SPPU

UNIT 3

GREEDY AND DYNAMIC ALGORITHMIC STRATEGIES

content

- Introduction to greedy strategy
- Knapsack Problem and 0/1 knapsack
- Job scheduling
- Activity Selection
- Introduction to Dynamic programming
- OBST
- Chain matrix multiplication

GREEDY STRATEGY

- A greedy strategy is a problem-solving technique.
- At each step, it chooses the best immediate (local optimum) option.
- It does not look ahead or reconsider earlier choices.
- Goal: Reach a global optimum using only local decisions.
- Simple, fast, and often gives an optimal or near-optimal solution.

Knapsack Problem

- fractional knapsack

ITEM	WEIGHT	VALUE/PROFIT
A	10	6
B	20	5
C	30	4

Total : 240

Q.5 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]

- n is no. of items
- m is capacity of bag
- Applying Fractional Knapsack here:

A) METHOD 1 : selecting item 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 $\left(x2, \frac{x1 * 5}{18}\right)$

B) METHOD 2 : selecting item 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 $\left(x2, \frac{x1 * 5}{18}\right)$

C) METHOD 3 : selecting item with minimum value/weight ratio i. e P/W

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

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 maximum profit as **31.66.**

Job Scheduling

- Select jobs (each taking 1 unit time) such that they finish before their deadlines and maximize profit.
- Use a greedy strategy: sort jobs in decreasing profit and place each job in the latest free slot before its deadline.
- The final scheduled jobs give the highest possible total profit while satisfying all deadlines.

Q.8 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]

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]

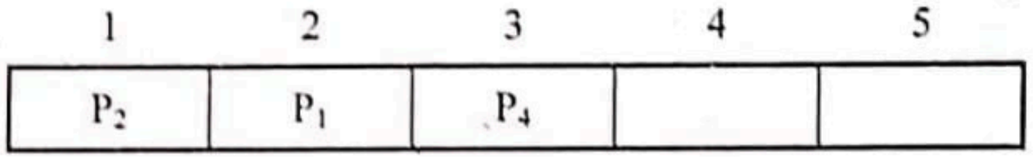
Step 3 : Add i^{th} Job in array J[ ] at index denoted by its deadline D_i.
First job is P_1, its deadline is 2.
Hence insert P_1 in the array J[ ] at 2^{nd} index.

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].



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**.

Activity Selection

- Select the maximum number of activities that do not overlap, where each activity has a start time and a finish time.
- Apply a greedy approach by sorting activities in increasing order of their finish times and always picking the activity that finishes earliest.
- This ensures the largest set of non-overlapping activities is chosen for maximum utilization of time.

Q.11 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).

Steps to follow

- **Step 1 : Sort the given activities in ascending order according to the 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.**

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 $\{I_2, I_4\}$.

Dynamic Programming

- Dynamic programming is a problem-solving technique used to efficiently solve problems that have overlapping subproblems and optimal substructure.
- Instead of solving the same subproblems again and again, dynamic programming stores the results of solved subproblems (usually in a table or array) and reuses them when needed.
- This reduces time complexity and makes many complex problems solvable in polynomial time.

Elements of dynamic programming

1) 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.

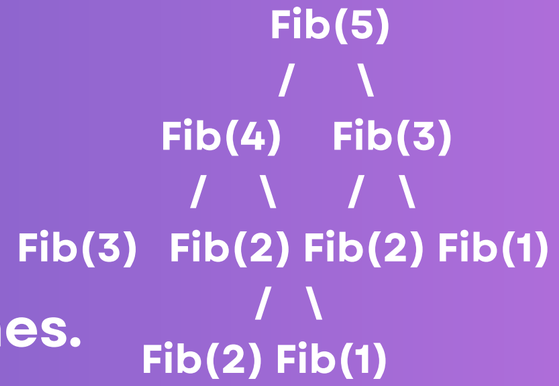
EXAMPLE:-

To calculate Fib(5):

Fib(5) = Fib(4) + Fib(3)

Fib(4) = Fib(3) + Fib(2)

Fib(3) = Fib(2) + Fib(1)



Here you can see: Fib(3) and Fib(2) are being repeated multiple times.

What is happening here?

