Lecture 11b: Dynamic Programming (cont.)

Spontaneous Right Action

Wholeness

A dynamic programming algorithm divides a problem into subproblems, then solves each subproblem just once and saves the solution in a table to avoid having to repeat that calculation. Memoization is a technique for implementing dynamic programming to make a recursive algorithm efficient. Pure intelligence governs the activities of the universe in accord with the law of least action.

Memoization

- The basic idea
  - Design the natural recursive algorithm
  - If recursive calls with the same arguments are repeatedly made, then memoize the inefficient recursive algorithm
    - Save these subproblem solutions in a table so they do not have to be recomputed
- Implementation
  - A table is maintained with subproblem solutions (as before), but the control structure for filling in the table occurs during normal execution of the recursive algorithm
- Advantages
  - The algorithm does not have to be transformed into an iterative one
  - Often offers the same (or better) efficiency as the usual dynamic-programming approach

Example: Calculate Fibonacci Numbers

Mathematical definition:

fib(0) = 0

fib(1) = 1

fib(n) = fib(n-2) + fib(n-1)   if n > 1

Fibonacci solution1

```
Algorithm Fib( n ):
  Input: integer n ≥ 0
  Output: the n-th Fibonacci number
   if n=0 then
     return 0
   else if n=1 then
     return 1
   else
     return Fib(n - 2) +  Fib( n - 1)
```


Fibonacci Solution 2

```
Algorithm Fib(n):
    Input: integer n ≥ 0
    Output: the n-th Fibonacci number

     F ← new array of size n+1
     for i ← 0 to n do
          F[i] ← -1

     return memoizedFib(n, F)
```

```
Algorithm memoizedFib(n, F):
    Input: integer n ≥ 0
    Output: the n-th Fibonacci number
     if F[n] < 0 then  // If Fib(n) has not been computed?
          if n=0 then
               F[n] ← 0
          else if n=1 then
               F[n] ← 1
          else
               F[n] ← memoizedFib(n-2) + memoizedFib(n-1)

     return F[n]
```

Summary:
Memoized Recursive Algorithms

- A memoized recursive algorithm maintains a table with an entry for the solution to each subproblem (same as before)
- Each table entry initially contains a special value to indicate that the entry has yet to be filled in
- When the subproblem is first encountered, its solution is computed and stored in the table
- Subsequently, the value is looked up rather than computed

Exercises

1. Memoize the algorithm to compute Fibonacci numbers using two integer parameters instead of table F
2. Memoize the algorithm to compute Fibonacci numbers using one integer parameter

Main Point

1. Memoization is a technique for doing dynamic programming recursively. It often has the same benefits as regular dynamic programming without requiring major changes to the original more natural recursive algorithm.
   Science of Consciousness: The TM program provides natural, effortless techniques for removing stress and bringing out spontaneous right action.

Developing a Dynamic Programming Algorithm

1. Characterize the structure of a solution
2. Tackle the problem "top-down" as if creating a recursive algorithm
   - Figure out how to solve the larger problem by finding and using solutions to smaller problems
3. Find computations that have to be done repeatedly
   - Define an appropriate table for saving results of smaller problems
   - Write a formula for computing the table entries
4. Determine how to compute the solution from the data in the table
   - Determine the order in which the table entries have to be computed and used (usually bottom up)
5. Construct an optimal solution from the computed information gathered during execution of step 4

Longest Common Subsequence (§9.4)

Dynamic Programming Example

Step 1:
Longest Common Subsequence

- Given two strings, find a longest subsequence that they share in common
- Substring vs. Subsequence
  - Substring: the characters in a substring of S must occur *contiguously* in S
  - Subsequence: the characters can be interspersed with *gaps*
- *Consider string cabd*
  - *How many subsequences does it have?*

- What is the longest common subsequence of the following two strings?
  - cababc
  - abdcb

Step 1: (characterize structure of solution)

- *Consider cababc* and *abdcb, what is the longest common subsequence*

alignment 1

`cababc.`
`.abd.cb`

the longest common subsequence is .ab..c with length 3

alignment 2

`caba.bc`
`.abdcb.`

the longest common subsequence is .ab..b with length 3

Step 1: (characterize structure of solution)

