Notes

Unit III

3 Greedy and Dynamic Programming Algorithmic Strategy

Syllabus

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.*

Contents

Part I: Greedy Strategy

3.1 Introduction to Greedy Strategy

SPPU : Dec.-06,07, May-07,09, Marks 8

3.1.1 Principle

- In an algorithmic strategy like Greedy, the decision of solution is taken based on the information available. The Greedy method is a straightforward method. This method is popular for obtaining the **optimized solutions**. In Greedy technique, the solution is constructed through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached. At each step the choice made should be,

Choice of solution made at each step of problem solving in Greedy approach

- **Feasible** - It should satisfy the problem's constraints.
- **Locally optimal** - Among all feasible solutions the best choice is to be made.
- **Irrevocable** - Once the particular choice is made then it should not get changed on subsequent steps.

- In short, while making a choice there should be a Greed for the optimum solution.
- In this chapter we will first understand the concept of Greedy method. Then we will discuss various examples for which Greedy method is applied.

3.1.2 Control Abstraction

In this section we will understand *"What is Greedy method?"*.

Algorithm

```
Greedy (D, n)
//In Greedy approach D is a domain
//from which solution is to be obtained of size n
//Initially assume
      Solution ← 0
            for i ← 1 to n do
                        {
        s ← select (D) // section of solution from D
        if (Feasible (solution, s)) then
          solution ← Union (solution, s);
                  }
        return solution
```

Check if the selected solution is feasible or not.

Make a feasible choices and select optimum solution.

In Greedy method following activities are performed.

1. First we select some solution from input domain.
2. Then we check whether the solution is feasible or not.
3. From the set of feasible solutions, particular solution that satisfies or nearly satisfies the objective of the function. Such a solution is called optimal solution.
4. As Greedy method works in stages. At each stage only one input is considered at each time. Based on this input it is decided whether particular input gives the optimal solution or not.

3.1.3 Applications of Greedy Strategy

- In this section we will discuss various problems that can be solved using Greedy approach -
  1) Knapsack problem
  2) Job sequencing with deadlines
  3) Minimum spanning trees
  4) Single source shortest path algorithm.
  5) Huffman's Tree

3.1.4 General Characteristics of Greedy Strategy

1. Greedy choice property : By this property, a globally optimal solution can be arrived at by making a locally optimal choice. That means, for finding the solution to the problem. We solve the subproblems, and whichever choice. We find best for that subproblem is considered. Then solve the subproblem is considered. Then solve the subproblem arising after the choice is made. This choice may depend upon the previously made choices but it does not depend on any future choice. Thus in greedy method, greedy choices are made one after the another, reducing each given problem instance to smaller one. The greedy choice property brings efficiency in solving the problem with the help of subproblems.

2. Optimal substructure : A problem shows optimal substructure if an optimal solution to the problem contains optimal solution to the sub-problems. In other words a problem has optimal substructure if the best next choice always leads to the optimal solution.

Example 3.1.1 *Explain the terms, feasible solution, optimal solution and objective function*

Solution : Feasible solution - For solving a particular problem there exists n inputs and we need to obtain a subset that satisfies some constraints. Then any subset that satisfies these constraints is called **feasible solution.**

Optimal solution - From a set of feasible solutions, particular solution that satisfies or nearly satisfies the objectives of the function such a solution is called optimal solution.

Objective function - A feasible solution that either minimizes or maximizes a given objective function is called objective function.

Review Question

1. *What are the characteristics of greedy method ?* **SPPU : Dec.-06,07, May-07,09, Marks 8**

3.2 Knapsack Problem

SPPU : May-12,16,17,18, Dec.-13,19, Oct.-16, April-19, Marks 8

- The Knapsack problem can be stated as follows.
- Suppose there are n objects from i = 1, 2,...n. Each object i has some positive weight w_i and some profit value is associated with each object which is denoted as p_i. This Knapsack carry at the most weight W.
- While solving above mentioned Knapsack problem we have the capacity constraint. When we try to solve this problem using Greedy approach our goal is,
  1. Choose only those objects that give maximum profit.
  2. The total weight of selected objects should be ≤ W.

  And then we can obtain the set of feasible solutions. In other words,

$$\text{maximized} \sum_{n}^{1} p_i x_i \text{ subject to } \sum_{n}^{1} w_i x_i \leq W$$

Where the Knapsack can carry the fraction x_i of an object i such that $0 \leq x_i \leq 1$ and $1 \leq i \leq n$.

3.2.1 Algorithm

The algorithm for solving Knapsack problem with Greedy approach is as given below.

```
Algorithm Knapsack_Greedy(W,n)
{
//p[i] contains the profits of i items such that 1 ≤ i ≤ n
//w[i] contains weights of I items.
//x[i] is the solution vector.
//W is the total size of Knapsack.
      for i : =1 to n do
{
if[w[i]<W) then//capacity of knapsack is a constraint
```

```
{
x[i] : =1.0;
W=W – w[i];
}
}
if(i<=n) then x[i] : =W/w[i];
}
```

This is the basic operation of selecting x[i] lies within for loop.
∴ Time complexity = Θ(n).

Example 3.2.1 *Consider the following instances of the knapsack problem : n = 3, m = 20, (P1, P2, P3) = (24, 25, 15) and (W1, W2, W3) = (18, 15, 20) find the feasible solutions.*

SPPU : May-18, Marks 6, Dec.-13, Marks 8

Solution : We will first find the feasible solutions. From these feasible solutions we will find the solution that gives us maximum profit. For obtaining feasible solution we use 3 approaches and these are -

1) Select object with maximum profit.

2) Select object with minimum weight.

3) Select object with maximum $\frac{\text{profit}}{\text{weight}}$ ratio.

Method 1 : Select object with maximum profit.

Object	x1	x2	x3
P	24	25	15
W	18	15	20

m = 20

Selected Object	Profit	Weight	Remaining weight
x2	25	15	20 – 15 = 5
x1	$24 * \frac{5}{18} = \frac{20}{3}$	$18 * \frac{5}{18} = 5$	5 – 5 = 0

Hence **Maximum profit** = $25 + \frac{20}{3} = 31.66$ with solution $\left(x2, \frac{x1 * 5}{18}\right)$

Method 2 : Select object with minimum weight.

M = 20

Selected Object	Profit	Weight	Remaining weight
x2	25	15	20 – 15 = 5
x1	$24 * \frac{5}{18} = \frac{20}{3}$	$18 * \frac{5}{18} = 5$	5 – 5 = 0

Hence **Maximum profit** = $25+\frac{20}{3} = 31.66$ with solution $\left(x2, \frac{x1*5}{18}\right)$

Method 3 : Select object with maximum $\frac{P}{W}$ ratio.

	x1	x2	x3
$\frac{P}{W}$	$\frac{4}{3}$	$\frac{5}{3}$	$\frac{3}{4}$

m = 20

Selected Object	Profit	Weight	Remaining weight
x2	25	15	20 – 15 = 5
x1	$24*\frac{5}{18}=\frac{20}{3}$	$18*\frac{5}{18}=5$	5 – 5 = 0

Hence **Maximum profit** = $25+\frac{20}{3} = 31.66$ with solution $\left(x2, \frac{x1*5}{18}\right)$

Thus by all the three methods we obtain the solution as $\left(x2, \frac{5\times x1}{18}\right)$ with maximu profit as **31.66.**

Example 3.2.2 *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).*

SPPU : May-16 (End Sem.), Marks 8

Solution : We will first find the feasible solutions. From these feasible solutions we w find the solution that gives us maximum profit. For obtaining feasible solution we use approaches and these are -

1) Select object with maximum profit.

2) Select object with minimum weight.

3) Select object with maximum $\frac{\text{profit}}{\text{weight}}$ ratio.

Method 1 : Select object with maximum profit.

Object	x1	x2	x3	x4	x5
p	10	20	30	40	50
w	20	30	66	40	60

m = 100

Selected object	Profit	Weight	Remaining weight
x5	50	60	100 – 60 = 40
x4	40	40	40 – 40 = 0

Hence **Maximum profit = 90**

with solution (x5, x4) = (0,0,0,1,1)

Method 2 : Select object with minimum weight.

m = 100

Selected object	Profit	Weight	Remaining weight
x1	10	20	100 – 20 = 80
x2	20	30	80 – 30 = 50
x4	40	40	50 – 40 = 10
x5	$50*\frac{1}{6} = 8.33$	$60*\frac{1}{6} = 10$	10 – 10 = 0

Hence **Maximum profit = 78.33**

with solution (1,1,0,1,1/6)

Method 3 : Select object with maximum profit/weight ratio.

$\frac{P}{W}$	x1	x2	x3	x4	x5
	$\frac{1}{2}$	$\frac{2}{3}$	$\frac{5}{11}$	1	$\frac{5}{6}$

m = 100

Selected object	Profit	Weight	Remaining weight
x4	40	40	100 – 40 = 60
x5	50	60	60 – 60 = 0

Maximum profit = 90

with solution (0,0,0,1,1)

This is a feasible solution.

Example 3.2.3 *Find the optimum solution to the Knapsack instance using greedy approach n = 7, m = 15, (p1, p2.....p7) = (10, 5, 15, 7, 6, 18, 3) and (w1, w2,.....w7) = (2, 3, 5, 7, 1, 4, 1)*

SPPU : Oct.-16 (In Sem), Marks 6

Solution : For solving this problem, we will find out the fraction of weight that can be used to fill up the Knapsack satisfying the given capacity with maximum weight. We will try various solutions for the same.

Feasible solutions can be

Solution	x_1	x_2	x_3	x_4	x_5	x_6	x_7
1	1/2	1/3	1/4	1/7	1	3/4	1
2	1/2	1/3	1	1/7	0	1	1/2
3	1	1/3	1	1/7	1	1	1
4	1	1/3	1	1/7	0	1	1
5	1	2/3	1	0	1	1	1

Solution 1 :

$$\sum w_i x_i = \left(2*\frac{1}{2}\right)+\left(3*\frac{1}{3}\right)+\left(5*\frac{1}{4}\right)+\left(7*\frac{1}{7}\right)+(1*1)+\left(4*\frac{3}{4}\right)+1$$

$$= 1 + 1 + 1.25 + 1 + 1 + 3 + 1$$

$$\sum w_i x_i = 9.25$$

$$\sum p_i x_i = \left(10*\frac{1}{2}\right)+\left(5*\frac{1}{3}\right)+\left(15*\frac{1}{4}\right)+\left(7*\frac{1}{7}\right)+(6*1)+\left(18*\frac{3}{4}\right)+3$$

$$= 5 + 1.67 + 3.75 + 1 + 6 + 13.5 + 3$$

$$\sum p_i x_i = 33.92$$

Solution 2 :

$$\sum w_i x_i = \left(2*\frac{1}{2}\right)+\left(3*\frac{1}{3}\right)+(5*1)+\left(7*\frac{1}{7}\right)+(1*0)+(4*1)+\left(1*\frac{1}{2}\right)$$

$$= 1 + 1 + 5 + 1 + 0 + 4 + 0.5$$

$$\sum w_i x_i = 12.5$$

$$\sum p_i x_i = \left(10*\frac{1}{2}\right)+\left(5*\frac{1}{3}\right)+(15*1)+\left(7*\frac{1}{7}\right)+(6*0)+(18*1)+\left(3*\frac{1}{2}\right)$$

$$= 5 + 1.67 + 15 + 0 + 18 + 1.5$$

$$\sum p_i x_i = 42.17$$

Solution 3 :

$$\sum w_i x_i = (2*1)+\left(3*\frac{1}{3}\right)+(5*1)+\left(7*\frac{1}{7}\right)+(1*1)+(4*1)+(1*1)$$

$$= 2 + 1 + 5 + 1 + 1 + 4 + 1$$

$$\sum w_i x_i = 15$$

$$\sum p_i x_i = (10*1)+\left(5*\frac{1}{3}\right)+(15*1)+\left(7*\frac{1}{7}\right)+(6*1)+(18*1)+(3*1)$$

$$= 10 + 1.67 + 15 + 1 + 6 + 18 + 3$$

$$\sum p_i x_i = 54.67$$

Solution 4 :

$$\sum w_i x_i = (2*1)+\left(3*\frac{1}{3}\right)+(5*1)+\left(7*\frac{1}{7}\right)+(1*0)+(4*1)+(1*1)$$

$$\sum w_i x_i = 14$$

$$\sum p_i x_i = (10*1)+\left(5*\frac{1}{3}\right)+(15*1)+\left(7*\frac{1}{7}\right)+(6*0)+(18*1)+(3*1)$$

$$\sum p_i x_i = 48.67$$

Solution 5 :

$$\sum w_i x_i = (2*1)+\left(3*\frac{2}{3}\right)+(5*1)+(7*0)+(1*1)+(4*1)+(1*1)$$

$$= 2 + 2 + 5 + 1 + 4 + 1$$

$$\sum w_i x_i = 15$$

$$\sum p_i x_i = (10*1)+\left(5*\frac{2}{3}\right)+(15*1)+(7*0)+(6*1)+(18*1)+(3*1)$$

$$= 10 + 3.33 + 15 + 0 + 6 + 18 + 3$$

$$\sum p_i x_i = 55.33$$

From above computations, clearly solution 5 gives maximum profit. Hence solution 5 is considered as optimum solution.

Example 3.2.4 *How does fractional greedy algorithm solves the following knapsack problem with capacity 8, Items (I_1, I_2, I_3, I_4), profit P = (25, 24, 15, 40) and weight w = (3, 2, 2, 5).* **SPPU : April-19, Marks 5**

Solution : Largest profit strategy : We will pick up the object with largest profit. If weight of object exceeds remaining knapsack capacity then take fraction of object, to fill up.

Step 1 : Select object 4.

The remaining capacity = $m - W_4 = m - 5 = 8 - 5 = 3$

Total profit P = 40

Step 2 : Select object 1.

The remaining capacity = $3 - W_1 = 3 - 3 = 0$

Total profit P = 40 + 25 = 65

Thus the solution = (1, 0, 0, 1)

Smallest weight strategy :

Step 1 : Select object 2 with weight 2

$\therefore$ $I_2 = 1$

P = 24, Capacity = 8 – 2 = 6

Step 2 : Select object 3 with weight 2

$\therefore$ $I_3 = 1$

P = 24 + 15 = 39, Capacity 6 – 2 = 4

Step 3 : Select object 1, with weight 3

$\therefore$ $I_1 = 1$

P = 39 + 25 = 64, Capacity = 4 – 3 = 1

Step 4 : If we select object 4 then capacity < W_5

$\therefore$ Capacity = $\frac{\text{Remaining weight}}{W_5} = \frac{1}{5}$

$\therefore$ Profit = $64 + \left(\frac{1}{5} * 40\right) = 72$

Step 5 : Now knapsack is full.

Hence $(I_1, I_2, I_3, I_4) = \left(1, 1, 1, \frac{1}{5}\right)$ with profit = 72

Comparing above two cases, the case with smallest weight strategy gives max profit.

Example 3.2.5 *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 sol using Greedy approach ?*

SPPU : Dec.-19, Mar

Solution : We will use largest profit strategy to pick up the object. For that purpo will compute $\frac{P_i}{W_i}$.

x_i	1	2	3
P_i	60	100	120
W_i	10	20	30
$\frac{P_i}{W_i}$	6	5	4

Step 1 :

Select $x_i = 1$

Total Profit = 60

Remaining Weight = $M - W_1$

$= 50 - 10$

$= 40$

Step 2 :

Now Select $x_i = 2$

Total Profit = $60 + 100 = 160$

Remaining weight = $40 - 20$

$= 20$

Step 3 :

We can not select items 3 as a whole. So, only select the weight which will fill up the knapsack completely.

∴ Take only wt 20 out of 30.

∴ Remaining weight = $20 - 20 = 0$

$$\sum W_i x_i = \left(10 \times 1 + 20 \times 1 + 30 * \frac{20}{30}\right) = 50$$

$$\sum p_i x_i = \left(60 * 1 + 100 * 1 + 120 * \frac{20}{30}\right)$$

$$= 60 + 100 + 80$$

$$= 240$$

The optimal solution is $\left(1, 1, \frac{2}{3}\right)$

Example 3.2.6 *Consider knapsack capacity W = 50, W = (10, 20, 40) and V = (60, 80, 100). Find maximum profit using greedy approach.*

Solution : We will first find the feasible solutions. From these feasible solutions we choose the solution that gives us maximum profit. For obtaining feasible solution we use 3 approaches, these are -

1) Select object with maximum profit

2) Select object with minimum weight

3) Select object with maximum P/W or V/W ratio

Method 1 : Select object with maximum profit.

Object	1	2	3
V	60	80	100
W	10	20	40

Selected object	Profit	Weight	Remaining Weight
x_3	100	40	50 – 40 = 10
x_1	60	10	10 – 10 = 0

Hence **maximum profit = 160** with solution (x_3, x_1).

Method 2 : Select object with minimum weight.

Selected object	Profit	Weight	Remaining Weight
x_1	60	10	50 – 10 = 40
x_2	80	20	40 – 20 = 20
x_3	$100 * \frac{1}{2} = 50$	$40 * \frac{1}{2} = 20$	20 – 20 = 0

Hence solution is $\left(x_1, x_2, \frac{x_3}{2}\right)$ with total profit = 190.

Method 3 : Select object with maximum V/W ratio.

V / W	x_1	x_2	x_3
	6	4	2.5

Selected object	Profit	Weight	V/W	Remaining Weight
x_1	60	10	6	50 – 10 = 40
x_2	80	20	4	40 – 20 = 20
x_3	$100 * \frac{1}{2} = 50$	$40 * \frac{1}{2} = 20$	2.5	20 – 20 = 0

Hence solution is $\left(x_1, x_2, \frac{x_3}{2}\right)$ with total profit = 190.

Example 3.2.7 *Solve the following knapsack problem using greedy method. Number of items = 5. Capacity W = 100. Weight vector {50, 40, 30, 20, 10} and Profit vector = (1,2,3,4,5).*

Solution : Let, the given data be -

Item	x_1	x_2	x_3	x_4	x_5
Profit (p)	1	2	3	4	5
Weight (w)	50	40	30	20	10
p/w	0.02	0.05	0.1	0.2	0.5

Method 1 : Maximum profit object.

Item selected	Profit	Weight	Remaining Weight
x_5	5	10	100 – 10 = 90
x_4	4	20	90 – 20 = 70
x_3	3	30	70 – 30 = 40
x_2	2	40	40 – 40 = 0

Total profit = 14

Total weight = 100

Method 2 : Minimum weight object.

Item selected	Profit	Weight	Remaining Weight
x_5	5	10	100 – 10 = 90
x_4	4	20	90 – 20 = 70
x_3	3	30	70 – 30 = 40
x_2	2	40	40 – 40 = 0

Total profit = 14

Total weight = 100

Method 3 : Maximum profit / weight ration.

Item selected	Profit	Weight	Remaining Weight
x_5	5	10	100 – 10 = 90
x_4	4	20	90 – 20 = 70
x_3	3	30	70 – 30 = 40
x_2	2	40	40 – 40 = 0

∴ Total profit = 14

and Total weight = 100

Thus the solution to this problem is (x_5, x_4, x_3, x_2) or **(5,4,3,2).**

Example 3.2.8 *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 knapsack. Find the optimal answer using greedy design technique. Also write the time complexity of greedy approach for solving knapsack problem. P = [40, 10, 50, 30, 60], W = [80, 10, 40, 20, 90], M = 110.*

Solution : Let n = 5 and M = 110 we will first find out P/W ratio select the objects decreasing order of P/W ratio.

Objects	1	2	3	4	5
Profit	40	10	50	30	60
Weight	80	10	40	20	90
P/W	$\frac{1}{2} = 0.5$	1	1.25	1.5	0.66

Now each time we will select max $\left(\frac{P}{W}\right)$ value, from remaining list.

Object selected	Profit	Weight	Remaining weight
4	30	20	110 – 20 = 90
3	50	40	90 – 40 = 50
2	10	10	50 – 10 = 40
5 Not selected	60	Can not select as W > remaining capacity	
Total profit = 90. Solution (2, 3, 4)			

From above table, note that we can not select object 5 and object 4 as it is a (0/1) knapsack problem and we can not put fractional weight in knapsack.

Thus now, total weight = 70

total profit = 90

The solution is (x_4, x_3, x_2).

Review Questions

1. *Explain general strategy of Greedy method with the help of its control abstraction for the subset paradigm. Write an algorithm which uses this strategy for solving the Knapsack problem.* **SPPU : May-12, Marks 10**
2. *Write an algorithm for Knapsack problem using Greedy strategy.* **SPPU : May-17, Marks 6**

3.3 Scheduling Algorithms - Job Scheduling

SPPU : Dec.-11,12,16, June-12, May-13, April-19, Marks 10

Consider that there are n jobs that are to be executed. At any time t = 1,2,3,.. only exactly one job is to be executed. The profits pi are given. These profits are gained by corresponding jobs. For obtaining feasible solution we should take care that the jobs get completed within their given deadlines.

Let n = 4

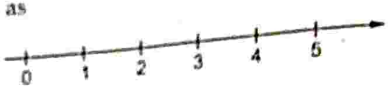
n	pi	di
1	70	2
2	12	1
3	18	2
4	35	1

We will follow following rules to obtain the feasible solution

- Each job takes one unit of time.

- If job starts **before or at its deadline**, profit is obtained, otherwise no profit.
- Goal is to schedule jobs to maximize the total profit.
- Consider all possible schedules and compute the minimum total time in the syste

Consider the Time line as



The feasible solutions are obtained by various permutations and combinations of

n	pi
1	70
2	12
3	18
4	35
1,3	88
2,1	82
2,3	30
3,1	88
4,1	105
4,3	53

Each job takes only one unit of time. Deadline of job means a time on which **before which the job has to be executed.** The sequence {2,4} is not allowed beca both have deadline 1. If job 2 is started at 0 it will be completed on 1 but we can start job 4 on 1 since deadline of job 4 is 1. The feasible sequence is a sequence t allows all jobs in a sequence to be executed within their deadlines and highest profit be gained.

- The optimal solution is a feasible solution with maximum profit.
- In above example sequence 3,2 is not considered as $d_3 > d_2$ but we have considered sequence 2,3 as feasible solution because $d_2 < d_3$.
- We have chosen job 1 first then we have chosen job 4. The solution 4,1 is feasible the order of execution is 4 then 1. Also $d_4 < d_1$. If we try {1,3,4} then it is not a feasib solution, hence reject 3 from the set. Similarly if we add job 2 in the sequence th the sequence becomes {1,2,4}. This is also not a feasible solution hence reject it. Fina the feasible sequence is 4,1. This sequence is optimum solution as well.

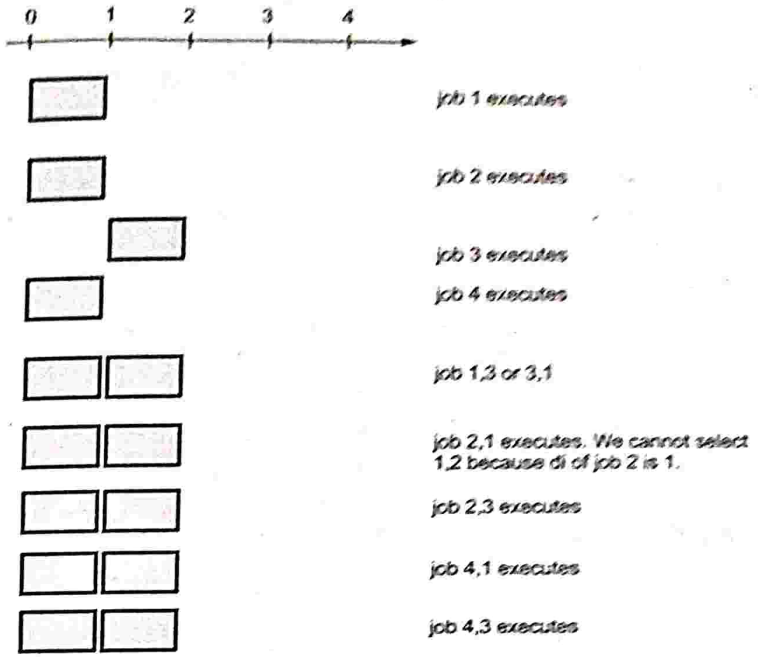


Fig. 3.3.1 Job sequencing with deadline

Example 3.3.1 *Solve the following instance of "job sequencing with deadlines" problem : " $n = 7$, profits $(p_1, p_2, p_3, \ldots\ldots, p_7) = (3, 5, 20, 18, 1, 6, 30)$ and deadlines $(d_1, d_2, \ldots\ldots, d_7) = (1, 3, 4, 3, 2, 1, 2)$*

SPPU : Dec.-11, 16, (End Sem Exam) Marks 8

Solution : Step 1 : We will arrange the profits p_i in descending order. Then corresponding deadlines will appear.

Profit	30	20	18	6	5	3	1
Job	P_7	P_3	P_4	P_6	P_2	P_1	P_5
Deadline	2	4	3	1	3	1	2

Step 2 : Create an array J [ ] which stores the jobs. Initially J [ ] will be filled with 0

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

J [7]

Step 3 : Add i^{th} job in array J [ ] at the index denoted by its deadline D_i. First job is p_7. The deadline for this job is 2. Hence insert p_7 in the array j [ ] at index.

1	2	3	4	5	6	7
	p_7					

Step 4 : Next job is p_3. Insert it in array J [ ] at index 4.

1	2	3	4	5	6	7
	p_7		p_3			

Step 5 : Next job is p_4. It has a deadline 3. Therefore insert it at index 3

1	2	3	4	5	6	7
	p_7	p_4	p_3			

Step 6 : Next job is p_6, it has deadline 1. Hence place p_6 at index 1.

1	2	3	4	5	6	7
p_6	p_7	p_4	p_3			

Step 7 : Next job is p_2, it has deadline 3. But as index 3 is already occupied and th is no empty slot at index < J [3]. Just discard job p_2. Similarly job p_1 and p_5 will discarded.

Step 8 : Thus the optimal sequence which we will obtain will be 6-7-4-3. The maxim profit will be **74.**

Example 3.3.2 *Solve the following job sequencing problem (maximizing the profit b completing jobs before their deadlines) using greedy algorithm.*

N (Number of jobs) = 4

Profits associated with jobs $(P_1, P_2, P_3, P_4) = (100, 10, 15, 27)$. *Deadlines associate with jobs* $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$.

SPPU : Dec.-12, Marks

Solution : Let Profits associated with jobs $(P_1, P_2, P_3, P_4) = (100, 10, 15, 27)$. Deadlines associated with jobs $(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$.

Step 1 : We will arrange the profits P_i in descending order. Then corresponding deadlines will appear.

Profit	100	27	15	10
Job	P_1	P_4	P_3	P_2
Deadline	2	1	2	1

Step 2 : Create an array J [ ] which stores the jobs. Initially J [ ] will be

1	2	3	4
0	0	0	0

J [4]

Step 3 : Add i^{th} job in array J [ ] at the index denoted by its deadline d_i. First job is P_1.The deadline for this job is 2. Hence insert P_1 in J [ ] at 2^{nd} index.

1	2	3	4
	P_1		

Step 4 : Next job is P_4. Insert it at index 1.

1	2	3	4
P_4	P_1		

Step 5 : Next job is P_3 with deadline 2. But the J[2] is already occupied. Hence discard P_3. Similarly job P_2 will also be discarded.

Step 6 : Thus the final optimal sequence which we will obtain will be $P_1 - P_4$. The maximum profit will be **127.**

Example 3.3.3 *Explain job-sequencing with deadlines. Solve the following instance :*

$n = 5.$

$(P_1, P_2, P_3, P_4, P_5) = (20, 15, 10, 5, 1)$

$(d_1, d_2, d_3, d_4, d_5) = (2, 2, 1, 3, 3)$

SPPU : June-12, May-13, Marks 10

Solution :

Step 1 : We will arrange the profits P_i in descending order, along with corresponding deadlines.

Profit	20	15	10	5	1
Job	P_1	P_2	P_3	P_4	P_5
Deadline	2	2	1	3	3

Step 2 : Create an array J [ ] which stores the jobs. Initially J [ ] will be

1	2	3	4	5
0	0	0	0	0

J [5]

Step 3 : Add i^{th} Job in array J[ ] at index denoted by its deadline D_i.
First job is P_1, its deadline is 2.
Hence insert P_1 in the array J[ ] at 2^{nd} index.

1	2	3	4	5
	P_1			

Step 4 : Next job is P_2 whose deadline is 2 but J[2] is already occupied. We will search for empty location < J[2]. The J[1] is empty. Insert P_2 at J[1].

Step 5 : Next job is P_3 whose deadline is 1. But as J[1] is already occupied discard this job.

Step 6 : Next job is P_4 with deadline 3. Insert P_4 at J[3].

1	2	3	4	5
P_2	P_1	P_4		

Step 7 : Next job is P_5 with deadline 3. But as there is empty slot at < = J[3], we will discard this job.

Step 8 : Thus the optimal sequence which we will obtain will be $P_2 - P_1 - P_4$. The maximum profit will be 40.

Example 3.3.4 *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

SPPU : April 19, Marks 5

Solution :

Step 1 : We will arrange the profits P_i in descending order, along with corresponding deadlines.

Profit	20	15	10	5	1
Job	P_1	P_2	P_3	P_4	P_5
Deadline	2	2	1	3	3

Step 2 : Create an array J [ ] which stores the jobs. Initially J [ ] will be

1	2	3	4	5
0	0	0	0	0

J [5]

Step 3 : Add i^{th} Job in array J[ ] at index denoted by its deadline D_i.
First job is P_1, its deadline is 2.
Hence insert P_1 in the array J[ ] at 2^{nd} index.

1	2	3	4	5
	P_1			

Step 4 : Next job is P_2 whose deadline is 2 but J[2] is already occupied. We will search for empty location < J[2]. The J[1] is empty. Insert P_2 at J[1].

Step 5 : Next job is P_3 whose deadline is 1. But as J[1] is already occupied discard this job.

Step 6 : Next job is P_4 with deadline 3. Insert P_4 at J[3].

1	2	3	4	5
P_2	P_1	P_4		

Step 7 : Next job is P_5 with deadline 3. But as there is empty slot at <= J[3], we will discard this job.

Step 8 : Thus the optimal sequence which we will obtain will be $P_2 - P_1 - P_4$. The maximum profit will be 40.

3.3.1 Algorithm

The algorithm for job sequencing is as given below

```
Algorithm Job_seq(D,J,n)
{
//Problem description : This algorithm is for job sequencing using Greedy method.
//D[i] denotes i^th deadline where 1 ≤ i ≤ n
//J[i] denotes i^th job
// D[J[i]] ≤ D[J[i+1]]
//Initially
D[0]←0;
J[0]←0;
J[1]←1;
count←1;
for ←2 to n do
{
 t←count;
 while(D[J[t]] > D[i]) AND D[J[t]]!=t)) do t←t-1;
 if((D[J[t]] ≤ D[i])AND(D[i] > t)) then
 {
      //insertion of i^th feasible sequence into J array
      for s←count to (t+1) step -1 do
      J[s+1] ← J[s];
      J[t+1]←I;
      count←count+1;
 }//end of if
}//end of while
return count;
}
```

The sequence of J will be inserted if and only if D[J[t]]!=t. This also means that the job J will be processed if it is in within the deadline.

The computing time taken by above Job_seq algorithm is $O(n^2)$, because the basic operation of computing sequence in array J is within two nested for loops.

Strategy to Solve Job Sequencing with Deadlines Problem

1. Arrange the list based on descending order of profits. Read the profits array from left to right.
2. Fill up the job array using the deadlines.
3. If the job array has vacant position at the location indicated by the deadline, then insert the P_i at corresponding index in job array. If it is not vacant then search for the < than current deadline indexes in job array. If empty location is found then insert P_i otherwise discard that job.
4. Finally read the job array to get the optimal sequence.

3.4 Activity Selection Problem

Activity selection problem is defined as follows :

Given a **resource** such as a CPU or a **lecture** hall and n set of **activities**, such as task or lectures that want to utilize the resource. The activity i have the **starting** and **finishing time** which can be denoted by S_i and F_i. Then activity selection problem is to find max_size subset A of **mutually compatible** activities. [i.e. maximize the number of lecture all in the lecture hall or maximize number of tasks assigned to the CPU].

Example 3.4.1 *Consider the following set of activities denoted by S. select the compatible set of activities.*

i	1	2	3	4	5	6	7	8	9	10	11
S_i	1	3	0	5	3	5	6	8	8	2	14
F_i	4	5	6	7	8	9	10	11	12	15	16

Solution :

Step 1 : Sort F_i i.e. finishing time into non-decreasing order. After sorting $F_1 \leq F_2 \leq F_3 \leq \ldots\ldots \leq F_n$

Step 2 : Select the next activity i to the solution set if i is compatible with each activity in the solution set.

Step 3 : Repeat step 2, until all the activities get examined.

From above set first of all we will select a_1, the finish time of a_1 is 4, Hence we will search for the activity with $S_i \geq 4$. We will select a_4 (because $S_4 = 5$ which ≥ 4). The F_4 is 7. Hence select next activity such that $S_i \geq 7$. We will get S_8. The $F_8 = 11$. Hence select

next activity such that $S_i \geq 11$. We obtain a_{11} activity. We select a_{11}. Hence the solution set = {1, 4, 8, 11}.

The algorithm for activity selection problem is as given below.

Algorithm :

```
Algorithm Greedy_Act_sel (S, F)
// Problem Description : This algorithm selects the activities
// that are compatible to each other
// Input : The set of starting time and finish time.
// Output : Set of selected activities.
    n ← length [S] // n represents total number of activities.
    A ← {a} // selection of first activity.
    i ← 1  // current activity is denoted by a_i.

    for m ← 2 to n                    <- F_i is maximum finish time of any activity in A
    {
    if (S_m ≥ F_i) then

        A ← A ∪ {a_m}                 <- Selection of next feasible activity
        i ← m
    }
        return A
```

Analysis : The algorithm using greedy approach for activity selection works in $\theta(n)$ time.

The activity selection problem can be solved using two approaches -

1. Greedy-choice property.
2. Optimal substructure.

1. Greedy choice property :

- In this approach of problem solution a globally optimal solution can be obtained by making a **locally optimal choice.**
- Select a **choice** whichever seems to be **best at the moment** and then solve the subproblem arising after the choice is made.
- The choice made by a greedy algorithm depends upon the **choices made so far** but it does not depend upon the future choices.

2 Optimal substructure

A problem shows **optimal substructure** if an optimal solution to the problem contains within it optimal solutions to sub-problems.

The optimal solution is denoted by A. The selection process starts by selecting activity 1 and then $A' = A - \{1\}$ where $i \in s$ and $S_i \geq f_1$.

Example 3.4.2 *Write greedy algorithm for activity selection problem. Give its time complexity. For following intervals, select the activities according to your algorithm.* $I_1(1-3)$, $I_2(0-2)$, $I_3(3-6)$, $I_4(2-5)$, $I_5(5-8)$, $I_6(3-10)$, $I_7(7-9)$.

Solution :

Step 1 : Sort the given activities in ascending order according to their finishing time.

Step 2 : Select the first activity from sorted array and add it to **solution** array.

Step 3 : For next subsequent activity if start time of currently selected activity is greater than or equal to finish time of previously selected activity, then add it to **solution** array.

Step 4 : Select next activity.

Step 5 : Repeat step 3 and step 4 for all remaining activities.

Time complexity is O(n log n).

Example :

Step 1 : We will sort the given activities in ascending order according to their finish time. The sorted table is

Start Time	Finish Time	Activity Name
0	2	I_2
1	3	I_1
2	5	I_4
3	6	I_3
5	8	I_5
7	9	I_7
3	10	I_6

Step 2 : Select first activity i.e. activity I_2 to add it to **solution** array.
Hence **Solution** = $\{I_2\}$.

Step 3 : Now select activity I_1. If start (I_1) < finish (I_2). So we will not add I_1 to **solution** array.

Step 4 : Now select activity I_4. Start (I_4) = Finish (I_4) = 2. Add I_4 to solution array.

$\therefore$ Solution = $\{I_2, I_4\}$

Step 5 : Select I_3. Start (I_3) < Finish (I_5). Hence do not add it to solution array.

Step 6 : Similarly activities I_3, I_5, I_7 and I_6 will not get added to solution array.

Step 7 : Hence solution is $\{I_2, I_4\}$.

Review Question

1. *Explain the two approaches that can be used for solving activity selection problem using greedy strategy.*

Part II: Dynamic Programming

3.5 Introduction

SPPU : May-07,08,10, Dec.-06,07,11,13,18, Aug.-15, Marks 17

- Dynamic Programming is invented by U.S Mathematician Richard Bellman in 1950.
- Dynamic Programming is a technique for solving problems with overlapping subproblems.
- In this method each sub-problem is solved only once. The result of each subproblem is recorded in a table from which we can obtain a solution to original problem.
- Dynamic programming is typically applied for optimization problem.
- For each given problem we may get any number of solutions we seek for optimum solution(i.e. minimum value or maximum value solution). And such solution becomes the solution to the given problem.

3.5.1 Steps of Dynamic Programming

Dynamic programming design involves **4 major steps :**

1. Characterize the structure of optimal solution. That means develop a mathematical notation that can express any solution and subsolution for the given problem.
2. Recursively define the value of an optimal solution.
3. By using bottom up technique compute value of optimal solution. For that you have to develop a recurrence relation that relates a solution to its subsolutions, using the mathematical notation of step 1.
4. Compute an optimal solution from computed information.

3.5.2 Principle of Optimality

- The dynamic programming algorithm obtains the solution using principle of optimality.
- The principle of optimality states that "in an optimal sequence of decisions or choices, each subsequence must also be optimal."
- When it is not possible to apply the principle of optimality it is almost impossible to obtain the solution using the dynamic programming approach.

- For example : Finding of shortest path in a given graph uses the principle of optimality.

3.5.3 Elements of Dynamic Programming

1) **Optimal substructure** - The dynamic programming technique makes use of principle of optimality to find the optimal solution from subproblems.

2) **Overlapping subproblems** - The dynamic programming is a technique in which the problem is divided into subproblems. The solutions of subproblems are shared to get the final solution to the problem. It avoids repetition of work and we can get the solution more efficiently.

3.5.4 Applications of Dynamic Programming

Dynamic programming is used for solving various problems such as

1. Multistage Graph
2. Traveling Salesman Problem
3. Matrix Multiplication Problem
4. Longest Common Subsequence Problem
5. Maximum Flow Problem
6. All Pair Shortest Path Problem

3.5.5 Difference between Dynamic Programming and Greedy Technique

Difference between Divide and Conquer and Greedy strategy

Sr. No.	Divide and conquer	Greedy algorithm
1.	Divide and conquer is used to obtain a solution to given problem.	Greedy method is used to obtain optimum solution.
2.	In this technique, the problem is divided into small subproblems. These subproblems are solved independently. Finally all the solutions of subproblems are collected together to get the solution to the given problem.	In greedy method a set of feasible solution is generated and optimum solution is picked up.
3.	In this method, duplications in subsolutions are neglected. That means duplicate solutions may be obtained.	In greedy method, the optimum selection is without revising previously generated solutions.
4.	Divide and conquer is less efficient because of rework on solutions.	Greedy method is comparatively efficient but there is no as such guarantee of getting optimum solution.
5.	Examples : Quick sort binary search	Examples : Knapsack problem, finding minimum spanning tree.

Difference between Dynamic Programming and Greedy Strategy

Sr. No.	Greedy method	Dynamic programming
1.	Greedy method is used for obtaining optimum solution.	Dynamic programming is also for obtaining optimum solution.
2.	In Greedy method a set of feasible solutions and the picks up the optimum solution.	There is no special set of feasible solutions in this method.
3.	In Greedy method the optimum selection is without revising previously generated solutions.	Dynamic programming considers all possible sequences in order to obtain the optimum solution.
4.	In Greedy method there is no as such guarantee of getting optimum solution.	It is guaranteed that the dynamic programming will generate optimal solution using principle of optimality.

Review Questions

1. *What are the common step in the dynamic programming to solve any problem ? Compare dynamic programming with greedy approach.* **SPPU : Dec.-11, Marks 6**
2. *Write control abstraction (General Strategy Algorithm) of dynamic programming.* **SPPU : Aug.-15 (In Sem.), Marks 4**
3. *What is the "Principle of optimality" ?* **SPPU : Dec.-06, 07, Marks 2**
4. *What is dynamic programming ? Is this the optimization technique ? Give reasons. What are its drawbacks ? Explain memory functions.* **SPPU : Dec.-13, Marks 17**
5. *Name the elements of dynamic programming ? How does the dynamic programming solve the problem ?* **SPPU : Dec.-06, 07, May-07, 08, Marks 6, May-10, Marks 4**
6. *State and explain the principle of dynamic programming. Name the elements of dynamic programming.* **SPPU : Aug.-15 (In Sem.) Marks 2**
7. *Comparison of greedy approach and dynamics programming.* **SPPU : Dec.-18, Marks 6**

3.6 Binomial Coefficients

- Computing binomial coefficient is a typical example of applying dynamic programming.
- In mathematics, particularly in combinatorics, binomial coefficient is a coefficient of any of the terms in the expansion of $(a+b)^n$. It is denoted by C(n, k) or $\binom{n}{k}$ where $(0 \leq k \leq n)$.

- The formula for computing binomial coefficient is,

$$C(n, k) = C(n-1, k-1) + C(n-1, k)$$

and $C(n, 0) = 1$

$C(n, n) = 1$

where $n > k > 0$

For example : Compute C (4, 2)

$n = 4, k = 2$

$$C(4, 2) = C(n-1, k-1) + C(n-1, k)$$

$$C(4, 2) = C(3, 1) + C(3, 2) \quad \text{... (3.6.1)}$$

As there are two unknowns : C(3, 1) and C(3, 2) in above equation we will compute these sub instances of C(4, 2).

$\therefore$ $n = 3, k = 1$

$$C(3, 1) = C(2, 0) + C(2, 1)$$

As $C(n, 0) = 1$ We can write

$C(2, 0) = 1$

$\therefore$ $C(3, 1) = 1 + C(2, 1)$ ... (3.6.2)

Hence let us compute C(2, 1).

$n = 2, k = 1$

$\therefore$ $C(2, 1) = C(n-1, k-1) + C(n-1, k)$

$= C(1, 0) + C(1, 1)$

But as $C(n, 0) = 1$ and $C(n, n) = 1$ we get

$C(1, 0) = 1$ and $C(1, 1) = 1$

$\therefore$ $C(2, 1) = C(1, 0) + C(1, 1)$

$= 1 + 1$

$$\boxed{C(2, 1) = 2} \quad \text{... (3.6.3)}$$

Put this value in equation (3.6.2) and we get

$C(3, 1) = 1 + 2$

$$\boxed{C(3, 1) = 3} \quad \text{... (3.6.4)}$$

Now to solve equation 1 we will first compute C(3, 2) with n = 3 and k = 2.

$\therefore$ $C(3, 2) = C(n-1, k-1) + C(n-1, k)$

$$C(3, 2) = C(2, 1) + C(2, 2)$$

But as C(n, n) = C(2, 2) = 1, we will put values of C(2, 1) [obtained in equation (3.6.3) and C(2, 2) in C(3, 2) we get,

$$C(3, 2) = C(2, 1) + C(2, 2)$$
$$= 2 + 1$$

$$\boxed{C(3, 2) = 3} \quad \ldots (3.6.5)$$

Put equation (3.6.4) and (3.6.5) in equation (3.6.1), then we get

$$C(4, 2) = C(3, 1) + C(3, 2)$$
$$= 3 + 3$$

$\boxed{C(4, 2) = 6}$ is the final answer.

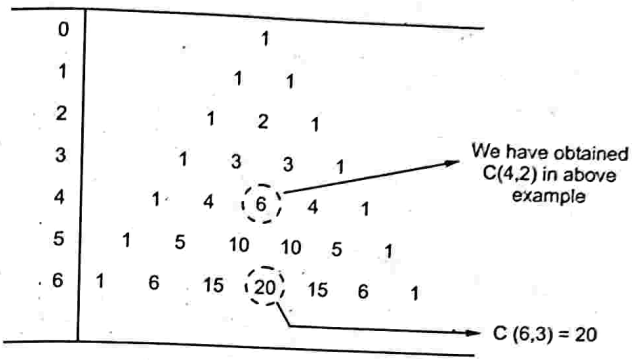
How Dynamic Programming approach is used ?

While computing C(n,k) the smaller overlapping sequences get generated by C(n – 1, k – 1) and C(n – 1, k). These overlapping, smaller instances of problem need to be solved first. The solutions which we obtained by solving these instances will ultimately generate the final solution. Thus for computing binomial coefficient dynamic programming is used.

If we record binomial coefficients n and k values ranging from 0 to n and 0 to k then

	0	1	2	3	4	5	...	(k – 1)	k
0	1								
1	1	1							
2	1	2	1						
3	1	3	3	1					
4	1	4	6	4	1				
5									
⋮									
k	1								1
⋮									
(n – 1)	1							C(n – 1, k – 1)	C(n – 1, k)
n	1								C(n,k)

This basically is a structure known as Pascal's triangle.



To compute C(n,k) we fill up the above given table row by row starting with C(n,0) = 1 and ending with C(n, n) = 1. The cell at current row is calculated by two adjacent cells of previous row. The algorithm for computing binomial coefficient is as given below,

Algorithm :

```
Algorithm Binomial (n,k)
//Problem Description : This algorithm is for
//computing C(n,k) i.e., Binomial coefficient
//Input : A pair of non negative integers n and k.
//Output : The value of C(n,k)
    for i ← 0 to n do
    {
        for j←0 to k do
//k is computed as min(i,k)
            if ((j=0 or (i=j)) then
            C[i,j] ← 1
            else //start filling the table
                C[i,j] ← C[i–1,j–1] + C[i–1,j]
    }
    return C[n,k] //Computed value of C(n,k)
```

Analysis :

The basic operation is addition i.e.,

$$C[i, j] \leftarrow C[i - 1, j - 1] + C[i - 1, j]$$

Let A(n,k) denotes total additions made in computing C(n,k).

$$A(n,k) = \sum_{i=1}^{k} \sum_{j=1}^{i-1} 1 + \sum_{i=k+1}^{n} \sum_{j=1}^{k} 1$$

$$= \sum_{i=1}^{k} [(i-1) - 1 + 1] + \sum_{i=k+1}^{n} (k - 1 + 1)$$

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

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

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

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

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

$$= \frac{(k-1)k}{2} + k\,(n-k) = \frac{k^2}{2} - \frac{k}{2} + nk - k^2$$

$$= nk$$

Hence time complexity of binomial coefficient is Θ (nk).

