Intro to Dynamic Programming

Fall 2020

RUTGERS UNIVERSITY QUANTITATIVE FINANCE CLUB

QCF

Definition

A mathematical optimization and computer programming method used to **SIMPLIFY** complicated problems into **SMALLER SUBPROBLEMS**, and solving those in a recursive manner (does not necessarily mean using recursion).

- If sub-problems can be nested recursively inside larger problems, then Dynamic Programming can be used
- Dynamic Programming is used to improve time efficiency

Two Conditions

1) **Overlapping subproblems**: we need to solve the same subproblem over and over again
2) **Optimal Substructure**: we can solve the problem by breaking it down into smaller problems

This will make sense after a few examples.

Example 1: Longest Common Subsequence

Problem: Given two sequences find the length of the longest subsequence present in both of them.

A subsequence is a sequence that appears in **relative** order but is not necessarily **contiguous**

What is the longest common subsequence of:

1. ABCDEFG
2. ABXDFG

1. AGGTAB
2. GXTXAYB

Example 1 (contd.)

What is the longest common subsequence of:

1. ABCDEFG
2. ABXDFG

1. AGGTAB
2. GXTXAYB

Common subsequences include:

A, B, D, F, G, AB, DF, DFG, ABD, ABDFG

Longest common subsequence: ABDFG

Longest common subsequence: GTAB

Example 1 - Brute Force

Find LCS of the input string “AGGTAB” (call this L1 and of length m) and “GXTXAYB” (call this L2 and of length n)

How to solve this?

Brute Force Approach

- Each character has 2 options: included in the subsequence or not
- Find all subsequences of each sequence - about 2^m subsequences per sequence
- Compare each subsequence with each subsequence of other sequence - 2^n comparisons per subsequence
- Total running time 2^m*2^n = O(2^(m+n))

That is the brute force and has an **exponential running time**.

Example 1 (contd.)

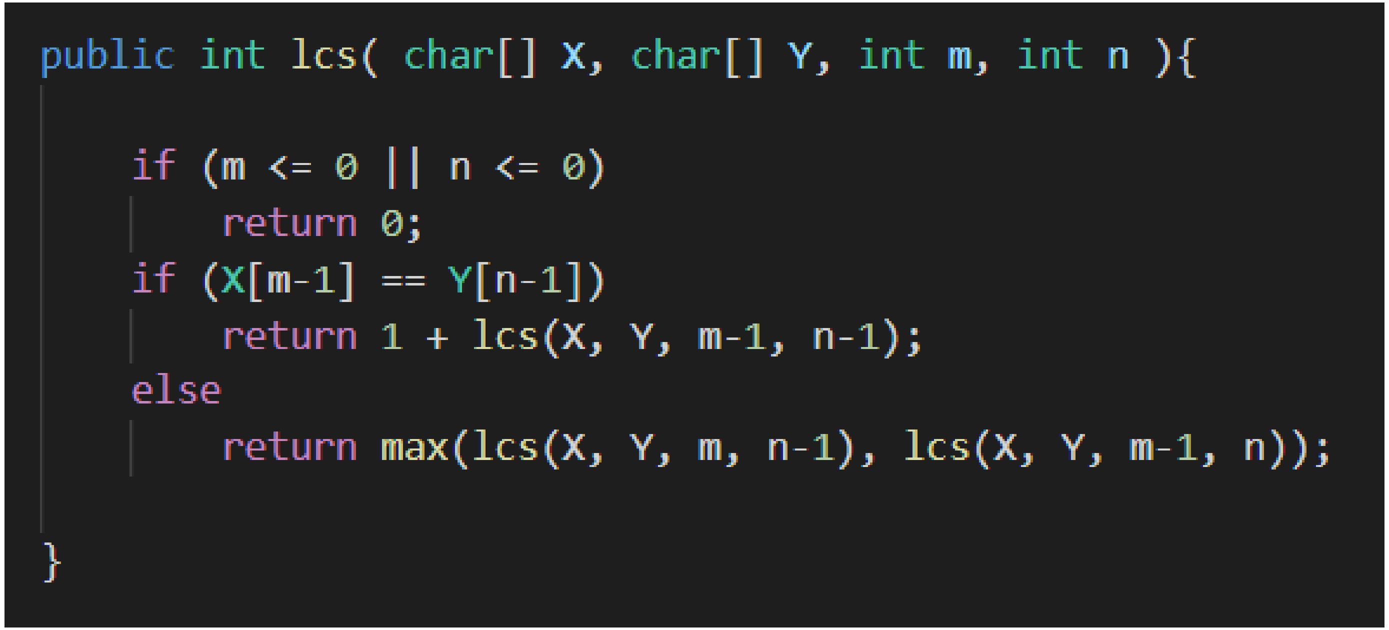
Where does one start:

