Longest Common Subsequence Problem

In this note we'll be studying about the longest common subsequence problem.

Definitions

Def.: Given arrays A and B, a common subsequence is a subsequence S of A that is also a subsequence of B.

Exemple: `A = "abbcdb"` `B = "acbdbcd"`
The sequence `S = "abb"` is a subsequence of A and B at the same time.
More common subsequences are:

- `S = "a"` ;
- `S = "b"` ;
- `S = ""` ;
- `S = "abbcd` ;
- `S = "acd` ;

and lots of others.

Def.: The longest common subsequence (LCS) is the subsequence with greater lenght (there can be more then one).

Problem

The problem is given two arrays, find the lenght of the LCS.

For solving this problem, we'll use dynamic programming.

The naive approuch is trying all subsequences of A againg all subsequences of B, but we can use some the *Optimal substructure* property to have some information.

Let's say that one instance of your DP is (i, j) where i is a pointer to the i-th element of A and j is a pointer to the j-th element of B.
Then we have three options:

1. If $A[i] = B[j]$, we can say that the solution to this instance is $LCS(i-1,\ j-1)+1$ where LCS means the solution to some instance;
2. We can ignore the i-th element of A and just say that the solution is $LCS(i-1,\ j)$;

3. And we can ignore the j-th element of B and just say that the solution is $LCS(i,\ j-1)$.

For any instance, that's all we can do: add one to the previous solution, ignore i and proceed without it or ignore j and proceed without it.

As we and to maxime the lenght, we have this formula:

$$LCS(i,j) = \begin{cases} max\left\{LCS(i-1,\ j-1)+1,\ LCS(i-1,\ j),\ LCS(i,\ j-1)\right\}, & if A[i] = B[j] \\ max\left\{LCS(i-1,\ j),\ LCS(i,\ j-1)\right\}, & if A[i] \neq B[j] \end{cases}$$

But we still need a base case! What must be the value of $LCS(i,j)$ if i or j are equal to 0?
Then we have some implementation problems: the easiest value of LCS is when one of the arrays is empty. Then LCS is just zero. But how can we say that the array is empty?

We can do somethings:
1. Adding a new rule that says if $i-1$ or $j-1$ result in a negative number, then the answer of that recursive call is zero. For instance, supose we want $LCS(3,0)$, then it will possibly cal $LCS(2,-1)$ and $LCS(3,-1)$ which are bad calls. Then we can just verity before calling then and instead of calling, we just say that their values are zero;
2. We can add a placeholder at the start of the arrays in the position 0 and implementing 1-indexed arrays. Then, we just need to set $LCS(i,0)$ and $LCS(0,i)$ to be zero for all possible $i \leq max(n,m)$ (where $|A| = n \land |B| = m$).

Any of these implementations is valid, but I'll be opting for the second one because then I won't need to check more things in each instance.

Implementation

```
int const N = 10e3;
int const M = 10e3;
int LCS[N][M]; // This is the DP table
int A[N], B[M]; // 1-index arrays of the problem
int n, m; //lenght of A and B respectively

int main() {
  // Input
  cin >> n >> m;
  for (int i = 1; i <= n; i++) cin >> A[i];
  for (int i = 1; i <= m; i++) cin >> B[i];

  // Base case
  for (int i = 0; i <= max(n, m); i++) LCS[i][0] = LCS[0][i] = 0;


  // The bottom-up solution
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
      LCS[i][j] = max(LCS[i-1][j], LCS[i][j-1]);
      if (A[i] == B[j]) LCS[i][j] = max(LCS[i][j], LCS[i-1][j-1] + 1);
    }
  }

  // Outputing the solution
  cout << LCS[n][m] << endl;

  return 0;
}
```

Analysis

As any DP, the complexity is just the number of instances times the complexity per instance.

The solution per instance is, in the wrost case, just the max operations, two assignment operations and one if. That's just $O(1)$. The number of instances is n*m. Therefore, the time complexity is $O(n \cdot m)$.