Unit-5 Dynamic Programming

DATE

5.1 Introduction

* **Concept of dynamic programming approach for algorithm design**

→ It is a design technique to solve optimization problems, where the goal is to find the best solution among the set of possible solutions.

→ It solves the problems by breaking down into smaller overlapping subproblems and storing the solutions to the subproblems in a table or array to avoid redundant computation.

→ Common examples of problems that can be solved using dynamic programming approach are: Knapsack problem, travelling salesman problem, longest common sequence

→ Dynamic programming works when a problem has following features

- **Optimal Substructure**: Ensures that an optimal solution to the problem can be constructed from optimal solutions to its subproblems.
- **Overlapping subproblem**: It is a problem that is a part of one or more larger problems, & the solution to the subproblem can be used to solve the larger problems. Ex, in the problem of finding nth fibonacci number, the solution to finding the $n-1^{th}$ & $n-2^{nd}$ fibonacci no. is used.

→ Without these properties, the problem can't be broken down into smaller subproblems & if it can't be broken down into smaller subproblems, it can't be solved using DP.

classmate

PAGE

DATE

Elements of Dp Strategy:

- **Substructure:** Divide the given problem into smaller subproblems & Express the solution of original problem in terms of the solution for smaller problem
- **Table Structure:** Choose the appropriate table structure to store the solutions of subproblems to avoid redundant computation
- **Bottom-up Computation:** Using table, combine the solutions of smaller sub-problems to solve larger sub problems and eventually solve the original problem

Greedy Algorithm vs Dynamic Programming

Greedy Algorithm	Dynamic programming
i) It is a method to solve optimization problems that involves making a locally optimal choice in order to achive the globally optimal solution Ex:- Fractional Knapsack	ii) It is a method that involves breaking down a problem into sub problems & storing the solutions to the subproblem in a table or array to achive global optimal. ex: 0/1 Knapsack
ii) It does not always guarantee to give an optimal solution.	iii) It always guarantees an optimal solution
iii) It doesn't require the problem to have overlapping subproblems but greedy choice & optimal substructure	iii) Requires the subproblem to have overlapping subproblem and optimal substructure
iv) Doesn't use memoization	iv) Uses memoization or tabulation to avoid redundant comp utation
v) More efficient & faster	v) Less efficient & slower

classmate

PAGE

DATE

* Recursion (divide & conquer) vs Dynamic programming

Recursion (D&C)	Dynamic programming
i) It is a method of solving a problem by breaking it down into subproblems & solving them recursively	i) It is a method of solving a problem by breaking it down into subproblems & storing the solutions to these subproblems to avoid redundant computations
ii) It does more work on sub-problems & hence has more time consumption	ii) It solves the sub-problems only once & then stores in the table
iii) Typically uses a stack to keep track of the recursive calls	iii) Typically uses a table or array to store the solutions to subproblems
iv) In this, sub-problems are independent of each other	iv) Sub problems are interdependent
v) Doesn't always guarantee an optimal solution	v) Always guarantee an optimal solution
vi) Suitable for problems that can be divided into similar sub problems	vi) Suitable for problems that have overlapping subproblems & optimal substructure
vii) Ex: Merge sort, Binary search, Factorial	vii) Ex: 0/1 Knapsack, Matrix multiplication

classmate

PAGE

DATE

Memoization Strategy

Memoization Strategy is a technique used to avoid the redundant computation in problems that have overlapping subproblems. It involves storing the solutions to subproblems in a table or array, so that they can be reused when the same subproblem is encountered again. This is often used in dynamic programming to improve the efficiency of algorithm

It is particularly useful when the number of subproblems is large and many subproblems have same solution, it can greatly reduce the time complexity of the algorithm

* Memoization vs dynamic programming

Dynamic Programming	Memoization
i) Method of solving a problem by breaking it down into smaller subproblems & storing the solutions to these subproblems to avoid redundant computation	i) Concept used in DP to solve the problems that have overlapping subproblems by storing the solutions to subproblems in a table or array.
ii) Always guarantees an optimal solution	iii) Improves the efficiency of the algorithm, but doesn't necessarily guarantee an optimal solution
iii) Typically used to solve optimization problems.	iii) Typically used in conjunction with dynamic programming & other algorithmic techniques as well
iv) It is bottom-up approach	iv) It is top-down approach

classmate

PAGE

DATE

5.2. * Dynamic Programming Algorithms

* Matrix Chain Multiplication:

