UNIT-5

Dynamic Programming:

Concept Imp!

Dynamic Programming is the most powerful design technique used to solve optimization problems. Dynamic Programming is used when the sub-problems are dependent, e.g, when they share the same sub-problems. Dynamic programming works when a problem has following features:

i) Optimal Substructure: If an optional solution contains optimal sub solutions then a problem shows optimal substructure.

ii) Overlapping sub-problems: When a recursive algorithm would visit the same sub-problems repeatedly, then a problem has overlapping sub-problems.

⊛. Greedy Algorithm vs. Dynamic Programming:

Dynamic Programming	Greedy Method
i) Dynamic Programming is used to obtain the optimal solution, which is guaranteed.	i) Greedy Method is also used to obtain the optimal solution, which has no guarantee.
ii) It requires DP table for memorization and it increases its memory complexity.	ii) It does not require DP table for memorization as it never revise previous choices.
iii) It is less efficient as compared to greedy approach.	iii) It is more efficient as compared to dynamic approach.
iv) In dynamic programming, we choose at each step, but the choice may depend on the solutions to sub-problems.	iv) In a greedy algorithm, we make whatever choice seems best at the moment and then solve the sub-problems arising after the choice is made.
v) Example: Knapsack problem (0-1)	v) Example: Fractional Knapsack

⊛ Divide and Conquer Algorithm vs. Dynamic Programming:

OR

Recursion vs. Dynamic Programming: [Imp]

Divide and Conquer Method	Dynamic Programming
i) It is recursive.	i) It is non recursive.
ii) It does more work on sub-problems and hence has more time consumption.	ii) It solves sub-problems only once and then stores in the table.
iii) It is a top-down approach.	iii) It is a bottom-up approach.
iv) In this sub-problems are independent of each other.	iv) In this sub-problems are interdependent.
v) For Example: Merge Sort & Binary Search.	v) For Example: Matrix Multiplication

⊛ Elements of Dynamic Programming Approach: [Imp]

There are basically three elements that characterize a dynamic programming algorithm.

i) **Substructure:** Decompose the given problem into smaller sub-problems. Express the solution of the original problem in terms of the solution for smaller problems.

ii) **Table Structure:** After solving sub-problems, store the results to the sub-problems in a table.

iii) **Bottom-up Computation:** Using table, combine the solution of smaller sub-problems to solve larger sub-problems and eventually arrives at a solution to complete problem.

⊛ Matrix Chain Multiplication: [Imp]

It is a dynamic programming approach in which previous output is taken as input for next. Here, chain means one matrix's column is equal to the second matrix's row. This algorithm does not perform the multiplications, but just determines the best order in which to perform the multiplications.

Recursive definition of optimal solution:

Let $m[i,j]$ denotes minimum number of scalar multiplications needed to compute $A_{i \ldots j}$.

$$m[i,j] = \begin{cases} 0 & \text{if } i=j \text{ (if sequence contain only one matrix)} \\ \min_{i \le k < j} \{ m[i,k] + m[k+1,j] + P_{i-1} P_k P_j \} & \text{if } i<j \end{cases}$$

Example: Consider matrices A1, A2, A3 and A4 of order 3×4, 4×5, 5×2 and 2×3. Then find the optimal sequence for the computation of multiplication operation.

Solution:

Here, $P = [3, 4, 5, 2, 3]$ ← (A1 to A3 सुरुको first i value र last A4 को i र j दुवै value i.e, $P_0=3$, $P_1=4$, ... $P_4=3$)

M Table (Cost of multiplication)

i \ j	1	2	3	4
1	0	60	64	82
2		0	40	90
3			0	30
4				0

S Table (points of parenthesis)

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

For $m[1,1] = m[2,2] = m[3,3] = m[4,4] = 0$ ← (Since 0 if $i=j$)

For $m[1,2] = \min\{m[1,1] + m[2,2] + P_0 \times P_1 \times P_2\}$ ← (according to above formula for if $i<j$)

$= \min(0 + 0 + 3 \times 4 \times 5)$

$= 60$

Similarly for $m[2,3] = 40$ & $m[3,4] = 30$

(P_0, P_1, P_2 are dimensions $P_0 = A_1$, $P_2 = A_2$ and so on)

(Here P_4 will be 3)

(Solving sequence should be diagonal like [1,2], [2,3], [3,4] and then, [1,3], [2,4] and finally [1,4])

(Values in S Table denote upon which value of k we got minimum value. For e.g. For [1,4] we will get min value 82 when k=3)

↔ (since k=1 & 2 here so, first do for 1 then 2 next time.)

