Unit 5: Dynamic Programming

5.1. Introduction:

- Greedy approach vs Dynamic Programming
- recursion vs Dynamic Programming
- Elements of DP Strategy
- Concept of Memoization

5.2. DP Algorithms:

- Matrix Chain Multiplication
- String Editing
- Zero-One Knapsack Problem
- Floyd Warshwall Algorithm
- Travelling Salesperson Problem

Prepared by: Shiv Raj Pant

What is Dynamic programming?

- Dynamic Programming is a powerful way of designing the algorithms
- It is widely used to solve the optimization problems.
-

Matrix chain multiplication

Brainstorming:
Consider a problem of multiplying three matrices A(40x30), B(30x40), C(40x20)

Since matrix multiplication is associative, we can have two ways to multiply above matrices: A*(B*C) and (A*B)*C

The resulting matrix is same.

But,
Performing A*(B*C) requires 40*30*20+30*40*20 = 48000 scalar multiplications
Performing (A*B)*C requires 40*30*40+40*40*20 = 80000 scalar multiplications

Observe the difference!
Multiplying a sequence of matrices carefully with appropriate parenthesization(precedence) can result in great efficiency.

The problem we will deal here is "how to find an optimal parenthesization ? such that the total number of scalar multiplications is minimum.

*To multiply two matrices of sizes (pxq) and (qxr) we need to perform p*q*r scalar multiplications*

Problem statement

Given a chain $\{A_1, A_2, \ldots, A_n\}$ of n matrices and their sizes, fully parenthesize the product $A_1A_2.. . A_n$ in a way that minimizes the number of scalar multiplications.

How to solve?

1. Naïve approach:
   - Try all possible parenthesizations and select the optimal one. This approach is very inefficient (exponential time).

2. Dynamic programming approach:
   - The problem can be solved efficiently by dynamic programming.
   - The basic steps for designing DP algorithm are as follows:

i. Characterizing the optimal substructure:

- Given a chain $A_1A_2.. . A_n$, a parenthesization splits the product at k as $(A_1A_2.. A_k)(A_{k+1}.. A_n)$.
- For k to be optimal split, the parenthesizations for the subchains $A_1A_2.. A_k$ and $A_{k+1}.. A_n$ must also be optimal.

ii. Recursive definition of optimal solution:

- let m[i,j] denotes minimum number of scalar multiplications needed to compute $A_i...A_j$. Then the minimum number of scalar multiplications needed to compute the whole problem would be m[1,n]
- Let $P=\{p_0,p_1,..,p_i)$ be the sizes the matrices where the size of A_i is $p_{i-1} \times p_i$
- Now we can define the solution recursively as follows:

if i=j (nothing to compute)

$$m[i,j] = \begin{cases} 0 \\ \min_{i \le k < j} (m[i,k] + m[k+1,j] + p_{i-1} \cdot p_k \cdot p_j) \end{cases}$$

if i<j

Also keep track of s[i,j] =k for optimal split.

iii.Computation of optimal cost in bottom up manner:

- Recursive computation of above recurrence would take exponential time hence we can use DP with memorization to compute the cost in bottom up manner and saving the solutions to the subproblems in m.

Example:

Given the matrices:

$A_1(20x15)$, $A_2(15x30)$, $A_3(30x5)$, $A_4(5x25)$, $A_5(25x10)$, $A_6(10x5)$

Find the optimal parenthesization for the product $A_1A_2A_3A_4A_5A_6$

Sol^n:

Here we have the sizes of the matrices

P = | 20 | 15 | 30 | 5 | 25 | 10 | 5 |

Now we compute the optimal parenthesization in bottom up manner.

Chain length = 1

$(A_1) = (A_2) = (A_3) = (A_4) = (A_5) = (A_6) = 0$

Chain length = 2

$(A_1)(A_2) = 20*15*30 = 9000$, optimal split(k) = 1

$(A_2)(A_3) = 15*30*5 = 2250$, optimal split(k) = 2

$(A_3)(A_4) = 30*5*25 = 3750$, optimal split(k) = 3

$(A_4)(A_5) = 5*25*10 = 1250$, optimal split(k) = 4

$(A_5)(A_6) = 25*10*5 = 1250$, optimal split(k) = 5

Chain length	parenthesization	# of multiplications	Optimal split position
3	$(A_1)(A_2A_3)$ $(A_1A_2)(A_3)$	3750 (optimal) 12000	1
	$(A_2)(A_3A_4)$ $(A_2A_3)(A_4)$	15000 4125 (optimal)	3
	$(A_3)(A_4A_5)$ $(A_3A_4)(A_5)$		
	$(A_4)(A_5A_6)$ $(A_4A_5)(A_6)$		
4	$(A_1)(A_2A_3A_4)$ $(A_1A_2)(A_3A_4)$ $(A_1A_2A_3)(A_4)$		
	$(A_2)(A_3A_4A_5)$ $(A_2A_3)(A_4A_5)$ $(A_2A_3A_4)(A_5)$		
	$(A_3)(A_4A_5A_6)$ $(A_3A_4)(A_5A_6)$ $(A_3A_4A_5)(A_6)$		
5			
6			

