Longest Common Subsequence

Example: `X = abadcda, Y = acbacadb`

`bcd` is a common subsequence of `X,Y`

`X = abadcda, Y = acbacadb`

`abacd` is an LCS of `X,Y` (though it is not the unique LCS)

`X = abadcda, Y = acbacadb`

- Length of LCS is a measure of the similarity of two strings
- The Unix `diff` command is based on LCS
- Similar to the edit distance of two strings

Problem: Given 2 strings X and Y, of lengths m and n, find:
- the length of an LCS of X and Y
- an actual LCS

Solution: use Dynamic Programming (DP)

Iterative DP

Let $f_{i,j}$ be the length of an LCS of the i^{th} prefix $X_i = X(1..i)$ of X and the j^{th} prefix $Y_j = Y(1..j)$ of Y. Then

$$f_{i,j} = \begin{cases} 1 + f_{i-1,j-1} & \text{if } X(i) = Y(j) \\ \max(f_{i,j-1}, f_{i-1,j}) & \text{otherwise} \end{cases}$$

with

$$f_{i,0} = f_{0,j} = 0 \quad \text{for all } i, j$$

Proof: Let Z be an LCS of X_i and Y_j, $|Z|=k$.

Case (i): $X(i)=Y(j)$. Then $Z(k)=X(i)=Y(j)$ and Z_{k-1} is an LCS of X_{i-1} and Y_{j-1}.

Case (ii): $X(i)\neq Y(j)$ and $Z(k)\neq X(i)$. Then Z is an LCS of X_{i-1} and Y_j.

Case (iii): $X(i)\neq Y(j)$ and $Z(k)\neq Y(j)$. Then Z is an LCS of X_i and Y_{j-1}.

Algorithm for Iterative DP
(simple version – just to find LCS length)

```
/** Returns the length of the LCS of strings x and y
  * Assume chars of a string of length r indexed 1..r */

public int lcs(String x, String y) {
   int m = x.length();
   int n = y.length();
   int [][] l = 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)
            l[i][j]=0;
         else if (x.charAt(i) == y.charAt(j))
            l[i][j] = l[i-1][j-1]+1;
         else
            l[i][j] = Math.max(l[i-1][j], l[i][j-1]);
   return l[m][n];
}
```

Dynamic programming table – example

Y			0	1	2	3	4	5	6	7	8
				a	c	b	a	c	a	d	b
	0		0	0	0	0	0	0	0	0	0
	1	a	0	1	1	1	1	1	1	1	1
	2	b	0	1	1	2	2	2	2	2	2
	3	a	0	1	1	2	3	3	3	3	3
X	4	d	0	1	1	2	3	3	3	4	4
	5	c	0	1	2	2	3	4	4	4	4
	6	d	0	1	2	2	3	4	4	5	5
	7	a	0	1	2	2	3	4	5	5	5

- To construct an actual LCS
  - trace a path from bottom right to top left
  - draw an arrow from an entry to the entry that led to its value
  - interpret diagonal steps as members of an LCS
  - solution is not necessarily unique

Complexity

- Each table entry is evaluated in O(1) time
- So overall, the algorithm uses O(mn) time and space
- Can easily reduce to O(n) space if only LCS length is required
- There is a subtle divide-and-conquer variant (Hirschberg's algorithm) that allows an actual LCS to be found in O(mn) time and O(n) space

Lazy LCS evaluation

- In the DP table, typically only a subset of the entries is needed – example later
- An alternative *lazy* approach evaluates only the entries that are needed, and therefore is potentially more efficient
- The lazy evaluation can be facilitated by use of *recursion*