$m[1,3] = \min\{\{m[1,1]+m[2,3]+p_0 \times p_1 \times p_3\}, \{m[1,2]+m[3,3]+p_0 \times p_2 \times p_3\}\}$

$= \min\{(0+40+3\times4\times2), (60+0+3\times5\times2)\}$

$= \min\{64, 90\}$

$= 64$

Similarly $m[2,4] = 90$ & $m[1,4] = 82$ → for this we will have k=1,2,3 from which we will select min value among three

Now, the optimal multiplication cost = 82 with the optimal sequence is: $(A1\ A2\ A3\ A4) \Rightarrow ((A1\ A2\ A3)(A4)) \Rightarrow ((A1)(A2\ A3)(A4))$. ← written based on S table

This means at first multiply matrix A2 and A3 then multiply their result with matrix A1 and finally multiply their result with A4.

See from top in S table (i.e, [1,4]) first 3 we see so first 3 digits in 1 & A4 final is seperate then we see [1,3] as 1 this means again seperate first digit A1. marked by tick (✓)

Algorithm:

```
MatrixChainMultiplication(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 and s
}
```

Analysis: The above algorithm can be easily analyzed for running time as $O(n^3)$, due to three nested loops, & the space complexity is $O(n^2)$.

⊛ String Editing Algorithm:

We are given two strings of symbols from finite set of alphabet.

$$X = x_1 x_2 x_3 \cdots x_n$$
$$Y = y_1 y_2 y_3 \cdots y_m$$

The task is to transform X into Y using a sequence of edit operations. The permitted operations are:-

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

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

change (x_i, y_j): change 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 and change respectively.

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

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

Example:

X = aabab
Y = babb

Costs:
Delete = 1, insert = 1, change = 2

		a (X1)	a (X2)	b (X3)	a (X4)	b (X5)
b	y1	0				
a	y2					
b	y3					
b	y4					

Fill the first row and first column applying base cases (first three cases).

Then, fill the other cells applying the 4th case of dynamic program. Note that if x_i, y_j are same symbols then the change cost is 0. After filling up the table, we can backtrack the table from last element to obtain the optimal sequence of operations.

⊛. 0-1 Knapsack problem:

A thief has a bag or knapsack that can contain maximum weight W of his loot. There are n items and the weight of ith item is w_i and it worth v_i. An amount of item can be put into the bag is 0 or 1 i.e x_i is 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
$W = \{w_1, w_2, \ldots w_n\}$ = weights of items
$V = \{v_1, v_2, \ldots v_n\}$ = value of items
C[i,w] = maximum profit earned with item i and with knapsack of capacity w.

Then the recurrence relation for 0/1 knapsack problem is given as;

$$C[i,w] = \begin{cases} 0 & \text{if } i=0 \text{ or } w=0 \\ C[i-1,w] & \text{if } w_i > w \\ \text{Max}\{v_i + C[i-1, w-w_i], C[i-1,w]\} & \text{if } i>0 \text{ and } w_i \le w \end{cases}$$

Example: Let the problem instance be with 7 items where
$V[] = \{2,3,3,4,4,5,7\}$
$W[] = \{3,5,7,4,3,9,2\}$
and W=9.
Then find maximum profit earned by using 0/1 knapsack problem.

Solution:
For i=0 or w=0
C[i,w] = 0

i.e, $C[0,1] = C[0,2] = C[0,3] = C[0,4] = C[0,5] = \ldots = C[1,0] = \ldots = 0$.

$C[1,1] = C[0,1] = 0$ since $w_1 > W$ i.e, 3>1 so it satisfied second case of the recurrence relation.

$C[1,3] = Max\{v_1 + C[0,3-3], C[0,3]\}$

$= Max\{2+0, 0\}$

$= 2$ since $W >= W_1$ i.e, $3 >= 3$ so it satisfied the third case.

Continue this process to calculate value of each cell and finally we get following table,

i \ W	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	2	2	2	2	2	2	2
2	0	0	0	2	2	3	3	3	5	5
3	0	0	0	2	2	3	3	3	5	5
4	0	0	0	2	4	4	4	6	6	7
5	0	0	0	4	4	4	6	8	8	8
6	0	0	0	4	4	4	6	8	8	8
7	0	0	7	7	7	11	11	11	13	15

Profit = $C[7][9] = 15$

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 run time of the algorithm is $O(nW)$.

⊛ Flyod Warshall Algorithms: [Imp] ↔ Concept, Algorithm & Analysis

This algorithm works even if some of the edges have negative weights. It is mainly used to find all pair shortest path of given weighted graph.