- Fib(3) is calculated more than once
- Fib(2) is calculated 3–4 times

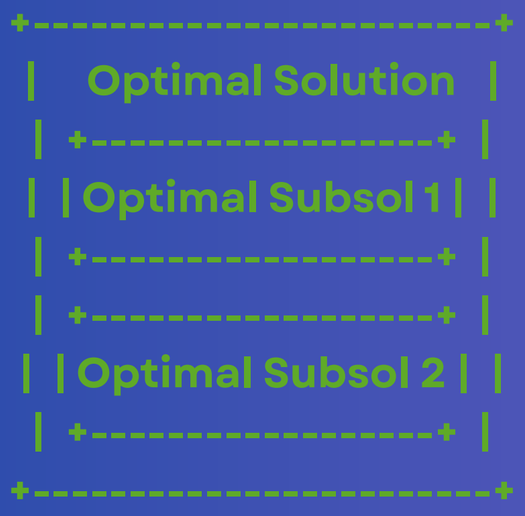
This repeated calculation of the same subproblems is called “overlapping subproblems.”

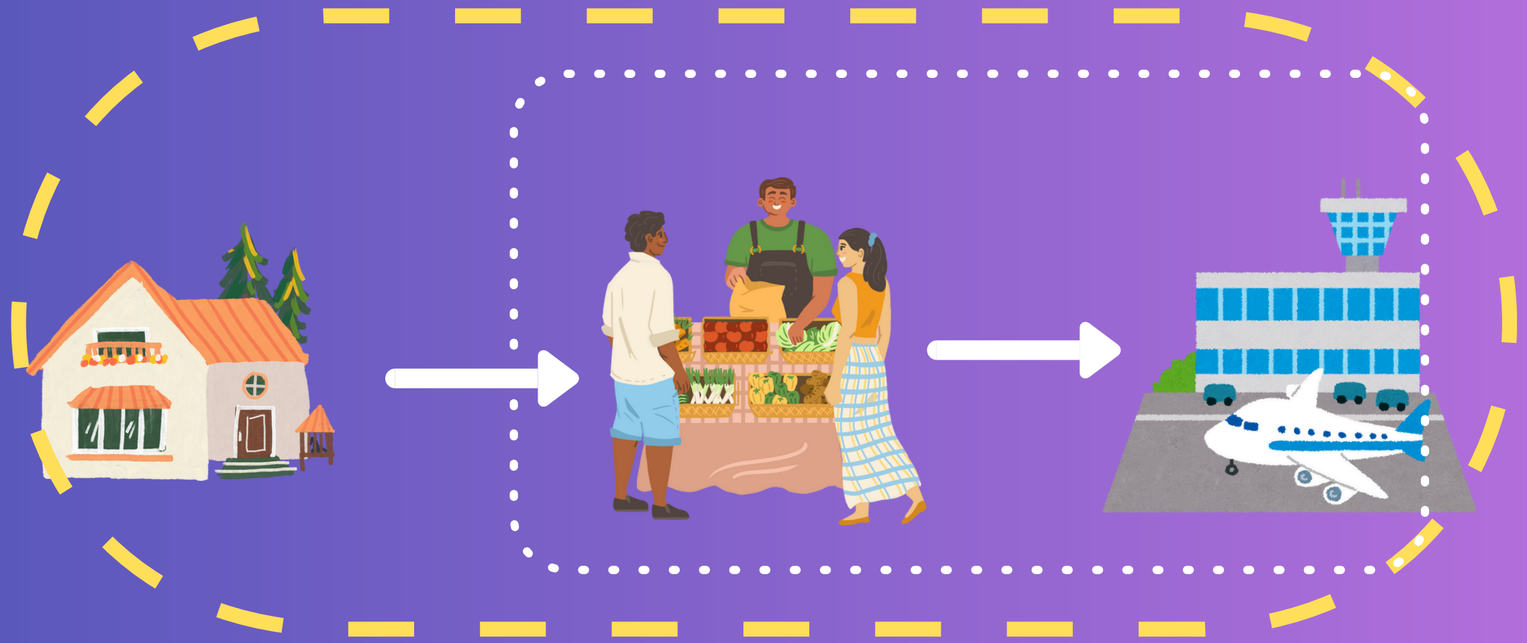
2) Optimal substructure -

- The dynamic programming technique which makes use of principle of optimality to find the optimal solution from subproblems.

Principal of optimality :-

The Principle of Optimality says:
If an optimal solution of a problem contains subproblems, then the subproblems must also be solved optimally.