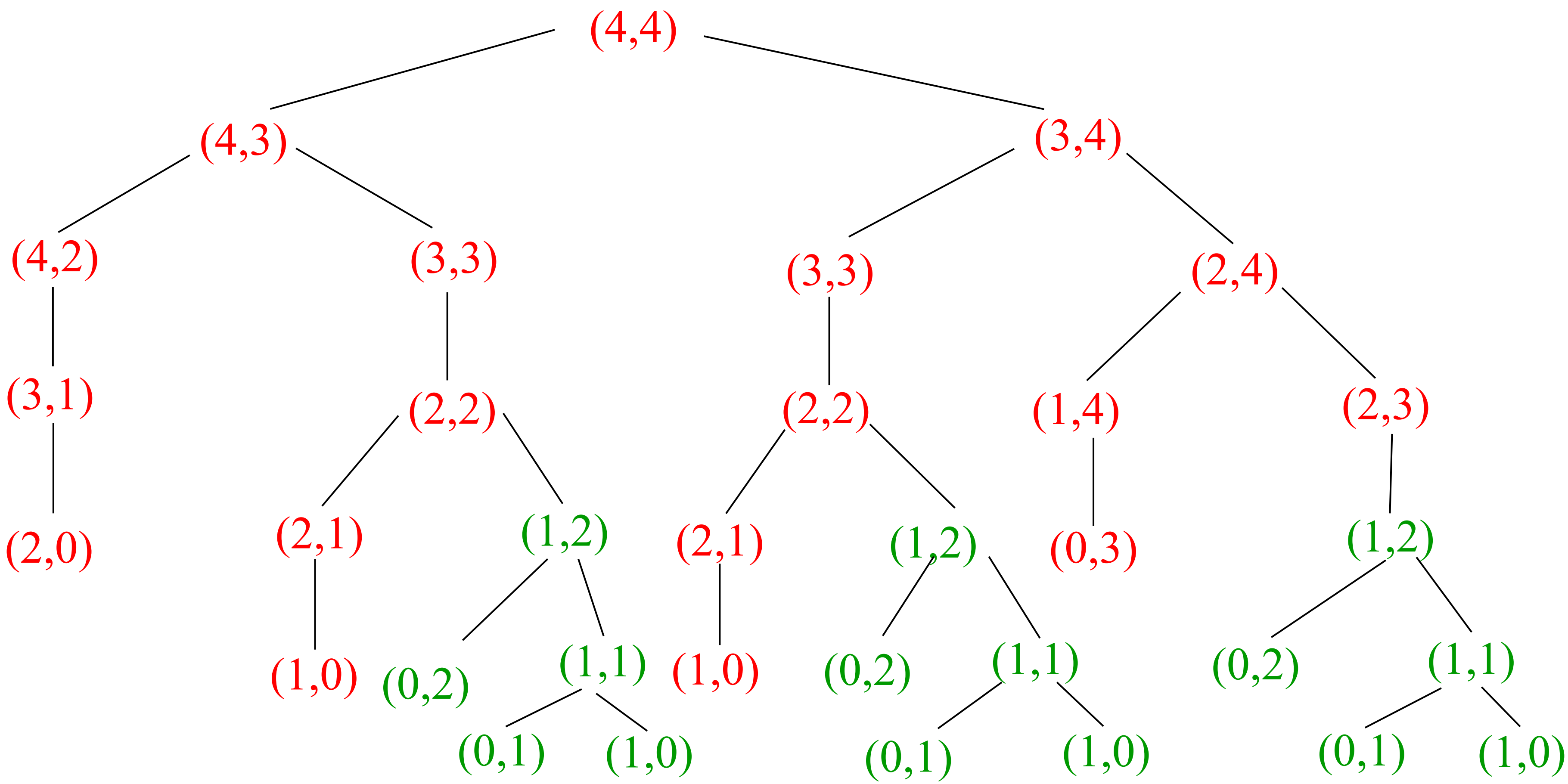
Recursive DP – first attempt

```
/** Return LCS length of xi and yj*/
private int rLCS(int i, int j) {
  if ( i==0 || j==0 )
    return 0;
  else if (x.charAt(i) == y.charAt(j))
    return 1 + rLCS(i-1,j-1);
  else
    return Math.max(rLCS(i-1,j), rLCS(i,j-1))
}
```

- Call function externally as `rLCS(m,n)`
- Good news: for some i and j, `rLCS(i,j)` is never computed
- Bad news: for other i and j, `rLCS(i,j)` is computed several times!
- Examples of both of these on the next slide