Let's give a score M to an alignment in this way,

M=sum $s(x_i,y_i)$, where x_i is the i[th] character in the first aligned sequence
y_i is the i[th] character in the second aligned sequence

$s(x_i,y_i)$= 1 if $x_i= y_i$

$s(x_i,y_i)$= 0 if $x_i \neq y_i$ or *any of them is a gap*

The score for alignment:

`cababc.`
`.abd.cb`

M= s(c,.)+s(a,a)+s(b,b)+s(a,d)+s(b,.)+s(c,c)+s(.,b)=3

To find the longest common subsequence between sequences S_1 and S_2 is to find the alignment that maximizes score M.

A brute force algorithm takes $O(2^n m)$. Why?

Step 2: (create recursive algorithm)

- Subproblem optimality

Consider two sequences

S_1: $a_1a_2a_3 \ldots a_i$
S_2: $b_1b_2b_3 \ldots b_j$

Let the optimal alignment be

$x_1x_2x_3 \ldots x_{n-1}x_n$
$y_1y_2y_3 \ldots y_{n-1}y_n$

For the last pair (x_n,y_n),

there are three possible cases :

Substitution $\boxed{a_1 \ldots}a_i$ / $\boxed{b_1 \ldots}b_j$

Gap $\boxed{a_1 \ldots a_i}$ - / $\boxed{b_1 \ldots\ldots}b_j$

Gap $\boxed{a_1 \ldots\ldots}a_i$ / $\boxed{b_1 \ldots b_j}$ -

Step 2: (create recursive algorithm)

There are three cases for (x_n,y_n) pair:

S_1: $a_1a_2a_3 \ldots a_i$
S_2: $b_1b_2b_3 \ldots b_j$

$x_1x_2x_3 \ldots x_{n-1}x_n$
$y_1y_2y_3 \ldots y_{n-1}y_n$

Substitution $\boxed{a_1 \ldots}a_i$ / $\boxed{b_1 \ldots}b_j$ — $L_{i,j} = L_{i-1, j-1} + S_{i,j}$ (match/mismatch)

Gap $\boxed{a_1 \ldots a_i}$ - / $\boxed{b_1 \ldots\ldots}b_j$ — $L_{i,j} = L_{i, j-1}$ (gap in S_1)

Gap $\boxed{a_1 \ldots\ldots}a_i$ / $\boxed{b_1 \ldots b_j}$ - — $L_{i,j} = L_{i-1, j}$ (gap in S_2)

$L_{i,j}$ = MAX {$L_{i-1, j-1}$ + S (a_i,b_j) (match/mismatch)
$L_{i, j-1}$ + 0 (gap in sequence #1)
$L_{i-1, j}$ + 0 (gap in sequence #2) }

$L_{i,j}$ is the score for optimal alignment between strings *a[1…i]* (substring of *a* from index *1* to *i*) and *b[1…j]*

Step 3: (locate subproblem overlap)

$L_{i,j}$ = MAX {
$L_{i-1, j-1}$ + $S(a_i,b_j)$,
$L_{i, j-1}$ + 0,
$L_{i-1, j}$ + 0
}

$S(a_i,b_j)$= 1 if a_i=b_j

$S(a_i,b_j)$= 0 if $a_i \neq b_j$ or *either of them is a gap*

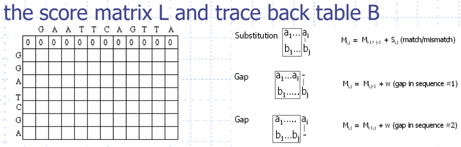
Examples:

G A A T T C A G T T A (sequence #1)

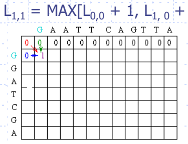
G G A T C G A (sequence #2)

Step 4:
(define table for storing results)

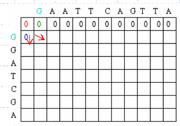
Fill the score matrix L and trace back table B