C Program

```
/*****************************************************************
Program for computing binomial coefficient C(n,k)
*****************************************************************/
#include<stdio.h>
#include<conio.h>
#define MAX 10
void main()
{
int n,k,result;
int Binomial(int n,int k);
clrscr();
printf("\n Enter the value of n and k");
```

```
scanf("%d%d",&n,&k);
result= Binomial(n,k);
printf("\n The Binomial Coefficient ");
        printf("C(%d,%d)= %d",n,k,result);

getch();
}
int Binomial(int n,int k)
{
int i,j;
int C[MAX][MAX];
for(i=0;i<MAX;i++)
for(j=0;j<MAX;j++)
        C[i][j]=0;

for(i=0;i<=n;i++)
{
        for(j=0;j<=k;j++)
        {
        if((j==0)||(i==j))
            C[i][j]=1;
         else
            C[i][j]=C[i-1][j-1]+C[i-1][j];
        }
}
return C[n][k];
}

int min(int a,int b)
{
if(a<b)
        return a;
    else
        return b;
}
```

Output

Enter the value of n and k = 4,2

The Binomial Coefficient C(4,2)=6

3.7 OBST

SPPU : Dec.-06,08,12,15, May-07,09,14,11, Aug.-15, Marks 16

- Suppose we are searching a word from a dictionary. And for every required word we are looking up in the dictionary then it becomes time consuming process. To perform this lookup more efficiently we can build the binary search tree of common words as key elements. Again we can make this binary search tree efficient by arranging frequently used words nearer to the root and less frequently words away from the root. Such a binary search tree makes our task more simplified as well as efficient. This type of binary search tree is also called optimal binary search tree (OBST).

Problem Description

- Let $\{a_1, a_2, \ldots a_n\}$ be a set of identifiers such that $a_1 < a_2 < a_3$. Let p(i) be the probability with which we can search for a_i and q(i) be the probability of searching element x such that $a_i < x < a_{i+1}$ and $0 \le i \le n$. In other words p(i) is the probability of successful search and q(i) be the probability of unsuccessful search. Also $\sum_{1 \le i \le n} p(i) + \sum_{1 \le i \le n} q(i)$ then obtain a tree with minimum cost. Such a tree with optimum cost is called optimal binary search tree.
- To solve this problem using dynamic programming method we will perform following steps.

Step 1 : Notations used

Let,

$$T_{ij} = \text{OBST}(a_{i+1}, \ldots, a_j)$$

C_{ij} denotes the cost(T_{ij}).

W_{ij} is the weight of each T_{ij}.

T_{0n} is the final tree obtained.

T_{00} is empty.

$T_{i,i+1}$ is a single-node tree that has element a_{i+1}.

During the computations the root values are computed and r_{ij} stores the root value of T_{ij}.

Step 2 : The OBST can be build using the principle of optimality. Consider the process of creating OBST.

- Let T_{0n} be an OBST for the elements $a_1 < a_2 < \ldots < a_n$, and let L and R be its left subtree and right subtree. Suppose that the root of T_{0n} is a_k, for some k.
- Then the elements in the left subtree L are $a_1, a_2, \ldots, a_{k-1}$ and the elements in the right subtree R are $a_{k+1}, a_{k+2}, \ldots, a_n$.

- The cost of computing the T_{0n} can be given as

$$C(T_{0n}) = C(L) + C(R) + p_1 + p_2 + ... + p_n + q_0 + q_1 + q_2 + ... + q_n$$

$$\text{i.e. } C(T_{0n}) = C(L) + C(R) + W$$

$$\text{where } W = p_1 + p_2 + ... + p_n + q_0 + q_1 + q_2 + ... + q_n$$

- If L is not an optimal BST for its elements, then we can find another tree L′ for the same elements, with the property C(L′) < C(L) (i.e. optimal cost).
  Let T′ be the tree with root a_k, left subtree L′ and right subtree R. Then

$$C(T') = C(L') + C(R) + W$$

$$\text{i.e. } C(T') < C(L) + C(R) + W$$

$$\text{i.e. } < C(T_{0n})$$

- That means ; T′ is optimal than T_{0n}. This contradicts the fact that T_{0n} is an optimal BST. Therefore, L must be an optimal for its elements.
- In the same manner we can obtain optimal tree for R.
- Thus we can obtain the optimal binary search tree by building the optimal subtrees. This ultimately shows that optimal binary search tree follows the principle of optimality.

Step 3 : We will apply following formula for computing each sequence.

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

$$W(i, j) = W[i, j - 1] + p[j] + q[j];$$

$$r[i, j] = k$$

Example 3.7.1 *Consider n = 4 and (q_1, q_2, q_3, q_4) = (do, if, int, while). The values for p's and q's are given as p(1:4) = (3, 3, 1, 1) and q(0:4) = (2, 3, 1, 1, 1). Construct the optimal binary search tree. We will apply following formulae for the computation of W, C and r.* **SPPU : Dec.-08, Marks 8, Dec.-12, Marks 10**

Solution :

$$W_{i, i} = q_i, r_{i, i} = 0, C_{i, i} = 0$$

$$W_{i, i+1} = q_i + q_{(i+1)} + p_{(i+1)}$$

$$r_{i, i+1} = i + 1$$

$$C_{i, i+1} = q_i + q_{(i+1)} + p_{(i+1)}$$

$$W_{i, j} = W_{i, j-1} + p_j + q_j$$

$$r_{i,j} = k$$

$$C_{i,j} = \min_{i<k\leq j} \{C_{(i,k-1)} + C_{k,j}\} + W_{i,j}$$

We will construct tables for values of W, C and r.

Let $i = 0$

$$W_{00} = q_0 = 2$$

When i = 1

$$W_{11} = 3$$

When i = 2

$$W_{22} = 1$$

When i = 3

$$W_{33} = 1$$

When i = 4

$$W_{44} = q_4 = 1$$

When i = 0 and j – i = 1 then

$$W_{01} = q_0 + q_1 + p_1$$
$$= 2 + 3 + 3$$
$$W_{01} = 8$$

When i =1 and j – i = 2

$$W_{12} = q_1 + q_2 + p_2$$
$$= 3 + 1 + 3$$
$$W_{12} = 7$$

When i = 2 and j – i = 3

$$W_{23} = q_2 + q_3 + p_3$$
$$= 1 + 1 + 1$$
$$W_{23} = 3$$

When i = 3 and j – i = 4

$$W_{34} = q_3 + q_4 + p_4$$
$$= 1 + 1 + 1$$

$$W_{34} = 3$$

Now, when i = 0 and j – i = 2

$$W_{ij} = W_{i, j-1} + p_j + q_j$$

$$W_{02} = W_{01} + p_2 + q_2$$

$$= 8 + 3 + 1$$

$$W_{02} = 12$$

When i =1 and j – i = 2

$$W_{13} = W_{12} + p_3 + q_3$$

$$= 7 + 1 + 1$$

$$W_{13} = 9$$

When i = 2 and j – i = 2 then

$$W_{24} = W_{23} + p_4 + q_4$$

$$= 3 + 1 + 1$$

$$W_{24} = 5$$

Now, when i = 0 and j – i = 3 then

$$W_{03} = W_{02} + p_3 + q_3$$

$$= 12 + 1 + 1$$

$$W_{03} = 14$$

When i = 1 and j – i = 3 then

$$W_{14} = W_{13} + q_4 + p_4$$

$$= 9 + 1 + 1$$

$$W_{14} = 11$$

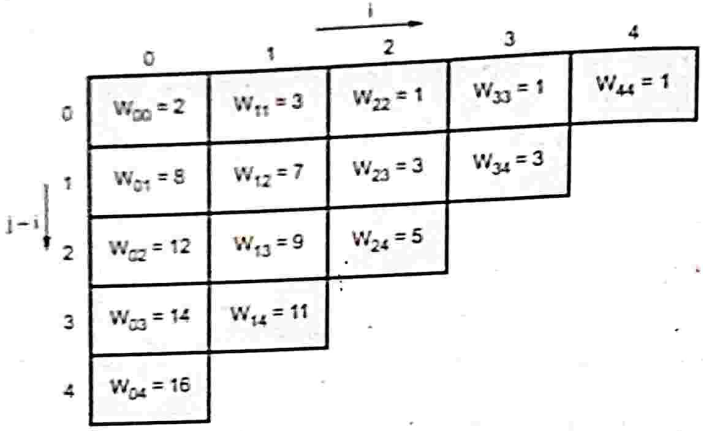
When i = 0 and j – i = 4 then

$$W_{04} = W_{03} + q_4 + p_4$$

$$= 14 + 1 + 1$$

$$W_{04} = 16$$

The table for W can be represented as



We will now compute for C and r.

As $C_{i,i} = 0$ and $r_{i,i} = 0$

$C_{00} = 0$ $C_{11} = 0$ $C_{22} = 0$ $C_{33} = 0$ $C_{44} = 0$

$r_{00} = 0$ $r_{11} = 0$ $r_{22} = 0$ $r_{33} = 0$ $r_{44} = 0$

Similarly, $C_{i,i+1} = q_i + q_{(i+1)} + P_{(i+1)}$ and $r_{i,i+1} = i+1$

∴ When i = 0

$$C_{01} = q_0 + q_1 + P_1$$
$$= 2 + 3 + 3$$
$$C_{01} = 8$$
$$r_{01} = 1$$

When i = 1

$$C_{12} = q_1 + q_2 + P_2$$
$$= 3 + 1 + 3$$
$$C_{12} = 7 \quad \text{and} \quad r_{12} = 2$$

When i = 2

$$C_{23} = q_2 + q_3 + P_3$$
$$= 1 + 1 + 1$$
$$C_{23} = 3 \quad \text{and} \quad r_{23} = 3$$

When $i = 3$

$$C_{34} = q_3 + q_4 + p_4$$
$$= 1 + 1 + 1$$
$$C_{34} = 3 \text{ and } r_{34} = 4$$

Now we will compute C_{ij} and r_{ij} for $j - 1 \geq 2$.

As $$C_{i,j} = \min_{i<k\leq j} \{C_{(i,k-1)} + C_{k,j}\} + W_{i,j}$$

Hence we will find k.

For C_{02} we have i = 0 and j = 2. For $r_{i,j-1}$ to $r_{i+1,j}$ i.e. for r_{01} to $r_{1,2}$. We will compute minimum value of C_{ij}.

Let $r_{01} = 1$ and $r_{12} = 2$. Then we will assume value of k = 1 and will compute C_{ij}. Similarly with k = 2 we will compute C_{ij} and will pick up minimum value of C_{ij} only.

Let us compute C_{ij} with following formula,

$$C_{ij} = C_{i,k-1} + C_{kj}$$

For k = 1, i = 0, j = 2.

$$C_{02} = C_{00} + C_{12}$$
$$C_{02} = 0 + 7$$
$$C_{02} = 7 \quad \text{...(1)}$$

For k = 2, i = 0, j = 2

$$C_{02} = C_{01} + C_{22}$$
$$= 8 + 0$$
$$C_{02} = 8 \quad \text{...(2)}$$

From equations (1) and (2) we can select minimum value of C_{02} is 7. That means k = 1 gives us minimum value of $C_{i,j}$.

Hence $r_{ij} = r_{02} = k = 1$

Now $C_{02} = \min\{C_{(i,k-1)} + C_{kj}\} + W_{ij}$
$= 7 + W_{02}$
$= 7 + 12$
$C_{02} = 19$

Continuing in this fashion we can compute C_{ij} and r_{ij}. It is as given below.

j-i \ i	0	1	2	3	4
0	$C_{00} = 0$ $r_{00} = 0$	$C_{11} = 0$ $r_{11} = 0$	$C_{22} = 0$ $r_{22} = 0$	$C_{33} = 0$ $r_{33} = 0$	$C_{44} = 0$ $r_{44} = 0$
1	$C_{01} = 8$ $r_{01} = 1$	$C_{12} = 7$ $r_{12} = 2$	$C_{23} = 3$ $r_{23} = 3$	$C_{34} = 3$ $r_{34} = 4$	
2	$C_{02} = 19$ $r_{02} = 1$	$C_{13} = 12$ $r_{13} = 2$	$C_{24} = 8$ $r_{24} = 3$		
3	$C_{03} = 25$ $r_{03} = 2$	$C_{14} = 19$ $r_{14} = 2$			
4	$C_{04} = 32$ $r_{04} = 2$				

Therefore T_{04} has a root r_{04}. The value of r_{04} is 2. From (a_1, a_2, a_3, a_4) = (do, if, int, while) a_2 becomes the root node.

$$r_{ij} = k$$

$$r_{04} = 2$$

Then r_{ik-1} becomes left child and r_{kj} becomes the right child. In other words r_{01} becomes the left child and r_{24} becomes right child of r_{ij}. Here $r_{01} = 1$ so a_1 becomes left child of a_2 and $r_{24} = 3$ so a_3 becomes right child of a_2.

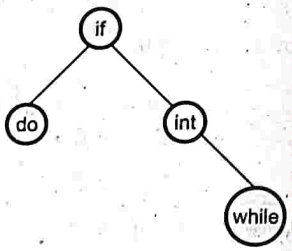


Fig. 3.7.1 OBST

For the node r_{24} i = 2, j = 4 and k = 3. Hence left child of it is $r_{ik-1} = r_{22} = 0$. That means left child of $r_{24} = a_3$ is empty. The right child of r_{24} is $r_{34} = 4$. Hence a_4 becomes right child of a_3. Thus all n = 4 nodes are used to build a tree. The optimal binary search tree is

Example 3.7.2 *What is dynamic programming approach to solve the problem ?*

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 approach. **SPPU : May-11, Marks 16, Aug.-15 (In Sem.), Marks 8**

Solution : For convenience we multiply the probabilities p_L and q_i by 100.

Hence $p_i = (50, 10, 5)$, $q_i = (15, 10, 5, 5)$

$W_{i,i} = q_i$, $r_{i,i} = 0$, $C_{i,i} = 0$

$$W_{i,i+1} = q_i + q_{(i+1)} + p_{(i+1)}$$

$$r_{i,i+1} = i + 1$$

$$C_{i,i+1} = q_i + q_{(i+1)} + p_{(i+1)}$$

$$W_{i,j} = W_{i,j-1} + p_i + q_j$$

$$r_{i,j} = k$$

$$C_{i,j} = i < \overset{\min}{k} \leq j \left\{ C_{(i,k-1)} + C_{k,j} \right\} + W_{i,j}$$

We will construct the tables for values of W, C and r.

Let, i = 0

$$W_{00} = q_0 = 15$$

When i = 1,

$$W_{11} = q_1 = 10$$

When i = 2,

$$W_{22} = q_2 = 5$$

When i = 3,

$$W_{33} = q_3 = 5$$

When i = 0 and j – i = 1 then,

$$W_{01} = q_0 + q_1 + p_1$$

$$= 15 + 10 + 50$$

$$W_{01} = 75$$

When i = 1 and j – i = 2 then,

$$W_{12} = q_1 + q_2 + p_2$$

$$= 10 + 5 + 10$$

$$W_{12} = 25$$

When i = 2 and j – i = 3

$$W_{23} = q_2 + q_3 + p_3$$

$$= 5 + 5 + 5$$

$$W_{23} = 15$$

When i = 0, j – i = 2

$$W_{ij} = W_{i,j-1} + p_j + q_j$$

$$W_{02} = W_{01} + p_2 + q_2$$

$$= 75 + 10 + 5$$

$$W_{02} = 90$$

When $i = 1$, $j - i = 2$

$$W_{13} = W_{12} + p_3 + q_3$$

$$= 25 + 5 + 5$$

$$W_{13} = 35$$

Now, when $i = 0$, $j - i = 3$ then

$$W_{03} = W_{02} + p_3 + q_3$$

$$= 90 + 5 + 5$$

$$W_{03} = 100$$

The table for W can be represented as

$j-i \downarrow$	0	1	2	3
0	$W_{00} = 15$	$W_{11} = 10$	$W_{22} = 5$	$W_{33} = 5$
1	$W_{01} = 75$	$W_{12} = 25$	$W_{23} = 15$	
2	$W_{02} = 90$	$W_{13} = 35$		
3	$W_{03} = 100$			

We will compute for C and r

$$C_{i,i} = 0, r_{i,i} = 0$$

Hence $C_{00} = 0$, $C_{11} = 0$, $C_{22} = 0$, $C_{33} = 0$

$$r_{00} = 0, \quad r_{11} = 0, \quad r_{22} = 0, \quad r_{33} = 0$$

$$C_{i,i+1} = q_i + q_{(i+1)} + p_{(i+1)} \quad \text{and} \quad r_{i,i+1} = i+1$$

$\therefore$ When $i = 0$

$$C_{01} = q_0 + q_1 + p_1$$

$$= 15 + 10 + 50$$

$$= 75$$

$$r_{01} = i + 1 = 1$$

When $i = 1$

$$C_{12} = q_1 + q_2 + p_2$$

$$= 10 + 5 + 5$$

$$C_{12} = 20, \quad r_{12} = 2$$

When $i = 2$

$$C_{23} = q_2 + q_3 + p_3$$
$$= 5 + 5 + 5$$
$$C_{23} = 15, \quad r_{23} = 3$$

Now we will compute C_{ij} and r_{ij} for $j - 1 \geq 2$.

As $C_{ij} = i < \overset{\min}{k} \leq j \left\{ C_{(i,k-1)} + C_{k,j} \right\} + W_{i,j}$

Hence we will find k.

For C_{02} we have i = 0 and j = 2 . For $r_{i,j-1}$ to $r_{i+1,\,j}$ i.e. for r_{01} to r_{12} we will compute minimum value of $C_{i,j}$.

Let, $r_{01} = 1$ and $r_{12} = 2$. Then we will assume k = 1 and k = 2 and then compute $C_{i,j}$. From these computations we will pick-up the minimum value of $C_{i,j}$.

Let,

with k = 1, i = 0, j = 2

$$C_{ij} = C_{i,k-1} + C_{k,j}$$
$$C_{02} = C_{00} + C_{12}$$
$$= 0 + 20$$
$$C_{02} = 20$$

with k = 2, i = 0, j = 2

$$C_{02} = C_{i,k-1} + C_{k,j}$$
$$= C_{01} + C_{22}$$
$$= 75 + 0$$
$$C_{02} = 75$$

Clearly select $C_{02} = 20$ when k = 1.

Now
$$C_{02} = \min\{C_{02}\} + W_{02}$$
$$= 20 + 90$$
$$C_{02} = 110$$

Now for C_{13} we have i = 1, and j = 3.

For $r_{i,j-1}$ to $r_{i+1,j}$ i.e. for r_{12} to r_{23} we will compute minimum value of $C_{i,j}$

Let $r_{12} = 2$ and $r_{23} = 3$. Then we will assume k = 2 and k = 3. then compute $C_{i,j}$

with k = 2, i = 1, j = 3

$$C_{i,j} = C_{i,k-1} + C_{k,j}$$
$$C_{13} = C_{11} + C_{23}$$
$$= 0 + 15$$
$$C_{13} = 15$$

with k = 3, i = 1, j = 3

$$C_{i,j} = C_{i,k-1} + C_{k,j}$$
$$C_{13} = C_{12} + C_{33}$$
$$= 20 + 0$$
$$C_{13} = 20$$

Clearly select $C_{13} = 15$ when k = 2

Now $C_{13} = \min \{C_{13}\} + W_{13}$

$= 15 + 35$

$C_{13} = 50$

Now, For C_{03} we have i = 0, and j = 3

For $r_{i, j-1}$ to $r_{i+1,j}$ i.e. for r_{02} to r_{13} i.e. with k = 1 to k = 2.

with k = 1, i = 0, j = 3

$C_{ij} = C_{i, k-1} + C_{k, j}$

$= C_{00} + C_{13}$

$= 0 + 15$

$C_{03} = 15$

with k = 1, i = 0, j = 3

$C_{i,j} = C_{i, k-1} + C_{k, j}$

$= C_{01} + C_{23}$

$= 75 + 15$

$C_{03} = 90$

Clearly select $C_{03} = 15$ when k = 1

Now $C_{03} = \min \{C_{03}\} + W_{03}$

$= 15 + 100$

$C_{03} = 115$

The C_{ij} and r_{ij} table can be as given below

	0	1	2	3
0	$C_{00} = 0$ $r_{00} = 0$	$C_{11} = 0$ $r_{11} = 0$	$C_{22} = 0$ $r_{22} = 0$	$C_{33} = 0$ $r_{33} = 0$
1	$C_{01} = 75$ $r_{01} = 1$	$C_{12} = 20$ $r_{12} = 2$	$C_{23} = 15$ $r_{23} = 3$	
2	$C_{02} = 110$ $r_{02} = 1$	$C_{13} = 50$ $r_{13} = 2$		
3	$C_{03} = 115$ $r_{03} = 1$			

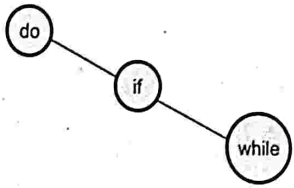
The tree T_{03} has a root r_{03}. The $r_{03} = 1$. That means 01 is the root.

The r_{ik-1} becomes left child and r_{kj} becomes the right child. The value of k = 1.

$\therefore$ $r_{00} = 0$ i.e. empty left child

$r_{13} = 2$ i.e. a_2 is right

Finally the tree will be -



Example 3.7.3 *Consider 4 elements $a_1 < a_2 < a_3 < a_4$ with $q_0 = 0.25$, $q_1 = 3/16$, $q_2 = q_3 = q_4 = 1/16$. $p_1 = 1/4$, $p_2 = 1/8$, $p_3 = p_4 = 1/16$.*

a) Construct the optimal binary search tree as a minimum cost tree.

b) Construct the table of values W_{ij}, C_{ij}, V_{ij} computed by the algorithm to compute the roots of optimal subtrees.

Solution : For convenience we will multiply the probabilities q_i and p_i by 16. Hence p_i and q_i values are $(p_1, p_2, p_3, p_4) = (4, 2, 1, 1)$

$(q_0, q_1, q_2, q_3, q_4) = (4, 3, 1, 1, 1)$

Let us compute values of W first. For W_{ii} computation we will use,

$$W_{ii} = q_i$$

Hence $W_{00} = q_0$

$\therefore W_{00} = 4$

$W_{11} = q_1$

$\therefore W_{11} = 3$

$W_{22} = q_2$

$\therefore W_{22} = 1$

$W_{33} = q_3$

$\therefore W_{33} = 1$

$W_{44} = q_4$

$\therefore W_{44} = 1$

Compute W_{ii+1} using formula

$$W_{ii+1} = q_i + q_{i+1} + p_{i+1}$$

Hence, $W_{01} = q_0 + q_1 + p_1$

$= 4 + 3 + 4$

$\therefore W_{01} = 11$

$$W_{12} = q_1 + q_2 + p_2$$
$$= 3 + 1 + 2$$

$\therefore \quad W_{12} = 6$

$$W_{23} = q_2 + q_3 + p_3$$
$$= 1 + 1 + 1$$

$\therefore \quad W_{23} = 3$

$$W_{34} = q_3 + q_4 + p_4$$
$$= 1 + 1 + 1$$

$\therefore \quad W_{34} = 3$

For computing W_{ij} we will use following formula

$$W_{ij} = W_{ij-1} + p_j + q_j$$

$\therefore \quad W_{02} = W_{01} + p_2 + q_2$
$= 11 + 2 + 1$

$\therefore \quad W_{02} = 14$

$\therefore \quad W_{13} = W_{12} + p_3 + q_3$
$= 6 + 1 + 1$

$\therefore \quad W_{13} = 8$

$\therefore \quad W_{24} = W_{23} + p_4 + q_4$
$= 3 + 1 + 1$

$\therefore \quad W_{24} = 5$

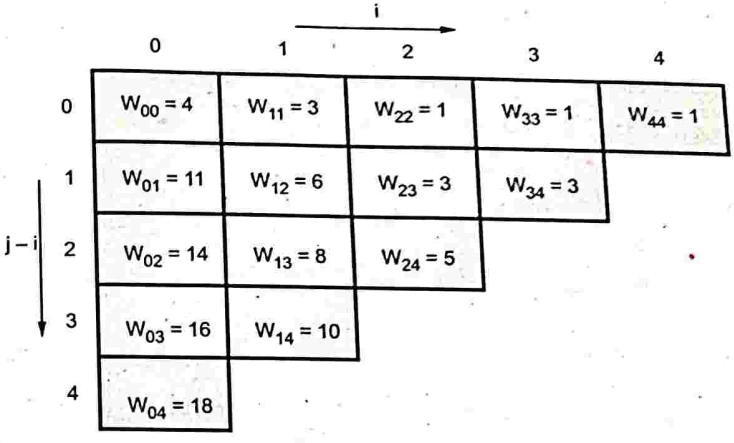
Now, $W_{03} = W_{02} + p_3 + q_3$
$= 14 + 1 + 1$

$\therefore \quad W_{03} = 16$

$W_{04} = W_{03} + p_4 + q_4$
$= 16 + 1 + 1$

$\therefore \quad W_{04} = 18$

To summarize W values



To compute C and r values we will use $C_{ii} = 0$ and $r_{ii} = 0$.

Hence $C_{00} = 0$

$r_{00} = 0$

$C_{11} = 0$

$r_{11} = 0$

$C_{22} = 0$

$r_{22} = 0$

$C_{33} = 0$

$r_{33} = 0$

$C_{44} = 0$

$r_{44} = 0$

To compute C_{ii+1} and r_{ii+1} we will use following formulae

$$r_{ii+1} = i + 1$$

$$C_{ii+1} = q_i + q_{i+1} + p_{i+1}$$

Hence $r_{01} = 1$

$C_{01} = q_0 + q_1 + p_1$

$= 4 + 3 + 4$

$\therefore C_{01} = 11$

$r_{12} = 2$

$$C_{12} = q_1 + q_2 + p_2$$
$$= 3 + 1 + 2$$

$\therefore$ $C_{12} = 6$

$r_{23} = 3$

$$C_{23} = q_2 + q_3 + p_3$$
$$= 1 + 1 + 1$$

$\therefore$ $C_{23} = 3$

$r_{34} = 4$

$$C_{34} = q_3 + q_4 + p_4$$
$$= 1 + 1 + 1$$

$\therefore$ $C_{34} = 3$

To compute C_{ij} and r_{ij} we will compute the values of k first.

The value of k lies between values of $r_{i, j-1}$ to $r_{i+1, j}$. The min $\{C_{i\,k-1} + C_{kj}\}$ decides value of k.

Consider i = 0 and j = 2.

To decide value of k, the range for k is $r_{01} = 1$ to $r_{12} = 2$ when k = 1 and i = 0, j = 2. We will compute C_{ij} using formula

$$C_{ij} = C_{ik-1} + C_{kj}$$

$\therefore$ $C_{02} = C_{00} + C_{02}$

$C_{02} = 6 \rightarrow$ Minimum value of C_{02}.

When k = 2, and i = 0, j = 2.

$$C_{02} = C_{01} + C_{22}$$
$$C_{02} = 11 + 0$$

$\therefore$ $C_{02} = 11$

In above computations we get minimum value of C_{02} when k = 1. Hence the value of k becomes 1.

As $r_{ij} = k$

$r_{02} = 1$

For $C_{ij} = W_{ij} + \min \{C_{i, k-1} + C_{kj}\}$

$\therefore$ $C_{02} = W_{02} + C_{02}$

$C_{02} = 14 + 6 = 20$

Now for r_{13} and C_{13}

k is between $r_{12} = 2$ to $r_{23} = 3$ then when k = 2 and i = 1, j = 3.

$$C_{13} = C_{11} + C_{23}$$
$$= 0 + 3$$

$\therefore$ $C_{13} = 3 \rightarrow$ Minimum value of C_{13}

When k = 3 and i = 1, j = 3.

$$C_{13} = C_{12} + C_{33}$$
$$= 6 + 0$$

$\therefore$ $C_{13} = 6$

Hence k = 2 gives minimum value of C_{13}.

$\therefore$ $r_{13} = 2$

and $C_{13} = W_{13} + C_{13}$ (Minimum value)

$$= 8 + 3$$

$\therefore$ $C_{13} = 11$

Now for r_{24} and C_{24}

k is between $r_{23} = 3$ to $r_{34} = 4$.

$\therefore$ i = 2, j= 4 when k = 3.

$$C_{24} = C_{22} + C_{34}$$
$$= 0 + 3$$

$\therefore$ $C_{24} = 3$

When k = 4, i = 2, j = 4.

$$C_{24} = C_{23} + C_{44}$$
$$= 3 + 0$$

$\therefore$ $C_{24} = 3$

That means value of k can be 3. Let us consider k = 3. Then,

$$r_{24} = 3$$

and $C_{34} = W_{34} + C_{34}$

$$= 5 + 3$$

$\therefore$ $C_{34} = 8$

Now for r_{03} and C_{03}

Value of k lies between $r_{02} = 1$ to $r_{13} = 2$

When k = 1 and i = 0, j = 3.

$$C_{03} = C_{00} + C_{13}$$
$$= 0 + 11$$

$\therefore$ $C_{03} = 11$ $\rightarrow$ Minimum value of C_{03}.

When k = 2 and i = 0, j = 3.

$$C_{03} = C_{01} + C_{23}$$
$$= 11 + 3$$

$\therefore$ $C_{03} = 14$

Hence value of k = 1.

$\therefore$ $r_{03} = 1$

and $C_{03} = W_{03} + C_{03}$
$= 16 + 11$

$\therefore$ $C_{03} = 27$

Now for r_{14} and C_{14}

Value of k is between $r_{13} = 2$ to $r_{24} = 3$.

When k = 2 and i = 1, j = 4 then

$$C_{14} = C_{11} + C_{24}$$
$$= 0 + 8$$

$\therefore$ $C_{14} = 8$ $\rightarrow$ Minimum value of C_{14}

When k = 3 and i = 1, j = 4 then

$$C_{14} = C_{12} + C_{34}$$
$$= 6 + 3$$

$\therefore$ $C_{14} = 9$

Hence value of k = 2

$\therefore$ $r_{14} = 2$

and $C_{14} = W_{14} + C_{14}$

$$= 10 + 8$$

$$C_{14} = 18$$

Now for computing r_{04} and C_{04}

Value of k is between $r_{03} = 1$ to $r_{14} = 2$ then,

When k = 1 and i = 0, j = 4.

$$C_{04} = C_{00} + C_{14}$$

$$= 0 + 18$$

$\therefore$ $C_{04} = 18 \rightarrow$ Minimum value of C_{04}

When k = 2 and i = 0, j = 4.

$$C_{04} = C_{01} + C_{24}$$

$$= 11 + 8$$

$\therefore$ $C_{04} = 19$

Hence value of k = 1

$$r_{04} = 1$$

$\therefore$ $C_{04} = W_{04} + C_{04}$

$$= 18 + 18$$

$\therefore$ $C_{04} = 36$

To summarize

i →

j − i	0	1	2	3	4
0	$C_{00} = 0$ $r_{00} = 0$	$C_{11} = 0$ $r_{11} = 0$	$C_{22} = 0$ $r_{22} = 0$	$C_{33} = 0$ $r_{33} = 0$	$C_{44} = 0$ $r_{44} = 0$
1	$C_{01} = 11$ $r_{01} = 1$	$C_{12} = 6$ $r_{12} = 2$	$C_{23} = 3$ $r_{23} = 3$	$C_{34} = 3$ $r_{34} = 4$	
2	$C_{02} = 20$ $r_{02} = 1$	$C_{13} = 11$ $r_{13} = 2$	$C_{24} = 8$ $r_{24} = 3$		
3	$C_{03} = 27$ $r_{03} = 1$	$C_{14} = 18$ $r_{14} = 2$			
4	$C_{04} = 36$ $r_{04} = 1$				

To build the OBST

$$r_{04} = 1$$

Hence a_1 becomes root node

To compute children of a_1 we will apply following formula

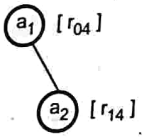
For node r_{ij} = k then

left child is r_{ik-1} and right child is r_{kj}

Hence for r_{04} = 1, for node a_1

i = 0, j = 4 and k = 1.

Left child = r_{00} = 0. That is empty node.



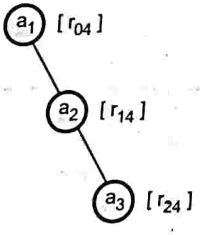
Right child = r_{14}= 2. That means a_2 is right child of a_1.

For node r_{14} = 2 for node a_2

i = 1, j = 4 and k = 2.

Left child = r_{11} = 0. That is empty node.

Right child = r_{24}= 3. That means a_3 is right child of a_2.



For node r_{24} = 3 i.e. for node a_3

i = 2, j = 4, and k = 3.

Left child = r_{22} = 0. That means no left child.

Right child = r_{34}= 4. That means a_4 is right child of node a_3.

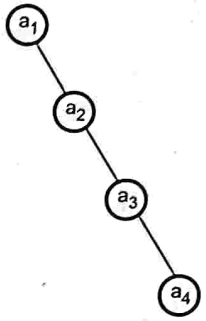


Fig. 3.7.2 OBST

is the required OBST.

Example 3.7.4 *Using algorithm OBST compute W(i, j), R(i, j) and C(i, j) $0 \le i \le j \le 4$, for the identifier set (a_1, a_2, a_3, a_4) = (end, goto, print, stop) with P(1) = 1/20, P(2) = 1/5, P(3) = 1/10, P(4) = 1/20. Q(0) = 1/5, Q(1) = 1/10, Q(2) = 1/5, Q(3) = 1/20, Q(4) = 1/20. Using R(i, j) construct the optimal binary search tree.*

Solution : We will multiply values of P's and Q's by 20, for convenience. Hence,

P(1) = 1, P(2) = 4, P(3) = 2, P(4) = 1. Q(0) = 4, Q(1) = 2, Q(2) = 4, Q(3) = 1, Q(4) = 1

Now we will compute,

$W_{i,i} = q_i$, $R_{ii} = 0$, $C_{ii} = 0$

$$W_{i,i+1} = q_i + q_{(i+1)} + P_{(i+1)}$$

$$R_{i,i+1} = i + 1$$

$$C_{i,i+1} = q_i + q_{(i+1)} + P_{(i+1)}$$

Let i = 0,

$$W_{00} = q_0 = 4$$

$$W_{11} = q_1 = 2$$

$$W_{22} = q_2 = 4$$

$$W_{33} = q_3 = 1$$

$$W_{44} = q_4 = 1$$

$$W_{01} = q_0 + q_1 + P_1$$

$$= 4 + 2 + 1$$

$$W_{01} = 7$$

$$W_{12} = q_1 + q_2 + p_2$$
$$= 2 + 4 + 4$$
$$W_{12} = 10$$
$$W_{23} = q_2 + q_3 + p_3$$
$$= 4 + 1 + 2$$
$$W_{23} = 7$$
$$W_{34} = q_3 + q_4 + p_4$$
$$= 1 + 1 + 1$$
$$W_{34} = 3$$

Now we will use formula

$$W_{ij} = W_{ij-1} + p_j + q_j$$

Hence, let i = 0, j= 2 then

$$W_{02} = W_{01} + p_2 + q_2$$
$$= 7 + 4 + 4$$
$$W_{02} = 15$$

Let i = 1, j = 3

$$W_{13} = W_{12} + p_3 + q_3$$
$$= 10 + 2 + 1$$
$$W_{13} = 13$$

Let i = 2, j = 4

$$W_{24} = W_{23} + p_4 + q_4$$
$$= 7 + 1 + 1$$
$$W_{24} = 9$$

Let i = 0, j = 3

$$W_{03} = W_{02} + p_3 + q_3$$
$$= 15 + 2 + 1$$
$$W_{02} = 18$$

Let i = 1, j = 4

$$W_{14} = W_{13} + p_4 + q_4$$

$$= 13 + 1 + 1$$

$$W_{14} = 15$$

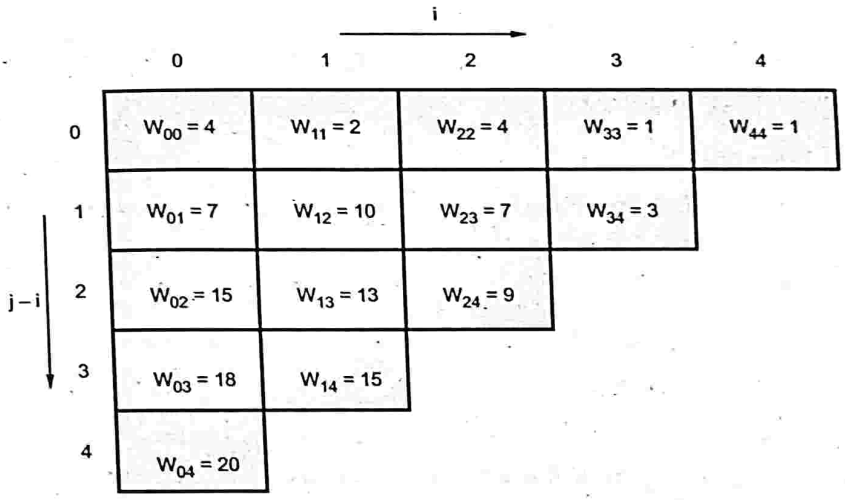
Let i = 0, j = 4

$$W_{04} = W_{03} + p_4 + q_4$$

$$= 18 + 1 + 1$$

$$W_{04} = 20$$

The tabular representation for values of W are as shown below.



The computation of C and R values is done as below.

$$C_{00} = 0$$

$$R_{00} = 0$$

$$C_{11} = 0$$

$$R_{11} = 0$$

$$C_{22} = 0$$

$$R_{22} = 0$$

$$C_{33} = 0$$

$$R_{33} = 0$$

$$C_{44} = 0$$

$$R_{44} = 0$$

For computation of $C_{i, i+1}$ and $R_{i, i+1}$

We will use following formula

$$C_{i, i+1} = q_i + q_{(i+1)} + P_{(i+1)}$$

$$R_{i, i+1} = i + 1$$

Let i = 0 then

$$C_{01} = q_0 + q_1 + P_1 \qquad R_{01} = 1$$

$$= 4 + 2 + 1$$

$$C_{01} = 7$$

$$C_{12} = q_1 + q_2 + P_2 \qquad R_{12} = 2$$

$$= 2 + 4 + 4$$

$$C_{12} = 10$$

$$C_{23} = q_2 + q_3 + P_3$$

$$= 4 + 1 + 2$$

$$C_{23} = 7 \qquad R_{23} = 3$$

$$C_{34} = q_3 + q_4 + P_4$$

$$= 1 + 1 + 1$$

$$C_{34} = 3 \qquad R_{34} = 4$$

To compute C_{ij} and R_{ij} we will use following formulae.

$$C_{ij} = \min_{i<k\le j} \{C(i, k-1) + C_{kj}\} + W_{ij}$$

$$R_{ij} = k$$

Value of k lies between $R_{i, j-1}$ and $R_{(i+1), j}$. Let us compute C_{02}.

Here i = 0 and j = 2. The value of k lies between $R_{01} = 1$ to $R_{12} = 2$.

When k = 1, i = 0, j = 2.

$$C_{02} = C_{00} + C_{12}$$

$$C_{02} = 0 + 10$$

$$C_{02} = 10$$

When k = 2, i = 0, j = 2.

$$C_{02} = C_{01} + C_{22}$$

$$C_{02} = 7 + 0$$

$$C_{02} = 7 \rightarrow \text{Minimum value of C.}$$

We will select k = 2, as we obtain C value as minimum when k = 2. Hence $R_{02} = 2$ then $C_{02} = W_{02} + C_{02} = 15 + 7 = 22$.

Let us compute C_{13} the range for k is $R_{12} = 2$ to $R_{23} = 3$.

When k = 2, i = 1, j = 3.

$$C_{13} = C_{11} + C_{23}$$
$$= 0 + 7$$
$$C_{13} = 7 \rightarrow \text{Minimum value}$$

When k = 3, i = 1, j = 3

$$C_{13} = C_{12} + C_{33}$$
$$= 10 + 0$$
$$C_{13} = 10$$

As we obtain minimum value of C_{13} with k = 2. Hence value of k becomes 2.

$$\therefore \quad R_{13} = 2$$
$$C_{13} = W_{13} + \min \{ C_{13} \}$$
$$= 13 + 7$$
$$\therefore \quad C_{13} = 20$$

Now we will compute C_{24}.

i = 2, j = 4.

Value of k lies between $R_{23} = 3$ to $R_{34} = 4$. When k = 3, i = 2 and j = 4.

$$C_{24} = C_{22} + C_{34}$$
$$= 0 + 3$$
$$C_{24} = 3 \rightarrow \text{Minimum value}$$

When k = 4, i = 2, j = 4

$$C_{24} = C_{23} + C_{44}$$
$$= 7 + 0$$
$$C_{24} = 7$$

Hence we set k = 3.

$$\therefore \quad R_{24} = 3$$
$$C_{24} = W_{24} + \min \{C_{24}\}$$
$$= 9 + 3$$

$\therefore$ $C_{24} = 12$

To compute C_{03}, $i = 0$, $j = 3$.

k lies between $R_{02} = 2$ to $R_{13} = 2$.

That means $k = 2$. $R_{03} = 2$.

$\therefore$ $C_{03} = \{C_{01} + C_{23}\} + W_{03}$

$= 7 + 7 + 18$

$\therefore$ $C_{03} = 32$

To compute C_{14}, $i = 1$, $j = 4$.

k lies between $R_{13} = 2$ to $R_{24} = 3$.

When $k = 2$, $i = 1$, $j = 4$

$C_{14} = C_{11} + C_{24}$

$= 0 + 12$

$\therefore$ $C_{14} = 12 \rightarrow$ Minimum value

When $k = 3$, $i = 1$, $j = 4$

$C_{14} = C_{12} + C_{34}$

$= 10 + 3$

$\therefore$ $C_{14} = 13$

We will set $k = 2$

$\therefore$ $R_{14} = 2$

$C_{14} = W_{14} + \min \{C_{14}\}$

$= 15 + 12$

$\therefore$ $C_{14} = 27$

To compute C_{04}

$i = 0$, $j = 4$

k lies between $R_{03} = 2$ to $R_{14} = 2$

That means $k = 2$ $\quad \therefore R_{04} = 2$

$\therefore$ $C_{04} = \{C_{01} + C_{24}\} + W_{04}$

$C_{04} = \{7 + 12\} + 20$

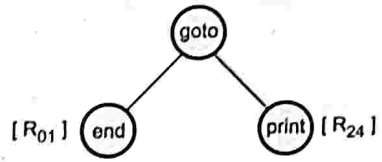
$\therefore$ $C_{04} = 39$

The values are summarized as below.

	0	1	2	3	4
0	$C_{00} = 0$ $R_{00} = 0$	$C_{11} = 0$ $R_{11} = 0$	$C_{22} = 0$ $R_{22} = 0$	$C_{33} = 0$ $R_{33} = 0$	$C_{44} = 0$ $R_{44} = 0$
1	$C_{01} = 7$ $R_{01} = 1$	$C_{12} = 10$ $R_{12} = 2$	$C_{23} = 7$ $R_{23} = 3$	$C_{34} = 3$ $R_{34} = 4$	
2	$C_{02} = 22$ $R_{02} = 2$	$C_{13} = 20$ $R_{13} = 2$	$C_{24} = 12$ $R_{24} = 3$		
3	$C_{03} = 32$ $R_{03} = 2$	$C_{14} = 27$ $R_{14} = 2$			
4	$C_{04} = 39$ $R_{04} = 2$				

Let us build tree using R_{ij}.

As $R_{04} = 2$, a_2 will be root.



Left child of $R_{ij} = k$ is R_{ik-1} i.e. R_{01}. As $R_{01} = 1$, a_1 becomes left child. Right child of $R_{ij} = k$ is R_{Kj} i.e. $R_{24} = 3$, a_3 becomes right child.

For $R_{01} = 1$ left child is $R_{00} = 0$. That means there is no left child to node 'end'. For $R_{01} = 1$ right child is $R_{11} = 0$. That means there is no right child to node 'end'. Consider $R_{24} = 3$, left child is $R_{22} = 0$. That means there is no left child to node 'print'. For $R_{24} = 3$, right child is $R_{34} = 4$. That means a_4 = stop becomes right child of node 'print'.

The final OBST is

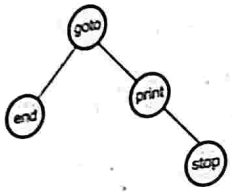


Fig. 3.7.3 OBST

3.7.1 Algorithm

```
Algorithm Wt(p,q,W,n)
//p is an input array [1..n]
//q is an input array [0..n]
// W[i,j] will be output for this function such that W[0..n,0..n]
{
  for i←1 to n do
    W[i,i] ← q[i]; // initialize
  for m←1 to n do
  {
   for I←0 to n–m do
    {
      k ← i+m;
      W[i,k]←W[i,k–1] + p[k] + q[k];
    }
  }
write(W[0:n]);//print the values of W
}
```

- For computing C_{ij}, r_{ij} following algorithm is used.

```
Algorithm OBST(p,q,W,C,r)
{
  //computation of first two rows
      for i=0 to n do
            {
                  C[i,i] ← 0;
                   r[i,i] ← 0;
                  C[i,i+1] ← q[i]+q[i+1]+p[i+1];
                   r[i,i+1]←i+1;
            }
            for m ← 2 to n do
            {
                  For ← 0 to n-m do
            {
                  j←i+m;
                  k←Min_Value(C,r,i,j); //call for algortihm Min_Value
```

```
            // minimum value of C_ij is obtained for deciding value of k
            C[i,j]←W[i,j] + C[i,k–1] + C[k,j];
            r[i,j]←k;
        }
        }
        write(C[0:n],r[0:n]);//print values of C_ij and r_ij
        }
Algorithm Min_Value(C,r,i,j)
{
    minimum← ∞;
// finding the range of k
    for (m←r[i,j-1] to r[i+1,j]) do
    {
        if(C[i,m-1]+C[m,j])< minimum then
        {
            minimum ← C[i,m-1]+C[m,j];
            k←m;
        }
        return k;//This k gives minimum value of C
    }
}
```

Following algorithm is used for creating the tree T_{ij}.

```
Algorithm build_tree(r,a,i,j)
{
  T ← new(node);        //allocate memory for a new node
  k ← r[i,j];
  T -> data ← a[k];
  if (j==i+1)
     return;
  T->left=build_tree(r,a,i,k–1);
  T->right=build_tree(r,a,k,j);
}
```

In order to obtain the optimal binary search tree we will follow :

1. Compute the weight using Wt function.
2. Compute C_{ij} and r_{ij} using OBST.
3. Build the tree using build_tree function.