0	9000	3750	6250	6000	5625
	0	2250	4125	4250	4125
		0	3750	2750	2250
			0	1250	1500
				0	1250
					0

S[i,j]		1	1	3	3	1
			2	3	3	3
				3	3	3
					4	5
						5

Algorithm: The input to the algorithm is array p of the orders $\{p_0, p_1, \ldots, p_n\}$

```
MatrixChainOrder(p)
{ n = length[p]-1;
for(i=1;i<=n;i++)
        m[i,i] = 0;
for(l = 2;l<=n;l++)                    //Chain length
        for(i=1;i<=n-l+1;i++)
        {       j = i + l –1;
                m[i,j]= ∝;
                for(k=i;k<=j-1;k++)    //find optimal split position k
                {       q = m[i,k]+m[k+1,j] + p[i-1].p[k].p[j];
                        if(q<m[i,j]) { m[i,j] = q;  s[i,j] = k;}
                }
        }
}
```

Analysis: ??

Construction of optimal solution:

The optimal solution can be obtained from the table given by s[i,j]. Since each s[i,j] holds value k for the optimal split the result for the sequence $A_{i...j}$ is $(A_{i...k})(A_{k+1...j})$. So the recovering sequence can be viewed as

$S[1,n] = (A_{1...s[1,n]})(A_{s[1,n]+1...n})$

$S[1,s[1,n]] = (A_{1...s[1,s[1,n]]})(A_{s[1,s[1,n]]+1...n})$ and so on. This can be written as recursive algorithm like

```
ParenthesizeChain(s,i,j)
{ if(i==j) print "Ai";
else{
        print "(";
        ParenthesizeChain(s,i,s[i,j]);
        ParenthesizeChain(s,s[i,j]+1,j);
        print ")";
}}
```

S[i,j]		1	1	3	3	1
			2	3	3	3
				3	3	3
					4	5
						5

String editing problem

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

$X = x_1\ x_2\ x_3 \ldots\ x_n$

$Y = y_1\ y_2\ y_3 \ldots\ y_m$

The task is to transform X into Y using a sequence of edit operations.

The permitted operations are:-

insert(y_j) : insert a 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 into Y.

Example:

X = *aabab*

Y = *babb*

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

There are many solutions:

i. *delete*(x_1) *change*(x_2, y_1) *change*(x_3, y_2) *change*(x_4, y_3)

total cost = 1+2+2+2 = 7

ii. *delete*(x_1) *delete*(x_2) insert(y_4)

total cost = 1+1+1 = 3

And so on..

There can be many such solutions but we need to find an optimal solution.

How to solve?

We can solve the problem using dynamic programming.

Let *cost(i,j)* be the minimum cost of an edit sequence for transforming $x_1\, x_2\, x_3 \ldots \ldots x_i$ into $y_1 y_2\, y_3 \ldots \ldots y_j$

Then the cost of optimal edit sequence is *cost(n,m)*

We can define the solution recursively as following dynamic program:

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

Example:

X = *aabab*

Y = *babb*

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

		a X1	a x2	b x3	a x4	b x5
		0				
b	y1					
a	y2					
b	y3					
b	y4					

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

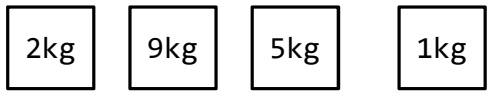
Then, fill other cells applying the 4th case of dynamic program.

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

- This is the knapsack problem, where the thief can not take fraction of any item.
- i.e. the thief can either take whole item or leave it.
- It is worth noting that this problem can not be solved by greedy approach. To prove this, consider an example:



Solution with greedy approach: (item1,item3,item4} with profit = Rs. 49

Optimal solution: (item2,item4) profit = Rs. 58

Dynamic program

Let c[i][w] be the optimal profit obtained from the problem with items 1,2,..,i and knapsack capacity w.