It is a problem in which we are given a chain of matrices and the goal is to find out an efficient way to multiply them. Here, Chain means one matrix's column is equal to the second matrix's row. This algorithm doesn't perform the multiplications but just determines the best order in which to perform the multiplications

Recursive definition for optimal solution

$$m[i,j] = \begin{cases} 0 & \text{if } i=j \text{ (if sequence contain only one matrix)} \\ \min\{ m[i,k] + m[k+1, j] + p_{i-1} p_k p_j \} & \text{if } i<j \text{ and } i \le k < j \end{cases}$$

Ex: Consider matrices A, B, C, D of order 2x1, 1x3, 3x4, 4x5 respectively. Then find the optimal sequence for the computation of multiplication operation

⇒ Given, $A_{2\times1}$ $B_{1\times3}$ $C_{3\times4}$ $D_{4\times5}$
p_0 p_1 p_1 p_2 p_2 p_3 p_3 p_4

∴ $P = \{p_0, p_1, p_2, p_3, p_4\} = \{2, 1, 3, 4, 5\}$

M Table (consists of multiplications)

i \ j	1	2	3	4
1	0	6	20	42
2		0	12	32
3			0	60
4				0

K table (parenthesis)

i \ j	1	2	3	4
1		1	1	1
2			2	3
3				3
4				

classmate

PAGE

Here, $M[1,1]$ = A, we can split it in only one way. for $k=1$

$M[2,3] = A|B|C|D$ ...

DATE

$M[1,2] = m[1,1] + m[2,2] + p_0 p_1 p_2 = 0 + 0 + 2 \times 1 \times 3 = 6$

$M[2,3] = m[2,2] + m[3,3] + p_1 p_2 p_3 = 0 + 0 + 1 \times 3 \times 4 = 12$

$M[3,4] = m[3,3] + m[4,4] + p_2 p_3 p_4 = 0 + 0 + 3 \times 4 \times 5 = 60$

$1 \le k < 3$ ie $k = 1, 2$

$M[1,3] = \min \begin{cases} m[1,1] + m[2,3] + p_0 p_1 p_3 = 0 + 12 + 2 \times 1 \times 4 = 20 \\ m[1,2] + m[3,3] + p_0 p_2 p_3 = 6 + 0 + 24 + 2 \times 3 \times 4 = 30 \end{cases}$

1st is min so $k=1$

Now: How $M[1,3]$ ABCD

A(BC) (AB)C

$k=1$ $k=2$

Choose the minimum and put the value of the minimum's k in table ∴ 1st is min. so $k=1$

$2 \le k < 4$ ie. $k = 2, 3$

$M[2,4] = \min \begin{cases} m[2,2] + m[3,4] + p_1 \times p_2 \times p_4 = 0 + 60 + 1 \times 3 \times 5 = 75 \\ m[2,3] + m[4,4] + p_1 \times p_3 \times p_4 = 32 \end{cases}$

A BCD

A(BCD) (ABC)D

$k=1$ $k=3$

$\min = 32$, So, $k = 3$

$1 \le k < 4$ ie $k = 1, 2, 3$

$M[1,4]$

ABCD

A(BCD) (AB)(CD) (ABC)D

$k=1$ $k=2$ $k=3$

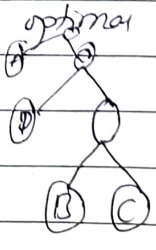
$m[1,1] + m[2,4] + 2 \times 1 \times 5 = 42$

$m[1,2] + m[3,4] + 2 \times 3 \times 5 = 96$

$m[1,3] + m[4,4] + 2 \times 4 \times 5 = 60$

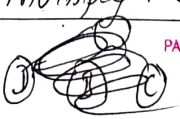
$\min = 42$ So $k = 1$

Now, the optimal multiplication cost = 42 with the optimal sequence:



$A(BC)D) \Rightarrow A((BC)D) \Rightarrow$

This means, 1st multiply B, C & then multiply the result with D, then multiply the result with A.