Analysis

In given algorithm, the basic operation is computation of C[i, j]. In this computation $j \leftarrow i + m$ i.e. $j - i = m$. The inner for loop executes for $n - m + 1$ times to compute C[i, j]. Inside this nested for loop a function Min-value is called. The purpose of this function is to compute minimum value of C array. The time complexity of this computation is **O(m)**.

Hence total time required by the algorithm to execute is $O(nm - m^2)$ which out to be $O(n^3)$. But D.E. Knuth has shown that if value of k is searched within the of r [i, j, –1] to r[i+1, j] then the time complexity of above algorithm will be $O(n^2)$.

Review Questions

1. *Write an algorithm for finding minimum cost binary search tree sing dynamic programm strategy. Show that the computing time of this algorithm is $O(n^2)$.* **SPPU : Dec.-06, May-07, 09, Marks**
2. *What is memory function ? Explain why it is advantageous to use memory functions.* **SPPU : Dec.-06, May-14, Marks**
3. *Write an algorithm for optimum binary search tree.* **SPPU : Dec.-15 (End Sem.), Marks**

3.8 The 0/1 Knapsack

SPPU : Dec.-07,14,11,12,15, April-18, May-07,08,14, Marks

Problem Description : If we are given n objects and a Knapsack or a bag in w the object i that has weight w_i is to be placed. The Knapsack has a capacity W. Then profit that can be earned is $p_i x_i$. The objective is to obtain filling of Knapsack maximum profit earned.

$$\text{maximized } \sum_{n}^{1} p_i x_i \text{ subject to constraint } \sum_{n}^{1} w_i x_i \leq W$$

where $1 \leq i \leq n$ and n is total number of objects. And $x_i = 0$ or 1

To solve Knapsack problem using dynamic programming we will write recurrence relation as :

$$\text{table [i, j]} = \text{maximum } \{\text{table}[i-1, j], v_i + \text{table}[i-1, j-w_i]\} \text{ if } j \geq w_i$$

or

$$\text{table [i – 1, j] if } j < w_i$$

That means, a table is constructed using above given formula. Initially, table [0, j] = 0 as well as table [i, 0] = 0 when $j \geq 0$ and $i \geq 0$.

The initial stage of the table can be

	0	1		$j-w_i$		j		W
0	0	0	...	0	...	0	...	0
	:			table [i−1, j−w_i]				
i − 1	0					table [i−1, j]		
i	0					table [i, j]		
n	: 0							table [n, W]

Goal i.e., maximum value of items

The table [n, W] is a goal i.e., its gives the total items sum of all the selected items for the knapsack.

From this goal value the selected items can be traced out. Let us solve the knapsack problem using the above mentioned formula -

Example 3.8.1 *For the given instance of problem obtain the optimal solution for the knapsack problem.*

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	6

The capacity of knapsack is W = 5. **SPPU : Dec.-15 (End Sem.) Marks 6**

Solution : Initially, table [0, j] = 0 and table [i, 0] = 0. There are 0 to n rows and 0 to W columns in the table.

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

Now we will fill up the table either row by row or column by column. Let us start filling the table row by row using following formula :

$$\text{table}[i, j] = \begin{cases} \text{maximum}\left\{\text{table}[i-1, j], v_i + \text{table}[i-1, j-w_i]\right\} \text{ when } j \geq w_i \\ \text{or} \\ \text{table}[i-1, j] \text{ if } j < w_i \end{cases}$$

table [1, 1] With $i = 1, j = 1, w_i = 2$ and $v_i = 3$.

As $j < w_i$ we will obtain table [1, 1] as

$$\text{table } [1, 1] = \text{table } [i - 1, j] = \text{table } [0, 1]$$

$\therefore$ table [1, 1] = 0

table [1, 2] With $i = 1, j = 2, w_i = 2$ and $v_i = 3$

As $j \geq w_i$ we will obtain table [1, 2] as

$$\text{table } [1, 2] = \text{maximum } \{\text{table}[i-1, j], v_i + \text{table}[i-1, j-w_i]\}$$

$$= \text{maximum } \{(\text{table}[0, 2]), (3 + \text{table}[0, 0])\} = \text{maximum } \{0, 3 + 0\}$$

$\therefore$ table [1, 2] = 3

table [1, 3] With $i = 1, j = 3, w_i = 2$ and $v_i = 3$

As $j \geq w_i$ we will obtain table [1, 3] as

$$\text{table } [1, 3] = \text{maximum } \{\text{table}[i-1, j], v_i + \text{table}[i-1, j-w_i]\}$$

$$= \text{maximum } \{\text{table } [0, 3], 3 + \text{table}[0, 1]\} = \text{maximum } \{0, 3 + 0\}$$

$\therefore$ table [1, 3] = 3

table [1, 4] With $i = 1, j = 4, w_i = 2$ and $v_i = 3$

As $j \geq w_i$, we will obtain table [1, 4] as

$$\text{table}[1,4] = \text{maximum}\{\text{table}[i-1,j], v_i + \text{table}[i-1, j-w_i]\}$$

$$= \text{maximum}\{\text{table}[0,4], 3 + \text{table}[0,2]\} = \text{maximum}\{0, 3+0\}$$

$\therefore$ table [1, 4] = 3

table [1, 5] With i = 1, j = 5, $w_i = 2$ and $v_i = 3$

As $j \geq w_i$ we will obtain table [1, 5] as

$$\text{table}[1,5] = \text{maximum}\{\text{table}[i-1,j], v_i + \text{table}[i-1, j-w_i]\}$$

$$= \text{maximum}\{\text{table}[0,5], 3 + \text{table}[0,3]\}$$

$$= \text{maximum}\{0, 3+0\}$$

$\therefore$ table [1, 5] = 3

The table with these values can be

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

Now let us fill up next row of the table.

table [2, 1] With i = 2, j = 1, $w_i = 3$ and $v_i = 4$

As $j < w_i$, we will obtain table [2, 1] as

$$\text{table}[2,1] = \text{table}[i-1, j]$$

$$= \text{table}[1,1]$$

$\therefore$ table [2, 1] = 0

table [2, 2] With i = 2, j = 2, $w_i = 3$ and $v_i = 4$

As $j < w_i$, we will obtain table [2, 2] as

$$\text{table}[2,2] = \text{table}[i-1, j]$$

$$= \text{table}[1,2]$$

$\therefore$ table [2; 2] = 3

table [2, 3] With i = 2, j = 3, $w_i = 3$ and $v_i = 4$

As $j \geq w_i$, we will obtain table [2, 3] as

table [2, 3] = maximum $\{table[i-1, j], v_i + table[i-1, j-w_i]\}$

= maximum {table [1, 3], 4 + table [1, 0]}

= maximum {3, 4 + 0}

$\therefore$ table [2, 3] = 4

table [2, 4] With i = 2, j = 4, $w_i = 3$ and $v_i = 4$

As $j \geq w_i$, we will obtain table [2, 4] as

table [2, 4] = maximum $\{table[i-1, j], v_i + table[i-1, j-w_i]\}$

= maximum {table [1, 4], 4 + table [1, 1]} = maximum {3, 4 + 0}

$\therefore$ table [2, 4] = 4

table [2, 5] With i = 2, j = 5, $w_i = 3$ and $v_i = 4$

As $j \geq w_i$, we will obtain table [2, 5] as

table [2, 5] = maximum $\{table[i-1, j], v_i + table[i-1, j-w_i]\}$

= maximum {table [1, 5], 4 + table [1, 2]} = maximum {3, 4 + 3}

$\therefore$ table [2, 5] = 7

The table with these computed values will be

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0					
4	0					

table [3, 1] With i = 3, j = 1, $w_i = 4$ and $v_i = 5$

As $j < w_i$, we will obtain table [3, 1] as

table [3, 1] = table [i - 1, j] = table [2, 1]

$\therefore$ table [3, 1] = 0

table [3, 2] With i = 3, j = 2, $w_i = 4$ and $v_i = 5$

As $j < w_i$, we will obtain table [3, 2] as

table [3, 2] = table [i − 1, j] = table [2, 2]

∴ table [3, 2] = 3

table [3, 3] with i = 3, j = 3, $w_i = 4$ and $v_i = 5$

As $j < w_i$, we will obtain table [3, 3] as

table [3, 3] = table [i − 1, j] = table [2, 3]

∴ table [3, 3] = 4

table [3, 4] With i = 3, j = 4, $w_i = 4$ and $v_i = 5$

As $j \leq w_i$, we will obtain table [3, 4] as

$$\text{table [3, 4]} = \text{maximum } \{\text{table}[i-1, j], v_i + \text{table}[i-1, j-w_i]\}$$

= maximum {table [2, 4], 5 + table [2, 0] } = maximum {4, 5 + 0}

∴ table [3, 4] = 5

table [3, 5] With i = 3, j = 5, $w_i = 4$ and $v_i = 5$

As $j \geq w_i$, we will obtain table [3, 5] as

$$\text{table [3, 5]} = \text{maximum } \{\text{table}[i-1, j], v_i + \text{table}[i-1, j-w_i]\}$$

= maximum {table [2, 5], 5 + table [2, 1] } = maximum {7, 5 + 0}

∴ table [3, 5] = 7

The table with these values can be

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0					

table [4, 1] With i = 4, j = 1, $w_i = 5$, $v_i = 6$

As $j < w_i$, we will obtain table [4, 1] as

table [4, 1] = table [i − 1, j] = table [3, 1]

∴ table [4, 1] = 0

table [4, 2] With i = 4, j = 2, $w_i = 5$ and $v_i = 6$

As $j < w_i$, we will obtain table [4, 2] as

table [4, 2] = table [i – 1, j] = table [3, 2]

∴ table [4, 2] = 3

table [4, 3] With i = 4, j = 3, $w_i = 5$ and $v_i = 6$

As $j < w_i$, we will obtain table [4, 3] as

table [4, 3] = table [i – 1, j] = table [3, 3]

∴ table [4, 3] = 4

table [4, 4] with i = 4, j = 4, $w_i = 5$ and $v_i = 6$

As $j < w_i$, we will obtain table [4, 4] as

table [4, 4] = table [i – 1, j]

= table [3, 4]

∴ table [4, 4] = 5

table [4, 5] With i = 4, j = 5, $w_i = 5$ and $v_i = 6$

As $j \geq w_i$, we will obtain table [4, 5] as

table [4, 5] = maximum $\{\text{table}[i-1, j], v_i + \text{table}[i-1, j-w_i]\}$

= maximum {table [3, 5] , 6 + table [3, 0]}

= maximum {7, 6 + 0}

∴ table [4, 5] = 7

Thus the table can be finally as given below

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	(7)

This is the total value of selected items

How to find actual knapsack items ?

Now, as we know that table [n, W] is the total value of selected items, that can be placed in the knapsack. Following steps are used repeatedly to select actual knapsack item

```
Let, i = n and k = W then
while (i>0 and k>0)
{
        if(table [i,k] ≠ table[i-1,k])then
        mark i^th item as in knapsack
        i = i-1 and k=k-w_i              //selection of i^th item
    else
        i = i-1  //do not select i^th item
}
```

Let us apply these steps to the problem given in example 3.8.1. As we have obtained the final table -

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

i = 4 and k = 5

i.e., table [4, 5] = table [3, 5]

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

∴ do not select i^{th} i.e., 4^{th} item.

Now set $i = i - 1$

$i = 3$

As table [i, k] = table [i – 1, k]

items ↓ / Capacity →	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

i.e., table [3, 5] = table [2, 5]

do not select i^{th} item i.e., 3^{rd} item.

Now set i = i – 1 = 2

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
✓2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

As table [i, k] ≠ table [i – 1, k]

i.e. table [2, 5] ≠ table [1, 5]

select i^{th} item.

That is, select 2^{nd} item.

Set i = i – 1 and k = k – w_i.

i.e. i = 1 and k = 5 – 3 = 2

	0	1	2	3	4	5
0	0	0	0	0	0	0
✓1	0	0	3	3	3	3
✓2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

As table [i, k] ≠ table [i – 1, k]

i.e. table [1, 2] ≠ table [0, 2]

select i^{th} item.

That is select 1^{st} item.

Set i = i – 1 and k = k – w_i

i.e. i = 0 and k = 2 – 2 = 0

Thus we have selected **item 1** and **item 2** for the knapsack. This solution can also be represented by solution vector **(1, 1, 0, 0).**

Example 3.8.2 *Consider 0/1 knapsack problem N = 3, w = (4, 6, 8), p = (10, 12, 15) using dynamic programming devise the recurrence relations for the problem and solve the same. Determine the optimal profit for the knapsack of capacity 10.*

SPPU : Dec.-11, 12, Marks 10

Solution : The recurrence relation would be

$$\text{table}[i,j] = \begin{cases} \max\{\text{table}[i-1,j], p_i + \text{table}[i-1, j-w_i]\} \text{ when } j \geq w_i \\ \text{or} \\ \text{table}[i-1,j], \text{ if } j < w_i \end{cases}$$

initially table [0, j] = 0 and table [i, 0] = 0.

There will be 0 to n rows and 0 to W columns in table.

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0										
2	0										
3	0										

Now we will fill up table row by row

table [1,1] with i = 1, j = 1, w_i = 4, p_i = 10

As j < w_i

table [1,1] = table [i–1, j]

= table [0, 1]

∴ **table [1, 1] = 0**

table $[1,2]$ with $i = 1$, $j = 2$, $w_i = 4$, $p_i = 10$

As $j < w_i$

$$\text{table}[1,2] = \text{table}[i-1, j]$$
$$= \text{table}[0,2]$$
$$\textbf{table}[1,2] = 0$$

Thus table $[1,3] = 0$

Now table $[1,4]$ with $i = 1$, $j = 4$, $w_i = 4$, $p_i = 10$ is

As $j = w_i$.

$$\text{table}[i,j] = \max\{\text{table}[i-1,j], p_i + \text{table}[i-1, j-w_i]\}$$
$$\text{table}[1,4] = \max\{\text{table}[0,4], 10 + \text{table}[0,0]\}$$
$$\textbf{table}[1,4] = 10$$

table $[1,5]$ with $i = 1$, $j = 5$, $w_i = 4$, $p_i = 10$.

$$\text{table}[i,j] = \max\{\text{table}[i-1,j], p_i + \text{table}[i-1, j-w_i]$$
$$= \max(\text{table}[0,5], 10 + \text{table}[0,1])$$
$$\text{table}[1,5] = 10$$

Similarly table $[1,6]$ = table $[1,7]$ = table $[1,8]$ = table $[1,9]$ = table $[1,10] = 10$

The table will then be

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	10	10	10	10	10	10	10
2											
3											

table $[2,1]$ with $i = 2$, $j = 1$, $w_2 = 6$, $p_2 = 12$

As $j < w_2$

$$\text{table}[2,1] = \text{table}[1,1]$$
$$\textbf{table}[2,1] = 0$$

Similarly table $[2,2]$ = table $[2,3]$ = table $[2,4]$ = table $[2,5] = 0$

Now table $[2,6]$ with $i = 2$, $j = 6$, $w_2 = 6$, $p_2 = 12$.

As $j = w_2 = 6$

$$\text{table}[2,6] = \max\{\text{table}[i-1,j], p_i + \text{table}[i-1, j-w_i]\}$$

$$= \max\{\text{table}[1,6], 12 + \text{table}[1,0]\}$$

$$\text{table}[2,6] = 12$$

Similarly **table [2, 7] = table [2, 8] = table [2, 9] = 12**

Now with table [2, 10], i = 2, j = 10, $w_2 = 6$, $p_2 = 12$.

$$\text{table}[2,10] = \max\{\text{table}[i-1,j], p_i + \text{table}[i-1, j-w_i]\}$$

$$= \max\{\text{table}[1,10], 12 + \text{table}[1,4]\}$$

table [2, 10] = 22

The table will then be

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	10	10	10	10	10	10	10
2	0	0	0	0	0	0	12	12	12	12	22
3	0										

table [3, 1] with i = 3, j = 1, $w_3 = 8$, $p_3 = 15$.

$j < w_i$ is true for j = 1 to 7.

Hence

table [3, 1] = table [2, 1] = 0

table [3, 2] = table [2, 2] = 0

table [3, 3] = table [2, 3] = 0

table [3, 4] = table [3, 5] = 0

table [3, 6] = table [2, 6] = 12

table [3, 7] = table [2, 7] = 12

Now with table [3, 8] i = 3, j = 8, $w_3 = 8$, $p_3 = 15$.

As $j = w_i$

$$\text{table}[3, 8] = \max\{\text{table}[i-1, j], p_i + \text{table}[i-1, j-w_i]\}$$

$$= \max\{\text{table}[2, 8], 15 + \text{table}[2, 0]\}$$

$$\text{table}[3, 8] = 15$$

Similarly table $[3, 9] = 15$

But with table $[3, 10]$ i $= 3$, $j = 10$, $w_3 = 8$, $p_3 = 15$.

$$\text{table}[3, 10] = \max\{\text{table}[2, 10], 15 + \text{table}[2, 2]\}$$

$$\text{table}[3, 10] = 22$$

The table will be

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	10	10	10	10	10	10	10	10
2	0	0	0	0	0	0	12	12	12	12	22
3	0	0	0	0	0	0	12	12	15	15	22

Now let us apply steps to identify selected items.

	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	10	10	10	10	10	10	10	10
2	0	0	0	0	0	0	12	12	12	12	22
3	0	0	0	0	0	0	12	12	15	15	22

$i = 3$, $k = 10$

$$\text{check table}[i, k] \overset{?}{=} \text{table}[i-1, k]$$

check if table $[3, 10] = $ table $[2, 10]$

i.e. $22 = 22$

$\therefore$ Do not select i^{th} i.e. 3^{rd} item.

Now set $i = i - 1$

$\therefore$ $i = 3 - 1 = 2$

Now $i = 2,\ k = 10$

$\therefore$ $table[2, 10] \stackrel{?}{=} table[1, 10]$

As $table[2, 10] \neq table[1, 10]$

i.e. $10 \neq 0$

$\therefore$ **Select i^{th} i.e. 2^{nd} item.**

Set $i = i - 1$ and $k = k - w_i$

$\therefore$ $i = 2 - 1 = 1$ and $k = 10 - 6 = 4$

Now $i = 1,\ k = 4$

$\therefore$ $table[1, 4] \stackrel{?}{=} table[0, 4]$

As $10 \neq 0$

Select i^{th} i.e. 1^{st} item

Set $i = i - 1$ $k = = - w_i$

$\therefore$ $i = i - 1$ $k = 4 - 4$

$i = 0$ $k = 0$.

Terminate the process by selecting item 1 and item 2

$\therefore$ Solution is $(1, 1, 0)$.

Example 3.8.3 *Consider a knapsack instance n = 3 , (w1, w2, w3) = (2, 3, 4), (p1, p2, p3) = (1, 2, 5) and M = 6. Find optimal solution using dynamic programming ?*

SPPU : April-18, Marks 5

Solution : The table can be constructed using dynamic programming as

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1
2	1	0	1	2	2	3	3
3	3	0	1	2	6	5	6

i=3 and k=6

table[3,6] ≠ table[2,6]

Hence select i^{th} i.e item 3

Now

i=i-1 i.e. i=2

k=k-w_i=6-4=2

table[2,2]=table[1,2]

Hence do not select i^{th} i.e. item 2.

i=i-1=2-1=1

table[1,2] ≠ table[0,2]

Hence select item 1

Thus by solving the given knapsack problem using dynamic programming select item 1 and item 3 with the maximum profit as 1 + 5 = 6.

Review Questions

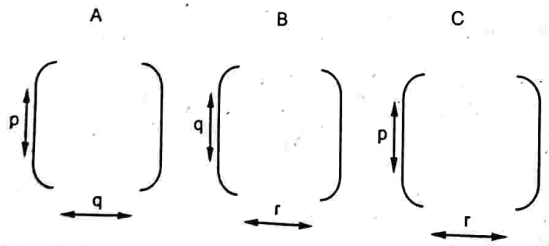
1. *Write an algorithm for 0/1 knapsack problem using dynamic programming approach.* **SPPU : May-07, 08, Dec.-07, Marks 8**
2. *Explain 0/1 Knapsack using dynamic programming with an example.* **SPPU : May-14, Marks 8**
3. *What is dynamic programming ? Define principle of optimality and explain it for 0/1 Knapsack.* **SPPU : Dec.-14, Marks 8**

3.9 Chain Matrix Multiplication

- **Input :** n matrices A_1, A_2,..., A_n of dimensions $P_1 \times P_2$, $P_2 \times P_3$, ... , $P_n \times P_{n+1}$ respectively.
- **Goal :** To compute the matrix product $A_1A_2...A_n$.
- **Problem :** In what order should $A_1A_2...A_n$ be multiplied so that it would take the minimum number of computations to derive the product.

For performing Matrix multiplication the cost is :

Let A and B be two matrices of dimensions p × q and q × r.



Then C = AB. C is of dimensions p × r.

Thus C_{ij} takes n scalar multiplications and (n – 1) scalar additions.

Consider an example of the best way of multiplying 3 matrices :

Let A_1 of dimensions 5×4, A_2 of dimensions 4×6, and A_3 of dimensions 6×2.

$(A_1A_2)A_3$ takes $(5 \times 4 \times 6) + (5 \times 6 \times 2) = 180$

$A_1(A_2A_3)$ takes $(5 \times 4 \times 2) + (4 \times 6 \times 2) = 88$

Thus, $A_1(A_2A_3)$ is much cheaper to compute than $(A_1A_2)A_3$, although both lead to the same final answer. Hence optimal cost is 88.

To solve this problem using dynamic programming method we will perform following steps.

Step 1 : Decide the structure of optimal parenthsization

In this step we have to find the optimal substructure. This substructure is used to construct an optimal solution to the problem. Suppose that there is a production sequence of matrix $A_i A_{i+1} \ldots A_j$ such that $i < j$. Then in optimal parenthsization we have to split this sequence into $A_1 \ldots A_k$ and $A_{k+1} \ldots A_j$. The cost of parenthsization is obtained so that total computing cost can be decided. Here k is some integer value ranging from $i \leq k < j$. Hence **optimal substructure problem** can be described as follow - " Consider that there is sequence $A_i A_{i+1} \ldots A_j$ which can be split into two products A_k and A_{k+1}. Then the parenthsization should produce subchain $A_i A_{i+1} \ldots A_k$ which is l of optimal cost. Thus optimal substructure is important for producing optimal solution. This suggests **bottom up** approach for computing optimal cost.

Step 2 : A recursive solution

For each i , which is within a range of 1 to n and for each j which is within a range of 1 to n the optimal cost m [i, j] is computed using following formula :

For all i, $1 \leq i \leq n$, m [i, i] = 0

and for all i and j such that

$1 \leq i < j \leq n$

$$m[i, j] = \min \{m[i, k] + m[k + 1, j] + P_{i-1} P_k P_j\}$$

where $i \leq k \leq j - 1$

Step 3 : Computing optimal costs

We will construct m [i, j] table using following formula -

i) For i = 1 to n set m [i, i] = 0

ii) For $l \leftarrow 2$ to n compute m [i, j] using

$$m[i, j] = \min \{ m[i,k] + m[k+1, j] + P_{i-1} P_k P_j \}$$

with $i \le k \le j - 1$

Let us understand the procedure of computing optimal cost with some example.

Example : Consider

Matrix	Dimension
A_1	5×4
A_2	4×6
A_3	6×2
A_4	2×7

We have to compute matrix chain order.

Here $P_0 = 5$, $P_1 = 4$, $P_2 = 6$, $P_3 = 2$, $P_4 = 7$

For all i, $1 \le i \le n$

$m[i, i] = 0$

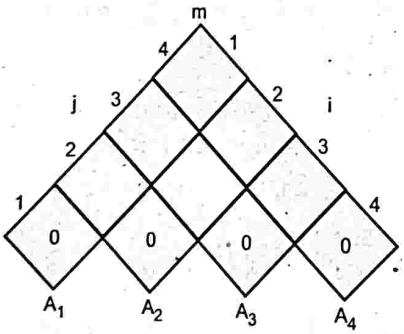
Hence $m[1, 1] = 0$

$m[2, 2] = 0$

$m[3, 3] = 0$

$m[4, 4] = 0$

Hence



Now we will fill up the table horizontally from left to right, assuming $i \le k \le j - 1$

Let $i = 1, j = 2, k = 1$

$$\therefore \quad m[1, 2] = m[i, k] + m[k+1, j] + P_{i-1} P_k P_j$$

$$= m[1,1] + m[2, 2] + P_0 P_1 P_2$$

$$= 0 + 0 + 5 \times 4 \times 6$$

$$\mathbf{m[1, 2] = 120}$$

Let, i = 2, j = 3, k = 2.

$$\therefore \quad m[2,3] = m[i,k] + m[k+1,j] + P_{i-1} P_k P_j$$

$$= m[2,2] + m[3,3] P_1 P_2 P_3$$

$$= 0 + 0 + 4 \times 6 \times 2$$

$$m[2,3] = 48$$

Let , i = 3, j = 4, k = 3

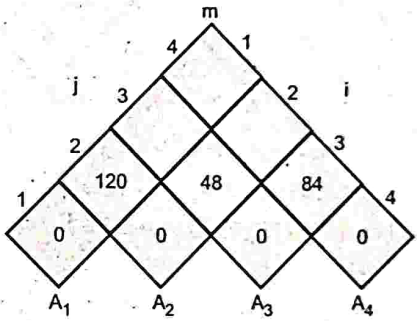
$$m[3,4] = m[i,k] + m[k+1,j] + P_{i-1} P_k P_j$$

$$= m[3,3] + m[4,4] + P_2 P_3 P_4$$

$$= 0 + 0 + 6 \times 2 \times 7$$

$$m[3,4] = 84$$

The table will be partially



Let i = 1 , j = 3, k = 1 or k = 2

$$m[1,3] = \min \begin{cases} (m[1,1]+m[2,3]+P_0 P_1 P_3) \rightarrow k=1 \\ (m[1,2]+m[3,3]+P_0 P_2 P_3) \rightarrow k=2 \end{cases}$$

$$= \min \begin{cases} (0+48+5 \times 4 \times 2) \\ (120+0+5 \times 6 \times 2) \end{cases}$$

$$= \min \begin{cases} 48+40=88 & \text{when } k=1 \\ 120+60=180 \end{cases}$$

$$m[1,3] = 88$$

Let i = 2 , j = 4, k = 2, or k = 3

$$m[2,4] = \min \begin{cases} (m[2,2]+m[3,4]+P_1 P_2 P_4 \rightarrow k=2 \\ (m[2,3]+m[4,4]+P_1 P_3 P_4 \rightarrow k=3 \end{cases}$$

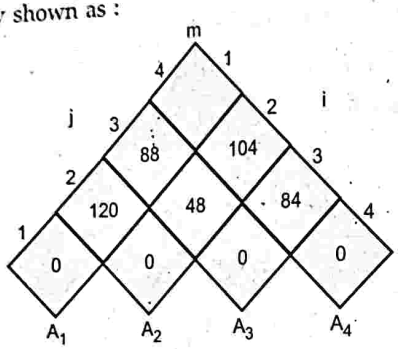
$$= \min \begin{cases} 0+84+4 \times 6 \times 7 \\ 48+0+4 \times 2 \times 7 \end{cases}$$

$$\min [2, 4] = \min \begin{Bmatrix} 84+168 \\ 48+56 \end{Bmatrix}$$

$$= \min \begin{Bmatrix} 252 \\ 104 \end{Bmatrix} \rightarrow \text{when } k=3$$

$$\min [2, 4] = 104$$

The table can be partially shown as :



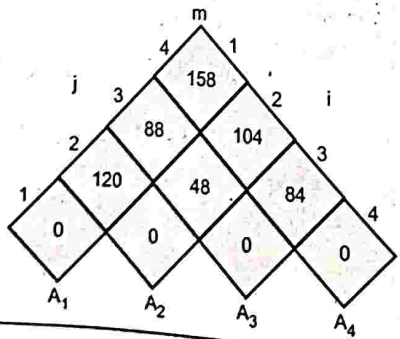
Now, Let $i = 1, j = 4$ then $k = 1$ or 2 or 3

$$m[1, 4] = \min \begin{cases} m[1,1]+m[2,4]+P_0 P_1 P_4 & \rightarrow k=1 \\ m[1,2]+m[3,4]+P_0 P_2 P_4 & \rightarrow k=2 \\ m[1,3]+m[4,4]+P_0 P_3 P_4 & \rightarrow k=3 \end{cases}$$

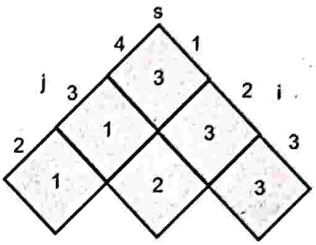
$$= \min \begin{cases} 0+104+5\times4\times7 \\ 120+84+5\times6\times7 \\ 88+0+5\times2\times7 \end{cases}$$

$$= \min \begin{cases} 244 \\ 414 \\ 158 \end{cases} \rightarrow \text{when } k=3$$

Hence the matrix table will be -



We will build the s table using s [i, j] = k



m [1, 4] determines the optimal cost i.e. 158

Step 4 : Printing optimal parenthsization

The sequence of optimal parenthsization which gives the optimum cost is obtained using following algorithm.

```
Algorithm display _ matrix _ order ( s, i, j )
// Problem Description : This algorithm prints
// the optimal parenthsization of matrix
if ( i = j ) then
      print ( " A " i )
else
{ print " ( "
      display_ matrix_order ( s, i, s [ i, j ] )
      display_ matrix_order ( s, s [ i, j ] + 1 , j )
      print " ) "
} // end of else
} // end of algorithm
```

From s table given above we find parenthesis s[1, 4] = 3. Hence put bracket after A_3.

$\therefore (A_1, A_2, A_3)(A_4)$.

Now consider (A_1, A_2, A_3).

As s[1, 3] = 1 put bracket after A_1. Hence now we get $(A_1)(A_2 A_3)(A_4)$.

3.9.1 Algorithm

The algorithm for building the matrix chain table m[i,j] and s[i,j] is as follows -

```
Algorithm Matrix_chain(p[0...n])
{
//Problem Description: This algorithm is to build the table m[i,j] and s[i,j]
for (i ← 1 to n) do
   m[i,i] ← 0
```

```
for(len ← 2 to n) do
{
 for (i←2 to (n – len+1) ) do
 {
   j ← i+len – 1
  m[i,j]=infinity
  for (k ← i to j – 1) do
  {
  q ← m [i,k] +m[k+1,j]+p[i–1]*p[k]*p[j]
   if (q<m[i,j]) then
   {
     m[i,j]← q
     s[i,j] ← k
   }//end of if
  }//end of k's for loop
 }//end of i's for loop
}//end of len's for loop
return m[1,n]
}//end of algorithm
```

Computing m[i,j] and s[i,j]

Analysis

The basic operation in this algorithm is computation of m[i,j] and s[i,j] which is within three nested for loops. Hence the time complexity of the above algorithm is $\Theta(n^3)$.

Example 3.9.1 *Consider the chain of matrices A1, A2, ... A6 with the dimensions given below. Give the optimal parenthsization to get the product A2 ... A5.*

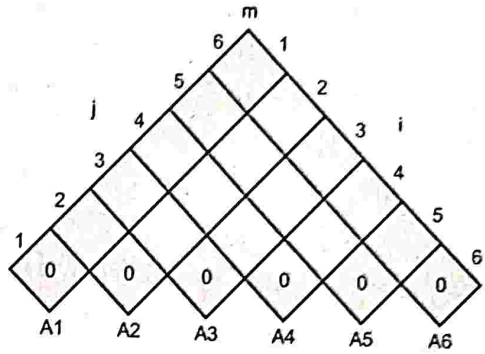
Matrix	Dimension
A1	30 × 35
A2	35 × 15
A3	15 × 5
A4	5 × 10
A5	10 × 20
A6	20 × 25

Solution : We will compute matrix chain order.

Here $P_0 = 30$, $P_1 = 35$, $P_2 = 15$, $P_3 = 5$, $P_4 = 10$, $P_5 = 20$, $P_6 = 25$

For all $1 \le i \le n$ $m[i, i] = 0$

$\therefore$ $m[1, 1] = m[2, 2] = m[3, 3] = m[4, 4] = m[5, 5] = m[6, 6] = 0$



Now, we will fill up the table horizontally from left to right, assuming $i \leq k \leq j - 1$

Let $i = 1,\ j = 2,\ k = 1$

$$\therefore \quad m[1, 2] = m[i, k] + m[k+1, j] + P_{i-1} P_k P_j$$

$$= m[1, 1] + m[2,2] + P_0 P_1 P_2 = 0 + 0 + 30 * 35 * 15$$

$$= 15750. \quad \textbf{with } k = 2$$

Similarly $m[2, 3] = m[2, 2] + m[3, 3] + P_1 P_2 P_3 = 0 + 0 + 35 * 15 * 5$

$m[2, 3] = 2625$ **with** $k = 2$

$m[3, 4] = m[3, 3] + m[4, 4] + P_2 P_3 P_4 = 0 + 15 * 5 * 10$

$m[3, 4] = 750$ **with** $k = 3$

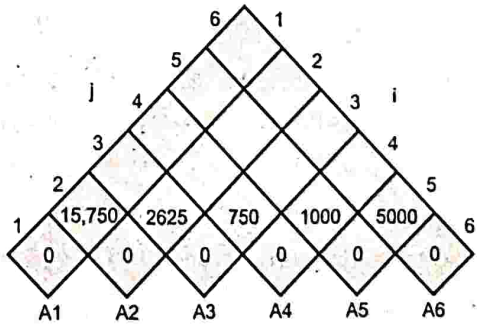
$m[4, 5] = m[4, 4] + m[5, 5] + P_3 P_4 P_5$

$m[4, 5] = 1000$ **with** $k = 4$

$m[5, 6] = m[5, 5] + m[6, 6] + P_4 P_5 P_6$

$m[5, 6] = 5000$ **with** $k = 5$

The table will be.



Now

i = 1, j = 3 k = 1 or 2.

$$m[1,3] = \min \begin{cases} m[1,1]+m[2,3]+P_0P_1P_2 = 0+2625+(30\cdot35\cdot5) = 7875 \\ m[1,2]+m[3,3]+P_0P_2P_3 = 15750+0+(30\cdot15\cdot5) = 18000 \end{cases}$$

m [1, 3] = 7875 with k = 1

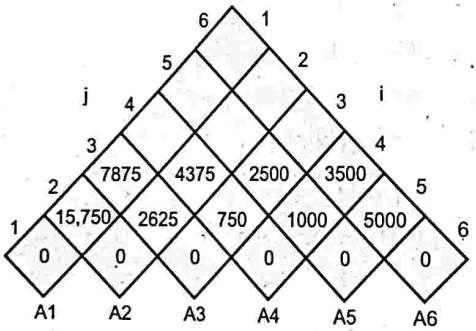
$$m[2,4] = \min \begin{cases} m[2,2]+m[3,4]+P_1P_2P_4 = 0+750+(35\cdot15\cdot10) = 6000 \\ m[2,3]+m[4,4]+P_1P_3P_4 = 2625+0+(35\cdot5\cdot10) = 4375 \end{cases}$$

m [2, 4] = 4375 with k = 3

$$m[3,5] = \min \begin{cases} m[3,3]+m[4,5]+P_2P_3P_5 = 0+1000+(15\cdot5\cdot20) = 2500 \\ m[3,4]+m[5,5]+P_2P_4P_5 = 750+0+(15\cdot10\cdot20) = 3750 \end{cases}$$ **min = 2500, k = 3**

$$m[4,6] = \min \begin{cases} m[4,4]+m[5,6]+P_3P_4P_6 = 0+5000+(5\cdot10\cdot25) = 6250 \\ m[4,5]+m[6,6]+P_3P_5P_6 = 1000+0+(5\cdot20\cdot25) = 3500 \end{cases}$$ **min = 3500, k = 5**

The table will then be -



$$m[1,4] = \min \begin{cases} m[1,1]+m[2,4]+P_0P_1P_4 = 0+4375+(30\cdot35\cdot10) = 14875 \\ m[1,2]+m[3,4]+P_0P_2P_4 = 15750+750+(30\cdot15\cdot10) = 21000 \\ m[1,3]+m[4,4]+P_0P_3P_4 = 7875+0+(30\cdot5\cdot10) = 9375 \end{cases}$$ **min = 9375, k = 3**

$$m[2,5] = \min \begin{cases} m[2,2]+m[3,5]+P_1P_2P_5 = 0+2500+(30\cdot15\cdot20) = 13000 \\ m[2,3]+m[4,5]+P_1P_3P_5 = 2625+1000+(35\cdot5\cdot20) = 5125 \\ m[2,4]+m[5,5]+P_1P_4P_5 = 4375+0+(35\cdot10\cdot20) = 11375 \end{cases}$$ **min = 7125, k = 3**

$$m[3,6] = \min \begin{cases} m[3,3]+m[4,6]+P_2P_3P_6 = 0+3500+(15\cdot5\cdot25) = 5375 \\ m[3,4]+m[5,6]+P_2P_4P_6 = 750+5000+(15\cdot10\cdot25) = 9500 \\ m[3,5]+m[6,6]+P_2P_5P_6 = 2500+0+(15\cdot20\cdot25) = 10000 \end{cases}$$ **min = 5375, k = 3**

$$m[1,5] = \min\begin{cases} m[1,1]+m[2,5]+P_0P_1P_5 = 0+7125+(30\cdot35\cdot20) = 25125 \\ m[1,2]+m[3,5]+P_0P_2P_5 = 15750+2500+(30\cdot15\cdot20) = 27250 \\ m[1,3]+m[4,5]+P_0P_3P_5 = 7875+1000+(30\cdot5\cdot20) = 11875 \\ m[1,4]+m[5,5]+P_0P_4P_5 = 9375+0+(30\cdot10\cdot20) = 15375 \end{cases}$$

min = 11875

k = 3

$$m[2,6] = \min\begin{cases} m[2,2]+m[3,6]+P_1P_2P_6 = 0+9375+(35\cdot15\cdot25) = 22500 \\ m[2,3]+m[4,6]+P_1P_3P_6 = 2625+3500+(35\cdot5\cdot25) = 10500 \\ m[2,4]+m[5,6]+P_1P_4P_6 = 4375+5000+(35\cdot10\cdot25) = 18125 \\ m[2,5]+m[6,6]+P_1P_5P_6 = 7125+0+(35\cdot20\cdot25) = 24625 \end{cases}$$

min = 10500

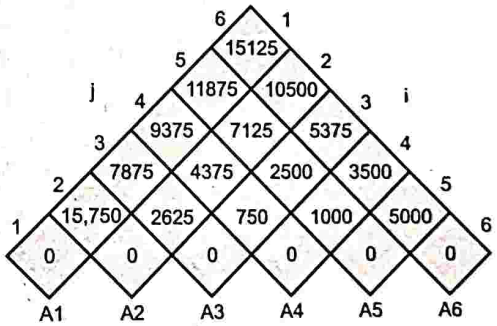
k = 3

$$m[1,6] = \min\begin{cases} m[1,1]+m[2,6]+P_0P_1P_6 = 0+10500+(30\cdot35\cdot25) = 36750 \\ m[1,2]+m[3,6]+P_0P_2P_6 = 15750+9375+(30\cdot15\cdot25) = 36375 \\ m[1,3]+m[4,6]+P_0P_3P_6 = 7875+3500+(30\cdot5\cdot25) = 15125 \\ m[1,4]+m[5,6]+P_0P_4P_6 = 9375+5000+(30\cdot10\cdot25) = 21875 \\ m[1,5]+m[6,6]+P_0P_5P_6 = 11875+0+(30\cdot20\cdot25) = 26875 \end{cases}$$

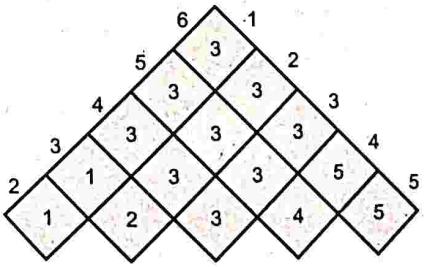
min = 15125

k = 3

The table will then be -



The table s [i, j] for k will be



Here s[1, 6] = 3

∴ (A1 A2 A3) (A4 A5 A6)

s[1, 3] = 1

∴ (A1 (A2 A3)) (A4 A5 A6)

s[4, 6] = 5

∴ (A1 (A2 A3)) (A4 A5) A6

The matrix chain order will be

((A1 (A2A3)) ((A4 A5) A6))

Example 3.9.2 *Using algorithm find an optimal parenthisization of matrix chain product whos sequence of dimension is (5, 10, 3, 12, 5, 50, 6) (use dynamic programming).*

Solution : We will compute matrix chain order.

Here $P_0 = 5$, $P_1 = 10$, $P_2 = 3$, $P_3 = 12$, $P_4 = 5$, $P_5 = 50$, $P_6 = 6$. Our first step is to $m[i, i] = 0$. for $1 \le i \le 5$

Hence $m[1, 1] = m[2, 2] = m[3, 3] = m[4, 4] = 0$.

Now $m[1, 2] = P_0 * P_1 * P_2 = 5 * 10 * 3 = 150$

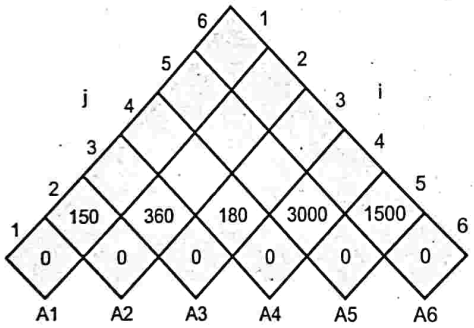
$m[2, 3] = P_1 * P_2 * P_3 = 10 * 3 * 12 = 360$

$m[3, 4] = P_2 * P_3 * P_4 = 3 * 12 * 5 = 180$

$m[4, 5] = P_3 * P_4 * P_5 = 12 * 5 * 50 = 3000$

$m[5, 6] = P_4 * P_5 * P_6 = 5 * 50 * 6 = 1500$

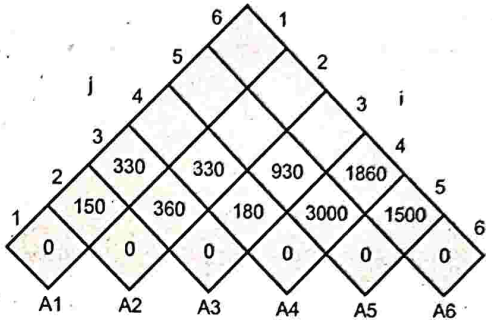
The table will then be



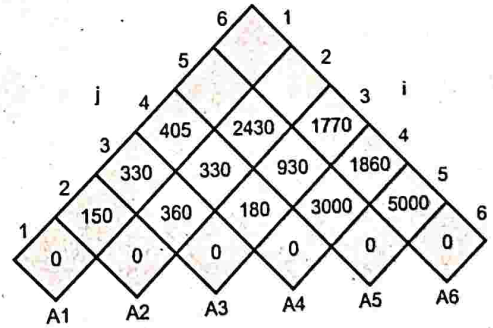
Now we will fill up the table for the values $m[1, 3]$, $m[2, 4]$, $m[3, 5]$ and $m[4,$

$m[1, 3] = \min$	$m[1,1] + m[2,3] + P_0 P_1 P_3$	$= 0 + 360 + (5 \cdot 10 \cdot 12) = 960$	min = 330 with k = 2
	$m[1,2] + m[3,3] + P_0 P_2 P_3$	$= 150 + 0 + (5 \cdot 3 \cdot 12) = 330$	
$m[2, 4] = \min$	$m[2,2] + m[3,4] + P_1 P_2 P_4$	$= 0 + 180 + (10 \cdot 3 \cdot 5) = 330$	min = 330 k = 2
	$m[2,3] + m[4,4] + P_1 P_3 P_4$	$= 360 + 0 + (10 \cdot 12 \cdot 50) = 6360$	
$m[3,5] = \min$	$m[3,3] + m[4,5] + P_2 P_3 P_5$	$= 0 + 3000 + (3 \cdot 12 \cdot 50) = 4800$	min = 930 k = 4
	$m[3,4] + m[5,5] + P_2 P_4 P_5$	$= 180 + 0 + (3 \cdot 5 \cdot 50) = 930$	
$m[4, 6] = \min$	$m[4,4] + m[5,6] + P_3 P_4 P_6$	$= 0 + 1500 + (12 \cdot 5 \cdot 6) = 1860$	min = 1860 k = 4
	$m[4,5] + m[6,6] + P_3 P_5 P_6$	$= 3000 + 0 + (12 \cdot 50 \cdot 6) = 6600$	

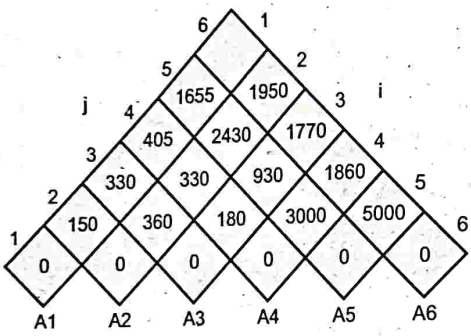
Now we will compute m[1, 4], m[2, 5] and m[3, 6]



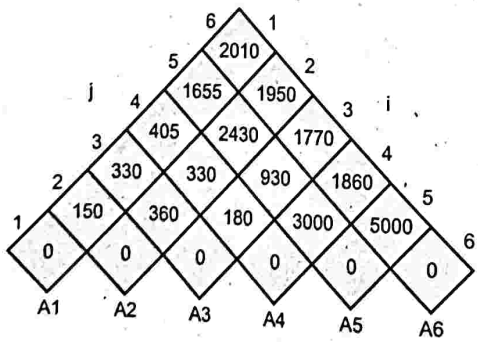
m [1, 4] = **min**	m [1,1] + m [2,4] + $P_0 P_1 P_4$	= 0 + 330 + (5·10·5) = 580	min = 405 k = 2
	m [1,2] + m [3,4] + $P_0 P_2 P_4$	= 150 + 180 + (5·3·5) = 405	
	m [1,3] + m [4,4] + $P_0 P_3 P_4$	= 330 + 0 + (5·12·5) = 630	
m [2, 5] = **min**	m [2,2] + m [3,5] + $P_1 P_2 P_5$	= 0 + 930 + (10·3·50) = 2430	min = 2430 k = 2
	m [2,3] + m [4,5] + $P_1 P_3 P_5$	= 360 + 3000 + (10·12·50) = 9360	
	m [2,4] + m [5,5] + $P_1 P_4 P_5$	= 330 + 0 + (10·5·50) = 2830	
m [3, 6] = **min**	m [3,3] + m [4,6] + $P_2 P_3 P_6$	= 0 + 1860 + (3·12·6) = 2076	min = 1770 k = 4
	m [3,4] + m [5,6] + $P_2 P_4 P_6$	= 180 + 1500 + (3·5·6) = 1770	
	m [3,5] + m [6,6] + $P_2 P_5 P_6$	= 930 + 0 + (3·50·6) = 1830	



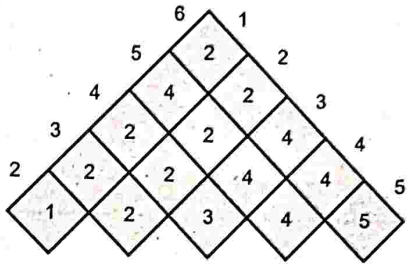
m[1, 5] = min	$m[1, 1] + m[2, 5] + P_0 P_1 P_5$	= 0+ 2430 + (5 ·10·50) = 4930	min =
	$m[1, 2] + m[3, 5] + P_0 P_2 P_5$	= 150 + 930 + (5·3·50) = 1830	k =
	$m[1, 3] + m[4, 5] + P_0 P_3 P_5$	= 330 + 3000 + (5·12·50) = 6330	
	$m[1, 4] + m[5, 5] + P_0 P_4 P_5$	= 405 + 0 + (5·5·50) = 1655	
m[2, 6] = min	$m[2, 2] + m[3, 6] + P_1 P_2 P_6$	= 0 + 1770 + (10·3·6) = 1950	
	$m[2, 3] + m[4, 6] + P_1 P_3 P_6$	= 360 + 1860 + (10·12·6) = 2940	min =
	$m[2, 4] + m[5, 6] + P_1 P_4 P_6$	= 330 + 1500 + (10·5·6) = 2130	k =
	$m[2, 5] + m[6, 6] + P_1 P_5 P_6$	= 2430 + 0 + (10·50·6) = 5430	



m[1,6] = min	$m[1, 1] + m[2, 6] + P_0 P_1 P_6$	= 0 + 1950 + (5·10·6) = 2250	min = 2010
	$m[1, 2] + m[3, 6] + P_0 P_2 P_6$	= 150 + 1770 + (5·3·6) = 2010	k = 2
	$m[1, 3] + m[4, 6] + P_0 P_3 P_6$	= 330 + 1860 + (5·12·6) = 2550	
	$m[1, 4] + m[5, 6] + P_0 P_4 P_6$	= 405 + 1500 + (5·5·6) = 2055	
	$m[1, 5] + m[6, 6] + P_0 P_5 P_6$	= 1655 + 0+ (5·50·6) = 3155	



The $s[i, j]$ table for the values of k will be



The optimal parenthesization will be

$(A1 * A2) * ((A3 * A4) * (A5 * 6))$

The s[1, 6] = 2

$\therefore$ (A1 A2) (A3 A4 A5 A6)

The s[3, 6] = 4

$\therefore$ (A1 A2) (A3 A4) (A5 A6)

Thus we obtain optimal parenthesization as -

$(A1 * A2) * (A3 * A4) * (A5 * A6)$

Notes

Unit IV

4 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.