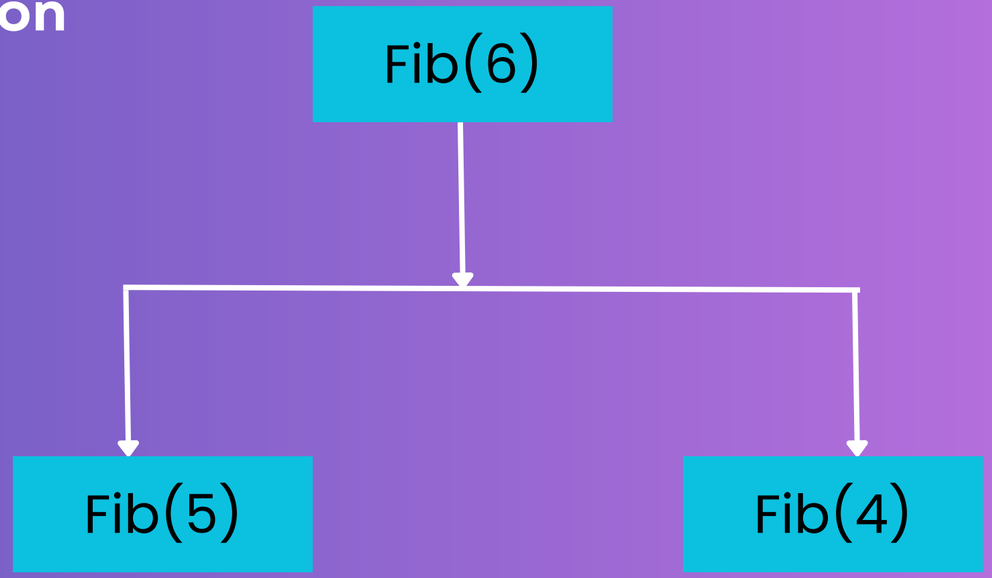


Steps of Dynamic Programming : -

1. Characterize the structure of the optimal solution
2. Recursively define the value of the optimal solution
3. Compute the optimal values using bottom-up DP table
4. Construct (rebuild) the optimal solution from the table

Example :-

Step 1: Characterize the Structure of the Optimal Solution
Goal: Express the problem in smaller subproblems.
For Fibonacci:
Fib(n) depends on Fib(n-1) and Fib(n-2)



Step 2: Recursively Define the Optimal Solution

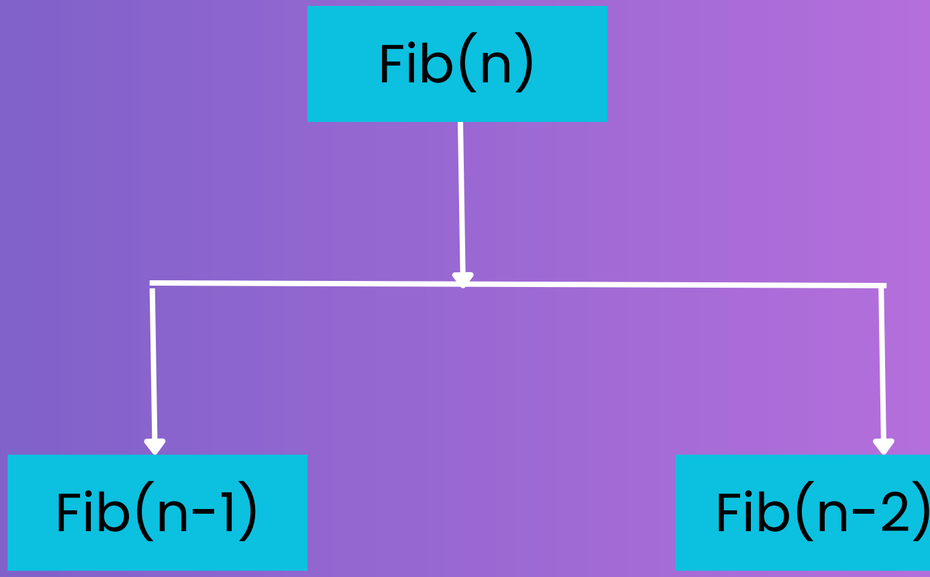
Write recurrence relation:

Fib(n) = Fib(n−1) + Fib(n−2)

Base cases:

Fib(0) = 0

Fib(1) = 1



Step 3: Compute Values Bottom-Up (DP Table)

DP Table:

0	1	1	2	3	5	8

Step 4: Construct the Optimal Solution

Final answer is taken from table:

Fib(6) = 8

DP Table: 0 1 1 2 3 5 8

↑

Take final value

0/1 Knapsack

- **Each item can only be chosen once.**
- **You cannot take a fraction of an item.**