PAGE

Algorithm: (book)

```
MatrixChainMult(p)
{
n= length[p]
for(i=1 ; i<n; i++)
    m[i,i]=0

for(l=2; l<n; l++)
  {
  for(i=1; i<n-l+1 ; i++)
    {
    j= i+l-1
   m[i,j]=∞
   for(k=i; k<=j-1; k++)
    {
   c= m[i,k]+m[k+1,j]+p[i-1]*p[k]*p[j]
   If c<m[i,j]
     {
     m[i,j] = c
      s[i,j]=k
     }
   } } }
   Return m & s }
```

Anaysis: Since there are three nested loops

$\therefore O(n) = n^3$

& The space complexity is $O(n^2)$.

DATE

* String editing

Dynamic programming can also be used to solve string editing problems, (also known as "edit distance" problem) which calculates the minimum number of single character edits (insertion, deletion, or substitutions) required to change one string to another.

We are given two strings of symbols from finite set of alphabet $X = x_1 x_2 x_3 \ldots x_n$

$Y = y_1 y_2 y_3 \ldots y_n$

The task is to transform X onto Y using a sequence of edit operations. The performed operations are:

insert (y_j): insert symbol $y_j \in Y$ into the j^{th} position of X.

delete (x_i): delete a symbol x_i from X.

change (x_i, y_j): change a symbol $x_i \in X$ by symbol $y_j \in Y$

There is a cost $I(y_j)$, $D(x_i)$, $C(x_i, y_j)$ associated with each operation insert, delete, & change respectively.

The problem is to identify a minimum-cost sequence of edit operations that transforms X. We can define the solution recursively as:

$$Cost(i,j) = \begin{cases} 0 \text{ if } i=j=0 \\ cost(i-1,0) + D(x_i), \text{ if } i>0, j=0 \\ cost(i,j-1) + I(y_j), \text{ if } i=0, j>0 \\ \min\{ cost(i-1,j) + D(x_i),\ cost(i,j-1) + I(y_j),\ cost(i-1,j-1) + C(x_i, y_j)\}, \text{ if } i>0, j>0 \end{cases}$$

DATE

Algorithm

```
StringEdit(string1, string2)
  {
    m = length of string1
    n = length of string2
create a 2D array of size (m+1) x (n+1)
  initialize dp[i][0] = i  for i=0 to m
  initialize dp[0][j] = j  for j=0 to n

for i=1 to m
  for j=1 to n
      if string1[i-1] == string2[j-1]
          dp[i][j] = dp[i-1][j-1]

      else
          dp[i][j] = 1 + min( dp[i-1][j], dp[j][j-1],
                              dp[i-1][j-1] )

return dp[m][n]
      }
```

Analysis:

Time complexity is O(m x n)
It uses nested loops so.
And space complexity is O(m x n) for 2D array of size (m+1)(n+1)

PAGE

DATE

0-1 Knapsack problem and its Complexity analysis

Definition- Same as previous ones:

There are n items and the weight of ith item is w_i and its profit is p_i. An amount of item put into the bag is 0 or 1 i.e x_i is either 0 or 1. Here, the objective is to collect the items that maximize the total profit earned

Let, W = capacity of Knapsack
n = No. of items/objects.
$w = \{w_1, w_2, w_3, \ldots, w_n\}$ = weights of items
$p = (p_1, p_2, p_3, \ldots, p_n)$ = value/profit of items
$C[i, W]$ = maximum profit earned with item i & with knapsack of capacity W.

Then, the recurrance relation for 0/1 knapsack is:

$$C[i, W] = \begin{cases} 0 & \text{if } i=0 \text{ and } W=0 \\ C[i-1, W] & \text{if } w_i > W \\ \max\{p_i + C[i-1, W-w_i]\}, C[i-1, W]\} & \text{if } i>0 \text{ and } w_i \le W \end{cases}$$

Here, 1st condition is when there are no items and knapsack capacity is 0

2nd condition is when weight of an object i is greater than ks capacity. then, we discard the object

3rd condition is when weight of an object ≤ ks capacity. Then we either put object and reduce weight capacity by the objects object & remove the object from the objects list so (i-1). &then add the profit of the object or we discard the object. & see the profit.

