), it is not killed.
- **ĉ(3)** = Bound_Bk(1,0,0) = 0 – 10 – 12 – (5/9*18) = **–32** ...(Item 1 is not added. Items 2, 3 can be completely added and the fraction 5/9 of item 4 can be added to a knapsack to give a greedy solution.)
- **u(3)** = Bound_B&B(1, 0, 0) = 0 – 10 – 12 = **–22** ...(Item 1 is not added. Items 2, 3 can be completely added and item 4 cannot be added as its addition exceeds a knapsack capacity M.)

- Since ĉ(3) = *upper* (i.e. –32 = –32), it is not killed.
- As all children of node 1 are explored it becomes a dead node. So, the live nodes are 2 and 3. As per the FIFO order, node 2 becomes a next E-node generating its children: node 4 (x_2 = 1) and node 5(x_2=0). *upper* = –32.

(3)

L:	~~1~~, 2, 3, 4 ,5
E:	~~1~~, 2
D:	1

- **ĉ(4)** = Bound_Bk(2, –20,6) = –20 – 12 – (3/9*18) = **–38**...(Items 1, 2 are already added. Item 3 can be completely added and the fraction 3/9 of item 4 can be added to a knapsack to give a greedy solution.)
- **u(4)** = Bound_B&B(2, –20,6) = –20 – 12 = **–32** ...(Items 1, 2 are already added. Item 3 can be completely added and item 4 cannot be added as its addition exceeds a knapsack capacity M.)
- Since ĉ(4) < *upper* (i.e. –38 < –32), it is not killed.
- **ĉ(5)** = Bound_Bk(2, –10, 2) = –10 – 12 – (7/9*18) = **–36** ...(Item 1 is added and item 2 is not added. Item 3 can be completely added and the fraction 7/9 of item 4 can be added to a knapsack to give a greedy solution.)
- **u(5)** = Bound_B&B(2, –10, 2) = –10 – 12 = **–22** ...(Item 1 is added and item 2 is not added. Item 3 can be completely added and item 4 cannot be added as its addition exceeds a knapsack capacity M.)
- Since ĉ(5) < *upper* (i.e. –36 < –32), it is not killed.
- As all children of node 2 are explored it becomes a dead node. Now the live nodes are 3, 4, 5. As per the FIFO order, node 3 becomes a next E-node generating its children: node 6 (x_2 = 1) and node7(x_2 = 0). *upper* = –32.

(4)

L:	~~1~~, ~~2~~, 3,4,5,6,7
E:	~~1~~, ~~2~~,3
D:	1,2

- **ĉ(6)** = Bound_Bk(2, –10, 4) = –10 – 12 – (5/9*18) = **–32**...(Item 1 is not added and item 2 is already added. The fraction 5/9 of item 4 can be added to a knapsack to give a greedy solution.)
- **u(6)** = Bound_B&B(2, –10, 4) = –10 – 12 = **–22** ...(Item 1 is not added and item 2 is already added.)
- Since ĉ(6) = *upper* (i.e. –32 = –32), it is not killed.
- **ĉ(7)** = Bound_Bk(2, 0, 0) = –12 – 18 = **–30**...(Items 1, 2 are not added and items 3, 4 can be completely added to give a greedy solution.)
- **u(7)** = Bound_B&B(2,0,0) = –20 – 18 = **–30**...(Items 1,2 are not added and items 3,4 can be completely added.)
- Since ĉ(7) > *upper* (i.e. –30 > –32), it is killed.
- As all children of node 3 are explored it becomes a dead node. Node 7 is also killed. Now the live nodes are 4, 5, 6. As per the FIFO order node 4 becomes a next E-node generating its children: node 8 (x_3=1) and node 9(x_3=0). *upper* = –32.

(5)

L:	~~1~~, ~~2~~, ~~3~~, 4,5,6, ~~7~~,8,9
E:	~~1~~, ~~2~~, ~~3~~, 4
D:	1,2,7,3

- **ĉ(8)** = Bound_Bk(3, –32,12) = –32 – (3/9*18) = **–38**...(Items 1, 2, 3 are already added and the fraction 3/9 of item 4 can be added to a knapsack to give a greedy solution.)
- **u(8)** = Bound_B&B(3, –32, 12) = –32 – 0 = –32 ...(Items 1,2,3 are already added and item 4 cannot be added as its addition exceeds a knapsack capacity M.)
- Since ĉ(6) < *upper* (i.e.–38 < –32), it is not killed.
- **ĉ(9)** = Bound_Bk(3, –20,6) = –20 – 18 = **–38** ...(Items 1,2 are already added and item 3 is not added. Item 4 can be completely added to give a greedy solution.)
- **u(9)** = Bound_B&B(3, –20, 6) = –20 – 18 = **–38**...(Items 1, 2 are already added and item 3 is not added. Item 4 can be completely added.)
- Since ĉ(7) < *upper* (i.e. –38 < –32), it is not killed. ***upper* = u(9) = –38.**
- As all children of node 4 are explored it becomes a dead node. Now the live nodes are 5, 6, 8, 9. As per the FIFO order node 5 becomes a next E-node. Since ĉ(5) > *upper* (i.e. –36> –38), it is killed. Then as per the FIFO order node 6 becomes a next E-node. Since ĉ(6) > *upper* (i.e. –32 > –38), it is also killed. Now, as per the FIFO order node 8 becomes a next E-node generating its children: node 10 (x_4 = 1) and node 11 (x_4 = 0). *upper* = –38.

(6)

L:	~~1~~, ~~2~~, ~~3~~, ~~4~~, ~~5~~, ~~6~~, ~~7~~, 8,9,10,11
E:	~~1~~, ~~2~~, ~~3~~, ~~4~~, 8
D:	1,2,7,3,4,5,6

- Node 10 with $x_4 = 1$ is killed as it is not feasible. The addition of item 4 exceeds a knapsack capacity M.
- $\hat{c}(11)$ = Bound_Bk(4, –32, 12) = **–32**...(Items 1, 2, 3 are already added and item 4 is not added.)
- **u(11)** = Bound_B&B(4, –32, 12) = **–32**...(Items 1, 2, 3 are already added and item 4 is not added.)
- Since $\hat{c}(11) > upper$ (i.e. –32 > –38), it is killed.
- As all children of node 8 are explored it becomes a dead node. Node 10 is also killed. Now, node 9 is the only live node, so it becomes a next E-node generating its children: node 10 ($x_4 = 1$) and node 11($x_4 = 0$). *upper* = –38.

(7)

L:	~~1~~, ~~2~~, ~~3~~, ~~4~~, ~~5~~, ~~6~~, ~~7~~, ~~8~~, 9, 10, ~~11~~,12,13
E:	~~1~~, ~~2~~, ~~3~~, ~~4~~, ~~8~~, 9
D:	1,2,7,3,4,5,6,10,11,8

- $\hat{c}$**(12)** = Bound_Bk(4, –38, 15) = **–38**...(Items 1, 2, 4 are already added and item 3 is not added.)
- **u(12)** = Bound_B&B(4, –38,15) = **–38**...(Items 1, 2, 4 are already added and item 3 is not added.)
- Since $\hat{c}(12) = upper$ (i.e. –38 = –38), it is not killed.
- $\hat{c}$**(13)** = Bound_Bk(4, –20,6) = **–20**...(Items 1,2 are already added and items 3,4 are not added.)
- **u(13)** = Bound_B&B(4, –20,6) = **–20**...(Items 1,2 are already added but Item 3 is not added.)
- Since $\hat{c}(13) > upper$ (i.e. –20 > –38), it is killed.
- As all children of node 9 are explored it becomes a dead node. Node 13 is also killed. Node 12 cannot be further explored. So the search terminates at node 12. It becomes dead. *upper* = –38.

L:	~~1~~, ~~2~~, ~~3~~, ~~4~~, ~~5~~, ~~6~~, ~~7~~, ~~8~~, ~~9~~, 10, ~~11~~,~~12~~,~~13~~
E:	~~1~~, ~~2~~, ~~3~~, ~~4~~, ~~8~~, ~~9~~
D:	1,2,7,3,4,5,6,10,11, 8,13,9,12

- Thus an optimal solution is given by node 12 as it is the least cost answer state. **The final solution tuple is (1,1,0,1) with a maximum profit of 38 units.**

Ex 6.5.6 : Consider 0/1 knapsack instance: n = 4; M = 15 and (p1, p2, p3 ,p4) = (10, 10, 12, 18); (w1, w2, w3, w4) = (2,4,6,9). Find maximum profit using FIFOBB and LCBB. Use fixed size formation of state space. **(18 Marks)**

☑ **Soln. :**

Refer to the solutions to UEx. 6.5.1 under section 6.5.1 (0/1 Knapsack problem by LCBB) and UEx. 6.5.5 under section 6.5.2 (0/1 Knapsack problem by FIFOBB).

Ex. 6.5.7 : Consider 0/1 knapsack instance n = 4 with capacity 10kg such that (Refer to Table). Find maximum profit using first in first out branch and bound (FIFOBB) method. Use fixed size formation of state space.

Item	Profit (in Rs)	Weight (in kg)
1	40	4
2	42	7
3	20	5
4	12	3

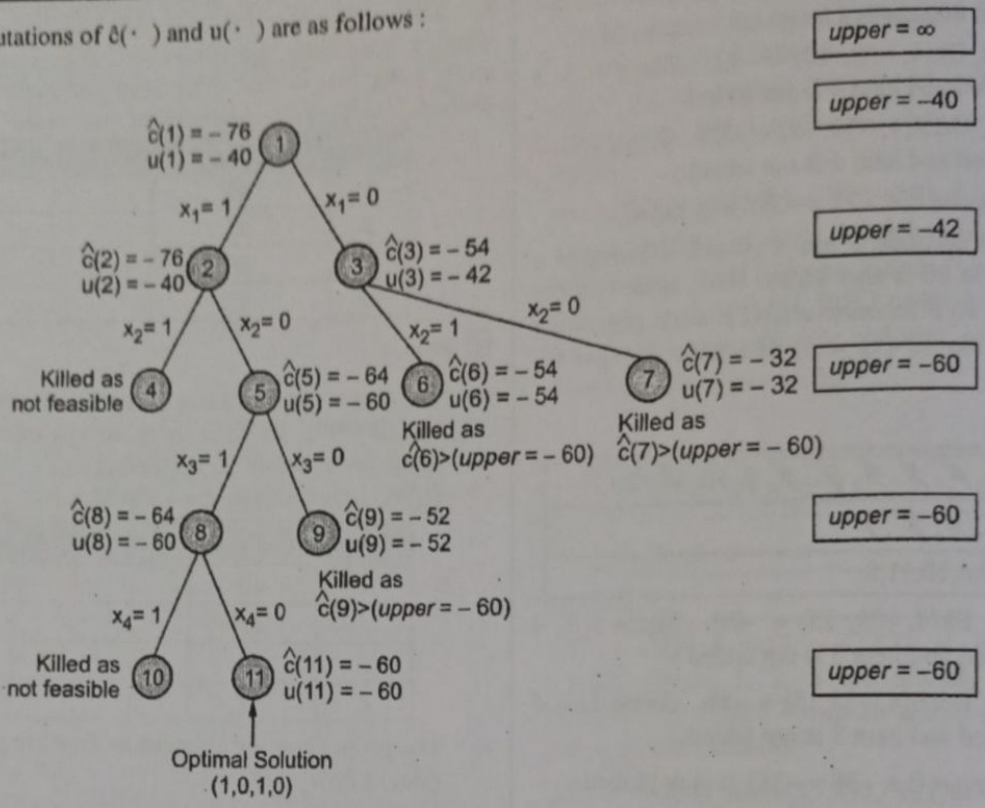
☑ **Soln. :**

- To solve a given instance of the 0/1 knapsack problem by FIFOBB, we first check that whether the given items are arranged in decreasing order of the ratio of P_i/W_i, $1 \le i \le n$. Here n = 4 and M = 10.

Item	P_i	W_i	P_i/W_i
1	40	4	10
2	42	7	6
3	20	5	4
4	12	3	4

- The given items are arranged in decreasing order of the ratio of P_i/W_i, $1 \le i \le 4$.
- To solve a given instance of the knapsack problem by FIFOBB, we convert it into a minimization problem by setting the objective function as: Minimize $-\sum p_i x_i$ where $x_i = 0$ or 1.
- We construct a state space tree with fixed size tuple formulation as shown in Fig. Ex. 6.5.7. At each node k, the values of the lower bound $\hat{c}(k)$ and the upper bound u(k) is computed so that $\hat{c}(k) \le c(k) \le u(k)$.
- For computing $\hat{c}(k)$, we use an algorithm Bound_Bk(k,cp,cw) that gives the greedy solution to a knapsack problem where $0 \le x_i \le$ 1. k, cp, cw have usual meanings.
- For computing u(k), we use an algorithm Bound_B&B(k,cp,cw) that gives the solution to a knapsack problem where $x_i = 0$ or 1.
- Initially, we set *upper* = ∞ and we kill the node k if $\hat{c}(k) > upper$.
- As per FIFOBB, each time a live node in the FIFO order is selected as a next E-node.
- Whenever a new answer state is obtained, the value of the *upper* is updated accordingly.
- Finally, the least-cost answer state gives an optimal solution to the problem.
- We maintain the lists of live nodes(L), E-nodes(E) and dead nodes(D).

- The computations of ĉ(·) and u(·) are as follows :



(1F6)**Fig. Ex. 6.5.7 : The state space tree**

(1) **Initially, node1 is created.**

L:	1
E:	--
D:	--

- **ĉ(1)** = Bound_Bk(0, 0, 0) = 0 – 40 – (6/7*42) = **–76** ...(Item 1can be completely added and the fraction 6/7 of item 2 can be added to a knapsack to give a greedy solution.)
- **u(1)** = Bound_B&B(0, 0, 0) = 0 – 40 = **–40** ...(Item 1 can be completely added and item 2 cannot be added as its addition exceeds a knapsack capacity M.)
- Since ĉ(1) < *upper* (i.e. –76 < ∞), it is not killed.
- As node 1 is the only live node in the list, it is explored to generate its children: node2 (x_1 = 1) and node 3 (x_1 = 0). Nodes 2 and 3 are added in the list of live nodes. ***upper*** **= u(1) = –40.**

(2)

L:	1, 2, 3
E:	1
D:	--

- **ĉ(2)** = Bound_Bk(1, –40, 4) = –40 – (6/7*42) = **–76** ...(Item 1is already added and the fraction 6/7of item 2 can be added to a knapsack to give a greedy solution.)
- **u(2)** = Bound_B&B(1, – 40, 4) = –40 + 0 = **–40**...(Items 1 is already added. Item 2 cannot be added as its addition exceeds a knapsack capacity M.)
- Since ĉ(2) < *upper* (i.e. –76 < –40), it is not killed.
- **ĉ(3)** = Bound_Bk(1, 0, 0) = 0 – 42 – (3/5*20) = **–54** ...(Item 1 is not added. Item 2 can be completely added and the fraction 3/5 of item 3 can be added to a knapsack to give a greedy solution.)
- **u(3)** = Bound_B&B(1, 0, 0) = 0 – 42 = **–42** ...(Item 1 is not added. Item 2 cannot be added as its addition exceeds a knapsack capacity M.)
- Since ĉ(3) < *upper* (i.e. –54 < –40), it is not killed.
- As all children of node 1 are explored it becomes a dead node. So, the live nodes are 2 and 3. As per the FIFO order, node 2 becomes a next E-node generating its children: node 4 (x_2 = 1) and node 5(x_2 = 0). ***upper*** **= u(3) = –42.**

(3)

L:	~~1~~, 2, 3, 4, 5
E:	~~1~~, 2
D:	1

Node 4 with $x_2 = 1$ is killed as it is not feasible. The addition of item 2 exceeds a knapsack capacity M.

ĉ(5) = Bound_Bk(2, –40, 4) = –40 – 20 – (1/3*12) = **–64** ...(Item 1 is already added and item 2 is not added. Item 3 can be completely added and the fraction 1/3 of item 4 can be added to a knapsack to give a greedy solution.)

u(5) = Bound_B&B(2, –40, 4) = –40 – 20 = **–60**...(Item 1 is already added and item 2 is not added. Item 3 can be completely added and item 4 cannot be added as its addition exceeds a knapsack capacity M.)

Since ĉ(4) < *upper* (i.e. –64 < –42), it is not killed.

As all children of node 2 are explored it becomes a dead node. Node 4 is also killed. Now the live nodes are 3, 5. As per the FIFO order, node 3 becomes a next E-node generating its children node 6 ($x_2 = 1$) and node 7 ($x_2 = 0$). ***upper* = u(5) = –60.**

(4)

L:	~~1~~, ~~2~~, 3, ~~4~~, 5, 6, 7
E:	~~1~~, ~~2~~, 3
D:	1,4,2

ĉ(6) = Bound_Bk(2, –42, 7) = –42 – (3/5*20) = **–54**...(Item 1 is not added and item 2 is already added. The fraction 3/5 of item 3 can be added to a knapsack to give a greedy solution.)

u(6) = Bound_B&B(2, –42, 7) = –42 – 12 = **–54**...(Item 1 is not added and item 2 is already added. Item 3 cannot be added, but item 4 can be added.)

Since ĉ(6) > *upper* (i.e.54 > –60), it is killed.

ĉ(7) = Bound_Bk(2, 0, 0) = 0 – 20 – 12 = **–32**...(Items 1, 2 are not added. Items 3, 4 can be completely added to a knapsack to give a greedy solution.)

u(7) = Bound_B&B(1, 0, 0) = 0 – 20 –12 = **–32**...(Items 1, 2 are not added. Items 3, 4 can be completely added.)

Since ĉ(7) > *upper* (i.e. –32 > –60), it is killed.

As all children of node 3 are explored it becomes a dead node. Nodes 6, 7 are also killed. As node 5 is the only live node it becomes a next E-node generating its children: node 8 ($x_3 = 1$) and node 9($x_3 = 0$). *upper* = – 60.