For every item:

- **1 → take the item**
- **0 → do not take the item**

Example : -

Item	Weight	Value
1	10	60
2	20	100
3	30	120

Take (1, 2) → weight = 30, value = 160

Take (1, 3) → weight = 40, value = 180

Take (2, 3) → weight = 50, value = 220 ✓

Take (1,2,3) → weight = 60
✗ exceeds capacity

1. Create a DP Table

- Rows represent items (i = 0 … n)
- Columns represent knapsack capacity (w = 0 … W)
- table[i][w] stores maximum profit using first i items and capacity w

2. Initialization

- If capacity = 0 → profit = 0
- table[i][0] = 0
- If no items → profit = 0
- table[0][w] = 0

3. Recurrence Relation

Case 1: Item fits in the knapsack

If weight[i] ≤ w:

$$table[i][w] = \max\left(profit[i] + table[i-1][w - weight[i]],\ table[i-1][w]\right)$$

- *Include item* → `profit[i] + table[i-1][remaining capacity]`
- *Exclude item* → `table[i-1][w]`

Case 2: Item does NOT fit

If weight[i] > w:
table[i][w] = table[i-1][w]

4. Final Answer

table[n][W] gives the maximum achievable profit

Example : -

b) Consider the 0/1 Knapsack problem with the total weight capacity of 7. The weights and associated values (profits) of 4 items are given below. Use dynamic programming approach to find the optimal solution to this problem. **[10]**

Item	Weight	Value(Profit)
A	1	1
B	3	4
C	4	5
D	5	7

Step 1:-

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

table[i][0] = 0 & table[0][w] = 0 as capacity = 0.

Step 2:- table[1][1]

w = 1

w_i = 1

w = w_i — case 1

table[1][1] = max(profit[i] + table[i-1][w-weight[1]], table[i-1][w])

in short

table[1][1] = max(include item, exclude item) $\uparrow^{w-w_i}$

∴ include item → profit[i] + table[i-1][remaining capacity]

include item → profit[1] + table[1-1][1-1]

→ 1 + 0

→ 1

exclude → table[i-1][w]

table[1-1][1]

0[1]

→ 0

table[1][1] = max(1, 0)

table[1][1] = 1

Now,

table[1][2]

w = 2 w_i = 1

1 ≤ 2 can i

include → profit[1] + table[1-1][2-1]

1 + 0

1

exclude = 0

table[1][2] = max(1, 0) = 1

table[1][3]

w = 3 w_i = 1

1 ≤ 3

table[1][3] = 1

• table[1][4]

w = 4

w_i = 1

1 ≤ 4

include → 1

exclude → 0

table[1][4] = 1

• table[1][5]

w = 5 w_i = 1

1 ≤ 5

include → 1

exclude → 0

table [2] [1]

$wi = 3$ $i = 2$

$w = 1$

$3 > 1$ —— case 2

table [i] [w] = table [i-1] [w]

= table [2-1] [1]

= table [1] [1]

table [2] [1] = 1

table [2] [2]

$wi = 3$ $w = 2$

$3 > 2$

table [i] [w] = table [i-1] [w]

= table [2-1] [2]

+ table [1] [2]

table [2] [2] = 1

table [2] [3]

$wi = 3$

$w = 3$

Case 1

table [2] [3]

include = profit [2] + table [1] [0]

⇒ 4 + 0

⇒ 4

exclude ⇒ table [1] [3]

⇒ 1

max (4, 1) ⇒ 4

table [2] [3] = 4

final table

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

- for maximum profit

start from table [n][w]

here table [4][7] & check if table [n][w] = table [n-1][w]

∴ table [4][7] = table [3][7]

9 = 9

dont select item 4

set n = n-1 w = 7

table [3][7] = table [2][7]

9 ≠ 5

select item 3

~~now~~ ~~n = n-1~~ remaining capacity = w - wi;

→ 7 - 4

→ 3

w = 3

table [2][3] = table [1][3]

4 ≠ 1

select item 2

rc → 3 - 3

rc = 0

items included 3 & 2

value/profit → 4+5

value = 9

Chain Matrix Multiplication

- Matrix Chain Multiplication is an optimization problem.
- Goal: Find the most efficient way to multiply a given sequence of matrices.
- Parenthesization changes the number of operations, although the result remains the same.
- Order of multiplication affects time complexity, not the final product.

