CSE 4810 – Algorithm Engineering Lab

Lab 4

Lecture Sheet

Topic: Dynamic Programming

Dynamic Programming is useful for problems that have the following two properties:

1. Optimal Substructure, 2. Overlapping Subproblem

Approach to Solve Dynamic Programming Problems:

Step 1. Formulate a **Naïve-Recursive top-down algorithm**

- Functions with bigger inputs will utilize the output of functions of small inputs.
- Problem: Too many calls to the same function with same inputs. This is very redundant.
- Solution: Store the output of functions for corresponding inputs in an array.
- This approach is top-down because you start with the functions biggest inputs and then go to the functions with smaller inputs.

Step 2. Memorize intermediate solutions

- Store the output of functions for corresponding inputs in an array. If the function with same inputs is called again, just lookup the output from the array. This array is called, **"DP Table"**.

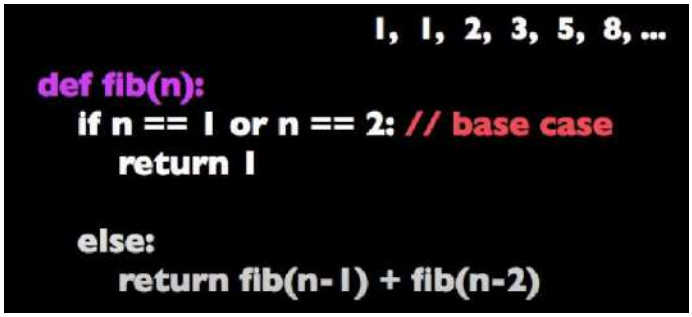
Step 3. (**Optional** for most problems)**:** Construct a **bottom-up** iterative (not recursive) approach.

- In this method, we first create an empty DP table. We write the solutions for the smallest inputs. Using those values, we iteratively calculate the solutions of all the other inputs. Thus, we find the solution for all inputs at one go.

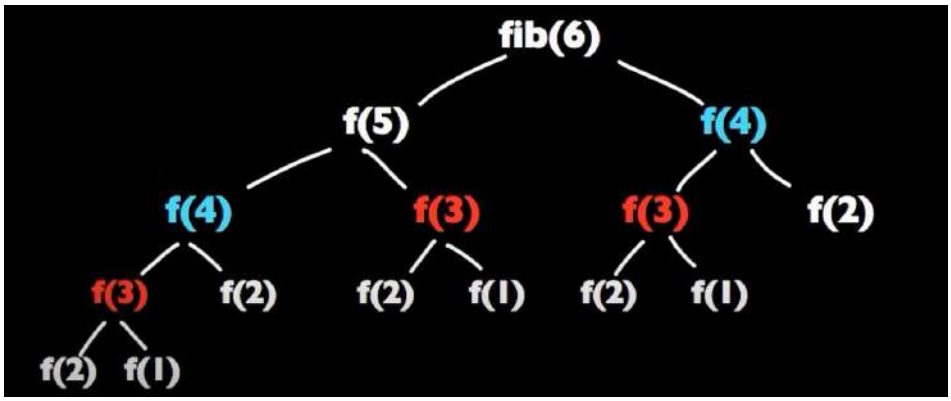
Some Example Problems

Fibonacci

Step 1: Naïve Recursive Top-Down Approach



Problem: Too many repeating function calls



Step 2: Memorize Intermediate Solutions

We will use a DP table by creating an One-Dimensional array called "seq". We will store the result of function calls here and reuse this output directly if the function is called again with the same input. The DP table is one dimensional because there is only one input to the function.

```
Initialize seq[n] with "-1"
def fib(n):
    if seq[n] != -1:
        return seq[n]
    if n==1 or n==2:
        return 1
    else:
        seq[n] = fib(n-1) + fib(n-2)
        return seq[n]
```

Step 3: Construct a Bottom-up approach

```
def fib(n): // assuming n > 2
    seq = zeros(n)
    seq[0] = seq[1] = 1
    for i from 2 to (n-1):
        seq[i] = seq[i -1] + seq[i - 2]
    return seq[n-1]
```

Knapsack

Step 1: Naïve Recursive Top-Down Approach

Weight (kg)	1	2	4	2	5
Value ($)	5	3	5	3	2

```
def KS(n, C):
    if n == 0 or C == 0: // base case
        result = 0
    else if w[n] > C:
        result = KS(n-1, C)
    else:
        tmp1 = KS(n-1, C)
        tmp2 = v[n] + KS(n-1, C - w[n])
        result = max{ tmp1, tmp2 }
    return result
```

Step 2: Memorize Intermediate Solutions

We will use a 2-Dimensional DP table where with N-rows and C-columns. The rows will be numbered from 0 to **N**, the columns will be numbered from 0 to **Capacity.**

```
// initialize arr[n][C] = undefined
def KS(n, C):
    if arr[n][C] != undefined: return arr[n][C
    if n == 0 or C == 0: // base case
        result = 0
    else if w[n] > C:
        result = KS(n-1, C)
    else:
        tmp1 = KS(n-1, C)
        tmp2 = v[n] = KS(n-1, C - w[n])
        result = max{ tmp1, tmp2 }
    arr[n][C] = result
    return result
```

Longest Common Subsequence

Problem: Find the longest two common subsequences in a string.