(5)

L:	~~1~~, ~~2~~, ~~3~~, 4, 5, ~~6~~, ~~7~~, 8, 9
E:	~~1~~, ~~2~~, ~~3~~, 5
D:	1, 4, 2, 6, 7, 3

- **ĉ(8)** = Bound_Bk(3, –60, 9) = –60 – (1/3*12) = **–64** ...(Items 1, 3 are already added and item 2 is not added. The fraction 1/3 of item 4 can be added to a knapsack to give a greedy solution.)
- **u(8)** = Bound_B&B (3, –60, 9) = – 60 – 0 = **–60**...(Items 1, 3 are already added and item 2 is not added. Item 4 cannot be added as its addition exceeds a knapsack capacity M.)
- Since ĉ(6) = *upper* (i.e. –60 = –60), it is not killed.
- **ĉ(9)** = Bound_Bk(3, –40, 4) = –40 – 12 = **–52** ...(Item 1is already added and items 2, 3 are not added. Item 4 can be completely added to give a greedy solution.)
- **u(9)** = Bound_B&B(3, –40, 4) = –40 – 12 = **–52**...(Item 1is already added and items 2, 3 are not added. Item 4 can be completely added.)
- Since ĉ(7) > *upper* (i.e. –52 > –60), it is killed.
- As all children of node 5 are explored it becomes a dead node. Node 9 is also killed. As node 8 is the only live node it becomes a next E-node generating its children: node 10 ($x_4 = 1$) and node 11 ($x_4 = 0$). *upper* = –60.

(6)

L:	~~1~~, ~~2~~, ~~3~~, ~~4~~, ~~5~~, ~~6~~, ~~7~~, 8, ~~9~~, 10, 11
E:	~~1~~, ~~2~~, ~~3~~, ~~5~~, 8
D:	1,4,2,6,7,3,9,5

- Node 10 with $x_4 = 1$ is killed as it is not feasible. The addition of item 4 exceeds a knapsack capacity M.
- **ĉ(11)** = Bound_Bk(4, –60, 9) = **–60**...(Items 1, 3 are already added and items 2, 4 are not added.)
- **u(11)** = Bound_B&B(4, –60, 9) = **–60**...(Items 1, 3 are already added and items 2, 4 are not added.)
- Since ĉ(11) = *upper* (i.e. –60 = –60), it is not killed.
- As all children of node 8 are explored it becomes a dead node. Node 10 is also killed. Node 11 cannot be further explored. So the search terminates at node 11. It becomes dead. *upper* = –60.

L:	X, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
E:	X, 2, 3, 5, 8
D:	1,4,2,6,7,3,9,5,10,8,11

- Thus an optimal solution is given by node 11 as it is the least cost answer state. **The final solution tuple is (1,0,1,0) with a maximum profit of 60Rs.**

Summary

- **The Branch-and-bound technique** is a state-space algorithm where an E-node remains an E-node until it is dead.
- **A feasible solution** is defined by the problem states that satisfy all the given constraints.
- **An optimal solution** is a feasible solution, which produces the best value of a given objective function.
- **Bounding function** is a heuristic function that evaluates the lower and upper bounds on the possible solutions at each node.
- If a node does not produce a solution better than the best solution obtained thus far, then it is abandoned without its further exploration. The algorithm then branches to another path to get a better solution. The desired solution to the problem is the value of the best solution produced so far.
- Different search techniques used in the B&B algorithms are listed as below:
  (1) **LC search** (Least Cost Search): In this search, the node with the least value of a cost function is selected as a next E-node.
  (2) **BFS** (Breadth First Search): In this search, the live nodes are searched in the FIFO order to make them the next E-nodes.
  (3) **DFS** (Depth First Search): In this search, the live nodes are searched in the LIFO order to make them the next E-nodes.
- Based on different search techniques the B&B algorithms have different variations as:
  (1) **LCBB** (Least Cost Branch and Bound) that uses the least cost search.
  (2) **FIFOBB** (First In First Out Branch and Bound) that uses BFS (breadth first search).
  (3) **LIFOBB** (Last In First Out Branch and Bound) that uses DFS (depth first search).
- **The traveling salesperson problem** asks to minimize the cost of a tour that visits all cities only once and ends at the starting city. Since it is a minimization problem, it is directly solved by the B&B algorithm without changing the sign of an objective function.
- **The 0/1 knapsack problem** is a maximization problem that asks to maximize the profit by adding the given items to a knapsack considering its capacity. To solve this maximization problem by the B&B algorithms the sign of an objective function is changed to describe this "maximization of profit problem" as the "minimization of loss problem".

Chapter Ends...

❑❑❑

UNIT V

CHAPTER 7

Amortized Analysis

Syllabus

Amortized Analysis: Aggregate analysis, Accounting method, Potential function method. Amortized analysis-binary counter, stack. Time-Space trade-off, Introduction to Tractable and Non-tractable Problems, Introduction to Randomized and Approximate algorithms, Embedded Algorithms : Embedded system scheduling (power-optimized scheduling algorithm), sorting algorithm for embedded systems.

7.1 Amortized Analysis 7-2

7.1.1 Significance of Amortized Analysis

7.1.1(A) Aggregate Method 7-3

UQ. Explain amortized analysis. Find the amortized cost with respect to stack operations. **SPPU - Q. 6(b), May 18, 8 Marks** 7-3

GQ. Applying three methods of amortized analysis find the amortized cost of incrementing a binary counter. **(9 Marks)** 7-3

7.1.2 Accounting Method 7-5

UQ. What is amortized analysis? Explain aggregate and accounting techniques with example. **SPPU - Q. 6(a), May 19, 8 Marks** 7-5

7.1.3 Potential Function Method 7-6

GQ. Discuss the potential method for amortized analysis. Discuss it with operations on the stack. **(6 Marks)** 7-6

7.2 TIME-SPACE TRADE-OFF 7-7

7.3 Introduction to Tractable and Non-tractable Problems 7-8

UQ. Explain Tractable and non-tractable problems with example. **SPPU - Q. 5(b), May 18, 8 Marks** 7-8

7.4 Randomized algorithms 7-9

UQ. Write a short note on Randomized algorithm. **SPPU - Q. 4(a), Dec. 15, 8 Marks** 7-9

UQ. Which are different approaches to writing a Randomized Algorithm? Write a Randomized sort Algorithm. **SPPU - Q. 3(a), Dec. 16, 8 Marks** 7-9

7.4.1 Randomized Quicksort 7-10

7.5 Approximation algorithms 7-11

UQ. Explain the concept of Randomized algorithm and Approximation algorithm in brief with example. **SPPU - Q. 6(b), Dec. 19, Q. 4(a), Dec. 17, Q. 5(a), May 19, 8 Marks** 7-11

7.5.1 Travelling Salesman Approximation 7-12

7.6 Embedded Algorithms 7-13

GQ. Explain embedded system. **(4 Marks)** 7-13

7.6.1 Embedded System Scheduling (Power-Optimized Scheduling Algorithm) 7-14

UQ. Explain embedded system? Explain the scheduling algorithm for embedded system in detail. **SPPU - Q. 5(b), May 19, 8 Marks, Q. 7(b), Dec. 15, 9 Marks** 7-14

7.6.2 A Sorting Algorithm for the Embedded Systems 7-15

UQ. Explain in detail a sorting algorithm for the embedded systems. **SPPU - Q. 7(b), May 17, 9 Marks, Q. 5(b), Dec. 19, 8 Marks** 7-15

UQ. What is an Embedded System? Explain embedded sorting algorithm. **SPPU - Q. 6(a), May 18, 8 Marks** 7-15

GQ. Explain the essential characteristics of the embedded sorting algorithm. **(4 Marks)** 7-15

➤ **Chapter Ends** 7-16

7.1 AMORTIZED ANALYSIS

Consider an analogy of performing some banking operations at the ATM centre. For the same first list out certain relevant tasks and maximum estimated time needed to complete it.

Sr. No.	Task	The maximum time for completion
1.	An enquiry in the bank for the procedure of opening an account and availing debit card facility.	1 day
2.	Gathering all necessary documents to open the bank account. (E.g. Birth certificate, residential proof, identity proof, photo etc.)	15 days
3.	Submission of duly filled application form with necessary documents to the bank authority for opening an account.	1 day
4.	Approval from the bank and activation of a new bank account.	2 days
5.	Reception and activation of debit card linked to your bank account.	2 days
6.	Money withdrawal at the ATM centre by a debit card.	2 minutes
7.	Printing mini-statement at the ATM centre through a debit card.	2 minutes
8.	Deposit a cheque in the dropbox at the ATM centre.	2 minutes
9.	Reaching to a nearby ATM centre.	10 minutes

Now suppose you want to perform the following 2 tasks at the ATM centre :

(i) Depositing 10 cheques in a dropbox

(ii) 3 transaction of money withdrawal by a debit card.

The algorithmic steps and estimated maximum time for the above two tasks can be given as below :

Sr. No.	Algorithmic step	Maximum time needed
(i)	Reach to the nearby ATM	10 minutes
(ii)	Deposit 10 cheques in a dropbox	20 minutes (10 cheques × 2 min per cheque)
(iii)	3 transactions of money withdrawal by a debit card. [For any debit card transaction, you must have a debit card, so you must have completed tasks 1 to 6 in the above list. Total maximum time to complete tasks 1 to 6 once = 21 days 2 minutes]	(3 transactions × 21 days 2 minutes)
	The total maximum time required to complete the above algorithm steps	63 days 36 minutes

- Practically, only for the first-ever transaction by a debit card, you need to complete tasks 1 to 6 that need a maximum of 21 days 2 minutes.
- For later transactions by a debit card, you need just 2 minutes per transaction. Hence, a total of 21 days 6 minutes is needed to perform 3 transactions of money withdrawal by a debit card.
- Thus, to complete all 14 operations (1 for reaching to ATM + 10 for depositing 10 cheques in a dropbox + 3 money withdrawal transactions through debit card) you need maximum 21 days 36 minutes in practice.

  $\therefore$ Average time per operation = $\frac{\text{21 days 36 minutes}}{\text{14 operations}}$
- This average time per operation is much lesser than the time needed by a single debit card transaction in the worst case i.e., 21 days 2 minutes. Such type of analysis is known as "**amortized analysis**".

7.1.1 Significance of Amortized Analysis

- **Amortized analysis** is useful to analyse the algorithms where an occasional activity needs more time but most of the remaining activities need very less time.
- The theoretical worst-case analysis is less suitable for the analysis of algorithms or data structures that have a particular expensive task and the majority of other less expensive tasks.
- In the amortized analysis, we analyse the sequence of data structure operations and guarantee the average performance in the worst case which is lower than the worst-case performance of a specific costlier data structure operation.
- It is achieved by averaging the total cost over a sequence of steps, even though a particular step within the sequence might be costlier.
- The amortized analysis gives an insight into a specific data structure which is helpful for design optimization.

Techniques of amortized analysis

(1) Aggregate analysis (2) Accounting method
(3) Potential method

7.1.1(A) Aggregate Method

UQ. Explain amortized analysis. Find the amortized cost with respect to stack operations.
SPPU - Q. 6(b), May 18, 8 Marks

GQ. Applying three methods of amortized analysis find the amortized cost of incrementing a binary counter.
(9 Marks)

- The aggregate analysis demonstrates that a sequence of n operations needs the worst-case time T(n) in total, $\forall\ n \geq 0$.
- Then the amortized cost (the average cost in the worst case) per operation can be given as T(n)/n. This amortized cost is assigned to each operation in a sequence irrespective of its type.

Example 1 : Stack operation

- Consider a stack data structure. It performs the following two basic operations each of which needs time O(1).
  (1) PUSH (X, p): This fundamental operation pushes an object 'p' onto the stack X.
  (2) POP (X): This fundamental operation pops the object at the top of the stack X and returns it.

  ∴ Total cost of a sequence that contains n PUSH and POP operations = Θ(n).
- Let us consider an augmented stack operation MULTI-POP (X, m) which pops the topmost m objects of the stack X. If the stack-size < m, then it pops all the objects of the stack X. This new operation MULTI-POP (X, m) can be defined as:

```
MULTI-POP (X, m)
{
while (not STACK_EMPTY (X) && m > 0 )
  {
  POP (X) ;
  m := m – 1 ;
  } //end while
}
```

- Here, Boolean function STACK_EMPTY (X) returns TRUE if the stack X is empty and FALSE, otherwise. The total cost of one MULTI_POP (X, m) = minimum (stack-size, m).
- Let us consider an empty stack X and n PUSH, POP and MULTI-POP operations are executed on that stack. Since there are n PUSH operations executed on the stack X, the stack-size will be at most n. So, the worst-case cost of a single MULTI-POP operation will be O(n).

Thus we have,

Operation	Worst-case cost per operation
PUSH (X, p)	O(1)
POP (X)	O(1)
MULTI-POP (X, m)	O(n)

- As a worst-case time of any stack operation is bounded by O(n), the sequence of n stack operations results in a worst-case cost of $O(n^2)$.
- Aggregate analysis can give better upper bound than $O(n^2)$ by analysing a complete sequence n stack operations. Practically, we can pop at most n objects of the stack if total n objects are pushed onto the initially empty stack.
- Hence, the number of function calls to POP including calls within MULTI-POP, on a non-empty stack is at most equal to the number PUSH operations performed on the initially empty stack.
- So, the total cost of any sequence of n PUSH, POP and MULTI-POP operations on the stack will be O(n) for $\forall$ n.
- Thus, we get the average cost of each stack operation in a sequence of n PUSH, POP and MULTI-POP operations = $\frac{O(n)}{n}$ = O(1). This is considered as the "amortized cost" of each stack operation.
- Thus, the aggregate analysis demonstrates the worst-case bound of O(n) on a sequence of n stack operations by assigning an average cost per operation to each of the operations in a sequence.

Example 2 : Binary Counter

- Consider a k-bit binary counter counting from 0 in upward direction. For its representation we consider an array of size k as X[0: k–1]. A binary counter performs a single basic INCREMENT operation.
- The lowest-order bit of a k-bit binary number y is stored in X[0] and the highest-order bit of y is stored in X[k – 1] so that

$$y = \sum_{i=0}^{k-1} X[i]\, 2^i$$

- Initially, y=0, and thus X[i] = 0 for i =0, 1, …, k–1. For increasing a counter by adding 1 (modulo 2^k) to the counter value, a binary counter performs the following procedure.

```
INCREMENT (X, k)
/* Input: X[0: k–1] is an array that represents a k-bit binary
counter counting from 0 in upward direction. k is the size of an
array X.*/
{
  i := 0;
  {
    X[i] := 0 ;
    i := i + 1 ;
  }
  if (i < k)
    X[i] := 1 ;
}
```

- Table 7.1.1 illustrates what happens to an 8-bit binary counter when incremented it 16 times, starting from 0 to 16. The shaded bits are flipped to get the next value. The bold-faced values the flipped bits in each INCREMENT. The running time cost for flipping bits is given at the right. It can be observed that the total running time cost is always lesser than twice the total number of INCREMENT operations.

Table 7.1.1 : An 8-bit binary counter counting from 0 to 16

Counter Value	X[7]	X[6]	X[5]	X[4]	X[3]	X[2]	X[1]	X[0]	Total Cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

- As given in INCREMENT procedure, each iteration of the *while* loop adds a 1 into position i. If X[i] = 1, then adding 1 flips the bit in position i to 0 and gets a carry of 1, to be added into the position i + 1 on the next iteration of the *while* loop. Otherwise, the *while* loop ends.
- Then, if i < k and we know that X[i] = 0, so that addition of a 1 into position i, flips the 0 to a 1.
- The running time cost of each INCREMENT operation is linearly growing with the number of bits flipped. Each INCREMENT operation takes time $\Theta(k)$ in the worst case when a k-bit array X contains all 1s. Thus, a k-bit binary counter with an initial value of 0 has the worst-case running time as O(nk) for performing a sequence of n INCREMENT operations.
- Aggregate analysis can give a better upper bound than O(nk) by analysing a complete sequence of n INCREMENT operations. It is observed that in each INCREMENT operation all bits of k-bit array do not flip.
- As shown in Table 7.1.1 X[0] flips each time INCREMENT function is called. The next upward bit X[1] flips only every other time when a sequence of n INCREMENT operations on a k-bit counter with initial value 0 causes X[1] to flip $\lfloor n/2 \rfloor$ times. Similarly, a bit X[2] flips only every fourth time, or $\lfloor n/4 \rfloor$ times when a sequence of n INCREMENT operations is executed.

- Hence, in general, for i = 0, 1, ..., k–1, a bit X[i] flips $\lfloor n/2^i \rfloor$ times when a sequence of n INCREMENT operations is performed on an initially zero k-bit counter. For i ≥ k, there is no bit X[i], and so it cannot flip. Thus, the total number of flips in a sequence of n INCREMENT operations is

$$\sum_{i=0}^{k-1} \left| \frac{n}{2^i} \right| < n \sum_{i=0}^{\infty} \frac{1}{2^i}$$

$$= 2n \quad \text{...\{By applying the rule of infinite geometric series\}}$$

$$= O(n)$$

- The worst-case running time for a sequence of n INCREMENT operations when performed on an initially zero k-bit counter is therefore O(n).
- Thus, we get the average cost of each INCREMENT operation in a sequence of n INCREMENT operations on a binary counter = $\frac{O(n)}{n}$ = O(1). This is considered as the "amortized cost" of each INCREMENT operation on a binary counter counting from 0.
- Thus, the aggregate analysis demonstrates the worst-case bound of O(n) on a sequence of n INCREMENT operations on an initially zero binary counter by assigning an average cost per operation to each of the operations in a sequence.

7.1.2 Accounting Method

UQ. What is amortized analysis? Explain aggregate and accounting techniques with example.
SPPU - Q. 6(a), May 19, 8 Marks

- This method of amortized analysis assigns different costs to different operations in a sequence. Some operations are assigned more amortized costs, and some are assigned lesser amortized costs than the actual cost.
- When the amortized cost of operation is larger than its actual cost, the excess amount is assigned to the particular object in a data structure as its **"credit"**. This credit is consumed for other operation whose amortized cost is lesser than its actual cost. Total credit in the data structure is always positive.
- For any sequence of operations, the total amortized cost defines the upper bound to the total actual cost. Suppose a_i is the actual cost of i^{th} operation and $\hat{a}_i$ is the amortized cost assigned to the i^{th} operation then,

$$\sum_{i=1}^{n} \hat{a}_i \geq \sum_{i=1}^{n} a_i$$

∴ Total credit on an object of a data structure by performing a sequence of n operations:

$$= \sum_{i=1}^{n} \hat{a}_i - \sum_{i=1}^{n} a_i$$

Example 1 : Stack operation

- Recall a stack example with operations PUSH (X, p), POP(X) and MULTI-POP (X, m) as described in the previous section of aggregate analysis. Consider different costs assigned to these operations :

Operation	Actual cost	Amortized cost
PUSH (X, p)	1	2
POP (X)	1	0
MULTI-POP(X, m)	min (stack-size, m)	0

- For n PUSH operations, the amortized cost will be 2n whereas the actual cost of n PUSH operations will be n. So, total credit = 2n – n = n will be assigned to the stack object after performing n PUSH operations.
- As there is no amortized cost assigned to POP and MULTI-POP operation these operations can be performed by using the credit assigned to the stack object.
- The actual cost of a single POP(X) operation is 1 and the credit of the stack object after n PUSH operations is n. Also, the credit cannot be negative. Therefore, at most n POP operation separately or including MULTI-POP can be performed by consuming the credit n of the stack object.
- Thus, for any sequence of n PUSH, POP and MULTI-POP operations, the total amortized cost defines the upper bound to the total actual cost. As in this example the total amortized cost = O(n), the total actual cost of n stack operations = O(n).