- Beginning of each array
- **End of each array**

There can be exactly two cases by starting at the end. Either the last characters match or the last characters do not match.

- **Case 1** - last characters match (L1[m-1] == L2[n-1]): Increment the LCS by 1 and do LCS on L1[m-2] and L2[n-2]
- **Case 2** - last characters do not match (L1[m-1] != L2[n-1]): Find the max of LCS(L1[m-1], L2[n-2]) and LCS(L1[m-2], L2[n-1])

Example 1: Recursion



```
public 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));

}
```

We did it, right?

Recursion Tree

Example recursion tree using previous rules using the sequences “AXYT” and “AYZX”

```
                      lcs("AXYT", "AYZX")
                       /                 \
          lcs("AXY", "AYZX")            lcs("AXYT", "AYZ")
            /            \                /              \
lcs("AX", "AYZX")  lcs("AXY", "AYZ")  lcs("AXY", "AYZ")  lcs("AXYT", "AY")
```

Note the existence of similar tree nodes. That’s going to come into play later.

Example 1: Running Time

Worst case running time is **O(2^n)**, which is **exponential time**. This is similar to brute forcing everything. This occurs when every single character of L1 and L2 are mismatched (LCS length is 0). Additionally, we solved the same recursion problem multiple times.

Also, there is no way we can find out what the LCS is, we can just figure out the LCS length.

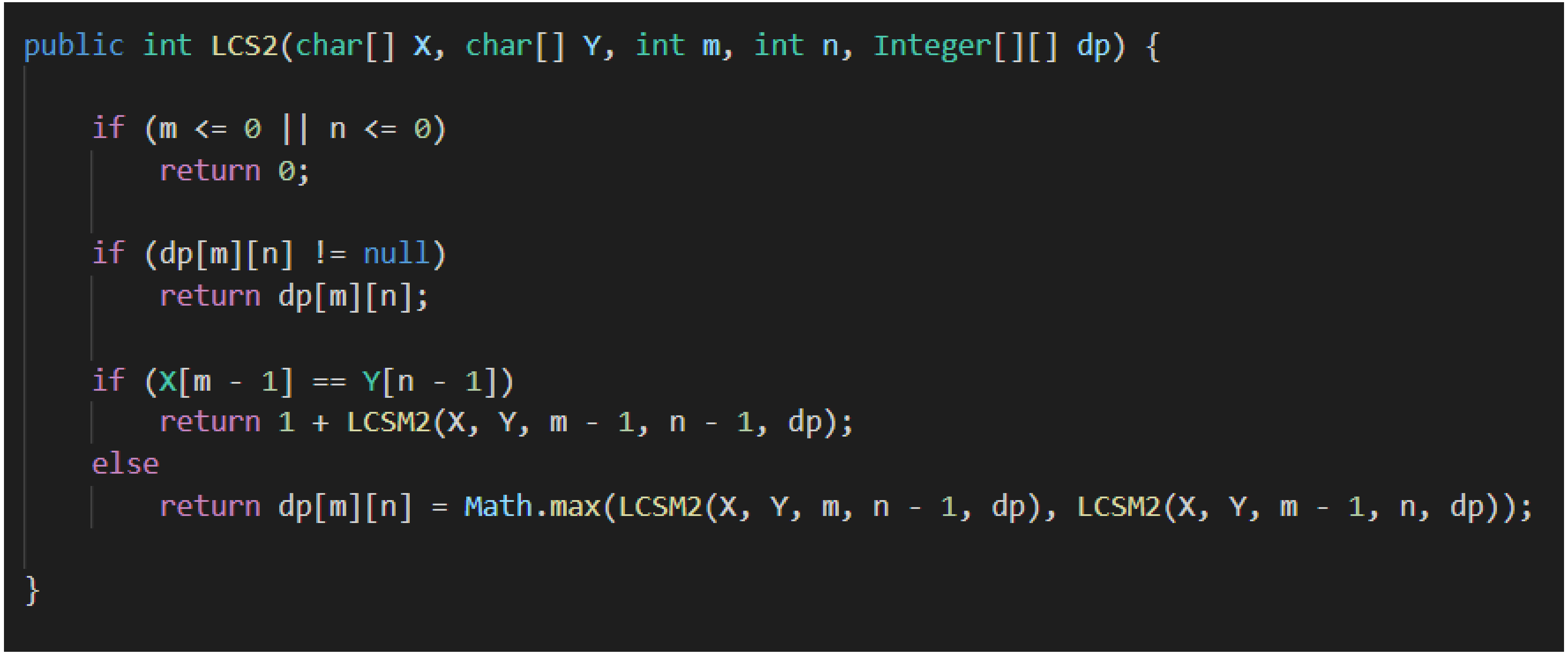
There has to be a better way!

