week6_lec2

Main Ideas

- Matrix Chain Multiplication
- Knapsack

Matrix Chain Multiplication

Problem : Given a sequence of matrices, find the most efficient way to multiply these matrices together. The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications.

Therefore, notice that on taking use of *associativity of matrix multiplication*, depending on the *order* in which we carry out the multiplications, we can have *different costs*.

p[] = {40, 20, 30, 10, 30}

There are 4 matrices of dimensions 40x20, 20x30, 30x10 and 10x30.Let the input 4 matrices be A, B, C and D.

Solution:

We need to find the solution as soon as possible otherwise if we waste time finding the optimal order among the exponentially many permutations it will be equivalent to doing a brute force algorithm.

This is where Dynamic Programming comes into the picture.

We first verify if the problem satisfies the two basic conditions for the usage of Dynamic Programming.

- Optimal Substructure:

  If we try and place the brackets in all possible positions and calculate the cost for each arrangement, we see that for a chain of matrices of size N, we can place the first set of brackets in N-1 ways. When we place this set of brackets, we find that we have broken down this problem into a subproblem of smaller size.

- Overlapping Subproblem:

If we see the recursive approach of the problem, we find that same subset of the problem is calculated multiple times and is making the approach ineffective. The better way is using memorization. The recursion map of the problem suggests that the problem has Overlapping Subproblem property.

Pseudo Code

```
int MatrixChainMultiplicationOrder(int p[], int n)
{

    int m[n][n];

    int i, j, k, L, q;

    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] = INT_MAX;
            for (k = i; k <= j - 1; 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;
            }
        }
    }

    return m[1][n - 1];
}
```

Time Complexity - $O(N^3)$

Space Complexity - $O(N^2)$

Knapsack

Problems : Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack.

Given two integer arrays val[0..n-1] and wt[0..n-1]

Given an integer W which represents knapsack capacity, find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to W.

Solution :

This is a *np-hard* problem. This problem is an example in which DP is applicable.

```
int max(int a, int b)
{
    return (a > b) ? a : b;
}

int knapSack(int W, int wt[], int val[], int n)
{
    int i, w;
      vector<vector<int>> K(n + 1, vector<int>(W + 1));

    for(i = 0; i <= n; i++)
    {
        for(w = 0; w <= W; w++)
        {
            if (i == 0 || w == 0)
                K[i][w] = 0;
            else if (wt[i - 1] <= w)
                K[i][w] = max(val[i - 1] +
                              K[i - 1][w - wt[i - 1]],
                              K[i - 1][w]);
            else
                K[i][w] = K[i - 1][w];
        }
    }
    return K[n][W];
}
```

Time Complexity: $O(N * W)$.

Space Complexity : $O(N * W)$.