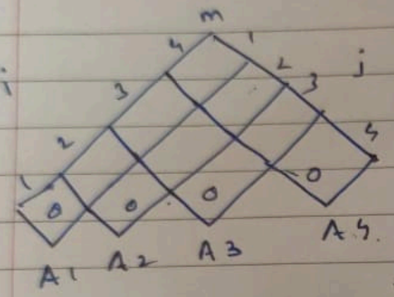
Example : -

Eg :-

Matrix	dimension
A1	5 × 4
A2	4 × 6
A3	6 × 2
A4	2 × 7

We have 4 matrices given with their respective dimensions
$P_0 = 5$, $P_1 = 4$, $P_2 = 6$, $P_3 = 2$, $P_4 = 7$

Initially :- $m[i,i] = 0$
$\therefore m[1,1], m[2,2], m[3,3], m[4,4] = 0$



Now, filling the table :-

here, $i = 1$ $j = 2$ $k = A_1 | A_2$
and $k \Rightarrow k = j - 1$
$k = 2 - 1$ $\boxed{k = 1}$ 1 → 2 only 1 gap

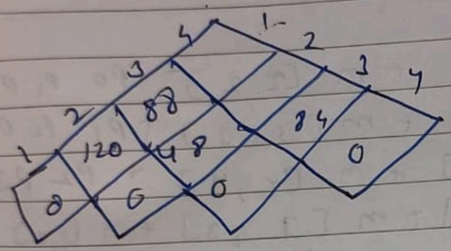
$m[1,2] \Rightarrow m[i,k] + m[k+1, j] + P_{i-1} P_k P_j$
$= m[1,1] + m[2,2] + P_0 P_1 P_2$
$= 0 + 0 + 5 \times 4 \times 6$
$m[1,2] = 120$

next $i = 2$ $j = 3$ $k = j - 1 \Rightarrow 3 - 1 = 2$ $A_1 | A_2 A_3$

$\therefore m[2,3] = m[i,k] + m[k+1, j] + P_{i-1} P_k P_j$
$= m[2,2] + m[3,3] P_1 P_2 P_3$
$= 0 + 0 + 4 \times 6 \times 2$
$m[2,3] = 48$

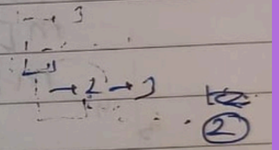
Now, $i = 3$ $j = 4$ $k = 3$
$m = 84$

$A_1 | A_2 A_3 A_4$ $k = 1$
$A_1 A_4 | A_3 A_4$ $k = 2$
$A_1 A_2 | A_3 A_4$ $k = 3$



Now, $i = 1$, $j = 3$ $k = 1$ or $k = 2$

$k = j - 1$
$k = 3 - 1$
$k = 2$



$m[1,3] = \min \begin{cases} [m[1,1] + m[2,3] + P_0 P_1 P_3) \rightarrow k = 1 \\ (m[1,2] + m[3,3] + P_0 P_2 P_3) \rightarrow k = 2 \end{cases}$

$\min \begin{cases} (0 + 48 + 5 \times 4 \times 2) \\ 120 + 0 + 5 \times 6 \times 2 \end{cases}$

$\min (88, 180) \longrightarrow k = 1$

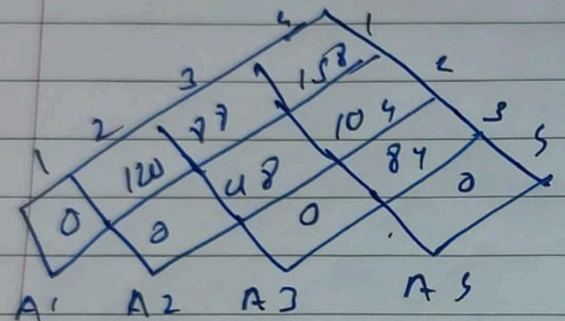
$m[1,3] = 88$

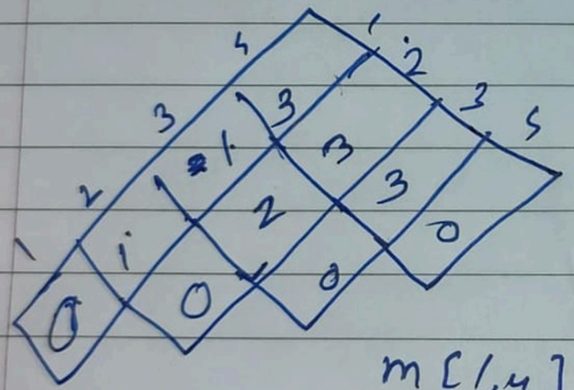
$m[2,4] \Rightarrow 104$

Now, $m[1,4] = \min \begin{cases} m[1,1] + m[2,4] + p_0 p_1 p_4 \rightarrow k=1 \\ m[1,2] + m[3,4] + p_0 p_2 p_4 \rightarrow k=2 \\ m[1,3] + m[4,4] + p_0 p_3 p_4 \rightarrow k=3 \end{cases}$