The one with max is chosen

PAGE

Ex: Let there are three item of weight & values:

W = 7, Items = {i1, i2, i3, i4}

Profit = {1, 4, 5, 7}

weight = {1, 3, 4, 5}

⇒ Construct a table of (n+1)(W+1) = 5 × 8.

	Wi	P	W=0	W=1	W=2	W=3	W=4	W=5	W=6	W=7
i=0	0	0	0	0	0	0	0	0	0	0
i=1	1	1	0	1	1	1	1	1	1	1
i=2	3	4	0	1	1	4	5	5	5	5
i=3	4	5	0	1	1	4	5	6	6	9
i=4	5	7	0	1	1	4	5	7	8	9

∴ n+1 = 5

For, C[1,0] = 0 Since, W = 0

For, C[1,1] = Max{ p_i + C[i-1, W-w_i], C[i-1, W] }

= Max{ p_1 + C[0,0]; C[0,1] }

= Max{ 1+0, 0 }

= 1

For C[1,2] = Max{ p_1 + C[0,1], C[0,2] }

= Max{ 1, 0 } = 1

For, C[2,1] = ~~Max~~ { C[1,1] = 1 Since w_i > W

For C[2,2] = C[1,2] = 1

For C[2,3] = Max{ C[1,0] + 4, C[1,3] } = 4

For C[2,4] = Max{ C[1,1] + 4, C[1,4] } = 5

⋮

So on

∴ Max profit = 9 &

I_1	I_2	I_3	I_4
0	1	1	0

DATE

Algorithm

```
DynaKnapsack(W, n, V, w)
{
  for(w=0; w ≤ W; w++)
      C[0,w] = 0;
  for(i=1; i ≤ n; i++)
      C[i,0] = 0
  for(i=1; i ≤ n; i++)
  {
    for(w=1; w ≤ W; w++)
    {
      if(w[i] < w)
      {
        if V[i] + C[i-1, w-w[i]] > C[i-1, w]
            C[i,w] = V[i] + C[i-1, w-w[i]]
        else
            C[i, w] = C[i-1, w];
      }
      else
          C[i, w] = C[i-1, w];
    }
  }
}
```

Analysis: For run time analysis, examining the above algorithm, the overall runtime is $O(nW)$

where, n is the no of objects
& W is the knapsack capacity

DATE

* Floyd warshall Algorithm

It is an algorithm for finding the shortest paths in a weighted graph with positive or negative weights. It is used to solve the all pairs shortest path problem in a graph, in which the shortest paths between every pair of vertices need to be determined.

Consider a weighted graph $G = (V, E)$, connecting vertices i and j by w_{ij}. Let D^k be a $n \times n$ matrix denoting a weight matrix of shortest paths i to j. From i to j. Then, we have computing path containing two case

i) $D^k(i,j) = D^{k-1}(i,j)$ when k is not an intermediate vertex

ii) $D^k(i,j) = D^{k-1}(i,k) + D^{k-1}(k,j)$ when k is an intermediate vertex

$\therefore D^k(i,j) = \min\{D^{k-1}(i,j), D^{k-1}(i,k) + D^{k-1}(k,j)\}$

Ex: Find all pair shortest path using Floyd warshall

⇒ Weight matrix of given graphs:

$$D^0 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & \infty & 2 \\ 2 & 5 & 0 & 6 & \infty \\ 3 & \infty & \infty & 0 & 4 \\ 4 & \infty & -1 & -3 & 0 \end{array}$$

Now, Let's choose vertex 1

$$D^1 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & \infty & 2 \\ 2 & 5 & 0 & 6 & 7 \\ 3 & \infty & \infty & 0 & 4 \\ 4 & \infty & -1 & -3 & 0 \end{array}$$

Let's choose vertex 2

$$D^2 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & 9 & 2 \\ 2 & 5 & 0 & 6 & 7 \\ 3 & \infty & \infty & 0 & 4 \\ 4 & 4 & -1 & -3 & 0 \end{array}$$

Let's choose vertex 3

$$D^3 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ 1 & 0 & 3 & 9 & 2 \\ 2 & 5 & 0 & 6 & 7 \\ 3 & \infty & \infty & 0 & 4 \\ 4 & 4 & -1 & -3 & 0 \end{array}$$