Contents

Part I : Backtracking

SPPU : Dec.-06,10,11,12,13, May-08,11,14, Marks 18

4.1 Principle

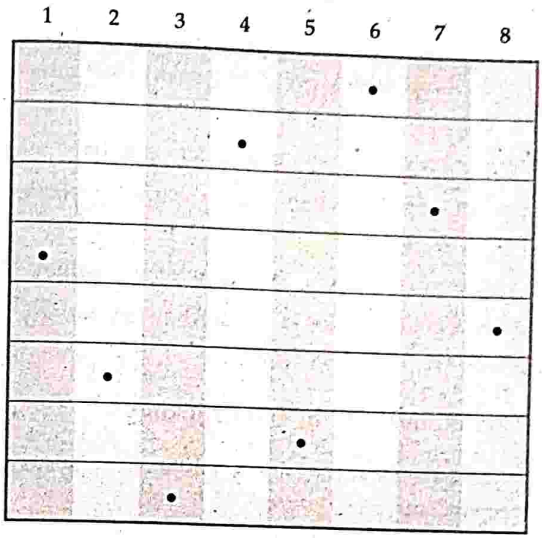
- In the backtracking method
  1. The desired solution is expressible as an n tuple $(x_1, x_2,...x_n)$ where x_i is chosen from some finite set S_i.
  2. The solution maximizes or minimizes or satisfies a criterion function C $(x_1, x_2,...x_n)$.
- The problem can be categorized into three categories.
  For instance - For a problem P let C be the set of constraints for P. Let D be the set containing all solutions satisfying C then
  Finding whether there is any feasible solution ? - Is the decision problem.
  What is the best solution ? - Is the optimization problem.
  Listing of all the feasible solution - Is the enumeration problem.
- The basic idea of backtracking is to build up a vector, one component at a time and to test whether the vector being formed has any chance of success.
- The major **advantage** of backtracking algorithm is that we can realize the fact that the partial vector generated does not lead to an optimal solution. In such a situation that vector can be ignored.
- Backtracking algorithm determines the solution by systematically searching the solution space (i.e. set of all feasible solutions) for the given problem.
- Backtracking is a **depth first search** with some **bounding function.**
- Backtracking algorithm solves the problem using **two types of constraints - Explicit constraint** and **Implicit constraints.**
- **Definition : Explicit constraints** are the rules that restrict each element x_i has to be chosen from given set only. Explicit constraints depends on particular instance I of the problem. All the tuples from solution set must satisfy the explicit constraints.
- **Definition : Implicit constraints** are the rules that decide which tuples in the solution space of I satisfy the criterion function. Thus the implicit constraints represent by which x_i in the solution set must be related with each other.

For example

Example 1 : 8-Queen's problem - The 8-queen's problem can be stated as follows. Consider a chessboard of order 8×8. The problem is to place 8 queens on this board such that no two queens can attack each other. That means no two queens can be placed on the same row, column or diagonal.

The solution to 8-queens problem can be obtained using backtracking method.

The solution can be given as below -



This 8 Queen's problem is solved by applying implicit and explicit constraints.

The **explicit constraints** show that the solution space S_i must be {1, 2, 3, 4, 5, 6, 7, 8} with $1 \leq i \leq 8$. Hence solution space consists of 8^8 8-tuples.

The **implicit constraint** will be - 1) No two x_i will be same. That means all the queens must lie in different columns.

2) No two queens can be on the same row, column or diagonal.

Hence the above solution can be represented as an 8-tuple {4, 6, 8, 2, 7, 1, 3, 5}.

Example 2 : Sum of subsets -

There are n positive numbers given in a set. The desire is to find all possible subsets of this set, the contents of which add onto a predefined value M.

In other words,

Let there be n elements given by the set $w = (w_1, w_2, w_3, \ldots, w_n)$ then find out all the subsets from whose sum is M.

For example

Consider n = 6 and $(w_1, w_2, w_3, w_4, w_5, w_6) = (25, 8, 16, 32, 26, 52)$ and M = 59 then we will get desired sum of subset as (25, 8, 26).

We can also represent the sum of subset as (1, 1, 0, 0, 1, 0). If solution subset is represented by an n-tuple $(x_1, x_2, \ldots x_n)$ such that x_i could be either 0 or 1. The $x_i = 1$

means the weight w_i is to be chosen and $x_i = 0$ means that weight w_i is not to chosen.

The **explicit constraint** on sum of subset problem will be that any element $x_i \in \{i|i$ an integer and $1 \le i \le n\}$. That means we must select the integer from given set elements.

The **implicit constraints** on sum of subset problem will be -

1) No two elements will be the same. That means no element can be repeated selected.
2) The sum of the elements of subset = M (a predefined value).
3) The selection of the elements should be done in an orderly manner. That mea (1, 3, 7) and (1, 7, 3) are treated as same.

4.1.1 Some Terminologies used in Backtracking

Backtracking algorithms determine problem solutions by systematically searching f the solutions using **tree structure.**

For example -

Consider a 4-queen's problem. It could be stated as "there are 4 queens that can placed on 4 × 4 chessboard. Then no two queens can attack each other".

Following Fig. 4.1.1 shows tree organization for this problem.

(See Fig. 4.1.1 on next page)

- Each node in the tree is called **a problem state.**
- All paths from the root to other nodes define the **state space of** the problem.
- The solution states are those problem states s for which the path from root to defines a **tuple** in the solution space.

  In some trees the leaves define the **solution states.**
- **Answer states :** These are the leaf nodes which correspond to an element in the set solutions, these are the states which satisfy the implicit constraints.

  For example

(See Fig. 4.1.2 on Page 4-6)

- A node which is been generated and all whose children have not yet been generate **is called live node.**
- The live node whose children are currently being expanded is called **E-node.**
- **A dead node** is a generated node which is not to be expanded further or all of whos children have been generated.
- There are two methods of generating state search tree -

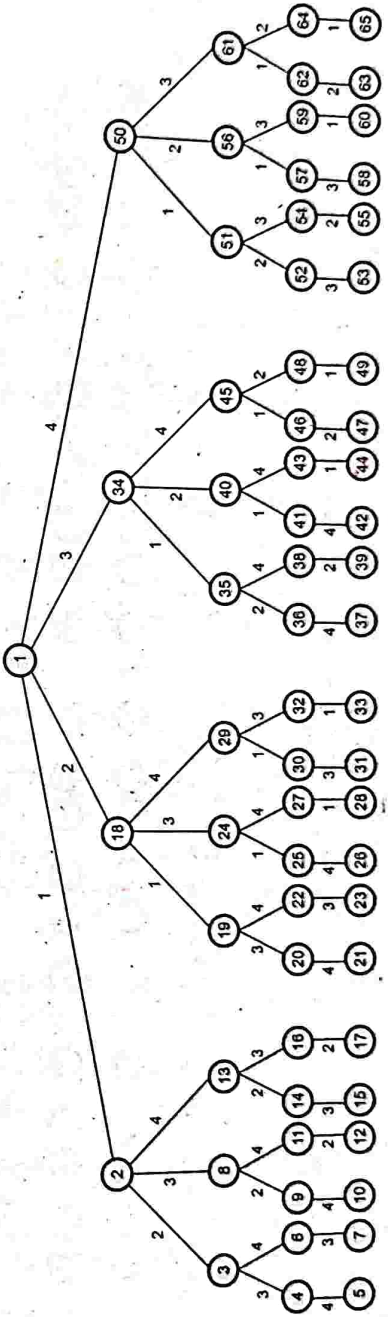


Fig. 4.1.1 State-space tree for 4-queen's problem

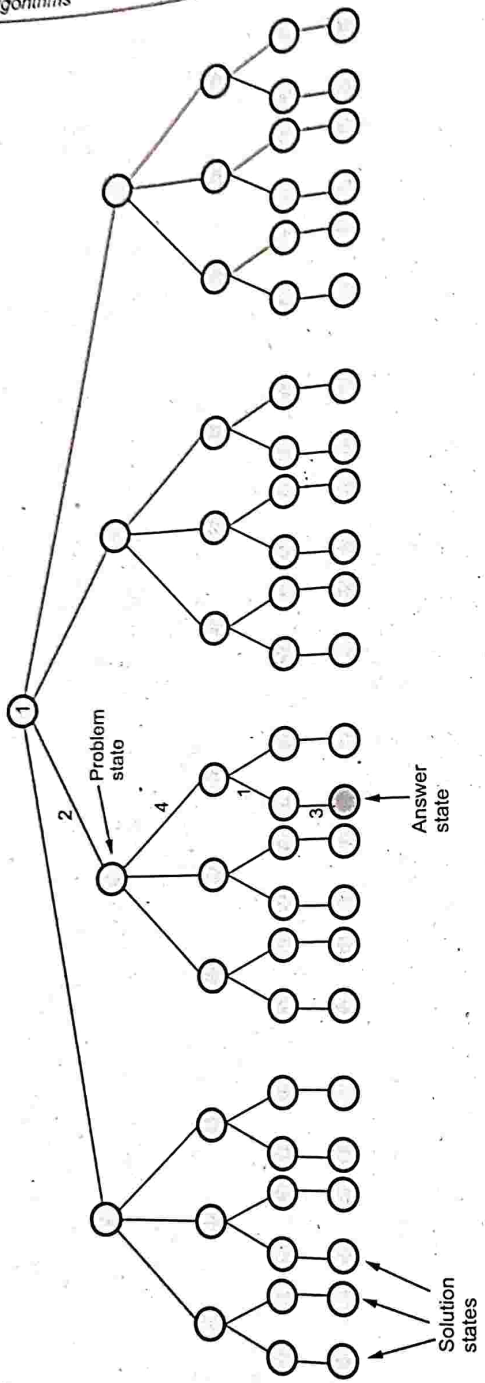


Fig. 4.1.2

i) **Backtracking** - In this method, as soon as new child C of the current E-node N is generated, this child will be the new E-node.

The N will become the E-node again when the subtree C has been fully explored.

ii) **Branch and Bound** - E-node will remain as E-node until it is dead.

- Both bracktracking and branch and bound methods use bounding functions. The bounding functions are used to kill live nodes without generating all their children. The care has to be taken while killing live nodes so that at least one answer node or all answer nodes are obtained.

Example 4.1.1 *Define following terms : State space, explicit constraints, implicit constraints, problem state, solution states, answer states, live node, E-node, dead node, bounding functions.* **SPPU : May-11, Marks 8, Dec.-11, Marks 16, Dec.-12, Marks 18**

Solution :

State space : All paths from root to other nodes define the state space of the problem. The tree organization for 4-queen's problems is shown by given Fig. 4.1.3. (Refer Fig. 4.1.3 on next page)

Explicit constraints : Explicit constraints are rules, which restrict, each vector element to be chosen from given set.

Implicit constraints : Implicit constraints are rules which determine which of the tuples in the solution space satisfy the criterion function.

Problem states : Each node in the state space tree is called problem state.

Solution states : The solution states are those problem states s for which the path from root to s defines a tuple in the solution space. In some trees the leaves define solution states.

Answer states : These are the leaf nodes which correspond to an element in the set of solutions. These are the states which satisfy the implicit constraints.

Live node : A node which is generated and whose children have not yet been generated is called live node.

E-node : The live node whose children are currently being expanded is called E-node.

Dead node : A dead node is a generated node which is not to be expanded further or all of whose children have been generated.

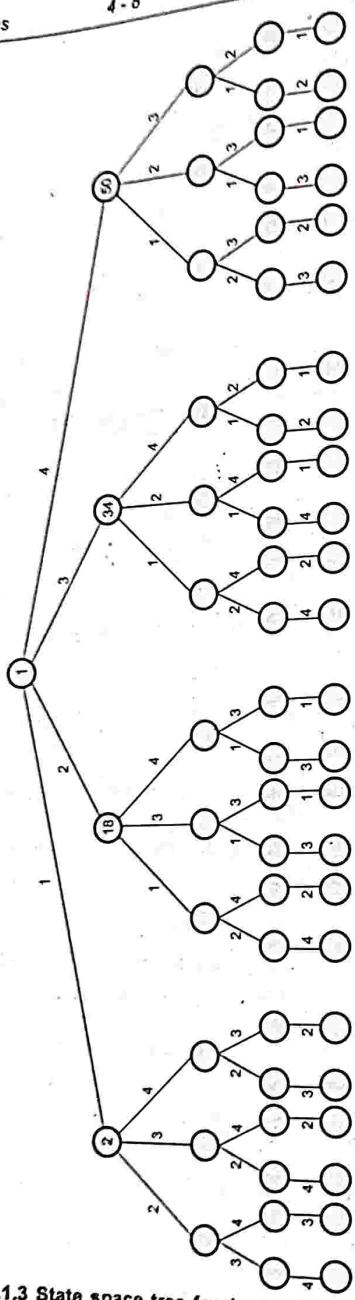


Fig. 4.1.3 State space tree for 4-queen's problems

Bounding functions : Bounding functions will be used to kill live nodes without generating all their children. A care should be taken while doing so, because atleast one answer node should be produced or even all the answer nodes be generated if problem needs to find all possible solutions.

Review Questions

1. *What are the constraints that must be satisfied while solving any problem using backtracking ? Explain briefly.* **SPPU : Dec.-06, Marks 4; Dec.-10, Marks 6**
2. *Enlist the characteristics of backtracking strategy.* **SPPU : May-08, Marks 2**
3. *Explain the following terms :*
   Live nodes, expanding nodes, bounding function and solution space. **SPPU : Dec.-13, Marks 8**
4. *Explain implicit and explicit constraints.* **SPPU : May-14, Marks 6**
5. *What are implicit and explicit constraints with respect to backtracking.* **SPPU : May-14, Marks 8**

4.2 Control Abstraction and Time Analysis

SPPU : Dec.-06, 07, May-07, Marks 8

Recursive Algorithm

```
Algorithm Backtrack()
//This is recursive backtracking algorithm
//a[k] is a solution vector. On entering (k–1) remaining next
//values can be computed.
//T(a1,a2,...ak) be the set of all values for a(i+1), such that
//(a1,a2,...ai+1) is a path to problem state.
// Bi+1 is a bounding function such that if Bi+1(a1,a2,...ai+1) is false
//for a path (a1,a2,...ai+1) from root node to a problem state then
//path can not be extended to reach an answer node
{
 for ( each ak that belongs to T((a1,a2,...ak-1) do
 {
  if(Bk(a1,a2,...ak) = true) then // feasible sequence
  {
    if ((a1,a2,...ak) is a path to answer node then
      print(a[1],a[2],...a[k]);
    if (k<n) then
      Backtrack(k+1); //find the next set.
  }
 }
}
```

Iterative Algorithm

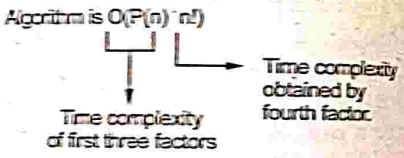
The non recursive backtracking algorithm can be given as -

```
Algorithm Non_Rec_Back(n)
// This is a non recursive version of backtracking
//a[1.....n] is a solution vector and each a1,a2,....ak will be used to print solution.
{
 k:=1;
 while(k = 0) do
 {
      if(any a[k] that belongs to T(a[1],a[2],...a[k-1]remains untried)
      AND (B_k (a[1],a[2]....a[k] is true) then
   {
    if((a[1],a[2]....a[k]) is a path to answer node) then
     write(a[1],a[2]...a[k]); // solution printed
    //consider next element of the set
    k:=k+1;
    else
      k : =k-1;  //backtrack to most recent value
   }
  }
}
```

Efficiency of Backtracking Algorithm

The efficiency of both the backtracking algorithm depends upon four factors -

1) The time required to generate a[k].
2) The number of a[k] elements which satisfy the explicit constraints.
3) The time required by bounding functions B_k to generate a feasible sequence.
4) The number of elements a[k] that satisfy the bounding functions B_k for all the values of k.



The first three factors are **independant** on the given **problem instance. And the time** complexity of these three factors is polynomial time. But the fourth factor varies according to the size of problem instance. That means, if there are n! nodes which satisfy the bounding function then the worst case time complexity of backtracking

Review Questions

1. *Write a recursive algorithm which shows a recursive formulation of the backtracking technique.* **SPPU : Dec.-06, Marks 8**
2. *Write a schema for an iterative backtracking method.* **SPPU : May-07, Dec.-07, Marks 8**

4.3 Applications of Backtracking

Various applications that are based on backtracking method are -

1. 8 Queen's problem : This is a problem based on chess games. By this problem, it is stated that arrange the queens on chessboard in such a way that no two queens can attack each other.
2. Sub of subset problem.
3. Graph coloring problem.
4. Finding Hamiltonian cycle.
5. Knapsack problem.

4.4 The 8-Queen Problem

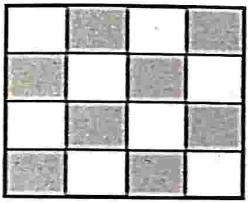
SPPU : May-07,11,12,14, Dec.-07,10,13,14, Aug.-15, Marks 12

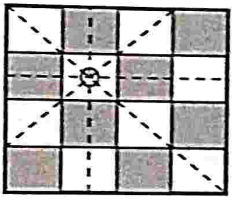
The n-queen's problem can be stated as follows.

Consider a n × n chessboard on which we have to place n queens so that no two queens attack each other by being in the same row or in the same column or on the same diagonal.

For example

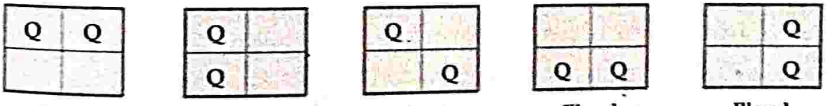
Consider 4 × 4 board





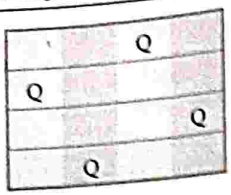
The next queen - if is placed on the paths marked by dotted lines then they can attack each other

- 2-Queen's problem is not solvable - Because 2-queens can be placed on 2 × 2 chessboard as



Illegal Illegal Illegal Illegal Illegal

- But 4-queen's problem is solvable.

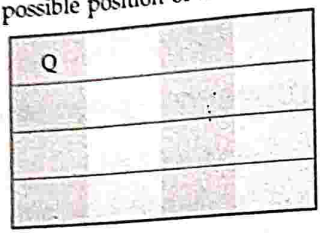


⇐ Note that no two queens can attack each other.

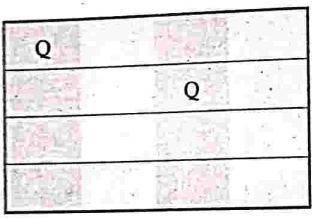
4.4.1 How to Solve n-Queen's Problem ?

Let us take 4-queens and 4×4 chessboard.

- Now we start with empty chessboard.
- Place queen 1 in the first possible position of its row i.e. on 1st row and 1st column.

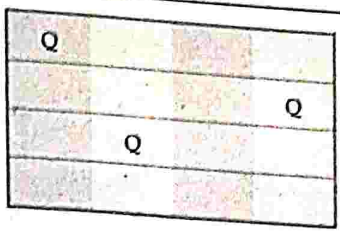


- Then place queen 2 after trying unsuccessful place - 1(1, 2), (2, 1), (2, 2) at (2, 3) i.e 2nd row and 3rd column.

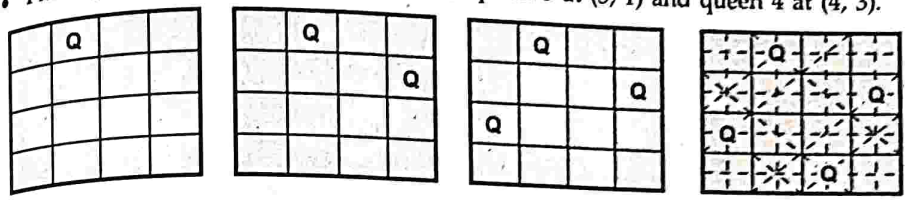


- This is the dead end because a 3rd queen cannot be placed in next column, as there i no acceptable position for queen 3. Hence algorithm **backtracks** and places the 2n queen at (2, 4) position.

- The place 3rd queen at (3, 2) but it is again another dead end as next quee (4th queen) cannot be placed at permissible position.



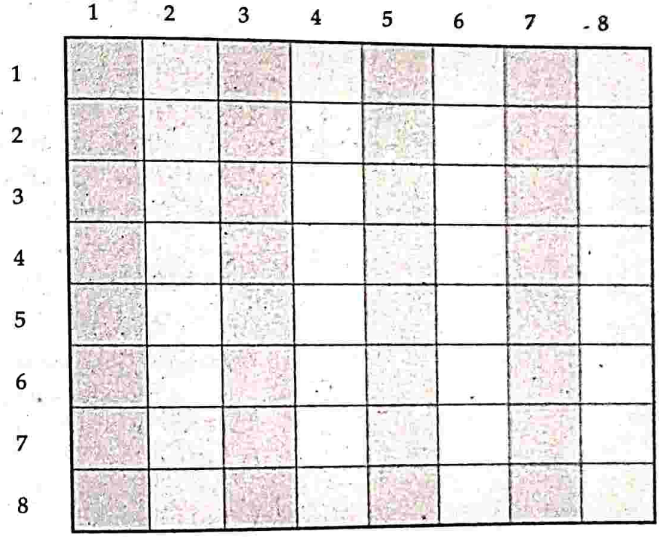
- Hence we need to backtrack all the way upto queen 1 and move it to (1, 2).
- Place queen 1 at (1, 2), queen 2 at (2, 4), queen 3 at (3, 1) and queen 4 at (4, 3).



Thus solution is obtained.
(2, 4, 1, 3) in rowwise manner.

The state space tree of 4-queen's problem is shown in Fig. 4.4.1 (See on next page)

Now we will consider how to place 8-Queen's on the chessboard.
Initially the chessboard is empty.



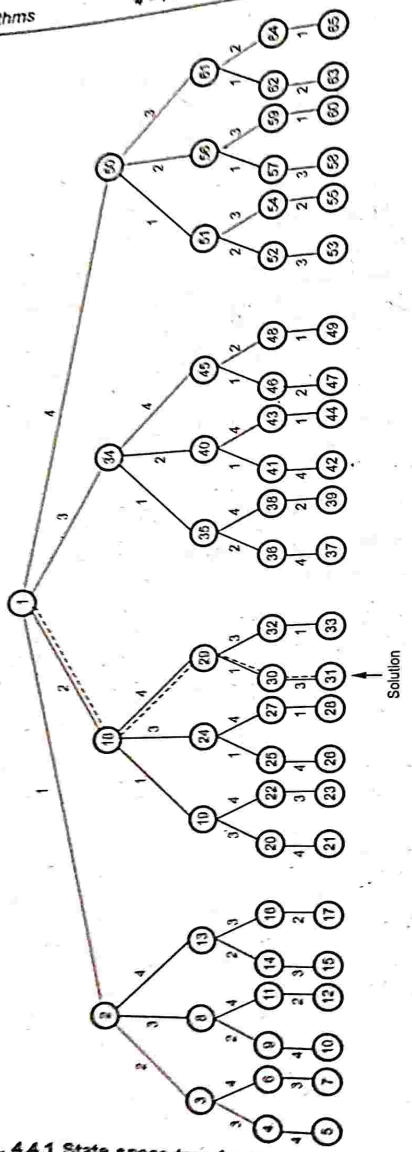
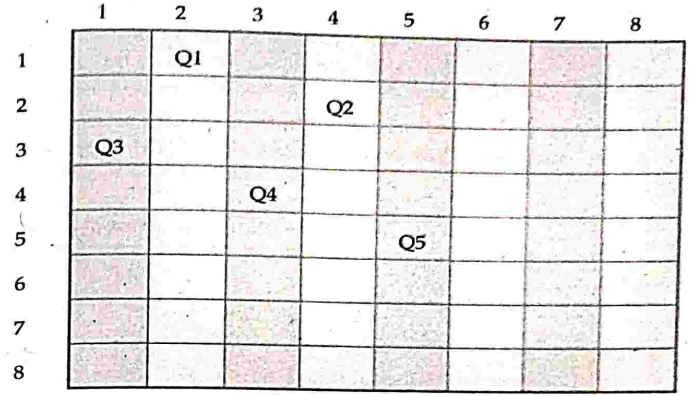
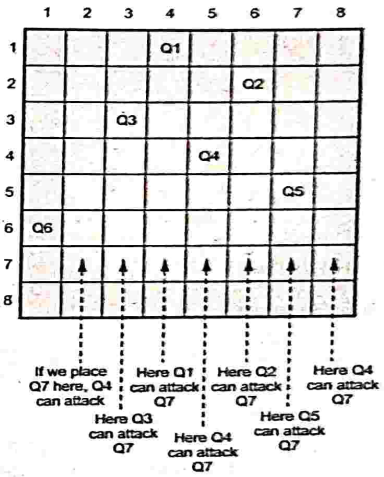


Fig. 4.4.1 State space tree for 4-queen's problem

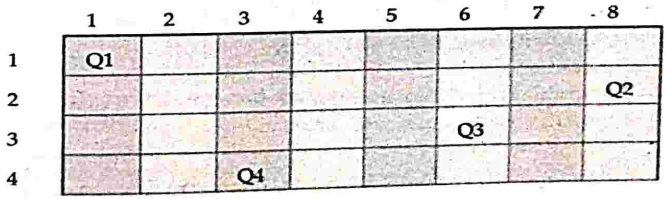
Now we will start placing the queens on the chessboard.

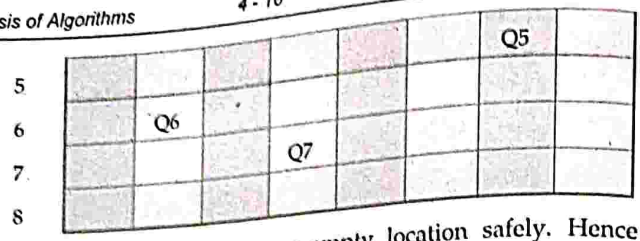


Thus we have placed 5 queens in such a way that no two queens can attack each other. Now if we want to place Q6 at location (6, 6) then queen Q5 can attack, if Q6 is placed at (6, 7) then Q1 can attack, if Q6 is placed at (6, 8) then Q2 can attack it. Similarly at (6, 5) the Q5 will attack it. At (6, 4) the Q2 will attack, at (6, 3) the Q4 will attack, at (6, 2) the Q1 will attack and at (6, 1) Q3 will attack the queen Q6. This shows that we need to backtrack and change the previously placed queens positions. It could then be -

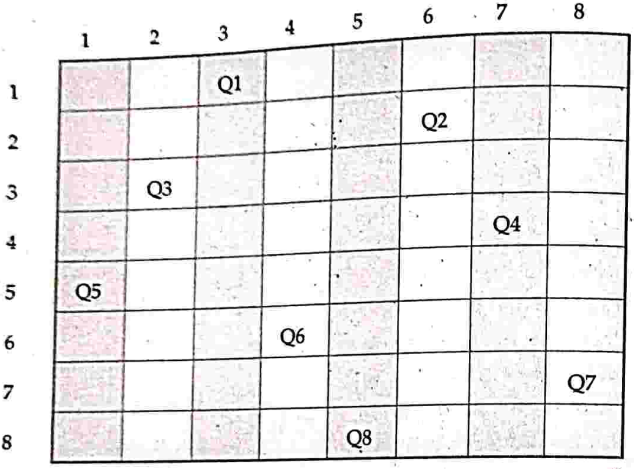


Hence we have to backtrack to adjust already placed queens.





But again Q8 cannot be placed at any empty location safely. Hence we need to backtrack. Finally the successful placement of all the eight queens can be shown by following figure.



4.4.2 Algorithm

```
Algorithm Queen(n)
//Problem description : This algorithm is for implementing n
//queen's problem
//Input : total number of queen's n.
        for column ←1 to n do
        {
            if(place(row,column))then
            {
             board[row]column//no conflict so place queen
            if(row=n)then//dead end
        print_board(n)
            //printing the board configuration
            else//try next queen with next position
                Queen(row+1,n)
            }
    }
```

This function checks if two queens are on the same diagonal or not.

Row by row each queen is placed by satisfying constraints.

```
Algorithm place(row,column)
//Problem Description : This algorithm is for placing the
//queen at appropriate position
//Input : row and column of the chessboard
//output : returns 0 for the conflicting row and column
//position and 1 for no conflict.
        for i ← 1 to row-1 do
        { //checking for column and diagonal conflicts
          if(board[i] = column)then
          return 0
        else if(abs(board[i]- column) = abs(i - row))then
              return 0
        }
        //no conflicts hence Queen can be placed
    return 1
```

Same column by 2 queen's

This formula gives that 2 queens are on same diagonal

C Functions

```
/* By this function we try the next free slot
and check for proper positioning of queen
*/
void Queen(int row,int n)
{
        int column;
        for(column=1;column<=n;column++)
        {
            if(place(row,column))

            {
              board[row] = column;//no conflict so place queen
              if(row==n)//dead end
                  print_board(n);
                   //printing the board configuration
              else  //try next queen with next position
                  Queen(row+1,n);
            }
        }
}
int place(int row,int column)
{
        int i;
        for(i=1;i<=row – 1;i++)
        { //checking for column and diagonal conflicts
            if(board[i] == column)
                return 0;
            else
                if(abs(board[i] – column) == abs(i – row))
                    return 0;
```

```
    }
    //no conflicts hence Queen can be placed
    return 1;
}
```

Example 4.4.1 *Solve 8-queen's problem for a feasible sequence (6, 4, 7, 1).*

Solution : As the feasible sequence is given, we will place the queens accordingly and then try out the other remaining places.

	1	2	3	4	5	6	7	8
1						Q		
2				Q				
3							Q	
4	Q							
5								
6								
7								
8								

The diagonal conflicts can be checked by following formula -

Let, $P_1 = (i, j)$ and $P_2 = (k, l)$ are two positions. Then P_1 and P_2 are the positions that are on the same diagonal, if

$$i + j = k + l \quad \text{or}$$

$$i - j = k - l$$

Now if next queen is placed on (5, 2) then

	1	2	3	4	5	6	7	8
1						Q		
2				Q				
3							Q	
4	Q							
5		(5,2)						
6								
7								
8								

→ (4,1) = P_1

If we place queen here then P_2 = (5,2)

4 – 1 =5 –2

∴ Diagonal conflicts occur. Hence try another position.

It can be summarized below.

Queen Positions								Action
1	2	3	4	5	6	7	8	
6	4	7	1					Start
6	4	7	1	2				As 4 – 1 = 5 – 2 conflict occurs.
6	4	7	1	3				5 – 3 ≠ 4 – 1 or 5 + 3 ≠ 4 + 1 ∴ Feasible
6	4	7	1	3	2			As 5 + 3 = 6 + 2. It is not feasible.
6	4	7	1	3	5			Feasible
6	4	7	1	3	5	2		Feasible
6	4	7	1	3	5	2	8	List ends and we have got feasible sequence.

The 8-queens on 8 × 8 board with this sequence is -

	1	2	3	4	5	6	7	8
1						Q		
2				Q				
3							Q	
4	Q							
5			Q					
6					Q			
7		Q						
8								Q

8-queens with feasible solution (6, 4, 7, 1, 3, 5, 2, 8)

Example 4.4.2 *Draw the tree organization of the 4-queen's solution space. Number the nodes using depth first search.*

Solution : The state space tree for 4-queen's problem consists of 4! leaf nodes. That means there are 24 leaf nodes. The solution space is defined by a path from root to leaf node.

The solution can be obtained for 4 queen's problem. For instance from 1 to leaf 31 represents one possible solution.

Example 4.4.3 *For a feasible sequence (7, 5, 3, 1) solve 8 queen's problem using backtracking.* **SPPU : May-14, Marks 10**

Solution : While placing the queen at next position we have to check whether th current position chosen is on the same diagonal of previous queen's position.

If $P_1 = (i, j)$ and $P_2 = (k, l)$ are two positions then P_1 and P_2 lie on the same diagona if $i + j = k + l$ or $i - j = k - l$. Let us put the given feasible sequence and try ou remaining positions.

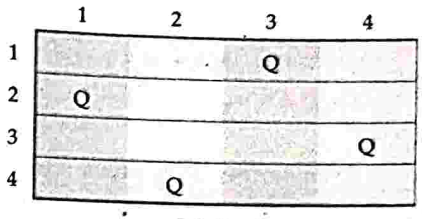
(1), 2, (3), 4, (5), 6, (7), 8 → Already occupied position.

i values i.e. row values	j → 1	2	3	4	5	6	7	8	
	7	5	3	1					
	7	5	3	1	2				The remaining vacant position is 2. But 4 – 1 = 5 – 2. Not feasible.
	7	5	3	1	4				No conflict.
	7	5	3	1	4	6			(3, 3) = (6, 6) ∴ Conflict occurs.
	7	5	3	1	4	8			List ends and we can not place further queen's.
	7	5	3	1	4				Hence backtrack.
	7	5	3	1	6				The next free slot 6 is tried.
	7	5	3	1	6	2			Not feasible because (7, 1) = (2, 6) 1 + 7 = 6 + 2. That is on same diagonal.
	7	5	3	1	6	4			(1) 2 (3) (4) (5) (6) (7) 8 Occupied are circled.
	7	5	3	1	6	4	2		
	7	5	3	1	6	4	2	8	Conflict because (3, 3) = (8, 8)
	7	5	3	1	6	4	2		Hence backtrack.
	7	5	3	1	6	4	8		(6, 5) = (8, 7). Not feasible.
	7	5	3	1	6	4			Backtrack.
	7	5	3	1	6	8			
	7	5	3	1	6	8	2		
	7	5	3	1	6	8	2	4	List ends with solution of 8 queen's.

Example 4.4.4 *Obtain any two solutions to 4-queen's problem. Establish the relationship between them.*

Solution : The solutions to 4-queen's problem are as given below.

Solution 1
(2, 4, 1, 3)

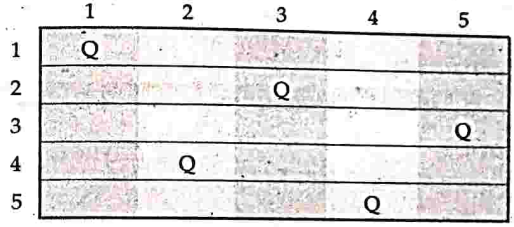


Solution 2
(3, 1, 4, 2)

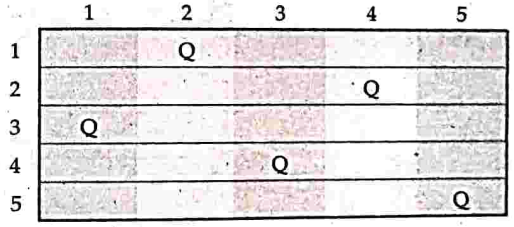
If these two solutions are observed then we can say that second solution can be simply obtained by reversing the first solution.

Example 4.4.5 *Generate at least 3 solutions for 5-queen's problem.* **SPPU : May-12, Marks 8**

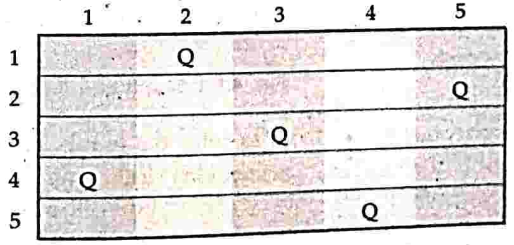
Solution : The solutions are as given below.



Solution 1
(1, 3, 5, 2, 4)



Solution 2
(2, 4, 1, 3, 5)



Solution 3
(2, 5, 3, 1, 4)

Review Questions

1. Write an algorithm to solve n queens problem using backtracking method. **SPPU : May-07, Marks 8**
2. Explain how backtracking strategy can be used to solve n-queen's problem. Give the pseudocode for the same. Discuss the time complexity for this problem. **SPPU : Dec.-07, Marks 12**
3. What is n-queen's problem ? Generate the state space tree for $n = 4$. **SPPU : Dec.-10, Marks 6**
4. Analyze the 8-queen problem using backtracking strategy of problem solving. **SPPU : May-11, Marks 8**
5. What is backtracking technique ? Find one solution for 4 - queen's problem. Show all the steps and explain why you need to backtrack. **SPPU : Dec.-13, Marks 8**
6. Write an iterative algorithm to solve n queen's problem using backtracking methods. What is the time complexity of this algorithm ? **SPPU : May-14, Marks 8**
7. Write an algorithms to solve 8-queens problem using backtracking. **SPPU : Dec.-14, Marks 8**
8. Write an algorithm to solve 4 queen's problem using backtracking method. Use mathematical modelling to support your answer. **SPPU : Aug.-15 (In Sem.), Marks 6**

4.5 Graph Coloring Problem

SPPU : May-08,10,11,14, Aug.-15, Dec.-12,13,14,15, Marks 12

Graph coloring is a problem of coloring each vertex in graph in such a way that no two adjacent vertices have same color and yet m-colors are used. This problem is also called **m-coloring problem**. If the degree of given graph is d then we can color it with d + 1 colors. The least number of colors needed to color the graph is called its chromatic number. For example : Consider a graph given in Fig. 4.5.1.

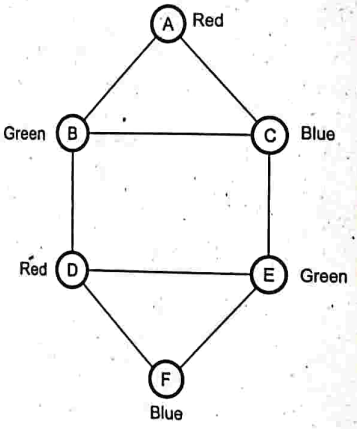
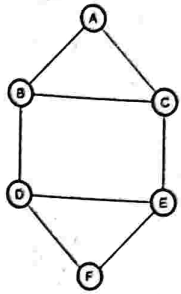


Fig. 4.5.1 Graph and its coloring

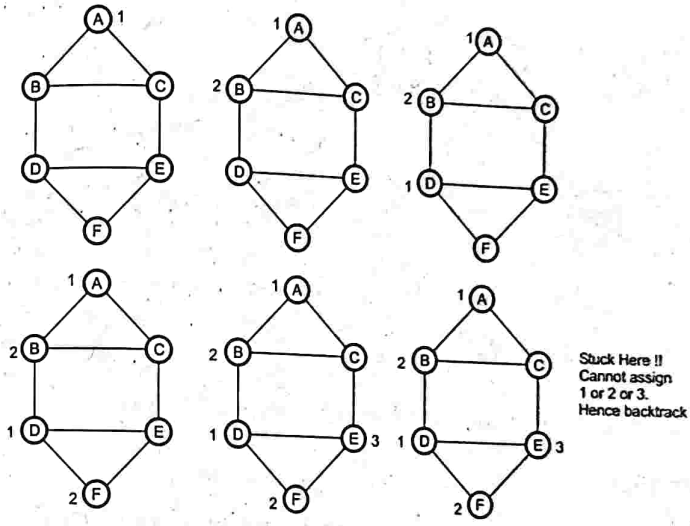
As given in Fig. 4.5.1 we require three colors to color the graph. Hence the chromatic number of given graph is 3. We can use backtracking technique to solve the graph coloring problem as follows -

Step 1 :

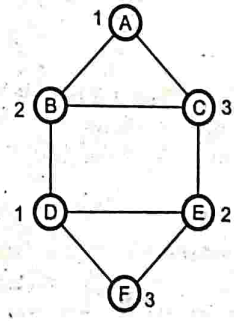
A Graph G consists of vertices from A to F. There are three colors used Red, Green and Blue. We will number them out. That means 1 indicates Red, 2 indicates Green and 3 indicates Blue color.



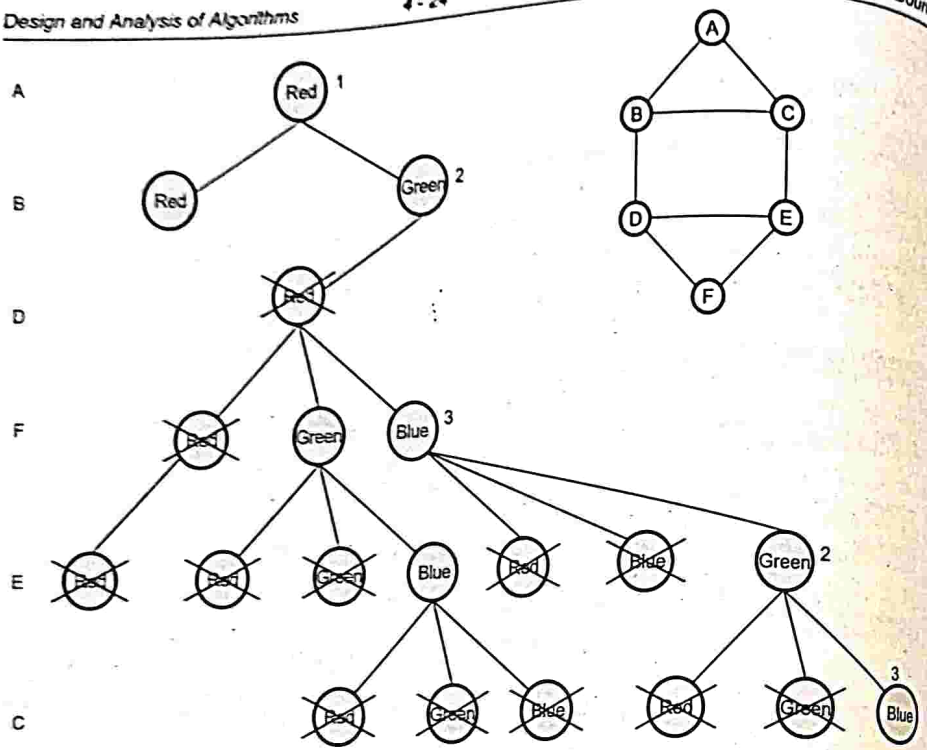
Step 2 :



Step 3 :



Thus the graph coloring problem is solved. The state-space tree can be drawn for better understanding of graph coloring technique using backtracking approach -



Here we have assumed, color index Red = 1, Green = 2 and Blue = 3.