Consider a weighted graph $G=(V,E)$, connecting vertices i and j by w_{ij}. Let D^k denote an $n \times n$ matrix such that $D^k(i,j)$ is defined as the weight of shortest path from vertex i to j using only intermediate vertices from $1,2,\ldots,k$ as intermediate vertices in the a^{th}. Then, we have computing path containing two cases:

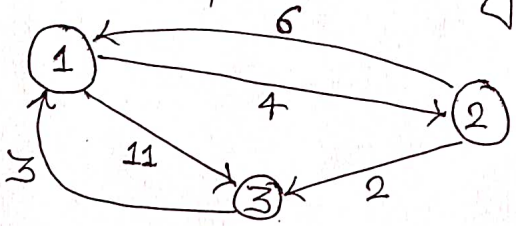
i) $D^k(i,j) = D^{k-1}(i,j)$ when k is not an intermediate vertex.

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

So from these relations we obtain;

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

Example: Find shortest path from every vertex to other vertices.



Solution:

Adjacency matrix (D^0 or W) of given graph is;

W or D^0	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

Case 1: When vertex (1) as intermediate vertex:

$D^1(1,1) = 0$

$D^1(1,2) = \min\{D^0(1,2), D^0(1,1) + D^0(1,2)\}$ = unchanged $= \min\{4, 0+4\} = 4$

$D^1(2,3) = \min\{D^0(2,3), D^0(2,1) + D^0(1,3)\}$ = may change $= \min\{2, 6+11\} = 2$

Similarly we can calculate all others.
Then the adjacency matrix can be modified as;

D^1	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

Case2: When vertex (2) as intermediate vertex:

$D^2(1,1) = 0$

$D^2(1,2)$ = unchanged $= 4$

$D^2(1,3) = \min\{D^1(1,3), D^1(1,2)+D^1(2,3)\}$ = may change $= \min\{11, 4+2\} = 6$

Similarly calculating others, adjacency matrix can be modified as;

D^2	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

Case3: When vertex (3) as intermediate vertex:

$D^3(1,1) = 0$

$D^3(1,2) = \min\{D^2(1,2), D^2(1,3)+D^2(3,2)\}$ = may change $= \min(4, 6+7) = 4$

$D^3(1,3)$ = unchanged $= 6$

Similarly calculating others, adjacency matrix can be modified as;

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

Thus the shortest path from 1 to 1 = 0
The shortest path from 1 to 2 = 4
The shortest path from 1 to 3 = 6
⋮
The shortest path from 3 to 3 = 0.

Algorithm:

```
FloydWarsal(W,D,n)
{ for (i=1; i<=n; i++)
  {
    for(j=1; j<=n; j++)
    { D[i][j]=W[i][j]; //Original D^0 matrix
    }
  }
  for (k=1; k<=n; k++)
  { for(i=1; i<=n; i++)
    {for (j=1 ; j<=n; j++)
      { if (D[i][j] >D[i][k]+D[k][j]) then
          Set, D[i][j]=D[i][k]+D[k][j]
      }
    }
  }
}
```

Analysis: The running time of the Floyd-Warshall algorithm is determined by triply nested for loops. Each loop executes at most $O(n)$ time. The algorithm thus runs in time $O(n^3)$.

⊛ Travelling Salesman Problem:

In this problem, a salesman must visit n cities. There is a non-negative cost $C(i,j)$ to travel from the city i to j. The goal is to find a tour of minimum cost.

Let i be the starting vertex, S be the set of remaining vertices except vertex i, k be the any one vertex of S and $g(i,S)$ be the minimum travel cost of TSP. Then their recurrence relation can be defined as below,

$$g(i,S)=Min_{k \in S}\{C_{ik}+g(k,S-(k))\}$$

Numerical example and other things I have escaped for this as its less imp compared to other and too lengthy.

⊛. Concept of Memorization: [Imp]

Memorization means recording the results of earlier calculations so that we don't have to repeat the calculations later. If our code depends on the results of earlier calculations, and if the same calculations are performed over-and-over again, then it makes sense to store interim results so that we can avoid repeating the math.

Dynamic programming can also be implemented using memorization. With memorization, we implement the algorithm recursively, but we keep track of all of the sub solutions. If we encounter a sub-problem that we have seen, we look up the solution. If we encounter a sub-problem that we have not seen, we compute it and add it to the list of sub solutions we have seen. Memorization increases efficiency of dynamic programming.

⊛. Dynamic Programming vs. Memorization: [Imp]

Dynamic Programming	Memorization
i) Dynamic Programming is the research of finding an optimized plan to a problem through finding the best substructure of the problem for reusing the computation results.	i) Memorization is the technique to "remember" the result of a computation, and reuse it the next time instead of recomputing it, to save time.
ii) Dynamic Programming is bottom-up approach.	ii) Memorization is top down approach.
iii) Code gets complicated when lot of conditions are required.	iii) Code is easy and less complicated.
iv) It is fast, as we directly access previous states.	iv) Slow due to lot of recursive calls and return statements.
v) It is preferrable when the original problem requires all subproblems to be solved.	v) It is preferrable when the original problem requires only some subproblems to be solved.