Longest Common Subsequence

An example of Dynamic programming

Question-

Given two sequences, find the length of longest subsequence present in both of them. A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous. For example, “abc”, “abg”, “bdf”, “aeg”, ‘”acefg”, .. etc are subsequences of “abcdefg”.

Example:

LCS for input Sequences “ABCDGH” and “AEDFHR” is “ADH” of length 3.

LCS for input Sequences “AGGTAB” and “GXTXAYB” is “GTAB” of length 4.

A naive solution for this problem is to generate all subsequences of both given sequences and find the longest matching subsequence.

This solution is exponential in term of time complexity. Hence we solve it using Dynamic Programming.

A recursive implementation:

```
int lcs( char *X, char *Y, int m, int n )
{
  if (m == 0 || n == 0)
    return 0;
  if (X[m-1] == Y[n-1])
    return 1 + lcs(X, Y, m-1, n-1);
  else
    return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n));
}
```

However in this approach we see that after drawing the recursion tree, most of the cases are repeated and we have to solve them again and again this results in exponential time complexity. Hence we use Tabulation or Memoization to avoid this scenario

Memoization- dp approach algorithm:

A memorized verison folliws a top down approach because we break the problem into sub problems and then calculate and store the value.

```
int  LCSLength(string X, string Y, int m, int n, auto &lookup)
{   // return if we have reached the end of either string
        if (m == 0 || n == 0)
                return 0;
        // construct a unique map key from dynamic elements of the input
        string key = to_string(m) + "|" + to_string(n);

        // if sub-problem is seen for the first time, solve it and
        // store its result in a map
        if (lookup.find(key) == lookup.end())
        {
                // if last character of X and Y matches
                if (X[m - 1] == Y[n - 1])
                        lookup[key] = LCSLength(X, Y, m - 1, n - 1, lookup) + 1;
                else
                // else if last character of X and Y don't match
                lookup[key] = max(LCSLength(X, Y, m, n - 1, lookup),
                                                LCSLength(X, Y, m - 1, n, lookup));
        }
        // return the subproblem solution from the map
        return lookup[key];
}
main()
{
        string X = "ABCBDAB", Y = "BDCABA";
        // create a map to store solutions of subproblems
        unordered_map<string, int> lookup;
        cout << LCSLength(X, Y, X.length(), Y.length(), lookup);
}
```

Tabulation approach:

Let X be “XMJAUZ” and Y be “MZJAWXU”. The longest subsequence is “MJAU” of length 4.

		0	1	2	3	4	5	6	7
		Ø	M	Z	J	A	W	X	U
0	Ø	0	0	0	0	0	0	0	0
1	X	0	0	0	0	0	0	1	1
2	M	0	1	1	1	1	1	1	1
3	J	0	1	1	2	2	2	2	2
4	Y	0	1	1	2	2	2	2	2
5	A	0	1	1	2	3	3	3	3
6	U	0	1	1	2	3	3	3	4
7	Z	0	1	2	2	3	3	3	4

```
int LCSLength(string X, string Y)
{
	int m = X.length(), n = Y.length();

	// lookup table stores solution to already computed sub-problems
	// i.e. lookup[i][j] stores the length of LCS of substring
	// X[0..i-1] and Y[0..j-1]

	int lookup[m + 1][n + 1];

	// first column of the lookup table will be all 0
	for (int i = 0; i <= m; i++)
		lookup[i][0] = 0;

	// first row of the lookup table will be all 0
	for (int j = 0; j <= n; j++)
		lookup[0][j] = 0;

	// fill the lookup table in bottom-up manner
	for (int i = 1; i <= m; i++)
	{
		for (int j = 1; j <= n; j++)
		{
			// if current character of X and Y matches
			if (X[i - 1] == Y[j - 1])
				lookup[i][j] = lookup[i - 1][j - 1] + 1;

			// else if current character of X and Y don't match
			else
				lookup[i][j] = max(lookup[i - 1][j], lookup[i][j - 1]);
		}
	}
```

```
        // LCS will be last entry in the lookup table
        return lookup[m][n];
}


 main()
{
        string X = "XMJYAUZ", Y = "MZJAWXU";

        cout << LCSLength(X, Y);

        return 0;
}
```