P = “BATD”

Q = “ABACD”

The LCS of P and Q is “BAD”

Step 1: Naïve Recursive Top-Down Approach

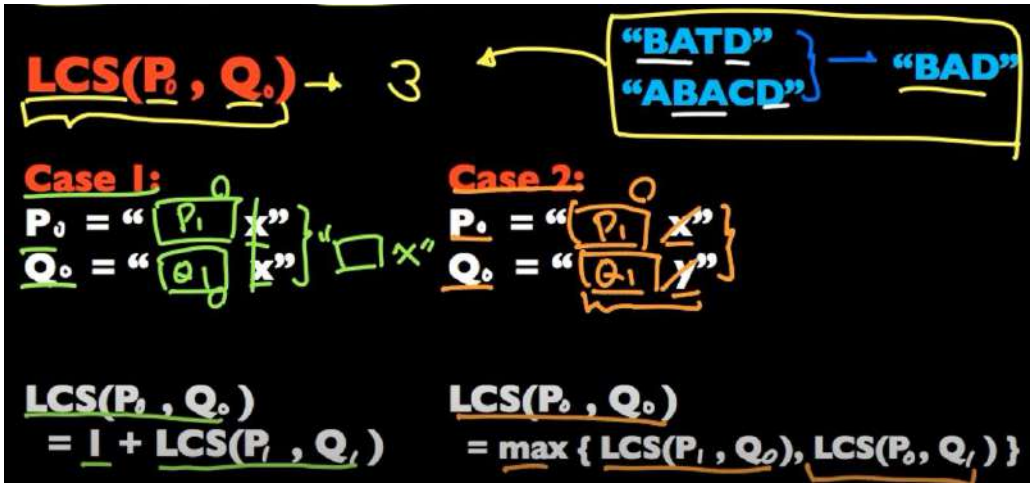
Idea: LCS of bigger strings can found out from the LCS of smaller strings created by removing one character at the end of the strings.

Case 1:

LCS(p0, q0) = 1 + LCS(p1, q1)

Case 2:

LCS(p0, q0) = max(LCS(p1, q0), LCS(p0, q1))



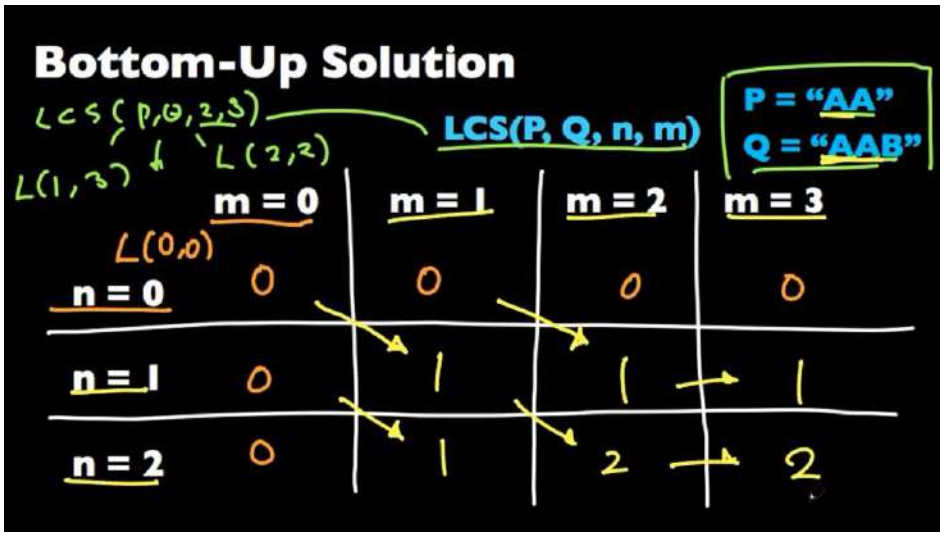

```
def LCS(P, Q, n, m)
    if n == 0 or m == 0: // base case
        result = 0
    else if P[n-1] == Q[m-1]:
        result = 1 + LCS(P, Q, n-1, m-1)
    else if P[n-1] != Q[n-1]: // just for clarity
        tmp1 = LCS(P, Q, n-1, m)
        tmp2 = LCS(P, Q, n, m-1)
        result = max{ tmp1, tmp2 }
    return result
```

Step 2: Memorize Intermediate Solutions

We will use a 2-Dimensional DP table where with N-rows and C-columns. The rows will be numbered from 0 to **N**, the columns will be numbered from 0 to **Capacity.**

```
// Initialize arr[n][m] to undefined
def LCS(P, Q, n, m)
    if arr[n][m] != undefined: return arr[n][m]
    if n == 0 or m == 0:
        result = 0
    else if P[n-1] == Q[m-1]:
        result =  1 + LCS(P, Q, n-1, m-1)
    else if P[n-1] != Q[n-1]: // just for clarity
        tmp1 = LCS(P, Q, n-1, m)
        tmp2 = LCS(P, Q, n, m-1)
        result = max{ tmp1, tmp2 }
    arr[n][m] = result
    return result
```

Step 3: Bottom-Up Solution



Resources:

https://youtu.be/Qf5R-uYQRPk

https://youtu.be/jaNZ83Q3QGc