Dynamic Programming =

'programming' means using a table to store previously computed values, NOT writing code

Dyn. Prog - solves problems by combining solutions to subproblems

i.e., the problem can be broken down into multiple, overlapping subproblems

in contrast to 'Divide-and-conquer' algorithms ⇒ Also break problems into INDEPENDENT subproblems

KEY DIFF = Dyn. Prog can reuse solutions to the overlapping subproblems; divide-and-conquer subproblems don't overlap

Divide-And-Conquer:

Calculate b^x

Sol'n: $b^x = b \cdot b^{x-1}$

```
double MyPow(double b, unsigned int x)
{ if( x >=1 )
    return b*MyPow(b,x-1);
  else
    return b;
}
```

Dynamic Prog:

Calculate Fibonacci(N)

Sol'N: Fib(N) = Fib(N-1) + Fib(N-2)

These subproblems will have further, overlapping subproblems

Dyn. Prog problem characteristics -

1) Optimal substructure
2) Overlapping subproblems

TWO STYLES OF SOLUTION:

A) 'Memoization'
B) Using tables to store previously calculated results

Memoization - one strategy to avoid recalculating values -

1) break the problem down into the naive recursive solution
2) Add a 'memoization layer' that stores (parameter, result) pairs
   - When called, it'll check if parameter is known, and return the result immediately
   - Otherwise, calculate result & add to table

// Ideal for a hash table

```
class MemoizationExample
{ private HashTable previousResults = new HashTable(...);
  int FibDyn(int N)
  {
    (A) if ( previousResults.Contains(N))
          return previousResults.getData(N);

        int temp = FibDyn(N-1);
        temp += FibDyn(N-2);

    (B) previousResults.addKeyData(N, temp);
        return temp;
  }
}
```

Exercise: The 'nondynamic programming' version is almost the same as the above code, except with the two lines @ (A) and 1 line @ (B) removed

Ex1: By hand, trace out the 'nondyname' version of Fib(4). Keep track of the number of times you call Fib.

Ex2: Call FibDyn(4), as pseudocoded above. Keep track of the # of calls, made to DynFib

Ex3: Assuming that you've already called DynFib(4), show the # of calls made for DynFib(5)

Dynamic Programming

applicable to problems that exhibit:

- optimal substructure - solution can be composed of optimal subsolutions
- overlapping subproblems - some subproblems will be the same

Longest Common Subsequence -

- sequence - a collection of elements (letters, numbers) in a particular order

  $X = \langle a, b, c, d \rangle$ // sequence of letters in the alphabet

- subsequence (of a given sequence) - same as the given sequence, with some elements (possibly none) left out

  $Z = \langle a, c, d \rangle$

- common subsequence - given sequences X, Y, Z is a common subsequence if it is a subsequence of X and of Y
- Longest Common Subsequence - maximum length common subsequence of X & Y sequences

→ Optimal Substructure.

Assume $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_N \rangle$

Assume $Z = \langle z_1, z_2, \ldots, z_k \rangle$ is an LCS of X & Y

1) If $x_M = y_N$ (i.e., last element of each matches) then $z_K = x_M = y_N$ and Z_{K-1} is an LCS of X_{M-1} & Y_{N-1}

2) If $x_m \neq y_N$ and $z_K \neq x_m \xRightarrow{\text{then}} Z$ is an LCS of X_{m-1} and Y

3) If $x_m \neq y_n$ and $z_K \neq y_n \Longrightarrow Z$ is an LCS of X and Y_{n-1}