4.5.1 Algorithm

The algorithm for graph coloring uses an important function Gen_Col_Value() for generating color index. The algorithm is -

```
Algorithm Gr_color(k)

//The graph G[1 : n,1 : n] is given by adjacency matrix.
//Each vertex is picked up one by one for coloring
//x[k] indicates the legal color assigned to the vertex
{
 repeat
 {    // produces possible colors assigned
      Gen_Col_Value(k);
if(x[k] = 0) then
   return; //Assignment of new color is not possible.
if(k=n) then // all vertices of the graph are colored
```

Takes O(nm) time

```
    write(x[1:n]); //print color index
   else
    Gr_color(k+1) // choose next vertex
   }until(false);
}
```

The algorithm used assigning the color is given as below

```
Algorithm Gen_Col_Value(k)
 //x[k] indicates the legal color assigned to the vertex
 // If no color exists then x[k] = 0
 {
// repeatedly find different colors
  repeat
   {
     x[k] ← (x[k]+1) mod (m+1) ; //next color index when it is
                                 //highest index
     if(x[k] = 0) then // no new color is remaining
           return;
     for (j ← 1 to n) do
     {
      // Taking care of having different colors for adjacent
      // vertices by using two conditions i) edge should be
      // present between two vertices
      // ii) adjacent vertices should not have same color
      if(G[k,j]!=0) AND (x[k]→x[j])) then
       break;
     }
    //if there is no new color remaining
     if ( j= n+1) then
        return;
   }until(false);
}
```

4.5.2 Analysis

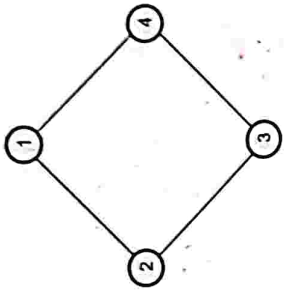
The Gr_color takes computing time of $\sum_{i=0}^{n-1} m^i$. Hence total computing time for this algorithm is $O(nm^n)$.

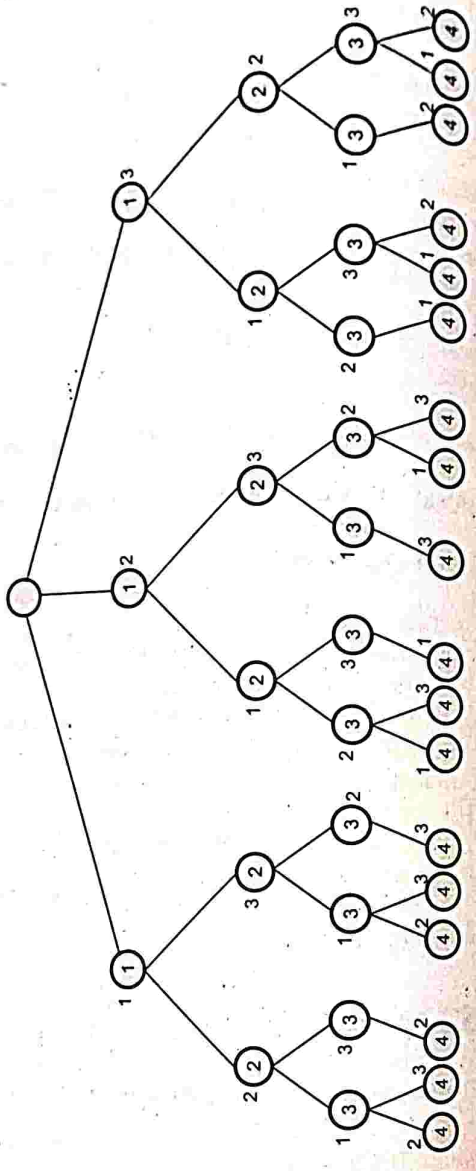
If we trace the above algorithm then we can assign distinct colors to adjacent vertices. For example : Consider, a graph given below can be solved using three colors.

Consider,

Red = 1, Green = 2 and Blue = 3.

The number inside the node indicates vertex number and the number outside the node indicates color index.





C Function

```
void Gen_Col_Value(int k,int n)
{
        int j;
        int a,b;
        while(1)
```

```
    {
    a=color_tab[k]+1;
    b=m+1;
    color_tab[k] = a%b; // next highest color
    if(color_tab[k]==0) return;    // all colors have been used
    for(j=1;j<=n;j++)
    {
    // check if this color is distinct from adjacent colors
    if(G[k][j] && color_tab[k]==color_tab[j])
     break;
    }
    if(j==n+1) return; // next new color found
  }
}
// such that adjacent vertices are assigned distinct integers
// k is the index of next vertex color.
void Gr_coloring(int k,int n)
{
      Gen_Col_Value(k,n);
      if(color_tab[k]==0) return;   // No new color available
if(k==n)  return;     // at most m colors have been
      else Gr_coloring(k+1,n);  // used to color the n vertices
}
```

Strategy to Solve Graph Coloring Problem

1. Choose some color say c and apply to some vertex say v.
2. Go on choosing different colors that are available from the list and apply to each successive vertex. After applying the color just check whether it is same with the adjacent color or not.
3. If the chosen color is not same to the adjacent colors then go to the next adjacent vertex otherwise try out some another color from the list or change the already assigned colors.
4. Repeat this procedure for coloring all the vertices.

Example 4.5.1 *Construct plannar graph for following map. Explain how to find m-coloring of this planar graph by using m-coloring backtracking algorithm.*

SPPU : May-11, Marks 10

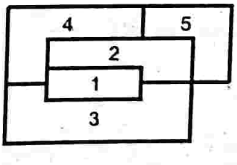


Fig. 4.5.2

Solution :

Steps for graph coloring -

1. Number out each vertex (V0, V1, V2,...Vn-1)
2. Number out the available colors (C0,C1,C2,...Cn-1)
3. Start assigning Ci to each Vi. While assigning the colors note that two adjacent vertices should not be colored by the same color. And least number of colors should be used.
4. While assigning the appropriate color, just backtrack and change the color if two adjacent colors are getting the same.

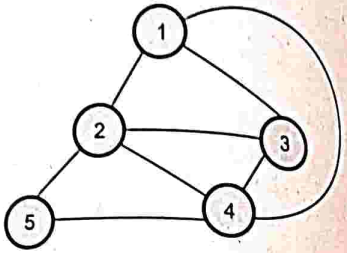


Fig. 4.5.3

Review Questions

1. *Write recursive backtracking schema for m-coloring of the graph. Determine the time complexity of the same.* **SPPU : May-08, 10, Marks 12**
2. *Discuss and analyze the problem of graph coloring using backtracking.* **SPPU : Dec.-12, Marks 6**
3. *Explain graph coloring problem.* **SPPU : Dec.-13, Marks 8**
4. *Explain graph coloring problem with respect to blacktracking.* **SPPU : Dec.-14, Marks 8**
5. *Write recursive algorithm on graph coloring using backtracking strategy. Determine the time complexity of the same.* **SPPU : May-14, Marks 8**
6. *Write an algorithm for graph colouring problem using backtracking method.* **SPPU : Aug.-15 (In Sem.), Marks 6**
7. *Write a short note on graph coloring problem. Write algorithm for the same.* **SPPU : Dec.-15 (End Sem.), Marks 6**

4.6 Sum of Subsets Problem

SPPU : May-10,11, Dec.-08,09,11,12, Aug.-15, Marks 12

Problem Statement

Let, $S = \{S_1, ... S_n\}$ be a set of n positive integers, then we have to find a subset whose sum is equal to given positive integer d.

It is always convenient to sort the set's elements in ascending order. That is,

$$S_1 \leq S_2 \leq ..... \leq S_n$$

Let us first write a general algorithm for sum of subset problem.

Algorithm

Let, S be a set of elements and d is the expected sum of subsets. Then -

Step 1 : Start with an empty set.

Step 2 : Add to the subset, the next element from the list.

Step 3 : If the subset is having sum d then stop with that subset as solution.

Step 4 : If the subset is not feasible or if we have reached the end of the set then backtrack through the subset until we find the most suitable value.

Step 5 : If the subset is feasible then repeat step 2.

Step 6 : If we have visited all the elements without finding a suitable subset and if backtracking is possible then stop without solution. This problem can be well understood with some example.

Example 4.6.1 *Consider a set S = {5, 10, 12, 13, 15, 18} and d = 30. Solve it for obtaining sum of subset.*

Solution :

Initially subset = { }	Sum = 0	–
5	5	Then add next element.
5, 10	15 ∵ 15 < 30	Add next element.
5, 10, 12	27 ∵ 27 < 30	Add next element.
5, 10, 12, 13	40	Sum exceeds d = 30 hence backtrack.
5, 10, 12, 15	42	Sum exceeds d = 30 ∴ Backtrack
5, 10, 12, 18	45	Sum exceeds d ∴ Not feasible. Hence backtrack.
5, 10		
5, 10, 13	28	
5, 10, 13, 15	33	Not feasible ∴ Backtrack.
5, 10		
5, 10, 15	30	Solution obtained as sum = 30 = d

∴ The state space tree can be drawn as follows.

{5, 10, 12, 13, 15, 18}

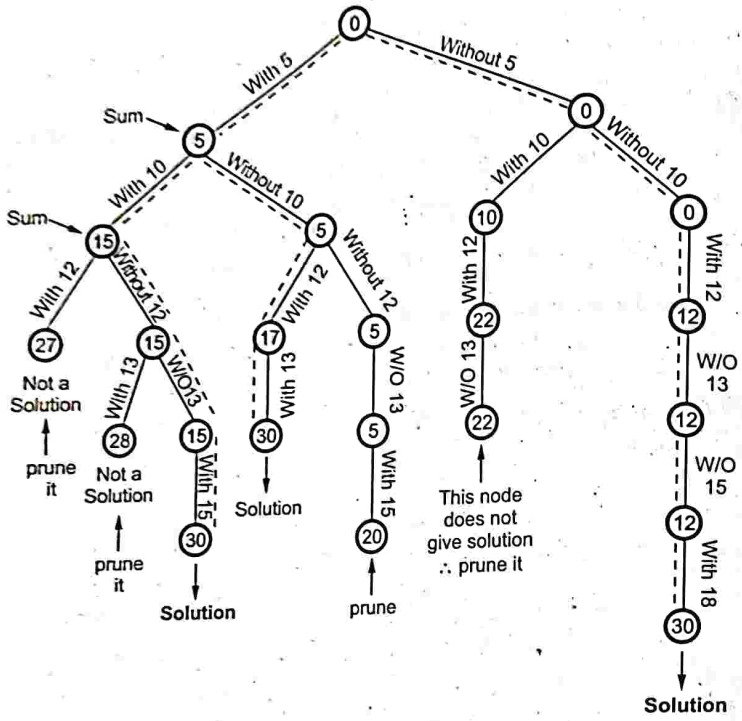


Fig. 4.6.1 State space tree for sum of subset

Example 4.6.2 *Let m = 31 and W = {7, 11, 13, 24}. Draw a portion of state space tree for solving sum-of-subset problem for the above given algorithm.*

Solution : Initially we pass (0, 1, 55) to sum-of-subset function. The sum = 0, index = and remaining_sum = 55 initially [∵ 7 + 11 + 13 + 24].

The recursive execution of the algorithm will be,

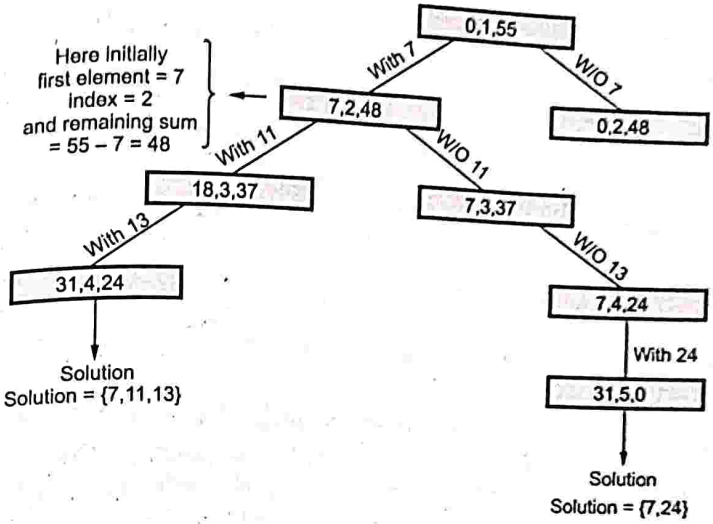


Fig. 4.6.2

Example 4.6.3 *Analyze sum of subset algorithm on data :*

M = 35 and

i) W = {5, 7, 10, 12, 15, 18, 20} ii) W = {20, 18, 15, 12, 10, 7, 5}

iii) W = {15, 7, 20, 5, 18, 10, 12}

Are there any discernible differences in the computing time ? **SPPU : May-11, Marks 10**

Solution :

Initially Subset = { }	**Sum = 0**	
5	5	Then add next element
5, 7	12 < 35	Add next element
5, 7, 10	22 < 35	Add next element
5, 7, 10, 12	34 < 35	Add next element
5, 7, 10, 12, 15	49 > 35	Sum exceeds ∴ Backtrack
5, 7, 10, 15	37 > 35	Backtrack
5, 7, 12	24 < 35	Select next element
5, 7, 12, 15	39 > 35	Backtrack
5, 10	15 < 35	Add next element

5, 10, 12	27< 35	Select next element
5, 10, 12, 15	42 > 35	Backtrack
5, 10, 15	30 < 35	Select next element
5, 10, 15, 18	48 > 35	Backtrack
5, 10, 18	33 < 35	Select next element
5, 10, 18, 20	53 > 35	Backtrack
5, 10, 20	35	Solution is found

The possible solutions are {5, 10, 20}, {15, 20} and {5, 12, 18}.

The sequence (ii) W = {20, 18, 15, 12, 10, 7, 5} and (iii) W = {15, 7, 20, 5, 18, 10, 12} are not in non-decreasing order. The (ii) sequence is in decreasing order and (iii) is totally unsorted.

For sum of subset problem, For efficient findings of solution the weights must be arranged in non-decreasing order. to show how this order is beneficial consider a scenario we have selected the elements {5, 7, 10, 12} which sums up 34. Now if we select the next element 15, then this sum will exceed the limit because 49 > 35.

So we will simply backtrack. It is not even necessary to examine the remaining elements {18, 20} of the list. If at all, the list is arranged according to decreasing order of the weights then we have to analyse all the elements of the given list. Similarly if the list is unsorted then each and every element needs to be analysed for required sum. Thus the sequences (ii) and (iii) makes the computing time worst.

Example 4.6.4 *What is backtracking method for algorithmic design ?*

Solve the sum of subset problem using backtracking algorithmic strategy for the following data

$N = 4\ (w_1, w_2, w_3, w_4) = (11, 13, 24, 7)$ *and* $M = 31$. **SPPU : Dec.-11,12, Marks 12**

Solution :

Initially subset = { }	Sum = 0	
11	13 < 31	Add next element
11, 13	24 < 31	Add next element
11, 13, 24	48 > 31	Backtrack
11, 13	24 < 31	Add 7
11, 13, 7	31	Solution is found i.e. { 11, 13, 7 }

The state space tree can be as given below -

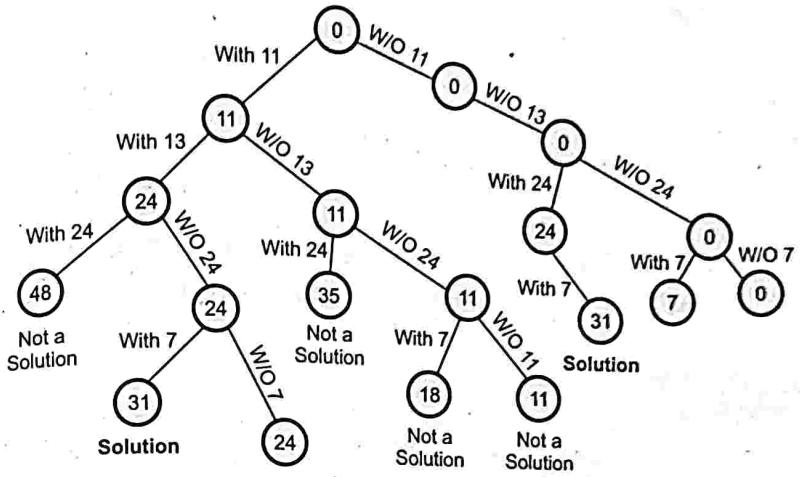


Fig. 4.6.3

Thus subsets {11, 13, 7} and {24, 7} are the solutions.

Example 4.6.5 *Draw a pruned state space tree for a given sum of subset problem.*
$S = \{3, 4, 5, 6\}$ *and* $d = 13$

Soltuion : Fig. 4.6.4 state space tree the total sum is given inside each node. Pruned nodes has cross at its bottom.

Hence the solution is **{3, 4, 6}.**

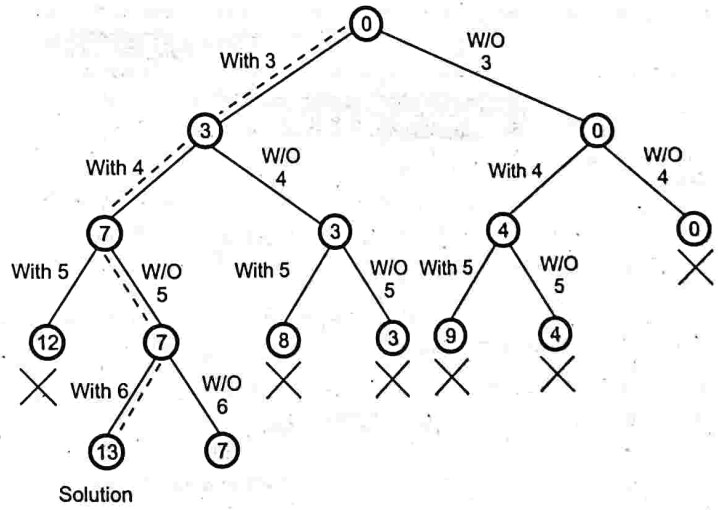


Fig. 4.6.4

Strategy to Solve Sum of Subsets Problem

1. Choose each number and sum it up. Compare it with the given sum d.
2. If the sum of chosen numbers is less than d then add the next subsequent element.
3. If the sum of chosen numbers is greater than d then remove one element and try some another number in the set.
4. Thus all the permutations of the numbers for summing up should be attempted.
5. When we get the sum=d then declare the chosen set of numbers as the solution.

Examples for Practice

Example 4.6.6 : *Apply backtracking algorithm to solve the instace of the sum of subset problem* $s=\{1,3,4,5\}$ *and* $d=11$.

Example 4.6.7 : *Draw the state space tree for sum of subset problem of the given instance* $s=\{5,7,8,10\}$ *and* $d = 15$.

Review Questions

1. *Write a recursive backtracking algorithm for sum of subsets of problem.*
   SPPU : Dec.-08, 09, May-10, Marks 8
2. *Explain in detail sum of subet problem using backtracking method with suitable example.*
   SPPU : Aug.-15 (In Sem.), Marks 4

Part II : Branch and Bound

4.7 Principle

- Branch and bound is a general algorithmic method for finding **optimal solutions** of various optimization problems.
- Branch and bounding method is a general optimization technique that applies where the greedy method and dynamic programming fail. However, it is much **slower.**
- It often leads to exponential time complexities in the worst case. On the other hand, if applied carefully, it can lead to algorithms that run reasonably fast on average.

4.8 Control Abstraction

SPPU : Dec.-06,07, May-07,09,19, March-20, Marks 8

- In branch and bound method a state space tree is built and all the children of E nodes (a live node whose children are currently being generated) are generated before any other node can become a live node.
- For exploring new nodes either a **BFS** or **D-search** technique can be used.
- In Branch and bound technique, **BFS-like** state space search will be called **FIFO** (First In First Out) search. This is because the list of live node is First In First Out list (**queue**). On the other hand the **D-search** like state space search will be called **LIFO** search because the list of live node is Last in First out list (**stack**).
- In this method a space tree of possible solutions is generated. Then partitioning (called as branching) is done at each node of the tree. We compute lower bound and upper bound at each node. This computation leads to selection of answer node.
- **Bounding functions** are used to **avoid the generation of subtrees that do not contain** an answer node.

4.8.1 General Algorithm for Branch and Bound

The algorithm for branch and bound method is as given below.

```
Algorithm Branch_Bound()
{
//E is a node pointer;
 E←new(node); // This is the root node which is the dummy
                        //start node
 //H is  heap for all the live nodes.
 // H is a min-heap for minimization problems,
 //H is a max-heap for maximization problems.
 while (true)
 {
  if (E is a final leaf) then
  {
        //E is an optimal solution
   write( path from E to the root);
   return;
  }
  Expand(E);
  if (H is empty) then //if no element is present in heap
        {
              write(" there is no solution");
              return;
        }
```

```
    E ← delete_top(H);
  }
}
```

Following is an algorithm named Expand is for generating state space tree -

```
Algorithm Expand(E)
{
  Generate all the children of E;
  Compute the approximate cost value of each child;
  Insert each child into the heap H;
}
```

For Example

Consider the 4-queens problem using branch and bound method. In branch and bound method a bounding function is associated with each node. The node with-optimum bounding function becomes an E-node. And children of such node get expanded. The state space tree for the same is as given below -

In Fig. 4.8.1 based on bounding function the nodes will be selected and answer node can be obtained. The numbers that are written outside the node indicates the order in which evaluation of tree takes place. (Refer Fig. 4.8.1 on next page)

Review Questions

1. *Explain in brief : Branch and bound method.* **SPPU : Dec.-06, Marks 6**
2. *Write a short note on branch and bound method.* **SPPU : May-07, 09, Marks 8**
3. *Describe in brief the general strategy used in branch and bound method.* **SPPU : Dec.-07, Marks 4**
4. *Explain branch and bound approach with suitable example. What are general characteristics of branch and bound ?* **SPPU : May-19, Marks 8**
5. *Write short note on - Branch and bound approach.* **SPPU : March-20, Marks 5**

4.9 Strategies - FIFO, LIFO and LC Approaches

SPPU : Dec.-08, 10, 14, May-13,14, Marks 8

- In branch and bound method the basic idea is selection of E-node. The selection of E-node should so perfect that we will reach to answer node quickly.
- Using FIFO and LIFO branch and bound method the selection of E-node is very complicated and somewhat blind.
- FIFO stands for First In First Out, LIFO stands for Last In First Out and LC stands for Least Cost search.

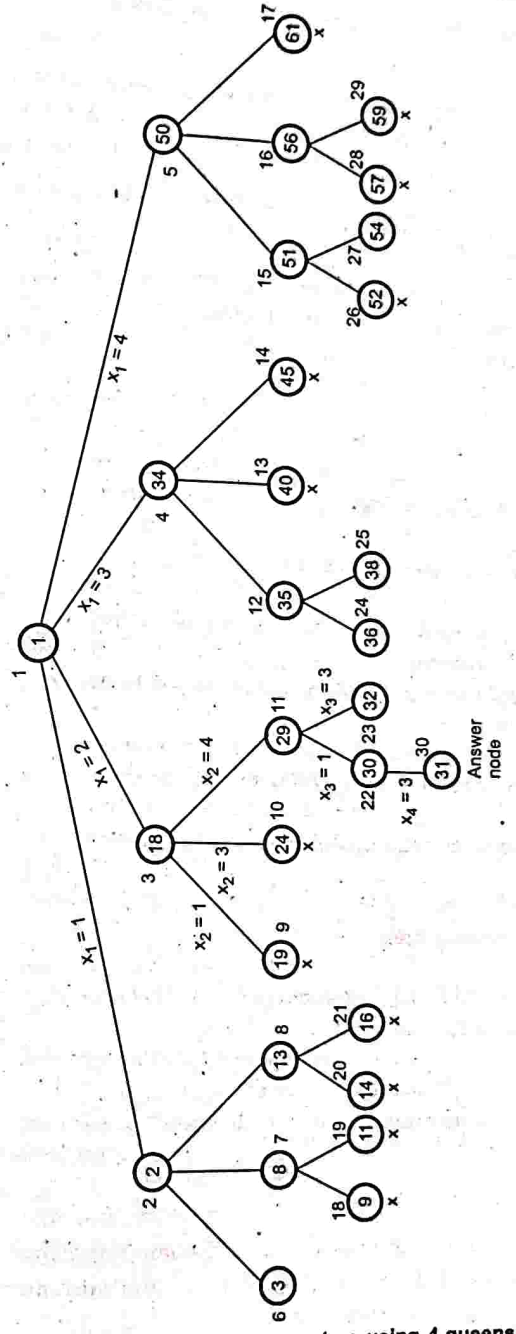


Fig. 4.8.1 Portion of state space tree using 4-queens

Algorithm for FIFO Branch and Bound is as given below

```
Algorithm FIFOBB()
//tr is a state space tree and x be the node in the tr
//E represents the E-node
//initialize the list of live nodes to empty
{
    if( tr is answer node) then
    {
        u:=min(cost[tr],(upper(tr)+E))
        write(tr);//output the answer node
        return;
    }
    E←tr //set the E-node
    repeat
    {
    for(each child x of E) do
    {
        if(x is answer node) then
        {
            output the path from x to root;
            return;
        } //the new live node x will be added in the list of live //nodes
    Insert_Q(x);
    x.parent←E;//pointing the path from x to root;
        if((x is answer node) AND cost(x)<u) then
        {
        u←min(cost[tr],(upper(tr)+E))
    }
 }
 if(no more live nodes) then
 {
        write("No more Live nodes now");
        write("least Cost is =",u);
        return;
 }                    // E points to current E-node
 E← Del_Q(); // deletes the least cost node from Q to set next
        //E-node
 } until(false);
}
```

In above algorithm if x is in **tr** then **c(x)** be the minimum cost answer node in ***tr.*** The computation of **u** value is done. We always choose the **minimum values of cost of the children** generated from the current E node.

The algorithm uses two functions Del_Q and () and Insert_Q() function to delete or add the live node from the list of live nodes stored in queue. Using above algorithm we can obtain path from answer node to root. We can use parent to trace the parent of node x. Initially root is the E node. The for loop used in the algorithm examines all the children of E-node for obtaining the answer node.

Algorithm for LIFO Branch and Bound is as given below

```
Algorithm LIFOBB()
//tr is a state space tree and x be the node in the tr
//E represents the E-node
//initialize the list of live nodes to empty
{
        if( tr is answer node) then
                {
                u←min(cost[tr],(upper(tr)+E))
                write(tr) ; //output the answer node
                return;
                }
                E←tr //set the E-node
                repeat
                {
                for(each child x of E) do
                {
                if(x is answer node) then
                {
                output the path from x to root;
                return;
            }
    //the new live node x will be added in the list of live nodes
       PUSH(x);
       x.parent← E;//pointing the path from x to root;
       if(  (x is answer node) AND cost(x)<u) then
            {
                u←min(cost[tr],(upper(tr)+E))
       }
   }
   if(no more live nodes) then
   {
       write("No more Live nodes now");
       write("least Cost is =",u);
       return;
   }
// E points to current E-node
```

```
    E← POP();
// deletes the least cost node from stack to set next E-node
} until(false);
}
```

In above algorithm if x is in tr then c(x) be the minimum cost answer node in **tr.** The computation of u value is done. We always choose the minimum values of cost of the children generated from the current E node.

The algorithm uses two functions **POP()** and **PUSH()** function to delete or add the live node from the list of live nodes stored in **stack.** Using above algorithm we can obtain path from answer node to root. We can use parent to trace the parent of node x. Initially root is the E node. The for loop used in the algorithm examines all the children of E-node for obtaining the answer node.

- For speeding up the search process we need to intelligent ranking function for live nodes. Each time, the next E-node is selected on the basis of this ranking function. For this ranking function additional computation (normally called as cost) is needed to reach to answer node from the live node.
- The **Least Cost (LC)** search is a kind of search in which least cost is involved for reaching to answer node. At each E-node the probability of being an answer node is checked.
- **BFS** and **D-search** are special cases of LC search.
- Each time the next E-Node is selected on the basis of the ranking function (smallest c^(x)). Let g^(x) be an estimate of the additional effort needed to reach an answer node from x. Let h(x) to be the cost of reaching x from the root and f(x) to be any non-decreasing function such that,

  $$c^(x) = f(h(x)) + g^(x)$$

- If we set g^(x)=0 and f(h(x)) to be level of node x then we have BFS.
- If we set f(h(x))=0 and $g^(x) \leq g^(y)$ whenever y is a child of x then the search is a D-search.
- An LC search with bounding functions is known as LC Branch and Bound search.
- In LC search, the cost function c(.) can be defined as,
  - i) If x is an answer node then c(x) is the cost computed by the path from x to root in the state space tree.
  - ii) If x is not an answer node such that subtree of x node is also not containing the answer node then $c(x) = \infty$.
  - iii) Otherwise c(x) is equal to the cost of minimum cost answer node in subtree x.
- c^(.) with f(h(x)) = h(x) can be an approximation of c(.).

Control Abstraction for LC Search

SPPU : Dec.-08, Marks 8; Dec.-10, Marks 6

The control abstraction for Least cost search is given as follows.

```
Algorithm LC_search()
//tr is a state space tree and x be the node in the tr
//E represents the E-node
//initialize the list of live nodes to empty
{       if( tr is answer node) then
        {
              write(tr);//output the answer node
              return;
        }
        E←tr //set the E-node
        repeat
        {
              for(each child x of E) do
              {
                    if(x is answer node) then
                    {
                          output the path from x to root;
                          return;
                    }
//the new live node x will be added in the list of live nodes
              Add_Node(x);
              x → parent ← E;//pointing the path from x to root;
              }
              if(no more live nodes) then
              {
                    write("Can not have answer node!!");
                    return;

              }
// E points to current E-node
              E← Least_cost(); //finds the least cost node to set
              next E-node
      }until(false);
}
```

In above algorithm if x is in tr then c(x) be the minimum cost answer node in tr. The algorithm uses two functions Least_cost and Add_Node() function to delete or add the live node from the list of live nodes. Using above algorithm we can obtain path from answer node to root. We can use parent to trace the parent of node x. Initially root is the E node. The for loop used in the algorithm examines all the children of E-node for obtaining the answer node. The function Least_cost() looks for the next possible E-node.

Review Questions

1. *Explain in detail control abstraction for LC search.* **SPPU : Dec.-08, Marks 8; Dec.-10, May-13, Marks 6**
2. *Write the control abstraction for LC-search. Explain how travelling salesperson problem is solved using LCBB.* **SPPU : May-14, Marks 8**
3. *What is branch and bound method ? Explain FIFO branch and bound algorithm.* **SPPU : Dec.-14, Marks 8**

4.10 Concept of Bounding

- As we know that the bounding functions are used to avoid the generation of su trees that do not contain the answer nodes. In bounding lower bounds and upp bounds are generated at each node.
- A cost function c^(x) is such that c^(x) ≤ c(x) is used to provide the lower bounds o solution obtained from any node x.
- Let upper is an upper bound on cost of minimum-cost solution. In that case, all th live nodes with c^(x)> upper can be killed.
- At the start the upper is usually set to ∞. After generating the children of curre E-node, upper can be updated by minimum cost answer node. Each time a ne answer node can be obtained.

4.10.1 Job Sequencing with Deadlines

We will consider an example of job sequencing with deadlines to understand th computation of c^(x) and upper. Let us see the problem statement first.

Problem statement :

Let there be n jobs with different processing times. With only one processor the job are executed on or before given deadlines. Each job i is given by a tuple (P_i, d_i, t_i) wher t_i is the processing time required by job i. If processing of job i is not completed b deadline d_i then penalty P_i will occur. The objective of this problem is to select a subs J such that penalty will be minimum among all possible subsets. Such a J should b optimal subset.

Example 4.10.1 *Let n = 4*

Job index	P_i	d_i	t_i
1	5	1	1
2	10	3	2
3	6	2	1
4	3	1	1

Then select an optimal subset J with optimal penalty. What will be the penalty corresponding to optimal solution ?

Solution : The state space tree can be drawn for proper selection of job i for creating J. There are two ways by which the state space tree can be drawn : Fixed tuple size formulation and variable tuple size formulation. The variable tuple size formulation can be as shown below.

The function c^(x) can be computed for each node x as

$$c\hat{}(x) = \sum_{i<m \; i \in S_x} P_i$$

where $m = \max \{ i | i \in S_x \}$

and S_x be subset of jobs selected for J at node x.

The upper bound can be computed using function u(x). Then,

$$u(x) = \sum_{i \notin S_x} P_i$$

Here u(x) corresponds to the cost of the solution S_x for node x.

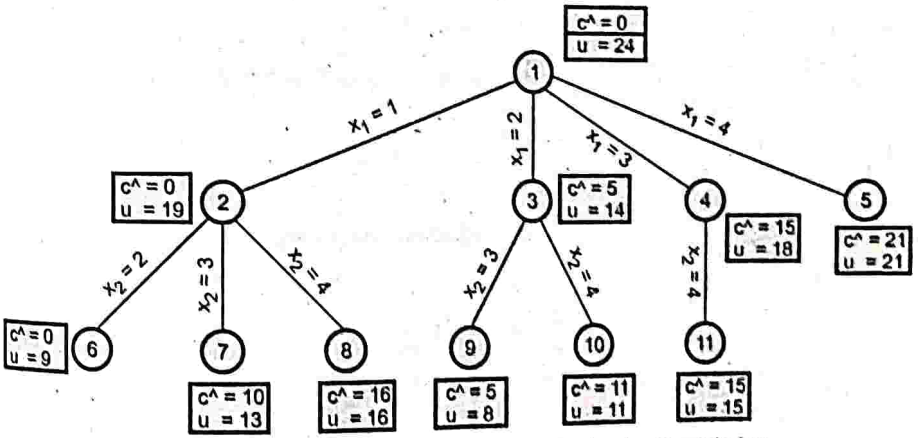


Fig. 4.10.1 State space tree with variable tuple size formulation

For node 1 (root) there are 4 childrens. These children are for selection of either 1 or 2 or 3 or for job 4. Hence

$x_1 = 2$ or $x_1 = 3$ $x_1 = 4$.

For node 2 we have with $x_1 = 1$ and therefore we can select either 2, 3 or 4. Hence $x_2 = 2$ or $x_2 = 3$ or $x_2 = 4$ corresponds to node 6, 7 or 8. Continuing in this fashion, we have drawn the state space tree.

At node 1 :

$c\hat{} = 0$

$u = 24$ $\because$ $\sum_{i=1}^{n} P_i = 5 + 10 + 6 + 3$

At node 2 :

$c\hat{} = 0$ $\because$ $\sum P_i$ where $i < 1 = 0$

$u = 19$ $\because$ the sum of P_i excluding $x_1 = 1$, i.e. $P_1 = 5$.

At node 3 :

$c\hat{} = 5$ $\because$ $\sum P_i$ where $i < 2$

$u = 14$ $\because$ $\sum P_i$ without $x_1 = 2$ i.e. $24 - 10 = 14$

At node 4 :

$c\hat{} = 15$ $\because$ $\sum P_i$ where $i < 3$

$u = 18$ $\because$ $\sum P_i$ without P_3 i.e. $24 - 6 = 18$

At node 5 :

$c\hat{} = 21$ $\because$ $\sum P_i$ where $i < 4$

$u = 21$ $\because$ $\sum P_i$ without P_4 i.e. $24 - 3 = 21$

At node 6 :

$c\hat{} = 0$ $\because$ $\sum_{i<2} P_i$ and not containing $x_1 = 1$ i.e. P_1

$u = 9$ $\because$ $\sum_{i=1}^{n} P_i$ such that $i \neq 2$ and $i \neq 1$. Hence $6 + 3 = 9$

At node 7 :

$c\hat{} = 10$ ∵ $\sum_{i<3} P_i$ where $i \neq 1$ and $i \neq 3$. Hence $P_2 = 10$.

$u = 13$ ∵ $\sum_{i=1}^{n} P_i$ such that $i \neq 1$ and $i \neq 3$. Hence $P_2 + P_4 = 13$

At node 8 :

$c\hat{} = 16$ ∵ $\sum_{i<4} P_i$ and $i \neq 1$ and $i \neq 4$. Hence $P_2 + P_3 = 16$.

$u = 16$ ∵ $\sum_{i=1}^{n} P_i$ and $i \neq 1$ and $i \neq 4$. Hence $P_2 + P_4 = 16$

At node 9 :

$c\hat{} = 5$ ∵ $\sum_{i<3} P_i$ but $i \neq 2$. Hence $P_1 = 5$.

$u = 8$ ∵ $\sum_{i=1}^{n} P_i$ but $i \neq 2$ and $i \neq 3$. That is $P_1 + P_4 = 8$

At node 10 :

$c\hat{} = 11$ ∵ $\sum_{i<4} P_i$ but $i \neq 2$. Hence $P_1 + P_3 = 5 + 6 = 11$.

$u = 11$

At node 11 :

$c\hat{} = 15$ ∵ $\sum_{i<4} P_i$ and $i \neq 3$. Hence $P_1 + P_2 = 5 + 10 = 15$

$u = 15$ ∵ $\sum_{i=1}^{n} P_i$ and $i \neq 3$ and $i \neq 4$. Hence $P_1 + P_2 = 5 + 10 = 15$

Now we will draw fixed tuple size formulation.

In fixed tuple size formulation, $x_i = 1$ means job i is selected $x_i = 0$ means job i is not selected for formation of J (the optimal subset). Initially, $c\hat{} = 0$ for node 1 as no job is selected. For node 3, it indicates omission of job 1. Hence, penalty is 5. For node 5 there is omission of job 2. Hence $c\hat{}$ = penalty = 10. For node 7 there is omision of job 1 and job 2. Hence $c\hat{}$=15. For node 13 there is omission of job 1 and job 3. Hence penalty is 11. Therefore $c\hat{} = 11$. Continuing in this fashion $c\hat{}$ are computed.

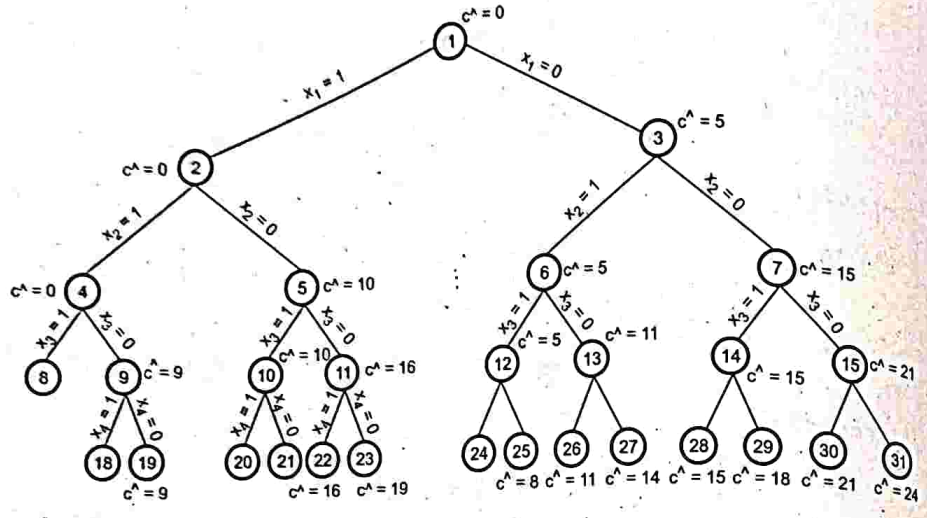


Fig. 4.10.2 State space tree for fixed tuple size formulation

4.10.2 FIFO Branch and Bound

To understand FIFO branch and bound, we will consider the variable tuple size formulation. The state space tree can be drawn as below. For job sequencing problem as-

For node u = 24, the upper is now 24. Then nodes 2, 3, 4 and 5 are generated u(2) = 19, u(3) = 14, u(4) = 18, u(5) = 21. The minimum value of u(x) will be upper. Hence upper will be updated to 14.

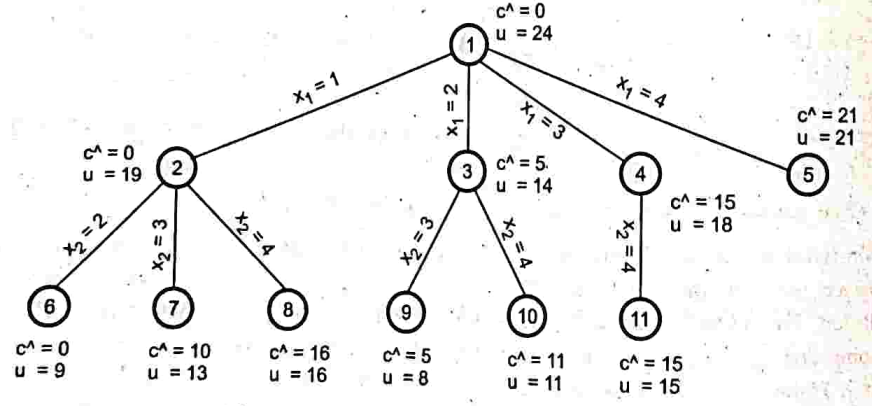


Fig. 4.10.3 Variable tuple sized formulation

upper = 14
For node 4
c^(4) = 15
c^(4) > upper
i.e. 15 > 14
Hence kill node 4.

upper = 14
For node 5
c^(5) = 21
c^(5) > upper
i.e. 21 > 14
Hence kill node 5.

Node 2 and 3 become live nodes. So, now we will consider 2 and 3. We will expand node 2 (2 becomes E-node). u(6) = 9, u(7) = 13, u(8) = 16. Hence minimum of u(x) becomes upper. As u(6) = 9 is minimum. The upper will be updated to 9.

Hence,

upper = 9
For node 7
c^(7) = 10
c^(7) > upper
Hence kill node 7.

upper = 9
For node 8
c^(8) = 16
c^(8) > upper
Hence node 8
is not considered

The live node 3 being live node becomes E-node now. Now children 9 and 10 are generated. u(9) = 8 and u(10) = 11. Hence upper is updated to 8.

Hence,

upper = 8
For node 10
c^(10) = 11
c^(10) > upper
Hence kill node 10.

Now node 6 becomes E-node. But as children of node 6 are infeasible we will not consider node 6. The only remaining live node is 9. The only child of 9 is infeasible. Now there is no live node remaining the minimum cost answer node is node 9. And it has a cost of 8. This method is referred as FIFO based branch and bound.

4.10.3 LC Branch and Bound

In LC branch and bound method. For node 1 upper = 24. Now, node 1 becomes an E-node. Then node 2, 3, 4 and 5 are generated. The upper is now 14. As c^(4) > upper and c^5 > upper. Hence node 4 and 5 are killed. For node 2 and 3 the cost c^(2) is 0, node 2 becomes an E-node. Hence children node 6, 7 and 8 are generated. The upper is now 9 (because u(6) = 9). Node 7 and 8 are killed. Node 6 is selected as it has minimum cost. But both the children of node 6 are infeasible. So kill node 6. Now, node 3 becomes E-node. Then node 9 and 10 are generated. The upper = 8 now, since c^(10) > upper we will kill node 10. Now only remaining node is 9. Next E-node becomes node 9. Its only child is infeasible. As there are no live nodes remaining, we will terminate search with node 9 as an answer node. The principle idea in LC search is choosing of minimum cost node each time. (Fig. 4.10.4)

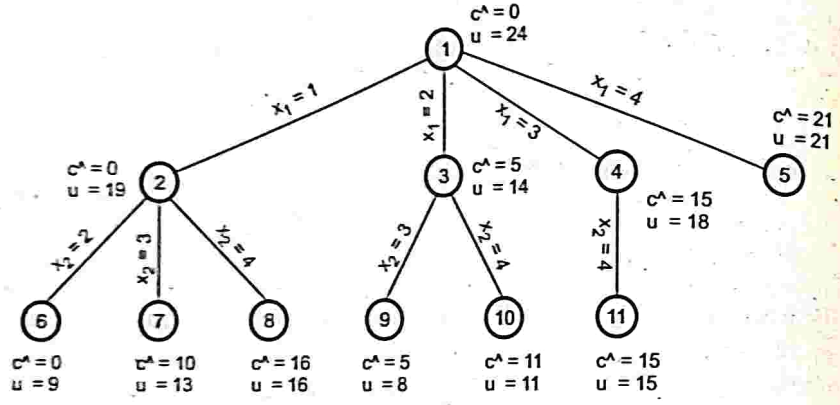


Fig. 4.10.4

This method of LC based branch and bound with appropriate c^(.) and u(.) is called as LCBB (LIFOBB).

4.11 Traveling SalesPerson Problem

SPPU : May-08,12,14, Dec.-08,11,12,13,14,15, Aug.-15, Marks 18

Problem Statement

If there are n cities and cost of travelling from any city to any other city is given. Then we have to obtain the cheapest round-trip such that each city is visited exactly once and then returning to starting city, completes the tour.

Typically travelling salesperson problem is represented by weighted graph.

For example : Consider an instance for TSP is given by G as,

$$G = \begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

There n = 5 nodes. Hence we can draw a state space tree with 5 nodes as shown in Fig. 4.11.1. (See Fig. 4.11.1 on next page.)

Tour(x) is the path that begins at root and reaching to node x in a state space tree and returns to root. In branch and bound strategy cost of each node x is computed. The travelling salesperson problem is solved by choosing the node with optimum cost. Hence,

$$c(x) = \text{cost of tour (x)}$$

where x is a leaf node.

The $c\hat{}(x)$ is the approximation cost along the path from the root to x.

4.11.1 Row Minimization

To understand solving of travelling salesperson problem using branch and bound approach we will reduce the cost of the cost matrix M, by using following formula.

$$\text{Red\_Row (M)} = \left[M_{ij} - \min\left\{ M_{ij} \middle| 1 \le j \le n \right\} \right] \quad \text{where } M_{ij} < \infty$$

For example : Consider the matrix M representing cost between any two cities.

$$M = \begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

We will find minimum of each row.