$\min \begin{cases} 0 + 104 + 5 \times 4 \times 7 \\ 120 + 84 + 5 \times 6 \times 7 \\ 88 + 0 + 5 \times 2 \times 7 \end{cases}$

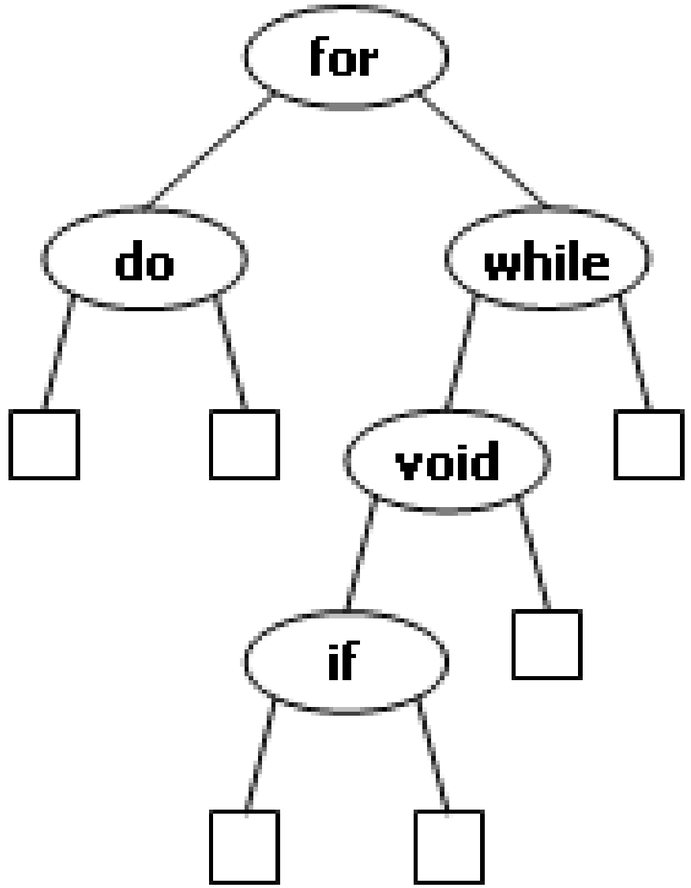
$\min \begin{cases} 244 \\ 414 \\ 158 \end{cases}$ → when $k=3$