Then we can develop a recurrence for the dynamic program as follows:

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

We can understand the above relation as follows:

1. if we have got no item or the maximum weight is zero then the solution is 0,

2. if the weight of the object that is being added exceeds the total weight remaining then the solution is the solution up to the previous item addition

3. and if there is enough capacity in knapsack for item being added, then the solution consists of the value that is maximum between the summed up value up to previous addition of item or the value after addition of the last item.

Example: v[] = {2,3,3,4,4,5,7,8,8}; w[] = {3,5,7,4,3,9,2,11,5} W = 15.

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

W	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
i=0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
i=1	0	0	0	2	2	2	2	2	2	2	2	2	2	2	2	2
i=2	0	0	0	2	2	3	3	3	5	5	5	5	5	5	5	5
i=3	0	0	0	2	2	3	3	3	5	5	5	5	6	6	6	8
i=4	0	0	0	2	4	4	4	6	6	7	7	7	9	9	9	9
i=5	0	0	0	4	4	4	6	8	8	8	10	10	11	11	11	13
i=6	0	0	0	4	4	4	6	8	8	8	10	10	11	11	11	13
i=7	0	0	7	7	7	11	11	11	13	15	15	15	17	17	18	18
i=8	0	0	7	7	7	11	11	11	13	15	15	15	17	17	18	18
i=9	0	0	7	7	7	11	11	15	15	15	19	19	19	21	23	23

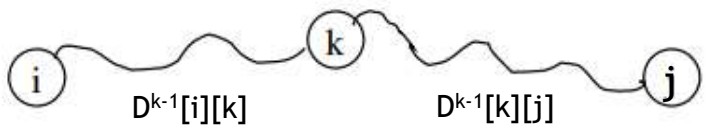
Pseudocode

```
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(w=1; w<=W;w++)
                        if(w[i]<W) then if v[i] +c[i-1,w-w[i]] > c[i-1,w] then
                                        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]; }}
```

Running time ?

Floyd Warshall Algorithm

- It is dynamic programing approach to solve all-Pairs shortest path Problem
- Also works for negative-weight edges
- Consider a weighted graph G = (V,E).
- Let the weight of edge connecting vertices i and j be w_{ij}.
- Let W be the adjacency matrix for the given graph G.
- Let D^k denote an nxn matrix where $D^k[i][j]$ is defined as the weight of the shortest path from the vertex i to vertex j using only vertices from 1,2,…,k as intermediate vertices in the path.

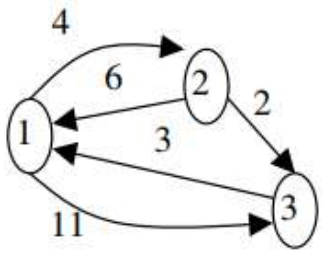


Now we can develop the dynamic program as follows:

$$D^k[i][j] = \begin{cases} 0 & , \text{if } i=j \\ \text{Min}\{D^{k-1}[i][j], D^{k-1}[i][k] + D^{k-1}[k][j]\}, & \text{if } i \neq j \end{cases}$$

Where, initially, $D^0[i][j] = W[i][j]$

Example: Compute the shortest paths between each pair of nodes in the following graph using Floyd-Warshall algorithm.



Sol^n:

Adjacency Matrix

W	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

= D^0

$Min\{D^{k-1}[i][j], D^{k-1}[i][k] + D^{k-1}[k][j]\}$

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

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

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

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

$D^1(3,1) = \min\{D^0(3,1), D^0(3,1)+ D^0(1,1)\} = \min\{3, 3+ 0\} = 3$

$\mathbf{D^1(3,2) = \min\{D^0(3,2), D^0(3,1)+ D^0(1,2)\} = \min\{\infty, 3+ 4\} = 7}$

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

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

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

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

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

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

$D^2(3,2) = \min\{D^1(3,2), D^1(3,2)+ D^1(2,2)\}$
$= \min\{7, 7+ 0\} = 7$

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

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

Pseudocode

```
FloydWarshalAPSP(W,D,n)    // W is adjacency matrix of graph G.
{
for(i=1;i<=n;i++)
   for(j=1;j<=1;j++)
        D[i][j] = W[i][j];      // initially D[][] is D^0.
For(k=1;k<=n;k++)
   for(i=1;i<=n;i++)
      for(j=1;j<=1;j++)
        D[i][j] = min{D[i][j], D[i][k]+ D[k][j]};   //D[][]'s are D^k's.
}
```

Analysis:

Clearly the above algorithm's running time is $O(n^3)$, where n is cardinality of set V of vertices.

