WEEK 6 - Lecture 2

INTRODUCTION:

We continued our discussion on Dynamic Programming and tried to solve the problem of Matrix Chain Multiplication using it. We also looked at another very famous problem called Knapsack.

Matrix Chain Multiplication:

In this problem we are given a chain of matrices which we must multiply in the given order (because matrix multiplication is only associative and not commutative) and our task will be to minimise the number of computations.

Let us understand the cost of multiplying two matrices say $A[n, m]$

and $B[m, o]$. So the resultant matrix will have $n * o$ elements and to calculate each cell of the resultant matrix , we would need to perform m multiplications. Hence the total cost of computation would be $n * m * o$.

Now since matrix multiplication is associative, we know there can be various ways of multiplying a chain of matrix .

For example: A[10 * 30] , B[30 * 5] and C[5 * 60]

We can multiply them using two methods:

```
(AB)C = (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operations
A(BC) = (30×5×60) + (10×30×60) = 9000 + 18000 = 27000 operations.
```

Hence out of all possible ways to multiply the given matrices, we need to choose the one with the minimum number of computations.

Dynamic Programming Approach :

For the dynamic programming solution , we must have optimal substructure property as well as the overlapping problems property .

Hence let $dp[i][j]$ denote the min cost of multiplying matrices from index i to j. To get $dp[i][j]$, we can iterate through all possible ways of multiplying that is multiply the first matrix with the remaining matrices, multiply first 2 matrix with remaining matrix and so on.

Formally speaking , we can iterate through all k from i to j and take the cost for a particular k as the cost of multiplying matrices from i to k plus cost to multiply from k+1 to j and finally the cost of multiplying these resulting two matrices which we can easily obtain from the relation we discussed above.

So we can observe we have the optimal substructure property in our array.

Hence the transition would look like:

Say A and B have index i and j with dimensions $n * m$ and $m * o$.

$$dp[i][j] = min(dp[i][k] + dp[k+1][j] + n * m * o) \forall i <= k < j$$

Base Case: $dp[0][0] = 0$ since there is zero cost in case of single matrix.

```
for i=1 to n: dp[i][i]=0
for s=1 to n-1:
  for i=1 to n-s:
    j=i+s
    for k=i to k=j
      dp[i][j] = min(dp[i][k]+dp[k+1][j]+m[i-1]*m[k]*m[j])
return dp[1][n]
```

The answer would be stored in $dp[1][n]$ which denotes the minimum cost to multiply matrices from index 1 to n that is all given matrices.

Time Complexity:

We can see that we can a 2-d dp and to find the value of each cell of that dp , we would need to iterate through all the elements in the range from i to j.

So there are n*n dp states and for each state , we need $O(n)$ time to calculate it resulting in the final complexity of $O(n^3)$

Knapsack Problem:

Here, we are given a knapsack of say capacity W which denotes the total weight of the items the knapsack can hold.

Now we are given n items and for each item we are given the corresponding weight and value. So we need to pick a subsets of items from the given items so that the total weight of items is less than equal to W while the total value is maximum.

Hence we need to maximise value given some constraint of the total weight

Dynamic programming approach:

We can assume

$dp[i][j]$ as the maximum value we can obtain considering the first i elements having total weight j .

So we can look at the optimal substructure that we have in this question that say we are analysing the i th item , we have two choices →

- Don't pick that element : If we don't pick the i th item , we will have to make the weight j from the previous i-1 items only so one possible contender is $dp[i-1][j]$.
- Pick that element : If we pick that element , then we will add $val[i]$ to our ans that is the value of the i th item but at the same item we will add $wgt[i]$ to our knapsack , so the resulting ans from this possibility would be $dp[i-1][j-wgt[i]] + val[i]$.

  Here we would need to make sure that j is greater than equal to wgt[i].

So value of $dp[i][j]$ would be

$$dp[i][j] = max(dp[i-1][j], dp[i-1][j-wgt[i]] + val[i]$$

Hence our answer would be

$$ans = max(dp[n-1][j]) \forall 1 <= j <= W$$

The pseudo code would look like :

```
for j=0 to n-1:
  for w=1 to W:
```

```
  if(w_j>w) : dp[w][j]=dp[w][j-1]
  else : dp[w][j]=max(dp[w][j-1],dp[w-w_j][j-1]+v_j)
int ans=0;
for j=1 to W
ans=max(ans,dp[n-1][j])
```

Food for thought:

Printing the Brackets in MCM problem:

In class, we discussed how to find the minimum cost of multiplying matrices but it is not something which is very useful.

The more realistic problem should be to print the optimal bracket sequence for multiplying the matrices.

Algorithm:

- For each $dp[i][j]$ we can also store the break point of that transition that is the k such that if we break at k , we get minimum cost.
- So we can maintain a parent array along with dp array of same dimensions and answer this problem.
- Now make a recursive function and call the function until i==j at which point we can just print it directly.
- if $i! = j$ we can first print the sequence for the left sub problem that is from i to k and then for the right sub problem that is from k+1 to j.

Pseudo code would look like:

```
void print_bracket(int i,int j,vector<vector<int>> &brackets, char &cur_name)
{
if(i == j){
        cout<<cur_name;
        cur_name++;
    } else {
        cout<<"(";
```

```
        // Reduce the problem into left sub-problem
        print_bracket(i, brackets[i][j], brackets, cur_name);
        // Reduce the problem into right sub-problem
        print_bracket(brackets[i][j]+1, j, brackets, cur_name);
        cout<<")";
    }
}
```