Overlapping Subproblems

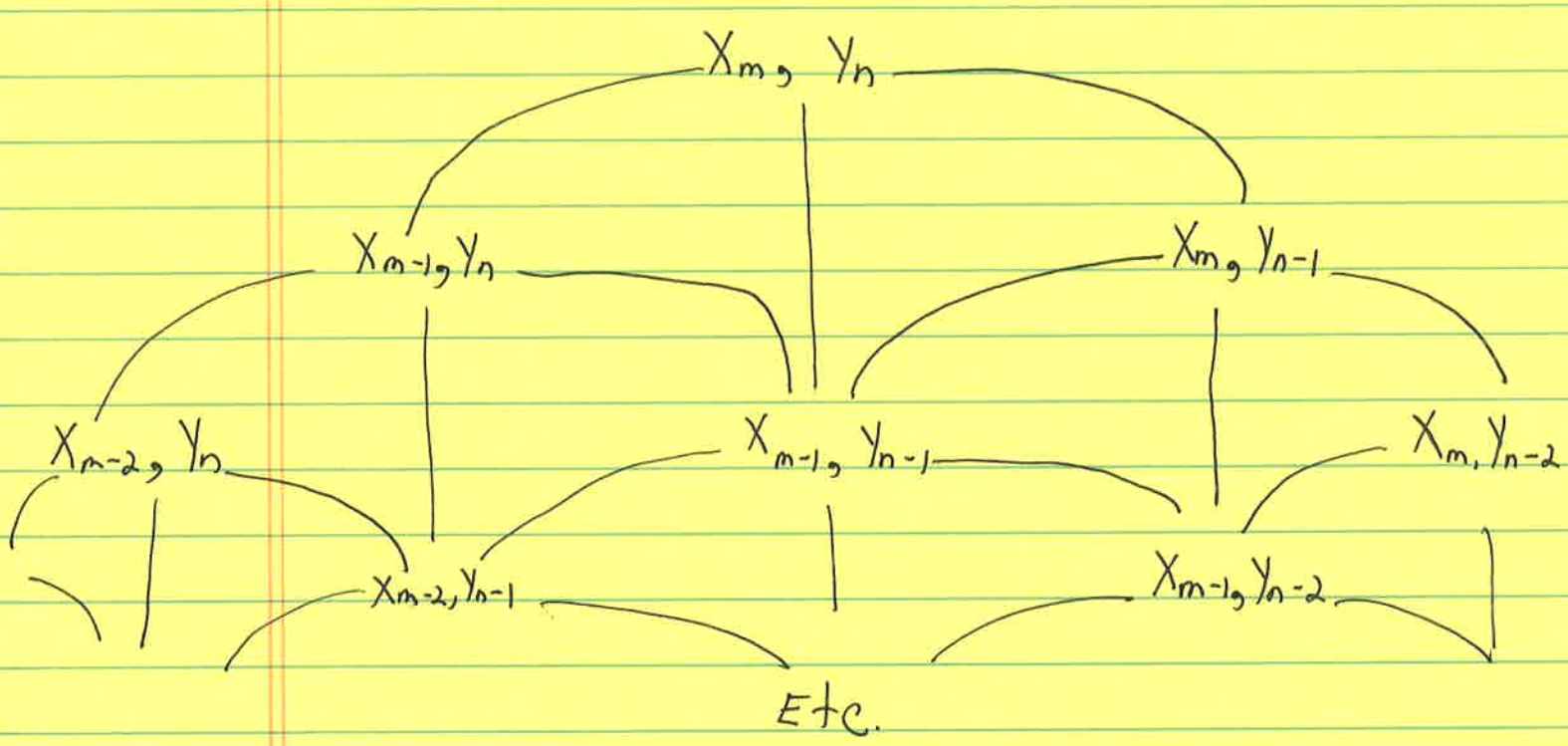
For any given X & Y, the LCS will be

X_{m-1}, Y_{n-1} (if $X_m = Y_n$)

X_{m-1}, Y (if $X_m \neq Y_n$ and it turns out that $Z_k \neq X_m$)

X_m, Y_{n-1} (if $X_m \neq Y_n$ and it turns out that $Z_k \neq Y_n$)

Visually, we can depict these subproblems as:



Let's define $c[i,j]$ to be the length of an LCS of sequences X_i and Y_j. If $i=0$ or $j=0$ then X or Y is empty and $c[i,j]=0$. Therefore-

$$c[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ c[i-1,j-1]+1 & \text{if } i>0 \,\&\&\, j>0 \,\&\&\, X_i = X_j \\ \max(c[i-1,j], c[i,j-1]) & \text{if } i>0 \,\&\&\, j>0 \,\&\&\, X_i \neq X_j \end{cases}$$

The naive recursive solution will work, but will re-solve many problems, as listed above

Match ACTCG vs. ACAGTAG

- no 'gap penalty'
- no 'mismatch penalty'

2.1 /4

1. Line up sequences along top, left side of 2D array

		0	A 1	C 2	T 3	C 4	G 5
	0	0	0	0	0	0	0
A	1	0					
C	2	0					
A	3	0					
G	4	0					
T	5	0					
A	6	0					
G	7	0					

moving down: leave a gap in Left-edge sequence

moving across: leave a gap in top-edge sequence

2. Fill in 1st row, leftmost column w/ 0's

3. Start @ ([1,1]), and iterate through. (row, col)

foreach box, take the max of:

results[row-1, col] // means leave a gap in Left Hand sequence

results[row, col-1] // means leave a gap in top sequence

results[row-1, col-1] + (X[row]==Y[col])?1:0

// means we've chosen to match X[row] against Y[col]

		A	C	T
	0	0	0	0
A	0	1	1	1
C	0	1	2	2
A	0	1	2	2

best match between AC & AC

stores best-known match between AC & ACT (leave off T, then match C's, then A's

best match between ACA & AC

Match ACTCG against ACAGTAG - no 'gap penalty'
no 'mismatch penalty'

2.2/4

Y →		0	A 1	C 2	T 3	G 4	G 5
X ↓	0	0	0	0	0	0	0
A	1	0	1	1	1	1	1
C	2	0	1	2	2	2	2
A	3	0	1	2	2	2	2
G	4	0	1	2			
T	5	0					
A	6	0					
G	7	0					

1. Line up sequences along top, left side of the 2D array

2. Fill in 0's down the left side, across the top row

---

3. Start @ (1,1).
   A. if (LeftEdgeSequence[row] == TopEdgeSequence[col])
      then We'll take the value of the match that's up & left one, add 1 to it (to reflect this match) &

Dynamic Solution.

Create one 2D table to store LCS-length <u>values</u>

Create a second 2D table to store LCS construction notes

(either "↖", "←", or "↑" – points to optimal subsolution)

```
LCS-Length ( X, Y)
    m = X.length();
    n = Y.length();
    for (int i = 0; i <= m ; i++)  c[i, 0]=0;
    for (int j = 0; j <= n ; j++)  c[0,j] = 0;
    for( i=1; i <= m ; i++)
      for(  j=1; j <= n ; j++)
      {  if( X[i] == Y[j])
         {
            c[i,j] =c[i-1, j-1] +1;
            b[i,j] = "↖";
         }
         else if( c[i-1,j] >= c[i,j-1])
         {
            c[i,j] = c[i-1, j];
            b[i,j] = "↑";
         }
         else // c[i,j-1] > c[i-1,j]
         {
            c[i,j] = c[i, j-1];
            b[i,j] = "←";
         }
      }
```

int[,] c = new int[X.length()+1, Y.length()+1];
int[,] b = new int[X.length()+1, Y.length()+1];

```
PrintLCS ( b, X, i, j )
    if ( i == 0 || j == 0 )
        return;
    if ( b[i, j] == "↖" )
    {   Print-LCS ( b, X, i-1, j-1 );
        print X[i];
    }
    else if ( b[i,j] == "↑" )
    {   Print-LCS (b, X, i-1, j);   }
    else // b[i,j] == "←"
        Print-LCS ( b, X, i, j-1 );
```