Example: `X=caab` `Y=abac`

Tree of recursive calls:



- **None of `rLCS(1,3)`, `rLCS(3,2)`, `rLCS(4,1)` is ever computed**
- **`rLCS(3,3)` is computed twice**
- **`rLCS(1,2)` is computed three times, etc.**

Avoiding repeated computation

- **The technique of *memoisation***
  - **maintains the 2-dimensional array as a global variable**
  - **looks up the array before making a recursive call**
  - ***makes the call only if the element has not previously been evaluated***
  - **when first determined, a value is entered into the table**
- **How can we decide if an element has been previously evaluated?**
  - **easy: just initialise the table, setting every entry to, say -1**
  - ***but*, this defeats the purpose, since it involves visiting every cell in the table**

Recursive DP with memoisation

```
/* l is a global variable; assume that l's
 * elements are all initialised with value -1 */
int [][] l = new int[x.length()+1][y.length()+1];

/* Lazy DP for LCS with memoisation
 * Returns LCS length of xi and yj */
private int mLCS(int i, int j) {

   if (l[i][j]==-1)       // required value not already computed
   { if (i==0 || j==0)  // trivial case
         l[i][j]=0;
     else if (x.charAt(i)==y.charAt(j)) // case (i)
         l[i][j] = 1 + mLCS(i-1,j-1);
     else                                // case (ii)
         l[i][j] = Math.max(mLCS(i-1,j), mLCS(i,j-1));
   }
   return l[i][j];
}
```

Example of lazy evaluation (an entry -1 indicates non-evaluation)

First string: dbacacbabcda **Second string:** dbcbdaadcabd

		d	b	c	b	d	a	a	d	c	a	b	d
	0	-1	0	0	0	-1	0	0	-1	-1	-1	-1	-1
d	-1	1	1	1	-1	1	1	1	1	-1	-1	-1	-1
b	0	1	2	2	2	2	2	2	2	-1	-1	-1	-1
a	0	1	2	2	2	2	3	3	3	-1	-1	-1	-1
c	0	1	2	3	3	3	3	3	3	-1	-1	-1	-1
a	0	1	2	3	3	3	4	4	4	-1	-1	-1	-1
c	0	1	-1	3	3	3	4	4	4	5	-1	-1	-1
b	0	1	2	3	4	4	4	4	4	5	-1	-1	-1
a	0	1	2	3	4	4	5	5	5	5	6	-1	-1
b	-1	-1	2	-1	4	4	5	5	5	5	6	7	-1
c	-1	-1	-1	3	4	4	5	5	-1	6	6	7	-1
d	-1	-1	-1	-1	-1	-1	-1	-1	6	6	6	7	8
a	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	7	7	8

- **But:** every cell has to be initialised (which defeats the purpose)
  - *or does it?*

Avoiding initialisation using “virtual initialisation”

How to insert values into an array at unpredictable positions

- **We want to determine, in O(1) time, for a given position, whether a value has been inserted**
- **We want to avoid initialising the whole array**
- ***Intuition suggests it can't be done***
  - **how to distinguish a genuine inserted value from a garbage value?**

The solution (for a 1-dimensional array, 2-dimensional version is similar)

- **Let the array, `a`, be an array of pairs `(v,p)`**
  - **`v` holds the actual value inserted**
  - **`p` is a pointer into a second companion integer array `b`**

The solution (cont.)

- Keep a count `n` of the number of values inserted
- On inserting next value, say in position `i`
  - increment `n`
  - set `a(i).v` to the required value
  - set `a(i).p = n` and set `b(n) = i`
- Suppose we want to know, at any point, whether `a[k].v` is a genuine value – use the following algorithm:

```
int j = a[k].p;
if ( j < 1 || j > n )
  /* no value can have been inserted in position k */
  return false;
else
  /* if a[k].v is genuine then b[j] must have
   * been set to k, and if not b[j] must have
   * been set to a different value */
  return ( b[j] == k );
```