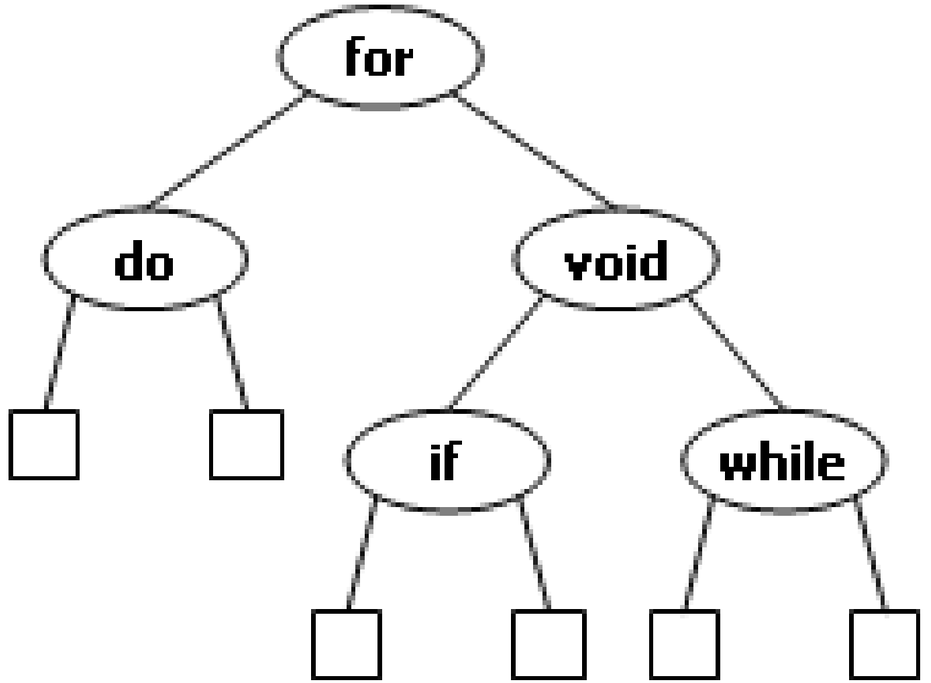


$m[1,4]$ determines optimal cost i.e 158

OBST(OPTIMAL BINARY SEARCH TREE)



(a)



(b)

Successful Search :- A successful search in an OBST means finding the required key in the tree, and its cost depends on the probability and depth of that key in the optimal binary search tree.

Denoted by : P ------->>>> 1<=i<=n

Unsuccessful Search : - An unsuccessful search in an OBST means the required key is not found in the tree, and the search stops at an external (dummy) node, whose cost depends on its probability and depth.

Denoted by : q ---------->>>> 0<=i<=n

Now , understand parameters to calculate optimal search tree

Weight(w) , Cost(c) , root(r)

To calculate the weight :- W(i,j) = p(j) + q(j) + w(i,j-1)

To calculate Cost

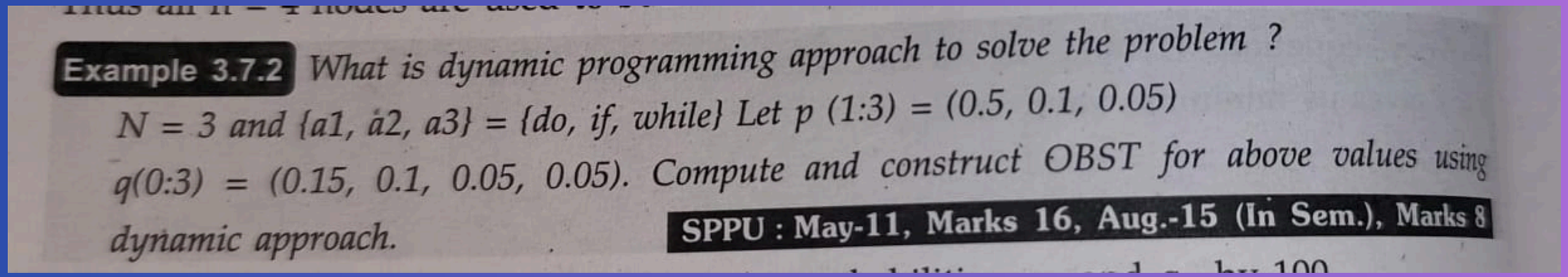
C(i , j) = min { c(i,k-1) + c(k,j)}+w(i,j) **---------->>>> i<k<=j**

After that solve row by row

Left node :- r(i,k-1)

right node :- r(k,j)

Example : -



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

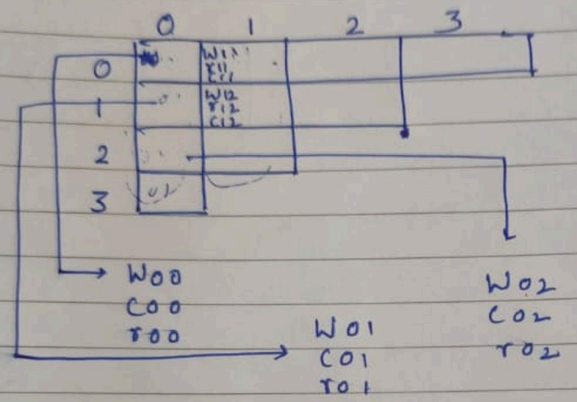
N=3

(a1, a2, a3) = {do, if, while}

(p1, p2, p3) = (0.5, 0.1, 0.05)

(q0, q1, q2, q3) = (0.15, 0.1, 0.05, 0.05)