Example 2 : Binary Counter

- Recall a binary counter example with an INCREMENT operation as described in the Example 1 in the previous section of aggregate analysis. As observed prior, the actual cost of INCREMENT operation is linearly varying with the number of bits flipped.
- Consider different costs assigned to flip a bit:

Operation	Actual cost	Amortized cost
Flipping a bit from 0 to 1 (Setting)	1	2
Flipping a bit from 1 to 0 (Resetting)	1	0

- For flipping a bit from 0 to 1 (Setting) a single unit cost (out of 2 units amortized cost charged) is used and a balance of 1 unit of amortized cost will be assigned as a credit to an account of a bit for flipping it back to 0 (Resetting).
- Thus, at any time, each bit with a value 1 in a binary counter has 1 unit of credit on it, that can be used for resetting it to 0. As the number of 1s in the counter never becomes negative, the amount of credit on a bit with value 1 stays always nonnegative.
- Now we perform the amortized analysis of an INCREMENT operation. Within the *while* loop the cost of resetting the bits is paid by the credit on the bits that are reset. The INCREMENT function sets at most one bit, in *if* loop, and thus, the amortized cost of a single INCREMENT operation is at most 2 units.
- Thus, for n INCREMENT operations, the total amortized cost is $2n = O(n)$, which gives the upper bound on the total actual cost.

7.1.3 Potential Function Method

GQ. Discuss the potential method for amortized analysis. Discuss it with operations on the stack. **(6 Marks)**

- The potential method of amortized analysis considers the prepaid task as **"potential energy"** of a whole data structure and utilizes that potential for paying other tasks in future.
- Consider that n operations are carried out on a data structure with an initial state S_0. Let S_i be the state of a whole data structure obtained after performing an i^{th} operation whose actual cost is a_i.
- The function Φ is selected such that it maps each state of a data structure S_i to a real number, $\Phi(S_i)$. Φ is known as a **"potential function"** and $\Phi(S_i)$ is called as the **"potential energy"** of a data structure S_i. The amortized cost of the i^{th} operation is $\hat{a}_i$. The potential energy-based amortized cost of i^{th} operation is defined as $\hat{a}_i = a_i + \{\Phi(S_i) - \Phi(S_{i-1})\}$.

$\therefore$ Total amortized cost of n operations $= \sum_{i=1}^{n} \hat{a}_i$

$$= \sum_{i=1}^{n} \{a_i + \Phi(S_i) - \Phi(S_{i-1})\}$$

$$\therefore \sum_{i=1}^{n} \hat{a}_i = \sum_{i=1}^{n} a_i + \Phi(S_n) - \Phi(S_0)$$

...(By applying scope rule)

- Here, $\Phi(S_n)$ is the potential associated with a data structure after performing n operations on it and $\Phi(S_0)$ is the potential associated with an initial state of a data structure.
- The potential function Φ is chosen such that $\Phi(S_n) \geq \Phi(S_0)$. Then the total amortized cost $\sum_{i=1}^{n} \hat{a}_i$ defines an upper bound on the total actual cost $\sum_{i=1}^{n} a_i$.
- If $\Phi(S_i) - \Phi(S_{i-1}) > 0$, then the amortized cost $\hat{a}_i$ indicates the overcharging to the i^{th} operation and a rise in potential $\Phi(S_i)$.
- If $\Phi(S_i) - \Phi(S_{i-1}) < 0$, then the amortized cost $\hat{a}_i$ indicates the undercharging to the i^{th} operation and a drop in potential $\Phi(S_i)$. It is because of paying for the actual cost of the i^{th} operation.

Example 1: Stack operation

- Recall the same example of stack operations PUSH (X, p), POP(X) and MULTI-POP (X, m) described in the section of aggregate analysis.
- Consider the potential function Φ is defined on the stack-size.
- Initially, for an empty stack $\Phi(S_0) = 0$. As stack-size is always positive, the stack S_i resulting after the i^{th} operation will have potential $\Phi(S_i) \geq 0$.
- Thus, the total amortized cost of n PUSH, POP and MULTI-POP operations on the stack by referring a potential function Φ gives an upper bound on the actual cost.
- Computation of amortized costs of different stack operations are given below :

(1) **PUSH (X, p):** As it pushes a single element onto the stack of size k, the stack size will increase to k + 1.

The actual cost of 1 PUSH operation is 1.

$\therefore$ The amortized cost of the PUSH operation is,

$$\hat{a}_i = a_i + \{\Phi(S_i) - \Phi(S_i - 1)\}$$
$$= 1 + \{(k+1) - k\} = 2$$

(2) **POP(X) :** As it pops the top of the stack of size k, the stack size will decrease to k – 1 after a pop operation.

The actual cost of 1 pop operation is 1.

$\therefore$ The amortized cost of the pop operation is,

$$\hat{a}_i = a_i + \{\Phi(S_i) - \Phi(S_{i-1})\}$$
$$= 1 + \{(k-1) - k\} = 0$$

(3) **MULTI-POP (X, m) :** It pops z = minimum (k, m) elements of the stack of size k. The actual cost of this operation is z.

∴ The amortized cost of the MULTI-POP operation is,

$$\hat{a}_i = a_i + \{\Phi(S_i) - \Phi(S_{i-1})\} = z + \{(k - z) - k\} = 0$$

- Since, the amortized cost of each PUSH, POP and MULTI-POP operation is O(1), the total amortized cost of sequence of n PUSH, POP and MULTI-POP operations is O(n).
- As $\Phi(S_i) \geq \Phi(S_0)$, the total amortized cost of n operations gives an upper bound on the total actual cost. Hence, the worst-case cost of n stack operations is O(n).

☞ Example 2: Binary Counter

- Recall the Example 2 of an incrementing binary counter using a single INCREMENT operation described in the section of aggregate analysis. Consider the potential function Φ is defined on the number 1s in a binary counter.
- Let p_i be the number 1s in the binary counter B_i resulting after the i^{th} INCREMENT operation. As the number of 1s in the counter never becomes negative, the potential $\Phi(B_i) \geq 0$. Thus, the total amortized cost of n INCREMNT operations on the initially zero binary counter by referring a potential function Φ gives an upper bound on the actual cost.
- Now we compute amortized cost of an INCREMENT operation on the initially zero binary counter.
- Assume that the i^{th} INCREMENT operation resets q_i bits to 0. Therefore, the actual cost of i^{th} INCREMENT operation is at most $1 + q_i$ as in addition to resetting q_i bits, each INCREMENT operation sets at most one bit to 1.
- If the number of 1s in a k-bit binary counter resulting after the i^{th} INCREMENT operation is zero i.e., if $\Phi(B_i) = p_i = 0$, that means the i^{th} INCREMENT operation resets all k bits, and so,

$$\begin{aligned}\Phi(B_{i-1}) &= p_{i-1}\\ &= q_i\\ &= k.\end{aligned}$$

- If $\Phi(B_i) > 0$ i.e., $p_i > 0$ then $p_i = p_{i-1} - q_i + 1$.
- In either case, $p_i \leq p_{i-1} - q_i + 1$, and the potential difference is given as below:

$$\begin{aligned}\Phi(B_i) - \Phi(B_{i-1}) &= (p_{i-1} - q_i + 1) - (p_{i-1})\\ &= (1 - q_i)\end{aligned}$$

∴ The amortized cost of the INCREMENT operation is,

$$\begin{aligned}\hat{a}_i &= a_i + \{\Phi(B_i) - \Phi(B_{i-1})\}\\ &\leq (1 + q_i) + \{(1 - q_i)\}\\ &= 2\end{aligned}$$

- As the amortized analysis gives an upper bound on the total actual cost, the worst-case running time of n INCREMENT operations on the initially zero binary counter is $2n = O(n)$.
- The potential function method easily analyses the counter even when it does not start at zero. The counter starts with the number 1s = p_0, and after n INCREMENT operations it has with the number 1s = p_n, where $p_0 \leq 0$ and $p_n \leq k$.
- Hence, a total cost of n INCREMENT operations is written as below:

$$\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} \hat{a}_i - \Phi(B_n) + \Phi(B_0)$$

- We have $\hat{a}_i \leq 2$, for all $1 \leq i \leq n$. Since $\Phi(B_0) = p_0$ and $\Phi(B_n) = p_n$ the total actual cost of sequence of n INCREMENT operations is given as

$$\begin{aligned}\sum_{i=1}^{n} a_i &= \sum_{i=1}^{n} 2 - p_n + p_0\\ &= 2n - p_n + p_0\end{aligned}$$

- Particularly as $p_0 \leq k$, as long as $k = O(n)$, the total actual cost of incrementing a k-bit binary counter is O(n). That means, to perform at least $n = \Omega(k)$ INCREMENT operations on a k-bit binary counter the total actual cost is O(n), irrespective of the initial value of the counter.

⏭ 7.2 TIME-SPACE TRADE-OFF

- The algorithm that consumes less memory space and less time to its completion is said to be the best algorithm.
- However, attaining both objectives is not always feasible in practice. Some algorithms are time-efficient, and some are space-efficient.
- Sometimes we have to sacrifice one at the cost of the other. Thus, there exists a **time-space trade-off** among algorithms.
- A problem can be solved by multiple approaches. Some approaches may consume lesser memory space but may need more running time and some faster approaches may need more memory space.

- If there is limited memory space, then we have to select an algorithm that needs less space at the cost of more running time.
- If time is our constraint, then we have to select a faster algorithm at the cost of more space.
- The lower bounds in time-space trade-off help to prove that, for specific problems, no algorithms exist that can satisfy both time and space constraints simultaneously.
- In real-time applications, the time-space trade-off is handled by selecting an optimal algorithm based on the given instance of a problem.

☞ Examples of time-space trade-off

(1) Recalculation Vs Lookup tables

- Recalculation of values as and when required causes more computation time but reduces memory requirements.
- However, the use of a lookup table saves computing time by avoiding redundant calculations at the cost of more memory space.

(2) Compressed Vs Un-compressed data

- Compressed data saves memory, but it takes more time to execute compression and decompression algorithms.
- Un-compressed data needs more space, but data access is faster.

(3) Iterative process Vs Recursive process

- The iterative process has faster execution but lengthy code whereas the recursive process has smaller code but slower execution due to the overhead of repeated function calls.

7.3 INTRODUCTION TO TRACTABLE AND NON-TRACTABLE PROBLEMS

UQ. Explain Tractable and non-tractable problems with example. **SPPU - Q. 5(b), May 18, 8 Marks**

- In Chapter 2 we have discussed the complexity classes of different algorithms. It is based on the asymptotic efficiency of an algorithm.

Running Time	Asymptotic efficiency class	Growth rate	Example
Polynomial	1	Constant	Single Push or Pop operation on a stack
	log n	Logarithmic or Sublinear	Binary search algorithm.
	n	Linear	Sequential search algorithm.
	n log n	Quasi-linear	Merge sort by divide and conquer strategy.
	n^2	Quadratic	Bubble sort.
	n^3	Cubic	Floyd-Warshall's algorithm.
Exponential	2^n	Exponential	Travelling salesperson problem (TSP) by dynamic programming.
	n!	Factorial	Travelling salesman problem (TSP) by brute-force method.

- Some problems have polynomial running time while some problems take exponential time for their execution.
- Based on the running time of the problems they are broadly categorized as:
  (1) Tractable problems
  (2) Non-tractable problems

▶ Tractable problems

- Tractability defines the feasibility of an algorithm to complete its execution in a reasonable time.
- Problems are said to be **tractable** if they get solved in polynomial time using the deterministic algorithms. E.g., Time complexities $O(n^2)$, $O(n^3)$, O(1), O(n log n).
- These problems are P-class problems.
- Some examples of tractable problems: linear search: O(n), binary search: O(log n), merge sort: O(n log n), Prim's algorithm: $O(n^2)$.

▶ Non-tractable problems

- **Non-tractability** (or **intractability**) defines the infeasibility of an algorithm to complete its execution in a reasonable time.
- Problems are said to be **non-tractable** (or **intractable**) if they cannot be solved in polynomial time using the deterministic algorithms.
- There are some non-tractable but decidable problems with exponential runtime. E.g., Time complexities: $O(2^n)$, $O(n^n)$, O(n!)

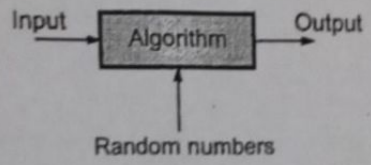
- These problems are NP-class problems or NP-Hard problems.
- Some examples of non-tractable problems: 0/1 knapsack problem: $O(2^{n/2})$, TSP: $O(n^2 2^n)$, m-colouring problem: $O(nm^n)$
- Some non-tractable problems can be effectively solved by approximation algorithms or randomized algorithms.

7.4 RANDOMIZED ALGORITHMS

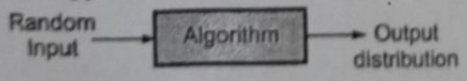
UQ. Write a short note on Randomized algorithm.
SPPU - Q. 4(a), Dec. 15, 8 Marks

UQ. Which are different approaches to writing a Randomized Algorithm? Write a Randomized sort Algorithm.
SPPU - Q. 3(a), Dec. 16, 8 Marks

- To overcome the drawback of the exponential running time of some deterministic algorithms, the randomized algorithms are used.
- An algorithm that makes some random choices during its execution depending on random numbers for its operation is called a **"randomized algorithm"**.
- In addition to input, the randomized algorithms take a source of random numbers to decide what to do next in the logic.



- Its behaviour is likely to be good for all inputs (depending on the likelihood of random numbers).
- The behavior of a randomized algorithm can vary even though its input is fixed.
- It is also known as a **"probabilistic algorithm"** as the input is supposed to be from a probability distribution.



- E.g., Randomized Quicksort: Here, a random number is selected as the next pivot (or an input array is randomly shuffled).

Significance of the randomized algorithms

- For some algorithms, it is preferable to randomly choose a course of action, rather than spending time to determine the best choice.
- Also, for some problems, there is no better technique than making random choices.
- The randomized algorithms avoid the worst-case behaviour of the classic deterministic algorithms.
- These algorithms attempt to achieve good performance in average case behaviour.
- These algorithms provide efficient approximate solutions to some intractable problems.
- These algorithms can be designed by applying different paradigms and approaches.

Properties of the randomized algorithms

- These algorithms are non-deterministic i.e. the same algorithm may produce different output for the same input in its multiple executions.
- These algorithms can make random but most of the times correct decisions.
- Sometimes these algorithms are not very precise. With more time they give the better precision.
- These algorithms may operate at different efficiencies in different runs for the same instance of a problem.
- Many times, these algorithms produce improved results than the exponential deterministic algorithms.

Classes of the randomized algorithms

(1) Las Vegas Algorithms

- This class of randomized algorithms always computes the same (correct or optimum) result for the same input.
- The running times of these algorithms vary with the output of the randomizer. The variation in execution time between runs is significant in Las Vegas algorithm.

(2) Monte Carlo Algorithms

- This class of randomized algorithms always computes different results in different runs for the same input.
- They produce output with some probability for the fixed input.
- They do not show much variation in running time between runs.

Complexity analysis of the randomized algorithms

- The complexity analysis of a randomized algorithm is given by the expected runtime.
- It is the expected value of the random variable selected during execution of a randomized algorithm.

- It effectively "averages" over all the sequences of random "numbers.
- Though it is similar to the average case analysis of the deterministic algorithms, practically it assures that the worst-case behavior is not elicited by any particular input.
- Some applications of the randomized algorithms
  (1) Cryptography
  (2) Load balancing
  (3) Gamification
  (4) Statistical analysis
  (5) Emergent system generation using genetic hybridization
  (6) Mathematical programming
  (7) Data Structures
  (8) Pattern recognition
  (9) Computational geometry.

7.4.1 Randomized Quicksort

- The best-case and the average case time complexity of quicksort algorithm is O(n log n) whereas its worst-case time complexity is $O(n^2)$.
- The imbalance sizes of the sub-problems result in the worst-case of quicksort. (E.g., if the input array of elements is already sorted.)
- Thus, the selection of the good pivot element is very crucial as it can split the original problem into the sub-problems having balanced sizes.
- This problem can be resolved by applying randomization for the selection of the pivot element.
- Randomization cannot abolish the worst-case, but it can make it less likely!
- The randomized quicksort each time picks a random element in an input array as the pivot.

Significance of the randomized quicksort

- Its time complexity is independent of the order of the input elements. It depends only on the sequence of random numbers.
- It does not assume any input distribution.
- The worst-case behavior is not determined by any specific input; however, it is determined only by the sequence of random numbers.
- In the randomized quicksort, the worst-case becomes less likely.
- The randomization reduces the worst-case running time of $O(n^2)$ of quicksort to O(n logn).