Memoization - Important DP Technique

Note that we were solving some subproblems over and over again (overlapping subproblems). In the recursion tree a few slides ago, we solved the same problem 2 times (highlighted in green).

A better idea is to just solve the problem once, write down the solution, and whenever we are faced with the same subproblem, just spit out the answer. This is exactly what memoization does: "**solve once, remember forever**".

Top-Down Approach



```java
public int LCS2(char[] X, char[] Y, int m, int n, Integer[][] dp) {

    if (m <= 0 || n <= 0)
        return 0;

    if (dp[m][n] != null)
        return dp[m][n];

    if (X[m - 1] == Y[n - 1])
        return 1 + LCSM2(X, Y, m - 1, n - 1, dp);
    else
        return dp[m][n] = Math.max(LCSM2(X, Y, m, n - 1, dp), LCSM2(X, Y, m - 1, n, dp));

}
```

Evaluation

We managed to eliminate solving the same tree multiple times.

However, there is an even better way, called the **bottom-up** iterative method that is even more efficient.

Iterative Matrix Method

LCS	ϕ	A	G	G	T	A	B
ϕ	0	0	0	0	0	0	0
G	0						
X	0						
T	0						
X	0						
A	0						
Y	0						
B	0						

Phi represents a subsequence of length 0

Matrix (contd.)

LCS	ϕ	A	G	G	T	A	B
ϕ	0	0	0	0	0	0	0
G	0	⇦ ⇧ 0	↖ 1	↖ 1	⇦ ⇧ 1	⇦ ⇧ 1	⇦ ⇧ 1
X	0	⇦ ⇧ 0	1	1	1	1	1
T	0						
X	0						
A	0						
Y	0						
B	0						

QUANT FINANCE

If the letters match, then the number is 1+the top left diagonal. Else, it is the greater of the top and left

Approach

If the last characters match:

- LCS[i][j] = LCS[i-1][j-1] + 1

If the last characters don’t match:

- LCS[i][j] = max(LCS[i-1][j], LCS[i][j-1])

Final Matrix

LCS	ϕ	A	G	G	T	A	B
ϕ	0	0	0	0	0	0	0
G	0	0	1	1	1	1	1
X	0	0	1	1	1	1	1
T	0	0	1	1	2	2	2
X	0	0	1	1	2	2	2
A	0	1	1	1	2	3	3
Y	0	1	1	1	2	3	3
B	0	1	1	1	2	3	4