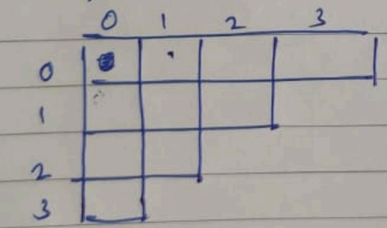
Now, constructing the cost table



- Calculating weight & cost

initially C(i,j)=0 ; r(i,i)=0 & w(i,i)=q(i)

so,



(i) $W_{00} = q(0) \rightarrow 0.15$

$C_{00} = 0$

$r_{00} = 0$

(ii) $W_{11} \rightarrow q(1) = 0.1$

$C_{11} = 0$

$r_{11} = 0$

(iii) $W_{22} = q(2) = 0.05$

$C_{22} = 0$

$r_{22} = 0$

(iv) $W_{33} = q(3) = 0.05$

$C_{33} = 0$

$r_{33} = 0$

b) $W_{01} = W(i,j) = p(j) + q(j) + w(i,j-1)$

$i=0$

$j=1$

$w(0,1) = p(1) + q(1) + w(0,0)$

$= 0.5 + 0.1 + 0.15$

$\boxed{w(0,1) = 0.75}$

$C(0,1) \rightarrow \min\{C(i,k-1) + C(k,j)\} + w(i,j)$

here $i < k <= j$

$0 < k <= 1$

$\boxed{k=1}$

$\min\{C(0,0) + C(1,1)\} + w(0,1)$

$\min\{0 + 0\} + 0.75$

$\boxed{C(0,1) = 0.75}$

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

$W_{12} = W(i,j) = p(j) + q(j) + w(i,j-1)$

$i=1 \quad j=2$

$p(2) + q(2) + w(1,1)$

$0.1 + 0.05 + 0.1$

$\boxed{W_{12} = 0.25}$

- $C(i,j) = \min\{C(i,k-1) + C(k,j)\} + w(i,j)$

$i=1$

$j=2 \quad \rightarrow \quad 1 < k <= 2$

$k=2$

$\min\{C(1,1) + C(2,2)\} + w(1,2)$

$\min\{0 + 0\} + 0.25$

$W_{23} \Rightarrow p(3) + q(3) + w(2,2)$

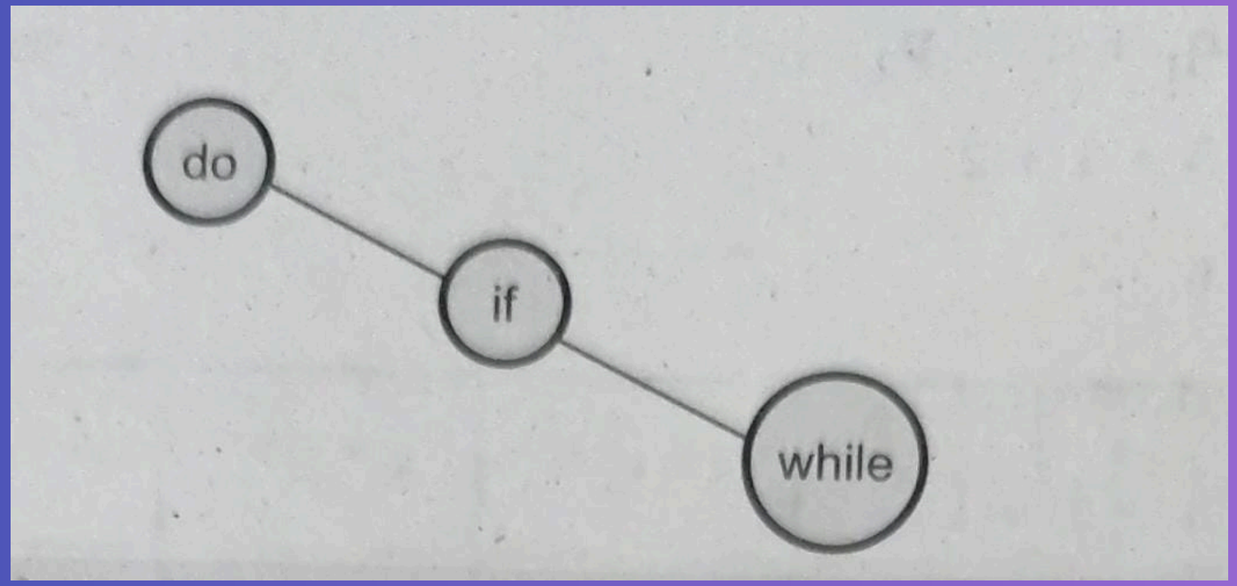
$0.05 + 0.05 + 5$ let's consider 0.05 & 0.05 as 5

$W_{23} = 5 + 5 + 5$

$W_{23} = 15$

$C_{23} = 15$

	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	$C_{03} = 115$ $r_{03} = 1$			



THANK YOU!