Randomized algorithm

```
Algorithm Random_QSort (X, first, last)
/*Input: X[first : last] is an array to be sorted in ascending order.
Output: A sorted array X in ascending order.*/
{
  if (first = last)
     return X[first] ;
                        /* There is only one element in X[], so it is sorted. */
  else if (first < last)
  {
     p := Random_Partition (X, first,last) ;
                        /* Random_Partition(X) randomly picks any element from an array X as a pivot element and returns its index.*/
     Random_QSort (X, first, p – 1);
                        /* Sort the subarrays by recursively calling an algorithm Random_QSort(). */
     Random_QSort (X, p + 1, last) ;
  }
}

Algorithm Random_Partition (X, first, last)
/* It applies randomization in pivot selection.
Input: An array X [first : last] whose elements are partitioned around a pivot element.
Output: Index p of a pivot element.*/
{
/*At each step of the algorithm the element X[first] is exchanged with a randomly selected element from
X[first : last]. The pivot element p = X[first] is equally likely to be any one of the last – first + 1 elements
of X.*/
  i := Random_Num(first, last);
                        /*Random_Num(first, last) randomly selects any index between first and last.*/
  swap X[first] and X[i];
  p := Partition(X, first, last);
                        /* Returns an index p of a pivot element around which an array X gets partitioned.*/
     return p;
```

```
Algorithm Partition (X, first, last)
* It uses the Hoare's partition as it is used in the non-
randomized quicksort algorithm.
Input: An array X [first : last] whose elements are partitioned
around a pivot element X[p], first ≤ p ≤ last and the elements
in X are rearranged in such a way that X[k] ≤ X[p] for first ≤
k < p and X[k] ≥ X[p] for p < k ≤ last.
Output: Index p of a pivot element. */
{
  pivot := X[first] ;        /* Consider the first element as
                                the initial pivot as proposed by
                                C.A.R. Hoare. */
  down := first ;            /* Index pointer for the start of an
                                array. */
  up := last + 1 ;           /* Index pointer for the end of an
                                array. */
  repeat
  {
    repeat
      down := down + 1 ;     /* To move forward from
                                the starting element of an
                                array. */
    until (X[down] ≥ pivot);

    repeat
      up := up – 1 ;         /* To move backward from
                                the last element of an array.*/
    until (X[up] ≤ pivot) ;
    if (down < up)           /* Swap X[up] and X[down] */
    {
      temp := X[down] ;
      X[down] := X[up] ;
      X[up] := temp ;
    }
  } until (down ≥ up) ;      /* Cross over point reaches
                                during forward and backward
                                scan*/
  X[first] := X[up] ;
  X[up] := pivot ;
  return up ;                /* Returns an index of a pivot
                                element around which an array X
                                gets partitioned. */
}
```

7.5 APPROXIMATION ALGORITHMS

UQ. Explain the concept of Randomized algorithm and Approximation algorithm in brief with example.

SPPU - Q. 6(b), Dec. 19, Q. 4(a), Dec. 17, Q. 5(a), May 19, 8 Marks

- An algorithm that produces a solution S which is **"close"** to the optimal solution S* is known as an **"approximation algorithm".**
- The **"closeness"** of a solution to the problem instance of size n is generally measured by the ratio bound $\rho(n)$ the algorithm produces.
- The ratio bound $\rho(n)$ is a function that satisfies $\max\left\{\frac{S}{S^*}, \frac{S^*}{S}\right\} \leq \rho(n)$ for any input size n.
- An algorithm with an approximation ratio of $\rho(n)$ is known as **"$\rho(n)$-approximation algorithm".**
- For a maximization problem, $0 < S^* \leq S$, and the ratio S^*/S gives the factor by which the value of an optimal solution is larger than the value of the approximate solution.
- For a minimization problem, $0 < S^* \leq S$, and the ratio S/S^* gives the factor by which the value of the approximate solution is larger than the value of an optimal solution.
- The approximation ratio $\rho(n)$ is never less than 1.
- $\rho(n) = 1 \Rightarrow S = S^* \Rightarrow$ a 1-approximation algorithm gives an optimal solution.
- The larger approximation ratio indicates that an approximation algorithm produces a worse solution than optimal.

Significance of the Approximation Algorithms

- The approximation algorithms naturally arise as a consequence of the generally believed $P \neq NP$ conjecture.
- These algorithms provide efficient near-optimal solutions to some intractable optimization problems.
- They are useful to give approximate solutions to NP-Hard problems; however, they can also be used to give fast approximations to P-class problems (that run in polynomial time.

- The closeness of the solution by an approximate algorithm to the optimal one can be mathematically proven.

Types of the Approximation Algorithms

(1) **Absolute approximation algorithm:** An algorithm is said to be an absolute approximation algorithm for a problem instance of size n if there exists some constant k such that $|S - S^*| \leq k$ where S is an approximate solution to the optimal solution S*.

(2) **Relative approximation algorithm:** An algorithm is said to be a relative approximation algorithm for a problem instance of size n if there exists some constant k such that $\max\left\{\frac{S}{S^*}, \frac{S^*}{S}\right\} \leq k$ where S is an approximate solution to the optimal solution S*.

7.5.1 Travelling Salesman Approximation

- **The travelling salesperson problem (TSP)** asks to minimize the cost (or length) of a tour that visits all cities only once and ends at the starting city.
- It is an NP-Complete problem with exponential complexity of $O(2^n)$ where n is the number of nodes in a graph.
- The complexity of a 2-approximation TSP algorithm with triangle inequality is $O(n^2)$ if Prim's algorithm is used to construct an MST of a given graph of n nodes.

The basic steps of an approximation algorithm to solve TSP:

(1) Compute a minimum spanning tree (MST) whose weight describes a lower bound on the cost (or length) of an optimal tour by a travelling salesman.

(2) Using this MST determine a tour whose cost is no more than twice of the MST's weight, provided that the cost function satisfies the triangle inequality.

- Let G = (V, E) be given graph where V is a set of nodes and E is a set of edges. The cost function f satisfies the triangle inequality if $f(u, v) \leq f(u, w) + f(w, v), \forall u,v, w \in V$.

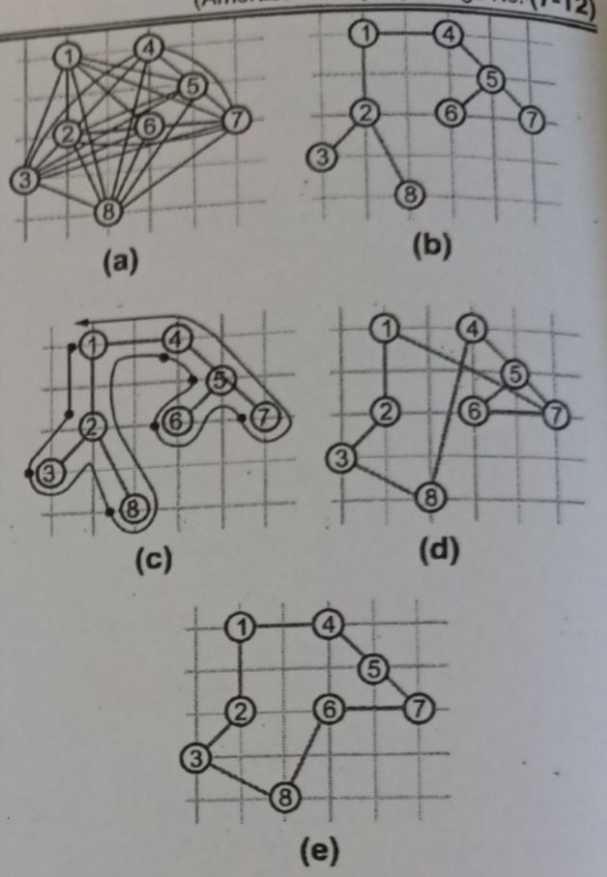


Fig. 7.5.1 : The working of TSP approximation

- Fig. 7.5.1 depicts the working of TSP approximation.
  - **Part(a)** shows a given graph G.
  - **Part(b)** shows an MST(T) for G from starting node 1.
  - **Part(c)** shows a preorder traversal of T from node 1. Here the dot indicates the first visit of a node. The complete walk W of an MST(T) is usually not a Hamiltonian cycle as it visits some nodes more than once. Thus W = 1, 2, 3, 2, 8, 2, 1, 4, 5, 6, 5, 7, 5, 4, 1.
  - **Part(d)** shows a Hamiltonian cycle S (approximate solution) in G. By removing the node-entries that are visited more than once the resultant W includes only the first visits to all nodes. Thus, W = 1, 2, 3, 8, 4, 5, 6, 7.
  - **Part(e)** shows an optimal solution S* to TSP.

Approximation Algorithm for TSP

```
Algorithm Approx_TSP (G, f)
/*It finds the approximate solution to TSP by using a cost
function based on triangle inequality.
Input: G = (V, E) be given graph where V is a set of
nodes (cities) and E is a set of edges (paths). F is a cost
function based on triangle inequality.
Output: A near-optimal travelling salesman tour. */
{
  Choose a starting node (root) v0 ∈ V;
  Construct a minimum spanning tree (MST) T for G
    from root v0;
  Let S be an ordering of nodes in which they are first
    visited in a preorder traversal of T;
  return the Hamiltonian cycle S;
}
```

7.6 EMBEDDED ALGORITHMS

GQ. Explain embedded system. **(4 Marks)**

Embedded Systems

- A computer hardware system with software embedded in it is called an **"embedded system"**.
- It is a reliable, microcontroller or microprocessor-based, software-driven, real-time control system.
- It is dedicated to the specific functionality of a complex larger system.

Characteristics of an Embedded System

(1) **Single-functioned**: Usually an embedded system is designed to perform a specialized operation.

(2) **Tightly constrained**: It has tight constraints on design metrics like its size, cost, memory, power, and performance.

(3) **Reactive and real-time**: It must persistently react to the changing system environment and must produce results in real-time.

(4) **Microprocessors-based**: It must be microcontroller or microprocessor-based. Embedded processors consume less power.

(5) **Memory**: The software of an embedded system usually embeds in ROM. It does not require secondary memory in the computer.

(6) **Connected**: All peripherals of an embedded system are connected to input and output devices.

(7) **H/W-S/W systems**: In an embedded system, the hardware provides security and performance and the software provides more flexibility and features.

Advantages of an embedded system

(1) It is easily customizable.
(2) It consumes low power.
(3) It has a low cost.
(4) It has real-time enhanced performance.

Disadvantages of an embedded system

(1) It needs high efforts for its development.
(2) Once configured, its upgradation or modification is not possible.
(3) It is difficult to troubleshoot.
(4) It takes larger time to market.

Embedded algorithms

- The algorithms implemented on the microcontroller or microprocessor-based systems are known as **"embedded algorithms"**.
- They solve the specialized operations in the real-time control systems.
- Generally, the embedded algorithms are similar to the algorithms used in the non-embedded systems, but they need to satisfy the tight constraints of the embedded systems.
- **The challenges for designing an embedded algorithm:**
  - **Limited memory:** The on-chip memory of the embedded processors is in KBs as against GBs for desktop PCs.
  - **Processor architectures:** The embedded systems consist of the microcontrollers or microprocessors. These processors are very small, with very small storage capacity. As these processors are programmed for specialized tasks, they have a variety of CPU architectures.
  - **Low power consumption:** An embedded algorithm must consume fewer computing resources and hence the less power.
  - **Deadline based scheduling:** The embedded systems need to produce prompt results in real-time according to the changes in the system environment. Hence the embedded algorithms must have dynamic priority-based scheduling.
  - **Less cost:** The cost of embedded systems should be less. Hence it needs optimized algorithms.

- **The basic guidelines to design the embedded algorithms:**
  - Adapt a fixed-point or integer-based representation to accommodate the processors with limited word lengths.
  - Integrate data management schemes, like streaming, buffering, pipelining, to meet the requirements of real-time data processing.
  - Explore alternative designs to fulfil the smaller memory requirement and optimized computations.
  - Avoid dynamic memory allocation as it causes heap fragmentation problems and allocation performance issues.
  - Design the non-recursive algorithms instead of the recursive procedures.
  - Design the algorithms with a nice bound on their running times.

7.6.1 Embedded System Scheduling (Power-Optimized Scheduling Algorithm)

UQ. Explain embedded system? Explain the scheduling algorithm for embedded system in detail.

SPPU - Q. 5(b). May 19, 8 Marks. Q. 7(b). Dec. 15. 9 Marks

- The software that decides which operation should be executed next is called a **"scheduler"**.
- The algorithm that describes the logic and the mechanism of the scheduler is called the **"scheduling algorithm".**
- The embedded systems need to produce prompt, reactive results in real-time. Thus, it needs deadline-based scheduling that can determine the execution order of operations based on their dynamic priorities.
- Also, power is another major constraint in the embedded systems. Energy consumption is a very critical issue in the embedded systems mainly, for battery-powered devices. And hence the power-optimized scheduling algorithms are highly essential in the embedded systems.
- **The embedded scheduling algorithm has to evaluate the following factors** to decide the execution order:
  - Interdependencies of the tasks and the resources
  - Dynamic priorities of the tasks
  - Requirements and availability of the computing resources
  - The cost incurred in the execution
  - Power consumption

Different approaches to the power-optimized scheduling algorithm:

(1) Efficient utilization of the processor

- Due to dependencies and timely varying workload of all tasks in real-time embedded systems the processor utilization is not 100%. The efficient utilization of the processor can reduce power consumption.
- So, the power-optimized scheduling algorithms need to detect the ordering of tasks such that their execution efficiently and effectively utilizes the computing resources leading to lesser power consumption.

(2) Enforcement policy

- The scheduler can apply an enforcement policy to remove or just pre-empt the task if it attempts to consume more resources and hence more power.
- Here, the scheduler has to do the cost-benefit analysis.

(3) Instruction rescheduling

- Reordering of the instructions can eliminate the pipeline stalls caused by structuring hazards, data hazards, and control hazards.
- It can thus improve the code performance, that results in effective power saving.
- The idea is to reorder the instructions while preserving functional correctness and performance.
- Thus, the power-optimized scheduling algorithm can focus on the instruction rescheduling to minimalize the Hamming distance of successive instructions without affecting the correctness.

(4) A system-level power management policy

- Shutting down the inactive resources can save power.
- The power-optimized scheduling algorithm can decide on scheduling tasks on different resources and thus can decide on periods of inactivity.
- Accordingly, a system-level power management policy can be adopted to determine how and when the various components of an embedded system should be shut down.

7.6.2 A Sorting Algorithm for the Embedded Systems

UQ. Explain in detail a sorting algorithm for the embedded systems.

SPPU - Q. 7(b), May 17, 9 Marks. Q. 5(b), Dec. 19, 8 Marks

UQ. What is an Embedded System? Explain embedded sorting algorithm.

SPPU - Q. 6(a), May 18, 8 Marks

GQ. Explain the essential characteristics of the embedded sorting algorithm. **(4 Marks)**

A sorting algorithm for the embedded system does not differ with the logical steps of a general sorting algorithm; however, it must have the following characteristics:

(1) **Sort in place:** It must sort in place. Thus, it saves memory and most importantly, it avoids the dynamic memory allocation.

(2) **Iterative:** It must be iterative rather than recursive. Recursion is slower and causes issues of stack overflow.

(3) **Invariable running time:** Its best, average and worst-case analysis should determine the similar running times. It should not vary enormously with the input data.

(4) **Reasonable code size:** Its code size should be proportionate with the problem.

(5) **Input size dependence:** Its running time should be a linear or logarithmic function of the input size of a problem instance (number of elements to be sorted).

(6) **Clean implementation:** Its implementation must be 'clean' i.e., it should avoid breaks and returns in the middle of a loop.

☞ Example of embedded sorting algorithm

- Insertion sort is a stable, non-recursive, in-place sorting algorithm.
- It scans through the input array, compares each pair of elements, and it swaps those elements if they are out of order. It develops the final sorted array one element at a time.
- It selects the first element from the upper end (right-hand side) of the unsorted array and compares it with every element starting from the right-most element in the non-decreasing sorted array.
- If the new element < the compared element, it swaps them.
- It is more efficient for small arrays, even faster than merge sort or quicksort. Also, it does not require extra space for sorting.
- For already sorted input array, quicksort takes the worst-case running time of $O(n^2)$; however, it is the best-case of insertion sort with a minimum time of O(n). Even for the nearly sorted input array insertion sort takes almost linear execution time.
- Thus in the best-case, insertion sort outperforms quicksort and merge sort.
- The worst-case and average-case time complexity of insertion sort is $O(n^2)$ time which is worse than O(n logn) of quicksort and merge sort. But quicksort requires extra stack space of O(logn) to record recursive calls and merge sort requires extra space O(n) to store merged sub-arrays into an auxiliary array.
- Though insertion sort and selection sort are similar, insertion sort has one distinct advantage. Selection sort compares all unsorted elements to determine the next smallest element. But insertion sort continues comparison with the next unsorted element until it has found the insertion point in the array.

```
Algorithm Insertion_Sort (X,n)
/*Input: An array X[1:n] of n elements to be sorted in increasing order, n ≥ 1.
Output: Array X[1:n] sorted in increasing order. */
{
  for (j := 2; j<n; j++)
  {
    /*X[1: j–1] is already sorted.*/
    i := j–1;
    item := X[j];
    while((i≥ 1) && item < X[i])
    {
      X[i+1]:=X[i];
      i := i–1;
    }
    X[i+1] := item;
  }
}
```

☞ Complexity analysis

Analysis	Number of comparisons	Number of swaps
Best-case	O(n)	No swaps
Average-case	$O(n^2)$	$O(n^2)$
Worst-case	$O(n^2)$	$O(n^2)$

Summary

- **Amortized analysis** assures the average performance in the worst case by averaging the cost over a sequence of steps even though the particular step within the sequence might be costlier.
- It gives an insight into a particular data structure. It is an effective way to analyse the operations on data structures.
- Amortized analysis has three methods: aggregate method, accounting method and potential method.
- The algorithm that consumes less memory space and less time to its completion is said to be the best algorithm. However, attaining both of these objectives is not always feasible in practice causing a **time-space trade-off** among algorithms.
- Problems are said to be **tractable** if they get solved in polynomial time using the deterministic algorithms. These problems are P-class problems.
- Some examples of tractable problems: linear search: O(n), binary search: O(log n), merge sort: O(n log n), Prim's algorithm: $O(n^2)$.
- Problems are said to be **non-tractable** (or **intractable**) if they cannot be solved in polynomial time using the deterministic algorithms.
- There are some non-tractable but decidable problems with exponential runtime. E.g., Time complexities: $O(2^n)$, $O(n^n)$, O(n!)
- An algorithm that makes some random choices during its execution depending on random numbers for its operation is called a **"randomized algorithm"**.
- **Classes of the randomized algorithms**

(1) **Las Vegas Algorithms**: They always compute the same (correct or optimum) result for the same input. The running times of these algorithms vary with the output of the randomizer.

(2) **Monte Carlo Algorithms**: They always compute different results in different runs for the same input.