Let's choose vertex 4

$$D^4 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ 1 & 0 & 1 & -1 & 2 \\ 2 & 5 & 0 & 4 & 7 \\ 3 & 8 & 3 & 0 & 4 \\ 4 & 4 & -1 & -3 & 0 \end{array}$$

PAGE

DATE

Now, let's choose vertex 1 as intermediate vertex. Then

$$D^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 2 \\ 5 & 0 & 6 & 7 \\ \infty & \infty & 0 & 4 \\ \infty & -1 & -3 & 0 \end{bmatrix} \end{array}$$

Now for Ex: (2,3) use (2,1) + (1,3) of D^0.

i.e. $D^1(2,3) = \min\{D^0(2,3), D^0(2,1) + D^0(1,3)\}$

Let's choose vertex 2 as intermediate vertex Then

$$D^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 9 & 2 \\ 5 & 0 & 6 & 7 \\ \infty & \infty & 0 & 4 \\ 4 & -1 & -3 & 0 \end{bmatrix} \end{array}$$

Let's choose vertex 3 as intermediate vertex

$$D^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 9 & 2 \\ 5 & 0 & 6 & 7 \\ \infty & \infty & 0 & 4 \\ 4 & -1 & -3 & 0 \end{bmatrix} \end{array}$$

Let's choose vertex 4 as intermediate node

$$D^4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & -1 & 2 \\ 5 & 0 & 4 & 7 \\ 8 & 3 & 0 & 4 \\ 4 & -1 & -3 & 0 \end{bmatrix} \end{array}$$

∴ Shortest path (weights) from

1 to 1 = 0

1 to 2 = 1

1 to 3 = -1

⋮

3 to 4 = 4

⋮

4 to 4 = 0

classmate

PAGE

Algorithm:

```
Floydwarshall(W)        // W: weight matrix (D^0)
{
  n = W.rows            // n represents no. of vertices which is equal
  D^0 = W               // to rows of matrix since it is n x n
for k = 1 to n          // D^0, D^1, D^2, D^3, D^4 ie. D^k, k = 1 to 4
{ let D^k = d^k_ij be a new matrix // i = row value, j = column value
  for i = 1 to n        // i = no. of rows
    for j = 1 to n      // j = no. of columns.
      d^k_ij = min{ d^(k-1)_ij , d^(k-1)_ik + d^(k-1)_kj }
  }
return D^n
}
```

Time complexity: since there are three nested loops each having O(n) time complexity

$\therefore T(n) = O(n^3)$

where n = no. of vertices or rows of weight matrix

DATE

Travelling salesman problem

It is a classical optimization problem in computer science. It is defined as follows:

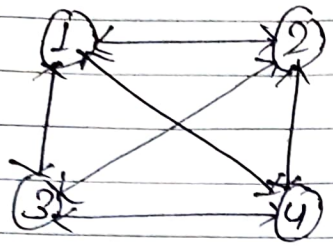
Given a set of cities and the distance between each pair of cities, find the shortest possible route that visits each city exactly once and returns to the starting city. It is an NP hard problem, which means there is no known efficient algorithm for solving it for large instances. However, there are several approximate algorithms that can be used to find near-optimal solutions such as branch and bound, heuristics, etc.

There is a non-negative cost $C(i,j)$ to travel from city i to city j. It can be solved using dynamic programming.

Let i be the starting vertex, S be the set of remaining vertices except vertex i. Let k be the any one vertex of S and $g(i,S)$ be the minimum travelling cost of TSP. Then their recurrance relation is

$$g(i,S) = \text{Min}_{k \in S}\{C(i,k) + g(k, S-\{k\})\}$$

Ex: Let's take a weighted graph (directed)