$L_{1,1} = MAX[L_{0,0} + 1, L_{1,0} + 0, L_{0,1} + 0] = MAX [1, 0, 0] = 1$



Score matrix L



Trace back table B

Longest Common Subsequence

We need to use trace back table to find out the best alignment, which has a score of 6

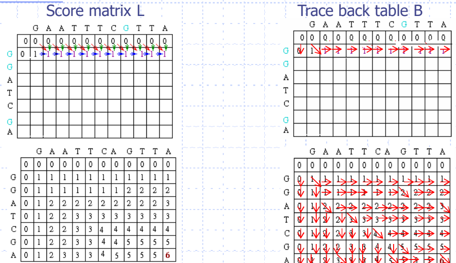
(1) Find the path from lower right corner to upper left corner

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

Longest Common Subsequence

Score matrix L

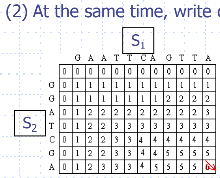
Trace back table B



$L_{7,11}$=6 (lower right corner of Score matrix)
This tells us that the best alignment has a score of 6
What is the best alignment?

Longest Common Subsequence

(2) At the same time, write down the alignment backward

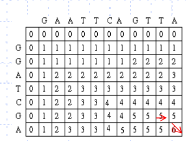



```
(Seq #1)   A
           |
(Seq #2)   A
```

↘:Take one character from each sequence

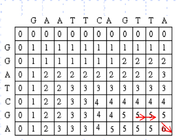
→:Take one character from sequence S_1 (columns)

↓:Take one character from sequence S_2 (rows)



```
(Seq #1)   T A
             |
(Seq #2)   _ A
```


Longest Common Subsequence

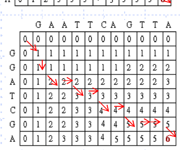


```
(Seq #1)   T T A
               |
(Seq #2)   _ _ A
```

↘:Take one character from each sequence

→:Take one character from sequence S_1 (columns)

↓:Take one character from sequence S_2 (rows)



```
(Seq #1)  G _ A A T T C A G T T A
          |   |   |   |   |     |
(Seq #2)  G G A _ T _ C _ G _ _ A
```


Longest Common Subsequence

Thus, an optimal alignment is

```
(Seq #1)  G _ A A T T C A G T T A
          |   |   |   |   |     |
(Seq #2)  G G A _ T _ C _ G _ _ A
```

The longest common subsequence is
G.A.T.C.G..A

There might be multiple longest common subsequences (LCSs) between two given sequences.

These LCSs have the same number of characters (not including gaps)

Recursive (brute force) Longest Common Subsequence

// $L_{i,j} = \text{MAX}\{ L_{i-1, j-1} + S(a_i,b_j),\ L_{i, j-1} + 0,\ L_{i-1, j} + 0 \}$

```
Algorithm LCS(S1, S2, m, n):
   Input: Strings S1 and S2 with at least m and n elements, respectively
   Output: Length of the LCS of S1[1..m] and S2[1..n]
     if n = 0  then
         return 0
     else if m = 0  then
         return 0
     else if S1[ m ] = S2[ n ]  then
         return LCS(S1, S2, m -1, n -1) + 1
     else
         return max { LCS(S1, S2, m, n -1), LCS(S1, S2, m -1, n) }
```


Iterative (Efficient) Version of Longest Common Subsequence

```
Algorithm LCS(X, Y):
  Input: Strings X and Y with m and n elements, respectively
  Output: L is an (m + 1)x(n +1) array such that L[i, j] contains the
    length of the LCS of X[1..i] and Y[1..j]
      m ← X.length
      n ← Y.length
      for i ← 0 to m do
         L[i, 0] ← 0
      for j ← 0 to n do
         L[0, j] ← 0
      for i ← 1 to m do
         for j ← 1 to n do
            if X[ i ] = Y[ j ]  then
               L[i, j] ← L[i -1, j -1] + 1
            else
               L[i, j] ← max { L[i -1, j], L[i, j -1] }
      return L
```