- **The randomized quicksort** applies randomization for the selection of the pivot element to reduce the worst-case running time of $O(n^2)$ of quicksort to O(n log n).
- An algorithm that produces a solution S which is "close" to the optimal solution S* is known as an **"approximation algorithm"**.
- The "closeness" of a solution to the problem instance of size n is generally measured by the ratio bound $\rho(n)$ the algorithm produces. The ratio bound $\rho(n)$is a function that satisfies $\max\left\{\frac{S}{S^*}, \frac{S^*}{S}\right\} \leq \rho(n)$ for any input size n.
- An algorithm with an approximation ratio of $\rho(n)$ is known as **"$\rho(n)$-approximation algorithm"**.
- **Types of the approximation algorithms**

(1) **Absolute approximation algorithm:** An algorithm is said to be an absolute approximation algorithm for a problem instance of size n if there exists some constant k such that $|S - S^*| \leq k$ where S is an approximate solution to the optimal solution S*.

(2) **Relative approximation algorithm:** An algorithm is said to be a relative approximation algorithm for a problem instance of size n if there exists some constant k such that $\max\{S/S^*, S^*/S\} \leq k$ where S is an approximate solution to the optimal solution S*.

- A computer hardware system with software embedded in it is called an "**embedded system**". It is a reliable, microcontroller or microprocessor-based, software-driven, real-time control system.
- The software that decides which operation should be executed next is called a **"scheduler"**.
- The algorithm that describes the logic and the mechanism of the scheduler is called the **"scheduling algorithm"**.
- The embedded scheduling algorithm has to evaluate the following factors to decide the execution order:
  - Interdependencies of the tasks and the resources
  - Dynamic priorities of the tasks
  - Requirements and availability of the computing resources
  - The cost incurred in the execution
  - Power consumption
- **Different approaches to the power-optimized scheduling algorithm:**
  - Efficient utilization of the processor,
  - Enforcement policy
  - Instruction rescheduling
  - A system-level power management policy
- **Characteristics of the embedded sorting algorithm:**
  (1) Sort in place
  (2) Iterative
  (3) Invariable running time
  (4) Reasonable code size
  (5) Input size dependence
  (6) Clean implementation

Chapter Ends...

□□□

UNIT VI

CHAPTER 8

Multithreaded and Distributed Algorithms

Syllabus

Problem Solving using Multithreaded Algorithms - Multithreaded matrix multiplication, Multithreaded merge sort.
Distributed Algorithms - Introduction, Distributed breadth first search, Distributed Minimum Spanning Tree.
String Matching- Introduction, The Naive string matching algorithm, The Rabin-Karp algorithm.

8.1 Multithreaded Algorithms 8-3
GQ. Explain multithreaded algorithms. **(4/6 Marks)** 8-3
GQ. Write a note on dynamic multi-threading. **(4 Marks)** 8-3
8.1.1 Performance Measures 8-5
UQ. Define performance measure of multithreaded algorithms. Write a multithreaded algorithm for Fibonacci Series and explain performance measure of Fibonacci(6) execution with suitable diagram. **SPPU - Q. 7(b), May 18, 9 Marks** 8-5
GQ. Explain the performance measures: work, span, speedup, parallelism and slackness. **(4/6 Marks)** 8-5
8.1.2 Analyzing Multithreaded Algorithms, Parallel Loops, Race Conditions 8-7
UQ. Explain multithreaded algorithms. How to analyze multithreaded algorithms? What is a race condition, parallel loops? **SPPU - Q. 7(a), May 19, 9 Marks** 8-7
GQ. Write the analysis of multithreaded algorithms with an example. **(4/6 Marks)** 8-7
GQ. Explain the concept of parallel loops. **(6/8 Marks)** 8-7
GQ. What are the race conditions in multithreaded algorithms? Explain in detail. **(4/6 Marks)** 8-7
8.2 Problem Solving using Multithreaded Algorithms 8-10
8.2.1 Multithreaded Matrix Multiplication 8-10
UQ. Give pseudocode for Multithreaded matrix multiplication. Analyze the same. **SPPU - Q. 8(a), May 19, 9 Marks** 8-10
GQ Write and explain the algorithm for multithreaded matrix multiplication. **(4/6 Marks)** 8-10
8.2.2 Multithreaded Merge Sort 8-10
UQ. Write and Explain Multithreaded Merge Sort Algorithm. **SPPU - Q. 7(a), May 18, 9 Marks** 8-10
UQ. Write and explain multithreaded merge sort algorithm. Analyze the same. **SPPU - Q. 8(b), Dec. 19, 9 Marks** 8-10
8.3 Distributed Algorithms 8-12
GQ. What is the distributed algorithm? **(4 Marks)** 8-12
8.3.1 Distributed Breadth First Search 8-13
UQ. What is Distributed algorithm? Explain distributed Breadth First Search algorithm with an example. **SPPU - Q. 8(a), May 18, Q. 7(a), Dec. 2019, 9 Marks** 8-13

GQ. Explain distributed breadth first search algorithm for 1D partitioning. **(6 Marks)** 8-13
GQ. Explain distributed breadth first search algorithm for 2D partitioning. **(6 Marks)** 8-13
8.3.1(A) A distributed Breadth First search Algorithm For 1-D Partitioning 8-14
8.3.1(B) A Distributed Breadth First Search Algorithm for 2-D Partitioning 8-15
8.3.2 Distributed Minimum Spanning Tree 8-16
UQ. What is distributed algorithm? Explain Distributed Minimum Spanning Tree.
SPPU - Q. 8(b), May 19, 9 Marks 8-16
8.4 String Matching Algorithms 8-17
UQ. What are string matching algorithms? Explain any one algorithm with an example.
SPPU - Q. 7(b), Dec 19, 9 Marks 8-17
UQ. Give and explain the string matching algorithm. SPPU - Q. 8(b), May 17, 9 Marks 8-17
UQ Compare and contrast String matching algorithms. Explain any one algorithm with an example.
SPPU - Q. 8(b), May 18, 9 Marks 8-17
8.4.1 The Naïve String Matching Algorithm 8-18
8.4.2 The Rabin-Karp Algorithm 8-19
UQ. Write and explain the Rabin-Karp algorithm. Explain the worst case and best case running time of Rabin Karp Algorithm? SPPU - Q. 7(b), May 19, Q. 8(a), Dec 19, 9 Marks 8-19
GQ. Write a detailed note on the Rabin-Karp String matching algorithm and discuss its complexity. **(6/8 Marks)** 8-19
➤ **Chapter Ends** 8-22

Nowadays the multiprocessor systems are widely used, so the study of parallel algorithms is essential. We have discussed many sequential algorithms suitable for single-processor systems. In this chapter, we shall study parallel algorithms that allow the parallel execution of multiple instructions on multiple processors. Some multiprocessor systems share a common memory and some have distributed memory. The multithreaded algorithms provide parallel processing in chip multi-core systems and parallel computing models with shared memory. However, the distributed algorithms facilitate parallel processing in the distributed computing environment.

8.1 MULTITHREADED ALGORITHMS

GQ. Explain multithreaded algorithms. (4/6 Marks)

- **A thread** is a dispatchable portion of a task within a process. Each thread executes its code sequentially and independently.
- The threads are interruptible. A processor can switch threads within the same process. It is cheaper than a process switch.
- It has own stack and program counter. Multiple threads share a common memory.
- The algorithms that permit the concurrent execution of multiple threads of a program on multiple cores in a system with shared memory are known as **"multithreaded algorithms"**.
- In these algorithms, a single program is split into several threads of control which interact with each other to solve a single problem.
- A set of runtime instructions executed by a processor using a multithreaded program is known as a **"multithreaded computation"**.

The categories of multithreaded algorithms

GQ. Write a note on dynamic multi-threading. (4 Marks)

- There are two broad categories of multithreaded algorithms:
  - Static multithreaded algorithms
  - Dynamic multithreaded algorithms

Static multithreaded algorithms

- In these algorithms, threads are alive for the entire period of computation. It happens in many applications because creation and destruction of threads are slower operations.
- In static multithreading, splitting of a task in multiple threads and load balancing is very complex. It needs some complex communication protocols for the same and the programmers have to work on it.

Dynamic multithreaded algorithms

- These algorithms provide a concurrency platform to synchronize, schedule and handle the parallel-computing resources.
- Thus they reduce the programmer's work of implementing communication protocols and load balancing. A programmer has to describe only the logical parallelism within a program.
- In these algorithms, creation and destruction of threads are as simple as ordinary subroutine calls and their returns.
- **Different keywords** used in the multithreaded algorithms are as below:
  (1) **parallel:** It is used with the loop construct like *for* loops to mention that all iterations can be concurrently executed.
  (2) **spawn:** It is used to provide the nested parallelism. In a general function call, the parent does not resume until its child returns, however, in the function call with a keyword spawn, the parent and child concurrently continue their execution.
  (3) **sync:** By executing a sync statement a procedure can safely use the return values of its spawned children. The procedure is suspended until all of its children have executed completely. Then the procedure resumes the instruction following the sync step.
- **A concurrency platform** for dynamic multithreaded algorithms provides:
  (1) **A scheduler** to facilitate automatic load-balancing.
  (2) **Nested parallelism** to permit a subroutine to be spawned.
  (3) **Parallel loops** to execute the iterations of the loop concurrently.

A model of multithreaded execution

- Let G = (V, E) be a directed acyclic graph (DAG) that represents the multithreaded computation. It is also called as a **"computation DAG"**.

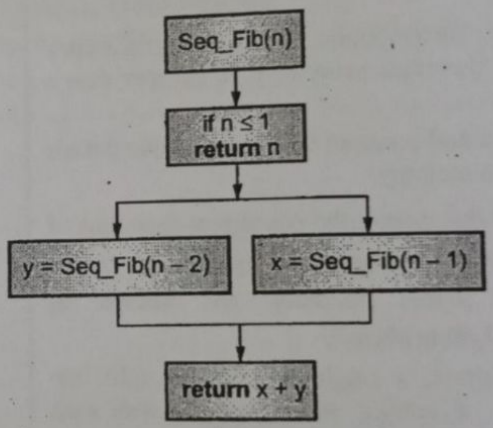
- Here, V is a set of vertices that represent the instructions and E is a set of edges that represent the dependencies between the instructions. An edge $\langle u, v \rangle \in E$ indicates that an instruction u must be executed before the execution of an instruction v.
- A strand represents a sequence of serial instructions. Such instructions do not contain any parallel control like spawn, sync, or return from a spawn.
- A directed path from strand u to strand v defines that the instructions u and v are logically sequential. Otherwise, they are logically in parallel.
- A computation begins with a single initial strand and terminates with a single final strand.
- Thus a multithreaded computation can be seen as a DAG of strands implanted in a tree of process instances.
- **A continuation edge** $\langle u, u' \rangle$, drawn horizontally, links a strand u to its successor u′ within the same process.
- **A spawn edge** $\langle u, v \rangle$, pointing downward, describes that a strand u spawns a strand v.
- **A call edge**, pointing downward, represents a normal function call.
- **A return edge** $\langle u, x \rangle$, pointing upward, describes that a strand u returns to its calling function and x is the strand instantly following the next sync in the calling function.

☞ Example: Calculating a Fibonacci series

A sequential recursive algorithm

A pseudocode

```
Algorithm Seq_Fib(n)
{
   if (n ≤ 1)
        return n;
   else
   {
        x := Seq_Fib (n – 1);
        y := Seq_Fib (n – 2);
      return (x + y);
   }
}
```

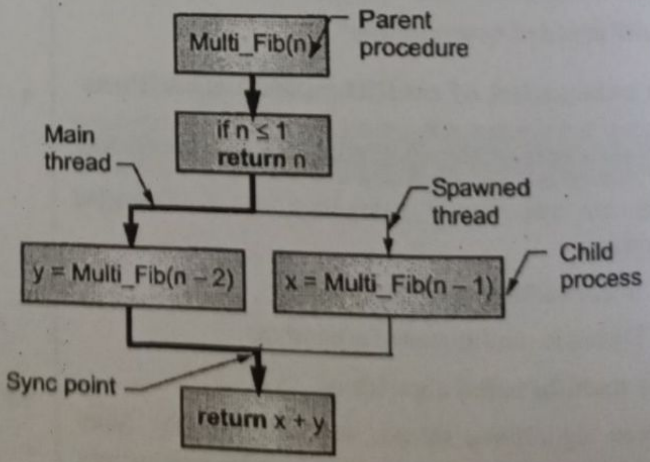


(2F1)**Fig. 8.1.1: A task dependency graph for Seq_Fib(n)**

A multithreaded parallel algorithm:

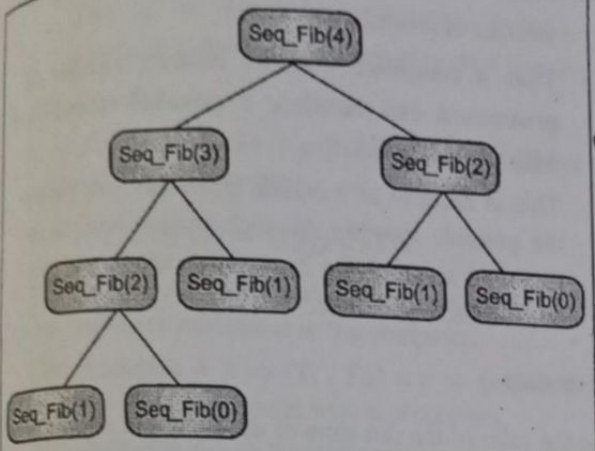
A pseudocode

```
Algorithm Multi-Fib(n)
{
   if (n ≤ 1)
        return n;
   else
   {
        x := spawn Multi-Fib(n – 1);
        y := Multi-Fib(n – 2);
        sync;
      return x + y;
   }
}
```

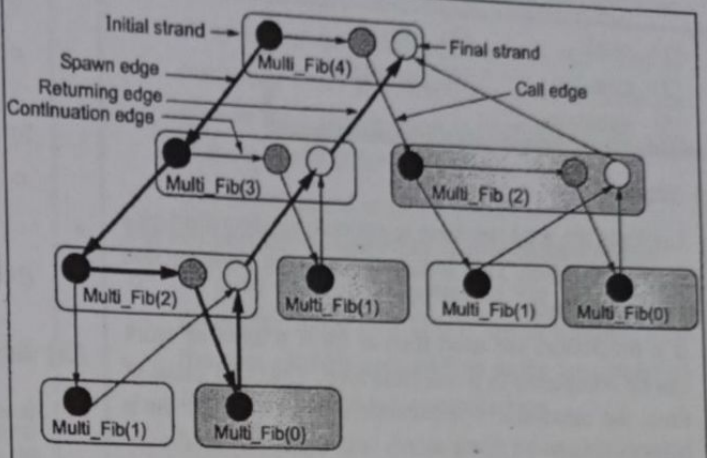


(2F2)**Fig. 8.1.2: A task dependency graph for Multi_Fib(n)**

The computing models of the sequential and parallel algorithms



(2F3)**Fig. 8.1.3: Recursion Tree of Seq_Fib(4)**



- **Circles represents strands**
- ● base case or the part of the procedure up to the spawn of Multi_Fib(n – 1) in line 4.
- ◉ part of the procedure that calls Multi_Fib(n – 2) in online 5 up to the sync in line 6, where it suspends until the spawn of Multi_Fib(n – 1) returns.
- ○ part of the procedure after the sync where it sums x and y up to the point where it returns the results.
- **Rectangles contain strands belonging to the same procedure**
- ▭ for spawned procedures
- ▭ for called procedures.

(2F4)**Fig 8.1.4: DAG model of Multi_Fib(4)**

Advantages of the multithreaded algorithms in parallel computing

- A simple serial algorithm can be extended as a multithreaded algorithm by using the keywords **parallel, spawn,** and **sync**.
- It provides a theoretical approach for enumerating parallelism by using the concepts of **work** and **span**.
- It facilitates the **nested parallelism** using divide and conquer strategy.
- It solves numerous challenging problems effectively.
- Different multithreading platforms are available:
  - OpenMP
  - Cilk
  - Task Parallel Library
  - Threading Building Blocks [Intel TBB]
- Variety of computer architectures support multithreading:
  - Supercomputers: Cray
  - Multi-core computers

8.1.1 Performance Measures

UQ. Define performance measure of multithreaded algorithms. Write a multithreaded algorithm for Fibonacci Series and explain performance measure of Fibonacci(6) execution with suitable diagram.
SPPU - Q. 7(b), May 18, 9 Marks

GQ. Explain the performance measures: work, span, speedup, parallelism and slackness. **(4/6 Marks)**

- There are different performance measures of multithreaded algorithms:

(1) work	(2) span
(3) speedup	(4) parallelism
(5) slackness	

(1) Work

- Let T_1 is the total run-time to execute an algorithm on a single processor. This indicates the **work** done by that single processor.
- If p processors are used then at most p units of work can be completed in a one-time step. Thus in T_p unit of time, the maximum computational work done by the p processors can be given as:

$$\textbf{Work} = p \cdot T_p$$

where p is the number of processors and T_p is the run-time to execute an algorithm on p processors.

- **The work law:**
  - It states that $T_p \geq T_1/p$.
  - It means the speedup of an algorithm on p processors can be no better than the run-time with a single processor divided by the number of processors p.
  - Thus **the parallelism on p processors at best offers a constant speedup where the constant is 1/p.**
- The parallelism does not change the asymptotic efficiency class of an algorithm.

(2) Span

- Let T_∞ is the total run-time to execute an algorithm on an infinite number of processors. Practically, it uses the maximum number of processors to allow the parallelism wherever it is feasible.
- T_∞ is known as the **span** since it is the longest path through a computation DAG.
- It corresponds to the maximum time required to execute the strands along any path in a computation DAG. It defines the lower bound on the run-time of an algorithm.
- E.g., For the **DAG model of Multi_Fib(4),** the span is represented by the thicker edges in Fig. 8.1.4.
- **The span law:**
  - It states that $T_p \geq T_\infty$
  - It means an ideal computer with p processors cannot run faster than a computer with an infinite number of processors.
  - Thus **a computer with an infinite number of processors can emulate a parallel computer with p processors.**
  - This is because at a certain point the span limits the possible speedup. So such strands need to be executed sequentially irrespective of the number of processors you have.

(3) Speedup

- It is the ratio of the run-time of an algorithm executed on a single processor to the run-time of that algorithm executed on p processors.

$$\therefore \textbf{Speedup} = T_1/T_p$$

where T_1 is the run-time to execute an algorithm on a single processor and T_p is the run-time to execute an algorithm on p processors.