$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix} \begin{matrix} 10 \\ 2 \\ 2 \\ 3 \\ 4 \end{matrix}$$

21 Total reduced cost.

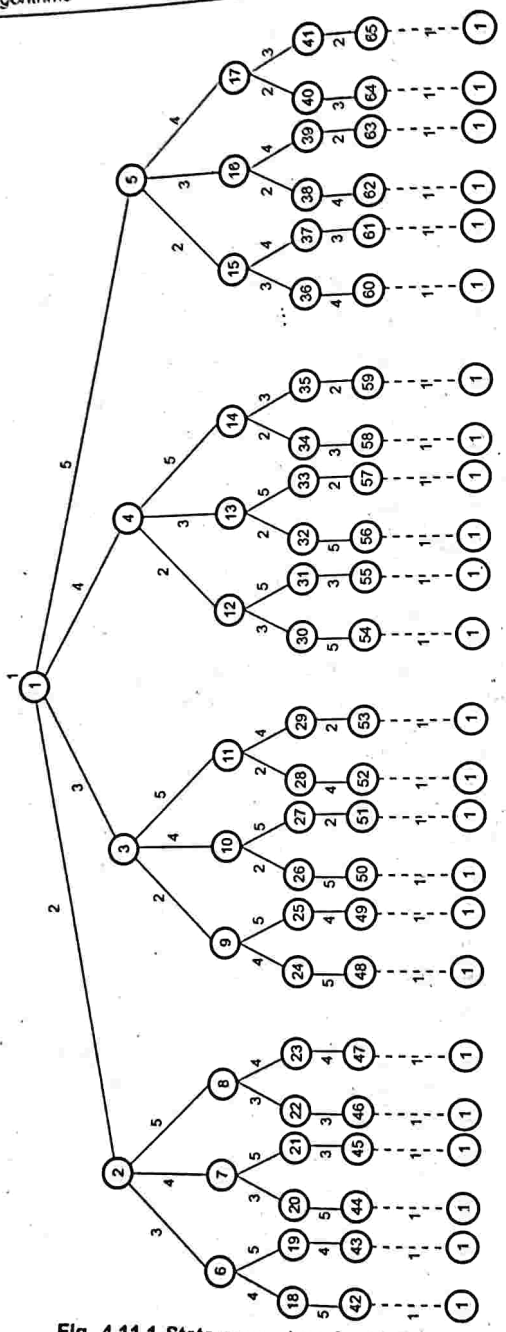


Fig. 4.11.1 State space tree for TSP

We will subtract the row_minimum value from corresponding row. Hence,

$$\text{Red\_Row(M)} = \begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 2 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix}$$

4.11.2 Column Minimization

Now we will reduce the matrix by choosing minimum from each column. The formula for column reduction of matrix is,

$$\text{Red\_Col(M)} = M_{ji}\_\min \left\{ M_{ji} \mid 1 \le j \le n \right\} \quad \text{where } M_{ji} < \infty$$

4.11.3 Full Reduction

Let M be the cost matrix for travelling salesperson problem for n vertices then M is called reduced if each row and each column consists of either entirely ∞ entries or else contain at least one zero. The full reduction can be achieved by applying both row_reduction and column reduction.

For example : Consider, the matrix M as given below and reduce it fully.

$$M = \begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

Red_Row (M) can be obtained as,

$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix} \begin{matrix} \rightarrow 10 \\ \rightarrow 2 \\ \rightarrow 2 \\ \rightarrow 3 \\ \rightarrow 4 \end{matrix}$$

21

∴ **Cost (Red_Row (M)) = 21**

$$\text{Red\_Row (M)} = \begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 2 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix}$$

Red_Col(M) be obtained as,

$$\begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 2 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix}$$

$\downarrow \quad \downarrow$

$1 \quad 3 \qquad$ total = 4

∴ **Cost (Red_Col (M)) = 4**

If row or column contains at least one zero ignore corresponding row or column.

$$\text{Red\_Col(M)} = \begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

Thus total reduced cost will be

= cost (Red_Row (M)) + cost (Red_Col(M))

= 21 + 4

= 25

$$\text{Thus} \begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 1 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix} \xrightarrow[\text{reduced matrix}]{\text{fully}} \begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

4.11.4 Dynamic Reduction

We obtained the total reduced cost as 25. That means all tours in the original graph have a length at least 25.

Using dynamic reduction we can make the choice of edge i → j with optimum cost.

Steps in dynamic reduction technique

1. Draw a state space tree with optimum cost at root node.
2. Obtain the cost of matrix for path $i \rightarrow j$ by making i^{th} row and j^{th} column entries as ∞. Also set M[i] [j] = ∞.
3. Cost of corresponding node x with path i, j is optimum cost + reduced cost + M[i][j].
4. Set node with minimum cost as E-node and generate its children. Repeat step 1 to 4 for completing tour with optimum cost.

For example :

$$M = \begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

Fully reduced matrix is,

$$\begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

Optimum cost = 21 + 4 = 25.

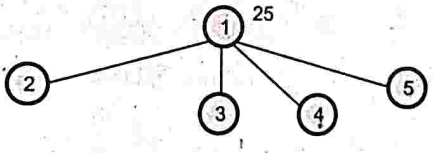
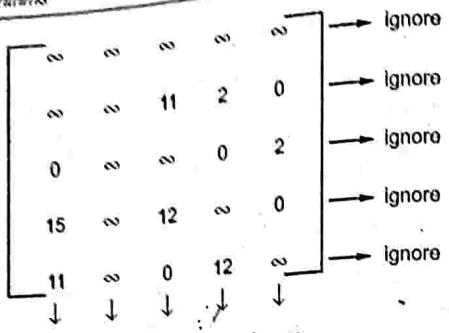


Fig. 4.11.2 Portion of state space tree

Consider path 1, 2. Make 1^{st} row and 2^{nd} column ∞. And set M [2] [1] = ∞.



Cost of node 2 is

$$25 + 0 + 10$$

25 — optimum cost; 0 — reduced cost; 10 — old value of M[1] [2]

$$= 35$$

Consider path 1, 3. Make 1st row = ∞. 3rd column = ∞ and M[3] [1] = ∞.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 15 & 3 & \infty & \infty & 0 \\ 11 & 0 & \infty & 12 & \infty \end{bmatrix}$$

$\downarrow$

11

Cost of node 3 is

$$= 25 + 11 + 17$$

25 — optimum cost; 11 — reduced cost; 17 — M[1][3]

$$= 53$$

Consider path 1, 4.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

Cost of node 4 is $= 25 + 0 + 0$

$= 25$

Consider path 1, 5.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & 2 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 15 & 3 & 12 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix} \begin{matrix} \\ \rightarrow 2 \\ \\ \rightarrow 3 \\ \\ \end{matrix}$$

Cost of node 5 is = 25 + 5 + 1

= 31

Now, as cost of node 4 is optimum, we will set node 4 as E-node and generate its children nodes 6, 7, 8.

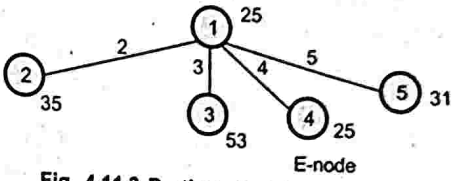


Fig. 4.11.3 Portion of state space tree

Consider path 1, 4, 2 for node 6. Set 1st row, 4th row to ∞. Set 2nd column to ∞. Also M[4] [1] = ∞, M[2] [1] = ∞.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

Cost of node 6 = 25 + M [4, 2]

= 25 + 3

= 28

Consider path 1, 4, 3 for node 7. Set 1st row, 4th row to ∞. Set 4th column, 3rd column to ∞. Set M [4] [1] = M[3] [1] = ∞.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ \infty & 3 & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & 0 & \infty & \infty & \infty \end{bmatrix} \begin{matrix} \\ \rightarrow 2 \\ \\ \\ \\ \end{matrix}$$

$$\uparrow$$

$$11$$

Cost of node 7 = 25 + 13 + M [4] [3]

= 25 + 13 + 12 = 50

Consider path 1, 4, 5 for node 8.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix} \rightarrow 11$$

Cost of node 8 = 25 + 11 = 36

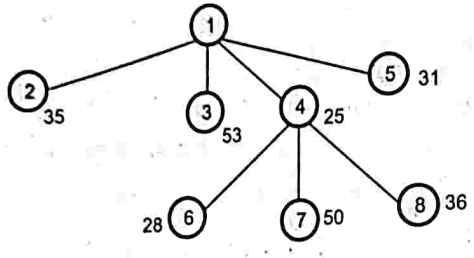


Fig. 4.11.4 Portion of state space tree

Now as cost of node 6 is minimum, node 6 becomes an E-node. Hence generate children for node 6. Node 9 and 10 are children nodes of node 6.

Consider path 1, 4, 2, 3 for node 9. Set 1st row, 4th row, 2nd row to ∞. Set 4th column, 2nd column and 3rd column to ∞.

Set M[4, 1] = M[2, 1] = M[3, 1] = ∞.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & \infty & \infty & \infty \end{bmatrix} \rightarrow 2$$

$$\downarrow$$

$$11$$

Cost of node 9

$$= \underset{\text{optimum cost}}{28} + \underset{\text{reduced cost}}{13} + \underset{M[2][3]}{11}$$

$$= 52$$

Consider path 1, 4, 2, 5 for node 10. Set 1st row, 4th row, 2nd row to ∞. Set 4th column, 2nd column and 5th column to ∞. Set M [4, 1] = M[2, 1] = M [5, 1] = ∞.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$$

Cost of node 10 = 28 + M[2] [5]

= 28 + 0

= 28

Node 10 becomes E-node now.

As for node 10 only child being generated is node 11. We set path as 1, 4, 2, 5, 3. To complete the tour we return to 1. Hence the state space tree is,

Hence the optimum cost of the tour is 28.

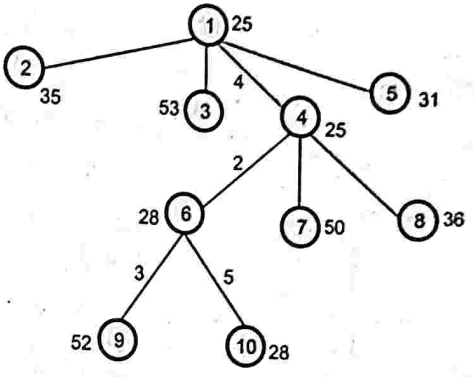


Fig. 4.11.5 Portion of state space tree

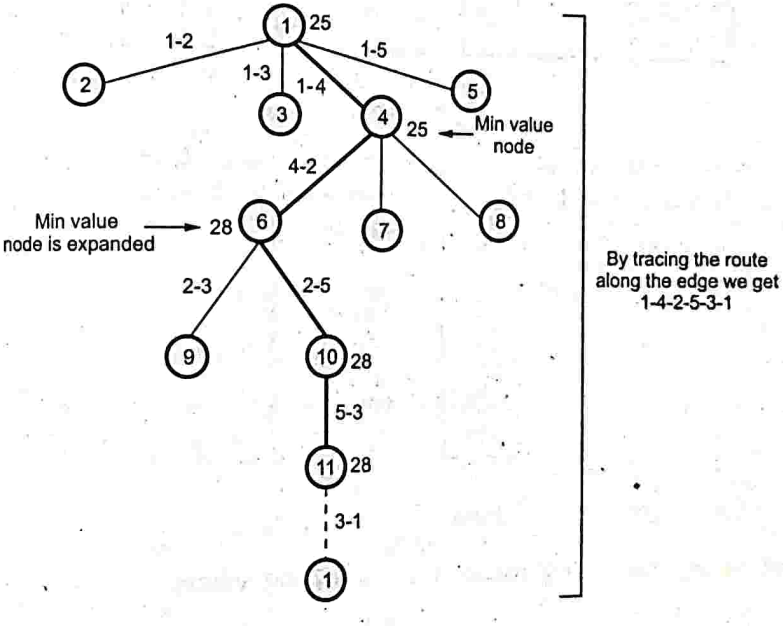


Fig. 4.11.6 State space tree

Example 4.11.1 *Apply the branch and bound algorithm to solve the TSP for the following cost matrix.*

$$\begin{bmatrix} \infty & 11 & 10 & 9 & 6 \\ 8 & \infty & 7 & 3 & 4 \\ 8 & 4 & \infty & 4 & 8 \\ 11 & 10 & 5 & \infty & 5 \\ 6 & 9 & 5 & 5 & \infty \end{bmatrix}$$

Solution :

Step 1 : We will find the minimum value from each row and subtract the value from corresponding row.

row						Minvalue					
	∞	11	10	9	6	→ 6		∞	5	4	3	0
	8	∞	7	3	4	→ 3		5	∞	4	0	1
	8	4	∞	4	8	→ 4	⟹	4	0	∞	0	4
	11	10	5	∞	5	→ 5		6	5	0	∞	0
	6	9	5	5	∞	→ 5		1	4	0	0	∞
						23		reduced Matrix				

Fig. 4.11.7

Now we will obtain minimum value from each column. If any column contains 0 then ignore that column and a fully reduced matrix can be obtained.

∞	5	4	3	0		∞	5	4	3	0
5	∞	4	0	1	Subtracting ⟹ 1 from 1st column	4	∞	4	0	1
4	0	∞	0	4		3	0	∞	0	4
6	5	0	∞	0		5	5	0	∞	0
1	4	0	0	∞		0	4	0	0	∞
↑	↑	↑	↑	↑						
1	ignore	ignore	ignore	ignore						

Total reduced cost = Total reduced row cost + Total reduced column cost

= 23 + 1

= 24

Now we will set 24 as the optimum cost.

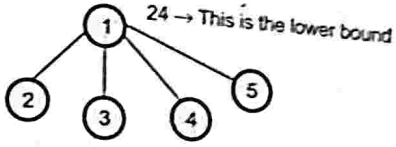


Fig. 4.11.8

Step 2 : Now we will consider the paths [1, 2], [1, 3], [1, 4] and [1, 5] of state space tree as given above consider path [1, 2] make 1st row, 2nd column to ∞. Set M [2] [1] = ∞.

∞	∞	∞	∞	∞	→ ignore
∞	∞	4	0	1	→ ignore
3	∞	∞	0	4	→ ignore
5	∞	0	∞	0	→ ignore
0	∞	0	0	∞	→ ignore

Now we will find min value from each corresponding column.

∞	∞	∞	∞	∞
∞	∞	4	0	1
3	∞	∞	0	4
5	∞	0	∞	0
0	∞	0	0	∞
↑	↑	↑	↑	↑
ignore	ignore	ignore	ignore	ignore

There is no minimum value from any column

Hence total reduced cost for node 2 is -

= Optimum cost + Old value of M [1] [2]

= 24 + 5

= 29

Consider path (1, 3). Make 1st row, 3rd column to be ∞.

Set M [3] [1] = ∞.

∞	∞	∞	∞	∞	→ ignore
4	∞	∞	0	1	→ ignore
∞	0	∞	0	4	→ ignore
5	5	∞	∞	0	→ ignore
0	4	∞	0	∞	→ ignore
	↑ ignore	↑ ignore	↑ ignore	↑ ignore	↑ ignore

There is no minimum value from any row and column

Hence total cost for node 3 is

= Optimum cost + M [1] [3]

= 24 + 4

= 28

Consider path (1, 4). Make 1st row, 4th column to be ∞. Set M [4] [1] = ∞.

∞	∞	∞	∞	∞	→ ignore
4	∞	4	∞	1	→ 1
3	0	∞	∞	4	→ ignore
∞	5	0	∞	0	→ ignore
0	4	0	∞	∞	→ ignore

Subtracting ⇒ 1 from 2nd row

∞	∞	∞	∞	∞
4	∞	3	∞	0
3	0	∞	∞	4
∞	5	0	∞	0
0	4	0	∞	∞
↑ ignore	↑ ignore	↑ ignore	↑ ignore	↑ ignore

The total cost for node 4 is

= Optimum cost + M [1] [4] + Minimum row cost

= 24 + 3 + 1

= 28

Consider the path (1, 5). Make 1st row, 5th column to be ∞. Set M [5] [1] = ∞.

∞	∞	∞	∞	∞	→ ignore
4	∞	4	0	∞	→ ignore
3	0	∞	0	∞	→ ignore
5	5	0	∞	∞	→ ignore
∞	4	0	0	∞	→ ignore
↑	↑	↑	↑	↑	
3	ignore	ignore	ignore	ignore	

Subtracting 3 from 1st column.

∞	∞	∞	∞	∞
1	∞	4	0	∞
0	0	∞	0	∞
2	5	0	∞	∞
∞	4	0	0	∞

The total cost at node 5 is

= Optimum cost + Reduced column cost + Old value M [1] [5]

= 24 + 3 + 0

= 27

The partial state space tree will be -

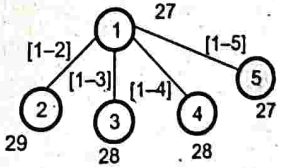


Fig. 4.11.9

The node 5 shows minimum cost. Hence node 5 will be an E node. That means we will select node 5 for expansion.

Step 3 : Now we will consider the paths [1, 5, 2], [1, 5, 3] and [1, 5, 4], of above state space tree do the further computations. Consider path 1, 5, 2. Make 1st row, 5th row and second column as ∞. Set M [5] [1] and M [2] [1] = ∞.

∞	∞	∞	∞	∞	→ ignore
∞	∞	4	0	1	→ ignore
3	∞	∞	0	4	→ ignore
5	∞	0	∞	0	→ ignore
∞	∞	∞	∞	∞	→ ignore
↑	↑	↑	↑	↑	
3	ignore	ignore	ignore	ignore	

Now subtracting 3 from 1st column, we get -

∞	∞	∞	∞	∞
∞	∞	4	0	1
0	∞	∞	0	4
2	∞	0	∞	0
∞	∞	∞	∞	∞

Hence total cost for node 6 will be -

= Optimum cost at node 5 + Column-reduced cost + M [5] [2]

= 27 + 3 + 4

= **34**

Consider path [1, 5, 3]. Make 1st row, 5th row and 3rd column as ∞.

Set M [5] [1] = M [3] [1] = ∞.

∞	∞	∞	∞	∞	→ ignore
4	∞	∞	0	∞	→ ignore
∞	0	∞	0	∞	→ ignore
5	5	∞	∞	∞	→ 5
∞	4	∞	0	∞	→ ignore

Subtracting ⇒ 5 from 4th row

∞	∞	∞	∞	∞
4	∞	∞	0	∞
∞	0	∞	0	∞
1	1	∞	∞	∞
∞	4	∞	0	∞
↑ 1	↑ Ignore	↑ Ignore	↑ Ignore	↑ Ignore

Subtract 1 from 1st column

∞	∞	∞	∞	∞
3	∞	∞	0	∞
∞	0	∞	0	∞
0	1	∞	∞	∞
∞	4	∞	0	∞

The total cost for node 7 will be -

= Optimum cost at node 5 + Column reduced cost + M [5] [3]

= 27 + 1 + 0

= 28.

Consider the path [1, 5, 4]. Make first row, fifth row and forth column to be ∞. Set M [5] [1] = M [4] [1] = ∞.

∞	∞	∞	∞	∞	→ ignore
4	∞	4	∞	1	→ 1
3	0	∞	∞	4	→ ignore
∞	5	0	∞	0	→ ignore
∞	∞	∞	∞	∞	→ ignore

Subtracting 1 from ⇒ 2nd row

∞	∞	∞	∞	∞
3	∞	3	∞	0
3	0	∞	∞	4
∞	5	0	∞	0
∞	∞	∞	∞	∞
↑ 3	↑ Ignore	↑ Ignore	↑ Ignore	↑ Ignore

Subtract 3 from ⇒ 1st column

∞	∞	∞	∞	∞
0	∞	3	∞	0
0	0	∞	∞	4
∞	5	0	∞	0
∞	∞	∞	∞	∞

The total cost for node 8 will be

= Optimum cost at node 5 + Column-reduced + Row-reduced cost + M [5] [4]

= 27 + (1 + 3) + 0

= 31

The partial state space tree will be -

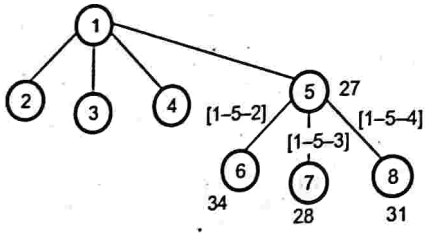


Fig. 4.11.10

The node 7 has optimum cost. Hence 7 becomes an E node and it will be expanded further.

Step 4 : Now we will consider paths [1, 5, 3, 2] and [1, 5, 3, 4] for further computations. Consider path 1, 5, 3, 2 and make 1^{st}, 5^{th}, 3^{rd} row as ∞ and 2^{nd} column as ∞. Set M [2] [1] = M [3] [1] = M[5][1] = ∞.

The matrix becomes -

∞	∞	∞	∞	∞	→ ignore
∞	∞	4	0	1	→ ignore
∞	∞	∞	∞	∞	→ ignore
5	∞	0	∞	0	→ ignore
∞	∞	∞	∞	∞	→ ignore
↑ 5	↑ ignore	↑ ignore	↑ ignore	↑ ignore	

Subtract 5 from 1st column.

∞	∞	∞	∞	∞
∞	∞	4	0	1
∞	∞	∞	∞	∞
0	∞	0	∞	0
∞	∞	∞	∞	∞

The reduced cost at node 9 [i.e. path [1, 5, 3, 2]] will be

= Optimum cost at node 7 + Column-reduced cost + M [3] [2]

= 28 + 5 + 0

= 33

Consider the path [1, 5, 3, 4]. Make 1st row, 5th row, 3rd row and 4th column as ∞.

Set M [5] [1] = M [3] [1] = M [4] [1] = ∞.

∞	∞	∞	∞	∞	→ ignore
4	∞	4	∞	1	→ 1
∞	∞	∞	∞	∞	→ ignore
∞	5	0	∞	0	→ ignore
∞	∞	∞	∞	∞	→ ignore

Subtracting 1 from second row,

∞	∞	∞	∞	∞
3	∞	3	∞	0
∞	∞	∞	∞	∞
∞	5	0	∞	0
∞	∞	∞	∞	∞

Subtracting 3 from 1st column and 5 from 2nd column

↑ ↑

3 5

∞	∞	∞	∞	∞
0	∞	3	∞	0
∞	∞	∞	∞	∞
∞	0	0	∞	0
∞	∞	∞	∞	∞

The total reduced cost at node 10 [i.e. path 1, 5, 3, 4] is

= Optimum cost at node 7 + Row-reduced cost + M [3] [4]

= 28 + (3 + 5) + 0

= 36

The partial state space tree will be -

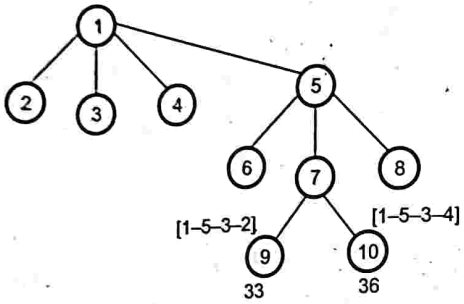


Fig. 4.11.11

Step 5 : Now consider path [1, 5, 3, 2, 4]

∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	5	0	∞	0
∞	∞	∞	∞	∞

↓

5 minvalue

Now subtract 5 from 2nd column

∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	∞	∞	∞	∞
∞	0	0	∞	0
∞	∞	∞	∞	∞

The reduced cost at node 11 will be -

= Optimum cost at node 9 + column reduced cost + M [2] [4]

= 33 + 5 + 0

= 38

The final state space tree will be -

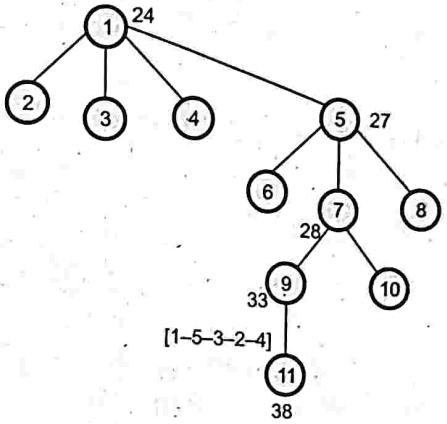
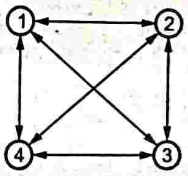


Fig. 4.11.12

The tour with minimum cost is 29. The path will be **1, 5, 3, 2, 4, 1.**

Example 4.11.2 *What is travelling salesperson problem ? Find the solution of the following travelling salesperson problem using dynamic approach and branch and bound approach.*

SPPU : May-12, Marks 16



$$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

Fig. 4.11.13

Solution :

Step 1 : We will find the minimum value from each row and subtract it from corresponding row.

∞	10	15	20	10
5	∞	9	10	5
6	13	∞	12	6
8	8	9	∞	8
				29

⇒

∞	0	5	10
0	∞	4	5
0	7	∞	6
0	0	1	∞

Reduced Matrix

Now we will obtain min value from each column. If some column contains 0 valu then ignore that column. The fully reduced matrix can be obtained as -

∞	0	5	10
0	∞	4	5
0	7	∞	6
0	0	1	∞
↑ ignore	↑ ignore	1	5

⇒

∞	0	4	5
0	∞	3	0
0	7	∞	1
0	0	0	∞

Total reduced cost = Total reduced row cost + Total reduced column cost

= 29 + 6 = 35

The 35 is now set as optimum cost

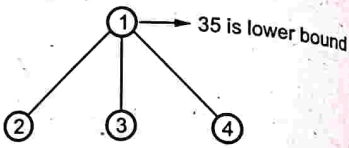


Fig. 4.11.14

Step 2 : Now consider path [1, 2]. Mark 1st row and 2nd column as ∞. Also set M[2][1] = ∞. Then we will find min from each row.

∞	∞	∞	∞	→ignore
∞	∞	3	0	→ignore
0	∞	∞	1	→ignore
0	∞	0	∞	→ignore

Now we will find min from each column

∞	∞	∞	∞
∞	∞	3	0
0	∞	∞	1
0	∞	0	∞
↑ Ignore	↑ Ignore	↑ Ignore	↑ Ignore

There is no min value from any column

Hence total reduced cost for node 2 is

= Optimum cost + Old value of M[1][2]

= 35 + 0 = 35

Now consider path (1, 3). Make 1st row and 3rd column to be ∞. set M[3][1] = ∞.

∞	∞	∞	∞	→ignore
0	∞	∞	0	→ignore
∞	7	∞	1	→1
0	0	∞	∞	→ignore

⇒

∞	∞	∞	∞
0	∞	∞	0
∞	6	∞	0
0	0	∞	∞

↑ ignore ↑ ignore

Total cost for node 3 is

= Optimum cost + Old value of M[1][3] + Reduced cost

= 35 + 4 + 1 = 40

Now consider path (1, 4). Marks 1st row and 4th column to be ∞. Set M[4][1] = ∞.

Optimum cost for node 3 is =

Optimum cost + Old value of M[1][4]

= 35 + 5 = 40

∞	∞	∞	∞	→ignore
0	∞	3	∞	→ignore
0	7	∞	∞	→ignore
∞	0	0	∞	→ignore

↑ ignore ↑ ignore ↑ ignore ↑ ignore

The partial tree will be as shown in Fig. 4.11.15.

Node 2 with minimum cost. Hence it will be expanded further.

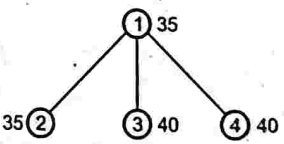


Fig. 4.11.15

Step 3 : Consider paths [1, 2, 3] , [1, 2, 4].

Consider path 1, 2, 3. Make 1st row, 2nd row and 3rd column as ∞. Mark M[2][1] and M[3][1] = ∞.

∞	∞	∞	∞	→X
∞	∞	∞	∞	→X
∞	7	∞	1	→1
0	0	∞	∞	→X

⇒

∞	∞	∞	∞
∞	∞	∞	∞
∞	6	∞	0
0	0	∞	∞

Hence total cost for node 5 will be

= Optimum cost at node 2 + Reduced cost + M[2][3]

= 35 + 1 + 3 = 39

Consider path [1, 2, 4]. Make 1st row, 2nd row and 4th column as ∞.

M[2][1] = M[4][1] = ∞

∞	∞	∞	∞	→X
∞	∞	∞	∞	→X
∞	7	∞	0	→X
0	0	∞	∞	
↑ X	↑ X	↑ X	↑ X	

No minimum value

∴ Optimum cost at node 6 will be

= Optimum cost at node 2 + Reduced cost + M[2][4]

= 35 + 0 + 0 = 35

The partial tree will be in Fig. 4.11.16.

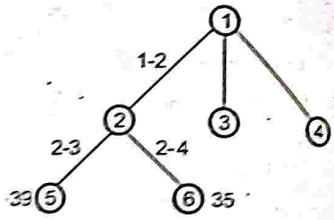


Fig. 4.11.16

Step 4 : Consider path [1, 2, 4, 3].

Mark 1st row = 2nd row = 4th row = 3rd column = ∞

M[3][1] = M[4][1] = M[2][1] = ∞

∞	∞	∞	∞	→X
∞	∞	∞	∞	→X
∞	7	∞	1	→1
∞	∞	∞	∞	→X

⇒

∞	∞	∞	∞
∞	∞	∞	∞
∞	6	∞	0
∞	∞	∞	∞

Subtract 6 from 2nd column.

⇒

∞	∞	∞	∞
∞	∞	∞	∞
∞	0	∞	0
∞	∞	∞	∞

Optimum cost = Optimum cost at node 6 + M[4][3]

= 35 + 0 = 35

Final state space tree will be in Fig. 4.11.17.

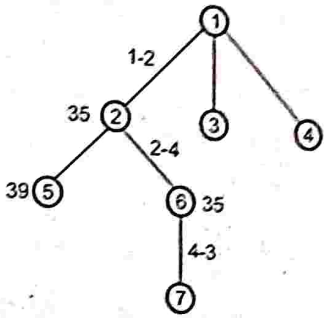


Fig. 4.11.17

The path will be 1 - 2 - 4 - 3 - 1.

The tour with minimum cost is 35.

Example 4.11.3 *Explain branch and bound strategy. Take an example of travelling salesman problem using branch and bound.*

SPPU : Dec.-13, Marks 18, Dec.-15 (End Sem.), Marks 8

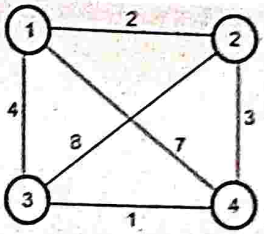


Fig. 4.11.18

Solution : **Branch and bound strategy -**

Refer section 4.11.1.

We will create adjacency matrix for given graph.

	1	2	3	4
1	∞	2	4	7
2	2	∞	8	3
3	4	8	∞	1
4	7	3	1	∞

Step 1 : We will find the minimum value from each row and subtract it from corresponding row.

∞	2	4	7	2
2	∞	8	3	2
4	8	∞	1	1
7	3	1	∞	1

Total reduced row cost 6

⇒

∞	0	2	5
0	∞	6	1
3	7	∞	0
6	2	0	∞

Now we will obtain minimum value from each column. If some column contains 0 value the ignore that column. The fully reduced matrix can be -

∞	0	2	5
0	∞	6	1
3	7	∞	0
6	2	0	∞

← This is reduced matrix

ignore ignore ignore ignore

Matrix M [Used in further steps]

∴ Total reduced cost = Total reduced row cost + Total reduced column cost

= 6 + 0 = 6

∴ The 6 is now set as optimum cost.

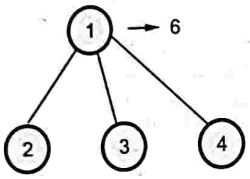


Fig. 4.11.19

Step 2 : Consider path [1,2]. Mark first row and 2nd column as ∞. Also set M[2,1] = ∞. We will then find minimum from each row.

The reduced matrix will be

∞	∞	∞	∞	→ ignore
∞	∞	6	1	→ 1
3	∞	∞	0	→ ignore
6	∞	0	∞	→ ignore

∞	∞	∞	∞
∞	∞	5	0
3	∞	∞	0
6	∞	0	∞

3 ignore

The reduced matrix will be

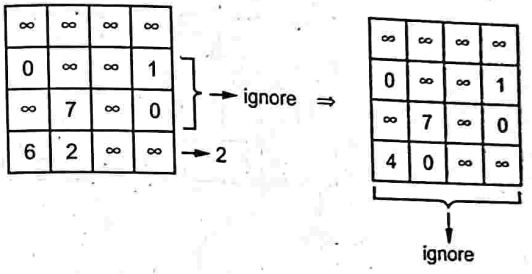
∞	∞	∞	∞
∞	∞	5	0
0	∞	∞	0
3	∞	0	∞

∴ Reduced cost for node 2 = Optimum cost + Old value M [1, 2] + Reduced cost

= 6 + 0 + (1 + 3)

= 10

Step 3 : Consider path (1, 3). Mark first row and 3rd column as ∞. Also set M [3, 1] = ∞

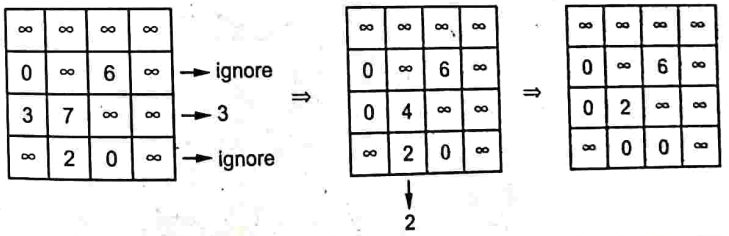


Cost for node 3 = Optimum cost + Old value M [1, 3] + Reduced cost

= 6 + 2 + (2 + 0)

= 10

Step 4 : Consider path [1, 4]. Mark first row and 4th column as ∞. Set M [4, 1] = ∞



Reduced cost at node 4

= Optimum cost + Old value M [1, 4] + Reduced cost = 6 + 5 + (3 + 2) = 16

The partial tree will be,

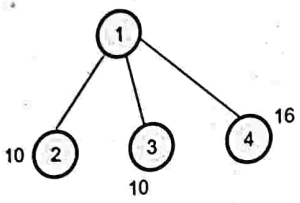


Fig. 4.11.20

We can choose either node 2 or node 3. Let us select node 3. We will expand further.

Step 5 : Consider path $[1,3,4]$ and $[1,3,2]$ First consider path $[1,3,2]$. Mark 1st, 3rd ro as ∞ and 2nd column as ∞. Also $M[2,1]=M[3,1]=\infty$.

∞	∞	∞	∞	
∞	∞	6	1	1
∞	∞	∞	∞	
6	∞	0	∞	

$\Rightarrow$

∞	∞	∞	∞
∞	∞	5	0
∞	∞	∞	∞
6	∞	0	∞

6

$\Rightarrow$

∞	∞	∞	∞
∞	∞	5	0
∞	∞	∞	∞
0	∞	0	∞

Reduced cost at node 5 will be optimum cost at node 3 + $M[3,2]$ + Reduced cost

$$= 10+7+(1+6)$$

$$= 24$$

∞	∞	∞	∞
0	∞	6	∞
∞	∞	∞	∞
∞	2	0	∞

Consider path $[1,3,4]$. Mark row 1 = row 3 = ∞ and column 4 = ∞. $M[4,1]=M[3,1]=\infty$

From Both row and column. we cannot obtain any minimum value. Hence reduced cost = 0.

$\therefore$ Reduced cost at node 6 will be optimum cost at node $3+M[3,4]+\text{Reduced cost} = 10+1+0 = 11$

Partial tree will be

Naturally node 6 will be expanded.

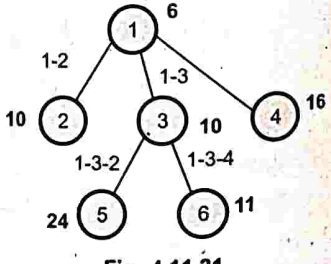


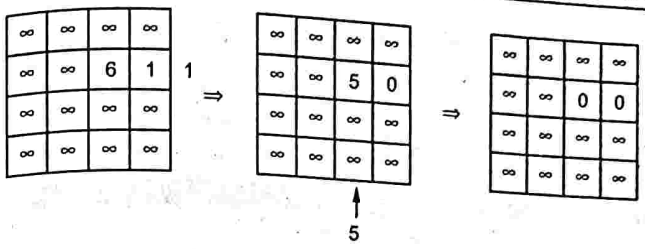
Fig. 4.11.21

Step 6 : Consider path $[1,3,4,2]$

Mark 1st row = 3rd row = 4th row = ∞

Mark 2nd column = ∞

$M[2,1]=M[4,1]=M[3,1]=\infty$



∴ Reduced cost at node 7

= Optinum cost at node $6+M[4,2]+$Reduced cost

= $11+2+(1+5)$

= 19

The tree will be

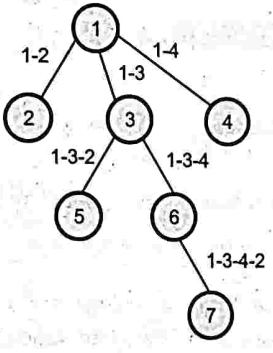


Fig. 4.11.22

Thus path with optinum tour length is **1-3-4-2-1.**

The cost of tour = 10.

Review Questions

1. *Explain how branch and bound method can be used to solve travelling salesperson problem.* **SPPU : May-08, Dec.-08, Marks 6**
2. *Explain the branch and bound algorithmic strategy for solving the problem take an example of traveling salesman problem using branch and bound.* **SPPU : Dec.-11, 12, Marks 10**
3. *Explain dynamic reduction with all steps with respect to travelling salesperson problem.* **SPPU : May-14, Marks 6**

4. *Explain the steps of solving travelling sales man problem using branch and bound.* **SPPU : Dec.-14, Marks 8**

5. *Explain how travelling salesperson problem is solved using branch and bound method with suitable example.* **SPPU : Aug.-15 (In Sem.), Marks 4**

4.12 Knapsack Problem

SPPU : Dec.-10, May-08, 09, 13, Oct.-16, Marks 10

Problem statement :

The 0/1 Knapsack problem states that - There are 'n' objects given and capacity of Knapsack is 'm'. Then select some objects to fill The knapsack in such a way that it should not exceed the capacity of Knapsack and maximum profit can be earned. The Knapsack problem is a maximization problem. That means we will always seek for maximum $P_i x_i$(where P_i represents profit of object x_i). We can also get $\sum P_i x_i$ maximum iff $- \sum P_i x_i$ is minimum.

$$\text{Minimize profit} - \sum_{i=1}^{n} P_i x_i$$

$$\text{subject to} \quad \sum_{i=1}^{n} W_i x_i$$

$$\text{Such that} \sum W_i x_i \leq m \text{ and}$$

$$x_i = 0 \text{ or } 1 \text{ where } 1 \leq i \leq n$$

We will discuss the branch and bound strategy for 0/1 Knapsack problem using fixed tuple size formulation. We will design the state space tree and compute c^(.) and u(.) where c^(x) represents the approximate cost used for computing the **least cost** c(x). Clearly u(x) denotes the **upper bound**. As we know upper bound is used to kill those nodes in the state space tree which can not lead to the answer node.

Let, x be the node at level j. Then we will draw the state space tree for fixed tuple formulation having levels $1 \leq j \leq n+1$.

Then we need to compute c^(x) and u(x). Such that $c^(x) \leq c(x) \leq u(x)$ for every node.

Algorithm

The algorithm for computing c^(x) is as given below.

```
Algorithm C_Bound(total_profit,total_wt,k)
//total_profit denotes the current total profit
//total_wt denotes the current total weight
// k is the index of last removed object
```

```
//w[i] represents the weight of object i
//p[i] represents the profit of object i
//m is the weight capacity of knapsack
{
    pt←total_profit;
    wt←total_wt;
    for(i←k+1 to n) do
    {
        wt←wt+w[i];
        if(wt<m) then pt←pt+p[i];
        else return (pt+(1 – (wt-m)/w[i])*p[i]);
    }
    return pt;
}
```

The algorithm for computing u(x) is as given below

```
Algorithm U_Bound(total_profit,total_wt,k,m)
```

SPPU : May-08, 09, 13, Dec.-10, Marks 6

```
//total_profit denotes the current total profit
//total_wt denotes the current total weight
// k is the index of last removed object
//w[i] represents the weight of object i
//p[i] represents the profit of object i
//m is the weight capacity of knapsack
{
    pt←total_profit;
    wt←total_wt;
    for(i←k+1 to n) do
    {
        if(wt+w[i]<=m) then
        {
            pt←pt – p[i];
            wt←wt+w[i];
        }
    }
    return pt;
}
```

Example 4.12.1 *Find an optimal solution for the following 0/1 Knapsack instance using branch and bound method : Number of objects n = 5, capacity of knapsack m = 100 Profits = (10, 20, 30, 40, 50), weights = (20, 30, 66, 40, 60)* **SPPU : Oct.-16, Marks 10**

Solution : We will arrange all the items in such a way that $\frac{P_i}{W_i} > \frac{P_{i+1}}{W_{i+1}} > \frac{P_{i+2}}{W_{i+2}} \cdots$

The arrangement will be

Item	P_i	W_i
4	40	40
5	50	60
2	20	30
1	10	20
3	30	66

- Here we will select item 4, item 5. Now if we select item 2 then total weight > m. Hence by selecting item 4 and item 5 the upper bound for profit is obtained i.e. 40 + 50 = 90 ∴ we will write ub = – 90 for root node, of BB tree. As by choosing item 4 and item 5 the total capacity of knapsack is fulfilled. Hence

  $c^{\wedge} = ub = -90$

- Now if we don't select item 4. Then consider selection of item 5 and item 2. Then upper bound for profit is ub = – 70. By this selection the total weight = 60 + 30 = 90. Now we need only weight 10 to reach to maximum capacity of knapsack.

  ∴ Just consider next item for selection i.e. item 1 but we need its fractional weight

∴ $c^{\wedge} = ub + \text{Fractional profit of next item}$

$$= (70) + \left(\frac{10}{20} * 10\right)$$

$$= 75$$

∴ $c^{\wedge} = -75$

- The partial state space tree will be -

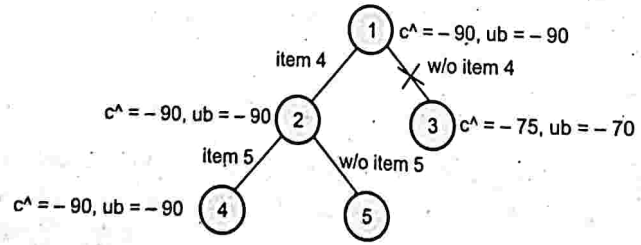


Fig. 4.12.1

We will choose node 2 because it is having less upper bound than node 3. The node 4 and node 5 has the same C^ and ub as that of node 2 because it considers selection of item 4 and item 5.

Now consider node 5 i.e. select item 4 and item 2 ub = – (40 + 20) = – 60. The next remaining weight of item 1 is 20 which can be added as a whole.

$$C^\wedge = -(\text{ub} + \text{weight of item 1})$$

$$\therefore \quad = -(60 + 10)$$

$$C^\wedge = -70$$

$$\therefore$$

At node 5 $C^\wedge = -70$, ub = – 60

$$\therefore$$

But as node 4 is having lesser upper bound than node 5. Select node 4.

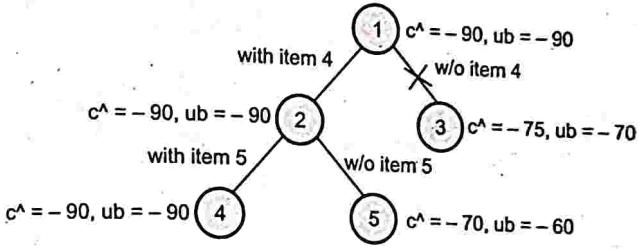


Fig. 4.12.2

Now can not further select item 2, item 1 or item 3 because by selecting item 4 and item 5 the knapsack has reached to its capacity.

Hence solution = **{item 4, item 5}** with maximum **profit = 90.**

4.13 Difference between Backtracking and Branch and Bound Techniques

Sr. No.	Backtracking	Branch and Bound
1.	The solution for backtracking is traced using depth first search.	In this technique it is not necessary to use depth first search for obtaining the solution, we can use breadth first search algorithm.
2.	Typically the decision problems are solved using backtracking.	Typically the optimization problems are solved using branch and bound.

3.	While finding the solution, the bad choices can be made.	It proceeds on better solutions. So there cannot be a bad solution.
4.	The state space tree is searched until the solution is obtained.	The state space tree needs to be searched completely as there may be the chances of being an optimum solution any where in state space tree.
5.	Applications - M-Coloring, eight queens problem.	Applications - Job sequencing, Traveling Salesperson Problem.

Unit V

5 Amortized Analysis

Syllabus

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.

Contents

5.1 Amortized Analysis

SPPU : Dec.-07, May-08, 18, 19, Marks 10

An amortized analysis means finding average running time per operation over a worst case sequence of operations. An amortized analysis indicates that average cost of a single operation is small if average of sequence of operations is obtained. This is true even if any one operation is expensive within the sequence. An amortized analysis guarantees the time per operation over worst case performance.

There is a difference between average case analysis and amortized analysis. In average case analysis we are averaging over all possible inputs and in amortized analysis we are averaging over a sequence of operations. An amortized analysis assumes worst-case input.

There are 3 commonly used techniques used in amortized analysis -

i) Aggregate analysis ii) Accounting method iii) Potential method

Key Point *The amortized cost of n operations is the total cost of the operations divided by n.*

Let us discuss each technique of amortized analysis one by one.

5.1.1 Aggregate Analysis

Aggregate analysis is a kind of analysis in which the analysis is made over sequence of n operations and these operations actually take worst case time T(n). In worst case an amortized cost per operation is T(n)/n. Let us now discuss two examples for making aggregate analysis in order to obtain amortized cost.

Example : Implementing stack as an array.

There are two basic operations in stack.

1. **To implement 'push (item)' we need :**

```
s [top] = item;
top ++;
```

2. **To implement item = pop () we need :**

```
top – – ;
item = s[top];
```

Thus **push (item)** and **pop ( )** are the operations each running in O(1) time. This also means that cost of each operation is O(1) and for n sequences of operations the total execution time will be **Θ (n)**. Now if we write some function for performing **multiple pop** operations then **k** top objects will be removed from the stack.

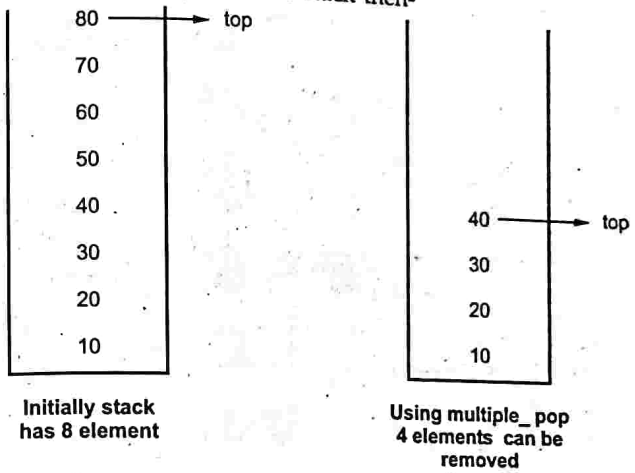
Algorithm multiple _ pop (st, k)

```
{
// Problem Description : This algorithm is for
// removing k top objects repeatedly using stack.
while ((! stempty (st)) AND kl = 0)
{
   pop (st)
   k← k – 1
} // end if while
} // end of algorithm
```

While stack do not gets empty the desired elements can be removed.

For example :

If we want to remove 4 elements from the stack then-

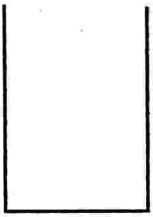


Initially stack has 8 element

Using multiple_ pop 4 elements can be removed

That means for the size 8 of the stack we can perform at the most 8 pop operations.