Exercise: Memoize Recursive LCS

```
Algorithm LCS(S1, S2, m, n):
   Input: Strings S1 and S2 with at least m and n elements, respectively
   Output: Length of the LCS of S1[1..m] and S2[1..n]
     if n = 0  then
         return 0
     else if m = 0  then
         return 0
     else if S1[ m ] = S2[ n ]  then
         return LCS(S1, S2, m -1, n -1) + 1
     else
         return max { LCS(S1, S2, m, n -1), LCS(S1, S2, m -1, n) }
```


Top Level of Recursive Longest Common Subsequence

```
Algorithm LCS(X, Y):
  Input: Strings X and Y with m and n elements, respectively
  Output: LCS of X and Y

      L ← new array with (m+1)x(n+1) elements
      m ← X.length
      n ← Y.length
      for i ← 0 to m do
        for j ← 0 to n do
           L[i, j] ← -1

      return LCS(X, Y, m, n)
```


Recursive (memoized) Longest Common Subsequence

```
Algorithm LCS(S1, S2, m, n):
   Input: Strings S1 and S2 with at least m and n elements, respectively
   Output: Length of the LCS of S1[1..m] and S2[1..n]
     if L[m, n] < 0  then  {not already computed}
        if n = 0  then
           L[m, 0] ← 0
        else if m = 0  then
           L[0, n] ← 0
        else if S1[ m ] = S2[ n ]  then
           L[m, n] ← LCS(S1, S2, m -1, n -1) + 1
        else
           L[m, n] ← max { LCS(S1, S2, m, n -1), LCS(S1, S2, m -1, n) }

     return L[m, n]
```


Step 5: Print the LCS

- How would we add the back trace matrix to the previous algorithm so we can print the longest common sequence?
  - Let b mean that we take one character from both strings
  - Let x mean that we take one character from string X
  - Let y mean that we take one character from string Y

LCS with Trace Back

```
Algorithm LCS(X, Y):
    Input: Strings X and Y with m and n elements, respectively
    Output: L is an (m + 1)x(n +1) array, L[i, j] contains length of the LCS of X[1..i]
    and Y[1..j]
     m ← X.length
     n ← Y.length
     for i ← 0 to m do
         L[i, 0] ← 0
     for j ← 0 to n do
         L[0, j] ← 0
     for i ← 1 to m do
         for j ← 1 to n do
           if X[ i ] = Y[ j ]  then
              L[i, j] ← L[i -1, j -1] + 1
              B[i, j] ← b
           else  if L[i -1, j] < L[i, j -1]  then
              L[i, j] ← L[i, j -1]
              B[i, j] ← y
           else
              L[i, j] ← L[i -1, j]
              B[i, j] ← x
     return L and B
```


Dynamic Programming The General Technique

- Simple subproblems:
  - Must be some way of breaking the global problem into subproblems, each having similar structure to the original
  - Need a simple way of keeping track of subproblems with just a few indices, like i, j, k, etc.
- Subproblem optimality:
  - Optimal solutions cannot contain suboptimal subproblem solutions
  - Should have a relatively simple combining operation
- Subproblem overlap:
  - This is where the computing time is reduced

Main Point

2. A dynamic programming algorithm divides a problem into subproblems, then solves each subproblem just once and saves the solution in a table to avoid having to repeat that calculation. Dynamic programming is typically applied to optimization problems to reduce the time required from exponential to polynomial time.
   Science of Consciousness: Pure intelligence governs the activities of the universe in accord with the law of least action.

Connecting the Parts of Knowledge with the Wholeness of Knowledge

1. A common text processing problem in genetics and software engineering is to test the similarity between two text strings. One could enumerate all subsequences of one string and select the longest one that is also a subsequence of the other which takes exponential time.

2. Through dynamic programming we can transform an infeasible (exponential) LCS algorithm into one that can be done efficiently.

3. **Transcendental Consciousness** is the unbounded home of all the laws of nature.
4. **Impulses within the transcendental field**: These dynamic natural laws within this unbounded field govern all the activities of the universe with perfect efficiency.
5. **Wholeness moving within itself**: In Unity Consciousness, one experiences the laws of nature as waves of one's own unbounded pure consciousness.