- It describes how much faster an algorithm gets executed on p processors as compared to a single processor.
- As per the work law, we have $T_p \geq T_1/p \Rightarrow T_1 / T_p \leq p$. Thus using p processors one can achieve the speedup of at most p.
- The speedup $T_1 / T_p = \Theta(p)$ gives a **linear speedup.**
- The speedup $T_1 / T_p = p$ gives a **perfect speedup.**

(4) Parallelism

- The **parallelism** of the computation is represented by the ratio of the work to the span.

$$\therefore \textbf{Parallelism} = T_1/T_\infty$$

- It can be understood using three perspectives:
  - **Ratio:** The parallelism determines the average amount of computational work that can be concurrently done in each step.
  - **Upper Bound:** The parallelism defines the maximum speedup that can be possibly attained by executing an algorithm on any number of processors.

- **Limit:** The parallelism gives the limit on the possibility of achieving the perfect linear speedup. Thus if we use more processors beyond the parallelism, then we get a lesser perfect speedup.

(5) Slackness

- It is the ratio of the parallelism to the number of processors in the computer.

$$\therefore (T_1/T_\infty)/p = T_1/(p \cdot T_\infty)$$

- Thus it gives a factor by which the parallelism exceeds the number of processors in the computer.
- **The slackness = 1** $\Rightarrow (T_1 / T_\infty) = p \Rightarrow$ perfect linear speedup using a computer with p processors.
- **The slackness < 1** $\Rightarrow$ the impossibility of perfect linear speedup (i.e. there are more processors in a computer than one can make use of.)
- **The slackness > 1** $\Rightarrow$ the work per processor gives limiting constraint and a scheduler can attempt for the linear speedup by allocating the work across the number of processors.

8.1.2 Analyzing Multithreaded Algorithms, Parallel Loops, Race Conditions

UQ. Explain multithreaded algorithms. How to analyze multithreaded algorithms? What is a race condition, parallel loops?
SPPU - Q. 7(a), May 19, 9 Marks

GQ. Write the analysis of multithreaded algorithms with an example. **(4/6 Marks)**

GQ. Explain the concept of parallel loops. **(6/8 Marks)**

GQ. What are the race conditions in multithreaded algorithms? Explain in detail. **(4/6 Marks)**

Analyzing the work

- The work done by a multithreaded algorithm is analyzed by ignoring the parallel constructs and concentrating on the analysis of the sequential algorithm.

  E.g., The **work** $T_1(n)$ of Multi_Fib(n) is given by considering the work of Seq_Fib(n).

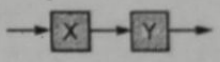
$$\therefore T_1(n) = T(n) = T(n-1) + T(n-2) + \Theta(1)$$

$$\therefore T(n) = \Theta(F_n),$$

where F_n grows exponentially in n. $F_n = \phi^n$ where ϕ = a golden ratio $= \frac{(1+\sqrt{5})}{2}$.

Analyzing the span

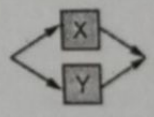
- **For the serial composition of sub-computations:**
  - If a set of sub-computations (or the vertices describing them in a DAG) are in series, then the span of their composition is obtained by the sum of the spans of those sub-computations.
  - It is like a typical sequential analysis.
- **For the parallel composition of sub-computations:**
  - If a set of sub-computations (or the vertices describing them in a DAG) are in parallel, then the span of their composition is the maximum of the spans of those sub-computations.
  - This is where the analysis of the multithreaded algorithms differs.
- The work and the span of the composition of sub-computations are depicted in Fig. 8.1.5.



Work: $T_1(X \cup Y) = T_1(X) + T_1(Y)$

Span: $T_\infty(X \cup Y) = T_\infty(X) + T_\infty(Y)$

(2F5)(a) A serial composition



Work: $T_1(X \cup Y) = T_1(X) + T_1(Y)$

Span: $T_\infty(X \cup Y) = \max(T_\infty(X), T_\infty(Y))$

(2F6)(b) A parallel composition

Fig. 8.1.5 : The work and the span of the composition of sub-computations

- E.g., Computing a Fibonacci series with input size n:
  - The **span** of Seq_Fib(n):

$$T_\infty(n) = (T_\infty(n-1) + T_\infty(n-2)) + \Theta(1)$$

  - The **span** of Multi_Fib(n):

$$T_\infty(n) = \max(T_\infty(n-1), T_\infty(n-2)) + \Theta(1)$$

$$\therefore T_\infty(n) = T_\infty(n-1) + \Theta(1) = \Theta(n).$$

- The **parallelism** of Multi_Fib(n):

$$\text{Parallelism} = T_1(n) / T_\infty(n) = \Theta(\phi^n) / \Theta(n) = \Theta(\phi^n/n).$$

This grows drastically and much faster (exponential growth) than n.

- The slackness of Multi_Fib(n):

Slackness = $\Theta(\phi n/n)/p$, where p is the number of processors.

As ϕn grows exponentially with n, for any $p > n$ there is a potential for attaining near-perfect linear speedup as n grows.

☞ Analyzing the parallel loops

- Multithreaded algorithms use a keyword **parallel** with the loop constructs to define the parallel execution of iterations.
- E.g., A matrix-vector multiplication:
  - Consider an $n \times n$ matrix $W = (w_{ij})$ is multiplied by an n-vector $X = (x_j)$ to get an n-vector $Y = (y_i)$.

  $\therefore \quad y_i = \sum_{1 \le j \le n} (w_{ij} x_j)$ for $1 \le i \le n$.
  - The algorithm MatVec_Mul() computes all values of Y in parallel.

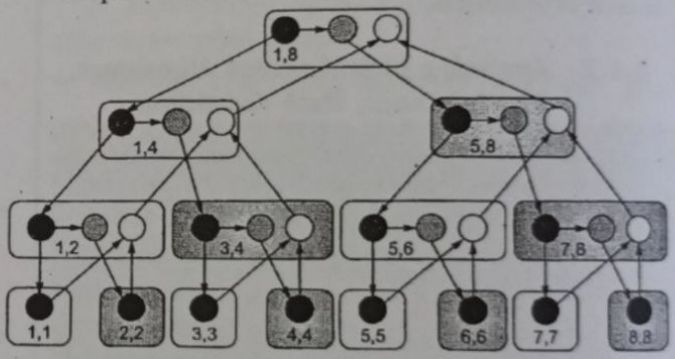
```
Algorithm MatVec_Mul(W, X)
/*Input: W[1:n,1:n] is an n × n matrix. X[1:n] is an n-vector.
Output: Y[1:n] is a resulting n-vector obtained by multiplying
W by X. */
{
  parallel for(i:=1; i≤ n; i+ +)
    Y[i]:=0;   //Initialization
  parallel for (i:=1; i≤n; i+ +)

  {
      for(j:=1; j ≤ n; j+ +)          */main for loop of
                                      computation /*
        Y[i]:= Y[i]+W[i, j]*X[ j] ;
  }
  return Y;
}
```

- All **parallel *for*** loops are implemented as the divide and conquer sub-routines by a compiler. It can be described by an algorithm MatVec_Loop()

```
Algorithm MatVec_Loop(W, X, Y, n, i, i' )
/*It describes the main for loop in an algorithm MatVec_Mul().
The spawning of parallel for loop is done by using divide and
conquer approach.*/
{
  if (i= i' )
  {
    for(j:=1; j ≤ n; j+ +)
      Y[i]:= Y[i]+W[i, j]*X[ j] ;
  }
  else
  {
    mid:= ⌊(i + i' )/2⌋;
    spawn MatVec_Loop(W, X, Y, n, i, mid);
    MatVec_Loop(W, X, Y, n, mid+1, i' );
    sync;
  }
}
```

- This is depicted in Fig. 8.1.6. Here, the values of parameters i and i′ are written within each rounded rectangle.
- The black circles symbolize the strands corresponding to either the portion of an procedure up to **spawn** MatVec_Loop(W, X, Y, n, i, mid) or the base case; the shaded circles symbolize the strands corresponding to the portion of an procedure that calls MatVec_Loop(W, X, Y, n, mid+1, i′) up to the **sync**; and the white circles symbolize the strands corresponding to the portion of the procedure after the **sync** until it returns.



(2F7)Fig. 8.1.6 : A computation DAG for MatVec_Loop(W, X, Y, 8, 1, 8)

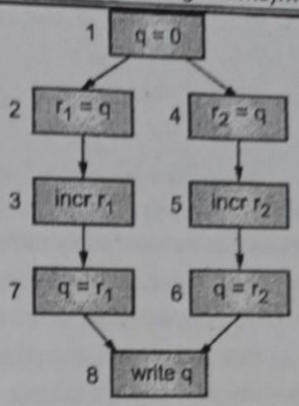
- **Analysis of MatVec_Mul():**
  - The **work** of MatVec_Mul() = $T_1(n) = \Theta(n^2)$ due to the nested *for* loops (*for* loops on i and j).
  - A **parallel *for*** is described by a recursion tree. Since this tree is a full binary tree, the n leaf nodes = the number of internal nodes + 1. All internal nodes divide an iteration range and the leaf nodes execute the iteration loop. Thus the extra work is $\Theta(n)$.
  - The extra work due to recursive spawning is amortized across the work of the iterations so that it gives only constant work.
  - The amount of recursive spawning can be reduced by executing numerous iterations in each leaf. This is achieved by the concurrency platforms.

- The **span** is increased by $\Theta(\log_2 n)$ due to the additional recursion tree of height $\Theta(\log_2 n)$ for MatVec_Loop(). But this upsurge can be ignored due to other dominating factors (E.g., The nested loops dominate the span in MatVec_Mul()).
- **Nested parallelism:** In MatVec_Mul(), the span is $\Theta(n)$ as in spite of with full parallelism, the inner for loop on j still needs $\Theta(n)$. So with the work of $\Theta(n^2)$, the parallelism is $\Theta(n^2)/\Theta(n) = \Theta(n)$.

Race conditions

- The **deterministic algorithms** produce the same result on the same input on multiple runs; while the **non-deterministic algorithms** may produce different results on the same input on different runs.
- Generally, the multithreaded algorithm produces the results non-deterministically as it includes a "determinacy race." The race condition is the curse of concurrency.
- A **determinacy race :** This happens when two logically parallel computations access the same memory and one of the computations writes on it. It causes the non-deterministic results of the computations. The determinacy races are difficult to detect using empirical testing.
- E.g., (1) The following algorithm describes the race condition. Here the algorithm produces the output 1or 2 based on the order in which the two threads interleave the access to q. It is depicted in Fig. 8.1.7. For some ordering like <1, 4, 5, 6, 2, 3, 7, 8> or <1, 2, 3, 7, 4, 5, 6, 8> the algorithm Race() produces correct result q =2.

```
Algorithm Race()
{
   q:=0;
      parallel for (i:=1; i ≤ 2; i+ +)
          q:=q+ 1;
   write q;
}
```



(2F8) (a) A computation DAG

Step	q	r_1	r_2	Remark
1	0	–	–	Initialization q: = 0;
2	0	0	–	Thread T1 on a processor1 loads q to r_1. $r_1 = q = 0$.
3	0	1	–	T1 in increments r_1. $r_1 = r_1 + 1 = 0 + 1 = 1$
4	0	1	0	Thread T2 on a processor2 loads q to r_2. $r_2 = q = 0$.
5	0	1	1	T2 increments r_2. $r_2 = r_2 + 1 = 0 + 1 = 1$
6	1	1	1	T2 stores r_2 to q $q = r_2 = 1$
7	1	1	1	T1 stores r_1 and q $q = r_1 = 1$

(b) The execution sequence

Fig. 8.1.7 : Example of the determinacy race condition in the Algorithm Race()

- E.g., (2) The algorithm for a matrix-vector multiplication with the race conditions.

```
Algorithm MatVec_Mul_Race(W, X)
/*It describes the determinacy race condition. */
{
   parallel for(i:=1; i ≤ n; i+ +)
      Y[i]:=0;   //Initialization
   parallel for (i:=1; i ≤ n; i+ +)
                          //main for loop of computation
   {
      parallel for (j:=1; j ≤ n; j+ +)
```

```
                /*Causes determinacy races in updating Y.*/
            Y[i]:= Y[i]+W[i, j]*X[ j] ;
        }
    return Y;
}
```

- The race conditions can be handled by different ways:
  - By using locks, mutual exclusion and other techniques of synchronization.
  - By ensuring that the strands executing in parallel are independent.
  - By making all the iterations in a **parallel *for*** loop independent of each other.
  - In the code portion between the **spawn** and associated **sync**, by making the spawned procedure independent of the parent procedure.

8.2 PROBLEM SOLVING USING MULTITHREADED ALGORITHMS

8.2.1 Multithreaded Matrix Multiplication

UQ. Give pseudocode for Multithreaded matrix multiplication. Analyze the same. **SPPU - Q. 8(a). May 19. 9 Marks**

GQ Write and explain the algorithm for multithreaded matrix multiplication. **(4/6 Marks)**

- We present an algorithm Multi_Mat_Mul() for multiplication of two square matrices.
- It is based on the serial matrix multiplication algorithm with running time = $T_1(n) = \Theta(n^3)$ where n is the number of elements in the given two square matrices.

```
Algorithm Multi_Mat_Mul(W, X)
/*It describes a multithreaded algorithm for multiplication of
two square matrices.
Input: W[1:n,1:n] and X[1:n,1:n] are two n × n matrices.
Output: Y[1:n,1:n] is a resultant n × n matrix obtained by
multiplying W by X. */
{
    parallel for(i:=1; i ≤ n; i+ +)
    parallel for (j:=1; j ≤ n; j+ +)
    {
        Y[i, j]:=0;                    //Initialization
        for (k:=1; k ≤ n; k+ +)
        {
            Y[i, j]:= Y[i, j]+W[i, k]*X[k, j] ;
        }
    }
    return Y;
}
```

- The **work** of the algorithm Multi_Mat_Mul() is obtained by considering a serial algorithm. So $T_1(n)= \Theta(n^3)$.
- The **span** of the algorithm Multi_Mat_Mul() is determined by considering the path for spawning the outer and inner parallel *for* loops on i and j respectively and then the n iterations of the innermost *for* loop on k. So, $T_\infty(n) = \Theta(n)$.
- The **parallelism** of the algorithm Multi_Mat_Mul() is given by $T_1(n) / T_\infty(n) = \Theta(n^3)/\Theta(n) = \Theta(n^2)$.
- The Strassen's matrix multiplication algorithm based on divide and conquer strategy has run-time $< \Theta(n^3)$. It can be also implemented using a multithreaded algorithm.

8.2.2 Multithreaded Merge Sort

UQ. Write and Explain Multithreaded Merge Sort Algorithm. **SPPU - Q. 7(a). May 18. 9 Marks**

UQ. Write and explain multithreaded merge sort algorithm. Analyze the same. **SPPU - Q. 8(b). Dec. 19. 9 Marks**

- Divide and conquer algorithms break the original larger problem into independent smaller sub-problems that can be solved separately. Hence, these types of algorithms are suitable for parallelism.
- We consider an example of merge sort and develop a multithreaded algorithm of merge sort.
- It is based on the serial merge sort algorithm with running time = $T_1(n) = \Theta(n \log_2 n)$ where n is the number of elements to be sorted. It already follows the divide and conquer strategy.

Parallelizing Merge_Sort()

- We divide the main procedure Merge_Sort() and parallelize it by spawning the first recursive call. It is described in the algorithm Multi_Merge_Sort1().

```
Algorithm Multi_Merge_Sort1(X, first, last)
/*It describes a multithreaded algorithm for merge sort that
applies multithreading only to merge sort but not to merging of
results.
Input: X[1: n] is an array of n elements to be sorted. first and
last represent the lower and the upper index respectively of the
sub-array X[first: last].
```

```
Output: A sorted array of n elements. */
{
 if (first < last)
 {
    mid:= ⌊(first + last)/2⌋;
    spawn Multi_Merge_Sort1(X, first, mid);
    Multi_Merge_Sort1(X, mid+1, last);
    sync;
 }
Merge_Results(X, first, mid, last);
                    /* Merges the two sorted sub-arrays
                       using a sequential algorithm.*/.
}
```

- Since an algorithm Merge_Results() remains a sequential algorithm, its work and span are $\Theta(n)$ where n is the input size.
- The **work** of an algorithm Multi_Merge_Sort1() is computed by constructing a recurrence formula same as the sequential algorithm.

$\therefore T_1(n) = 2\,T_1(n/2) + \Theta(n) = \Theta(n \log_2 n)$

- The recurrence formula for the **span** $T_\infty(n)$ of Multi_Merge_Sort1() includes only a single term of (n/2) as the recursive calls to Multi_Merge_Sort1() with size (n/2) run in parallel.

$\therefore T_\infty(n) = T_\infty(n/2) + \Theta(n) = \Theta(n)$

- Thus the **parallelism** is $T_1(n)/T_\infty(n) = \Theta(n \log_2 n) / \Theta(n) = \Theta(\log_2 n)$.
- The parallelism of an algorithm Multi_Merge_Sort1() is low ($\Theta(\log_2 n)$). That means even for a larger input size n, we would not get an advantage from having hundreds of processors.

☞ Parallelizing Merge_Results()

- An algorithm Merge_Results() merges two sorted sub-arrays to construct a single sorted array of input size n.
- It can be implemented using multithreading to attain the parallelism. It uses the divide and conquer strategy for the same.
- The two input sub-arrays to an algorithm Merge_Results() are sorted. These sub-arrays are split into four sub-arrays, two of which form the head and the other two form the tail of the final array after merging.

The basic steps to find the four sub-arrays:

(1) Select the longer sub-array to be the sub-array $T[first_1: last_1]$ in Fig. 8.2.1.

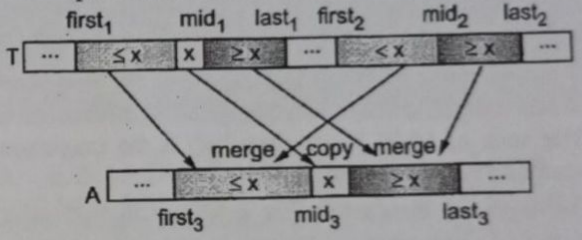
(2) Determine the middle element (median) of the first sub-array (x at mid_1).

(3). Apply binary search to get the position (mid_2) of this element if it were to be added into the second sub-array $T[first_2: last_2]$.

(4) Recursively merge

- The first sub-array up to just before the median $T[first_1: mid_1-1]$ and the second sub-array up to the insertion point $T[first_2: mid_2-1]$.
- The first sub-array from just after the median $T[mid_1+1: last_1]$ and the second sub-array after the insertion point $T[mid_2: last_2]$.

(5) Combine the results with the median element placed between them, as shown in Fig. 8.2.1.