- From above given sequence of execution we can state that worst case cost for **multiple_pop** is O(n) when the size of stack is at the most n. Hence a sequence of n operations costs $O(n^2)$ because we can have O(n) multiple_pop operations costing O(n) each. The cost $O(n^2)$ is correct but not tight.



Empty stack after removing 4 elements

- We can broadly state that each object can be popped off once. That also means : Total number of pop = Total number of push.
- The aggregate analysis suggests to define amortized cost to be average cost.
- Hence average cost in amortized analysis is nothing but average cost of an operation = O(n)n

Example : Binary counter

This problem is for implementing binary counter for a k-bit number. We will use an array B in which a binary number is stored. Bits ranging from 0 to k – 1.

For example :

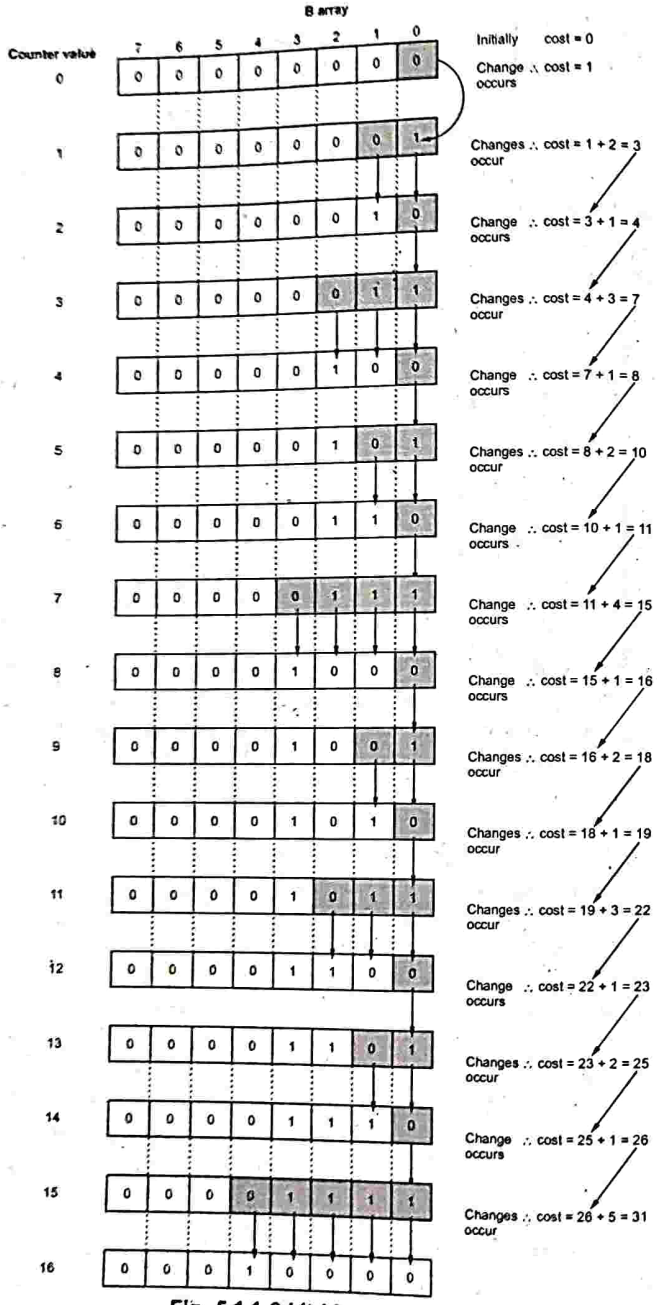


Fig. 5.1.1 8-bit binary counter

From above given counter method we can make out few observations -

B [0] flips/changes every time, total n times

B [1] flips/changes every other time [n/2] times

B [2] flips/changes for [n/4] times

B [3] flips/changes for [n/8] times

. . .

. . .

for i = 0,1... k–1, B [i] changes for $[n/2^i]$ times.

Thus total number of flips is

$$\sum_{i=0}^{k-1}\left[n / 2^i\right]$$

$$< n\sum_{i=0}^{\infty} 1 / 2^i$$

$$= 2n$$

The worst case time will be **O(n)**. The average cost of each operation i.e. **amortized** cost per operation = **O(n)/n = O(1)**.

The algorithm which is used to obtain binary counter values is as given below.

Algorithm

Algorithm Increment (B[0... k–1])

{

// **Problem Description** : This algorithm is

// for obtaining binary counter by flipping

// the corresponding bits.

i ← 0

while ((i < length (B[ ]) **AND** (B[i] = 1))

{

B[i] ← 0

i ← i + 1

} // end of while

if (i < length (B [ ]) **then**

B[i] ← 1

} // end of algorithm

5.1.2 Accounting Method

The counting method is based on the charges that are assigned to each operation. The idea of accounting method is as follows -

- Assign different charges to different operations.
- The amount of charge is called **amortized cost.**
- Then there is **actual cost** of each operation. The amortized cost can be more or less than actual cost.
- When amortized cost > actual cost, the difference is saved in specific objects called **credits.**
- When for particular operation amortized cost < actual cost the stored **credits** are **utilized.**
- Thus in accounting method we can say

Amortized cost = actual cost + credits (either deposited or used up)

- In aggregate analysis method, it is **not** necessary to have amortized cost to each **operation** but in accounting method each operation must have some amortized cost. Following are the conditions used in accounting method -
- Let c_i be the actual cost of i^{th} operation in the sequence, and c_i' be the amortized cost of i^{th} operation in the sequence.
- There should be -

$$\sum_{i=1}^{n} c_i' \geq \sum_{i=1}^{n} c_i$$

That means we need the total amortized cost is an upper bound of total actual cost. This holds true for all sequences of operations.

- **Total credit** $= \sum_{i=1}^{n} c_i' - \sum_{i=1}^{n} c_i$

Which should be **non negative.** Let us understand the method of obtaining amortized cost using accounting method with the help of two examples :

1) Stack operations and 2) Binary counter

Example 1 : Stack operations.

Let us first assign some charges to each stack operation.

Push (item) : = 1

Pop ( ) : 1

multiple_pop: min (st, k) where st is stack size and k is number of multiple pops. These cost are the **actual costs.**

Let us now assign amortized costs as -

push (item) : 2

pop ( ) : 0

multiple_pop : 0

Note that **multiple_pop** operation has variable actual cost and 0 amortized cost. Now consider some sequence of operations starting from empty stack. For now, if we push some object onto the stack then associated with it $2 as amortized cost. We then can use $1 to pay the actual cost of push and left with $1 as credit. If we go on pushing the objects the credits will get accumulated each time. Now, while popping the object each credit will be used to prepay the cost of pop operation. Thus by charging more on push operation we need **not** have to **charge** anything on **pop operation**. The same is true for multiple_pop operation. This also ensures that credits are always **non negative.**

Hence total amortized cost for n push (item), pop, multiple_pop is **O(n)**. Hence average amortized cost for each operation is O(n) / n = 1

Example 2 : Binary counter

The accounting method is applied on binary counter as follows -

- Let $1 is the charge assigned for each unit cost.
- Let, the amortized cost of $2 is charged for setting a bit to 1.
- When the bit is set, then out of $2 we can use $1 to pay the actual cost and store $1 on the bit as credit. Thus on any point of time the set bit will have $1 as credit.
- When a bit is reset (i.e. changing bit to 0) the credit of $1 is used to pay the cost.
- Thus the amount of credit is always **non negative.**
- The total amortized cost of n operations is **O(n).**

5.1.3 Potential Method

This method is similar to the accounting method in which the concept of "prepaid" is used.

- In this method there will not be any credit but there will be some potential "energy" or "potential" which can be used to pay for future operations.
- Instead of associating potential with specific object it is associated with the data structure as a whole. The **working** of potential method is as follows -
- Let, D_0 be the initial data structure. Hence for n operations D_0, D_1, D_2.....D_n will be the data structure. Let c_1, c_2, ... c_n denotes the actual cost.
- Let Φ is a potential function which is a **real** number. $\Phi(D_i)$ is called potential of D_i.

- Let, c_i' be the amortized cost of i^{th} operation

$$c_i' = c_i + \underbrace{\Phi(D_i) - \Phi(D_{i-1})}_{\text{Potential change}}$$

Actual cost

$$\sum_{i=1}^{n} c_i' = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \quad \text{and}$$

$$\sum_{i=1}^{n} c_i' = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

- If $\Phi(D_n) \geq \Phi(D_0)$, then amortized cost is an **upper bound** of actual cost.
- For any sequence of operation we do not know the exact number of operations Hence it is required to have $\Phi(D_i) \geq \Phi(D_0)$ for any value of i.
- Let $\Phi(D_0) = 0$. So $\Phi(D_i) \geq 0$ for all i.
- If **potential change is positive** i.e. if $\Phi(D_i) - \Phi(D_{i-1}) > 0$ then c_i' is overcharge an potential of data structure increases.
- If potential change is negative i.e. if $\Phi(D_i) - \Phi(D_{i-1}) < 0$ then c_i' is an undercharge i.e discharge the potential to pay actual cost.

Let us now obtain amortized analysis by potential method for

1) Stack operations and 2) Binary counter.

Example 1 : Stack operation

The number of objects in the stack is the potential for the stack. So

$\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$ i.e. non negative value.

We can now obtain amortized cost of each stack operation as follows -

Push operation

Potential change $= \Phi(D_i) - \Phi(D_{i-1})$

$= (s + 1) - s$

$= 1$

s + 1 means inserting one item to existing stack of size s

Amortized cost = actual cost + potential change

$= c_i + \Phi(D_i) - \Phi(D_{i-1})$

$= 1 + 1$

$c_i' = 2$

Pop operation

Potential change $= \Phi(D_i) - \Phi(D_{i-1})$

$= (s - 1) - s$

$= -1$

s – 1 means popping off one item from stack of size s

Amortized cost = actual cost + potential change

$= c_i + \Phi(D_i) - \Phi(D_{i-1})$

$= 1 + (-1)$

$c_i' = 0$

Multiple_pop

Potential change $= \Phi(D_i) - \Phi(D_{i-1})$

$= k'$

where $k' = \min(st, k)$ with k denotes number of pop and st denotes stack size.

Amortized cost = Actual cost + Potential change

$= c_i + \Phi(D_i) - \Phi(D_{i-1}) = k' + k'$

$c_i' = 0$

Thus **amortized cost** of each operation is O(1) and total amortized cost of n operations is **O(n).**

The **worst case** cost of n operations is **O(n).**

Example 2 : Binary counter

The potential of the counter after i^{th} increments is $\Phi(D_i) = b_i$ i.e. the number of 1's. Let us now compute amortized cost of an operation -

Let, there be i^{th} operation which resets t_i bits.

∴ Actual cost c_i of the operation is $t_i + 1$

∴ $c_i = t_i + 1$

- If $b_i = 0$ then the i^{th} operation resets all k bits. Hence

$$b_{i-1} = t_i = k$$

- If $b_i > 0$ then $b_i = b_{i-1} - t_i + 1$

From these two conditions

$b_i \leq b_{i-1} - t_i + 1$

Hence potential change is

$\Phi(D_i) - \Phi(D_{i-1}) \leq b_{i-1} - t_i + 1 - b_i$

Potential change = $1 - t_i$

$\therefore$ Amortized cost c_i' = Actual cost + Potential change

$= c_i + \Phi(D_i) - \Phi(D_{i-1}) \leq t_i + 1 - 1 - 1 - t_i$

$= 2$

The total amortized cost of n operations is O(n)

The worst case cost is O(n).

Comparison between methods of Amortized analysis

- **Aggregate method :** In this method the entire sequence is analysed first and t amortized cost per operation is calculated.
- **Accounting method :** In this method, we assign differing charges to differ operations, with some operations charged more or less than they actually cost. T amount we charge an operation is called its amortized cost. When an operatio amortized cost exceeds its actual cost, the difference is assigned to specific objects the data structure as credit.
- **Potential method :** In this method, we define a function a prove that it is a potent function.

Review Questions

1. *Explain in brief amortized analysis. Find the amortized cost with respect to stack operations.*
   SPPU : Dec.-07, Marks 10
2. *Name and explain in two or three sentences three popular methods to arrive at amortized costs fo the operations.*
   SPPU : May-08, Marks 6
3. *Explain amortized analysis. Find the amortized cost with respect to stack operations.*
   SPPU : May 18, Marks 8
4. *What is amortized analysis ? Explain aggregate and accounting techniques with example.*
   SPPU : May 19, Marks 8

5.2 Time-Space Tradeoff

- **Definition :** Time space tradeoff is basically a situation where either a spac efficiency (memory utilization) can be achieved at the cost of time or a time efficienc (performance efficiency) can be achieved at the cost of memory.
- **Example 1 :** Consider the programs like compilers in which symbol table is used t handle the variables and constants. Now if entire symbol table is stored in th program then the time required for searching or storing the variable in the symbo table will be reduced but memory requirement will be more. On the other hand, i we do not store the symbol table in the program and simply compute the tabl entries then memory will be reduced but the processing time will be more.

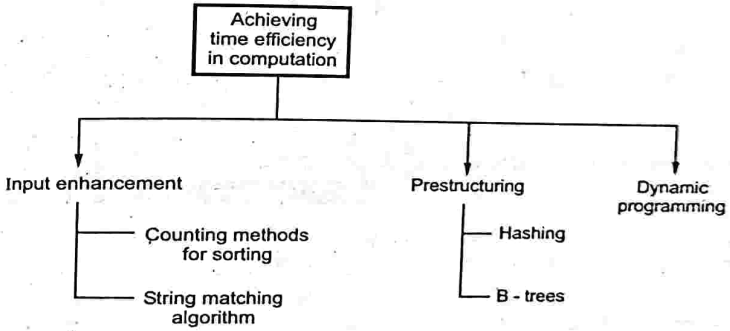
- **Example 2 :** Suppose, in a file, if we store the uncompressed data then reading the data will be an efficient job but if the compressed data is stored then to read such data more time will be required.
- **Example 3 :** This is an example of reversing the order of the elements. That is, the elements are stored in an ascending order and we want them in the descending order. This can be done in two ways -

i) We will use another array b[] in which the elements in descending order can be arranged by reading the array a[] in reverse direction. This approach will actually increase the memory but time will be reduced.

ii) We will apply some extra logic for the same array a[] to arrange the elements in descending order.This approach will actually reduce the memory but time of execution will get increased.

How to achieve time efficiency ?

If we want to get the results of computation quickly at the cost of space, then in such a case input enhancement, prestructuring and dynamic programming approaches are used.



The **input enhancement** is based on the preprocessing the instance to obtain additional information that can be used to solve the instance in less time. Examples are: counting methods for sorting, string matching algorithm.

The **prestructuring** is based on the idea of using extra space to get fast execution or Flexible access to data.

Examples of prestructuring approach are : Hashing and indexing with B-tress.

Dynamic programming is based on recording of solutions to smaller instances so that each smaller instance is **solved once.** Typically in this method **table** is built and stored values of table are reused to obtain solution of next instance.

Issues in Space-Time tradeoffs

1. When time and space compete with each other an **efficient algorithm** with **space-efficient data structure** has to be used.

   For example in sparse Matrix (the matrix which contains very less number of values, and almost all the matrix contains zero).

2. **Data compression** techniques should be applied in order to reduce the size of data used in computation.

5.3 Introduction to Tractable and Non-tractable Problems

- **Tractable problem :** A problem that is solvable by a polynomial-time algorithm. The upper bound is polynomial.
- **Examples of tractable problems are -**
  i) Searching an unordered list ii) Searching an ordered list
  iii) Sorting a list iv) Multiplication of integers
  v) Finding a minimum spanning tree in a graph.
- **Intractable problem :** A problem that cannot be solved by a polynomial-time algorithm. The lower bound is exponential.
- **Examples of intractable problems -**
  i) Traveling Salesman Problem
  ii) Knapsack Problem

5.4 Introduction to Randomized Algorithms

SPPU : Dec.-15, 18, 19, May-16, 19, Marks 8

In probability theory various "experiments" are carried out. The outcomes or results of those experiments determine the specific characteristics.

For example : Picking up a card from a deck of 52 cards, tossing coin for five times, choosing a red ball from an urn containing red and white balls, rolling die four times. Each possible result of such experiment is called sample point. The set of all sample points is called **sample space.** The sample space is denoted by S. The sample space S is finite set. An event E occurs from sample space. For **m** sample points there are 2^m possible events.

Probability : The probability of event E is the ratio of E to S. Hence

$$\text{Probability} = \frac{|E|}{|S|}$$

For example : When a coin is tossed, then we may get either head (H) or tail (T). Suppose, we have tossed four coins together then there are 16 possible outcomes : HHHH, HHHT, HHTH, HHTT, HTHH, HTHT, HTTH, HTTT, THHH, THHT, THTH,

THTT, TTHH, TTHT, TTTH, TTTT. For 4 events {HHTH, THTT, TTHH, TTTT}, the probability is $\frac{4}{16} = \frac{1}{4}$.

Mutual exclusion : Two events A and B are said to be mutually exclusive if they do not have any common sample point. Hence $A \cap B = \phi$. For example A = {HHTH, HHTT}, B = {HTTT, THHT} are mutually exclusive.

Independence : Two events A and B are said to be independent if

$$P[A \cap B] = P[A] * P[B]$$

Random variable : The random variable is basically a function that maps elements of S to set of real numbers.

For sample point $a \in S$ the F (a) denotes the mapping. If F denotes a finite set of elements then it is called **discrete.** Thus the random variables can be discrete random variables.

For example - If we pick up four balls from an urn containing red and white balls then the number of red balls that get selected is F (RRRW) = 3 or F (RWWW) = 1 and so on.

Probability distribution : If F is discrete random variable for sample space S then probability distribution can be defined for a range of elements $\{a_1, a_2, \ldots a_n\}$ such that $P[F = a_1]$, $P[F = a_2]$ ..., $P[F = a_n]$. For a probability distribution $\sum_{i=1}^{n} P[F = a_i] = 1$.

For example if we pick up four balls randomly from an urn containing red and white balls and F is number of red balls, then F can take on five values 0, 1, 2, 3 and 4 then

0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

⇐ Here 1 means presence of red balls and 0 means absence of red ball from the four balls that are picked up

Hence probability distribution of F is given by

$$P[F = 0] = \frac{1}{16}$$ i.e. no red ball present.

$$P[F = 1] = \frac{4}{16}$$ i.e. only one red ball present.

$$P[F = 2] = \frac{6}{16}$$ i.e. two red balls present from picked up balls.

$$P[F = 3] = \frac{4}{16}$$ i.e. when 3 red balls present from picked balls.

$$P[F = 4] = \frac{1}{16}$$ i.e. when 4 red balls are present from the picked balls.

5.4.1 Random Number Generator or Randomizer

A random number generator or randomizer makes use of randomized algorithms. In randomized algorithms the decision is always taken based current output.

For a randomized algorithm -

i) Execution time may vary from run to run.

ii) For different inputs there may be different outcomes on each execution.

iii) For same inputs there may be different outcomes on each execution.

There are two types of randomized algorithms.

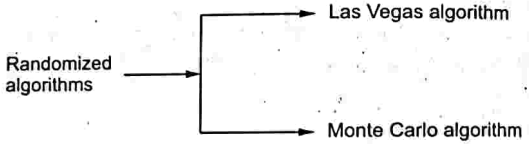


Fig. 5.4.1

Las Vegas algorithms : Are the type of randomized algorithms which produce same outcome on each execution for same input.

Monte Carlo algorithms : Are the type of randomized algorithms which produce different outcomes on each execution for same input.

5.4.2 Examples

Example 1 : Identifying the repeated elements

Consider an array a [ ] which consists of n elements. This array may contain n/2 distinct elements and $\frac{n}{2}$ duplicate elements. Then we have to identify repeated or duplicate elements. Now if we have some deterministic algorithm for solving this

problem then we may require $\frac{n}{2}$ +2 time, steps. But there is a simple and efficient Las Vegas algorithm then it may take O(logn) time. The notation used to denote running time of Las Vegas is $\tilde{O}$ ( ). Hence for finding out the duplicate elements using Las Vegas $\tilde{O}$(logn) time will be required. The working of this algorithm will be as follows.

We will pick up 2 elements randomly and check whether they are equal or not. But in this random pick up there may be a chance of picking same two elements many times. Hence we must check whether two elements that we have picked randomly are from different positions and whether they are similar or not. The algorithm can be as given below.

```
Algorithm Repeated (a, n)
{
//Problem Description : This algorithm finds
//repeated elements from a [1....n]
while (true)
{
    i ← Random ( ) % (n+1)
    j ← Random ( ) % (n+1)
    if (( i! = j) AND (a[i] = a[j])) then
      return i
}
} //end of algorithm
```

Different positional elements with same value.

As each while loop takes O(1) time the algorithm takes $\tilde{O}$ (logn) time.

Example 2 : Primality testing

Let us now discuss another problem which can be solved using randomized algorithm is **primality testing**. 'Deciding whether the given number n is prime or not is a problem of primality testing'. The application of primality testing is cryptology.

Any integer greater than one is said to be prime if it is divisible by 1 or by that number itself. We consider 1 as non-prime number but 2, 3, 5, 7, 11 and 13 are some prime numbers. But if a number n is non-prime (or composite) then it must have a divisor $\leq \lfloor \sqrt{n} \rfloor$. To check whether given number n is prime or not we must check every m elements from interval 2 to $\lfloor \sqrt{n} \rfloor$ whether m divides n. If there is no such element which divides n then n is prime. Otherwise it is non-prime or composite. The following algorithm is used for primality testing.

```
Algorithm Prime_number (num)
{
//Problem Description : If num is prime
//then it returns true otherwise false
      j ← num – 1
for (j←1 to n) do //n indicates some range
{
      temp ← j
      y ← 1
      a ← Random ( ) mod j + 1
      temp1 ← a
      while (temp > 0) do
      {
            while (temp mod 2 == 0)
            {
                  temp 1 ← temp1² mod num
                  temp ←| temp / 2 |
            }
            temp ← temp – 1

            y ← (y*temp1) mod num
      }
      if (y ! = 1) then
      return 0
      }
      return 1
}
```

Choosing some random number from 1 to num–1

Computing a^{num-1} mod num

Not a prime number

In the above algorithm, we have used

$$a^{n-1} \equiv 1 \bmod n$$

This equation is from **Fermat's theorem**. Suppose, we want to test if n is prime, then we can pick random a's in the interval, and see whether equality holds or not. If equality does hold then that means n is not prime. But even if equality holds then we can say that n is probably prime. Thus from above algorithm may or may not get correct prime number.

Example 5.4.1 *Explain randomized algorithm for quick sort.* **SPPU : Dec.-18, Marks 8**

Solution : The randomized algorithm for quick sort is as follows -

RANDOMIZED-PARTITION(A, p,r)

1 i ← RANDOM(p,r)

2 exchange A[r] ↔ A[i]

3 return PARTITION(A, p,r)

The new quicksort calls RANDOMIZED-PARTITION in place of PARTITION :

RANDOMIZED-QUICKSORT(A, p,r)

1 if p < r
2 then q ← RANDOMIZED-PARTITION(A, p,r)
3 RANDOMIZED-QUICKSORT(A, p, q – 1)
4 RANDOMIZED-QUICKSORT(A, q + 1,r)

5.4.3 Advantages and Disadvantages

There are two major advantages of randomized algorithms.

1. These algorithms are simple to implement.
2. These algorithms are many times efficient than traditional algorithms.

However randomized algorithms may have some drawbacks -

1. The small degree of error may be dangerous for some applications.
2. It is not always possible to obtain better results using randomized algorithm.

Review Questions

1. *Write a short note on randomized algorithm.* **SPPU : Dec.-15 (End Sem.), Marks 8**
2. *Explain the concept of randomized algorithm and approximation algorithm in brief with example.* **SPPU : May-16 (End Sem.), Marks 8**
3. *Explain the concept of randomized algorithm and approximation algorithm in brief with example.* **SPPU : May-19, Marks 8**
4. *Explain the concept of - randomized algorithm.* **SPPU : Dec.-19, Marks 4**

5.5 Introduction to Approximate Algorithms

SPPU : May-11, Dec.-19, Marks 10

In this section we will discuss two important issues namely, **"What is NP-hard problem ?"** and **"How approximation algorithms are used for NP-hard problems ?"** Let us start our discussion with the understanding of NP-hard problems.

In computational complexity theory there are different types of problems. Some problems are **decision problems** for which answer is yes or no, others are **search problems** and many others are **optimization problems.**

The solvability of problems in polynomial time is generally tested by **nondeterministic Turing machines.** Hence the complexity class of problems that are

intrinsically **harder** than those that can be solved by a nondeterministic turing machine in polynomial time are called NP-hard problems.

Let us have a formal definition of NP-hard.

If **decision version** of a combinatorial optimization problem is proved to be **NP-complete** then optimization version is **NP-hard.**

For example : Consider a problem of Hamiltonian cycle. If we want to find Hamiltonian cycle with length less than k then this is a decision problem and it is a NP complete problem. Because it is easy to determine the Hamiltonian cycles with length less than k. But (optimization version) "What is the shortest Hamiltonian cycle?" is a NP-hard problem. Because it is not easy to determine if the cycle obtained is shortest or not.

Approximation Algorithm

The approximation algorithms are algorithms used to find approximate solutions to optimization problems. Approximation algorithms are often associated with NP-hard problems because it is very difficult to get an efficient polynomial time exact algorithm for solving NP-hard problems. Note that approximate algorithms are mainly useful for solving those problems where exact polynomial algorithms are known but running such algorithm is too expensive because of their sizes of data sets. The philosophy behind approximation algorithm is *find 'good' solution fast.* That means you won't get the exact solution but you will get the approximate solution quickly. For approximation algorithm we start with inaccurate data, hence optimization may be as good as optimal solution. Hence approximation algorithms are based on heuristics (i.e. proceeding to solution by trial and error).

A **heuristics** can be defined as a collection of common sense rules defined from experience.

For example : In travelling salesperson problem going to next nearest city, or in knapsack problem start with highest value and minimum weight item.

Accuracy Ratio

It is always necessary to know the accuracy of approximation to the actual optimal solution. Hence accuracy ratio is defined as

$$r(s_a) = \frac{f(s_a)}{f(s^*)}$$

where s_a denotes approximate solution, $r(s_a)$ denotes accuracy ratio, $f(s_a)$ is a value of objective function for solution given by approximation algorithm, $f(s^*)$ is a

value of objective function. Generally $r(s_a) >= 1$. When $r(s_a)$ reaches close to 1 then it is a better approximate solution.

Performance Ratio

The best upper bound on accuracy ratio taken over all instances of the problem is called performance ratio. It is denoted by R_A. By knowing performance ratio one can judge quality of approximation algorithm. The approximation algorithms with R_A value nearer to 1 is supposed to be a better approximation algorithm.

c-Approximation Algorithm

If there exists a value c which is $>= 1$ and $r(s_a) \leq c$ for all instances of problem then that algorithm is called c-approximation algorithm. If c value is 1 then corresponding c-approximation algorithms are good. For a c-approximation algorithm for any instance of problem is -

$$f(s_a) \leq c\, f(s^*)$$

5.5.1 Approximation Algorithms for the Travelling Salesperson Problem

The Travelling Salesman Problem (TSP) is based on the idea of obtaining optimum tour when travelling between many cities. The decision version of this algorithm belongs to a class of NP-complete problem and optimization version of this algorithm belongs to NP-hard class of problems. There are two approximation algorithms used for TSP : A NP-hard class of problem and those are

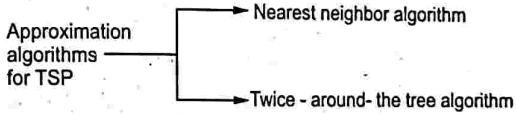


Fig. 5.5.1

Let us discuss each algorithm with some simple example.

1. Nearest neighbor algorithm

This algorithm is based on the idea of choosing nearest-neighbor while travelling from one city to another. This nearest neighbor should be unvisited one. Let see the algorithm.

1. Start at any arbitrary city.
2. Repeat until all the nodes are visited : Go to the nearest city (the unvisited one) each time.
3. Return to the starting city.

Let us apply this algorithm on some example.

Example 5.5.1 *Using nearest-neighbor algorithm, obtain the optimal solution for given travelling salesman problem.*

Also find the accuracy ratio for the same.

SPPU : May-11, Marks 10

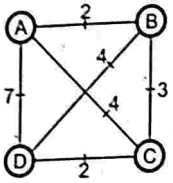


Fig. 5.5.2

Solution : We will apply nearest neighbor algorithm for the tour for given traveling salesman problem. Hence nearest neighbor will produce :

A-B-C-D-A.,

$\therefore \quad s_a = 2 + 3 + 2 + 7$

i.e. $\quad s_a = 14$

Now the optimal solution can be obtained by exhaustive search. Then tour will be :

A-B-D-C-A

$\therefore \quad s^* = 2 + 4 + 2 + 4$

$s^* = 12$

The accuracy ratio $r(s_a)$ will be -

$$r(s_a) = \frac{f(s_a)}{f(s^*)} = \frac{14}{12}$$

$$r(s_a) = 1.16$$

Hence the tour s_a is 16 % longer than optimum tour s^*.

This algorithm is simple to use but it has some drawback. The major drawback of this method is that there may exist some long path which has to be traversed to return to the starting city (Note that we will go on choosing small distance of the neighbours but to return back to starting point there may not be short path and in such a case we have to traverse a long distance only !). For instance, in above discussed example in the tour A-B-C-D-A, the distance D-A will plays an important role (i.e. edge A-D), hence accuracy ratio $r(s_a)$ will be

$$r(s_a) = \frac{f(s_a)}{f(s^*)}$$

$$r(s_a) = \frac{7+w}{12}$$ where w is the distance (A-D).

As value of w increases, $r(s_a)$ increases. For larger value of w, $r(s_a)$ tends to ∞. Hence our approximation algorithm will fail to obtain the optimal solution.

2. Twice around-the-tree algorithm

This algorithm is based on important subset instance called **Euclidean** by which accuracy for the nearest neighbor algorithm can be obtained -

Eudidiean instances satisfy following conditions -

1. Triangle inequality

dist [i, j] ≤ dist [i, k] + dist [k, j]

where i, j and k denote cities.

2. Symmetry

dist [i, j] = dist [j, i]

The distance from city i to j should be same as distance between city j to i.

The Eudidiean instances satisfy following conditions about the accuracy ratio

$$r(s_a) \quad \text{i.e.} = \frac{f(s_a)}{f(s^*)} \leq \frac{1}{2}\left(\lceil \log_2 n \rceil + 1\right)$$

where n is total number of cities.

Now let us discuss another approximation algorithm i.e. twice-around-the tree algorithm. This is a 2-approximation (i.e. c-approximation with c to be 2) algorithm for travelling salesman problem with Euclidean distances.

Algorithm

1. Compute minimum spanning tree from the given graph.
2. Start at any arbitrary city and walk around the tree (i.e. depth first search) and record nodes visited.
3. Eliminate duplicates from the generated node list.

Example 5.5.2 *Consider the graph as given below and apply the twice-around-the-tree algorithm.*

Solution :

Step 1 : Now we will obtain the Minimum Spanning Tree (MST) for given graph.

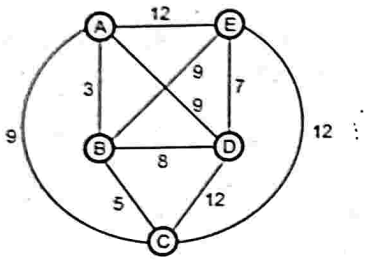


Fig. 5.5.3

Step 2 : Start from city A and have a DFS walk around the tree.

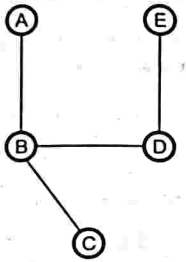


Fig. 5.5.4

Step 3 : Record the visited nodes

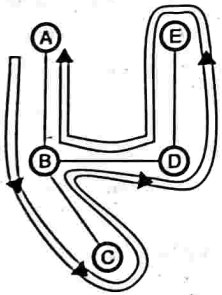


Fig. 5.5.5

A - B - C - B - D- E - D - B - A. Eliminate duplicates then A-B-C-D-E-A . This basically gives Hamiltonian circuit.

But the tour obtained is not the optimal tour.

Review Questions

1. *What is approximation algorithm ? Explain the approximation algorithm for traveling salesman problem.*
2. *Explain the concept of approximation algorithm.* **SPPU : Dec.-19, Marks 4**

5.6 Embedded Algorithms

SPPU : Dec.-15 19, May-18, 19, Marks 8

Definition of embedded system : The embedded system is a computer system with a dedicated function within a larger mechanical or electrical system.

The embedded system typically does not look like a computer, often there is no keyboard or monitor or mouse. But like a computer it has a processor, memory and software. The word embedded means a permanent part within a big system. For example - Smart Card, Washing Machines, Micorwave Ovens, answering machine, Mobile Phones, digital Cameras, Vending Machines and so on.

Characteristics of embedded systems

1. **Single Functional :** The embedded systems are usually designed to perform dedicated functions.
2. **Tightly constrained :** Embedded systems are computing systems with tightly coupled hardware and software integration.
3. **Part of large systems :** The word embedded means the system is an integral part of large system.
3. **Reactive and Real time :** These systems are real time systems and respond to the events and triggers. These systems are generally time bounded. Modern embedded systems have user friendly interface.
4. **Programmable :** The embedded systems contain at least one programmable unit (micro-controller).

5.6.1 Embedded System Scheduling

- Scheduling is a method by which threads and processes can access the system resources at specific time. The scheduling is generally done in order to balance the load and share system resources effectively.
- In real time environment such as embedded systems the scheduler must ensure that processes can meet deadlines
- Priorities are assigned to tasks, and the operating system always executes the ready task with highest priority.
- Most algorithms are classified as fixed priority, dynamic priority, or mixed or hybrid priority. A mixed-priority algorithm has both static and dynamic components.

Fixed priority algorithm

- The fixed-priority algorithm assigns all priorities to the tasks at design time. Tho priorities remain constant for the lifetime of the task.
- The implementation of this algorithm is as follows -

Step 1 : Initially the tasks reside in the **sleep queue.**

Step 2 : When the tasks are ready for execution they enter in the **ready queue.** O ready queue for each distinct priority is maintained.

Step 3 : The highest priority task will be executed first.

Fig. 5.6.1 (a) illustrates this implementation.

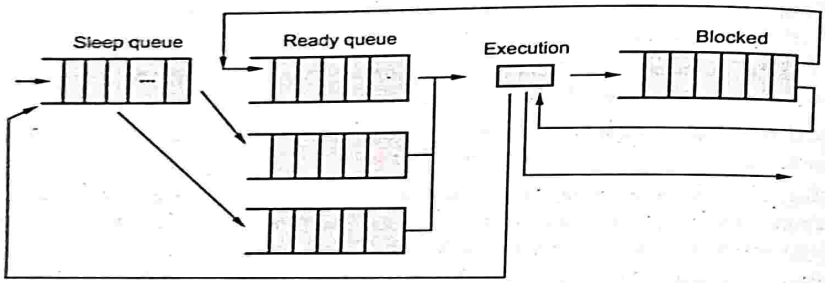


Fig. 5.6.1 (a) Priority queues for fixed priority algorithm

- The advantage is this algorithm is it is simple to execute.

Dynamic priority algorithm

- All the scheduling decisions such as assignment of priorities to the task is at run time. This scheduling algorithm takes task deadlines into consideration which are used for deciding the priorities. The mutual exclusion and synchronization is enforced in dynamic priority algorithm.
- The implementation of this algorithm is as follows -

Step 1 : Initially the tasks reside in the **sleep queue.**

Step 2 : When the tasks are ready for execution they enter in the ready queue. One ready queue for each distinct priority is maintained.

Step 3 : For each highest priority task, check the deadlines of highest priority queue and then these tasks will be executed.

Fig. 5.6.1 (b) illustrates this implementation

- The advantage of this algorithm is that it is flexible and it is used only when the actual resources are claimed.

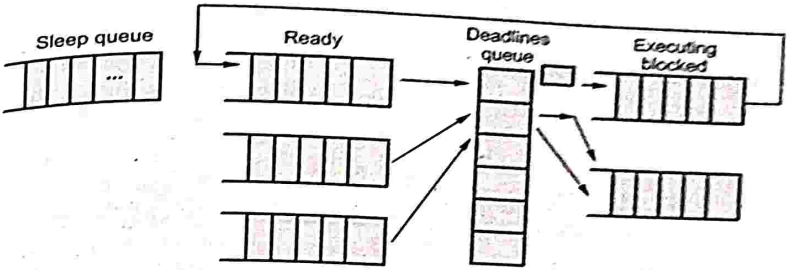


Fig. 5.6.1 (b) Priority queues for dynamics priority algorithm

5.6.2 Sorting Algorithm for Embedded Systems

The sorting algorithms for embedded systems must have following characteristics.

1. The sorting algorithm must be in place.
2. The algorithm must not be recursive.
3. The code size must be as per the need of the problem.
4. The sorting algorithm should be such that its running time should increase linearly or logarithmically with the number of elements sorted.

The sorting algorithm is called in place sorting algorithm if it requires very little additional space besides the initial array holding the elements that are to be sorted. The in-place sorting algorithm is an algorithm in which the input is normally overwritten by output and to execute the sorting method it does not require any more additional space.

Various sorting methods that can be used in embedded systems are Bubble sort, insertion sort, heap sort and so on. Let us discuss the method of sorting using insertion sort technique

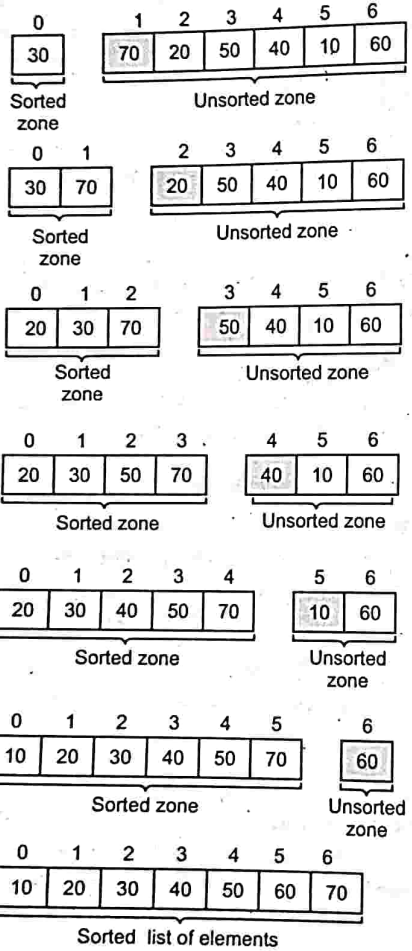
Insertion sort : Generally the insertion sort maintains a zone of sorted elements. If any unsorted element is obtained then it is compared with the elements in the sorted zone and then it is inserted at the proper position in the sorted zone. This process is continued until we get the complete sorted list. Hence the name of this method is insertion sort.

For example

Consider a list of elements as,

0	1	2	3	4	5	6
30	70	20	50	40	10	60

The process starts with first element



Algorithm

Although it is very natural to implement insertion using recursive(top down) algorithm but it is very efficient to implement it using bottom up(iterative) approach.

```
Algorithm Insert_sort(A[0...n-1])
//Problem Description: This algorithm is for sorting the
//elements using insertion sort
//Input: An array of n elements
//Output: Sorted array A[0...n-1] in ascending order
for i ← 1 to n-1 do
{
    temp ← A[i]//mark A[i]th element
    j ← i-1//set j at previous element of A[i]
    while(j>=0)AND(A[j]>temp)do
    {
    //comparing all the previous elements of A[i] with
    //A[i].If any greater element is found then insert
    //it at proper position
    A[j+1] ← A[j]
    j ← j-1
    }
    A[j+1] ← temp //copy A[i] element at A[j+1]
}
```

Analysis

Best case :

It happens for **almost sorted array.** The number of key comparisons can be

$$C_{best}(n) = \sum_{i=1}^{n-1} 1$$

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

$$= (n-1)$$

$$\therefore \quad C_{best}(n) = \Theta(n)$$

Thus the best case time complexity of insertion sort is $\Theta(n)$.

Average case :

When an array contains randomly distributed elements then it results in average case time complexity. As compared to worst case atleast half number of comparisons are needed to obtain average case complexity.

$$\therefore \quad C_{avg}(n) = \frac{(n-1)\,n}{2} \times \frac{1}{2}$$

$$= \frac{n^2 - n}{4} \approx \frac{n^2}{4}$$

$$\therefore \quad C_{avg}(n) = \Theta(n^2)$$

Thus the average case time complexity of insertion sort is $\Theta(n^2)$.

The algorithm can be analysed for worst case, best case and average case time complexities.

Worst case :

When the list is sorted in descending order and if we want to sort such a list in ascending order then it results in worst case behavior of algorithm.

1. The basic operation of algorithm is a key comparison A[j]>temp. [Note that the operation given in the innermost loop is called basic operation].
2. The basic operation depends upon the size of list i.e. n–1. Hence the basic operation is considered for the range 1 to (n – 1).
3. The number of key comparisons for such input is

$$C_{worst}(n) = \sum_{i=1}^{n-1} \left(\sum_{j=0}^{i-1} 1 \right)$$

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

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

$$C_{worst}(n) = \Theta(n^2)$$

$\because \sum_{l}^{u} 1 = u - l + 1$

$\therefore \sum_{j=0}^{i-1} 1 = (i-1) - 0 + 1 = i$

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

That means worst case time complexity of insertion sort is $\Theta(n^2)$.

Review Questions

1. *What is embedded system ? Explain embedded system scheduling.*

SPPU : Dec.-15 (End Sem.), Marks 9, May-19, Marks 8

2. *What is embedded system ? Explain embedded sorting algorithm*

SPPU : May-18, Dec.-19, Marks 8

Unit VI

6 Multithreaded and Distributed Algorithms

Syllabus

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..

Contents

6.1 Multithreaded Algorithms

SPPU : May-18, 19, Marks 9

6.1.1 Introduction

- There are serial algorithms that run on **uniprocessors** while the **parallel algorithms** run on **multiprocessors.**
- Many languages (e.g. Java) support the production of separately runnable processes called **threads.** Each thread looks like it is running on its own and the operating system shares time and processors between the threads. In the multi-threading model, the exact parallel implementation is left to the operating system.
- **Multithreaded algorithms** are **parallel algorithms** which can run on parallel computers that permits multiple instructions to execute concurrently. This exhibits the model of **dynamic multithreaded algorithms.**
- **Static threading** is a kind of programming in which the processors on which the thread is executed is managed explicitly. On the other hand, during **dynamic multithreading** the programmers specify the parallelism. The **concurrency platform** manages parallel computing resources.
- Dynamic multithreading allows programmers to specify parallelism in applications without worrying about communication protocols, load balancing, and so on.
- The **Concurrency constructs** are used to achieve dynamic multithreading. In this concurrency constructs there are three keywords that are used to achieve parallel computing practice -
  - **parallel :** This keyword is used along with the for loop to indicate that each iteration can be executed in parallel.
  - **spawn :** Use of this keyword allows to create a new sub-process and then keep executing current process.
  - **sync :** This keyword forces to wait until all active parallel threads created by the instance of program finish.
  - These keyword help to bring the parallelism without affecting the remaining sequential program.
- **Example**

We will have both simple recursive Fibonacci algorithm without parallelism and with parallelism

Without Parallelism

```
Fib(n)
{
if n<=1
    return n;
else
    x=Fib(n-1);
    y=Fib(n-2);
return x + y;
}
```

With Parallelism

```
ParFib(n)
{
if n<=1
    return n
else
    x=spawn ParFib(n-1);
    y=ParFib(n-2);
      sync
return x + y;
}
```

The parallel Fibonacci algorithm achieves parallelism with the help of keywords **spawn** and **sync**. The keyword **spawn** create a child which is computing **parFib(n-1)** and there is also execution of parent which ultimately executes procedure **ParFib(n-2).** Thus both the processes are executed in parallel.

Finally these executions are synchronized using **sync** keyword.

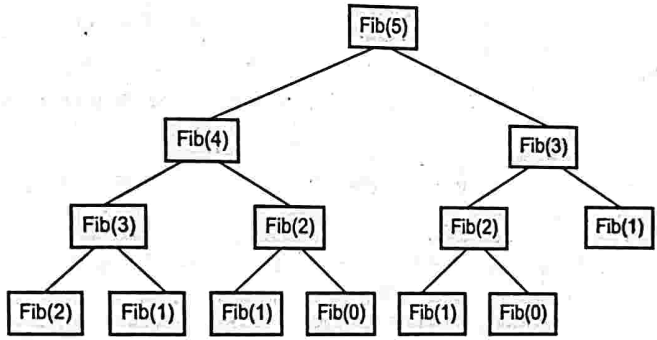
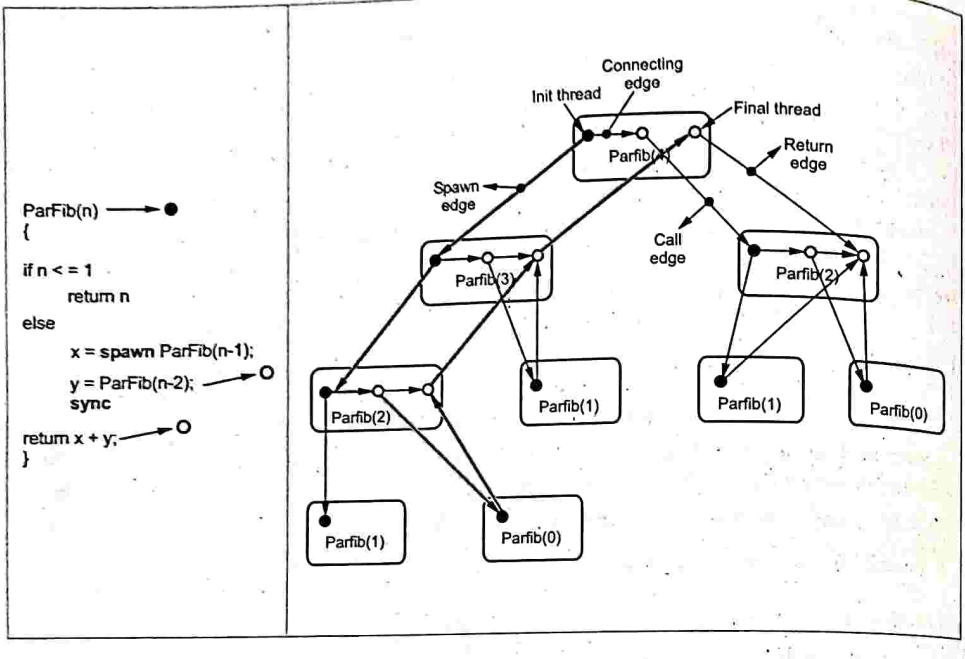


Fig. 6.1.1 Tree of recursion

Model of Multithreaded Execution

- This model helps to think a **multithreaded computation**(a set of runtime instructions) executed by a processor on behalf of a multithreaded program as a **directed acyclic** graph G=(V,E) which is also called as **Computational DAG**
- **Example**

  As an example the parallel algorithm for Fibonacci number is used.