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

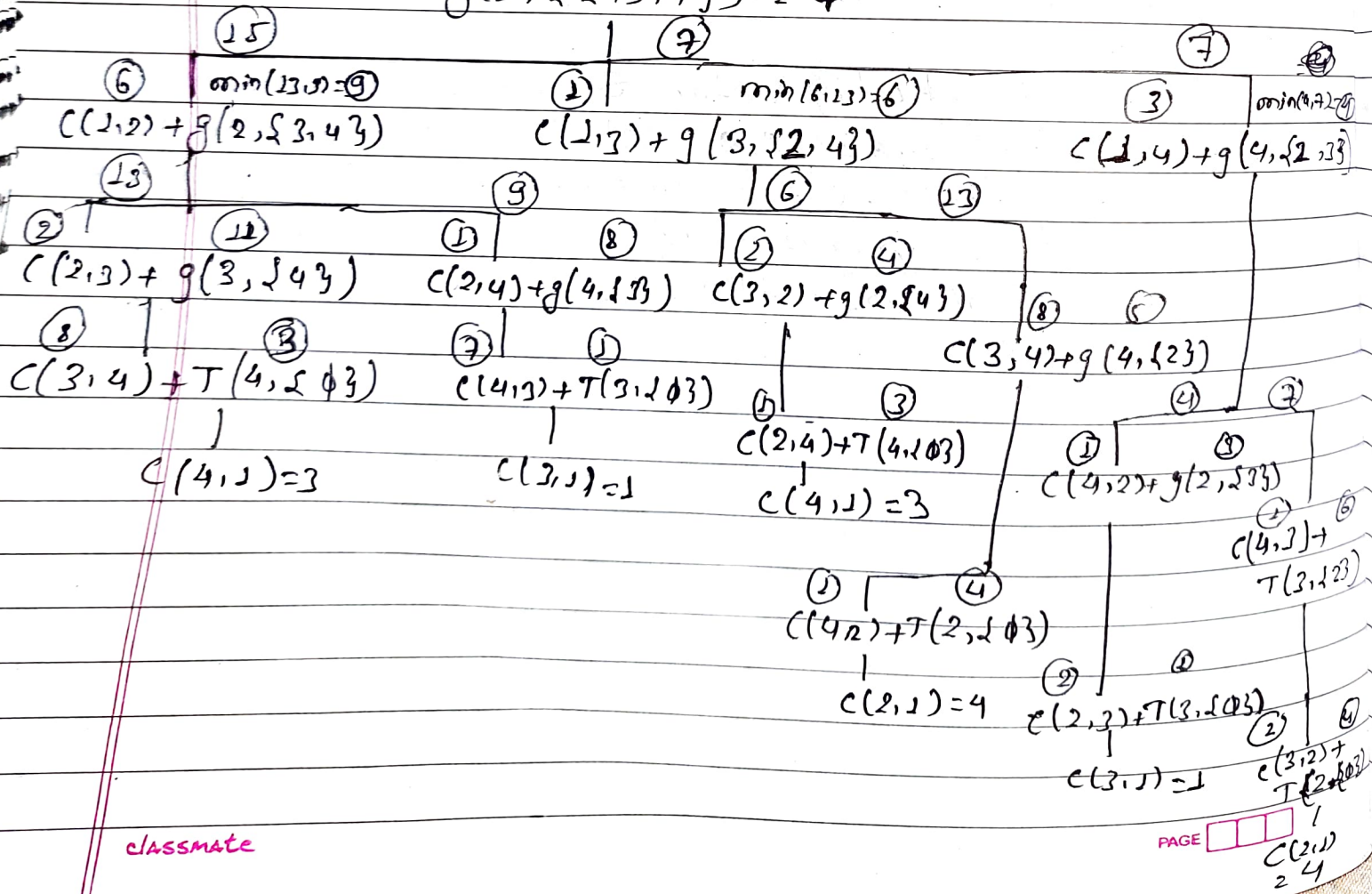
⇒ The given formula can be modified as

$$g(1,\{2,3,4\}) = \min_{k\in\{2,3,4\}} \{c(1,k) + g(k,\{2,3,4\} - \{k\})\}$$

Ex: $g(1,\{2,3,4\})$

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

$g(1,\{2,3,4\}) = 7$



DATE

Here, minimum cost = 7

And the path is

$1 \to 3 \to 2 \to 4 \to 1$

or

$1 \to 4 \to 2 \to 3 \to 1$

Analysis : Total number of distinct function calls : 2^{n-2}

$\therefore T(n) = O(2^{n-2})$

$= O(2^n)$.

$\therefore T(n) = O(n \cdot 2^n)$

when using dynamic programming

But when brute force method $T(n) = O(n^n) = O(n!)$