(2F9) **Fig. 8.2.1 : Parallel Merge_Results()**

- The algorithm Multi_Merge_Results() describes the multithreaded merging of two sorted sub-arrays. It uses an algorithm Binary_Search() with the worst-case running time of $\Theta(\log_2 n)$.

```
Algorithm Multi_Merge_Results(T, first1, last1, first2, last2, A, first3)
/*Input: T is an array [1: n] that contains two sorted sub-arrays T[first1: last1] and T[first2: last2]. n is the input size. A[1: n ] is an auxiliary array whose starting index is first3.
Output: An array A[1: n ] that stores the resultant final sorted array after merging two sub-arrays.*/
{
 n1:= last1 – first1 +1;   /* size of the first sorted sub-array.*/
 n2 := last2 – first2 +1; /* size of the second sorted sub-array*/
   if (n1 < n2)                    /* Ensure that n1 ≥ n2 */
   {  swap first1 and first2;
      swap last1  and last2;
      swap n1  and  n2;
```

```
    }
    if (n1 = 0)          /* if both sub-arrays are empty */
        return;
    else
    {
        mid1 := ⌊(first1 + last1)/2⌋;
        mid2 := Binary_Search(T[first2: last2], T[mid1]);
                          /* Apply binary search to find an
                             element T[mid1] in an array
                             T[first2: last2]. */
        mid3 := first3 + (last1 - first1) + (last2 - first2);
        A[mid3] := T[mid1];
        spawn Multi_Merge_Results(T, first1, mid1 - 1, first2,
        mid2 - 1, A, first3);
        Multi_Merge_Results(T, mid1 + 1, last1, mid2 + 1, last2,
        A, mid3 + 1);
        sync;
    }
}
```

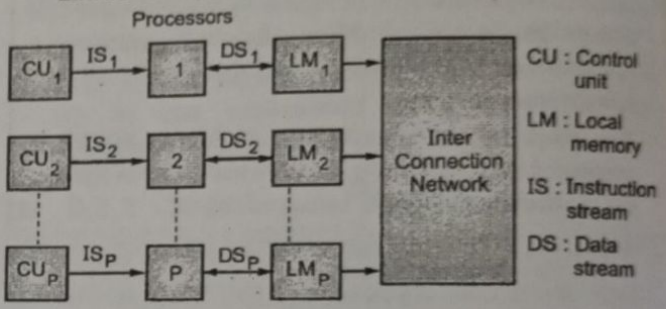
- The span of Multi_Merge_Results() is the maximum span of a parallel recursive call.
- Although we divide the first sub-array in half (x at mid_1), it results that x's insertion point mid_2 is at the start or end of the second sub-array.
- So (informally), we can claim that the maximum recursive span of Multi_Merge_Results() is 3n/4 (as at best we have "cut off" ¼ elements of the first sub-array).
- The recurrence formula for the **span** $T_\infty(n)$ of Multi_Merge_Results() is given as:
  $T_\infty(n) = T_\infty(3n/4) + \Theta(\log_2 n) = \Theta(\log_2 n)^2$
- The recurrence formula for the **work** of Multi_Merge_Results() with $1/4 \le \alpha \le 3/4$ for the unknown splitting of the second sub-array is given as:
  $T_1(n) = T_1(\alpha n) + T_1((1-\alpha)n) + O(\log_2 n)$
- With some extra work $T_1(n) = \Theta(n)$. So the **parallelism** of Multi_Merge_Results() is $\Theta(n/(\log_2 n)^2)$.
- When the final multithreaded merge sort is implemented by using both the algorithms Multi_Merge_Sort1() and Multi_Merge_Results() the **work** for the completely multithreaded merge sort is given as $T_1(n \log_2 n) = \Theta(n)$, and its **span** is given as $T_\infty(n) = \Theta(\log_2 n)^3$.
- The **parallelism** of completely multithreaded merge sort is $T_1(n \log_2 n)/\Theta(\log_2 n)^3 = \Theta(n/(\log_2 n)^2)$, which is much better than $\Theta(\log_2 n)$ of Multi_Merge_Sort1().

8.3 DISTRIBUTED ALGORITHMS

GQ. What is the distributed algorithm? (4 Marks)

Distributed computing environment

- It consists of the autonomous computers (with independent processors and memory) connected by an interconnection network. It is depicted in Fig. 8.3.1.



(1H19) **Fig. 8.3.1: Distributed Computing Environment**

Distributed algorithms

- A distributed algorithm is a type of parallel algorithms that facilitates concurrent execution in the distributed computing environment.
- In distributed algorithms, separate portions of the algorithm are simultaneously executed on independent processors with distributed memory. These algorithms have more independent computations.

Different Transparencies

- Though the portions of a distributed algorithm are independently executed on different processors at different locations, it should represent as if it is executed on a single processor.
- A distributed algorithm should provide the following transparencies:
  (1) **Location transparency**: It hides the location of computing resources.
  (2) **Access transparency**: It hides a variety of data representations and access methods.
  (3) **Migration transparency**: It hides the movement of resources in a distributed system.
  (4) **Relocation transparency**: It hides the relocation of resources while they are accessed.

(5) **Replication transparency**: It hides the existence of multiple copies of the resources.

(6) **Concurrency transparency**: It hides the sharing of resources.

(7) **Failure transparency**: It hides the failure of resources and their recovery.

(8) **Persistence transparency**: It hides whether the resource is on a disk or the main memory.

Uncertainties of distributed algorithms

- These algorithms are characterized by higher uncertainty as they have limited information about the distributed computing environment.
- Some of the uncertainties are as below:
  - unknown number of processors,
  - unknown network topology,
  - unknown message ordering,
  - different inputs and different speeds of multiple programs executing in parallel at different locations,
  - unpredictable failures of processors and links
- Designing, implementation and analysis of distributed algorithms are very complex.
- The major challenging task is the coordination of different portions of the algorithm executing on independent processors.
- It becomes more challenging in case of the failures of some processors and/or links in the distributed system.
- The selection of an appropriate algorithm to solve a given problem in the distributed environment depends on both the nature of the problem and characteristics of the distributed computing system in which the algorithm is to be executed.

Types of the distributed algorithms

(1) **Synchronous distributed algorithm**: It refers to the synchronous network model of the distributed system.

(2) **Asynchronous distributed algorithm**: It refers to the asynchronous network model of the distributed system.

Complexity analysis of the distributed algorithms

It is based on four performance measures:

(1) **Computational Rounds:** It describes the time needed by the synchronous algorithms by counting the number of computational rounds until termination.

(2) **Space:** It describes a global bound on the total memory required by all computers in the algorithm or a local bound on the memory required by per computer.

(3) **Local Running Time:** Computing the global running time of a distributed algorithm is very complex. Hence the local running time for computation on a particular computer in the distributed system is analyzed.

(4) **Message Complexity:** It is used to analyze either an asynchronous or asynchronous message-passing algorithm in the distributed system. It describes the maximum of the total number of messages transmitted between each pair of computers during the execution of a distributed algorithm.

Some of the classic applications of the distributed algorithms:

(1) Distributed operating systems
(2) Distributed database systems
(3) Network and communication systems
(4) Shared memory multiprocessor systems
(5) Digital circuits
(6) Real-time process control systems
(7) Distributed information processing
(8) E-commerce

8.3.1 Distributed Breadth First Search

UQ. What is Distributed algorithm? Explain distributed Breadth First Search algorithm with an example.
SPPU - Q. 8(a), May 18, Q. 7(a), Dec. 2019, 9 Marks

GQ. Explain distributed breadth first search algorithm for 1D partitioning. (6 Marks)

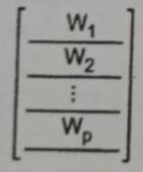
GQ. Explain distributed breadth first search algorithm for 2D partitioning. (6 Marks)

- A breadth-first search (BFS) algorithm is applied to traverse or search a graph or tree data structures. It is a level-wise search that follows the first-in-first-out order.
- It begins at some random (or some specified) node of a graph (or the tree root) and explores all of the adjacent vertices at the present level before reaching the vertices at the next level.
- Breadth-first search has many applications in a relational data graph, flow network, mesh numbering, serialization or deserialization of a binary tree, bipartiteness test of a graph etc.
- A distributed breadth first search algorithm performs BFS over a distributed network.
- It requires the efficient and scalable solution to BFS problem in a distributed environment.

- For the same, a distributed breadth first search algorithm is applied to a partitioned large-scale graph over a distributed network. There are two such solutions based on different partitioning of a graph.
  (1) A distributed breadth first search algorithm for 1-D partitioning.
  (2) A distributed breadth first search algorithm for 2-D partitioning

8.3.1(A) A distributed Breadth First search Algorithm For 1-D Partitioning

- In 1-D partitioning, a graph is partitioned such that each node and the edges originating from it are possessed by one processor.
- The set of nodes associated with a processor is known as **"local nodes"** of that processor.
- Fig. 8.3.2 depicts a p-way 1-D partitioning of a graph where p is the number of processors used.
- Here, a graph is represented using an adjacency matrix, say, W which is symmetrically reordered so that the local nodes of vertices of a processor are contiguous.
- The subscripts specify the index of the processor maintaining that data.
- The edges originating from node v form its edge list and it is the list of node indices in row v of the adjacency matrix W.
- The balanced partitioning is done by allotting approximately the same number of nodes and edges to each processor.



(2F10)**Fig. 8.3.2: A p-way 1-D partitioning of a graph**

The general procedure

- Let F is a set of nodes at the current level. Such nodes are known as **"frontier nodes"**.
- At each level, each processor in a distributed system has a set F of frontier nodes.
- Let N is a set of neighbors obtained by merging the edges of the nodes in F. The nodes in N will be possessed by the same or different processors.
- For the neighbors associated with different processors, the messages are sent to those processors to add the nodes $\in$ N to their sets F for the next level.
- Each processor merges the received sets of neighbors to form a set $\bar{N}$ of nodes owned by it.
- If a processor receives some overlapping neighbors (already present in its $\bar{N}$) then it ignores those messages.

Distributed algorithm

```
Algorithm BFS_1D (G, P, v_o)
/* Input : G = (V, E) is given graph of n nodes. V is a set of
nodes and E is a set of edges in G. P is the set of available
processors in a distributed system. v_o is a starting node in G
from which BFS of G begins.
Output : Distributed breadth first expansion starting from node
v_o of G with 1-D partintioning.*/
{
  /* Initialization of an array L [1 : n] that stores a level of
each node v ∈ V from a starting node v ∈ V. */
  for (i := 1; i ≤ n; i ++)
  {
    if (i = v_o)
      L [i] := 0;
    else L[i] := ∞;
  }
    for (j := 0; j ≤ ∞; j + +)
    {
      F := { v | L[v] = j, v ∈ V};
                    /* A set of local nodes with level j */
      if (F = Φ for all processors q ∈ P)
        break;
      N := {neighbors of nodes v_1 ∈ F (not necessarily
      local)};
      for (all processors q ∈ P)
      {
        N_q := { nodes v_2 ∈ N owned by processor q };
        Send N_q to processor q;
        Receive N̄_q from processor q;
      }
      N̄ := ∪_q N̄_q ;          /* N̄_q may overlap. */
      for (v_3 ∈ N̄ && L [v_3] = ∞)
      {
        L[v_3] := j + 1;
      }
```

```
    } // end for j
}
```

8.3.1(B) A Distributed Breadth First Search Algorithm for 2-D Partitioning

- In 2-D partitioning a graph is partitioned such that each edge is possessed by one processor. Similarly, each node is possessed by one processor.
- A processor stores the edges originating either from its local nodes or from some other nodes.
- Fig. 8.3.3 depicts a p-way 2-D partitioning of a graph where p is the number of processors used.
- Here a graph is represented using an adjacency matrix, say, W which is symmetrically reordered so that the local nodes of vertices of a processor are contiguous.
- The number of processors p = RC are arranged in an RxC mesh.
- 2-D partitioning divides the matrix W into RC block rows and C block columns.
- $W^{(*)}_{i,j}$ describes a block possessed by a processor (i, j). Each processor has C blocks.
- The nodes are partitioned such that a processor (i, j) owns the nodes of the block row $(j - 1)R + i$.
- The balanced partitioning is done by allotting approximately the same number of nodes and edges to each processor.
- The 1-D partitioning is equivalent to the 2-D partitioning with R = 1 or C = 1.

$W^{(1)}_{1,1}$	$W^{(1)}_{1,2}$	...	$W^{(1)}_{1,C}$
$W^{(1)}_{2,1}$	$W^{(1)}_{2,2}$	...	$W^{(1)}_{2,C}$
⋮	⋮	⋱	⋮
$W^{(1)}_{R,1}$	$W^{(1)}_{R,2}$	...	$W^{(1)}_{R,C}$
⋮			
⋮			
⋮			
$W^{(C)}_{1,1}$	$W^{(C)}_{1,2}$	...	$W^{(C)}_{1,C}$
$W^{(C)}_{2,1}$	$W^{(C)}_{2,2}$	...	$W^{(C)}_{2,C}$
⋮	⋮	⋱	⋮
$W^{(C)}_{R,1}$	$W^{(C)}_{R,2}$	...	$W^{(C)}_{R,C}$

(2F11)Fig. 8.3.3 : A p-way 2-D partitioning of a graph

The general procedure

- Let the edge list for a given node is a column of the adjacency matrix W. Thus each block includes partial edge lists.
- At each level, each processor in a distributed system has a set F of frontier nodes. Let a vertex $v \in F$.
- The owner processor of v sends messages to other processors in its processor-column telling them that v is a frontier node, as any of these processors may have partial edge lists for v.
- Each processor merges its partial edge lists to form the set N, which has the nodes on the next frontier (level).
- The nodes in N are then sent to their owner processors to add these nodes to their sets F for the next level.
- In 2-D partitioning, these owner processors belong to the same processor row.
- 2-D partitioning is more effective than 1-D partitioning as the processor-column and processor-row communications in 2-D partitioning involve only R and C processors, respectively; however, with 1-D partitioning, all p processors are involved in the communication.

Distributed algorithm

```
Algorithm BFS-2D (G, P, v0)
/* Input : G = (V, E) is a given graph of n nodes. V is a set of
nodes and E is a set of edges in G. P is the set of available
processors in a distributed system. v0 is a starting node in G
from which BFS of G begins.
Output : Distributed breadth-first expansion starting from node
v0 of G with 2-D partitioning.
{
  /* Initialization of an array L [1 : n] that stores a level
  of each node v ∈ V from a starting node v0 ∈ V. */
  for (i := 1; i ≤ n; i + +)
  {
    if (i = v0)
      L [i] := 0;
    else L[i] := ∞;
  }
  for (j := 0; j ≤ ∞; j + +)
  {

    F := { v | L [v] = j, v ∈ V};
/* A set of local nodes with level j. */
```

```
if (F = ϕ for all processors in P)
    break;

/* Traverse nodes by sending message to the
   selected processor */
for (all processors q in this selected processor-
   column)
{
   Send F to processor q;
   Receive F̄_q from processor q;
}
F̄ := Union (F̄_q);    /* Function Union( )
        combines the received messages F̄_q for
        all q to form the frontier level. Sets F̄_q
        are disjoint. */

N := {neighbors of nodes v_1 ∈ F̄ using edge lists on
   this selected processor};

for (all processors q in this selected processor-
   row)
{
   N_q := { nodes v_2 ∈ N owned by processor q};
   Send N_q to processor q;
   Receive N̄_q from processor q;
}      /* All-to-all communication by sending
        message to their owner processor. */

N̄ := Union (N̄_q);    /* Function Union( )
        combines the received messages N̄_q to
        form the next vertex frontier. */
for (v_3 ∈ N̄ && L [v_3] = ∞)
{
   L[v_3] := j + 1;    /* Updates the level for
                the next vertex frontier. */
}
} //end for j
}
```

8.3.2 Distributed Minimum Spanning Tree

UQ. What is distributed algorithm? Explain Distributed Minimum Spanning Tree.

SPPU - Q. 8(b), May 19, 9 Marks

- **A minimum spanning tree (MST)** of a connected weighted graph G = (V, E) is a spanning tree T = (V, E') with the smallest cost where the cost of a tree is expressed as the sum of the costs of all its edges.
- **The minimum spanning tree problem** is to find a minimum-cost spanning tree of a given graph. It describes the cheapest way to connect all the vertices in a graph.

Asynchronous distributed algorithm

- It begins with the individual nodes as the connected components at level 0. Thus inductively we can get level i components each having own spanning tree and a known leader within it.
- To obtain the level i + 1 connected components each level i component searches along its spanning tree for its minimum weighted outgoing edge.
- The leader within a connected component broadcasts search requests along the tree edges.
- Accordingly, each node determines its own minimum weighted outgoing edge by broadcasting along all non-tree edges to check whether the other end is in the same connected component or not.
- This is checked by checking the leader's identifier used at all the nodes of a component as the component's identifier.
- Then the nodes send their responses toward the leader by selecting the minimums along the path.
- When all connected components determine their minimum weighted outgoing edges, these components are merged by connecting the minimum weighted outgoing edges.
- The leader sends a message to the source of the minimum weighted outgoing edge telling it to mark the edge.
- Finally, for each new connected component, a new leader is elected.
- The time to find the component for a level takes the worst-case time of $\Theta(n)$ where n = |V| = the number of nodes. As the number of levels is O(log n), the total **time complexity** is O(n log n).

- The **message complexity** is O((n + m) log n) as in each level O(n) messages are transmitted along all the tree edges and O(m) additional messages are needed to search for the locally minimum weighted outgoing edges, where m = |E| = the number of edges.

Synchronous distributed algorithm

- It involves the identification of all connected components and the minimum weighted edge joining the connected component.
- Each round of a distributed algorithm for MST follows the steps below:
  - To identify the connected components the algorithm performs Leader-Election computation using an identifier of each node. Thus, the smallest identifier of each node identifies each connected component.
  - Each node v∈ V computes the minimum weighted edge <u, v > ∈ E and u∈ V incident on v such that its endpoints u and v belong to different connected components. If no such edge is present, then the algorithm uses a dummy edge with infinite weight.
  - Then, the algorithm again performs a Leader-Election computation using the edges linked with each node and their weights.
  - For each connected component C, the second Leader-Election computation finds the minimum-weighted edge connecting it to another connected component.
- The last round of the algorithm determines the weight of the minimum-weighted edge to be infinity.
- There are total O(log n) rounds where n = |V| = the number of nodes. At each round O(m) constant-size messages are sent where m = |E| = the number of edges. So, the total **message complexity** is O(m log n).