In above fig,

- The vertices V represent the instructions or strands the sequence of non parallel instructions.
- Edges in E represent dependencies between instructions or strands: (u, v) ∈ E means u must execute before v.
- Following terms are used in this model for various types of edges -
  - (i) **Continuation edges** (u, v) are drawn horizontally and indicate that v is the successor to u in the sequential procedure.
  - (ii) **Call edges (u, v)** point downwards, indicating that u called v as a normal subprocedure call.
  - (iii) **Spawn edges (u, v)** point downwards, indicating that u spawned v in parallel.
  - (iv) **Return edges** point upwards to indicate the next strand executed after returning from a normal procedure call, or after parallel spawning at a sync point.
- A strand with multiple successors means all but one of them must have spawned.
- A strand with multiple predecessors means they join at a sync statement.

Example 6.1.1 *Explain performance measure of Fibonacci(6) execution with suitable diagram.*

SPPU : May-18, Marks 6

Solution :

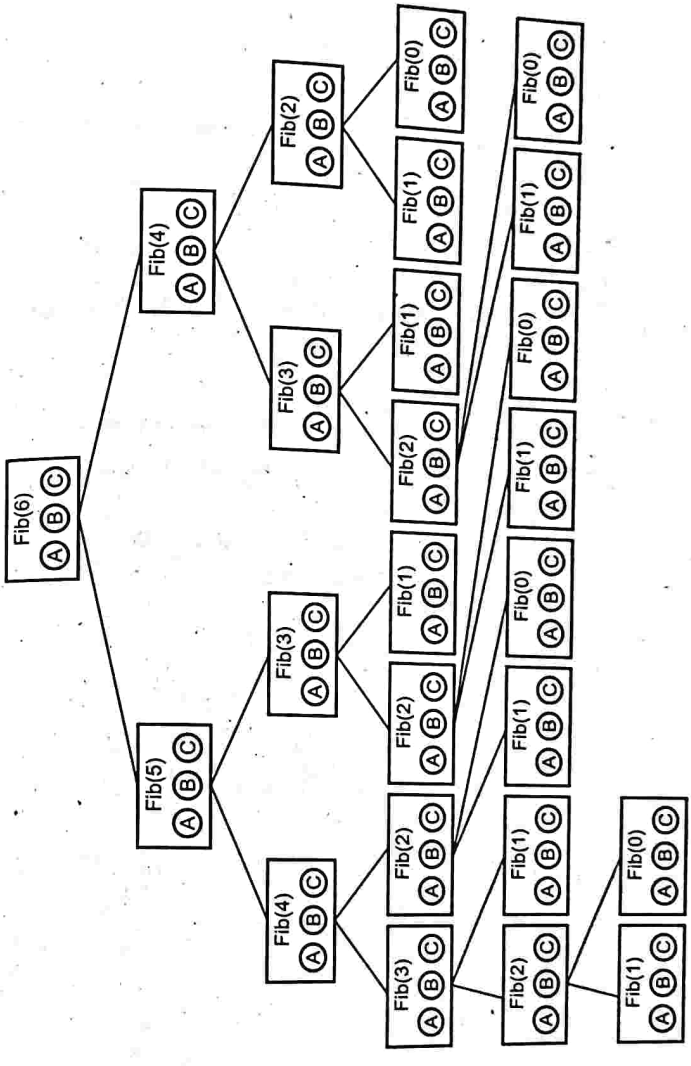


Fig. 6.1.2

6.1.2 Performance Measure

Theoretical efficiency of multithreaded algorithms is computed using two metrics -

1. Work 2. Span

Let us discuss them in detail

1. Work : The work is defined as the total time to execute the entire computation o one processor. In other words, the work is the sum of the times taken by each of th strands.

Work Law (T_1)

Let,

T_1 is the total time to execute algorithm on one processor.

T_p is the total time taken to execute algorithm on P processors.

As $T_p \geq T_1$ we get the work law as

$$T_p \geq T_1/P$$

2. Span (T_∞) : The span is the longest time to execute the strands along any path in th directed acyclic graph.

Span Law

Let, T_∞ is the total time to execute an algorithm on an infinite number of processors.

T_∞ is called the span because it corresponds to the longest time to execute th strands along any path in the computation dag.

Hence the **span law** states that a P-processor ideal parallel computer cannot ru faster than one with an infinite number of processors :

$$T_p \geq T_\infty$$

Performance Measure of Fibonacci(4) Execution

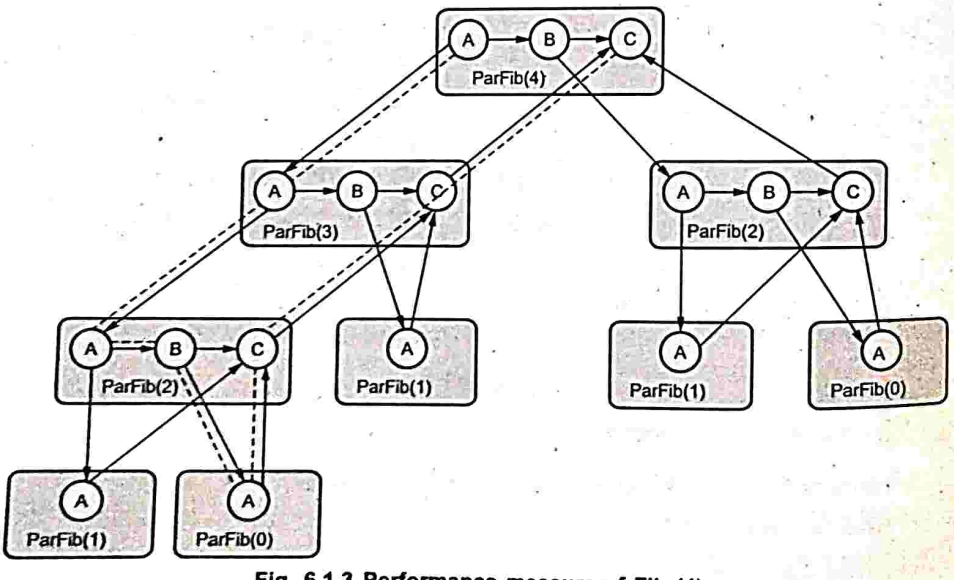


Fig. 6.1.3 Performance measure of Fib (4)

Here there are in all 17

vertices = 17 threads

8 vertices on longest path

Assuming unit time for each thread we get

Work = 17 time units

Span = 8 time units

Speedup

The ratio T_1 / T_P defines how must speedup we get with P processors.

By the work law,

$$T_P \geq T_1 / P,$$

so $$T_1 / T_P \geq P$$

That means one cannot have any more speedup than the number of processors.

When the speedup $T_1 / T_P = \Theta(P)$ we have **linear speedup**, and when $T_1 / T_P = P$ we have **perfect linear speedup.**

Parallelism

The ratio T_1 / T_∞ of the work to the span gives the potential parallelism of the computation.

Parallelism = Work/Span

It can be interpreted in three ways :

(i) **Ratio :** The average amount of work that can be performed for each step of parallel execution time.

(ii) **Upper bound :** the maximum possible speedup that can be achieved on any number of processors.

(iii) **Limit :** The limit on the possibility of attaining perfect linear speedup. Once the number of processors exceeds the parallelism, the computation cannot possibly achieve perfect linear speedup. The more processors we use beyond parallelism, the less perfect the speedup.

Parallel Slackness

The parallel slackness for multithreaded computation executed on ideal parallel computers with P processors is the ratio of **parallelism** by P

It is given as -

$$(T_1 / T_\infty) / P$$
$$= T_1 / (P \cdot T_\infty)$$

If slackness is less than 1 then perfect linear speedup is not possible: you have mor processors than you can make use of. If slackness is greater than 1, then the work pe processor is the limiting constraint and a scheduler can strive for linear speedup b distributing the work across more processors.

6.1.3 Analyzing Multithreaded Algorithms

First of all Let us discuss how to analyse work and span.

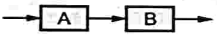
Analysing work for parallel algorithms is very simple. Just ignore the parall constructs and analyze the serial algorithm. For example, the work of ParFib(n) is T_1(n = T(n) = Θ(Fn).

Analyzing span requires additional efforts.

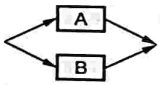
For example -

Consider that there are two processors A and B. Following figure shows what will b the work and span when these processors are connected in series and in parallel

In series simple sum of



$$\text{Work} : T_1(A \cup B) = T_1(A) + T_1(B)$$
$$\text{Span} : T_\infty(A \cup B) = T_\infty(A) + T_\infty(B)$$



$$\text{Work} : T_1(A \cup B) = T_1(A) + T_1(B)$$
$$\text{Span} : T_\infty(A \cup B) = \max(T_\infty(A), T_\infty(B))$$

Analysis of Parallel Fibonacci Algorithm

Consider the parallel Fibonacci algorithm as follows -

```
ParFib(n)
if n < 2 then return n;
x = spawn ParFib(n-1);  // parallel execution
y = spawn ParFib(n-2) ;  // parallel execution
sync;  // wait for results of x and y
return x + y;
```

Work Analysis

The work T_1 is straightforward, since it amounts to compute the running time of the serialized algorithm.

$$T_1(n) = T_n(n) = \Theta\left(\left(\frac{(1+\sqrt{5})}{2}\right)^n\right) = (\phi)^n$$ Here ϕ is called golden ratio whose value is $\left(\frac{(1+\sqrt{5})}{2}\right)$

Span Analysis

The span T_∞ in the longest path in the computational DAG. Since ParFib(n) spawns

- ParFib(n-1)
- ParFib(n-2)

we have

$$T_\infty(n) = \max(T_\infty(n-1), T_\infty(n-2)) + \Theta(1)$$
$$= T_\infty(n-1) + \Theta(1)$$

which gives

$$T_\infty(n) = \Theta(n)$$

Thus the parallelism for Parallel Fibonacci algorithm is

$$T_1(n)/T_\infty(n) = \Theta\left(\left(\frac{(1+\sqrt{5})}{2}\right)^n \Big/ n\right)$$
$$= \Theta\left((\phi)^n/n\right)$$

6.1.3.1 Parallel Loops

In parallel algorithms we can use the keyword **parallel** with **for** construct instead of **spawn.**

Suppose we want to multiply an n x n matrix $A = (a_{ij})$ by an n-vector $x = (x_j)$. This yields an n-vector $y = (y_i)$ where :

$$y_i = \sum_{j=1}^{n} a_{ij} x_j$$

for $i = 1, 2, ..., n.$

The following algorithm does this in parallel :

```
MatrixVector(A,x)
1  n = A . rows
2  let y be a new vector of length n
3  parallel for i = 1 to n
4  y_i = 0
5  parallel for i = 1 to n
6  for j = 1 to n
7  y_i = y_i + a_ij x_j
8  return y
```

Row multiplication

The **parallel for** keywords indicate that each iteration of the loop can be executed concurrently.

Analysis

In above algorithm

$T_1(n) \in \Theta(n^2)$

$$T_\infty(n) = \Theta(\log n) + \Theta(n)$$

Parallel for — Row multiplication

6.1.3.2 Race Conditions

- A multithreaded algorithm is deterministic if and only if does the same thing on the same input, no matter how the instructions are scheduled.
- A multithreaded algorithm is nondeterministic if its behavior might vary from run to run.
- The multithreaded algorithms are supposed to be deterministic but they are actually nondeterministic because they contain **determinacy race** situation.
- **Determinacy race** is a situation which occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write. This condition is called race **condition.**
- For example - Consider following algorithm, that brings the race condition

```
RaceCondition()
a = 0
parallel for i = 1 to 2 do
                    //parallel execution of instructions
  a = a+1           //Writing at the same memory location.
print a
```

- There are three operations performed in above algorithm and those are
  1) Read a from memory into one of the processor's registers
  2) Increment the value of the register
  3) Write the value in the register back into a in memory
- This can be illustrated by following Fig. 6.1.4.

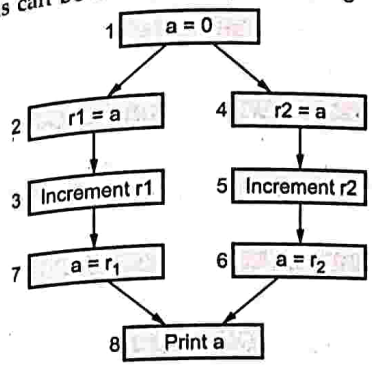


Fig. 6.1.4 Determinacy race

Step	a	r_1	r_2
1	0	-	-
2	0	0	-
3	0	1	-
4	0	1	0
5	1	1	1
6	1	1	1
7		1	1

- In above case, the execution order (1, 2, 3, 4, 5, 7, 6, 8) or(1, 2, 4, 3, 5, 7, 6, 8) causes bugs.
- Of course, many executions do not cause the bug. For example, if the execution order were (1, 2, 3, 7, 4, 5, 6, 8) or (1, 4, 5, 6, 2, 3, 7, 8), we would get the correct result.
- Generally, most orderings produce correct results-For instance in given Fig. 6.1.4, the algorithm instructions on the left execute before the instructions on the right, or vice versa. But some orderings generate improper results when the instructions interleave.

Review Questions

1. *Define performance measure of multithreaded algorithms. Write multithreaded algorithm for fibonacci series and explain performance measure of fibonacci(6) execution with suitable example*
   SPPU : May-18, Marks 9
2. *Explain multithreaded algorithms. How to analyze multithreaded algorithms ? What is race condition, parallel loops ?*
   SPPU : May-19, Marks 9

6.2 Problem Solving using Multithreaded Algorithms

SPPU : May-18, 19, Dec.-18, 19, Marks 9

6.2.1 Multithreaded Matrix Multiplication

Consider the matrix multiplication algorithm. We can write the parallel matrix multiplication algorithm as follows -

```
MatMul(matrix A,matrix B)
{
    n=A.rows;
    Let C is a new n*n matrix
    parallel for i=1 to n
        parallel for j=1 to n
            cij=0
            for k=1 to n
                cij=cij+aik*bkj;
    return C
}
```

In above algorithm,

$$T_1(n) = \Theta(n^3) \quad \ldots(1)$$

The span of this algorithm

$$T_\infty(n) = \Theta(n) \quad \ldots(2)$$

From equations (1) and (2), we get -

Hence the parallelism is

$$T_1(n)/\ T_\infty(n) = \Theta(n^3)/\ \Theta(n) = \Theta(n^2).$$

Multithreaded Matrix Multiplication Algorithm using Divide and Conquer Method

Matrix multiplication operation is possible by applying divide and conquer strategy.

To multiply two $n \times n$ matrices, we perform 8 matrix multiplications of $(n/2) \times (n/2)$ matrices and one addition of $n \times n$ matrices.

Following formula is used to multiply the two matrices using divide and conquer strategy

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Then we can write the product of two matrices as -

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$= \begin{pmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{pmatrix} + \begin{pmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{pmatrix}$$

Using the above formula, the parallel matrix multiplication algorithm is as given below

```
PMatMulRec(Matrix C, Matrix A, Matrix B)
```

```
1  n = A.rows
2  if n == 1
3  c11 = a11b11
4  else let T be a new n * n matrix
5  partition A, B, C, and T into n/2*n/2 submatrices
       A11,A12,A21,A22, B11,B12,B21,B22,C11,C12,C21,C22;
       and T11, T12, T21, T22 respectively
6  spawn PMatMulRec(C11,A11,B11)
7  spawn PMatMulRec (C12,A11,B12)
8  spawn PMatMulRec(C21,A21,B11)
9  spawn PMatMulRec(C22,A21,B12)
10 spawn PMatMulRec(T11,A12,B21)
11 spawn PMatMultRec(T12,A12,B22)
12 spawn PMatMultRec(T21,A22,B21)
13 PMatMultRec (T22,A22,B22)
14 sync
15 parallel for i = 1 to n
16 parallel for j = 1 to n
17 cij = cij + tij
```

Analysis

In above algorithm, there are 8 multiplications performed recursively of n/2*n/2 matrices. Each recursion is of $\Theta(1)$ time. Hence in all we get 8 $M_1(n/2)$ Similarly there are two parallel for loops for performing matrix addition. Hence this addition costs $\Theta(n^2)$

The recursion for work is then given by

$$M_1(n) = 8M_1(n/2)+\Theta(n^2)$$

$$M_1(n) = \Theta(n^3) \quad \ldots(1)$$

Similarly to determine the span $M_\infty(n)$, the eight parallel recursive calls that execute are of same size, the maximum span for any recursive call is just the span of any one. Similarly the span for two parallel for loops that perform addition is of Θ (logn).

Hence recurrence for span is given be

$$M_\infty(n) = M_\infty(n/2)+\Theta(\log n).$$

$$M_\infty(n) = \Theta(\log^2 n) \quad \ldots(2)$$

From equations (1) and (2) the parallelism can be

$$\text{Parallelism} = M_1(n)/M_\infty(n)$$
$$= \Theta(n^3)/\Theta(\log^2 n)$$

Thus parallelism is very high. High parallelism means we can dispatch more tasks simultaneously to different resources.

6.2.2 Multithreaded Merge Sort

- The key operation of merge sort algorithm is merging of two sorted sequences in the combined step.
- We call merge (A, a, b, c) where A is an array a, b and c are indices into array such that $a \le b \le c$.
- The procedure assumes that the subarrays A[a....b] and A[b + 1....c] are in sorted order. It merges them into sorted subarray that replaces the current subarray - A[a.....c].
- We will first discuss, how to parallelize the merge sort. In this version, we simply spawn the first recursive call. The Algorithm is as follows

```
MergeSort(Array A, a,c)
{
    If(a<c) then
        b=floor((a+c)/2)
        spawn MergeSort(A,a,b)
        MergeSort(A,b+1,c)
        Sync
        Merge(A,a,b,c)
}
```

Analysis

- There are two recursive calls for n/2 elements and merge operation is for all n elements. Hence the **work** for above algorithm is given by

$$MS_1(n) = 2MS_1(n/2) + \Theta(n)$$
$$= \Theta(n \log n) \quad \ldots(1)$$

- Similarly the **span** of above algorithm can be computed as

$$MS_\infty(n) = MS_\infty(n/2) + \Theta(n)$$
$$= \Theta(n) \quad \ldots(2)$$

- From equations (1) and (2), we get

$$\text{Parallelism} = MS_1(n)/MS_\infty(n)$$
$$= \Theta(\log n)$$

- This is low parallelism, meaning that even for large input we would not benefit from having hundreds of processors.

Multithreaded Merge Sort Algorithm

- For implementing the multithreaded merge sort, we have to make the merge operation parallel.
- The divide and conquer strategy will imply that the sorting is done to break the list into four lists. Two of them will be merged to form **head** and other two will be merged to form the **tail.**
- To find the four lists, following steps are applied -
  (1) Choose the longer list to be the first list, $T[a_1..c_1]$ in the Fig. 6.2.1.
  (2) Find the middle element (median) of the first list (x at b_1).
  (3) Use binary search to find the position (b_2) of this element if it were to be inserted in the second list $T[a_2..c_2]$.
  (4) Recursively merge.
    (a) The first list up to just before the median $T[a_1..b_1-1]$ and the second list up to the insertion point $T[a_2..b_2-1]$
    (b) The first list from just after the median $T[b_1+1..c_1]$ and the second list after the insertion point $T[b_2..c_2]$
- Assemble the results with the median element placed between them, as shown below.

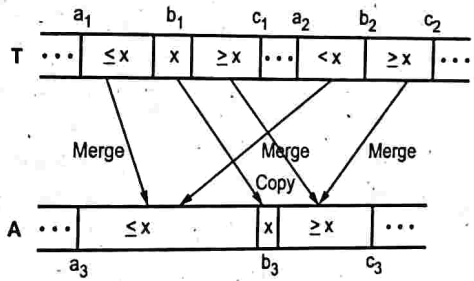


Fig. 6.2.1

Thus the parallel merge sort algorithm PMergeSort(Array A, a, c,Array B, m) which sorts the elements in array A[a...c] and stores them in B [m...m+c-a]

The complete multithreaded merge sort algorithm is as follows -

```
PMergeSort(Array A, a, c,Array B, m)
{
 n = c - a + 1
 if n == 1
      B[m] = A[a]
 else let T[1...n] be a new array
 b = floor((a+c)/2)
 b' = b-a+1
 spawn PMergeSort(A,a,b,T,1)
 PMergeSort(A,b+1,c,T,b'+1)
 sync
 PMerge(T,1,b',b'+1,n,B,m)
}
```

In above algorithm we have called PMerge i.e. parallel merge function. The algorithm for this is as given below -

```
PMerge(T, a1, c1, a2, c2,A, a3)
{
 n1 = c1 - a1 + 1
 n2 = c2 - a2 + 1
 if n1 < n2 // Here  n1 >= n2
      exchange a1 with a2
      exchange c1 with c2
      exchange n1 with n2
 if n1 == 0
      return
 else
      b1 = floor((a1 + c1)/2)
 b2 = BinarySearch(T[b1],T, a2,c2)
 b3 = a3 + (b1 - a1)+ (b2 - a2)
 A[b3]=T[b1]
 spawn PMerge(T, a1, b1-1, a2, b2-1,A, a3)
 PMerge(T, b1+1,c1,b2,c2,A,b3+1)
 sync
}
```

Analysis

- The work $PMS_1(n)$ of PMergeSort algorithm is given using parallel Merge i.e. PMerge algorithm. It is as follows -

$$PMS_1(n) = 2PMS_1(n/2) + PM1(n)$$
$$= 2PMS_1(n/2) + \Theta(n)$$
$$PMS_1(n) = \Theta(n \log n)$$

- The span $PM_\infty(n) = PMS_\infty(n/2) + PM_\infty(n)$
$$= PMS_\infty(n/2) + \Theta(\log^2 n)$$
$$PMS_\infty(n) = \Theta(\log^3 n)$$

- Therefore the parallelism is
$$PMS_1(n)/PM_\infty(n) = \Theta(n \log n / \log^3 n)$$
$$= \Theta(n/\log^2 n)$$

Review Questions

1. *Write and explain, multi-threaded merge sort algorithm.*

SPPU : May-18, Dec.-19, Marks 9, Dec.-18, Marks 8

2. *Give pseudo code for multithreaded matrix multiplication. Analyze the same.*

SPPU : May-19, Marks 9

6.3 Distributed Algorithms

SPPU : May-18, 19, Dec.-18, 19, Marks 9

6.3.1 Introduction

Definition : Distributed algorithms are those algorithms that are supposed to work in distributed network or on multiprocessor. The distributed netwok is created by interconnected processors.

- Distributed algorithms are a sub-type of parallel algorithm, typically executed concurrently, with separate parts of the algorithm being run simultaneously on independent processors, and having limited information about what the other parts of the algorithm are doing.
- Varied **application areas** in which the distributed algorithms are used are -
  - Distributed computing
  - Telecommunication
  - Scientific computing
  - Distributed information processing

- Distributed database systems
- Real time process control

- One of the major challenges in developing and implementing distributed algorithms is successfully coordinating the behavior of the independent parts of the algorithm in case of processor failures and unreliable communications links.

6.3.2 Distributed Breadth First Search

The distributed Breadth First Search algorithm minimizes the maximum communication time from the process at distinguished node to all other processes in the network.

For the Breadth First Search (BFS) we assume that the network is strongly connected and there is a distinguished source node i_0. The algorithm is supposed to output the structure of breadth first search directed tree of network graph rooted at i_0

The output should appear in distributed fashion - that means each process other than i_0 should have parent component that gets set to indicate the node which is its parent in the tree.

The algorithm is as follows -

Step 1 : At any point during execution, there is set of processes that is **marked** initially as i_0

Step 2 : Process i_0 sends out **search message** at round 1 to all its outgoing neighbors.

Step 3 : From round 2 to n if unmarked process receives **search message** , it marks itself and chooses one of the processes from which it receives search message as its **parent.** Then it sends the **search message** to all its outgoing neighbors..

Step 4 : Finally a BFS tree is obtained.

Analysis

The concurrent execution of BFS makes use of at the most $|E|$ messages where E is the number of edges or links in the network graph, each having b bits. Hence number of communication bits in the above algorithm is $O(|E|^2 \cdot b)$

6.3.3 Distributed Minimum Spanning Tree

Spanning Tree : A spanning tree of a graph G is a subgraph which contains all the vertices of G containing no circuit.

Minimum Spanning Tree (MST) :

A minimum spanning tree of weighted connected graph G is a spanning tree with minimum or smallest weight.

For example

Consider a graph G as given below. This graph is called weighted connected graph because some weights are given along every edge and the graph is a connected graph.

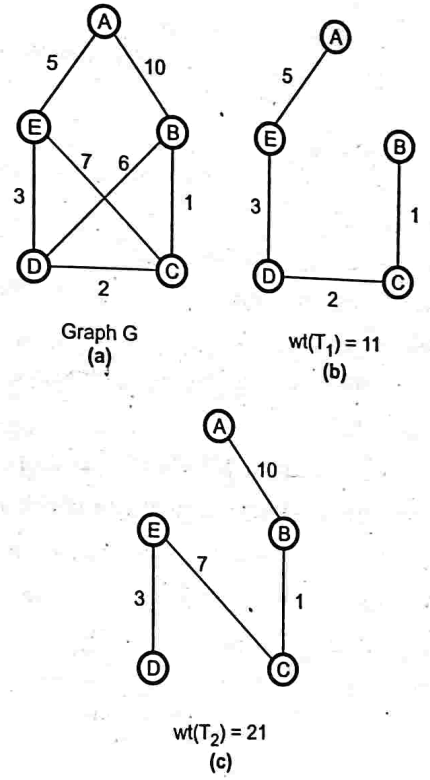


Fig. 6.3.1 Graph and two spanning trees out of which (b) is a minimum spanning tree

Distributed MST

- Gallagher-Humblet-Spira distributed MST uses Kruskal's strategy.
- This algorithm works in phases.
- After each phase, we have a spanning forest, in which each component tree has a leader.
- In each phase, each component finds Min Weight Outgoing Edge (MWOE), then components merge using all MWOEs to get components for next phase.

- It works as follows -
  - Each node is initially a component by itself(level 0 component)
  - **Phase 1 (produces Level 1 Component)**
    - Each node uses its min weight edge as the component MWOE.
    - Send connect message across MWOE.
    - There is a unique edge that is the MWOE of two components.
    - Leader of new component is higher-id endpoint of this unique edge.
  - **Phase K+1 (produces Level k+1 Component)**
    - Leader of each component initiates search for MWOE (broadcast initiate on tree edges).
    - Each node finds its MWOE :

    i) Send test on potential edges, wait for accept (different component) or reject (same component).

    ii) Test edges one at a time in order of weight.
    - Report to leader (convergecast report); remember direction of best edge.
    - Leader picks MWOE for fragment.
    - Send change-root to MWOE's endpoint, using remembered best edges.
    - Send connect across MWOE.
    - There is a unique edge that is the MWOE of two components.
    - Leader of new component is higher-id endpoint of this unique edge.

Analysis

The complexity analysis for

1. Messages Complexity : O(nlogn+|E|)
2. Time complexity O(nlogn)

Review Questions

1. *What is distributed algorithm ? Explain distributed breadth first algorithm with example*

   SPPU : May-18, Dec.-19, Marks 9, Dec.-18, Marks 7

2. *What is distributed algorithm ? Explain distributed minimum spanning tree.*

   SPPU : May-19, Marks 9

6.4 String Matching

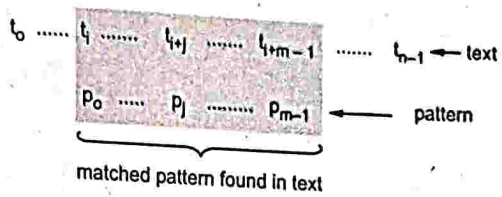
SPPU : May-18, Dec.-18, 19, Marks 10

- String matching algorithms are normally used in **text processing.**

- Normally text processing is done in compilation of program. In software design or in system design also text processing is a vital activity. And while processing the text, string matching is an important activity which is needed, most of the time.
- String matching means finding one or more generally all the occurrences of a string in the text. These occurrences are called as **pattern**. Hence sometimes string matching algorithms are also called as **pattern matching** algorithms.

Let, Text T is denoted by t_o .....t_{n-1} and

pattern P is denoted by p_o ....p_{m-1}



6.4.1 Naïve Algorithm

- This is the simplest method which works using **Brute Force** approach.
- The Brute Force is a straightforward approach of solving the problem. This method has "Just do it" approach.
- This algorithm performs a **checking at all positions** in the text between 0 to n–m, whether an occurrence of the pattern starts there or not.
- Then after each attempt, it shifts the pattern by exactly one position to the right.
- If the match is found then it returns otherwise the matching process is continued by shifting one character to the right.
- If there is no match at all in the text for the given pattern even then we have to do n comparisons. Let us understand this method with the help of some example -

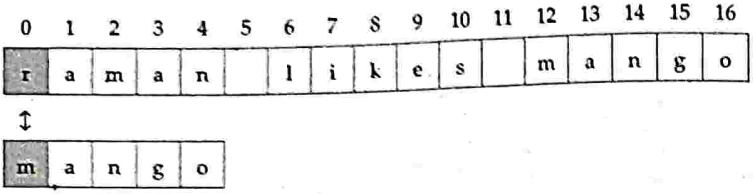
Example

Consider the text and pattern as given below.

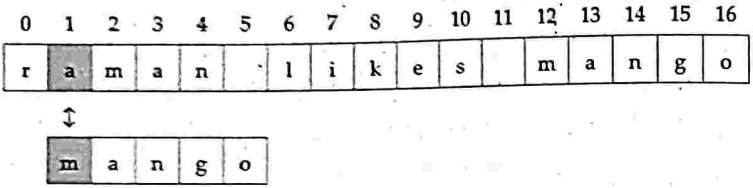
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	
r	a	m	a	n		l	i	k	e	s		m	a	n	g	o	Text

0	1	2	3	4	
m	a	n	g	o	Pattern

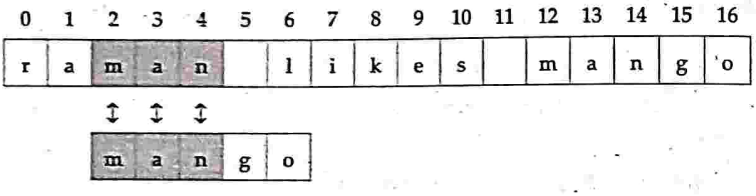
We will start finding match for pattern from 0^{th} location in Text. If the match is not found the shift to the right by 1 position.



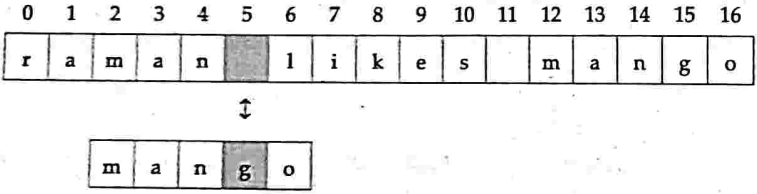
No match is found. ∴ Shift to the right by 1 position



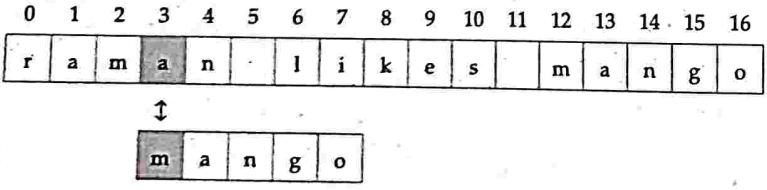
No match



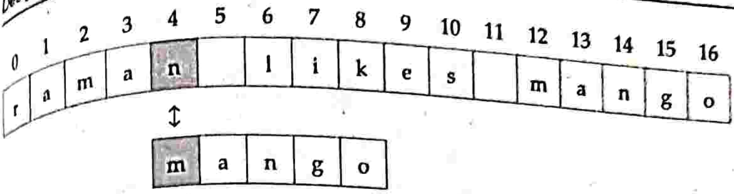
Three symbols are matching.



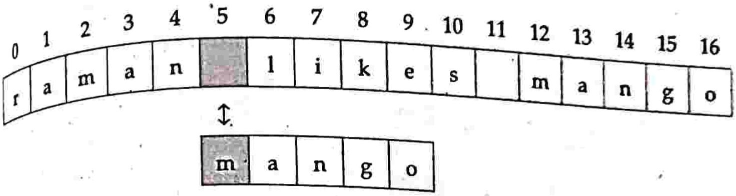
No match. ∴ Shift to the right by 1 position.



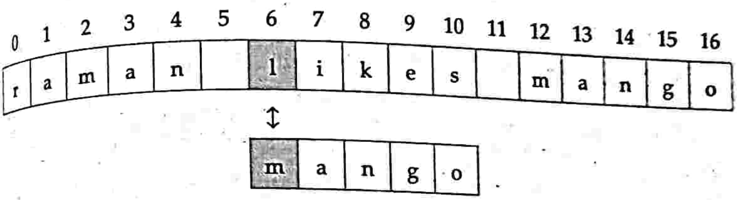
No match found.



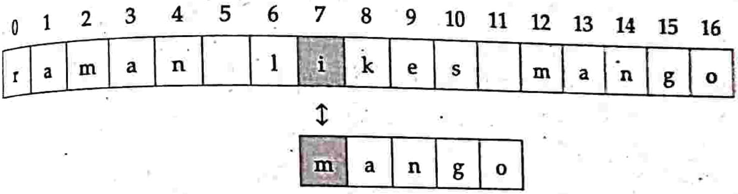
No match found.



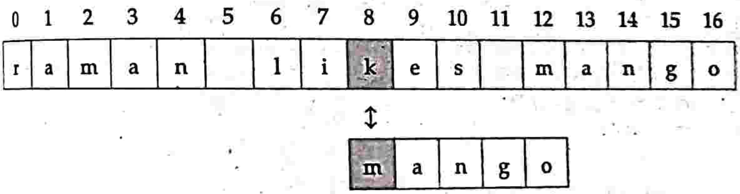
No match found.



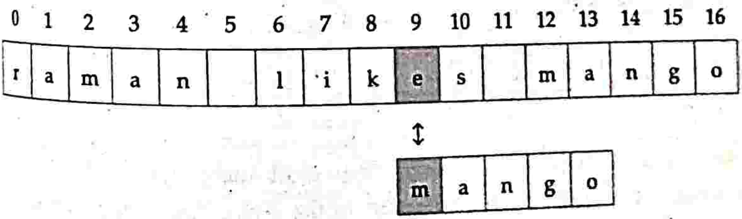
No match found.



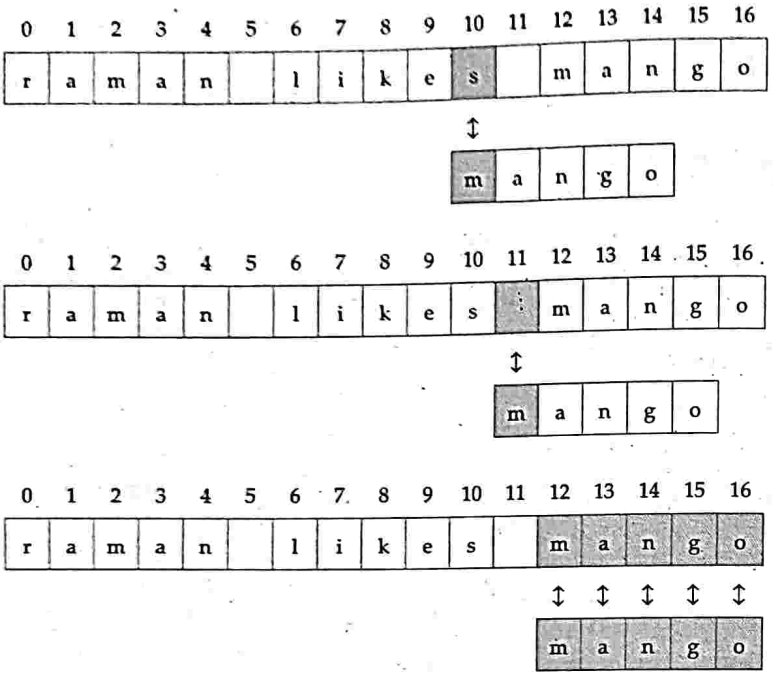
No match found.



No match found.



No match found.



Hence return index 12, because a match with the pattern is found from that location in **text.**

Algorithm

```
Algorithm Naive (T[1....n], P[1.....m])
{
// Problem Description : This algorithm finds
// the string matching using Naive method.
// Input : The array text T and pattern P
for (s ← 0 to n–m) do
{
      while (P[1.....m] = T[s+1.....s+m] then
      print ("pattern finding with shift", S);
}
} // end of algorithm
```

Analysis

In the given example, if a match is not found then shift the pattern to right by 1 position i.e. almost always we are shifting the pattern to the right. The worst case occurs when we have to make all the m comparisons. In above implementation, we can see that for loop is executed at the most $(n - m + 1)$ times and inner while loop executes for

m times. Hence running time in worst case is $O((n-m+1)m)$ i.e. $O(nm)$. For a typical word search in natural language the average case efficiency is $\Theta(n)$.

Example 6.4.1 *Show the comparisons the Naive-String matcher makes for the pattern P = {1001} in the text T = {0000100010010}*

Solution : This is the simplest method of string matching. The match for the pattern can be obtained as follows -

Step 1 :

	0	1	2	3	4	5	6	7	8	9	10	11	12
Text	0	0	0	0	1	0	0	0	1	0	0	1	0

↕ No match ∴ Shift the pattern to right by 1 place

Pattern	1	0	0	1

Step 2 :

	0	1	2	3	4	5	6	7	8	9	10	11	12
	0	0	0	0	1	0	0	0	1	0	0	1	0

↕ No match

	1	0	0	1

Step 3 :

	0	1	2	3	4	5	6	7	8	9	10	11	12
	0	0	0	0	1	0	0	0	1	0	0	1	0

↕ No match

	1	0	0	1

Step 4 :

	0	1	2	3	4	5	6	7	8	9	10	11	12
	0	0	0	0	1	0	0	0	1	0	0	1	0

↕ No match

	1	0	0	1

Step 5 :

0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	1	0	0	0	1	0	0	1	0

↕ ↕ ↕ Only three characters match. Hence shift the pattern to right

1	0	0	1

Step 6 :

0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	1	0	0	0	1	0	0	1	0

↕ No match

1	0	0	1

Step 7 :

0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	1	0	0	0	1	0	0	1	0

↕ No match

1	0	0	1

Step 8 :

0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	1	0	0	0	1	0	0	1	0

↕ No match

1	0	0	1

Step 9 :

0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	1	0	0	0	1	0	0	1	0

↕ ↕ ↕ ↕

1	0	0	1

Thus match is found at index 8 in text.

6.4.2 The Rabin Karp Algorithm

The Rabin-Karp method is based on hashing technique. This algorithm assumes each character to be a digit in radix-d notation. It makes use of equivalence of two numbers modulo a third number. Let, p[1....m] be a pattern then p denotes its corresponding decimal value. The d = 10 for decimal number, d = 2 for binary number.

Similarly,

Let, T [1....n] be a text then t_s denotes the decimal value of the substring T[s+1....s+m] for s= 0, 1, ... n – m.

The method follows following steps -

1. Compute p in O(m) time.
2. Compute all t_i values in total of O(n) time.
3. Find all valid shifts s in total of O(n) time.

The computation of p can be done using Horner's rule as follows.

$$p = p[m] + d(p[m-1] + d(p[m-2] + \ldots d(p[2] + d(p[1]))))$$

For example : To compute pattern

p[1...m] = 41603 we have m = 5 (i.e. length of pattern), d = 10. Then

$$p = p[m] + 10\,(p[m-1]) + 10^2(p[m-2] + \ldots\ldots 10^{m-1}(p[1])$$

$$= 3 + 10\,(0) + 100\,(6) + 1000\,(1) + 10000(4)$$

$$p = 41603$$

We can then compute to from T[1.....m] in O(m) time. Then remaining t_i can be computed in O(n–m) time as :

$$t_{s+1} = d(t_s - d^{m-1}\,T[s+1] + T[S+m+1])$$

where d = 10.

But these values of p and t_s may be too large hence we use mod value.

Example 6.4.2 *Using the Rabin - Karp algorithm for text T = 3141592653589793 find for pattern P = 26. With modulo q = 11*

Solution :

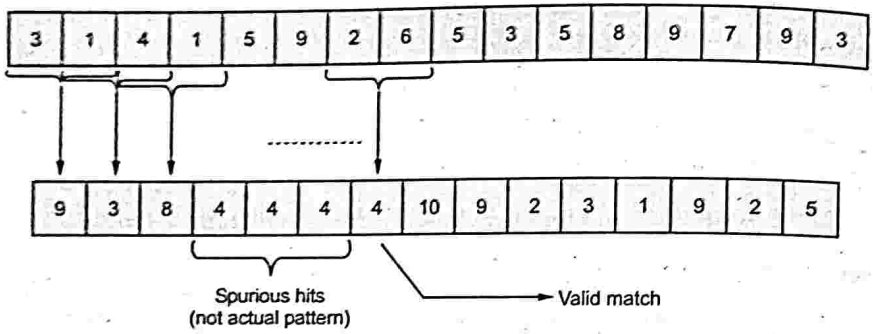
Text	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3

Pattern	2	6

The 26 mod 11 = 4

We will now obtain mod 11 value for each text.

Step 1 :



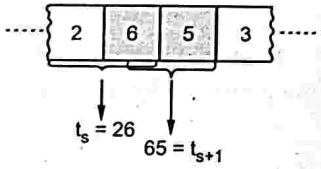
For m = 2, consider $t_s = 26$ then we can compute t_{s+1} as follows

t_{s+1} = 10 ($t_s - d_{10}^{m-1}$. old high order digit) + new low order digit

= $10(26 - 10 \cdot 2) + 5 = 10(26 - 20) + 5$

= 60 + 5

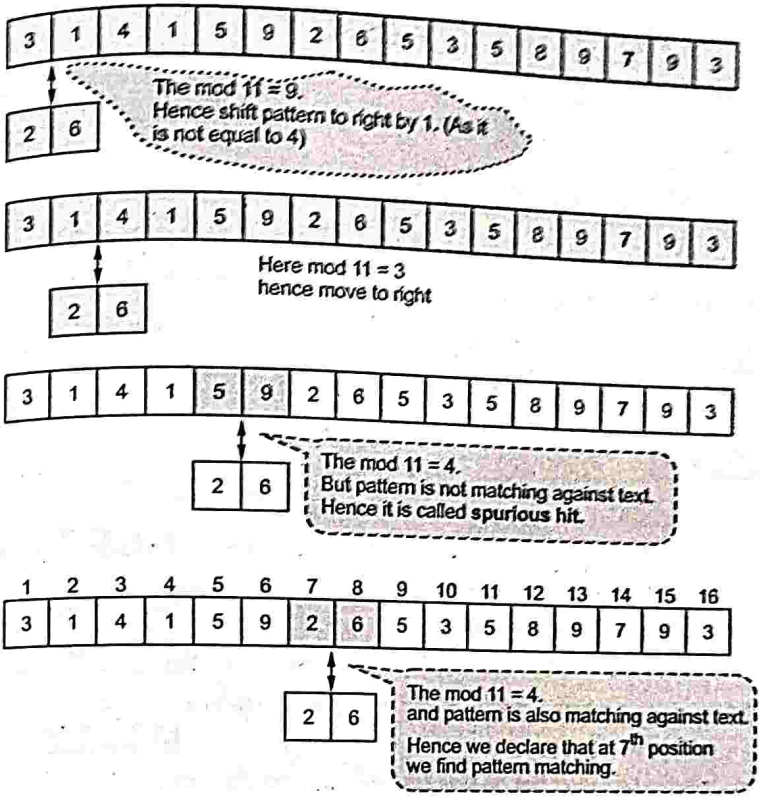
t_{s+1} = 65



Thus we can obtain every successive bit in text T.

According to Rabin-Karp method we start matching the pattern against text from beginning itself.

The algorithm is as follows -



Algorithm RabinKarp (T[1....n], P[1.....m], d, q]

{

// **Problem Description :** This algorithm is

// for matching the pattern using Rabin Karp method

h ← pow (d, m–1) **mod** q ← high order digit position

p ← 0

t_0 ← 0

for (i ← 1 **to** m)

{

p ← (dp + P[i]) **mod** q

t_0 ← (dt_0 + T[i] **mod** q } Θ (m) time

}

for (s← 0 to n – m) // with shift value each time

{ // incremented by one we

```
                                    // go on matching pattern
        if (p = t_s) then           // against text.
        {
            if (T[1.....m] = T [s+1 ....s+m] then
        write(" pattern found with shift," s)
}
        if (s<n – m)

        t_{s+1} ← (d (t_s – T[s+1]h) + T(s+m+1)mod q

} // end of for
} // end of algorithm
```

Yet end of text is not reached.

Computing t_{s+1}

The matching time required by above algorithm is Θ ((n – m+1)m). Hence worst case running time of this algorithm is Θ **((n – m+1)m).**

Review Questions

1. *Compare and contrast string matching algorithms. Explain any one algorithm with example.* **SPPU : May-18, Marks 9**
2. *Write naive string matching algorithm. What is the time complexity of algorithm ? Explain complexity of algorithm with example.* **SPPU : Dec.-18, Marks 9**
3. *Write and explain Rabin-Karp string matching algorithm.* **SPPU : Dec.-18, Marks 10**
4. *What are string matching algorithms ? Explain any one algorithm with example.* **SPPU : Dec.-19, Marks 9**
5. *Explain Rabin-Karp algorithm. Explain the worst case and best case running time of Rabin Karp algorithm.* **SPPU : Dec.-19, Marks 9**

❑❑❑