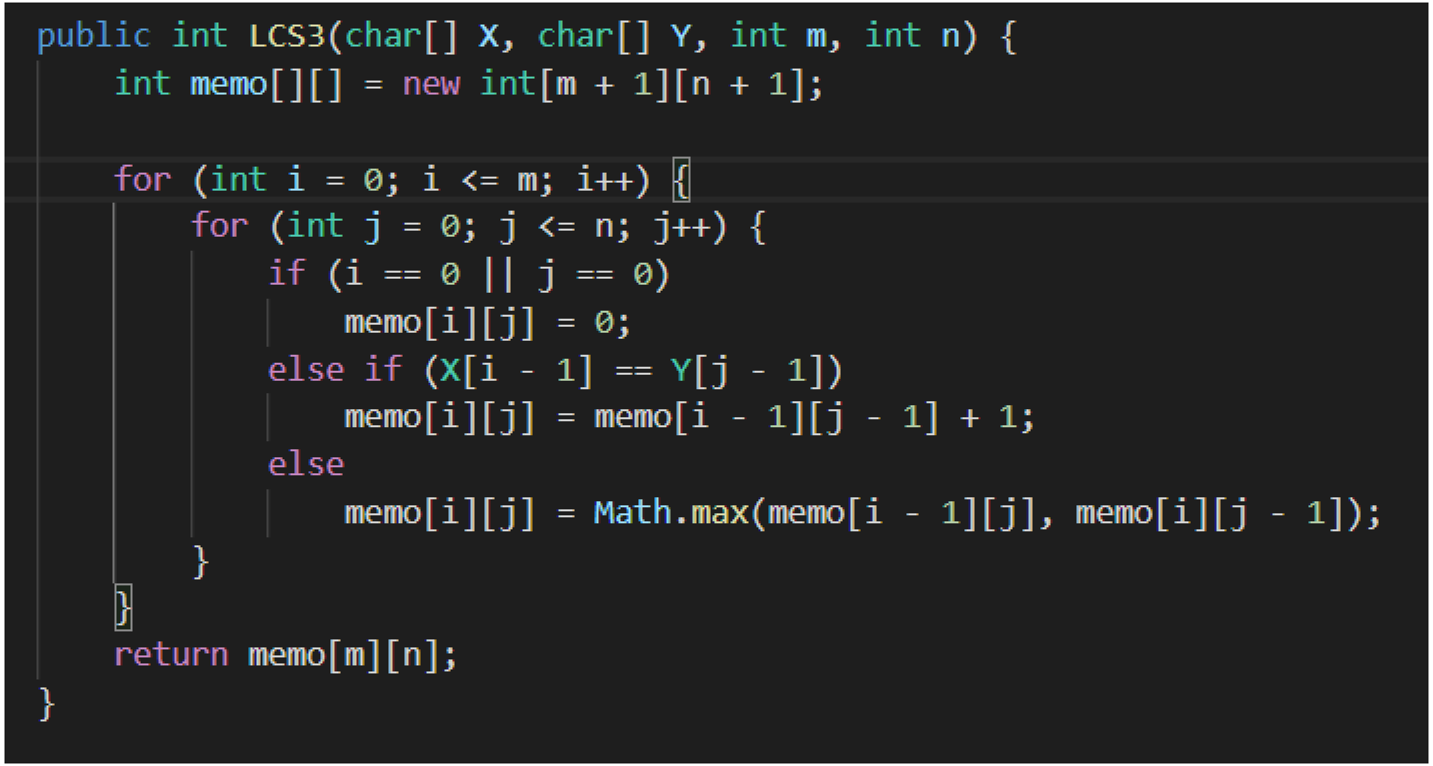
Finding Longest Subsequence

LCS	ϕ	A	G	G	T	A	B
ϕ	0	0	0	0	0	0	0
G	0	0	1	1	1	1	1
X	0	0	1	1	1	1	1
T	0	0	1	1	2	2	2
X	0	0	1	1	2	2	2
A	0	1	1	1	2	3	3
Y	0	1	1	1	2	3	3
B	0	1	1	1	2	3	4

Start at bottom right corner and check what is up and left. If current position is unequal to both, then it's part of the LCS and move up-left diagonally. Else, move to the position that's the same as the current position (either up or left)

QUANT FINANCE

Bottom Up Code



```java
public int LCS3(char[] X, char[] Y, int m, int n) {
    int memo[][] = new int[m + 1][n + 1];

    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0 || j == 0)
                memo[i][j] = 0;
            else if (X[i - 1] == Y[j - 1])
                memo[i][j] = memo[i - 1][j - 1] + 1;
            else
                memo[i][j] = Math.max(memo[i - 1][j], memo[i][j - 1]);
        }
    }
    return memo[m][n];
}
```

Runtime

O(m*n)

Huge improvement from the previous runtime of O(2^n) or more!