Travelling salesperson problem

- Consider a salesperson S who wants to sell items in a city C. The city has many locations $l_1, l_2, \ldots l_n$.
- The problem is that S must visit all the locations and return to initial location by travelling minimum total distance.
- Mathematically, TSP can be defined as follows:

  Let G=(V,E) be directed graph with edge cost $C_{ij} > 0$ from node i to j. Here, $C_{ij} = \infty$ if there is not arrow from i to j.

  A **tour** of G is a directed simple cycle that includes every vertex in V.

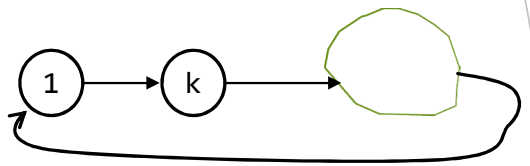
  The TSP is to find a **tour** of minimum cost.

Solution:

Let us number the vertices as V = {1,2,.. .n). Let the start vertex be 1.

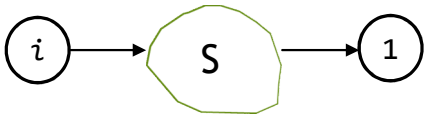
Let k ϵ V-{1} be a vertex.

Then, each tour consists of an edge (1,k) and a path from K to 1.



For an optimal tour, the path from k to 1 must be shortest.

Let g(i,S) be the length of the shortest path from vertex i, going through all vertices in set S and terminating at vertex 1.

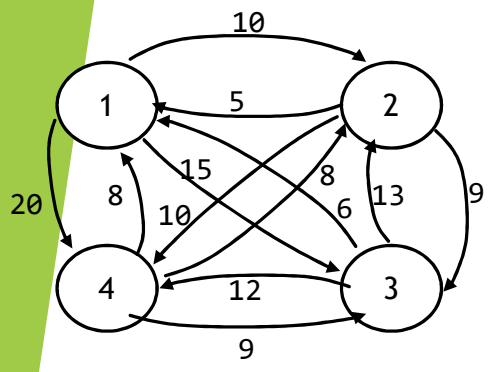


The optimal solution for the whole problem is g(1, V-{1})

Now we can define g(i,S) recursively as follows:

$$g(i,S) = \begin{cases} C_{i1} & , \text{if } S=\phi \\ \min\limits_{j \in S} (C_{ij} + g(j,S-\{j\}) & \end{cases}$$

Example: Find an optimal tour from vertex 1.



	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

Adjacency matrix(C)

	Φ	1			2		
		{2}	{3}	{4}	{2,3}	{3,4}	{2,4}
2	5	--			--		
3	6		--			--	
4	8			--			--

g-table

For |S| = 0 (S = ϕ)

$g(2, \phi) = C_{21} = 5$

$g(3, \phi) = C_{31} = 6$

$g(4, \phi) = C_{41} = 8$

For |S| = 1

$g(2, \{3\}) = \min(C_{23}+g(3, \phi)) = 15$

$g(2, \{4\}) = \min(C_{24}+g(4, \phi)) = 18$

$g(3, \{2\}) = \min(C_{32}+g(2, \phi)) = 18$

$g(3, \{4\}) = \min(C_{34}+g(4, \phi)) = 20$

$g(4, \{2\}) = \min(C_{42}+g(2, \phi)) = 13$

$g(4, \{3\}) = \min(C_{43}+g(3, \phi)) = 15$

For |S| = 2

$g(2,\{3,4\}) = \min\{C_{23}+g(3,\{4\}), C_{24}+g(4,\{3\})\} = 25$

$g(3,\{2,4\}) = \min\{C_{32}+g(2,\{4\}), C_{34}+g(4,\{2\})\} = 25$

$g(4,\{2,3\}) = \min\{C_{42}+g(2,\{3\}), C_{43}+g(3,\{2\})\} = 23$

For |S| = 3

$G(1,\{2,3,4\}) = \min\{C_{12}+g(2,\{3,4\}), C_{13}+g(3,\{2,4\}), C_{14}+ g(4,\{2,3\})\}$

$= 35$

Note: Here, we have not shown the g(i,S) computations for i=1 in the subproblems because it is not really needed.

Finding optimal tour

The optimal tour can be computed by tracking, in bottom up manner, which vertex provided minimum value during the course of computation.

For G(1,{2,3,4}) , the vertex providing optimal value is 2.

So the tour goes from 1 to 2.

Then for g(2,{3,4}), the vertex is 4

The tour goes 1->2->4

Then for g(4, {3}) , the vertex is 3

Hence the optimal tour is : 1->2->4->3->1

Complexity of the algorithm

See Horowitz book.

End of unit 5

Thank You!