Chapter 12: Dynamic Programming Part 2

Dr. Sirasit Lochanachit

Dynamic Programming

- Dynamic Programming solves a given complex problem by breaking it into subproblems and stores the results of subproblems so that we do not have to re-compute the same results again.
- 2 Main properties of a problem that can be solved using DP:
  a. Overlapping Subproblems
  b. Optimal Substructure

Overlapping Subproblems

- Similar to divide and conquer, DP combines solutions to sub-problems.
- DP is mainly used when solutions of same subproblems are needed.
  - Computed solutions to subproblems are stored in a table.
  - DP is not useful when there are no common subproblems (overlapping).
    - For example, binary search.

Fibonacci Number

```
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)        # Recursion = 2^n (Exponential)
```

```
                      fib(5)
                    /        \
              fib(4)            fib(3)
             /     \           /      \
        fib(3)    fib(2)    fib(2)    fib(1)
        /    \    /    \    /    \
    fib(2) fib(1) fib(1) fib(0) fib(1) fib(0)
    /   \
fib(1) fib(0)
```

Overlapping Subproblems

- There are 2 ways to store the values for reuse:
    a. Memoization (Top Down)
        - A small modification on recursive solution by adding a lookup table.
        - The algorithm looks into the lookup table before computing solutions.
    b. Tabulation (Bottom Up)
        - Builds a table starting from the first entry, then adding entries one by one as it solves more subproblems.

Memoization

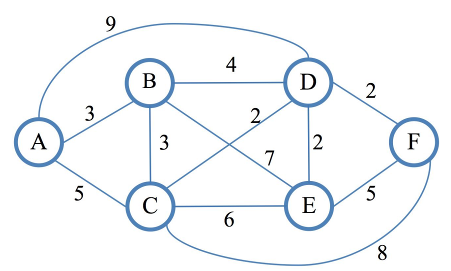

```
def fib(n, lookup):
    if n == 0 or n == 1:          # Base case
        lookup[n] = n
    if lookup[n] is None:
        lookup[n] = fib(n-1 , lookup)  + fib(n-2 , lookup)
    return lookup[n]

lookup = [None]*(6)
fib(5, lookup)
```

Tabulation


```
def fibonacci_DP(n):
    f[0] = 0
    f[1] = 1
    for i in range(2, n+1)
        f[i] = f[i-1] + f[i-2]
    return f[i]                           # n (Linear)
```

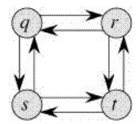
Optimal Substructure

- A given problem has optimal substructure if an optimal solution can be obtained by using optimal solutions of its subproblems.
    - For example, the shortest path has an optimal substructure property.



Optimal Substructure

- On the other hand, the Longest Path problem doesn't have the optimal substructure property.
  - Longest Path means the longest simple path (without cycle) between two nodes.



There are 2 longest paths from q to t: q→r→t and q→s→t.

For example, the longest path q→r→t is not a combination of longest path from q to r and longest path from r to t, because the longest path from q to r is q→s→t→r and the longest path from r to t is r→q→s→t.

Longest Common Subsequence (LCS)

- A common text-processing problem is to test the similarity between 2 text strings.
  - 2 strings could correspond to 2 strands of DNA, for which we want to compute similarities.
  - They could also come from 2 versions of source code for the same program, for which we want to determine changes made from one version to the next.

Longest Common Subsequence (LCS)

- Given a string $X = x_0 x_1 x_2 \ldots x_{n-1}$, a **subsequence** of X is any string that is of the form $x_{i1} x_{i2} \ldots x_{ik}$, where $i_j < i_{j+1}$
  - it is a sequence of characters that are not necessarily contiguous but are nevertheless taken in order from *X*.

For example, the string AAAG is a subsequence of the string CGATAATTGAGA.

Another example is 'abc', 'abg', 'bdf' are subsequences of 'abcdefg'.

Longest Common Subsequence (LCS)

- In the **Longest Common Subsequence (LCS)** problem, given 2 character strings $X = x_0 x_1 x_2 \ldots x_{n-1}$ and $Y = y_0 y_1 y_2 \ldots y_{m-1}$ over some alphabet, determine the longest string *S* that is a subsequence of both *X* and *Y*.
  - One way to solve this is to enumerate all subsequences of *X* and take the largest one that is also a subsequence of *Y*.
  - First, find the number of subsequences with lengths ranging from 1 to *n*-1.
    - Based on combination theory, a string of length n has 2^n -1 different possible subsequences.
      - Excluding the subsequence with length 0.

Longest Common Subsequence (LCS)

- In the **Longest Common Subsequence (LCS)** problem, given 2 character strings $X = x_0x_1x_2 \ldots x_{n-1}$ and $Y = y_0y_1y_2 \ldots y_{m-1}$ over some alphabet, determine the longest string S that is a subsequence of both X and Y.
  - One way to solve this is to enumerate all subsequences of X and take the largest one that is also a subsequence of Y.
    - Since each character of X is either in or not in a subsequence, there are potentially 2^n different subsequences of X, each of which requires $O(m)$ time to determine whether it is a subsequence of Y.
    - Thus, this brute-force approach yields an exponential-time algorithm that runs in $O(2^n m)$ time.

Longest Common Subsequence (LCS)

Examples:

LCS for ‘ABCDGH’ and ‘AEDFHR’ is ?

LCS for ‘AGGTAB’ and ‘GXTXAYB’ is ?

Optimal Substructure for LCS

Following is the recursive definition of L(X[0..n-1], Y[0..m-1]).

- If last characters of both sequences match (or X[n-1] == Y[m-1])
  - Then L(X[0..n-1], Y[0..m-1]) = 1 + L(X[0..n-2], Y[0..m-2])
- If last characters of both sequences do not match (or X[n-1] != Y[m-1])
  - Then L(X[0..n-1], Y[0..m-1]) =

    MAX(L(X[0..n-2], Y[0..m-1]), L(X[0..n-1], Y[0..m-2]))

Optimal Substructure for LCS

Consider the input strings ‘AGGTAB’ and ‘GXTXAYB’.

L(‘AGGTAB’, ‘GXTXAYB’) = 1 + L(‘AGGTA’, ‘GXTXAY’)

	A	G	G	T	A	B
G	-	-	4	-	-	-
X	-	-	-	-	-	-
T	-	-	-	3	-	-
X	-	-	-	-	-	-
A	-	-	-	-	2	-
Y	-	-	-	-	-	-
B	-	-	-	-	-	1

Optimal Substructure for LCS

Consider the input strings ‘ABCDGH’ and ‘AEDFHR’.

L(‘ABCDGH’, ‘AEDFHR’) =

MAX(L(‘ABCDG’, ‘AEDFHR’), L(‘ABCDGH’, ‘AEDFH’))

LCS Implementation

```
def lcs(X, Y, n, m):
    if n == 0 or m == 0:
        return 0;
    elif X[n-1] == Y[m-1]:
        return 1 + lcs(X, Y, n-1, m-1);
    else:
        return max(lcs(X, Y, n, m-1), lcs(X, Y, n-1, m));
```


LCS DP Implementation

```
def lcs(X , Y):
    n, m = len(X) , len(Y)
    L = [[0]* (m+1) for i in range(n+1)]          # (n+1) x (m+1) table
    """Note: L[i][j] contains length of LCS of X[0..i-1] and Y[0..j-1]"""

    for i in range(n):
        for j in range(m):
            if X[i] == Y[j]:
                L[i+1][j+1] = L[i][j]+1
            else:
                L[i+1][j+1] = max(L[i][j+1] , L[i+1][j])
    return L[n][m]                                #length of a LCS of X and Y
```