8.4 STRING MATCHING ALGORITHMS

UQ. What are string matching algorithms? Explain any one algorithm with an example.
SPPU - Q. 7(b), Dec 19, 9 Marks

UQ. Give and explain the string matching algorithm.
SPPU - Q. 8(b), May 17, 9 Marks

UQ Compare and contrast String matching algorithms. Explain any one algorithm with an example.
SPPU - Q. 8(b), May 18, 9 Marks

String

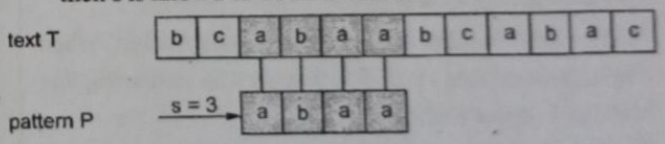
- Σ* describes the set of all finite-length strings produced by using characters from the finite set of alphabets Σ.
- The empty (zero-length) string symbolized by ε∈Σ*.
  |T| = the length of a string T.
- TR represents the concatenation of two strings T and R and consists of the characters in T followed by the characters in R. |TR| = |T| + |R|.

The string matching problem

- Let an array T[1: n] represents the text of length n and an array P[1: m] represents the pattern of length m, where m << n.
- Also, assume that the elements of P and T are characters in the finite set of alphabets ∑.
- Then the string matching problem is to find the pattern P in the string T.
  E.g. ∑ = {a, b, c} and we want to find P = 'abaa' in T = 'bcabaabcabac'). Refer to Fig. 8.4.1 for the same example.

Shift

- If a pattern P occurs with shift s in a string T, then s is known to be a **valid shift**.
- If a pattern P does not occur with shift s in a string T, then s is known to be an **invalid shift.**



(2F12)**Fig. 8.4.1: An example of a string matching problem**

Prefix

- String Q is a prefix of a string T if T = QR for some string R ∈ Σ*.
- It is denoted by Q⊏T.
- When a string Q is a prefix of a string T then there exists some string R that when added onto the end of a string Q makes Q = T.

Suffix

- String Q is a suffix of a string T if T = RQ for some string R∈ Σ*.

- It is denoted by Q]T.
- When a string Q is a suffix of a string T then there exists some string R that when added onto the front of a string Q makes Q = T.

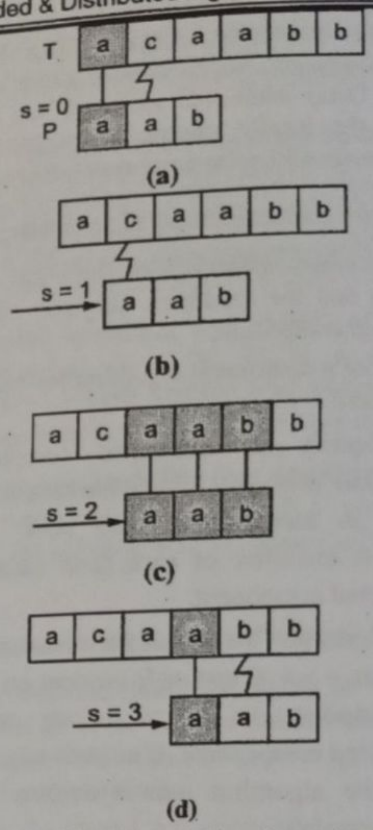
Some applications of the string matching algorithms

(1) Computational biology
(2) Molecular chemistry
(3) Finding a particular DNA sequence
(4) Searching patterns in the documents formed using a large set of alphabets
(5) Word processing
(6) Web searching
(7) Desktop search (Google, MSN)
(8) Matching byte-strings containing graphical data or machine code
(9) grep command in Unix

8.4.1 The Naïve String Matching Algorithm

- This algorithm follows the brute force approach.
- Let an array T[1: n] is the given base string of length n and an array P[1: m] is the pattern of length m that is to be searched for in T.
- The algorithm slides the pattern P across the base string T looking for a match.
- Let s is the number of times P has been shifted. When the algorithm finds P in T it outputs s by indicating that "Pattern occurs with shift s".
- This algorithm does not give an optimal solution since it ignores the information gained about the base string T for one value of s while considering the other values of s.

E.g., Consider a string matching problem instance: T = acaabb and P = aab. The working of the naïve string matching algorithm is depicted in Fig. 8.4.2. Here a pattern P slides over a string T as shown in (a)-(d). The vertical straight line connects the matching character (shaded block) and a jagged line connects the first mismatching character if any. In this example, the algorithm finds the matching pattern with shift s = 2 as shown in (c).



(2F13) **Fig. 8.4.2: Working of the naïve string matching algorithm for T = acaabb and P = aab.**

Algorithm

```
Algorithm Naïve_Str_Match (T, P)
/* Input : An array T [1 : n] is a given text string and an array
P [1 : m] is a pattern to be searched in T.
Output : The number of a shift s with which a pattern P occurs
in T. */
{
   n := Get_Length (T);
   m := Get_Length (P);
                    /* Get_Length (T) returns the length
                    of a string T. Same for the
                    Get_Length (P). */
   for (s := 0; s ≤ n – m; s + +)
   {
      if (P [1: m] = T [s + 1 : s + m])
      write ("pattern occurs with shift", s);
   }
}
```

Complexity analysis

- The time complexity of the naïve string matching algorithm is O((n – m + 1) m), where n = |T| = the length of a string being searched and m |P| = the length of a pattern (substring) being compared.

8.4.2 The Rabin-Karp Algorithm

UQ. Write and explain the Rabin-Karp algorithm. Explain the worst case and best case running time of Rabin Karp Algorithm?

SPPU - Q. 7(b), May 19, Q. 8(a), Dec 19, 9 Marks

GQ. Write a detailed note on the Rabin-Karp String matching algorithm and discuss its complexity.

(6/8 Marks)

- This algorithm refers to the concept of the equivalence of two numbers modulo a third number.
- Instead of comparing two strings, it compares their hash values. For efficiency, the algorithm easily calculates the hash value of the next position in the string from the hash value of the current position.
- Generally, this algorithm considers each character in a string as a digit in radix-D notation, where D = |Σ|=10, Σ = {0,1,...,9}.
- Thus a string of k consecutive characters is considered as a length-k decimal number. E.g. A character string 41325 thus corresponds to the decimal number 41,325.

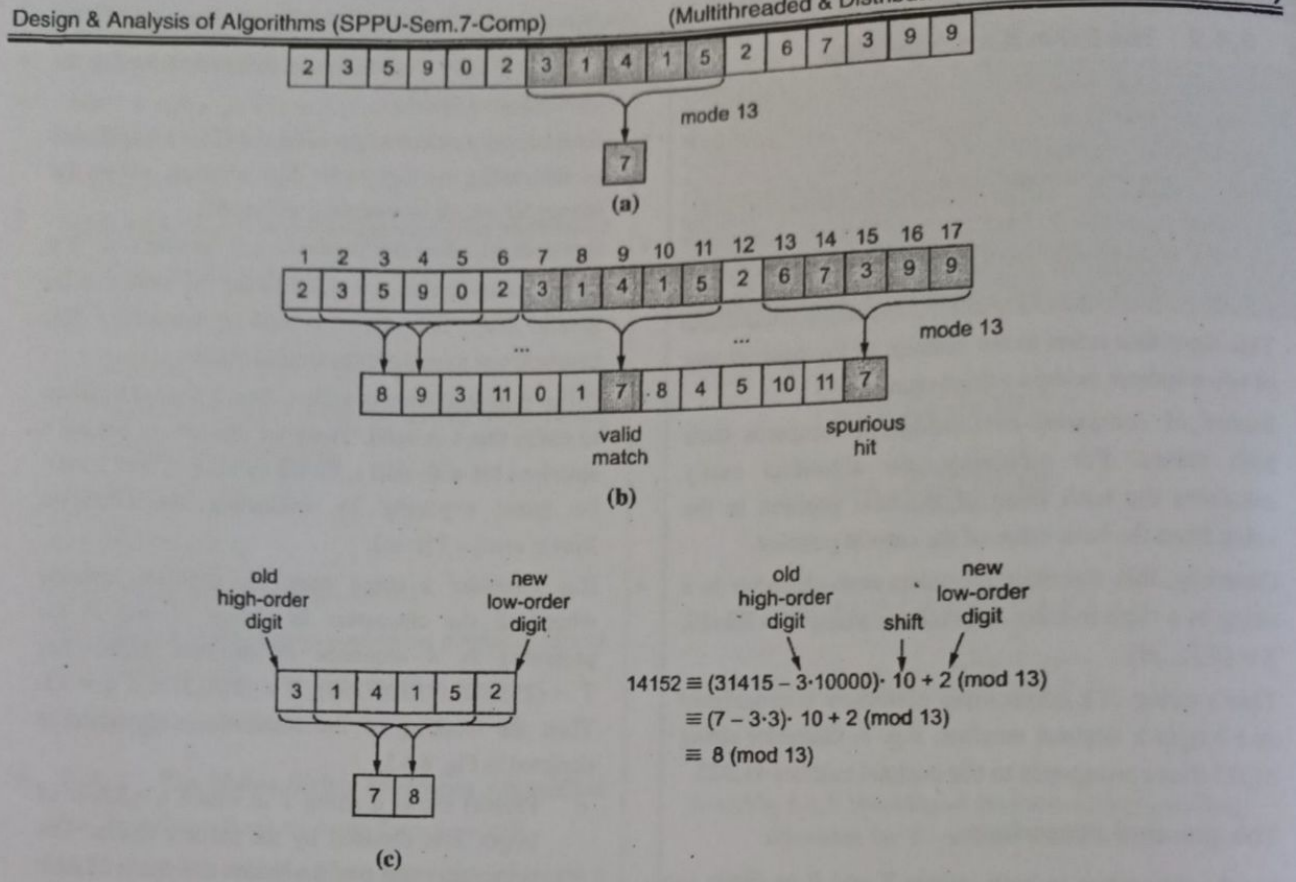
The general procedure

- Let the characters in both strings T and P be digits in radix-D notation, where D = |Σ| = 10, Σ = {0,1,...,9}. Let n = |T| and m = |P|.
- Let p be the decimal value of the characters in an m-length string P. p is computed in time O(m) by using Horner's rule:

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

- Let t_s be the decimal value of the characters in an m-length sub-string T[s+1: s+m], $0 \le s \le n-m$. t_s is also computed in time O(m) by using Horner's rule.

  $t_s = p$ iff a sub-string T[s+1: s+m]= P[1:m], thus, s is a valid shift iff $t_s = p$.
- Select any prime number q so that it fits in a memory word to speed the computations.
- Calculate (p mod q). This value of p mod q is used to find all matches of P in T.
- Calculate (T[s+1: s+m] mod q) , $0 \le s \le n-m$.
- Compare P only with those sub-strings in T having the same value of (mod q).
- Incrementally calculate the values of (T[s+1: s+m] mod q) subtracting the high-order digit, shifting, adding the low-order bit, all in modulo q arithmetic.
- ($t_s \equiv p \bmod q$) does not imply ($t_s = p$); however, ($t_s \not\equiv p \bmod q$) assures that $t_s \neq p$ indicating the shift s as an invalid shift. Thus ($t_s \equiv p \bmod q$) presents a fast heuristic test to rule out the invalid shifts s.
- If ($t_s \equiv p \bmod q$) for any shift s, then it does not suffice to claim that s is valid. There are chances of having a **spurious hit** with shift s. So the validity of shift s must be tested explicitly by evaluating the condition T[s+1: s+m] = P[1: m].
- E.g. Consider a string matching problem instance where all the characters in strings T and P are presented as a sequence of decimal digits. Let T = (23590231415267399), P = (31415) and q = 13. Then the working of the Rabin-Karp algorithm is depicted in Fig. 8.4.3.
  - **Part(a)** shows a string T in which a window of length 5 is depicted by the shaded blocks. The numerical value p of the shaded number is 31,415. So p mod q = 31415 mod 13 = 7.
  - **Part(b)** shows a string T with the calculated values of modulo 13 for each possible position of a window of length 5. As 31415 ≡ 7 mod 13, the algorithm searches the windows of length 5 whose decimal value modulo 13 is 7. There are two such windows depicted by shaded blocks. The first shaded window is an occurrence of the pattern P, while the second shaded window is a spurious hit.
  - **Part(c)** shows the incremental computation of the decimal value of a window in constant time by using the decimal value of the previous window. The decimal value of the first window of length 5 = 31415. So, 31415 mod 13 = 7. Subtracting the high-order digit 3, shifting left (multiplying by 10), and then adding in the low-order digit 2 produces the decimal value 14152 of the next window of length 5. Thus, 14152 mod 13 = 8.



(2F14)**Fig. 8.4.3: Working of the Rabin-Karp string matching algorithm**

☞ Algorithm

```
Algorithm RabinKarp_Str_Match (T, P, D, q)
/* Input: An array T[1 : n] is a given string and an array
P[1 : m] is a pattern to be searched in T. D is a radix used to
represent the characters of both P and T as digits. q is a
selected prime number used in modulo operations.
Output : The number of a shift s with which a pattern P occurs
in T. */
{
  n := Get_Length (T);
  m := Get_Length (P);
                    /* Get_Length (T) returns the
                    length of a string T. Same for the
                    Get_Length (P).*/
  r := (D^(m-1) % q);
  p := 0;
  t0 := 0;
/* Preprocessing :Computing the decimal values of P and T. */
  for (i := 1; i ≤ m; i + +)
  {
    p := (Dp + P[i]) % q;
    t0 := (Dt0 + T[i]) % q;
  }
/* String matching */
  for (s := 0; s ≤ n – m; s + +)
  {
    if (p = ts)
    {
      if (T [s + 1 : s + m] = P [1 : m])
      {
        write ("pattern occurs with shift", s);
      }
    }
    if (s < n – m)
    {
      ts + 1 = (D (ts – T [s + 1] r) + T [s + m + 1])% q;
    }
  } end for s
}
```

Complexity analysis

- The Rabin-Karp algorithm uses $\Theta(m)$ preprocessing time to represent the characters of string T and P in radix-D notation, where $D = |\Sigma| = 10$, $\Sigma = \{0,1,...,9\}$, $n=|T|$ and $m=|P|$.
- If $T = a^n$ and $P = a^m$ then each shift is a valid shift. It represents the worst-case of the Rabin-Karp algorithm with the matching time of $O((n-m+1)m)$. It is the same as that of the naïve string matching algorithm.
- The average-case of the Rabin-Karp algorithm may perform few valid shifts, say, some constant c and some spurious hits that are bounded by $O(n/q)$. Then its matching time can be given as $O((n-m+1) +cm+ mn/q)$.
- If the number of valid shifts is small ($O(1)$) and if the prime q is selected such that $q \geq m$ then the average case of the Rabin-Karp algorithm needs a matching time of $O(n + m)$. As $m \leq n$, the algorithm takes the matching time of $O(n)$ which is better than that of the naïve string matching algorithm.

Summary

- The algorithms that permit the concurrent execution of multiple threads of a program on multiple cores in a system with shared memory are known as **"multithreaded algorithms"**.
- A set of runtime instructions executed by a processor using a multithreaded program is known as a **"multithreaded computation"**.
- There are two broad categories of multithreaded algorithms:
  - **Static multithreaded algorithms:** In these algorithms, threads are alive for the entire period of computation.
  - **Dynamic multithreaded algorithms:** These algorithms provide a concurrency platform to synchronize, schedule and handle the parallel-computing resources.
- **A concurrency platform** for dynamic multithreaded algorithms provides:
  (1) **A scheduler** to facilitate automatic load-balancing.
  (2) **Nested parallelism** to permit a subroutine to be spawned.
  (3) **Parallel loops** to execute the iterations of the loop concurrently.
- A simple serial algorithm can be extended as a multithreaded algorithm by using the keywords **parallel, spawn,** and **sync.**
- A multithreaded algorithm is modeled as a computation DAG.
- Different performance measures of multithreaded algorithms are: **work, span, speedup, parallelism** and **slackness.**
  (1) **Work= $p \cdot T_p$**
  (2) **Span** (T_∞) is the longest path through a computation DAG.
  (3) **Speedup = T_1 / T_p**
    - The speedup$T_1/T_p = \Theta(p)$ gives a linear speedup.
    - The speedup $T_1/T_p = p$ gives a perfect speedup.
  (4) **Parallelism = T_1/T_∞**
  (5) **Slackness = $(T_1/T_\infty) /p = T_1 / (p \cdot T_\infty)$**
- The **deterministic algorithms** produce the same result on the same input on multiple runs.
- The **non-deterministic algorithms** may produce different results on the same input on different runs.
- A **determinacy race**: This happens when two logically parallel computations access the same memory and one of the computations writes on it. It causes the non-deterministic results of the computations.
- **Distributed computing environment** consists of the autonomous computers connected by an interconnection network.
- **The distributed algorithm** is a type of parallel algorithms that facilitates concurrent execution in the distributed computing environment.
- **Types of the distributed algorithms:**
  (1) Synchronous distributed algorithm: It refers to the synchronous network model of the distributed system.
  (2) Asynchronous distributed algorithm: It refers to the asynchronous network model of the distributed system.
- Complexity analysis of the distributed algorithms is based on four performance measures: **computational rounds, space, local running time and message complexity**
- **Some of the classic applications of the distributed algorithms:**
  (1) Distributed operating systems
  (2) Distributed database systems
  (3) Network and communication systems
  (4) Shared memory multiprocessor systems

(5) Digital circuits
(6) Real-time process control systems
(7) Distributed information processing
(8) E-commerce

- **The distributed breadth first search algorithm** performs BFS over a distributed network.
- **A minimum spanning tree (MST)** of a connected weighted graph G = (V, E) is a spanning tree T = (V, E') with the smallest cost where the cost of a tree is expressed as the sum of the costs of all its edges.
- **The minimum spanning tree problem** is to find a minimum-cost spanning tree of a given graph.
- **The string matching problem** is to find the pattern (sub-string) P in the string T.
- **Some applications of the string matching algorithms:**

(1) Computational biology
(2) Molecular chemistry
(3) Finding a particular DNA sequence
(4) Searching patterns in the documents formed using a large set of alphabets
(5) Word processing
(6) Web searching
(7) Desktop search (Google, MSN)
(8) Matching byte-strings containing graphical data or machine code
(9) grep command in Unix

Chapter Ends...

❑❑❑