Better than top-down approach because there is no recursion involved (recursion overloads the computer stack). Also, we only need to compute this array **ONCE**.

Further Reading

- https://leetcode.com/problems/longest-common-subsequence/discuss/398711/ALL-4-ways-Recursion-greater-Top-down-greaterBottom-Up-greater-Efficient-Solution-O(N)-including-VIDEO-TUTORIAL

- https://leetcode.com/problems/longest-common-subsequence/discuss/351689/JavaPython-3-Two-DP-codes-of-O(mn)-and-O(min(m-n))-spaces-w-picture-and-analysis

Example 2: Maximum Sum Subarray

Problem: You are given an array, and you are to find the maximum sum of any **continuous** subarray

-2	-3	4	-1	-2	1	5	-3

-2	-3	4	-1	-2	1	5	-3

Maximum Sum Subarray is 4+(-1)+(-2)+1+5 = 7

QUANT FINANCE

How to solve it?

We can try to use **Brute Force**. That includes:

- Check all subarrays - there are O(n^2) subarrays
- Find subarray with the maximum sum - it takes O(n) time to compute the sum of a single subarray

Altogether it takes **O(n^3)** to do this method.

We can do much better by using dynamic programming!

Dynamic Programming Method

Let `dp[i]` be the maximum sum of any subarray that ends at index `i`. How can we use `dp[i]` to calculate `dp[i + 1]`?

There are two cases:

1) The max-sum-subarray ending at index `i + 1` includes index `i`:

```
dp[i + 1] = dp[i] + a[i + 1];
```

2) The max-sum-subarray ending at index `i + 1` doesn't include index `i`:

```
dp[i + 1] = a[i + 1];
```

Pick the higher of these two.

Method contd.

We will iterate through the array from beginning to end and will use two main variables, *max_so_far* and *curr_max*.

The variable *max_so_far* will keep track of the MSS to the current point and *curr_max* will keep track of the continuous segments using the method from the previous slide.

The variable *max_so_far* will update based on *curr_max*.

Known as **Kadane's algorithm**.

Dynamic Programming Solution

```
public static int maxSubArraySum(int a[], int size)
{
    int max_so_far = a[0];
    int curr_max = a[0];

    for (int i = 1; i < size; i++)
    {
        curr_max = Math.max(a[i], curr_max+a[i]);
        max_so_far = Math.max(max_so_far, curr_max);
    }
    return max_so_far;
 }
```

Variable Changes

Original array

-2	-3	4	-1	-2	1	5	-3

curr_max

-2	-3	4	3	1	2	7	4

max_so_far

-2	-2	4	4	4	4	7	7

Running Time

Loop through each element in the array exactly once

O(n)

Significant improvement from previous runtime of O(n^3)

Applications

Used in a variety of fields such as:

- Aerospace Engineering
- Genetics
- Software Developing
- Economics/Finance

Applications Within Finance

- Create more efficient algorithms that can run on “parallel architectures”, thereby expanding the breadth of problems to be addressed
- Three components to create models/algorithms using dynamic programming:
  - **Optimization** - to preserve certain mathematical features such as convexity and differentiability
  - **Approximation** - to illustrate efficiency of new methods
  - **Integration** - to build the algorithm

Exercise 1: Knapsack Problem

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. In other words, given two integer arrays val[0..n-1] and wt[0..n-1] which represent values and weights associated with n items respectively. Also 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. You cannot break an item, either pick the complete item, or don’t pick it (0-1 property).